Digital signal conditioning on multiprocessor systems

Lee Gould

Digital Signal Conditioning on Multiprocessor Systems

Lee Gould BSc. MSc.

Submitted in Partial Fulfilment of the Degree of Doctor of Philosophy

University of Durham

1992

Declaration

I declare that the work reported in this thesis, unless otherwise stated, was carried out by the candidate, that it has not previously been submitted for any degree and that it is not currently being submitted for any other degree.

Statement of Copyright

Table of Contents

Acknowledgements

I would like to thank my supervisor, Dr. Alan Purvis, and my second supervisor, Prof. P. Mars, for allowing this project to be initiated and for arranging the initial SERC funding. I would also like to thank Dr. Purvis for his efforts in helping to secure additional funding from British Gas plc.

I am also grateful to my industrial supervisor, Dr. J. P. Allen of British Gas ERS, Killingworth, for his efforts in securing my funding, and for his guidance during my funding period.

This work has been funded by SERC, British Gas plc and the University of Durham, to whom I am very grateful.

The work presented in this thesis would not have been possible without the jovial support and long enduring patience of the electronics and microprocessor centre technical staff. I am especially grateful to Mr. Ian Hutchinson for those times when vital test equipment needed to be found.

I am most grateful to my friends and colleagues, Ken Linton and Norman Powell, for our many helpful discussions and their moral support throughout the period of this project. I am especially indebted to Norman for the time he spent proof reading some of these chapters, and for his help in producing the frequency response plots.

Abstract

An important application area of modern computer systems is that of digital signal processing. This discipline is concerned with the analysis or modification of digitally represented signals, through the use of simple mathematical operations. A primary need of such systems is that of high data throughput. Although optimised programmable processors are available, system designers are now looking towards parallel processing to gain further performance increases.

Such parallel systems may be easily constructed using the transputer family of processors. However, although these devices are comparatively easy to program, they possess a general von Neumann core and so are relatively inefficient at implementing digital signal processing algorithms. The power of the transputer lies in its ability to communicate effectively, not in its computational capability.

The converse is true of specialised digital signal processors. These devices have been designed specifically to implement the type of small data intensive operations required by digital signal processing algorithms, but have not been designed to operate efficiently in a multiprocessor environment.

This thesis examines the performance of both types of processors with reference to a common signal processing application, multichannel filtering. The transputer is examined in both uniprocessor and multiprocessor configurations, and its performance analysed. A theoretical model of program behaviour is developed, in order to assess the performance benefits of particular code structures and the effects of such parameters as data block size. The transputer implementation is contrasted with that of the Motorola DSP56001 digital signal processor. This device is found to be much more efficient at implementing such algorithms on a single device, but provides limited multiprocessor support.

Using the conclusions of this assessment, a hybrid multiprocessor has been designed. This consists of a transputer controlling a number of signal processors, communicating through shared memory, separating the tasks of computation and communication. Forcing the transputer to communicate through shared memory causes problems, and these have been addressed. A theoretical performance model of the system has been produced. A small system has been constructed, and is currently running performance test software.

Chapter 1

Introduction

From the inception of the first microprocessor based systems in the early 1970s, their range of application has steadily increased. This has been aided by the ongoing development of integrated circuit fabrication technology, which has resulted in the production of relatively inexpensive, powerful processors, and by continuing software development, which has produced compilers, operating systems and development tools used to ease the programming task.

One particular area which has benefitted significantly from these developments is that of digital signal processing (DSP), which is concerned with the modification or synthesis of signals represented in the digital domain. Although some DSP operations mimic their analogue counterparts, many may be realised only in the digital domain. This versatility, the ease by which the characteristics of a digital processing system may be altered and the simplicity with which many of the basic building block operations may be implemented has led to the widespread popularity of such systems.

Although once only a spin-off from general purpose microprocessor technology, digital signal processing systems are now very sophisticated, and may be said to constitute a major branch of modern computing systems. The range of applications is wide, recent developments having made a significant impact in the area

of consumer audio products with the advent of compact disc and digital audio tape (DAT) systems. Other application areas include sound synthesis, medical imaging, seismic signal processing, speech recognition, graphics rendering and image processing. The continuing alliance of digital signal processing with the area of parallel computing promises the development of systems orders of magnitude more powerful than those of today.

This chapter continues by giving a brief overview of microprocessors and parallel computing. Digital signal processing is introduced, and an appraisal of modern digital signal processing devices is presented, with a discussion of the application of parallel computing techniques to digital signal processing. The chapter concludes by describing the subject matter of this thesis.

1.1 A Brief History of Microprocessors

Modern microprocessors are the products of continually advancing semiconductor technology, which began with the invention of the transistor in 1947. These advances have allowed the dimensions of devices to decrease, increasing both the amount of circuitry per unit silicon area and the operational speed.

The first microprocessor, the Intel 4004, was launched in 1971. This was a slow 4bit device, with a limited addressing capability. This device was followed up by the 8bit 8008 in 1972. As part of its efforts to convince engineers to use their microprocessors, Intel also developed a range of programming tools.

By 1976, when Intel launched the 5V supply 8085, a number of 8bit microprocessors were available, including the Zilog Z80 and the Motorola 6800. These devices were produced in large quantities, reducing their cost which made them more

attractive for use in consumer products. Some of these microprocessors were made available with on-chip memory and termed "microcomputers".

Although the first 16bit microprocessor was introduced in 1977, it was not until the launch of the Intel 8086 in 1978 that any significant performance increase was attained. These processors offer more advanced architectures than their 8bit counterparts, many incorporating internal 32bit architectures, various memory modes, large address spaces and high clock speeds.

1983 saw the introduction of the first truly 32bit microprocessor, the National Semiconductors NS32032. Other 32 bit devices include the Intel 80386, the Motorola 68020/30 and the Inmos transputer [1], [2]. As a result of decreased feature size and an increase in die size, modern processors are capable of operating at higher clock speeds (50MHz) and incorporate many features such as memory, cache, peripherals and specialised execution units (ie floating point units) on-chip. Included in this new generation are the Intel 80486 and the Motorola 68040, both of which are instruction compatible with their predecessors but offer significantly higher performance. The Intel i860 utilises a 64bit architecture and incorporates a 3D graphics processing unit on-chip, in addition to a floating point unit and multiple caches. The new generation transputer, the T9000, should significantly increase the performance of transputer based systems, when it is finally released. This device operates at a higher clock rate and uses faster links ($100Mbits^{-1}$). Performance is enhanced by the provision of an on-chip cache, a communications co-processor and an enlarged instruction set.

The performance of processors is often described in terms of MIPS (millions of instructions per second), MOPS (millions of operations per second) or MFLOPS (millions of floating point operations per second). However, the architecture of

processors is now so diverse that these ratings should be used only as a rough guide when comparing the performance of different processors. An operation that is executed in a single instruction cycle on one processor may take several cycles to execute on another. Manufacturers always quote the maximum possible attainable performance of their processors, which generally corresponds to the use of on-chip resources and a permanently full instruction pipeline. Fig 1.1 shows the increase in processor performance with time.

1.2 Issues in Processor Design

In 1945, while working as a consultant with the Moore School group, von Neumann issued a memo concerning the design of a new computer (EDVAC). This report, reputedly for the first time, referred to a memory *organ*, used to hold all the different types of data required by the computer.

This memo contained the first reference to what has become to be known as the "von Neumann Architecture" [3], which was used as the basis for processor architectures for well over 30 years. This type of architecture, shown in Fig 1.2, has four main characteristics:

i A single computing element consisting of a processor, memory and an input/output device.

ii A linear organisation of fixed size memory cells.

iii A low level machine language with instructions performing simple operations on elementary operands.

iv Sequential, centralised control of computation.

Data and instructions are stored in the same memory and are accessed via a single bus. If the performance of the processor exceeds that of the memory, then the

processor is forced to wait and the situation known as "bus bottlenecking" occurs. Bottlenecking represents the major limitation of von Neumann architectures, and is most apparent in high speed systems.

In order to increase the performance of such processors, various architectural and implementational modifications have been made to this basic structure [4].

1.2.1 Pipelining

The processor must fetch an instruction, decode it and then act upon it. The idea of pipelining is to use dedicated execution units for each of these functions, allowing them to operate simultaneously [4], [5]. This divides up the work required of the processor and increases performance. The basic form is the fetch, decode and execute pipeline which allows an instruction to be fetched (*pre-fetched*) while another is being decoded and another executed. Fig 1.3 demonstrates the action of such a pipeline when executing three consecutive instructions, A, B and C. The pipeline may be lengthened to increase the amount of operational parallelism, perhaps by including units to compute the address of operands.

Pipelining works most efficiently whenever consecutive instructions are accessed. Jump, call and context switching instructions render some portions of the pipeline invalid. In such instances, the whole pipeline must be refilled. This obviously reduces performance, and the advanced microprocessors incorporate mechanisms used to reduce the impact of this pipeline "flushing".

1.2.2 The Harvard Architecture

The performance limitations imposed by storing data and instructions in a single

addressable memory area may be alleviated somewhat by providing separate memories for data and instructions. The Harvard architecture, shown in Fig 1.4, allows the processor to fetch instructions and operands simultaneously, significantly increasing performance. This basic architecture may be extended, allowing multiple operand fetches to occur simultaneously, Fig 1.5. Due to its high data bandwidth capability, the Harvard architecture gas been utilised in a number of digital signal processing (DSP) devices [6].

1.2.3 Caches Memories

Modern processors operate at high clock speeds, requiring fast memory access. Dynamic RAM is unable to cope with the access speed requirements of the fastest processors, and the size of static RAM allows only a small amount of memory to be located on the processor board. Connecting a memory extension board slows down access times. The consequence of this is that fast systems may only access small memory areas at full speed.

The solution is to use a small, fast, memory to act as a storage buffer between main memory and the processor. Such a memory is called a "cache", and is sometimes incorporated on-chip for really fast access [7].

The effectiveness of a cache depends upon its access time and its "hit ratio" — how often the processor finds useful information in the cache.

Caches may be used to store data or instructions, requiring sophisticated control algorithms to maintain a high hit ratio.

The main motivation for using a cache based architecture is to decrease system cost. Fig 1.6 shows the breakeven points between caches and various memory speeds.

1.2.4 Extended Processing Units

Adding more instructions to a processor's instruction set in order to increase its performance also increases its complexity and die size. One method of increasing functionality without incurring this complexity is to use extended processing units (EPUs), or "coprocessors" [2]. Common EPUs include floating point coprocessors, DMA processors, memory management units and vector coprocessors.

1.2.5 RISC

Continued development of microprocessors resulted in devices utilising many complex instructions, requiring a large microprogrammed ROM and several cycles to execute — the so-called complex instruction set computers (CISC). Although this does ease the task of writing a compiler, it does tend to limit the performance of a processor. By using a smaller, simpler and more regular instruction set, instruction cycle times may be reduced. This is the approach taken by the reduced instruction set computer (RISC) philosophy [8].

Although the definition of RISC is far from standardised, any RISC should exhibit at least some of the following properties:

i Single cycle instructions.

ii Only LOAD and STORE instructions access memory, all other instructions access internal registers.

iii Simple instruction formats.

iv Hardwired, rather than microcoded, control units.

v A small, efficient instruction set.

Complex instructions are broken down into a series of shorter instructions. As the memory bandwidth requirement of RISCs is high, they must use high speed memory

in order to maintain a performance advantage. As the control unit is small, this releases space which may be used to provide a fast on-chip memory area. Example RISC processors include the Acorn Risc Machine (ARM), the MIPS 2200 and the Inmos Transputer.

1.3 Multiprocessors

When the performance requirement of a particular application cannot be met by a single processor, then a multiprocessor system must be used [9], [10]. Many types of multiprocessor are available, ranging from highly specialised to general purpose systems. Multiprocessors are commonly described in terms of Flynn's Taxonomy [11], which classifies architectures according to the presence of single or multiple instruction and data streams, below.

SISD (single instruction, single data) — serial computers.

MISD (multiple instruction, single data) — a generally impractical approach.

SIMD (single instruction, multiple data) — the same instruction is simultaneously executed on different data.

MIMD (multiple instruction, multiple data) — multiple processors autonomously operate on diverse data.

Not all multiprocessor architectures fit neatly into these categories, some may possess properties attributed to more than one taxon. Multiprocessors may be thought to consist of a number of processing elements (PEs) connected to memory units (MUs) through an interconnection network (IN). The size and nature of these three elements varies enormously among different multiprocessors [12], [13], [14], [15], [16].

A task may be broken down into processes which may operate in parallel. The size of these processes is termed the "grainsize". A program running small processes is said to exhibit *fine grain* parallelism, whereas one running large processes is said to exhibit *coarse grain* parallelism [6].

1.3.1 Interconnection Networks

The variation of IN topologies is considerable, a sample of the most popular is shown in Fig 1.7. Some, such as the FFT butterfly, have been designed to implement a particular class of algorithm, whereas others, such as the hypercube, have been designed to implement a large number of algorithms with optimum efficiency. Some multiprocessors utilise reconfigurable IN topologies, which considerably increases their versatility, but also their complexity. The extent to which a multiprocessor supports additional processors is termed its "scalability", and is heavily influenced by the interconnection network topology.

1.3.2 SIMD

Fig 1.8 presents a representation of the standard SIMD model. Processor and systolic arrays are the two most common forms of SIMD architecture. Processor arrays are used for numerically intensive applications which require regular, synchronous, computation. The most popular IN schemes used in such architectures are the mesh and crossbar.

Some array processors, such as Illiac IV [2], incorporate processors utilising wordlengths of up to 64bits. A number of systems utilise simple, 1bit, processing elements. These processors use *planes* of memory, and are particularly efficient at

implementing image processing algorithms. Example systems include the ICL Distributed Array Processor (DAP) and Thinking Machines Connection Machine, which utilises up to 65,536 processors.

Systolic architectures were first proposed by H.T. Kung in the early 1980s [17], [18]. The term "systolic" arises from the manner in which data is "pulsed" through the system. The processors are tightly synchronised and connected by a regular IN. Although the IN topology of these processors is highly optimised to implement particular applications, reconfigurable arrays are available which are significantly more versatile. Systolic arrays are particularly efficient at implementing certain signal processing algorithms.

1.3.3 MIMD

MIMD systems generally make use of more sophisticated processors than SIMD systems, and lend themselves to coarse grain parallelism. The processors operate asynchronously and often possess their own memory area. Whereas each processor in a SIMD system is controlled by a centralised controller, the processors in an MIMD system operate autonomously. MIMD systems may be broadly categorised as either shared memory or distributed memory architectures [6], [14].

1.3.3.1 Shared Memory

The processors in this type of architecture communicate through an area of shared memory. It is important to ensure that the data in this area is not corrupted by uncontrolled access. This is usually carried out by using a "semaphore" protocol [6], [19], [20]. A semaphore consists of a word in memory and controls access to

an area, or *domain*, of memory. The state of the semaphore determines whether or not the domain is in use by a processor, and so determines whether or not other processors may access it. The processors test the semaphore, and act appropriately. Whenever a process gains access to a domain, it "locks" it by setting the semaphore, and "unlocks" it when finished by resetting the semaphore. In order for a semaphore protocol to work, the processors must use so-called "atomic" instructions to test the semaphore and set it, if appropriate, in a single bus cycle. This eliminates the possibility of semaphore ambiguity when two processors interleave their memory accesses.

Repeatedly testing and failing a semaphore, "spin locking", can degrade performance by increasing the memory traffic [21]. The simple shared memory architecture shown in Fig 1.9 is especially susceptible to this problem as the von Neumann bottlenecking problem is increased due to the additional processors. More sophisticated semaphore protocols do not allow spin locking, which helps to reduce the amount of bus traffic [22].

Various interconnection networks have been introduced in order to reduce the bus saturation problem, including the crossbar network and hierarchical bus structures, Fig 1.10 and Fig 1.11. These may be either "static", as in the hierarchical bus, or "dynamic", as in the crossbar switch. Dynamic interconnection networks allow communications paths to be made "on the fly" and are able to offer higher communication bandwidths and lower latencies. However, they are complex and hence expensive to implement,[6],[13],[20].

Addition of local memory and caches increases the performance of any of the above configurations. If shared data is held in a cache, it must be updated whenever other processors change any related variables in other caches or main memory. This

"cache coherency" requires the addition of extra hardware or software, which increases complexity and may reduce performance [23],[24],[25],[26], .

1.3.3.2 Distributed Memory

Distributed memory systems consist of nodes comprising a processor and memory pair which are connected via an interconnection network, and may take on any of the forms outlined in Section 1.3.1. Data is transferred by passing messages across the IN.

The development of distributed memory systems has been motivated by the desire to produce large, scalable systems capable of providing a high performance for a variety of applications.

The hypercube, in particular, is a popular interconnection network configuration, possessing a high degree of interconnectivity and relatively low communications diameter. Commercially available hypercube distributed memory systems include the Cosmic Cube [27], the AMTEK 2010 and the Intel iPSC2 [28]. The new generation of hypercube machines will utilise specialised communications processors to provide efficient routing through the use of "wormholing" [29], which reduces the communications latency whenever a message is routed through intermediate processors.

The transputer, in particular, has been designed with large scale distributed memory systems in mind [29]. The provision of four bidirectional serial communication "links" allows very large systems to be easily constructed with these devices.

Modules are available which connect a transputer to other processors, which are used as slaves. These companion processors include the Motorola DSP56001

programmable digital signal processor, the Motorola DSP56200 FIR chip (both from Perimos), the Intel i860 and Zoran vector processors [30]. These processors certainly boost the apparent performance of the transputer, but often the interprocessor communication bandwidth is low, and scalability is not supported.

1.3.4 Multiprocessor Performance

The maximum speedup that may be attained by a multiprocessor comprising n processors is n times that of a single processor. This ideal performance is only attainable if the interconnection network is capable of sustaining the total communications bandwidth required by the processors. The communication bandwidth of the interconnection network is the limiting factor in the performance and scalability of a multiprocessor. Hence the choice of interconnection network must be carefully considered when designing a multiprocessor system [6], [20], [31].

Transputer systems offer a high communications bandwidth which is proportional to the number of processors. Thus, the scalability of such systems is large. However, performance will suffer whenever a message issued by a transputer must be routed through intermediate transputers in order to reach its destination, as the intermediate transputers must devote time to through-routing the message [29].

Due to the vast variety of multiprocessor architectures, it is difficult to apply benchmark programs as a means of comparing the performance of different systems. The development of multiprocessor benchmarking programs is a growing area of research [32] [33].

1.4 Digital Processing of Signals

Although the mathematical theories and tools forming the basis of the eclectic field of digital signal processing had been brought together by the middle of this century, practical implementation was severely limited by the available technology.

The development of digital filter theory [34] and the Fast Fourier Transform algorithms [35] coupled with the development of integrated circuit technology resulted in the emergence of feasible digital signal processing systems in the mid 1960s. Digital signal processing has now grown into an established and ever expanding discipline. Application areas include audio and video processing, communications, seismology and tomography [36].

Digital processing of signals offers more control, and higher predictability, than its analogue counterpart. Some applications may only be implemented using digital techniques. Some applications may be too expensive, or be too slow, to implement digitally, however, and must use analogue technology.

1.4.1 Sampling Theory

A digital signal consists of a series of values defined at discrete intervals of time. When an analogue signal is modulated by a set of pulses (delta functions), the resultant output is a quantised form of the input. This process is known as "sampling" [37], and the frequency at which the pulses are applied is termed the "sampling frequency". Sampling theory maintains that the maximum useful frequency content of a digital signal is limited to half the sampling frequency (the Nyquist frequency). Any frequency component higher than the Nyquist frequency is "aliased", or folded around the Nyquist frequency, into the sub-Nyquist range, resulting in signal distortions.

Furthermore, the frequency spectrum of the sampled signal exhibits a periodicity.

Whenever an analogue signal is sampled, it must first be band limited by a low pass analogue filter, to half the sampling rate, which eliminates aliasing. When a sampled signal is to be converted back to the analogue domain, it must be passed through a similar filter in order to properly reconstitute the signal. The entire process is outlined in Fig 1.12.

1.4.2 Filter Structures

Digital filters utilise multiplication and addition operations to modify a signal's frequency and phase spectra. The most widely used digital filtering types are the finite impulse response filter (FIR) and the infinite impulse response filter (IIR), example architectures of which are shown in Fig 1.13 and Fig 1.14. Although IIR filters are more economical, their inherent feedback properties render them liable to unstable behaviour. FIR filters are stable and offer linear phase characteristics, but tend to require more operations than IIR filters [38].

1.4.3 Quantisation Effects

Due to the finite length of their registers, digital devices can represent information with only a finite precision. An 8bit device is capable of half the precision of a 16bit device, and so on. This has consequences relating to dynamic range, signal to noise ratio (SNR) and filter response approximations. The limited precision with which filter coefficients may be represented forces the possible frequency responses to be quantised. In addition, the truncation caused by transferring data from a long accumulator to a shorter memory location introduces noise into the system. Noise is

also introduced by the analogue to digital converter. As a rough measure, 1bit of noise reduces the SNR by 6dB. Noise considerations are an important design aspect of digital hardware systems [37].

1.4.4 Programmable Signal Processors

Advances in processor design methods, coupled with the desire to make signal processing hardware more compact and manageable resulted in the production of the first programmable digital signal processor, the NEC μPD7720 in 1980. The main difference between digital signal microprocessors and their general purpose counterparts is in the provision of a fast hardware multiplier [7], [39].

In order to provide maximum data throughput, these processors incorporate dedicated registers which act as multiplier input buffers. This allows operands to be fetched while the multiplier is operating. Arithmetic precision is maintained through the use of double length multiplier output registers (accumulators), which are often extended to accommodate overflows.

As the speed of multipliers increased, so did the need to supply them with data. Some form of the Harvard architecture is used in every recent processor, including areas of on-chip memory which may be simultaneously accessed at full bus bandwidth.

The use of register indirect addressing modes helps to speed up memory accessing by removing the need to explicitly calculate addresses. The more recent signal processors incorporate a number of address registers which may be modified in parallel with memory and multiplier operations. A summary of presently available signal processors is given in Table 1.1. More detailed descriptions may be found in

[40],[41],[42],[43],[44],[45],[46].

The most recently introduced signal processor from Texas Instruments, the TMS320C40, incorporates six byte wide communication interfaces, each capable of transferring data at 20Mbytes^{-1}. This is the first processor to have been designed to interface, at high speed, with other similar devices, allowing point to point interconnection network topologies such as the 3D mesh and 6D hypercube to be directly implemented. The TMS320C40 points to the convergence of two areas of high performance computing — digital signal processing and parallel computing.

1.5 Concurrent Digital Signal Processing

Parallel signal processing systems based on SIMD architectures have been in existence for a number of years [47]. These tend to be highly synchronous and are capable of implementing only a small class of algorithms efficiently. The application of MIMD architectures to digital signal processing applications is an area of active research. Transputer arrays have proved popular, as they are easily constructed and programmed [48], [49]. However, the development of architectures designed specifically to cater for the requirements of specialised digital signal processing elements has not yet reached maturity [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61].

The problems of utilising signal processors in MIMD architectures are three-fold. Firstly, their data requirement is very high, often requiring up to three memory accesses per instruction cycle, which puts considerable strain on the interconnection network and limits scalability. Secondly, although performance models of multiprocessor systems do exist [32], they tend to be stochastic rather than deterministic and so are not applicable to real-time digital signal processing

applications, which are generally deterministic. Thirdly, signal processors have been optimised to pass data through their multipliers as quickly as possible, not to interact in a multiprocessor environment. Hence, any overheads attached to interprocessor communication management may significantly affect the performance of the processor.

Conversely, these properties also aid the system designer. Digital signal processing algorithms generally require large amounts of data, and very little (if any) control information. This allows data to be transferred efficiently to the processors in large buffered packets (vectors). As the execution of the signal processing algorithms tends to be fixed, then the intervals at which the processors require data is also fixed. This allows the data transfers to be staggered, reducing communications resource contention.

Multiprocessor architectures need to be found that allow fast data transfer, to keep the signal processors fed, without incurring excessive communications management overheads, which would slow down the processors.

1.6 Summary

This introduction has provided an overview of the growth areas of high performance multiprocessing and digital signal processing. Although the architecture of the earlier digital signal processors differed markedly from their general purpose counterparts, more recent devices have started to incorporate a blend of architectural strategies. For example, digital signal processors are accessing larger memory spaces, and may be programmed with high level languages, whereas general purpose processor are breaking away from the von Neumann architecture by using multiple bus memory architectures.

Digital signal multiprocessor systems tend to suffer from low performance or reduced scalability as a consequence of relatively low bandwidth interprocessor communication mechanisms. This is changing as more research is aimed at the communications requirements of these systems.

The interprocessor communication problem has been acknowledged by Texas Instruments, in their new "parallel" signal processor, the TMS320C40, which uses autonomous DMA ports in a similar manner to the transputer. This device incorporates six byte wide ports, capable of a total transfer rate of 120 $Mbytes^{-1}$. This is a high transfer rate (over twelve times faster than the transputer), and the six ports allow 3D meshes or 6D hypercubes to be directly implemented. But it is important not to get carried away with this specification. The DMA ports will only operate at full speed when accessing internal memory; external accesses are multiplexed onto a single interface which must be shared with other DMA transfers and cpu instruction /data fetches. This device is best suited to a point to point communications scheme, which are prone to through-routing latencies and a corresponding performance decrease. Finally, these devices possess a high pin count (which increases pcb costs), relatively high power consumption and are expensive.

Some applications may not require 32bit floating point operations, or such a high degree of interconnectivity, but would nevertheless benefit from a multiprocessor implementation. The problem here is that the lower range signal processors provide limited multiprocessor communications support, which results in an inefficient system. Developing an optimally efficient interprocessor communication mechanism for such systems would allow more data processing to take place, increasing the effective number of MIPs per processor and reducing overall cost. The resultant multiprocessor

need not be homogeneous (consisting of identical processors), a heterogeneous system (consisting of different type of processor) could be used to maximise efficiency. Such a system could be used either as an inexpensive stand-alone signal processor, or as an add-on accelerator. It would be important in the design of such a system to ensure that the operation of the processors was fully understood, especially the mechanisms involved in computation and communication.

The assessment of two different types of microprocessor in terms of signal processing and interprocessor communications, with a view to combining them to form an efficient and inexpensive digital signal multiprocessor forms the subject matter of this thesis. Chapters 2 to 4 introduce the Inmos transputer, assessing its applicability to signal processing. Chapters 5 and 6 outline the architecture and operation of the Motorola DSP56001 digital signal processor. Both processors are compared by their ability to implement a multichannel digital filtering application. The conclusions drawn from these chapters are used in the design of a hybrid (heterogeneous) multiprocessor (Hymips) in chapters 7 and 8. Although this multiprocessor was developed as a general purpose digital signal processing platform, the research involved in its development was closely aligned to a particular high performance audio bandwidth application. The reader is referred to the list of conference papers presented in Appendix F for further information. Chapter 9 offers a theoretical analysis of system performance, together with empirical verification of the performance equations. Finally, chapter 10 provides a conclusion and suggestions for further work.

Firm	Model	Date	Description	Mac Time
AMI	S2811	1978	The first DSP designed; 12/16-bit fixed point; not released until 1982 because of technology problems	300
	S28211/2	1983	An update of the 2811	-
Analog Devices	ADSP-2100	1986	16/40bit fixed point	125
	ADSP-2100A	1988	An update of the 2100	80 or 100
	ADSP-2101/2	1988	A2100A with internal RAM and peripherals; 2102 has mask programmable program ROM.	-
	ADSP-2111	1990	2101 with host port	-
AT&T	DSP1	1979	Early 16/20bit device, marketed internally	800
	DSP32/32C	1984/88	32bit floating point	160/80
	DSP16/16A	1987/88	16bit fixed point	55/25
	DSP16C	1990	DSP16A with voice band Codec	-
Motorola	DSP56001	1987	24bit fixed point, on-chip peripheral ports	97.5/75/50
	DSP56000	1987	56001 with mask programmable ROM	-
	DSP96001	1990	32bit IEEE floating point.	75
	DSP96002	1990	96001 with additional memory port.	-
NEC	μPD7720	1980	A popular early DSP.	250
	μPD7720A	-	Update of 7720	244
	μPD77230	1985	32bit floating point	150
	μPD77220	1986	24/48bit fixed point.	100
	μPD77C25	1988	7720A upgrade	122
	μPD77240	1990	Update of 77230	90
Texas Instruments	TMS32010	1982	Popular 16bit fixed point DSP	390
	TMS32020	1985	Update of 32010	195
	TMS320C25	1987	CMOS update of 32020, with additional instructions	100
	TMS320C30	1988	32bit floating point	60
	TMS320C50	1990	16bit fixed point	35
	TMS320C40	1992	32bit floating point with 6 byte wide DMA ports. Designed for multiprocessing	60
Sharp	LH9124	1991	24bit fixed point frequency and time domain processor.	-
	LH9320	1991	Address generator for the LH9124	-

Table 1.1 A Summary of Popular Programmable Digital Signal Processors

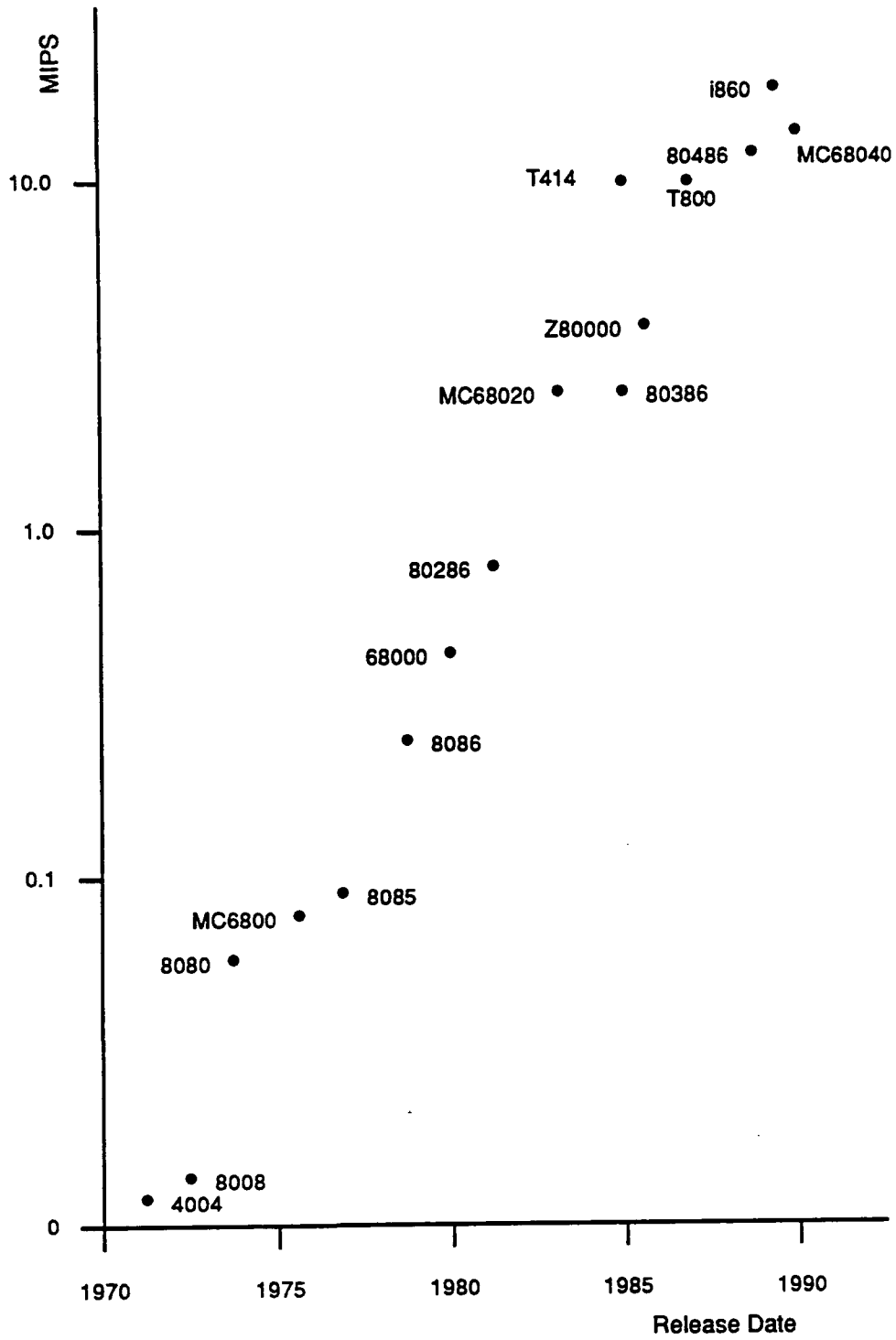


Fig 1.1 The Increase of Processor Performance with Time

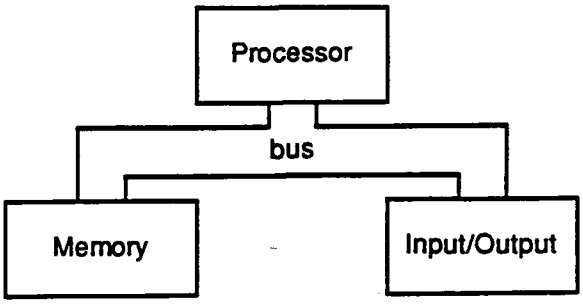


Fig 1.2 The von Neumann Architecture

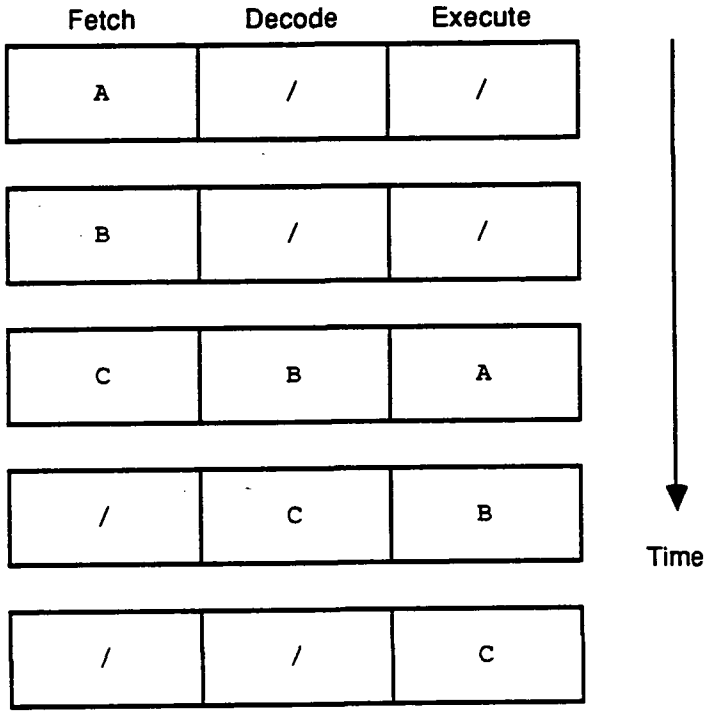


Fig 1.3 The Execution of an Instruction Pipeline

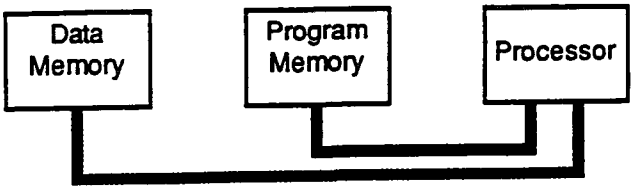


Fig 1.4 The Harvard Architecture

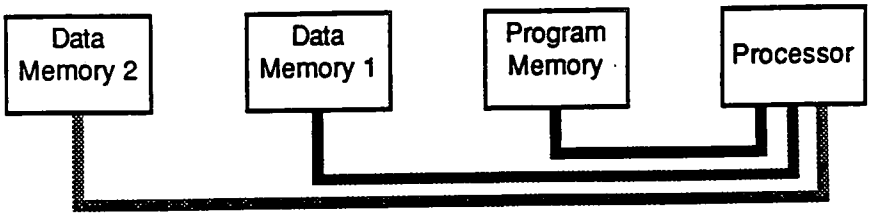


Fig 1.5 A Modified Harvard Architecture

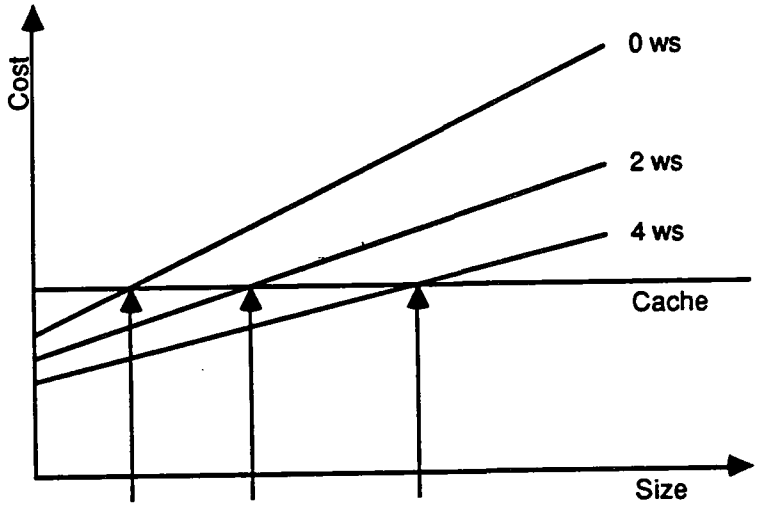


Fig 1.6 Cache / Main Memory Breakeven Points

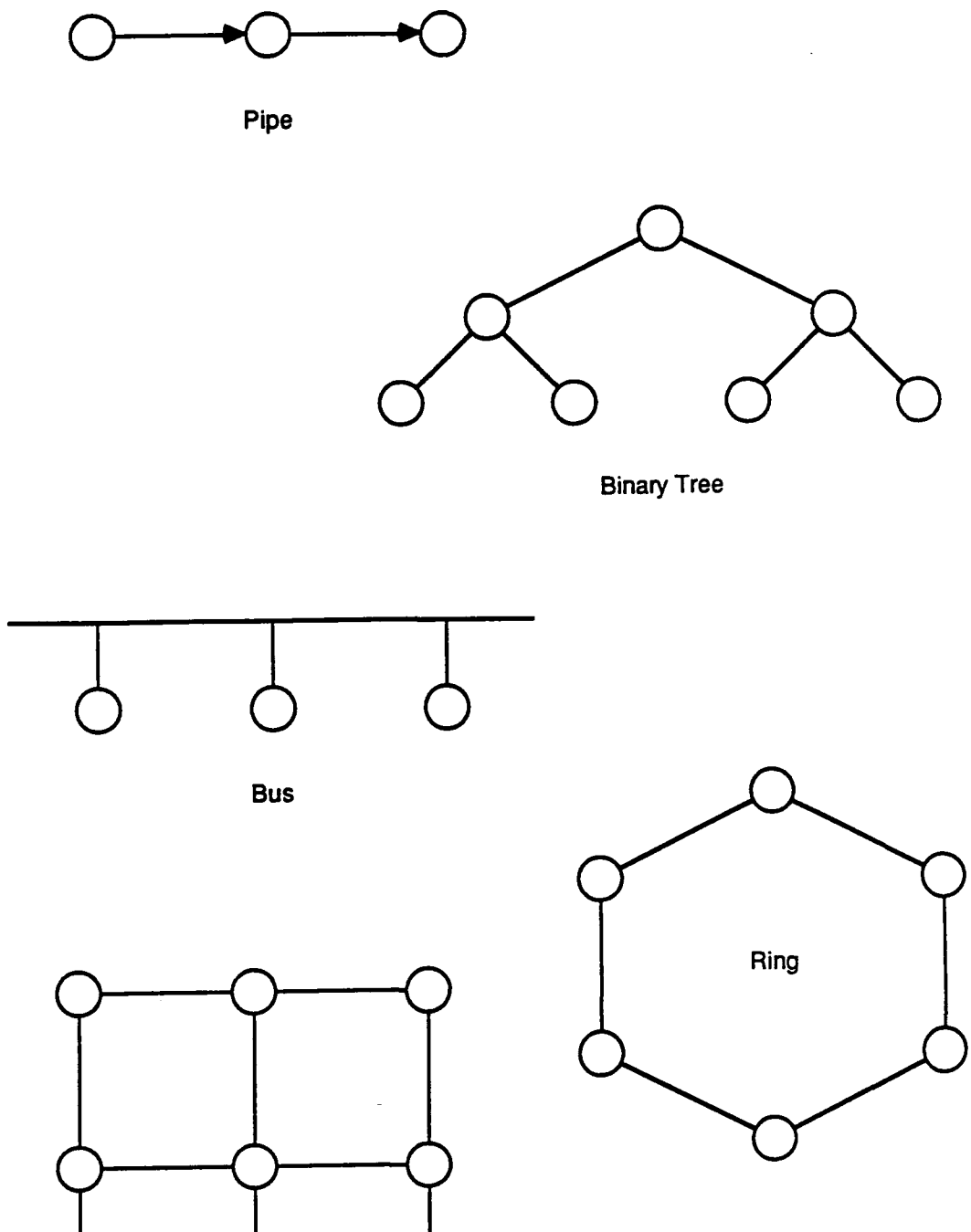


Fig 1.7 Example Interconnection Network Topologies

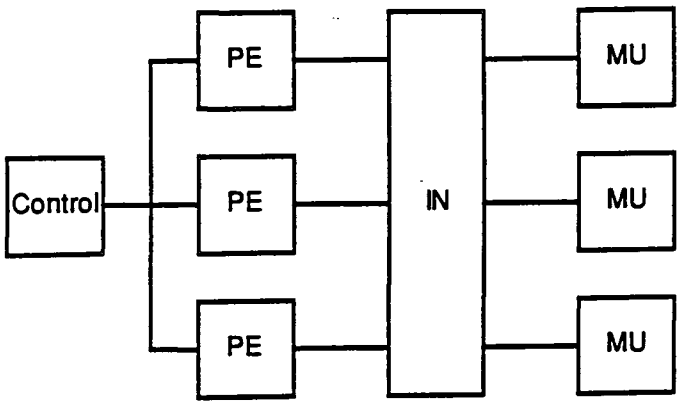


Fig 1.8 The SIMD Model

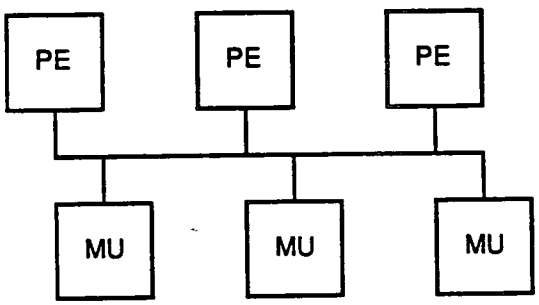


Fig 1.9 The Simple Shared Memory Architecture

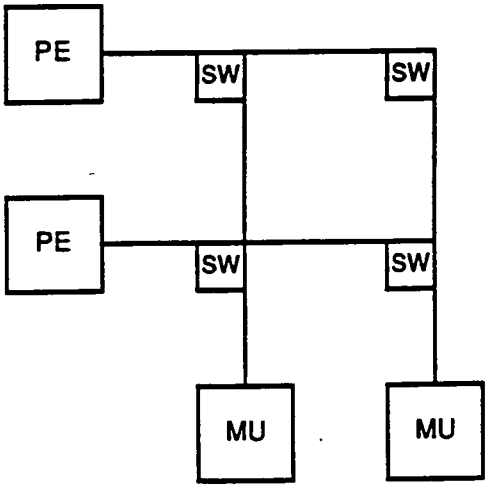


Fig 1.10 The Crossbar Interconnection Scheme

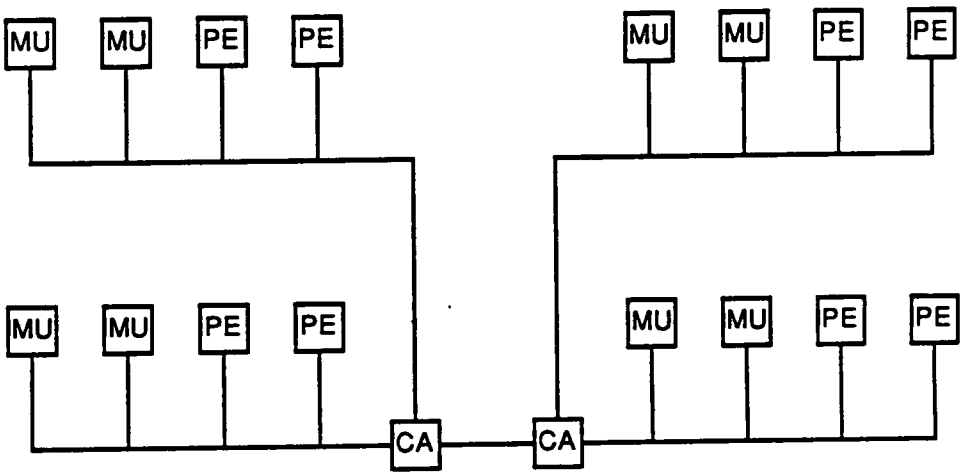


Fig 1.11 An Example of a Hierarchical Bus Structure

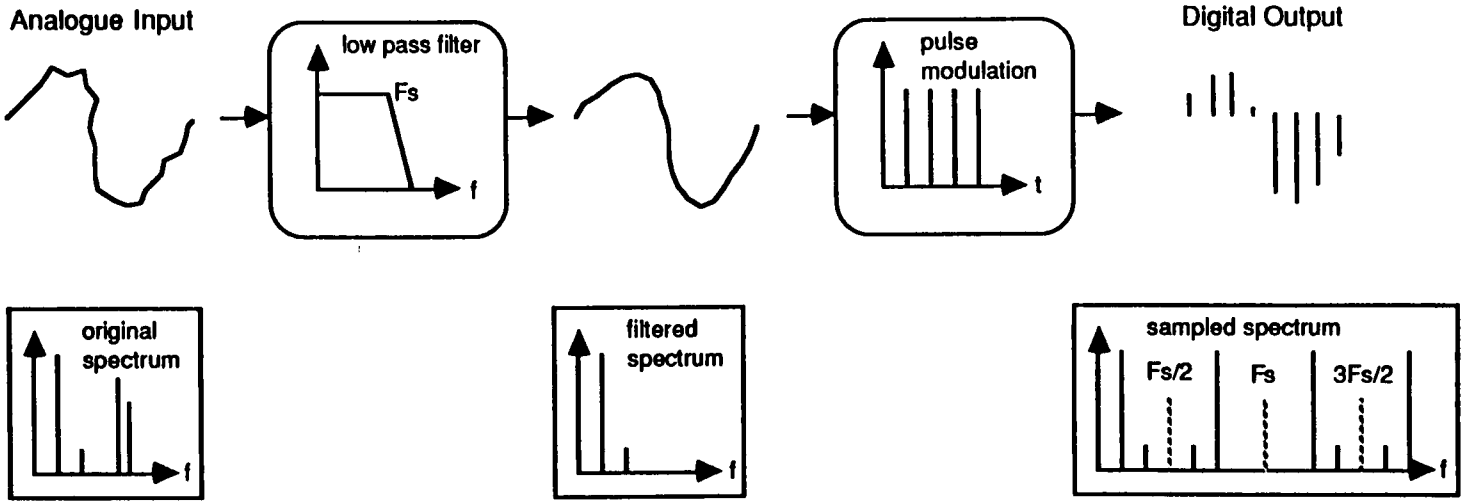


Fig 1.12 The Sampling Process

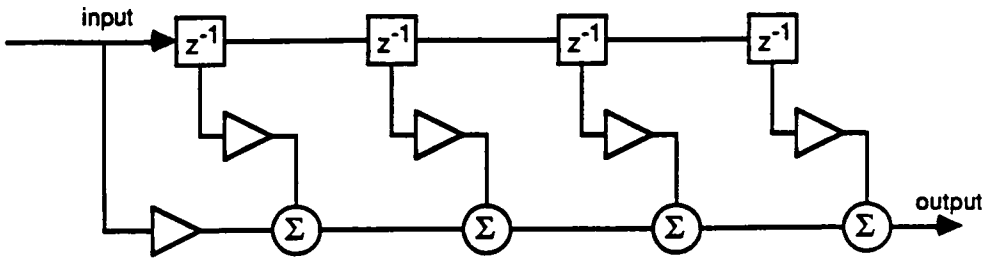


Fig 1.13 An FIR Structure

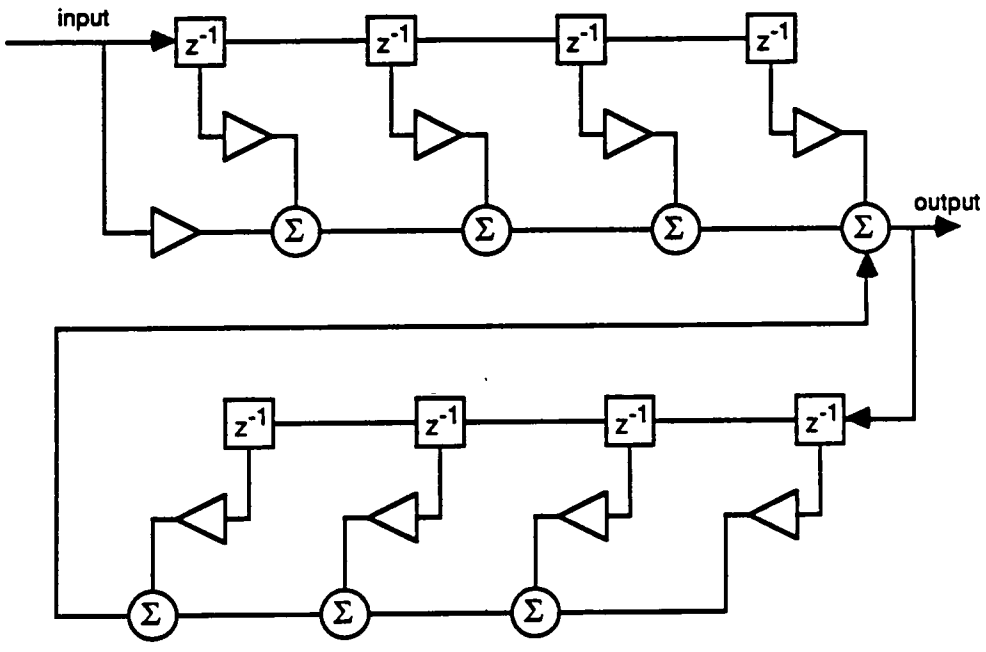


Fig 1.14 An IIR Structure

Chapter 2

The Transputer

2.1 Introduction

The term "transputer" refers to a family of RISC - like microcomputers manufactured by Inmos (now a subsidiary of SGS Thomson Microelectronics Group) [9], [30]. The major differences between the devices are the wordlength (16 bit or 32 bit), the size of the internal memory (2kbyte or 4kbyte) and the incorporation of a floating point unit (fpu - T80x transputers only), the core architecture remaining similar. The generic term "transputer" will be used to refer to the family as a whole in this thesis, any particular architectural difference being pointed out when necessary. As the transputer was designed with embedded systems applications in mind, it requires only an additional bootstrap ROM for stand-alone operation.

The key feature of the transputer architecture lies in the inclusion of "link engines". These are essentially DMA controllers which transfer data between memory and an external, bidirectional, serial interface. The link engines operate concurrently and asynchronously both with themselves and the central processing unit (cpu). These links allow any transputer to be directly connected with up to four other transputers, allowing asynchronous communication to occur concurrently with cpu operation. This ability to overlap communication and computation is the main feature distinguishing

the transputer from other commercially available processors[1]. It is these links that allow a network of an arbitrary number of transputers to be easily implemented, Fig 2.1.

The links provide for a point to point message passing communication paradigm to be implemented. The advantages of point to point links over multi-processor buses are [30]:

i. There is no contention for the communications mechanism, regardless of the number of transputers in the system.

ii. There is no capacitive load penalty as transputers are added to the system.

iii. The communications bandwidth does not saturate as the size of the system increases. Rather, the larger the number of transputers in the system, the higher the total communications bandwidth of the system. However large the system, all the connections between transputers can be short and local.

It must be considered, however, that the communications bandwidth across a link is lower than may be obtained over a bus. Furthermore, only four links are provided, which limits the topologies which may be realised using direct connections. Whenever messages must be routed through intermediate processors, not only is there a possibility of link communication contention, but the routing software must be explicitly programmed, which further adds to the communications overhead and detracts from overall performance.

The point to point message passing paradigm is efficiently implemented by the

[1] An exception to this is the TMS320C40 programmable digital signal processor from Texas Instruments, which has been recently released.

transputer's "native" language, Occam (now Occam2) [62]. The transputer was designed around the ideas embodied in Occam, which itself is based upon the theory of Communicating Sequential Processes (CSP) [63].

Occam allows any number of parallel processes to be incorporated into a program. The processes communicate over Occam "channels", and may run either on a single transputer or be mapped onto several transputers for true concurrency and increased performance. Parallel processes running on a single transputer communicate through "soft" or "internal" channels, whereas those running on different transputers communicate over "hard" or "external" channels, implemented by the link interfaces.

Parallel processes running on a single transputer are managed by a microcoded scheduler. The scheduler ensures that no process monopolises the cpu by periodically "timeslicing" processes, deschedules processes when they are no longer able to proceed, and reschedules them again when they are. The operation of the scheduler is normally transparent to the programmer.

Section 2 introduces the architecture of the transputer, and its main execution units. The instruction set is covered in section 3. Section 4 goes on to discuss the effect which the manner in which the transputer deals with instructions has on performance. The construction of sequential and parallel processes are described in Sections 5 and 6 respectively. The communication mechanisms are covered in Section 7, followed by the memory map, events and bootstrap procedures. Section 11 introduces methods of optimising performance, and finally section 12 offers a summary.

2.2 Transputer Architecture

The architecture of each processor in the transputer family is similar. A schematic representation of the major blocks is shown in Fig 2.2. for integer and floating point transputers. The individual blocks will now be discussed separately.

2.2.1 The Central Processing Unit

The microcoded central processing unit (cpu) contains six registers which are used when implementing sequential processes. These are:

i. Wptr The workspace pointer, which points to an area of memory where local variables and process parameters are stored.

ii. Iptr The instruction pointer, which contains the address of the next instruction to be executed.

iii. Oreg The operand register, used to store instruction operands.

iv. Areg The top of the evaluation stack.

v. Breg The intermediate evaluation stack register.

vi. Creg The bottom of the evaluation stack.

The evaluation stack is used for integer and address arithmetic. Loading a value onto the stack pushes Areg into Breg, and Breg into Creg, before loading Areg. Storing a value from the stack pops Breg into Areg, and Creg into Breg, after Areg is stored. The floating point unit contains three similar, floating point, registers that behave in the same way.

The microcoded scheduler also resides in the cpu. Using a microcoded scheduler removes the need for a software kernel and so allows efficient management of concurrent processes.

The cpu also allows for real-time programming by incorporating two timer registers, one operating at a resolution of 1μs for use by high priority processes, the other at 64 μs for use by low priority processes.

2.2.2 Internal Memory

The transputer incorporates either 2 or 4kbytes of static memory on chip (device dependent), which occupies the lowest area of the memory map. In accordance with the RISC philosophy, this memory is accessed in a single processor cycle. Any frequently used variables should reside here, in preference to the slower external memory area.

2.2.3 External Memory Interface

The external memory interface (EMI) provides access to up to 4Gbyte of memory. Most transputers incorporate a versatile EMI, which is able to interface to most types of dynamic as well as static RAM. This feature greatly simplifies hardware designs that use dynamic RAM.

The address and data buses are multiplexed. The full 32-bit data bus is used, but only the 30 most significant address lines are brought out to the EMI, which corresponds to a word aligned external addressing scheme. Individual bytes are accessed using individual byte strobes. One of the seventeen possible EMI configurations is selected after processor reset.

Since the data and address lines are multiplexed, external memory access is significantly slower than internal memory access, even when no wait states are used. Using this EMI, external memory access is three times slower than internal memory

access. If faster external memory access is required, then the T801 transputer may be used. This floating point transputer uses non-multiplexed data and address buses, resulting in an external access time of two processor cycles. However, due to the extra bus pins, most of the EMI's control and strobe lines have been lost. The EMI of the T801 is very simple, and is only suitable for direct connection to static RAM.

2.2.4 Links

Most transputers support four bidirectional link interfaces (the exceptions being the budget T400 and the M212 Disk Controller), which are used to connect either to other transputers or to other types of device, through a link adapter. Each link consists of an input and an output channel. A single byte is sent at a time, and for each byte sent an acknowledge packet is received on the input of the same link. Data and acknowledge packets may be multiplexed on the same link. The acknowledge packet is transmitted as soon as an input packet begins, allowing for continuous communication (except on the early revA T414s, which did not implement this overlapping protocol). The structure of the data and acknowledge packet is shown in Fig 2.3.

Links may operate at 5, 10 or 20 $Mbits^{-1}$, regardless of the internal clock speed, allowing transputers of different clock speeds to be linked together. Links can carry information at a maximum of 1.74$Mbytes^{-1}$ in unidirectional mode, and 2.35$Mbytes^{-1}$ in bidirectional mode [30].

2.2.5 The Floating Point Unit

Some transputers (the T80x series) incorporate an on chip floating point unit (fpu), conforming to ANSI-IEEE 754-1985 standard [30]. This operates on operands, the addresses of which are supplied by the cpu, and executes concurrently with the cpu. The fpu is capable of sustaining 2MFLOPS (for a 20MHz processor).

2.3 The Transputer Instruction Set

The transputer instruction set is byte orientated, and so is independent of processor wordlength. Thus, all the transputer family may use the same compiler. Each instruction has a similar format [64].

An instruction consists of a single byte, divided into two four bit "nibbles". The most significant nibble represents a function code, the least significant represents the operand of the function. The least significant nibble is loaded into the lowest nibble of the operand register, Fig 2.4.

2.3.1 Direct Instructions

The four bit representation allows sixteen instructions to be directly implemented, each with an operand value ranging from zero to fifteen. According to the RISC design philosophy, Inmos have implemented the most common instructions in this manner. Among these are the local load and store instructions, which according to Inmos, are most commonly used with small operands (ie values less than sixteen) [30].

2.3.2 Prefixing

Of course, the transputer uses more than sixteen instructions, and uses operands of up

to 32 bits. These other instructions and larger operands must be "built" using the prefixing instructions, which are included in the set of direct functions. The prefix (`pfix`) instruction first loads its operand into the operand register, then left shifts this value four places. The negative prefix (`nfix`) instruction operates in a similar manner, except that the operand register is complemented before the operand is loaded. Operands of up to 32 bits may be loaded in this way, using additional prefixing instructions. The number of prefix operations used to load an operand will be termed the *level* of prefixing.

2.3.3 Indirect Instructions

The operate (`opr`) function has been included in the set of direct functions. The operand of this instruction is interpreted as another instruction, which operates on the evaluation stack. Sixteen indirect functions may be encoded in a single byte. Other indirect instructions may be invoked by extending the operand register, using the prefix function. Examples of instruction encoding are given in Table 1.

Occam Program	Assembler Mnemonics
`x := 0`	`LDC 0` `STL x`

`x := -256`	`NFIX 1`
	`PFIX 0`
	`LDC 0`
	`STL x`

`Areg + Breg`	`OPR 5`
`Areg AND Breg`	`PFIX 4` `OPR 6`

Table 2.1 Examples of direct, prefix and indirect instructions

2.4 Performance Implications of the Instruction Set

Inmos claim [30] that *"about 70% of executed instructions are encoded in a single byte...Many of these, such as LDC and ADD require just one processor cycle"*. It would certainly seem that coding is efficient, although the user would be able to make a more objective judgement if the source code for these programs were made available. The byte wide instruction format does have consequences relating to overall performance.

Although many instructions require only a single processor cycle to execute, they often require prefixing to load in their operands, which adds to the overall execution time of the instruction. It must be remembered that the timing information that Inmos publishes relates only to the cycle times of the instructions, and does not include any time taken to extend the operand.

The cpu reads in a *word* of program memory at a time, allowing up to four instructions to be loaded in one processor cycle, providing that on-chip memory is used. This decreases the bus bandwidth required by the cpu, increases the efficiency of instruction prefetch and reduces the overheads attached to jumping. The reduced bus bandwidth requirement is one of the reasons why link operation only minimally degrades cpu performance.

2.5 The Implementation of Sequential Processes

Sequential processes are executed using the six registers contained within the cpu. Every sequential process uses two areas of memory. The first is the program area, which is referenced by the instruction pointer (Iptr) and provides the instructions. The second is the workspace, which is referenced by the workspace pointer (Wptr) and is

used to store local variables and values associated with timers and alternatives. Expressions are evaluated on the evaluation stack. Local variables are addressed relative to the workspace pointer, non-local variables are addressed relative to the address held in Areg. A schematic representation is given in Fig 2.5.

2.6 The Implementation of Concurrent Processes

The instruction set, together with the scheduler, allows for the efficient implementation of logically concurrent programs on the transputer. A parallel program consists of a set of sequential processes, which usually communicate with each other. The scheduler ensures that these processes are all given an equal share of processing time, although it may interrupt or deschedule a process under certain circumstances. Whenever it is interrupted, a process completes the instruction that it is executing before the contents of the registers are saved in the auxiliary registers (reserved locations in internal memory). The process may be resumed at a later time by restoring these register values. Whenever a process is descheduled, however, the registers are not saved. It is thus important that no important information is held in the evaluation stack when a descheduling instruction is executing, as this information will be lost if descheduling occurs.

A parallel program running on the transputer, then, may be thought of as a set of sequential processes that are scheduled, interrupted and descheduled under the control of a run time management kernel — the scheduler.

What follows is a description of the software and hardware mechanisms used by the transputer to implement parallel programs, priority and the structure of both non-prioritised and prioritised programs. The description is fairly detailed, as an

appreciation of the construction of, and the overheads associated with, single processor concurrent programming is important if the effects on performance are to be fully understood.

2.6.1 Workspace

Each process uses its own workspace (WS) in a similar manner to the way workspace is used in purely sequential programs. The cpu registers are also used in the same way. Whenever a process begins execution, its workspace and instruction pointers are loaded into the appropriate registers in the cpu. In addition to storing variables, timer and alternative information, the workspace is also used to store the process instruction pointer, values associated with communication and scheduling information. All these non-variable values are stored in negative workspace locations.

2.6.2 The Process Descriptor

The descriptor of a process is the sum of its workspace address (which is word aligned — ie its byte selector is zero) and its priority (either 1 or 0), which occupies the lsb. The process may be completely identified in a program by its descriptor.

2.6.3 Scheduling Lists

A process may be either *active* (scheduled) — being executed or waiting to be executed — or *inactive* (descheduled) — waiting for communication or until a specific time. Inactive processes consume no cpu time. The scheduler manages the processes by maintaining two linked lists (or queues) of processes, one for each priority level. The scheduler uses two registers for each list, one pointing to the front, the other to

the back, Fig 2.6. Whenever a process is descheduled, then its instruction pointer is saved in workspace location -1, it is taken off the queue and the next process on the queue is executed. It takes about 18 processor cycles to reschedule a process.

2.6.4 Priority

The transputer supports two levels of priority, high (0) and low (1). High priority processes run in preference to low priority processes.

A low priority process will run until either

i. it has been executing for two "timeslice" periods (2048 high priority timer "ticks", about 2μs), in which case it is put to the back of the queue at the earliest opportunity.

ii. it has to wait for communication or timer input, in which case it is descheduled.

iii. a high priority process becomes active, in which case the low priority is interrupted and execution switched to the high priority process at the earliest opportunity.

A high priority process will run until it is unable to proceed as it is waiting for a communication or timer input, in which case it is descheduled.

The scheduler will normally interrupt a low priority process in order to execute a high priority process at the end of the current instruction. However, there are six "interruptible" instructions, concerned either with communication or timer input [65]. It is important that no additional information is contained in the stack when these instructions are executing. Once one of these instructions is interrupted, then the instruction pointer of the low priority process is saved in its workspace, and the high priority process allowed to begin. Typical process switching latency is 18 processor cycles.

Similarly, any process may be descheduled only when it is executing one of the twelve "descheduling" instructions [30]), which are concerned with communication, timers, jumps, errors or concurrent process initialisation and termination.

2.6.5 The Construction of Parallel Programs

This section describes the way in which the transputer instruction set is used to implement parallel programs from processes of equal priority. Consider the processes, P,Q and R, which are to be executed in "parallel". The Occam construct for this is

```
PAR
  P
  Q
  R
```

The transputer instruction sequence to implement this is

Instructions			Comments
	LDC	3	Number of concurrent processes stored in WS location 1
	STL	1	
	LDC	(L5-L6)	Pointer to first instruction of successor process stored in WS location 0.
	LDPI		
L6:	STL	0	
	LDC	(L1-L2)	Load instruction offset and WS address of P and put it in the queue.
	LDLP	WP	
L2:	STARTP		
	LDC	(L3-L4)	Similarly for Q.
	LDLP	WQ	
L4:	STARTP		
	R		R continues from initial process
	LDLP	0	End R, pointing to successor process WS, (R - parent).
	ENDP		
L1:	P		Code for P.
	LDLP	-WP	End P, pointing to successor process WS, (R - parent).
	ENDP		
L3:	Q		Code for Q.
	LDLP	-WQ	End Q, pointing to successor process workspace.
	ENDP		
L5:			The program continues.

Table 2.2 Implementing a Parallel Program

Where WP is the offset from the workspace of R to that of P, and WQ is the offset from the workspace of R to that of Q.

There are only two `startp` instructions, as the process used to set up the concurrent processes in fact continues as process R. Hence, a PAR construct may be though of as a "parent" or "main" process which generates one or more "child" or "sub" processes.

The main process stores the number of subprocesses that it generates in its workspace, which the scheduler uses as a count down counter to determine how many subprocesses have yet to complete. Whenever a subprocess executes an `endp` instruction (relinquishing its workspace by using -WS), this counter value is decremented by one. When this value reaches zero, the main process may continue, or execute an `endp` itself.

Parallel processes of equal priority may be nested to any level, and so P,Q or R may themselves define further parallel processes. A schematic representation of the above parallel construct is represented in Fig 2.7.

2.6.6 The Construction of Prioritised Parallel Programs

Prioritised parallelism is implemented in Occam using the PRI PAR construct. This construct runs a high priority (priority 0) and a low priority (priority 1) process in parallel. The Occam representation is

```
PRI PAR
  P
  Q
```

where P is the high priority process, Q the low priority process.

The transputer instruction sequence used to implement this is

Instruction			Comments
	LDC	2	Number of parallel processes stored in WS location 1.
	STL	1	
	LDC	(L3-L4)	Pointer to first instruction of successor process (deprioritising code) stored in WS location 0.
	LDPI		
L4:	STL	0	
	LDC	(L1-L2)	Load pointer to first instruction of P...
	LDPI		
L2:	LDLP	(WP-1)	... and store it in location -1 of P's WS.
	STNL	0	
	LDLP	WP	Load pointer to WS of P, and place P on the high priority queue.
	RUNP		
	Q		Code for Q.
	LDLP	0	End Q.
	ENDP		
L1:	P		Code for P.
	LDLP	-WP	End P, pointing to its successor (Q - the parent).
	ENDP		
L3:	LDLP	0	Define a "null" process, using the present WS. Explicitly set to low priority, run it then immediately stop it (take it off the queue).
	LDC	1	
	OR		
	RUNP		
	STOPP		
L5:			The program continues.

Table 2.3 Implementing a Prioritised Parallel Program

Here, P is explicitly set to run at high priority by runp. Areg should contain the process descriptor when runp is executed. In this case, the process descriptor points to P and has an lsb equal to zero, and so P is placed in the *high* priority queue.

The PRI PAR construct is continued as process Q. The code appearing at L3 is the successor to the prioritised construct - ie this code will be executed whenever the processes inside the PRI PAR have both completed. This code runs a second version of the prioritising code, explicitly starting it at low priority. This is necessary, since the priority of the process starting at L3 would otherwise be determined by the priority of the process in the PRI PAR that finished last, and so would be indeterminate.

PRI PAR constructs may not be nested, although the two processes may

themselves contain further PAR constructs. Prioritised processes are useful whenever external communication is used. If the communicating process is run at high priority, then the link will be serviced as soon as possible. The link transfer may then take place while the cpu is executing the low priority code, making full use of the autonomous nature of the link interface. A representation of a PRI PAR construct is shown in Fig 2.8.

2.7 Communication

2.7.1 Overview

Concurrent processes communicate through channels. Communication is point to point, synchronised and unbuffered. A channel between two processes on the same processor ("soft" channel) is implemented with a word in memory, whereas a channel between two processes on different processors ("hard" channel) is implemented with a link.

Communication is carried out by first loading the stack with a pointer to the message, the channel address and the size of the message in bytes, then by executing one of the channel transfer instructions. The instruction sequence is the same for both hard and soft channels as the processor uses the channel address to determine the appropriate action to take — external channels use special reserved internal memory locations.

As communication is unbuffered, the transfer takes place only when both processes are ready. The process that becomes ready first must wait for the second process to become ready.

In order to estimate the performance of a transputer program, it is important to understand the mechanisms by which messages are passed. Hard and soft channels are implemented differently, and so they will be considered separately.

2.7.2 Internal Communication

Soft channels are implemented by using a single word in memory, which contains either a pointer to a workspace or the special value "empty". A soft channel must first be initialised to the value "empty" before it is used.

When a process wishes to use a channel, the value stored in the channel word is first checked. If the value is "empty" then the workspace pointer of the process is stored in the channel (the workspace contains the address of the message to be transferred), and the process is descheduled. When the second process becomes ready, it also checks the value of the channel word. This time, the value of the channel is not "empty", and the message is copied. The second process continues execution, the first process is rescheduled and the channel is reset. This is shown in Fig 2.9, where a process P outputs a message to a process Q over channel C.

Note that only one process is descheduled, and the actual transfer is carried out by the cpu.

2.7.3 External Communication

Hard channels are implemented through a link by using a link interface, which manages message synchronisation and transfer. The link engines are able to work concurrently with the cpu.

Whenever a transfer instruction is executed by a process, and found to be

external, the information held in the stack is transferred to the link interface registers and the process is then descheduled. The corresponding recipient process does likewise on another processor. When both link interfaces have been initialised, the transfer takes place. Both processes are rescheduled after the transfer has been completed. This is shown in Fig 2.10.

Note that both processes are descheduled, and that communication is overlapped with computation by an amount dependent on the size of the message. Because the overheads associated with setting up the link transfer are independent of the message size, it is more efficient to transfer larger rather than smaller messages.

2.8 Memory Map

The transputer uses a byte orientated addressing scheme, in that an address word points to a byte in memory, not to a word. Each address word may be decomposed into two portions — a word address and a byte selector. For 32-bit processors, the byte selector occupies the two least significant bits of the address word.

A signed address space is used, with the bottom of the address space being represented by the most negative number (`#80000000`). The total addressable space for a 32 bit processor is 4Gbyte. Internal memory extends from `#80000000` to `#80000FFF` (for a 32 bit processor). The locations up to `#8000006F` are used as an extended register set by the processor, to store information concerning links, events, timers and interrupted processes.

Although the transputer uses a byte orientated addressing scheme, Occam uses a word orientated scheme, Fig 2.11. The byte selector is not brought out on the external memory interface (EMI). Individual bytes of external memory are accessed

using the byte strobes of the EMI.

The EMI also provides facilities for DMA of the external memory space, and for external wait state generation.

2.9 Event Pins

The event pin, and its associated event acknowledge and event request pins, allows for an asynchronous handshaking interface between the transputer and an external device. These pins allow an external event to interrupt an Occam program.

2.10 Booting

The transputer may be booted either from a link or from an external ROM. Link booting is used exclusively for the work presented in this thesis.

2.11 Optimising Performance

This section presents a brief overview of the various techniques available to optimise the performance of a transputer system. The reader is referred to [65] for a more complete treatment.

The section is divided into two parts. The first deals with optimising performance on a single transputer, the second discusses how performance may be increased in a multi-transputer system.

2.11.1 Uni-Processor Optimisation

Performance optimisation on a single transputer is compiler dependent. In this case, the compiler was the D700D version of Occam (Occam2). As internal memory may

be accessed at least twice as quickly as external memory, most of the methods presented here are concerned with making as much use of internal memory as possible.

The Occam compiler assigns the process workspaces to the lowest area of memory, then the program code and finally a section optionally reserved for vectors. Hence, the program space will be forced off chip in preference to the workspaces. This is sensible, as the transputer is able to load in four instructions, but only one data item, in a single cycle, and so the additional external memory access time makes less impact on the program space than the data space.

The compiler allocates workspaces for procedures and parallel processes as a falling stack, ie the last procedure/process to be declared has its workspace placed at the lowest location. Similarly, for each process, the variables are allocated as a falling stack within the workspace. So, if a process uses time critical data, then it should be declared last, and the data within the process should also be declared last. This keeps the critical variable within internal memory space, and keeps its access time as low as possible. The exception to this is large vectors, which may force other areas off chip.

Variables should be declared locally to a process whenever possible, as this allows the use of local load and store instructions, which are more efficient than their non-local counterparts.

Abbreviations may be used to bring non-local variables into local scope (The scope of a process refers to those variables which may be accessed locally). In particular, sections of non-local vectors may be abbreviated by sub-vectors using constant index terms, which speeds up vector access.

Vectors should not be assigned using a loop, but by the block move facility, which is far more efficient.

Whenever vectors are used inside a replicated SEQ loop, it is always advisable to explicitly access a number of consecutive vector elements inside the loop. This is known as "opening out" the loop, and reduces the overall overhead associated with performing the loop.

For time critical sections of code, then the GUY construct may be used. This feature allows the programmer to incorporate sections of transputer assembly code into an Occam program. Care must be taken with this option, however, as only a limited compile time checking facility is available.

Finally, certain compiler options should be turned off once the program has been tested and verified. An example is range checking, which inserts extra run time code in order to test for subscript overflows, which obviously decreases performance. Once the program has been tested, however, there is no need for this code, and the program may be re-compiled without this option.

2.11.2 Multi-Processor Optimisation

Optimising code to run on a multi-transputer system essentially involves optimising the operation of external communication. Multi-processor optimisation is much more sensitive to the particular application than uni-processor optimisation, and is not so well defined.

Link performance must be optimised. Communication on the links must be allowed to overlap with cpu operation, and the overhead per word of transferred data must be reduced as much as possible.

In order to allow this cpu/link overlap, link communication and cpu operation must be decoupled. This involves placing all link communication statements in one process (which may itself contain parallel sub-processes), all the computation statements in another, and running them in parallel. Whenever an external communication statement is executed, then that process is descheduled, leaving the computation process to continue while information is being transferred on the link.

If the two processes have the same priority, then the communication process may have to wait, for *at least* a timeslice period, for the computation process to be interrupted before it can transfer data. This delay may cause the computation process to be starved of data, or further communication delays on other transputers, both of which will degrade system performance. The solution is to run the communication process at high priority, which then allows data to be transferred with the minimum of delay.

In order to remove any soft channel communication associated with buffering data between the communication and computation process, Inmos [65] recommend the use of a looped three stage pipeline. Each pipeline element has the same structure — a `PRI PAR` with parallel external input and output processes at high priority, and a computation process at low priority. There are no inter-process soft channels, as data is passed by reference within the pipeline. This is indeed an efficient structure, but is not always the optimal solution, as the overheads associated with setting up each `PRI PAR` construct are quite considerable.

The overheads associated with transferring data may be reduced by transferring vectors rather than words. This spreads the overheads associated with the setup over many more words. However, if the message is too long, then the transfer time may

impede performance.

2.12 Summary

This chapter has introduced the transputer as a powerful element from which large multi-processor systems may be constructed. The major points of the architecture have been outlined, in particular the link engines which allow the transputer to overlap communication and computation. The nature of the instruction set has been described, together with the performance implications of the non-standard method of constructing instructions. The structure of both the sequential and the parallel Occam structures have been discussed, and their implementation shown to be particularly efficient due to the run-time scheduler. The inter-process communication mechanisms for hard and soft channels have been shown to be similar. Finally, a treatment of performance optimisation techniques has been given.

Due to its RISC-like design, internal memory and concurrent communication and computation capability, the transputer is a powerful processor. Its run time microcoded scheduler and uniform communication instructions, in conjunction with the links, allow multi-transputer networks of arbitrary size to be easily implemented.

Although the transputer is indeed a powerful general purpose processor, the constraints imposed by real-time signal processing applications often force it to operate at its performance limit. These constraints, and their performance implications, are covered in the following chapters.

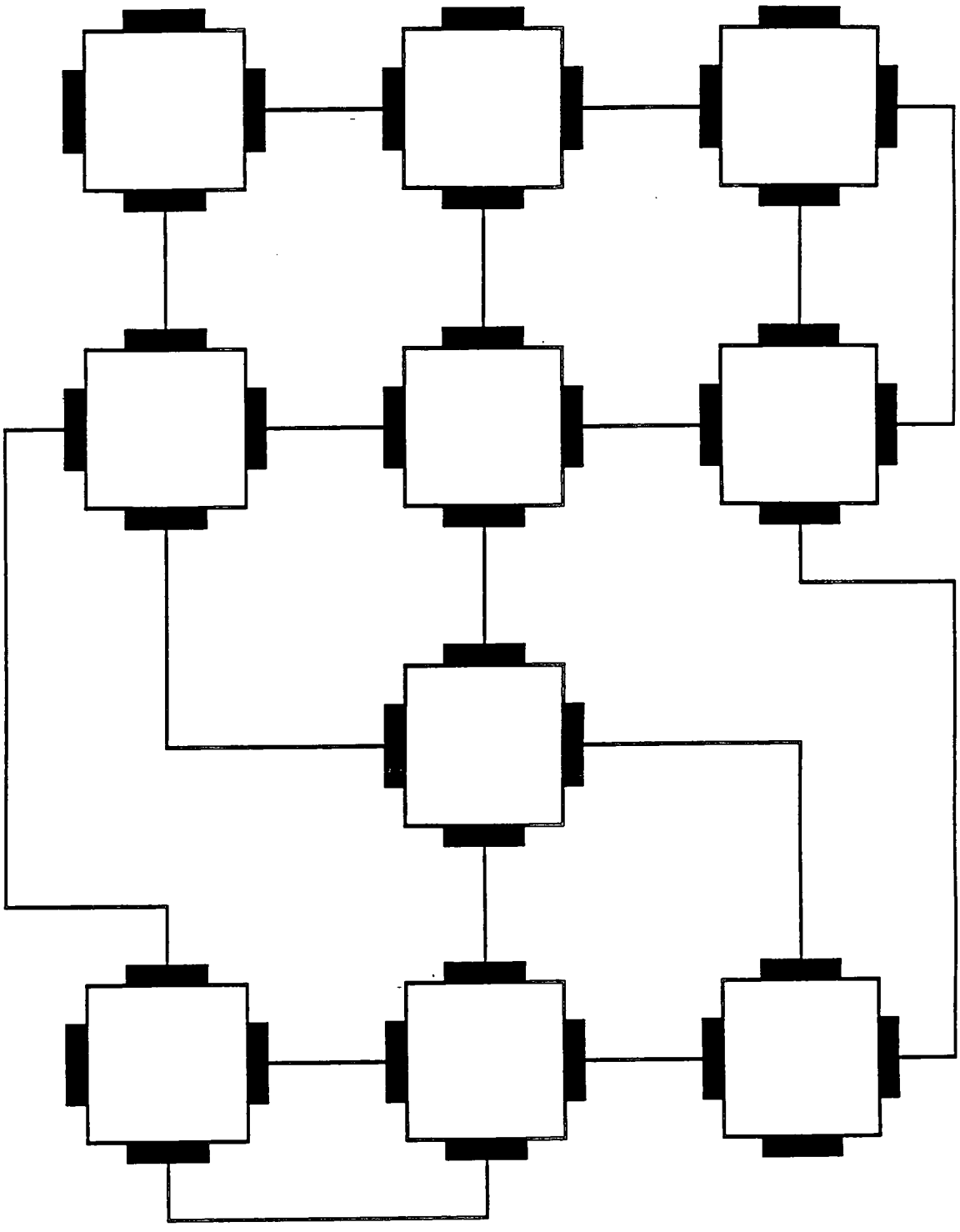


Fig 2.1 An Arbitrary Transputer System Topology

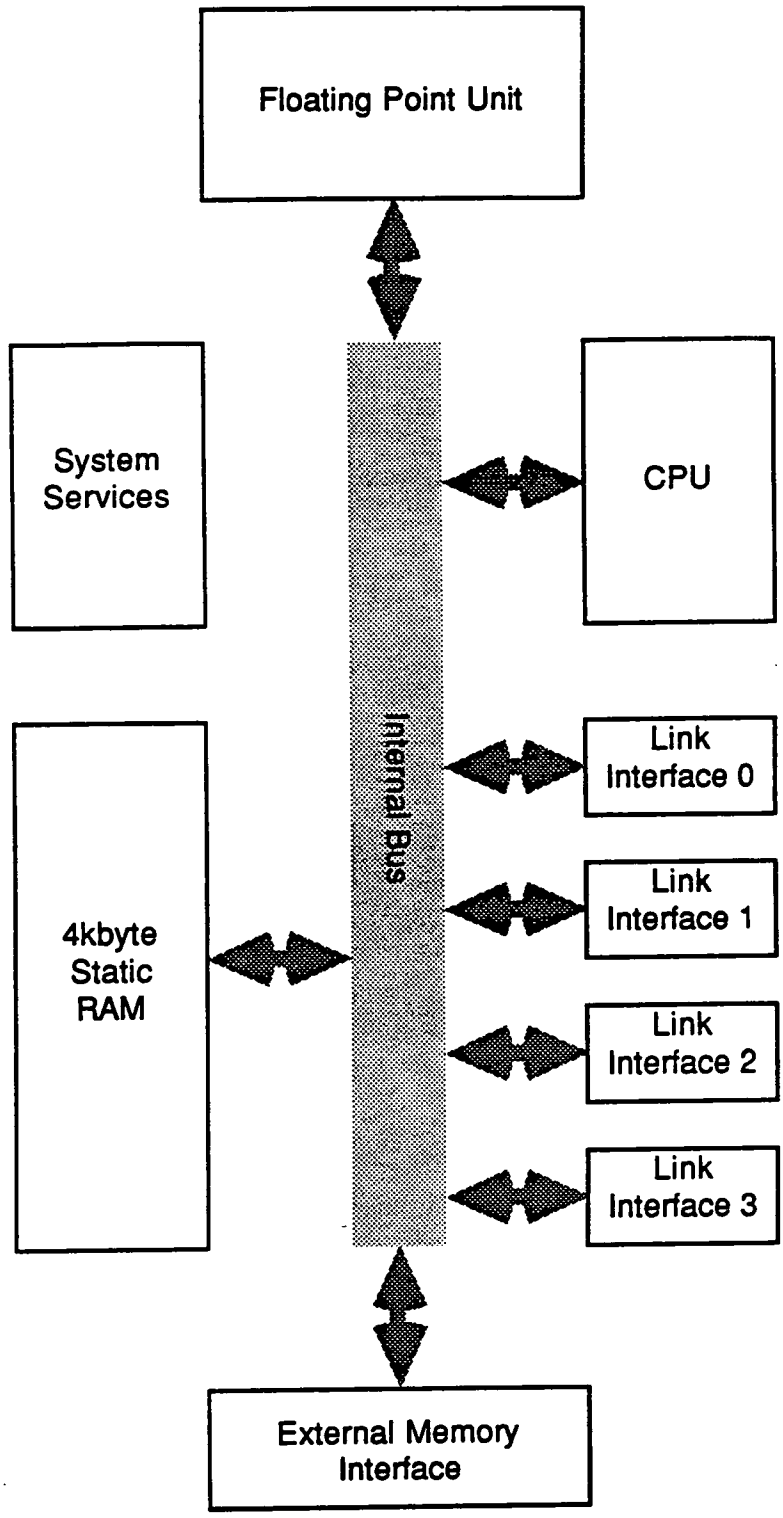
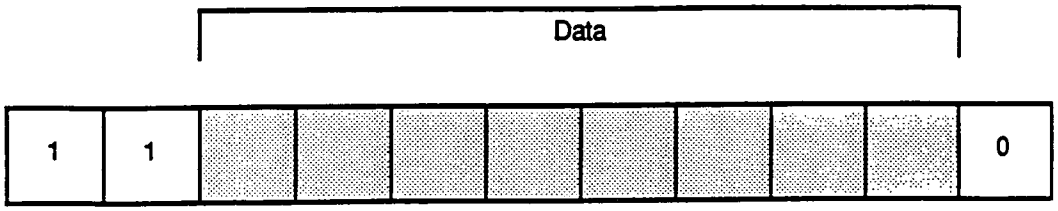
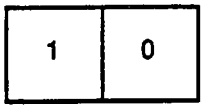


Fig 2.2 Schematic of Transputer Architecture



Data Packet



Acknowledge

Fig 2.3 Link Data and Acknowledge Formats

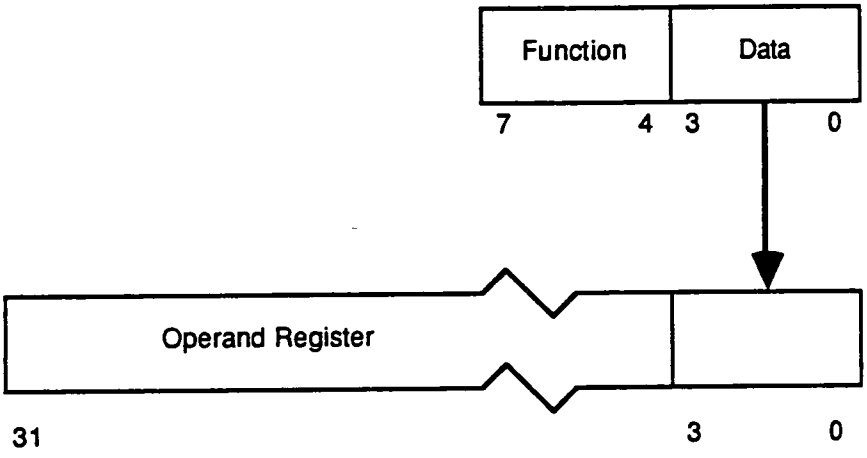


Fig 2.4 Instruction Format

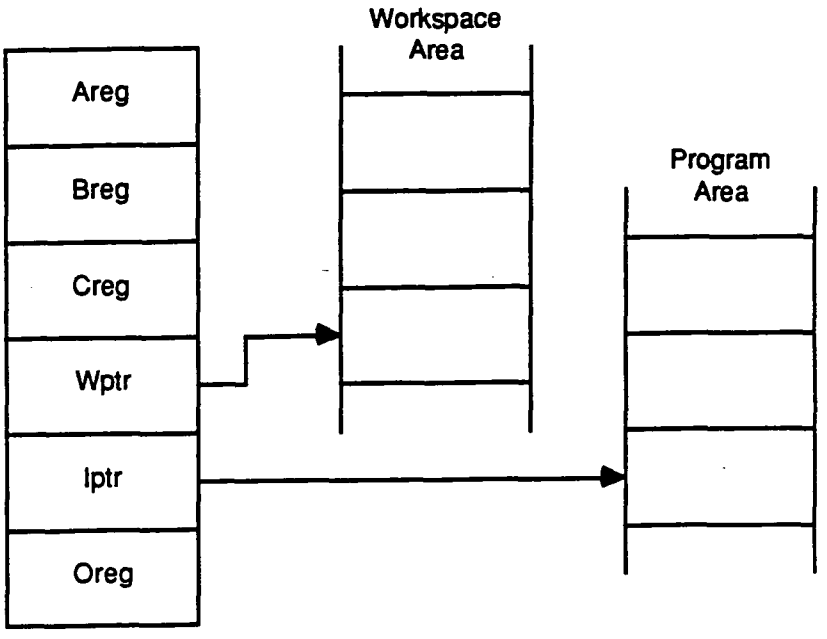


Fig 2.5 Implementing A Sequential Process on the Transputer

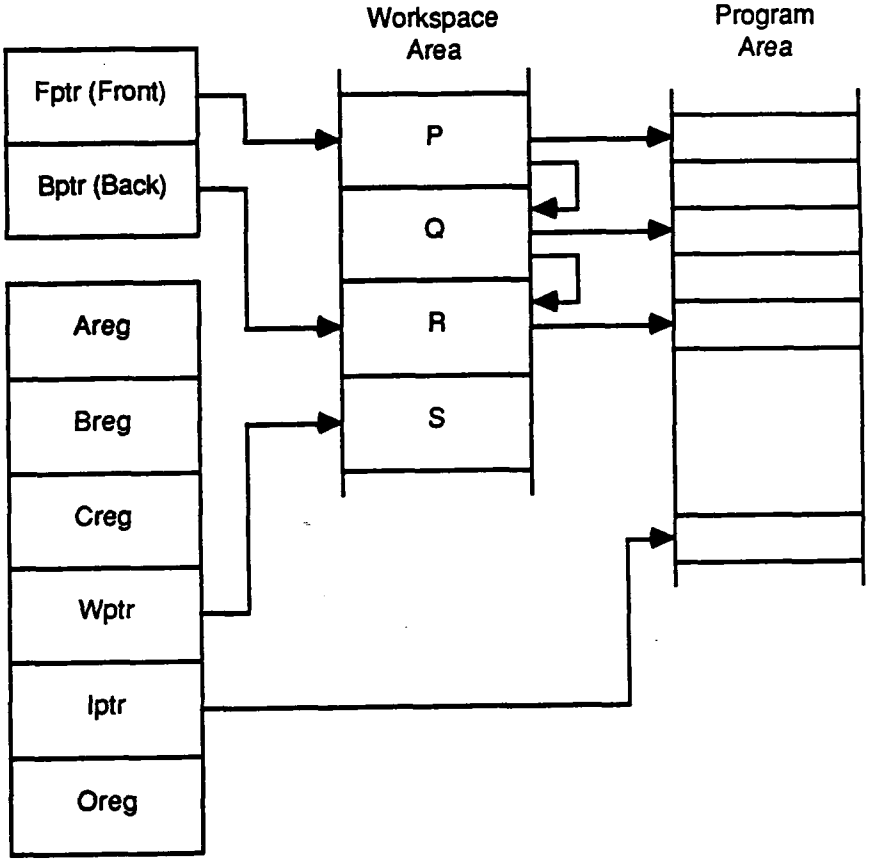


Fig 2.6 Implementing Concurrent Processes on the Transputer

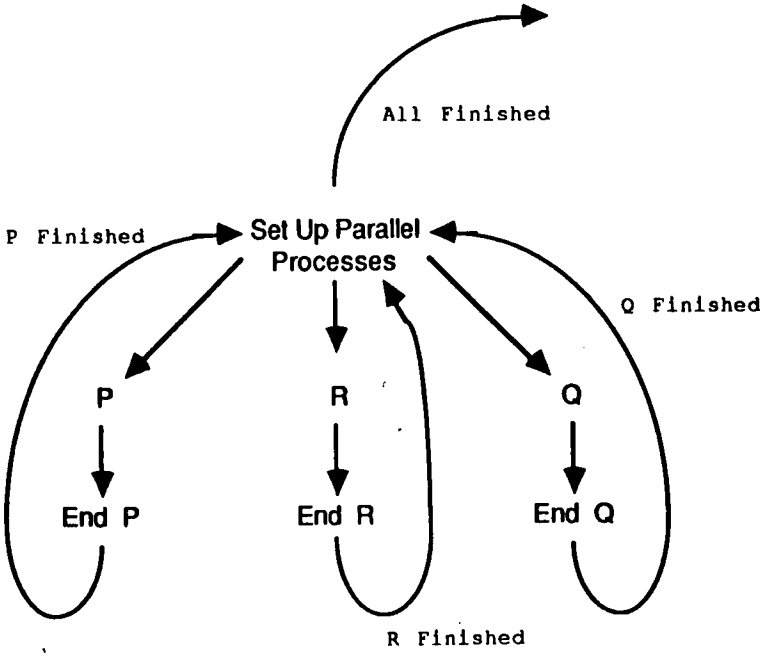


Fig 2.7 Construction of a Non-Prioritised Program

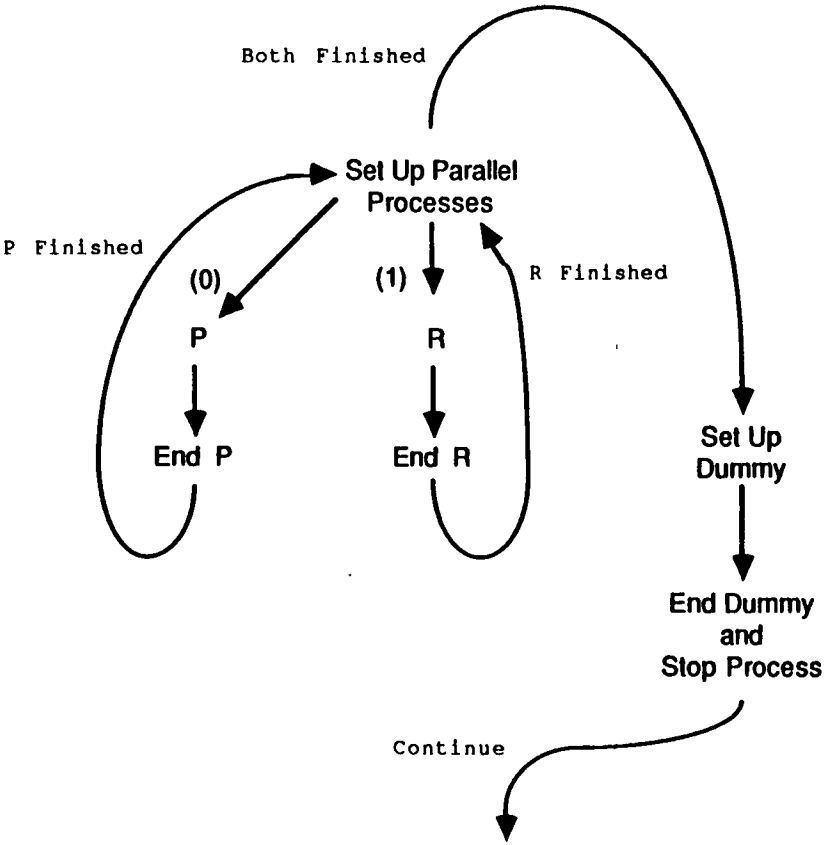
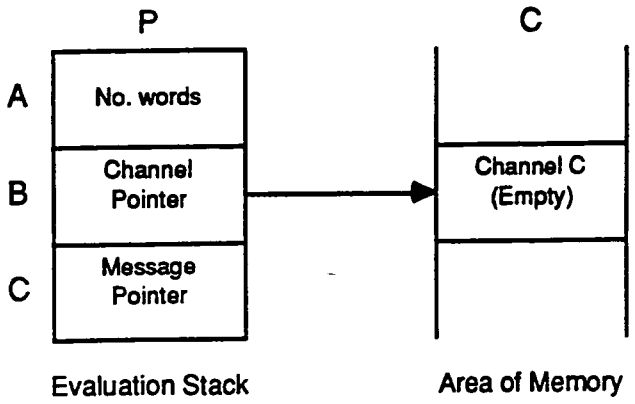
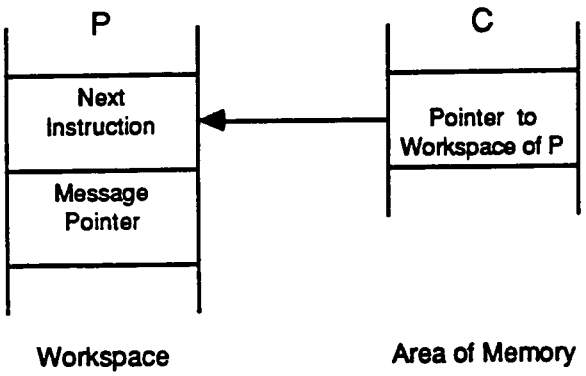


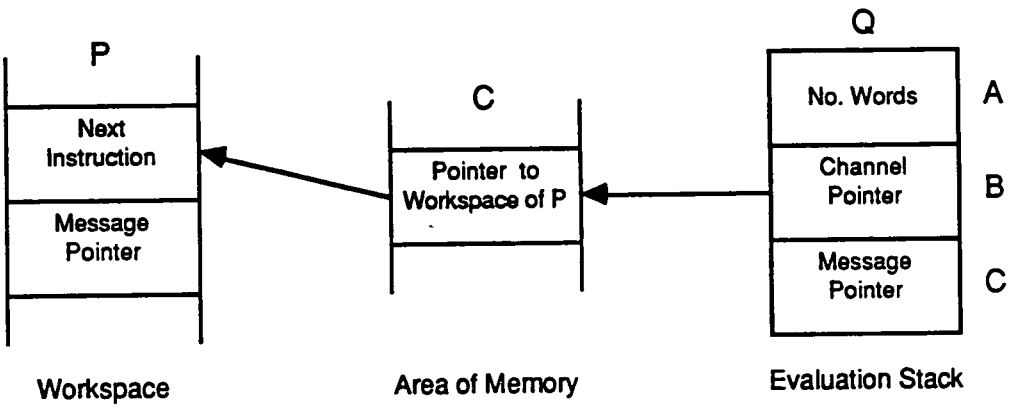
Fig 2.8 Construction of a Prioritised Program



Process P checks channel C, finds that it is empty, executes an output instruction and is descheduled.

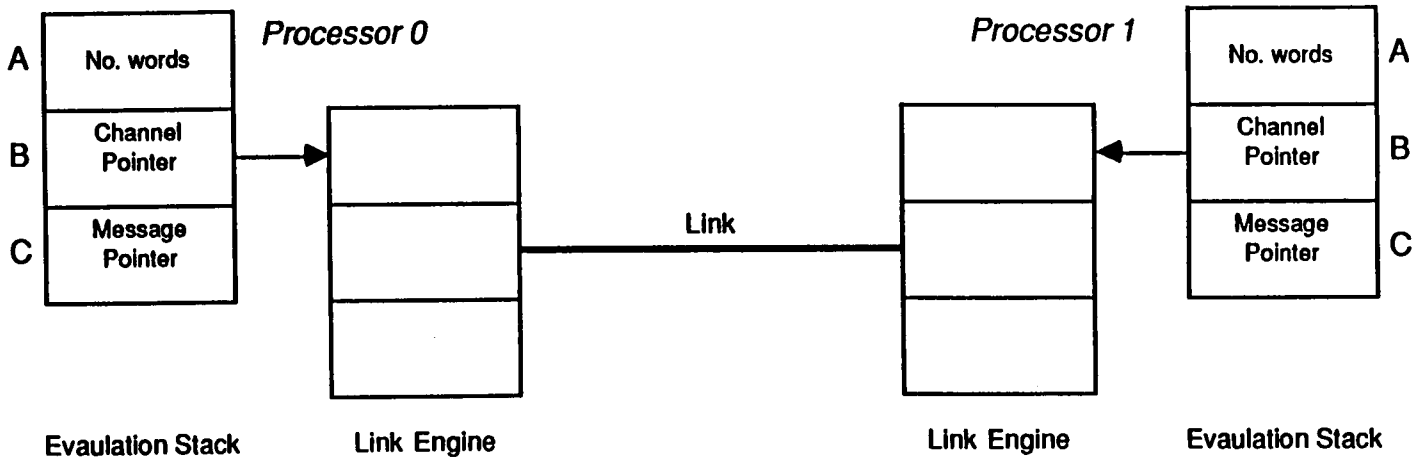


Channel C now contains a pointer to the workspace of P, which itself contains a pointer to the message.

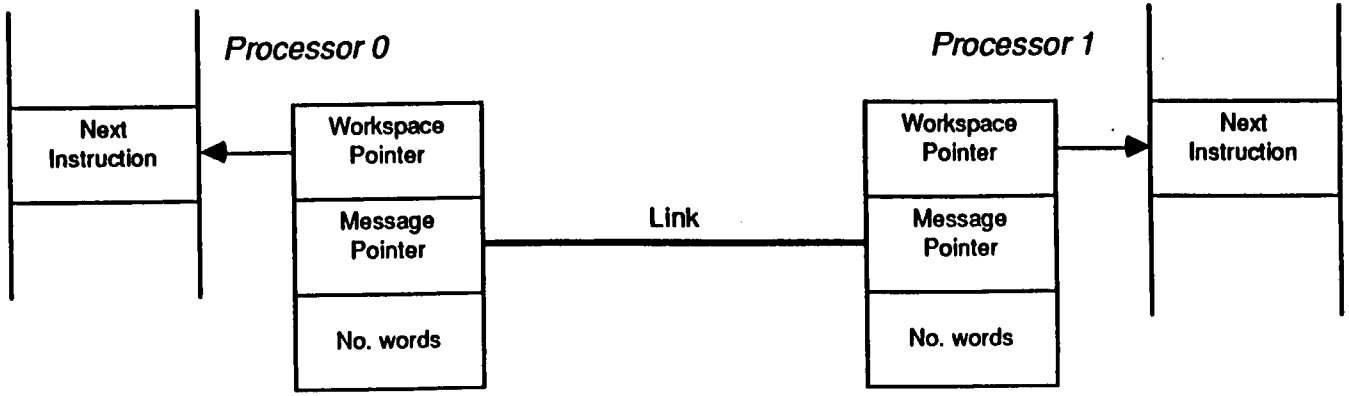


Process Q executes an input on channel C and finds that it has been initialised. The transfer takes place (memory to memory block copy), C is reset and P rescheduled.

Fig 2.9 Internal Communication



Upon executing an external input or output instruction, the link engines are initialised and the calling process descheduled



The link engine registers contain workspace pointers to their respective processes, a pointer to the message to be transferred and its size. Transfer takes place when both link engines have been initialised, after which both processes are rescheduled.

Fig 2.10 External Communication

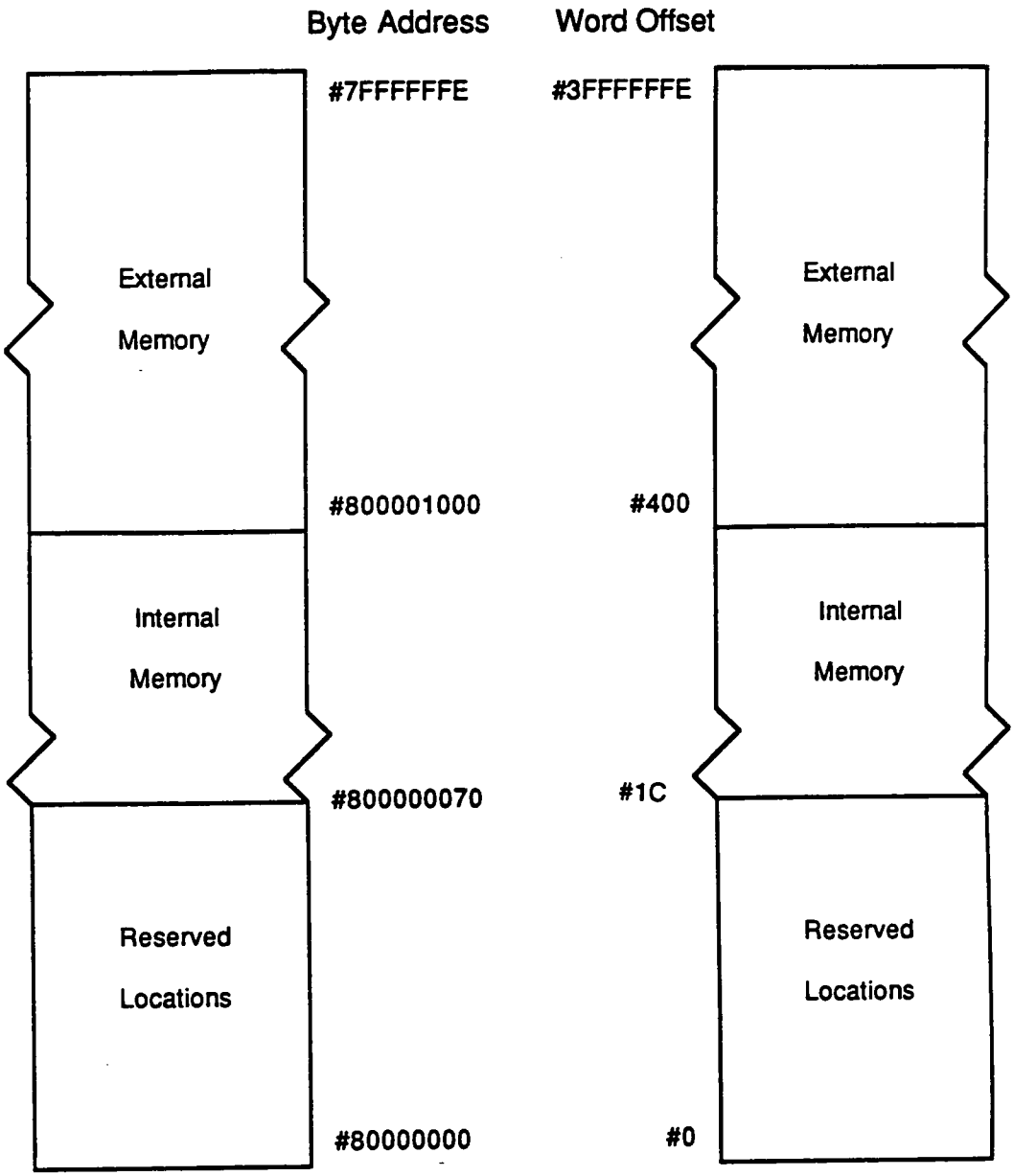


Fig 2.11 Comparison of Transputer and Occam Memory Map

Chapter 3

Digital Filtering on the Transputer

3.1 Introduction

With its concurrent communication and computation capability, and its relatively high clock rate, the transputer has potential for use in high performance signal processing applications [66]. This chapter investigates the implementation of one such application — a digital filter — on the transputer.

The code was mapped onto one, two and three transputers in order to investigate the impact of concurrency on performance. Furthermore, two intra-processor communication structures (or *harnesses*) were utilised for each mapping, and their effect on performance investigated.

The computation code running on the transputers is shown to be relatively short. Because of this, the performance of each implementation is sensitive to any unnecessary overheads. The full optimisation of the code is described.

The application requires that many data streams (or *channels*) are processed. It is shown that once a fully optimised single channel filter is implemented, it is a straightforward step to modify the code in order to produce a multichannel filter.

The working environment for this application is such that power consumption

and occupied space are at a premium. The implementation of the filter on the transputer is assessed, then, in terms of the dual criteria of overall performance (total throughput) and the performance per processor (total throughput per processor, or per unit silicon area).

3.2 The Filter

The filter possesses a three pole bandpass Butterworth response, the characteristics of which are given in Appendix A. The filter comprises single pole high and low pass sections, connected as in Fig 3.1. The single pole sections are constructed as shown in Fig 3.2.

The structure of the single pole sections is notable in that it does not include a multiplier, the multiplication (division) function is effectively carried out by a shift operation. This renders the filter suitable for implementation on processors, such as the transputer, that do not possess a fast multiplier. The structure of this filter, then, departs from the more usual filter architectures that use fast (fractional) multipliers [37].

3.3 Implementation on the Transputer

The implementation of the filter on the transputer may be divided into two areas. The first deals with the mapping of the processes onto the processors, and the second deals with the structure of the processes themselves. These will now be considered in turn.

3.3.1 Mapping the Processes onto the Processors

From Fig 3.1, the overall structure of the filter may be mapped into three sections. This partitioning provides a natural mapping onto one, two and three processors, Fig 3.3a to Fig 3.3c.

Other partitioning schemes are possible, of course, but these would involve decomposing the single pole computation sections. The computation times of these sections are very small already, the total execution time being dominated by the communication bandwidth, and so further partitioning would provide little, if any, performance increase. Furthermore, this would increase the number of processors, which would in turn increase the power and space requirement of the system.

3.3.2 The Structure of the Processes

The structure of the processes consists of a computation section, running at low priority, embedded in a larger structure, termed the *harness*, that defines the communication structure of the process and includes the external communication statements, which run at high priority.

Two harness structures, Type I and Type II were implemented. The same computation section was used in both harnesses, for a given mapping. Minor modifications were made to the harnesses and the computation section for the different mappings. The harnesses and the computation section are considered separately, below.

3.3.2.1 The Harnesses

The structure of the two harnesses is shown in Fig 3.4 and Fig 3.5, together with their

"hedgehog" diagrams. Type I uses a pair of PRI PAR statements inside a WHILE TRUE loop. Inside each PRI PAR, the communications are held at high priority, and the computation at low priority. There is no communication between the communication and computation processes within the PRI PAR. While the communication processes are dealing with data set A, the computation process is dealing with data set B, and vice versa in the next PRI PAR statement. Hence communication and computation are decoupled, and the data sets are passed "by reference" between the two PRI PAR statements, which is the approach recommended in [65]. This structure is inherently multi-channel, and may be extended to an arbitrary number of channels simply by adding more PRI PAR statements.

However, the PRI PAR statement does take many cycles to set up, depending on the number of parallel processes it contains (Section 4.5.2) — roughly 65 cycles for these applications. No "useful" work can be carried out during this set up period, and as this happens for every invocation of the PRI PAR, then it could represent a considerable overhead.

This overhead is eliminated in type II by implementing a single PRI PAR, within which are placed WHILE TRUE loops for the communication and computation. The communication loops are configured in parallel in order to maximise the external communication overlap. A consequence of this structure, however, is that the communications and computation processes must communicate through internal channels (ie they are coupled). Internal communication is carried out by the cpu and so detracts from the overall performance of the process.

Thus both harness types possess their own peculiar overheads; for instance, type I requires roughly twice as much memory as type II. The relative merits of each

are discussed in Section 4.5.2.

The harnesses for the processes used in the one and two processor partitionings are similar, as these use the same single output, single input structure. The three processor partitioning, however, requires a third link between the second and third processors. The requirement for a third external communications channel adds to the overheads experienced by the three processor mapping.

3.3.2.2 The Computation Section

The filter is constructed from a combination of single pole high and low pass sections, Fig 3.1, which possess a similar architecture, differing only in the point at which the output is taken, Fig 3.2.

The single pole structure contains a shift right (by fifteen places), effectively a division by 2^{15}. Multiplication and division on the transputer are expensive. An integer multiply requires 39 cycles to complete, an integer division 40 cycles (including prefix overheads), a floating point multiply 11 cycles and a floating point division 17 cycles (not including setup, but carried out concurrently with cpu operation — for floating point transputers only). This structure requires a division operation, which is expensive.

A more efficient method is to directly use a shift right instruction. The transputer is able to right shift a single length integer in 15 + 2 + 1 (18) cycles, and a double length integer in 15 + 1 (16) cycles. There is a complication here, however, in that the shifts are not *arithmetic*, but *logical*, and so the leading vacated bit positions of the word are zero filled. The transputer operates with a two's complement data format, and so for a negative number not only is polarity lost upon shifting, but

the value of the variable is distorted. It is thus very important that the post shifted value is sign extended, which unfortunately requires extra instructions.

3.3.2.3 The Occam2 Version

The most straightforward method of coding the filter is to use a high level language, in this case Occam2. The computation code for the high pass and low pass sections, together with the disassembled form of the highpass section, is given in Fig 3.6, the code for all the computation sections is given in Appendix B. Sign extension is catered for by an IF statement here. The IF is placed after the shift, and tests to see whether the pre-shifted sign bit was set, and if so sets the msbs with a bit-wise OR instruction, thus restoring arithmetic validity. The code uses 30 bytes of program memory space and executes in 48 cycles for a positive pre-shifted value, and 56 cycles for a negative pre-shifted value, the differing times being a consequence of the conditional branch. This is a hard real-time application, however, and so worst case times must always be assumed. Hence, the execution time of this piece of code must be given as 56 cycles.

The section of code used for sign extension uses 12 cycles and 9 bytes of program memory. More efficient routines, in terms of execution speed and memory requirement, may be implemented by directly using transputer instructions.

One such method makes use of the XWORD instruction, which sign extends a part-word value to a single length value, [65]. The code for this method of sign extension is shown in Fig 3.7. This section of code is also placed after the shift operation, but is unconditional in its operation. It uses 6 bytes of program memory and takes 12 cycles to complete. This method, then, is only marginally preferable to the

Occam2 version, as it uses less memory space but completes in the same number of cycles.

A more efficient method uses the XDBLE and LSHR instructions, [65]. The XDBLE instruction converts a single length value held in Areg into a double length value in Areg and Breg. The LSHR instruction logically right shifts a double length value. The code to implement a sign extended shift using this method is given in Fig 3.8. The action here is that the sign extension bits, held in the most significant word, are shifted into the msbs of the least significant word (the actual data word). Thus, the sign bits are preserved and the value held in Areg is arithmetically shifted.

This code operates on the pre-shifted value, and incorporates the right shift operation. As the double length shift is executed in two cycles less than the single length shift, this routine effectively adds a sign extension overhead of a single cycle, compared to the other versions, making it by far the most suitable method. Furthermore, this method is not at all affected by data prefixing.

3.3.2.4 The Assembly Version

The Occam2 compiler does not allow data to be passed from one statement (line of code) to another through the stack. This is essential if the code is to be secure, but does not optimise performance as additional STL and LDL instructions must be used. Information may be passed through the stack, providing a higher performance, if statements are compounded onto a single line.

For example, as shown in Fig 3.9, the code assigning `c:=a+b` then `e:=c-d` is compiled down to a sequence lasting twelve cycles, as the variable `c` is stored at the end of the first assignment and loaded again at the beginning of the second

assignment. However the compounded code of `e:=(a+b)-d` is compiled down to a sequence lasting only nine cycles, as the variable `c` is kept on the stack.

This is also an example of what may be achieved by hand coding critical sections in transputer assembly language. Although the Occam2 compiler is highly optimised, often the number of instructions may be cut down to the bare minimum, and optimum use made of the stack, only by fine tuning sections of code by hand. This is what has been done in the fully assembly language versions of the computation code. One `STL` and one `LDL` may be eliminated from the high pass section, saving 3 (5, with prefix) cycles. Two `STL` and `LDL` instructions may be eliminated from the lowpass section, saving 6 (10) cycles. Of course, all the optimisation methods outlined in Section 2.11.1 were used for the code.

3.3.2.5 Compounding Filter Sections

From Fig 3.3a and Fig 3.3b, the one and two processor mappings require that a combination of single pole sections be placed on a single processor. The single pole sections, written in transputer assembly, are combined sequentially, with any supplementary code, such as addition, being inserted where required. The stack is used to pass values between sections.

It would be possible to configure the processes in parallel, passing data either through internal channels or by reference. The code sections are very small, however, and so their throughput would be greatly affected by the overheads incurred by setting up both the parallelism and the internal communication. Maximum performance on a single processor is attained by running a sequential computation process in parallel with a communications process.

3.3.2.6 The Use of Vectors

It has already been mentioned that efficient inter-processor communication can only be carried out by passing vectors, rather than single words, of data. It follows, then, that for a given optimised computation section, maximum performance will only be attained if data is passed in vectors. This approach has also been used in non-transputer based digital signal processing systems [62].

Consider a data vector as in Fig 3.10. For every element of the input vector, there exists both a corresponding element in the output vector and a logical computation section. Now, if the output of the *ith* element is allowed to form the input of the *i+1th* element (by using non-vectored variables to pass data between sections — "internal" variables), then any given value of an element in the output vector depends upon the previous elements in both the present input and output vectors. As the data elements are processed in a logically sequential and dependent manner, the input vector may be considered to contain a number of samples from the same data source — the filter is processing a single channel of data.

Consider, now, Fig 3.11. If the internal variables of each filter section are vectorised and are not passed from one section to another, then any given value of an element in the output vector depends upon the values of the same element in the preceding input and output vectors. The data is processed in a logically parallel and independent (orthogonal) manner. The input vector may be thought to contain a single sample from a number of data sources — the filter is processing a number of channels of data.

For a given vectorised structure, then, the multiplicity of data channels being processed is determined by the amount by which individual computation sections share

their internal variables. This method may be developed to provide a combination of logically sequential and parallel data channels, Fig 3.12.

For a given vector size, the overall sample rate is constant. If, for instance, the overall sample rate is 80kHz, and a four element vector is used, then either a single filter with 80kHz sampling frequency, or two filters with 40kHz sampling frequency, or four filters with 20kHz sampling frequency, or any combination, may be implemented. Thus, not only multi-channel, but also multi-rate filters may be implemented using this method, if suitable input/output buffering is utilised, Fig 3.13.

3.3.2.7 Structuring the Computation Code

For the single channel case, the most obvious way of structuring the code is to embed the computation section inside a replicated SEQ. An example section of code is shown in Fig 3.14, in which the input and output data are defined as vectors, whereas the internal variables are defined as simple variables.

There are two drawbacks with this structure, however. Firstly, there is an overhead of approximately 1μs (20 cycles) attached to the looping operation [30], [65]. The length of the computation code ranges from about 40 cycles to about 120 cycles in this application, and so this overhead is far from negligible.

Secondly, the elements of the input and output vectors are not accessed directly, their addresses are calculated in runtime by use of the WSUB instruction. The instruction sequence required to access an element is `LDL i, LDL input.base, WSUB, LDNL` which introduces an additional overhead of at least six cycles per element [65].

The overhead attached to the looping operation may be reduced by "opening out" the loop, effectively increasing the amount of computation carried out in each

pass of the loop. The overhead concerned with accessing the vector elements may be reduced by opening out the loop in sections of sixteen and using abbreviations to directly access the elements, with zero prefixing, [66]. However, opening out the loop in this way requires additional run-time calculations, and the looping overhead is never totally eliminated. This type of loop optimisation is suitable only for larger computation sections, or for many more iterations than are required here.

As the memory space required by a computation section is small, and the number of loop iterations is relatively low, then it becomes feasible to dispense with the loop structure altogether and explicitly define each computation section, which eliminates the loop overhead. The overheads associated with addressing the vector elements are also removed, as they may now be explicitly addressed as local variables. An opened out version of Fig 3.14 is given in Fig 3.15. The memory requirement for this structure is obviously greater than that of the replicated SEQ structure, but as the code sections are small, large vectors may still be used before the effects of external memory access are seen (the actual threshold vector size depends on the size of the computation section and which harness is being used).

The most obvious way of dealing with the multi-channel case is to use a replicated PAR structure, Fig 3.16. In addition to the overheads associated with looping and non-local indirect element access in the replicated SEQ structure, there is also an additional overhead caused by setting up a number of parallel processes on each iteration. This overhead is proportional to the number of parallel processes and is in any case considerable. Furthermore, this structure does not allow different length computation sections, and so multi-rate filters may not be implemented. There is no performance gain in implementing parallel processes on a single transputer. The code

may equally be structured as a replicated SEQ, which would reduce the overheads related to parallelism. But it has already been shown that the most efficient way, in this case, of structuring the sequential code is to fully open the loop. The best solution for the multi-channel case, then, as for the single channel case, is to use a linear code structure. In order to maintain orthogonality, the internal variables must be defined as vectors. This also allows computation sections of different lengths to be implemented.

3.3.3 Measuring Performance

The filter was implemented and tested using an in-house transputer system with an Inmos B004 board acting as host. The in-house system comprises 3U boards containing an IMST800C-20 transputer and 128kbytes of zero wait state static RAM, interconnected via cables connected to the front of the boards [67].

The filter configuration was placed between a "source" transputer, which supplied data to the filter, and a "receiver" transputer, which acted as a data sink and a stopwatch. The "host" transputer, running on the B004, collected the timing data from the stopwatch, post processed it and displayed the results.

The source process outputted vectors to the filter in batches of a thousand. The stopwatch process measured the time taken for the filter to output a thousand vectors and output the elapsed time to the host. The host collected the elapsed time in batches of a hundred and calculated the mean time.

Both the source and the stopwatch processes, Figs 3.15 and 3.16, utilised all the performance maximisation techniques mentioned in Section 2.11.1. In particular, vector output and input was accomplished by using linear code — a thousand transfers in a row, which eliminated the possibility of mistimings due to excessive transfer set

up overheads.

Results were taken for all three processor mappings, using both harnesses, for a number of vector lengths using the configuration shown in Fig 3.19. The results are presented in the next chapter.

The filter response could not be measured directly, as ADC and DAC systems were not available. Instead, input files were created with Hypersignal Workstation® and fed to the filter through the host filing system. The result files produced were also analysed using Hypersignal. The response of this filter is very severe, making the use of a multiple frequency test signal (noise) impracticable as the number of points required to generate a useful FFT is prohibitive. Individual sinusoids were produced, and their processed amplitudes and phases analysed in order to build up a picture of the filter's response.

3.4 Summary

This chapter has outlined the implementation of a multi-channel digital filter on the transputer. The particular constraints imposed on multi-processor systems have been addressed by this implementation. The structure of the filter has been given, and its partitioning onto one, two and three transputers described. Two program structures, or harnesses, have been implemented, and their relative merits discussed.

Although ADC and DAC hardware was not available, the performance of the transputer system and the response of the filter were tested in software.

This chapter has provided a means of investigating the optimum method of implementing small scale digital signal processing algorithms on multi-transputer networks. The results produced, and their analysis, provide an insight into the

applicability of the transputer to DSP applications, and are discussed fully in the following chapter.

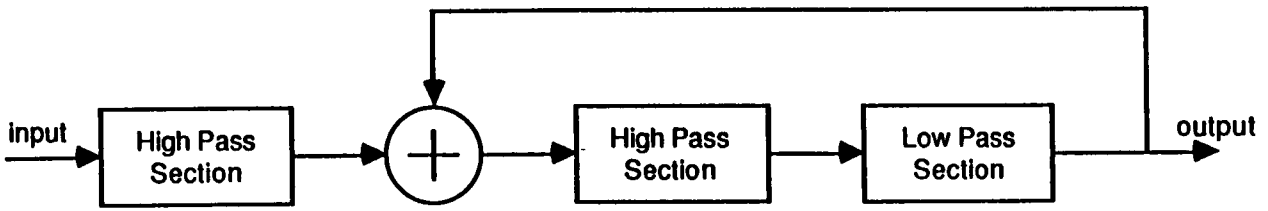


Fig 3.1 Block Diagram of Filter Structure

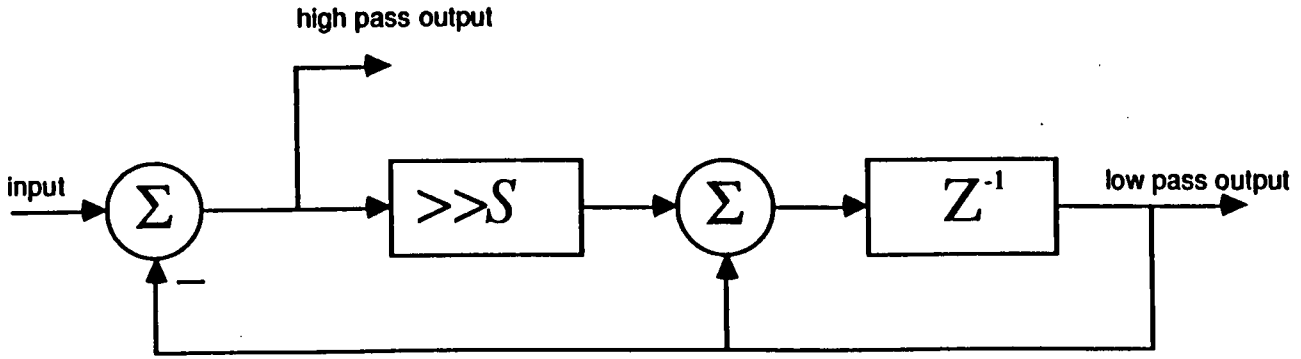


Fig 3.2 Schematic of a Single Pole Section

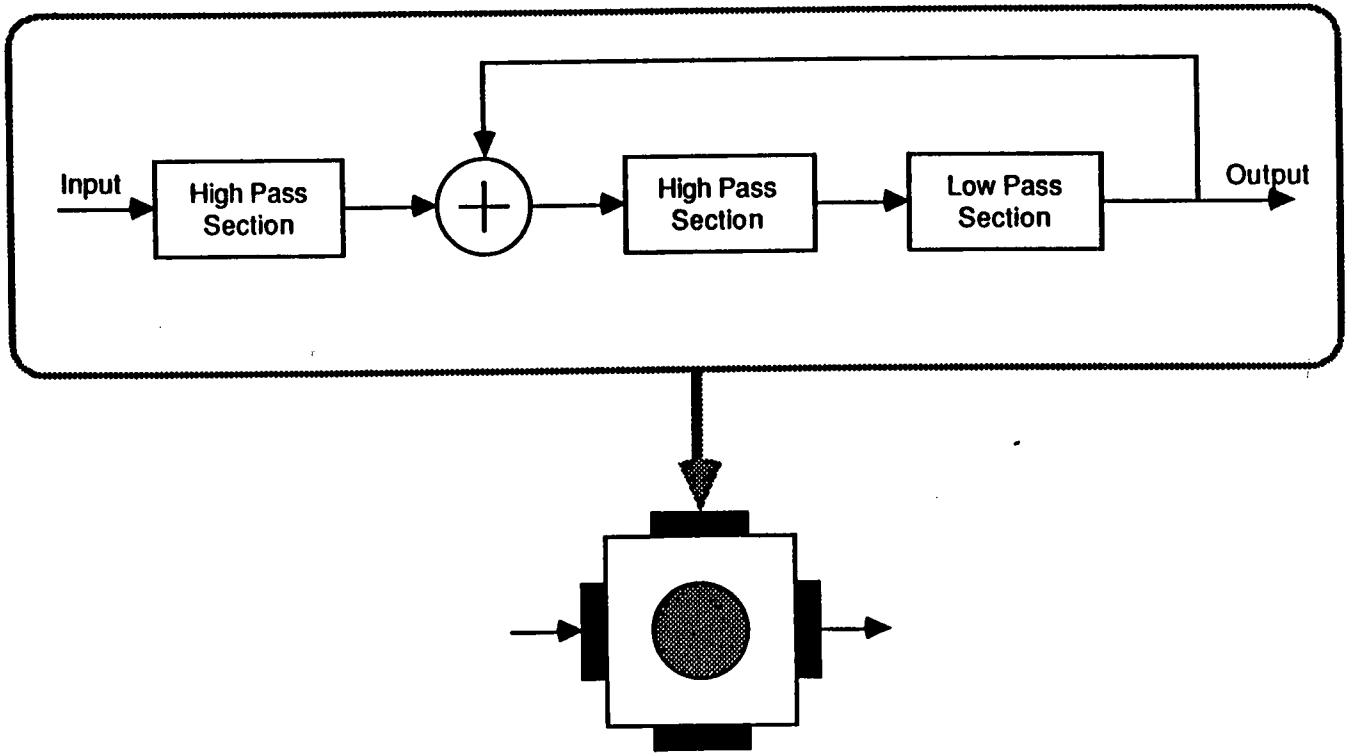


Fig 3.3a The Single Processor Mapping

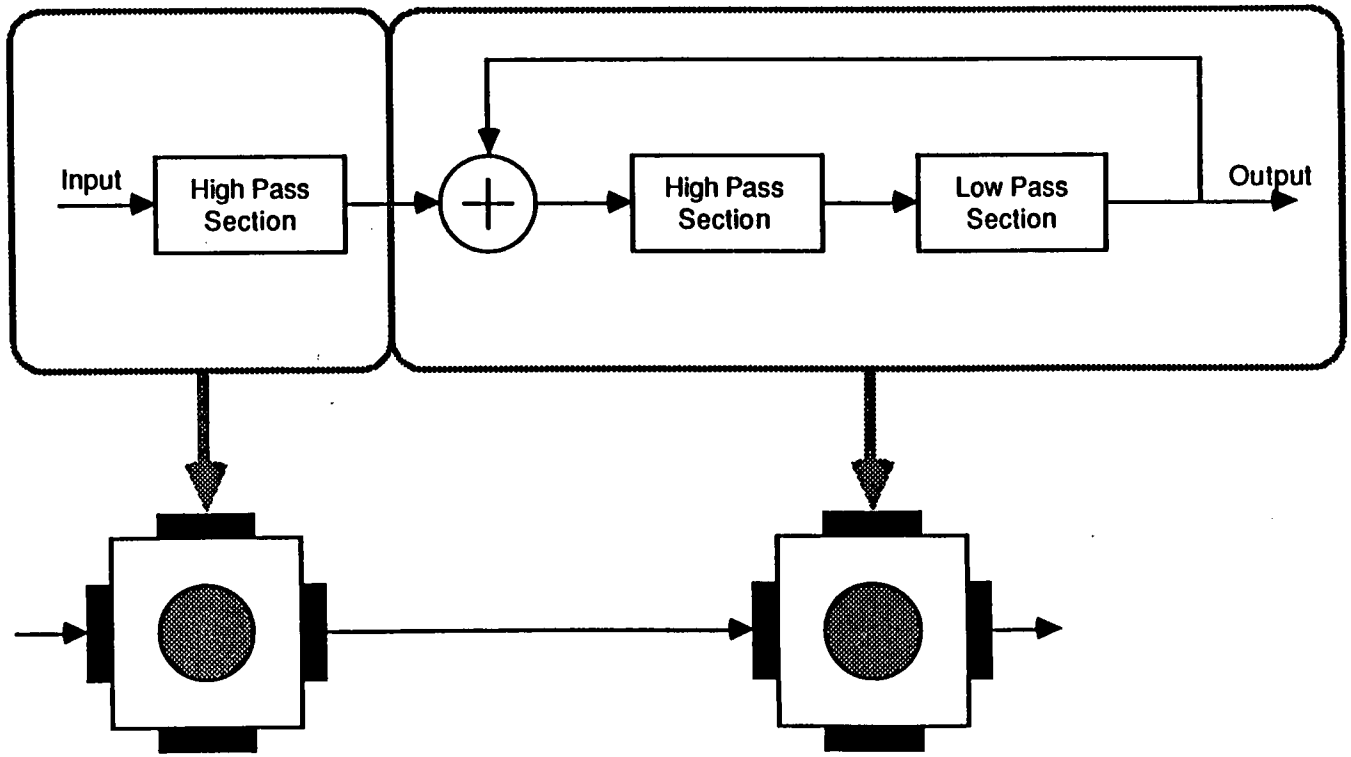


Fig 3.3b The Two Processor Mapping

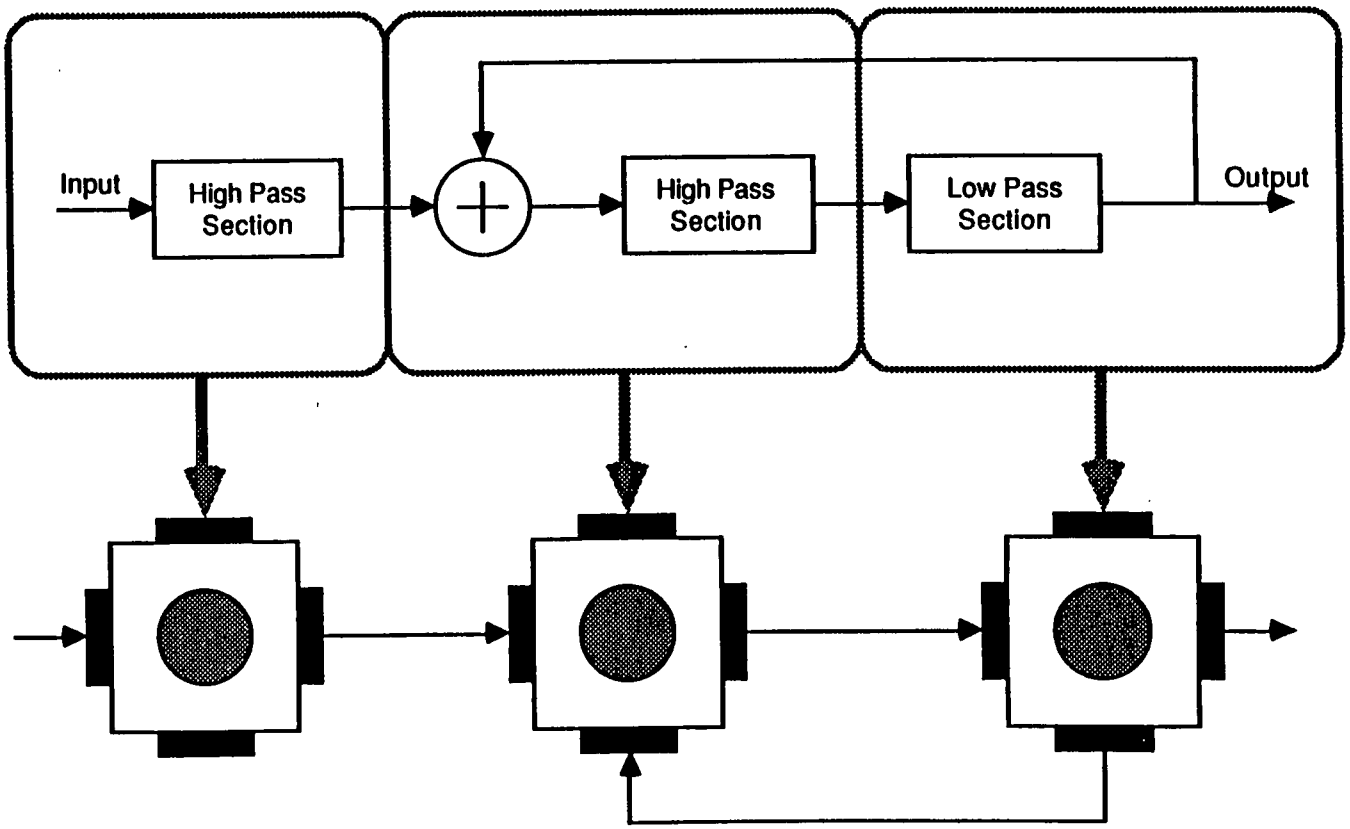


Fig 3.3c The Three Processor Mapping

```
PROC high.pass.A(CHAN OF ANY input,output)

  ... Declarations
  ... Initialisation

  WHILE TRUE
    SEQ

      PRI PAR
        PAR
          ... communicate set A
        SEQ
          ... compute set B

      PRI PAR
        PAR
          ... communicate set B
        SEQ
          ... compute set A

:
```

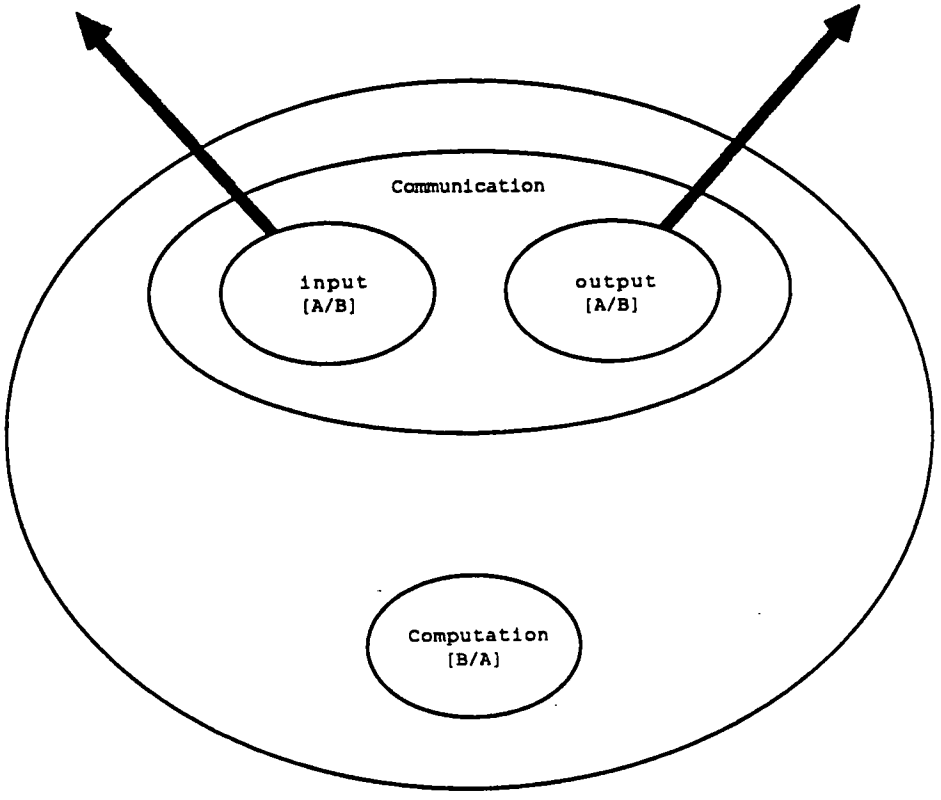


Fig 3.4 Harness Type I

```
PROC high.pass.B(CHAN OF ANY input,output)

  ... Declarations
  ... Initialisation

    PRI PAR
      PAR
        WHILE TRUE
          SEQ
            ... input.buffer
        WHILE TRUE
          SEQ
            ... output.buffer

      SEQ
        ... input.data.from buffer
        ... compute
        ... output.data.to.buffer

:
```

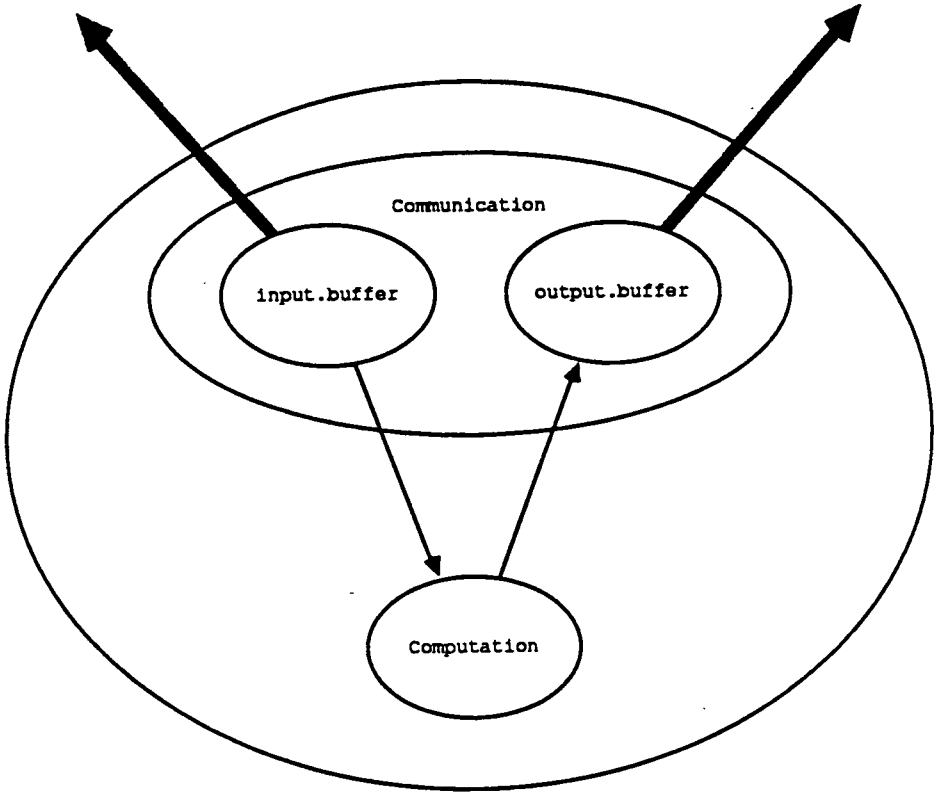


Fig 3.5 Harness Type II

```
internal := ((in - lp.out) >> 15)
IF
  internal >=#10000
    internal := internal \/ #FFFF0000
  TRUE
    SKIP
lp.out := lp.out + internal

hp.out := in - lp.out
internal := (hp.out >> 15)
IF
  internal >=#10000
    internal := internal \/ #FFFF000
  TRUE
    SKIP
lp.out := lp.out + internal
```

```
LDL   in
LDL   lp.out
SUB
STL   hp.out
LDL   hp.out
LDC   15
SHR
STL   internal
LDC   65536 (#10000)
LDL   internal
GT
EQC   0
CJ    -9
LDL   internal
LDC   -65536 (#FFFF0000)
OR
STL   internal
LDL   lp.out
LDL   internal
ADD
STL   lp.out
```

Fig 3.6 Occam2 Versions of High and Low Pass Filter Sections and the Disassembly of the High Pass Section

```
hp.out := in - lp.out
internal := (hp.out >> 15)
GUY
  LDL internal
  LDC #10000
  XWORD
  STL internal
lp.out := lp.out + internal
```

Fig 3.7 Arithmetic Shifting Using Explicit Sign Extension

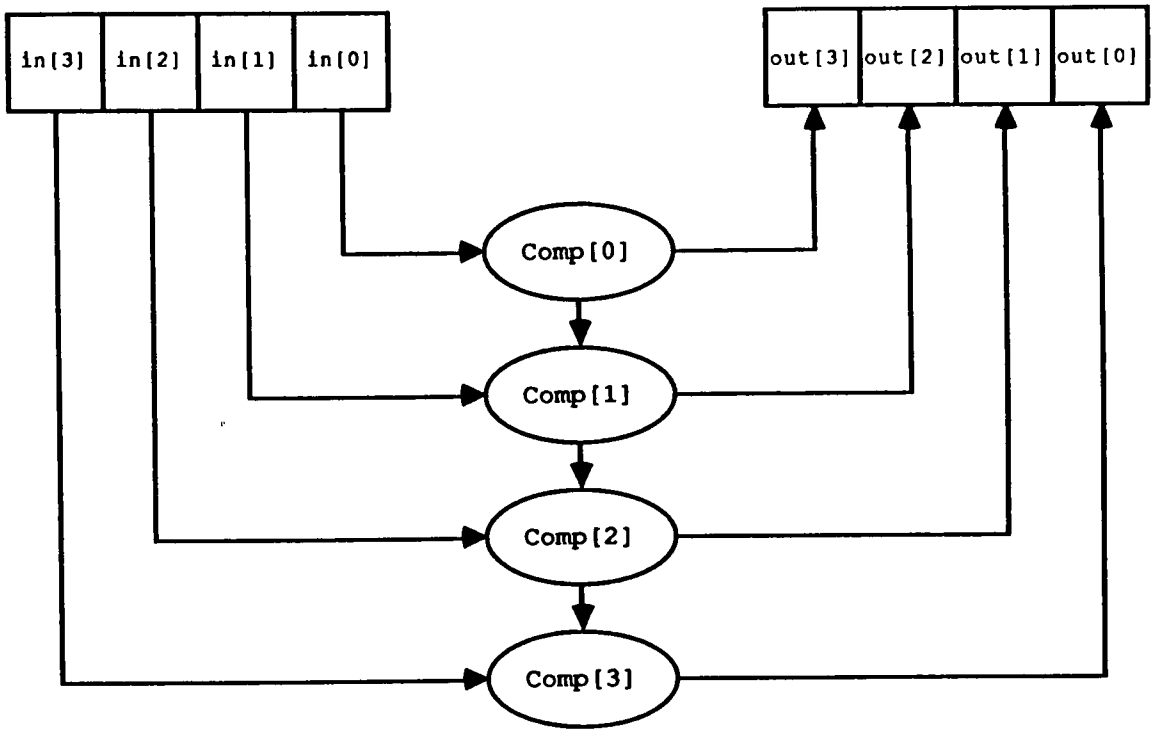
```
LDL   in
LDL   lp.out
SUB
STL   hp.out
LDL   hp.out
XDBLE
LDC   #0F
LSHR
LDL   lp.out
ADD
STL   lp.out
```

Fig 3.8 Arithmetic Shifting Using Implicit Sign Extension

```
LDL   a      c := a + b
LDL   b      e := c - d
ADD
STL   c
LDL   c
LDL   d
SUB
STL   e
```

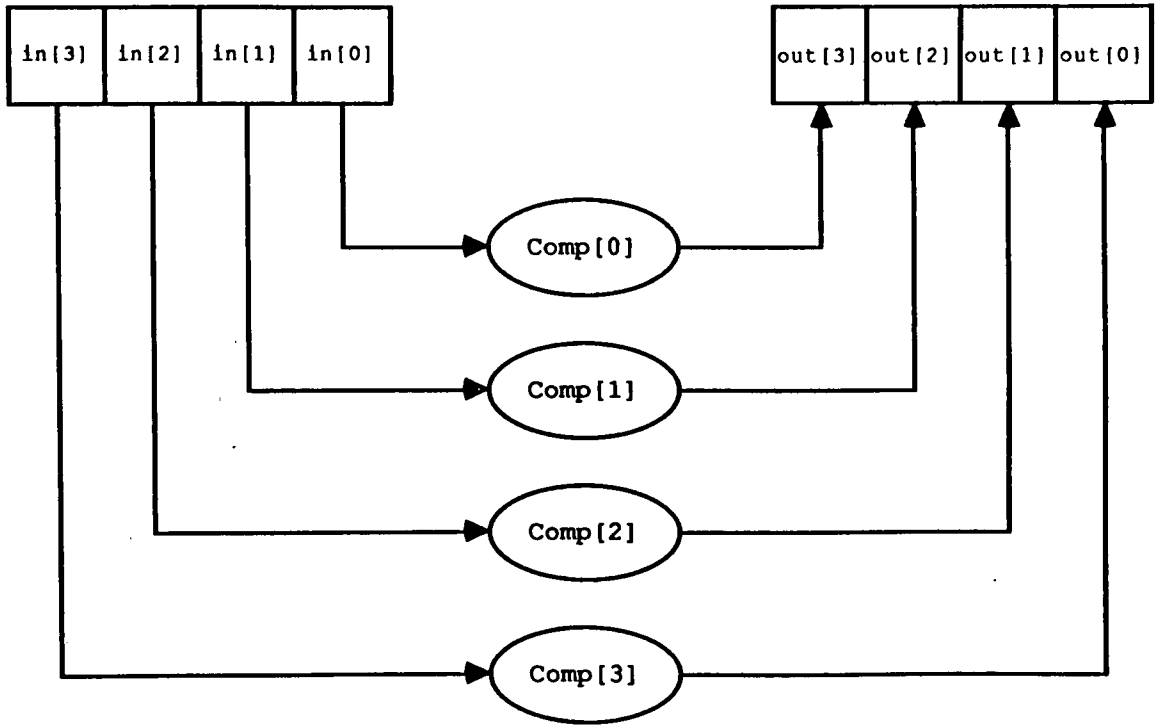
```
LDL   a      e := (a + b) - d
LDL   b
ADD
LDL   d
SUB
STL   e
```

Fig 3.9 Compounding Code onto a Single line




```
        out[i] := a0*section_ip+a1*previous_ip
                  + b1*previous_op + b2*preprevious_op
   previous_ip := in[i]
preprevious_op := previous_op
   previous_ip := out[i]
```

Fig 3.10 Processing A Single Channel Using Vectors




```
            out[i] := a0*section_ip[i]+a1*previous_ip[i]
                      + b1*previous_op[i] + b2*preprevious_op[i]
    previous_ip[i] := in[i]
preprevious_op[i] := prevous_op[i]
    previous_ip[i] := out[i]
```

Fig 3.11 Processing Multiple Channels Using Vectors

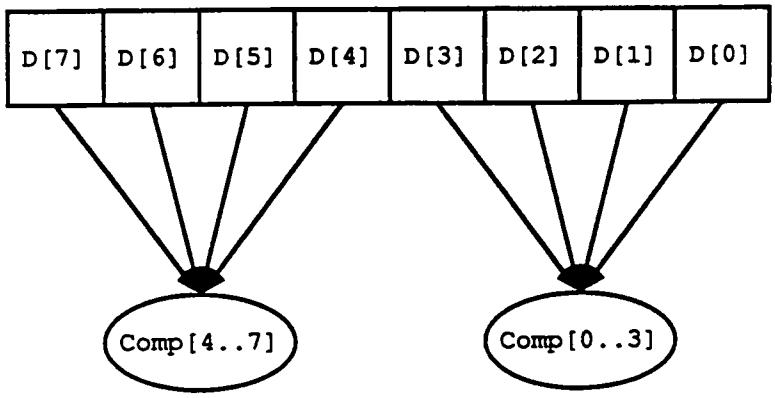


Fig 3.12 Buffered Multiple Channel Processing Using Vectors

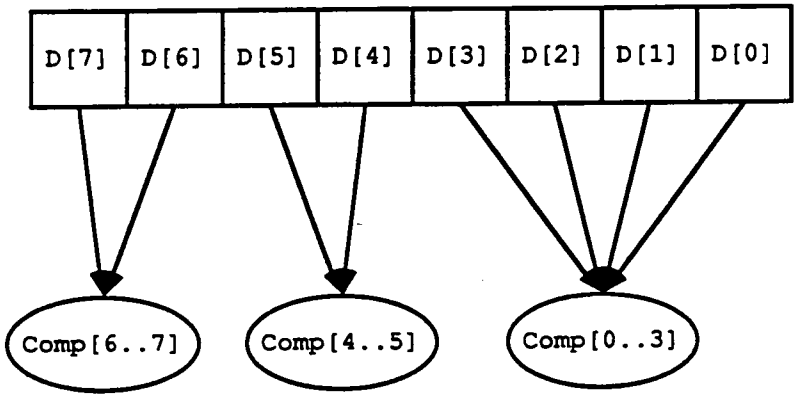


Fig 3.13 Multiple Channel, Multirate Processing Using Vectors

```
WHILE TRUE
  SEQ
    SEQ i = 0 FOR vector.size
      SEQ
        internal.val1 := in[i] - out[i]
        internal.val2 := internal.val1 >> 1
        out[i] := internal.val2 + 1
```

```
LDC   0
STL   0
LDC   4
STL   1
LDL   0
LDLP  6
WSUB
LDNL  0
LDL   0
LDLP  2
WSUB
LDNL  0
SUB
STL   11
LDL   11
LDC   1
SHR
STL   10
LDL   10
ADC   1
LDL   0
LDLP  2
WSUB
STNL  0
LDLP  0
LDC   30
LEND
```

Fig 3.14 A Replicated SEQ Structure and its Disassembly

```
WHILE TRUE
  SEQ
    internal.val1 := in[0] - out[0]
    internal.val2 := internal.val1 >> 1
    out[0] := internal.val2 + 1
    internal.val1 := in[1] - out[1]
    internal.val2 := internal.val1 >> 1
    out[1] := internal.val2 + 1
    internal.val1 := in[2] - out[2]
    internal.val2 := internal.val1 >> 1
    out[2] := internal.val2 + 1
    internal.val1 := in[3] - out[3]
    internal.val2 := internal.val1 >> 1
    out[3] := internal.val2 + 1
```

```
LDL   in[0]
LDL   out[0]
SUB
STL   internal.val1
LDL   internal.val1
LDC   1
SHR
STL   internal.val2
LDL   internal.val2
ADC   1
STL   out[0]
LDL   in[1]
LDL   out[1]
SUB
STL   internal.val1
LDL   internal.val1
LDC   1
SHR
STL   internal.val2
LDL   internal.val2
ADC   1
STL   out[1]
LDL   in[2]
LDL   out[2]
SUB
STL   internal.val1
LDL   internal.val1
LDC   1
SHR
STL   internal.val2
LDL   internal.val2
ADC   1
STL   out[2]
LDL   in[3]
LDL   out[3]
SUB
STL   internal.val1
LDL   internal.val1
LDC   1
SHR
STL   internal.val2
LDL   internal.val2
ADC   1
STL   out[3]
```

Fig 3.15 "Opening Out" a Sequential Loop

```
PROC replicated.par(CHAN OF ANY in,out)
  VAL vector.size IS 4 :
  [vector.size]INT in, out, internal.val1, internal.val2 :
  SEQ
    ... Initialisation
    WHILE TRUE
      PAR i = 0 FOR vector.size
        SEQ
          internal.val1[i] := in[i] - out[i]
          internal.val2[i] := internal.val1[i] << 1
          out[i] := internal.val2[i] + 1
:
```

Fig 3.16 A Replicated PAR Structure

```
PROC inputter(CHAN OF ANY to.filter)
  VAL vector.size IS 1 :
  CHAN OF ANY from.host :
  PLACE from.host AT #05 :    --link1 input
  {{{  declarations
  INT len,error,val,char,any :
  }}}
  SEQ
    {{{
    WHILE TRUE
      [vector.size]INT output.val :
      SEQ
        SEQ i = 0 FOR vector.size
          output.val[i] := 1
        {{{  100 outputs
         to.filter ! output.val
        }}}
    }}}
:
```

Fig 3.17 The Filter Source Process

```
PROC watch(CHAN OF ANY in)
  VAL array.len IS 1 :
  INT i :
  WHILE TRUE
    SEQ
      SEQ i = 0 FOR 20
        PRI PAR
          {{{  DECS
          [20]INT start.time,end.time :
          CHAN OF ANY tohost,fromhost :
          TIMER clock :
          [array.len]INT any :
          PLACE tohost AT #02 :
          PLACE fromhost AT #06 :
          }}}
          SEQ
            clock ? start.time[i]
            {{{  100 inputs
            in ? any
          }}}
            clock ? end.time[i]
            tohost ! start.time[i] / array.len
            tohost ! end.time[i] / array.len
          SKIP
:
```

Fig 3.18 The Filter Stopwatch Process.

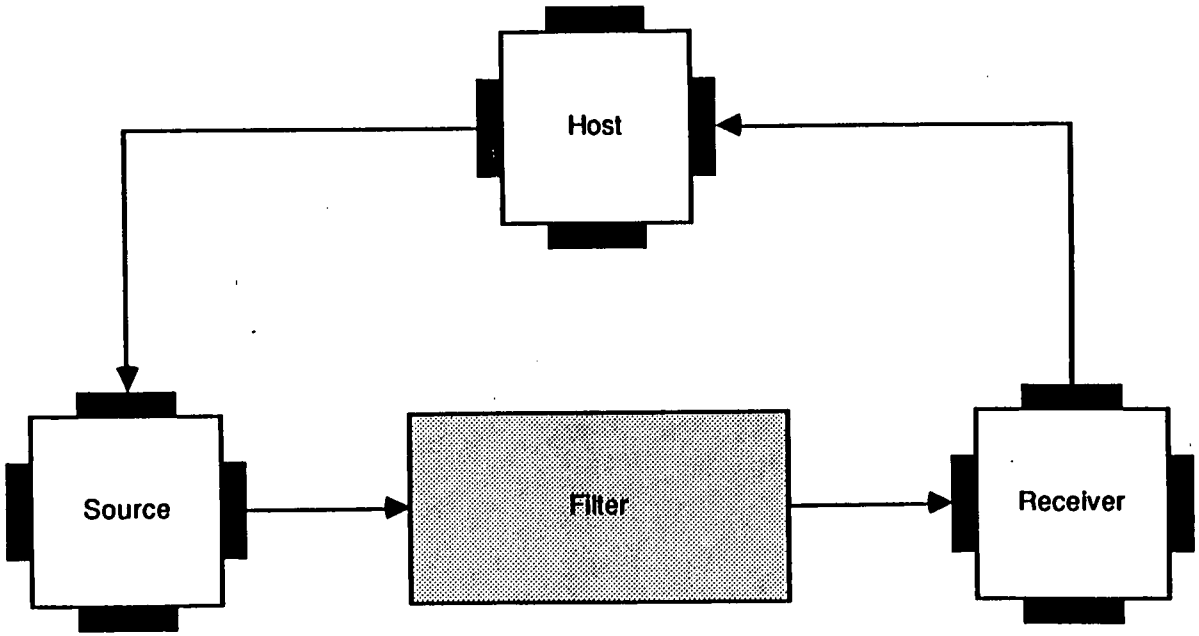


Fig 3.19 Schematic of Filter Performance Test Configuration

Chapter 4

Transputer Code: Performance Analysis and Results

4.1 Introduction

The code running on any particular transputer usually consists of two or more parallel processes. At which time each of these processes is executed, and for how long, is controlled by the transputer scheduler and depends upon the state of the communication channels, timers and the timeslice period. The scheduler schedules, deschedules, reschedules and executes the processes according to their state and their position in the active process queues. The operation of the scheduler is largely transparent to the programmer, and so very little information concerning the detailed execution of the program may be obtained by analysing only its Occam2 source code.

In order to fully appreciate the effect of program structure upon performance, and to assess the impact of such parameters as vector length, it is necessary to break down the Occam2 source into transputer instructions, and to determine how the transputer executes this code.

This approach has been used in this chapter to assess the impact of program structure and vector length upon the performance of the Occam2 filter programs. Not only does this allow the performance of the code to be predicted, but also the

overheads associated with each harness to be assessed. The latter enables the most appropriate harness to be chosen for similar applications, taking into account the number of communications channels, the vector length and the execution time of the low priority process.

The theoretical results obtained using this method are presented, together with the corresponding empirical results, and a comparison made between the two. The accuracy of the theoretical predictions is used to assess the methods limitations, and its applicability in predicting the performance of similar programs.

Section 2 outlines the manner in which an Occam2 program may be decomposed into machine instructions, and its operation determined using scheduling charts. Section 3 describes an operational model of the transputer, which is used to generate the scheduling charts. The operation of each harness, for a particular mapping, is described in Section 4. Section 5 presents the theoretical and empirical results, compares them and makes an assessment of the decomposition method. Finally, Section 6 provides a summary.

4.2 Occam2 Programs — A Method of Decomposition

This section describes a method of analysing Occam2 programs in order to produce an estimate of the time taken to execute the code. A flow chart showing the steps involved in this method is shown in Fig 4.1.

The first step in decomposing a piece of Occam2 code is to convert the high level source code to solely transputer instructions. This disassembly was carried out using the TDS Debugger [68], which also provides a hex dump of the code. The disassembled instruction mnemonics, the hex representation of the instructions (used

as a double check for prefixing), their memory location and the number of processor cycles required were placed in a table.

Using this table, the code was grouped into its main components — general process and channel initialisation, vector initialisation loops, concurrent process initialisation and the communication and computation code sections (the recognisable Occam2 processes). A representation of the location of the main components of the program is thus constructed, an example of which is shown in Fig 4.2, for harness type II, single processor mapping.

This decomposition may be used to construct a more graphical representation of the structure of the program, Fig 4.3. This representation labels the major sections of the code, together with their execution cycles, and shows not only the parallel nature of the program but also the logical flow of execution of each process.

This "graphical" representation of the program is used in conjunction with an operational model of the transputer to construct a further chart, the *scheduling chart*, which is used to determine the operation of the code. A section of the scheduling chart for the process depicted in Figs 4.2 and 4.3 is shown in Fig 4.4. It references the same blocks of code as the graphical representation, and uses the same labelling strategy, but also provides information concerning the currently executing process, the currently active and inactive processes at each priority level, and the state of the communication channels. This allows the time required to complete any section of code to be determined, in addition to providing information concerning when processes are descheduled, rescheduled or interrupted.

For this particular application, the sequence of instructions will settle down into a loop, and it is the length of the loop, in instruction cycles, that must be determined

in order to provide a performance estimate for the program. The scheduling chart allows the length of the loop to be easily determined. The scheduling chart, importantly, also allows the overheads involved in executing the program to be quantitatively assessed.

4.3 The Transputer — an Operational Model

The scheduling chart is constructed by applying a set of rules concerning the operation of the transputer to the graphical representation of the program. These rules constitute an operational model, and are concerned primarily with the manner in which the transputer both allocates cpu time to parallel processes and performs channel communication. The operation of the transputer has been considered in some detail in Chapter 2, and will not be repeated here. However, three main rules associated with the operational model are listed below:-

i. A high priority process becomes active immediately upon inception by either a RUNP or a STARTP instruction. If the process is initialised by a STARTP then, in the code presented here, a high priority process is already running, and the process will be placed at the end of the high priority queue. If the process is initialised by a RUNP, however, then a low priority process is executing. In this case, the low priority process is interrupted, its state stored in internal memory, and the high priority process executed.

ii. Interruption of a low priority process by a high priority process requires 18 processor cycles.

iii. Descheduling requires 18 processor cycles.

There are other factors affecting the overall timing of the program which are independent of the operation of the scheduler. The timing of some of these depends upon the vector size and determines the order of execution of the code. In order to alleviate the need for a different model for different vectors sizes, it has been assumed that the vector size is a particular value whenever such instances arise. These

additional rules are listed below:-

iv. Any external channel communications are immediately serviced, there is no communication latency.

v. Internal memory is used exclusively.

vi. Accessing local variables in the computation section may or may not require a single level of prefixing for small values vector sizes, depending on which harness is being used. However, the overall prefixing level rapidly approaches one for any reasonable vector size. Hence the level of prefixing in the computation section is assumed to be one.

vii. At some points in the execution of the code, the ordering of operations is dependent upon a threshold value of w. In such cases, the value of w is assumed to be 16, a "large" value.

viii. Whenever a low priority process is interrupted, it is allowed to complete execution of its present instruction. The execution time of this instruction is taken to be the average instruction execution time of the low priority (computation) section, 4 cycles.

ix. The link speed is fixed at $20Mbits^{-1}$

In addition, for the multi-processor operation of the transputer, it is assumed that:

ix. The performance of any multi-processor implementation is determined by the performance of the processor running the largest computation section. (This is assumed to be the case for any program using a single sequential low priority process).

4.4 The Operation of the Harnesses

This section outlines the sequence of operations involved in executing the single processor mapping in each harness. The single processor mapping has been chosen as it most readily demonstrates the difference in memory requirements of the two harnesses. For any given harness, the sequence of operations is similar for all processor mappings, the main difference occurring in the additional complication of the second and third processes of the three processor mapping due to the extra communication channel. The operation of each harness will be considered in turn.

4.4.1 Harness Type I

This harness is described by Fig 4.5 in terms of its hedgehog diagram and Occam2 code.

After general process and vector initialisation is completed, the main loop begins. The high priority processes are first invoked in turn. These both initiate external communication transfers and so are descheduled. The low priority computation process is then allowed to proceed until the first external communications transfer completes and a high priority process rescheduled. The state of the low priority process is stored, and the high priority process allowed to proceed, continuing by ending itself. The low priority process is again allowed to continue until the second communications transfer completes, and the second high priority process becomes active. Once more, the state of the low priority process is saved and the high priority process allowed to proceed, which does so by ending itself. The completion of this final high priority process signals to the "parent" of the communications processes that all of its "children" have completed, and that it may call its successor process. The successor process in this case is the standard de-prioritising code, which also ends itself upon completion. The low priority process is then allowed to continue unhindered until it too ends itself. This signals to the process controlling the `PRI PAR` construct that all of its constituent processes have completed, and that it may invoke its successor, which is the next `PRI PAR` construct and operates in exactly the same way as the first construct.

There are no internal communication channels (the processes are decoupled, as depicted in the hedgehog diagram), as data is passed by reference between the communication and computation processes. For instance, inside one `PRI PAR`

construct, data set A may be communicated and data set B computed, whereas in another PRI PAR construct data set B is communicated and data set A computed. Internal communication is avoided, then, but at the expense of memory space. The memory requirement of this type of harness is high, as the code is duplicated inside each PRI PAR construct. The code may well spill out into external RAM, which is accessed much more slowly than internal memory, thus affecting performance. The overall operation of the harness, then, is of a repeating sequence of PRI PAR constructs which are continually set up and closed down. Inside each PRI PAR, high and low priority processes are themselves initiated and terminated. The overheads associated with this harness are those incurred by this continual initiation and termination of processes and constructs.

4.4.2 Harness Type II

The hedgehog diagram and Occam2 code for this harness are given in Fig 4.6.

After general initialisation, the high priority communication processes are invoked. The first process enters its WHILE TRUE loop and tries to execute a transfer on an empty internal channel and so is descheduled in preference to the second process. This process enters its WHILE TRUE loop and executes an external communications transfer, also causing it to be descheduled. The low priority process continues by initialising its local vectors and enters its WHILE TRUE loop by trying to execute a transfer on an empty internal channel, whereupon it is descheduled. There is now a delay until the first communications process completes its transfer and is rescheduled. This process continues by executing an internal transfer, which also reschedules the low priority process. The high priority process continues by jumping

to the beginning of its loop and executing an external transfer, thus being descheduled. This allows the low priority process to continue by entering its computation section. The low priority process continues until it is interrupted by the newly rescheduled communications process. This high priority process continues by trying to execute a transfer on an empty internal channel, and so is descheduled. This allows the low priority process to complete its computation section. This process continues by executing an internal transfer, which also reschedules the second high priority process. This causes the low priority process to be interrupted by the second communication process, which continues by executing an external transfer, so being descheduled. This once again allows the low priority process to continue, by jumping to the top of its loop and executing an internal transfer, rescheduling the first communication process. The low priority process is interrupted by this communications process, which continues by jumping to the top of its loop. At this point, all processes have completed a single pass of their `WHILE TRUE` loops. Execution continues in a similar manner, although the delay incurred by waiting for an external transfer does not occur again.

The operation of this type of harness is more complex than that of the other harness. Each communication process is coupled to the computation process via an internal channel, as shown in the hedgehog diagram.The overall memory requirement is nearly half of that of Type I, allowing larger computation sections to be implemented in internal memory. The individual processes never terminate as they continually repeat inside local `WHILE TRUE` loops. The `PRI PAR` construct is initiated only once at the beginning of the program. Thus the overheads relating to process initiation and termination incurred by Type I do not occur here. The main source of

overheads for this harness is the internal communication, which takes the form of a cpu intensive memory to memory transfer.

The relative effects of the overheads of the two harnesses are investigated in the next section.

4.5 Results

This section presents both the theoretical performance estimates and the results obtained from the transputer system for each harness and mapping. Initially, the theoretical performance of each harness and mapping is given, together with their associated overheads. The empirical results are then discussed, followed by a comparison of the theoretical and empirical results.

4.5.1 Theoretical Performance Figures

The procedure outlined in Sections 4.2 and 4.3 was applied to each of the mappings for both of the harnesses, and the performance of each derived as a function of vector length, w. These results are presented in Appendix C, and their graphical representations shown in Fig 4.10. From rule ix of the performance model outlined in section 4.3, only the program using the largest computation section was considered in the multiple processor mappings.

4.5.2 Code Overheads

The term "overhead" will be applied to any operation other than computation or external communication. Hence, internal communication and descheduling / rescheduling operations are considered overheads, since they do not involve operations

directly related to the function of the filter.

The overheads are derived from the scheduling chart and are presented for each harness and mapping in Tables 4.1 and 4.2, a breakdown of the overheads incurred by the single processor mapping of each harness is presented in Tables 4.3 and 4.4. It may be seen from this table that for a given harness, the overheads are the same for the one and two processor mapping, but are larger for the three processor mapping. This is to be expected, as the three processor mapping makes use of an additional communications process. Each communication process incurs an overhead of *64* cycles for harness Type I, and *2w+53* cycles for Type II. Tables 4.5 and 4.6 provide a summary of the total number of cycles required to execute the code, and the total overheads incurred, for each harness and mapping.

4.5.2.1 The Impact of Overheads on Performance

The overhead associated with type I is not dependent upon vector length, whereas that of Type II is, due to internal communication transfers. This implies that the theoretical performance difference between types I and II varies with vector length. For short vectors, type II offers the highest performance. The changeover point occurs when

$$\textit{No. cycles required by TypeII} > \textit{No. cycles required by TypeI}$$

$$\textit{ie}\ 241 \quad 124w > 295 \quad 120w$$

$$4w > 54$$

$$w > 13.5(14)$$

So, type II should offer the best performance for vector lengths below 14.

The overhead associated with type II is more sensitive to the number of high

priority (external communications) processes. For type I, each high communications process incurs an additional *64* cycles, whereas for type II, this becomes *2w+53*.

4.5.2.2 The Effect of Vector Length and Computation Code Size

Harness type II offers the better performance if small vectors are used. This harness, then, would be best suited to applications requiring a relatively large computation section, as low vector sizes must be used in order to constrain the program to internal memory.

Harness type I is twice as sensitive to the required amount of computation code than type II. Code will tend to be forced into external memory for a lower vector size, and so type I will tend to favour smaller vector sizes than type II for large computation sections.

4.5.2.3 Summary

It is clear that which harness provides the best performance depends upon the vector size, the computation code length and the number of external communications channels required. The exact boundaries of vector size and computation code length will be dictated by the particular program under scrutiny. However, the analysis of the single processor mappings of this particular application may be summarised as follows.

Harness Type II provides the best performance for $w < 14$. For $w \geq 14$, then providing that external memory is not used, harness Type I offers the best performance. Harness type II will provide the best performance for values of computation code length and vector size outside this region. Eventually, external

memory accesses and communications overheads of harness Type II decrease its performance, and type I once again provides the best performance.

The performance of harness type II is more sensitive to external communications channels than Type I. If any more than two external communication channels are to be used, then Type I offers the better performance overall.

4.5.3 Empirical Results

This section examines the measured performance of the multi-channel filter as implemented on the transputer system. The analysis of these results is divided into two main groups:

i. Harnesses

ii. Processor mappings

Group i allows comparison of different processor mappings for a given harness, whereas group ii allows comparison of harnesses for a given processor mapping. The complete set of results is presented in Appendix C. Line plots of the time to compute one word of data against vector length are shown in Fig 4.10, for all mappings and harnesses.

4.5.3.1 Group i

Plots showing the performance of the mappings for each harness are given in Fig 4.8 a&b. They will be considered in turn.

Harness Type I — Observations:-

i All three mappings exhibit the trend of higher performance for larger vector

size.

ii The single processor mapping offers the lowest (absolute) performance.

iii For the single processor mapping, an increasing reduction in performance may be seen for vector sizes greater than 24.

iv The three processor mapping offers lower performance than the two processor mapping up to vector sizes of 32.

v The single processor mapping seems to experience a sharper decrease in performance than the two processor mapping. The three processor mapping does not seem to suffer any decrease in performance in the given range of vector size.

vi The maximum absolute performance is provided by the two processor mapping up to a vector size of 32, when the three processor mapping becomes the fastest.

Harness Type I - Explanations:-

i The general trend of increasing performance with vector size is due to the decrease in relative importance of the communications set up overheads.

ii The single processor mapping uses the largest computation code section. The time required to compute a word of data (computation time) dominates the time required to communicate a word of data (communication time), and so the

computation time will dominate the overall performance. Hence, the single processor mapping offers the lowest performance.

iii This harness requires a relatively large amount of workspace and program memory space, dependent upon the vector size and the size of the computation code section. The single processor mapping uses the largest computation code section and so requires the largest amount of memory. As the vector length is increased, then, the amount of space required to contain the workspace and program areas increases. At some particular value of vector size, the total memory requirement will exceed that available in internal memory, and the program area will begin to use slower external memory, causing the decrease in performance. Using the debugger, it was seen that for this mapping and harness, external memory was first used at a vector size of 24, which matches the point at which performance begins to be degraded. As four instructions are read every memory cycle, the performance is not as impaired as it would be if data areas where also placed off-chip, as in the case of very large workspace areas or by using the "separate vector space" option of the Occam compiler.

iv Surprisingly, the three processor mapping does not provide the maximum performance for all vector sizes. The computation code section of this mapping is small, and requires fewer cycles to complete than a link transfer. Thus this mapping is dominated by communication, in contrast to the other mappings. More communication is required in this mapping than in the others, and so any additional communications overhead or delay will significantly affect performance. The effects

of external memory access cause the performance of the two processor mapping to fall below that of the three processor mapping at $w = 32$.

v Performance degradation at large values of vector size is due to an increased usage of external memory. The single processor mapping requires more workspace and program code space per word than the other mappings, and so will make more use of external memory for a given increase in vector size, causing a larger decrease in performance. For the given range of vector size, the memory requirements of the three processor mapping may be met solely by internal memory, and so no performance degradation is exhibited.

vi Although the maximum overall performance is provided by the two and three processor mappings, the single processor case offers the highest performance *per processor*. This is not surprising, perhaps, as if this were not the case, it would imply that the overall overheads associated with this harness are reduced in the multiple processor mappings, which surely cannot be the case. The best that could have been expected was a linear speed up with an increased number of processors.

Harness Type II — Observations:-

i The mappings of this harness exhibit the same general trend of increasing performance with vector size.

ii The performance of the single processor mapping begins to degrade at a vector

size of around 44, but not as rapidly as for harness type I.

iii The three processor mapping, surprisingly, offers the lowest performance.

iv Maximum performance is attained by the two processor mapping.

Harness Type II - Explanations:-

i As for harness type I, this increase in performance is due to the relative decrease in importance of overheads.

ii The communication and computation processes are not duplicated in this harness, and so less memory per word is required. The single processor mapping uses more memory than the other mappings, and so it will require external memory at lower values of vector size, causing a corresponding decrease in performance. As less memory is required by this harness, however, then performance degradation will occur at higher vector sizes than for the other harness.

iii As outlined above, the performance of the three processor mapping is dominated by communication rather than by computation. This harness is experiences more communications' overhead than Type I, and so will experience more of a performance degradation as a result.

iv It would be expected that the two processor mapping be faster than the single processor mapping, due to the smaller computation code size.

4.5.3.2 Group ii

Plots showing the performance of both harnesses for each of the mappings are shown in Fig 4.9.

Observations:-

i For the single processor mappings, the performance of each harness is similar, although harness type II performs marginally better for vector sizes below 12 and above 32. The minimum sampling period (maximum sampling frequency) of 6.41 microseconds (156kHz) is attained by harness type I at a vector size of 24.

ii The performances of both harnesses of the two processor mapping are also very similar. Harness type I performs slightly better than type II up to a vector size of 32. The performance of harness type II is not degraded within the given range of vector sizes, providing the minimum sampling period of 4.624 microseconds (216.2kHz) at a vector size of 48.

iii In contrast to the two cases above, markedly different performances are provided by the harnesses for the three processor mapping. Harness type I performs much better than harness type II, providing the minimum sampling period of 4.73 microseconds (211.4kHz) at a vector size of 48. Neither harness experiences a performance degradation within the given range of vector size.

Explanations:-

i The better performance offered by harness type II below a vector size of 12 is probably due to a lower proportion of operational overheads, although this does require theoretical confirmation. The better performance of harness type I between vector sizes of 12 and 32 is similarly caused by a difference in operational overheads. The performance of harness type I begins to degrade at a vector size of 24, due to external memory accesses. At a vector size of 32, the inherent overheads of harness type I, combined with the additional overhead incurred by external memory access become greater than those experienced by harness type II, resulting in harness type II providing the better performance.

ii Harness type I begins to feel the effects of external memory access at a lower value of vector size than harness type II, hence the degradation at a vector size of 36. Harness type II does not need to use external memory for the given range of vector size, and so experiences no performance degradation.

iii The computation code sections of the three processor mapping is small, its execution time being less than the time required to transfer data over a link. Thus, any additional overheads will have significant impact on performance. The soft communications' overheads experienced by harness Type II will be particularly significant.

4.5.3.3 Summary

It may be seen that the performances of the two harnesses for the one and two

processor mappings are very similar. Harnesses type I and II provide the best performances for the one and two processor mappings respectively. The effects of external memory access may be seen in both mappings, especially for harness type I.

The harnesses for the three processor mapping provide very different performances, however. Here, harness type I provides more than double the performance of type II. This is probably because of the proportional increase in the overhead of type II, caused by the additional external / internal communication channel. It is interesting to note that for harness type II, the three processor mapping actually provides the poorest performance of all the mappings. In this case, parallelising the code actually causes a performance decrease.

The maximum performance per processor is always attained by the single processor mapping.

4.5.4 Comparison of Empirical and Theoretical Results

The theoretical predictions of the performance of the mappings for both harnesses are derived from Appendix C. The performance equations are summarised in Tab 4.2, and shown in graphical form, together with the corresponding empirical performance curve in Fig 4.10a,b,c,d,e,f. Also included in these plots is a measure of the accuracy of the theoretical predictions — the percentage error — which is defined as

$$Percentage\,Error = \frac{(Empirical value - Theoretical value)}{Empirical Value} \times 100$$

and the key is given by Mapping_Harness Type, where Mapping = 1,2 or 3, Harness = I or II and Type = Emp (Empirical), Thy (Theoretical) or % (Percentage Error).

These comparison curves all exhibit various similar properties that serve to

demonstrate the limitations, and the accuracies, of the performance models. These properties will be listed, and discussed.

Observations:-

i It may be seen from these plots that the theoretical curves all predict a lower performance for small vector sizes than is actually attained.

ii This is more noticeable for harness type I than for harness type II.

iii For the single processor mappings, and to a lesser extent the two processor mapping of harness type I, the theoretical curves diverge from the empirical curves at large values of vector size. This is most noticeable in the single processor mapping of harness type I.

Explanations:-

i The model assumes that the computation section variable accesses all incur a single prefixing overhead. However, very few, if any, prefixing instructions will be required to access variables if the vector size is small. Hence, the models predict a longer computation cycle and hence lower performance.

ii The additional internal communications overhead experienced by harness type II could mask this difference in computation length to some extent, resulting in the decreased difference between theoretical and empirical performance.

iii The theoretical models assume that only fast internal memory accesses are made, and so they do not take into account slower external memory usage. It has already been shown that for large vector sizes, the code will eventually spill out into external memory, causing a performance decrease. This is happening in the empirical curves for the single processor harnesses, and the two processor mapping of harness type I.

4.6 Summary

A systematic method of decomposing Occam2 programs was developed, in order to allow the performance of a program to be predicted and to investigate the effects of program structure upon performance. An operational model of the transputer was applied to a graphical representation of the program, producing a scheduling chart which gave information concerning the status of constituent processes and associated communications channels at any given time. Information such as the execution period of a program and additional overheads incurred may be derived from this chart.

Theoretical performance figures were obtained for each harness and processor mapping using this method. The information obtained concerning the operational overheads of each harness allows the appropriate harness to be chosen for programs of a similar structure, for any given vector size.

The empirically obtained performance data has also been presented in this chapter. Surprisingly, the three processor mappings did not offer significantly better performance than the other mappings. This was explained by the low execution time of the low priority processes and the increased communications requirement of this mapping. Maximum performance within the given range of vector size was offered by

quoted vector size of 48, and so would be expected to further increase with larger vector sizes. The two processor mapping will experience decreased performance due to external memory access at a smaller vector size than the three processor case, and so the three processor mapping would be expected to out perform the two processor case above a particular value of vector size. The maximum vector size used experimentally was limited to 48 by the available compiler memory space.

The predicted performance figures were compared with the empirical data and found to match to within less than 10% for the most part, any deviation being explained by the limitations imposed by the operational model. Exceptions to this were the three processor mappings, whose predicted execution periods were less than those obtained empirically. This highlighted a major limitation of the operational model when applied to multi-processor mappings making use of short low priority code sections, namely the inability to adequately take into account the effect of communications synchronisation.

Nevertheless, this model performs very well for most of the programs analysed in this chapter. An increased sensitivity to external communication synchronisation could be incorporated as an extension to the present analysis method.

The information gained by analysing the application filter programs in this way may be used to determine the most efficient form of implementation of similarly structured application programs.

Overheads incurred by Harness Type I	
Description	Cycles Required
Set up main parallel process	34
Set up high priority parallel process	23
High priority ENDPs	32
De-prioritisation code ENDP	16
Main process ENDP	16
De-prioritisation code	25
Context switching	126
Total	272

Table 4.1

Overheads Incurred by Harness Type II	
Description	Cycles Required
Soft communication transfer	C x (2w + 19)
Soft communication transfer set up	2C x 6
WHILE TRUE jumps	12
Context switching	144
Total	C2w + 218

Table 4.2

C - number of channels

Description	Mapping		
	1	2	3
Set up main parallel process	34	34	34
Set up high priority parallel process	23	23	35
Deprioritising code	32	32	48
High priority endps	16	16	16
Main process endp	16	16	16
Deprioritising code endp	25	25	25
Interrupt/scheduling	126	126	162
Total	272	272	336

Table 4.3 Breakdown of Overheads for Harness Type I

Description	Mapping		
	1	2	3
Internal communications set up	2(2w+19)	2(2w+19)	3(2w+19)
Internal communications transfer	12	12	18
Loop jumps	12	12	16
Interrupt/scheduling	144	144	180
Total	4w+218	4w+218	6w+271

Table 4.4 Breakdown of Overheads for Harness Type II

Mapping	Harness	
	I	II
1	120w+295	124w+241
2	75w+295	79w+241
3	46w+262	42w+298

Table 4.5 Number of cycles required to execute code loop

Mapping	Harness	
	I	II
1	272	4w+218
2	272	4w+218
3	336	6w+271

Table 4.6 Summary of overheads

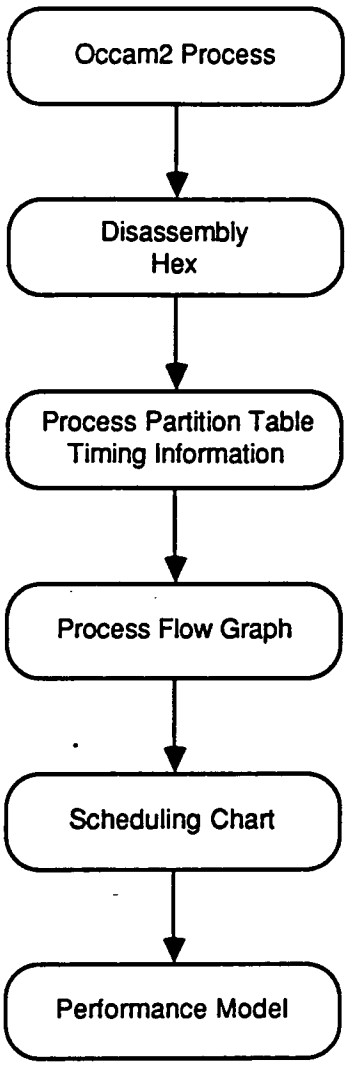


Fig 4.1 Flow Chart of Process Decomposition

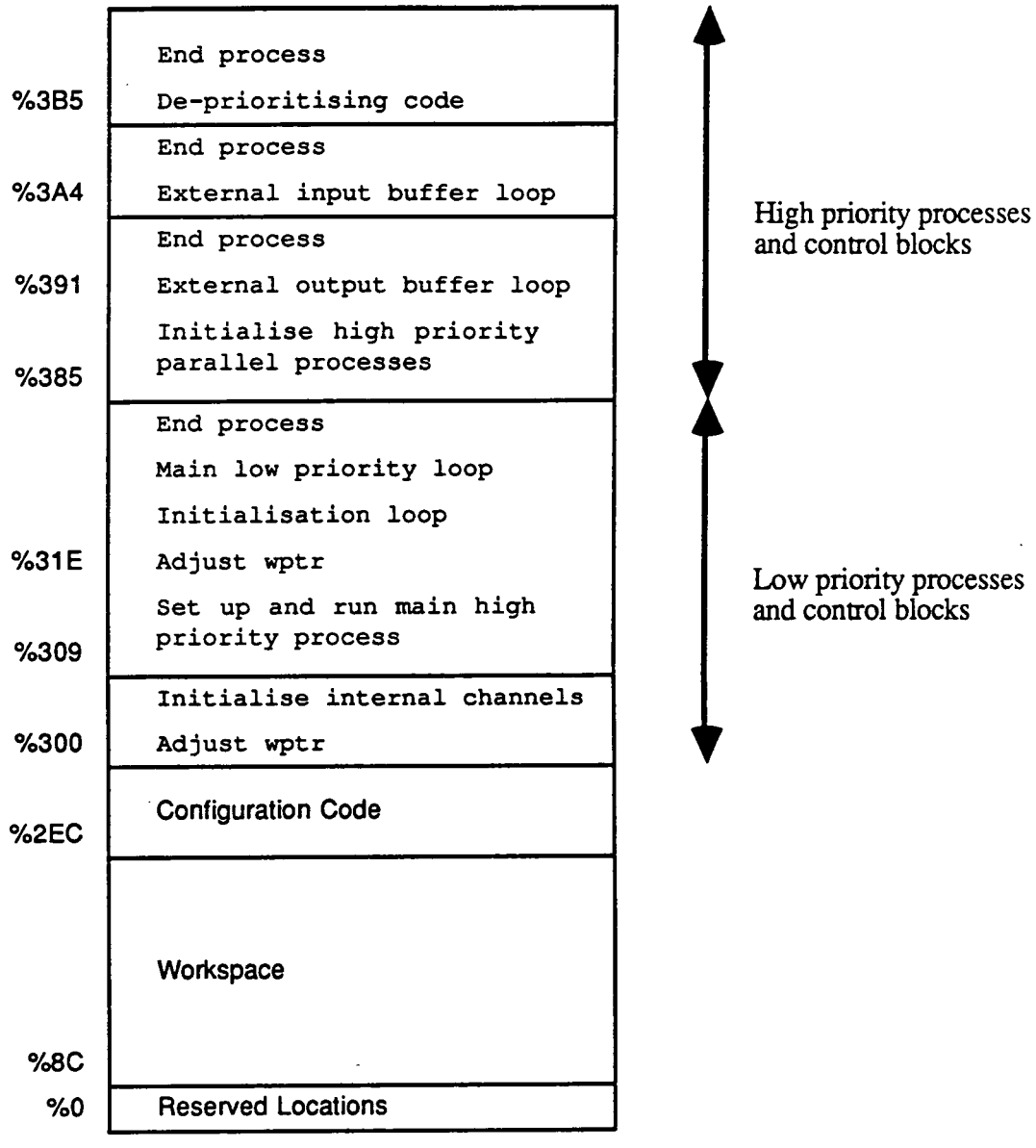


Fig 4.2 Memory Utilised by the Second Process of the Two Processor Mapping, Harness Type II

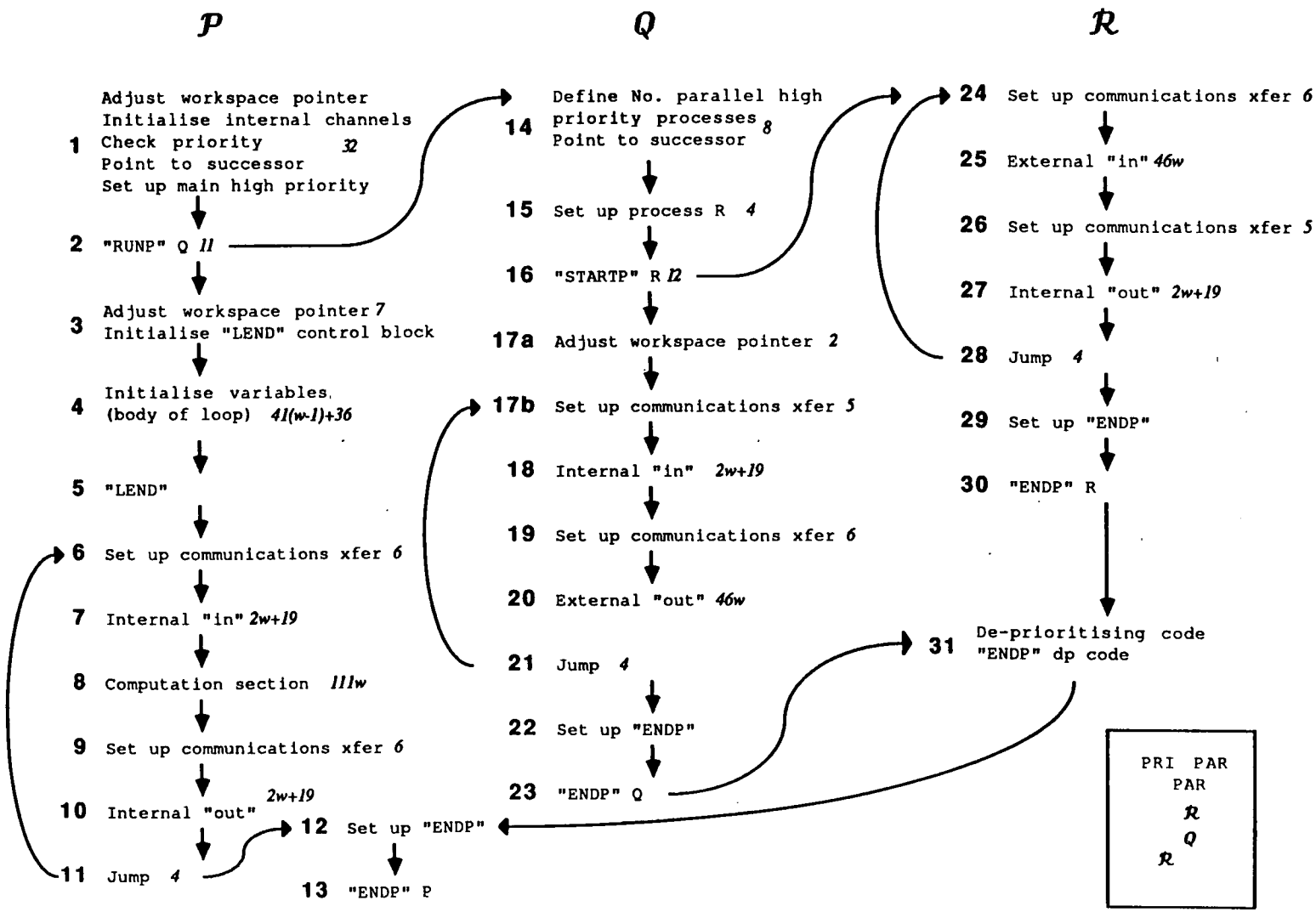


Fig 4.3 Flow Graph Representation of a Process

Note ref.	Processor Cycles	Label	PROCESS STATUS											
			SCHEDULING					COMMUNICATION						
			Execute	Active		Inactive		P		Q		R		
				P0	P1	P0	P1	in	out	in	out^E	in^E	out	
1	32	1	P'											
2	11	2		Q'	P'									
3	8	14	Q'		P'									
4	4	15	Q'		P'									
5	12	16	Q'	R	P'									
6	7	17	Q	R	P'									
7	19	18	R		P'	Q				H				
8	6	24	R		P'	Q				"				
9	19	25	P'			Q,R				"		xfer		
10	7	3	P			Q,R				"		"		
11	41(w-1) + 36	4	P			Q,R				"		"		
		5	P			Q,R				"		"		
12	6	6	P			Q,R				"		"		

Fig 4.4 A Section of the Scheduling Chart for the Two Processor Mapping of Harness Type II

```
PROC high.pass.A(CHAN OF ANY input,output)

  ... Declarations
  ... Initialisation

  WHILE TRUE
    SEQ

      PRI PAR
        PAR
          ... communicate set A
        SEQ
          ... compute set B

      PRI PAR
        PAR
          ... communicate set B
        SEQ
          ... compute set A

:
```

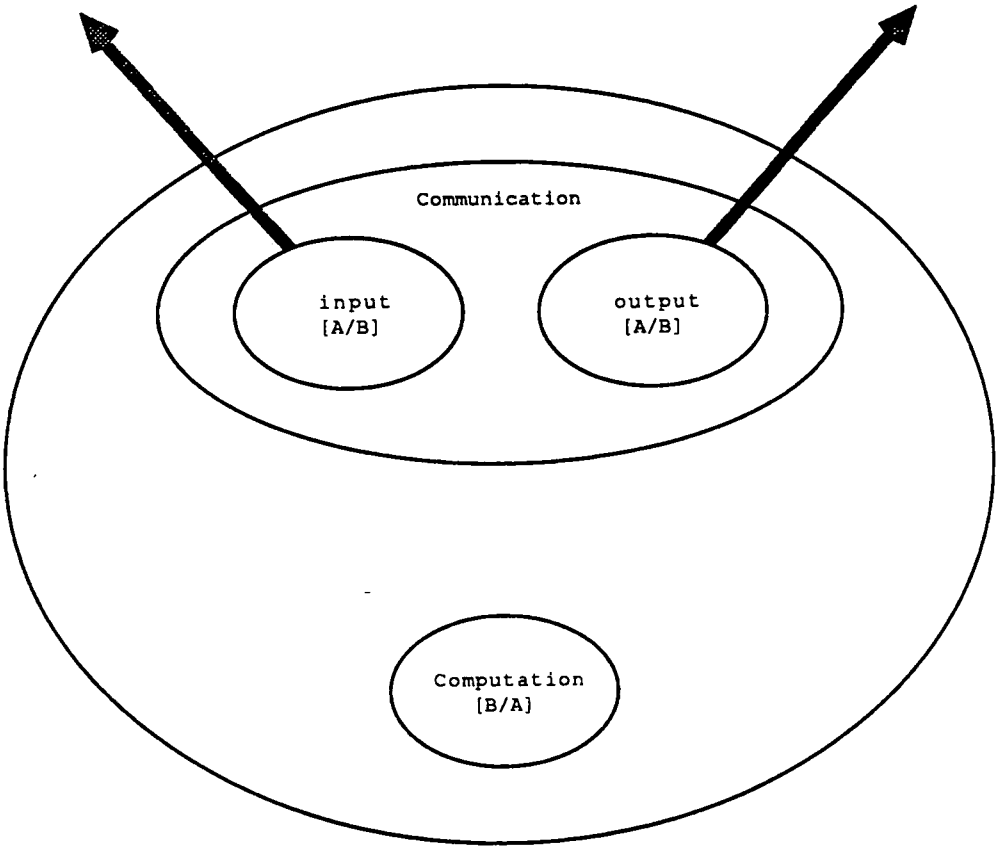


Fig 4.5 Harness Type I

```
PROC high.pass.B(CHAN OF ANY input,output)

  ... Declarations
  ... Initialisation

    PRI PAR
      PAR
        WHILE TRUE
          SEQ
            ... input.buffer
        WHILE TRUE
          SEQ
            ... output.buffer

      SEQ
        ... input.data.from buffer
        ... compute
        ... output.data.to.buffer

:
```

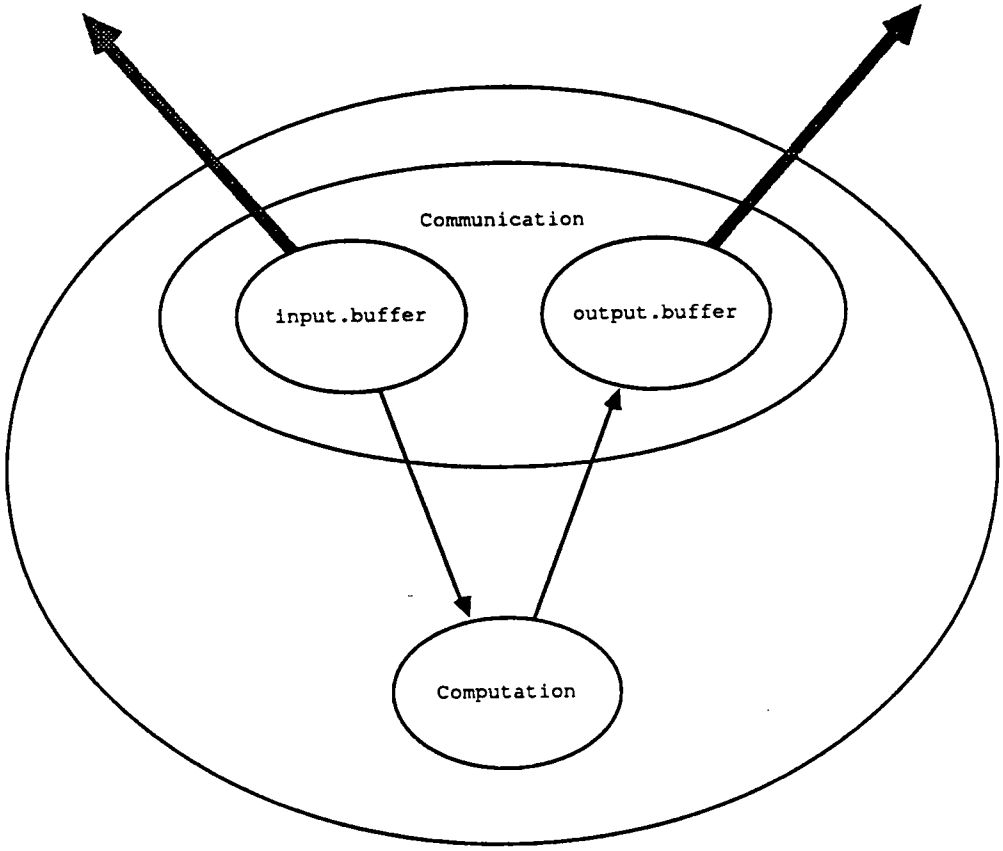


Fig 4.6 Harness Type II

Fig 4.8a
Comparison of Empirical Results
of Each Mapping, Harness Type I

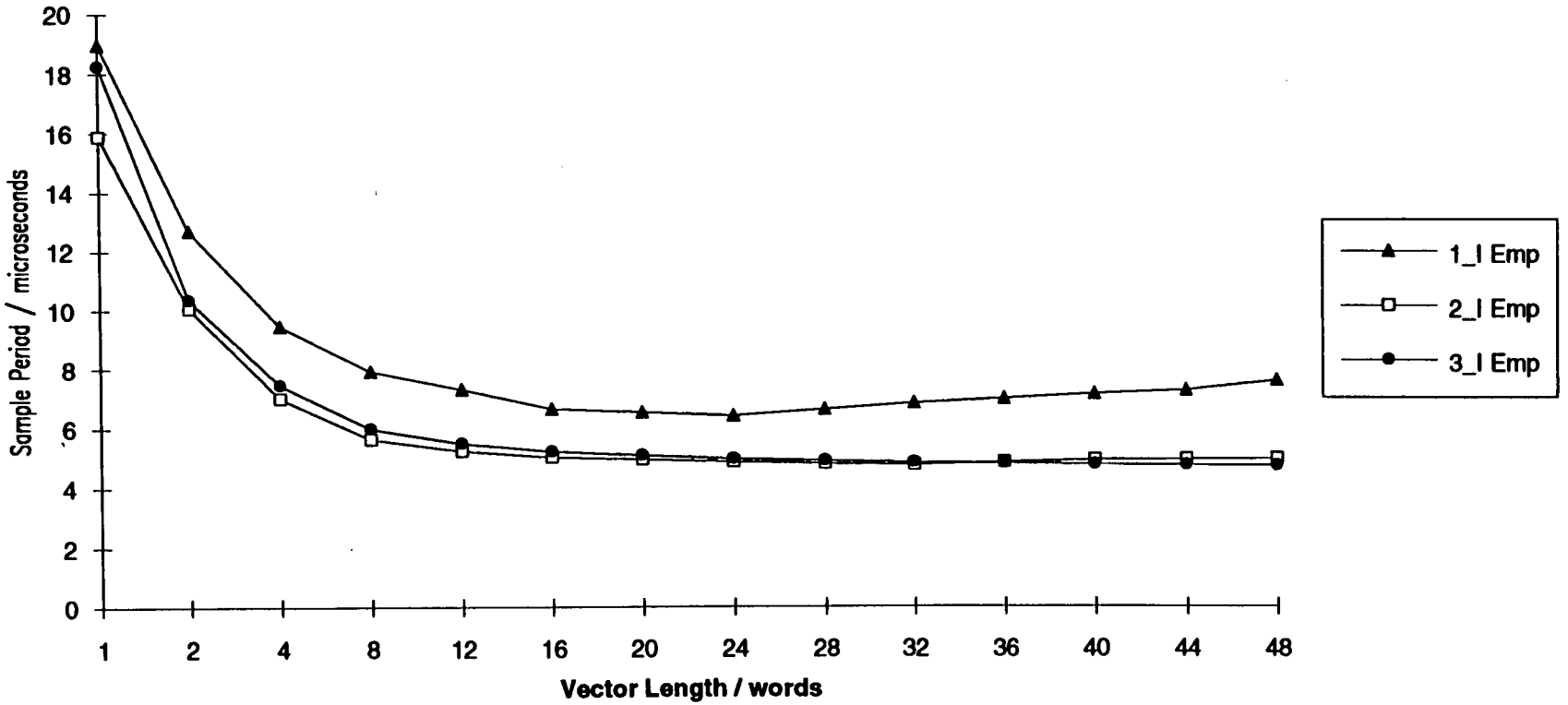


Fig 4.8b
Comparison of Empirical Results
for Each Mapping, Harness Type II

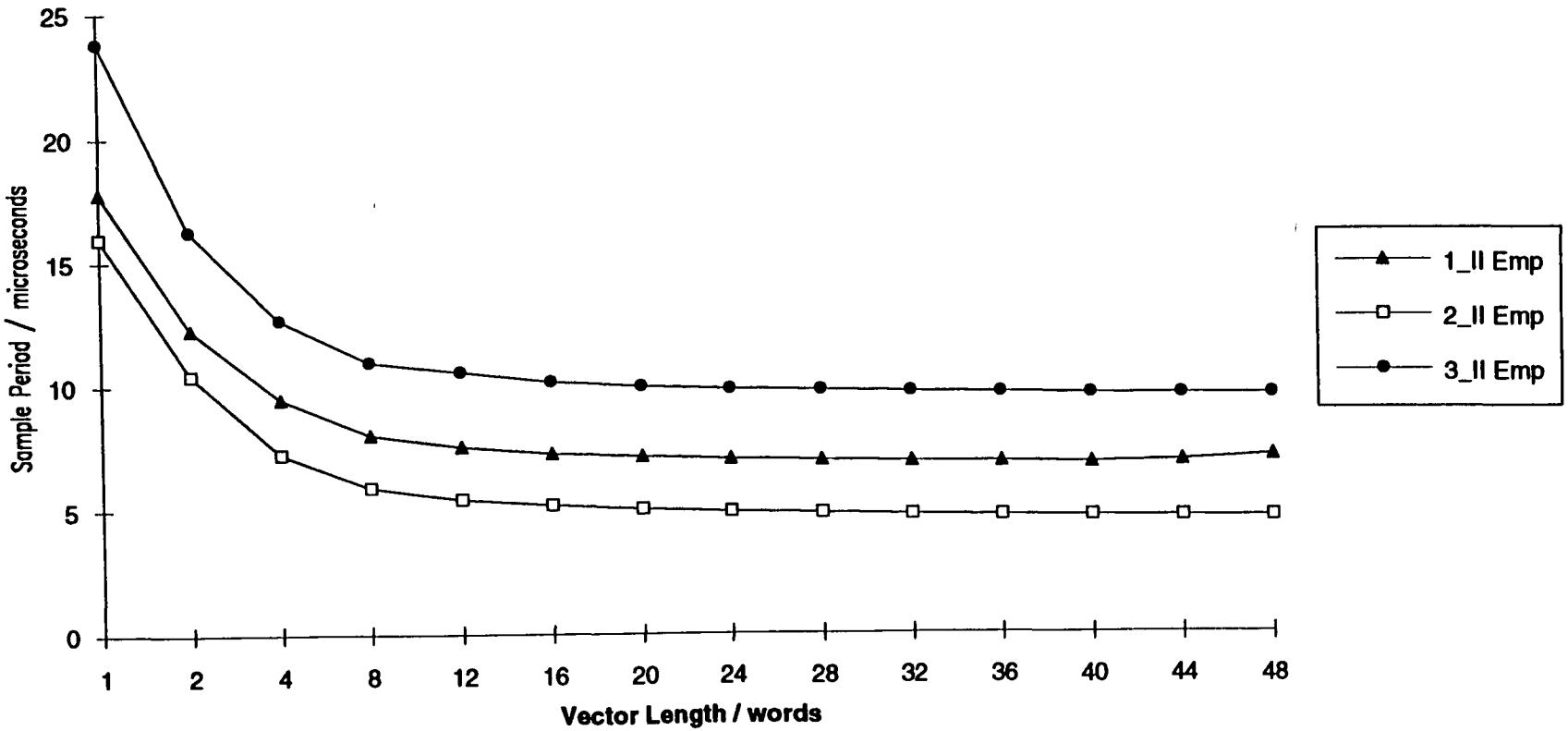


Fig 4.9a
Comparison of Empirical Results of
Each Harness, Single Processor Mapping

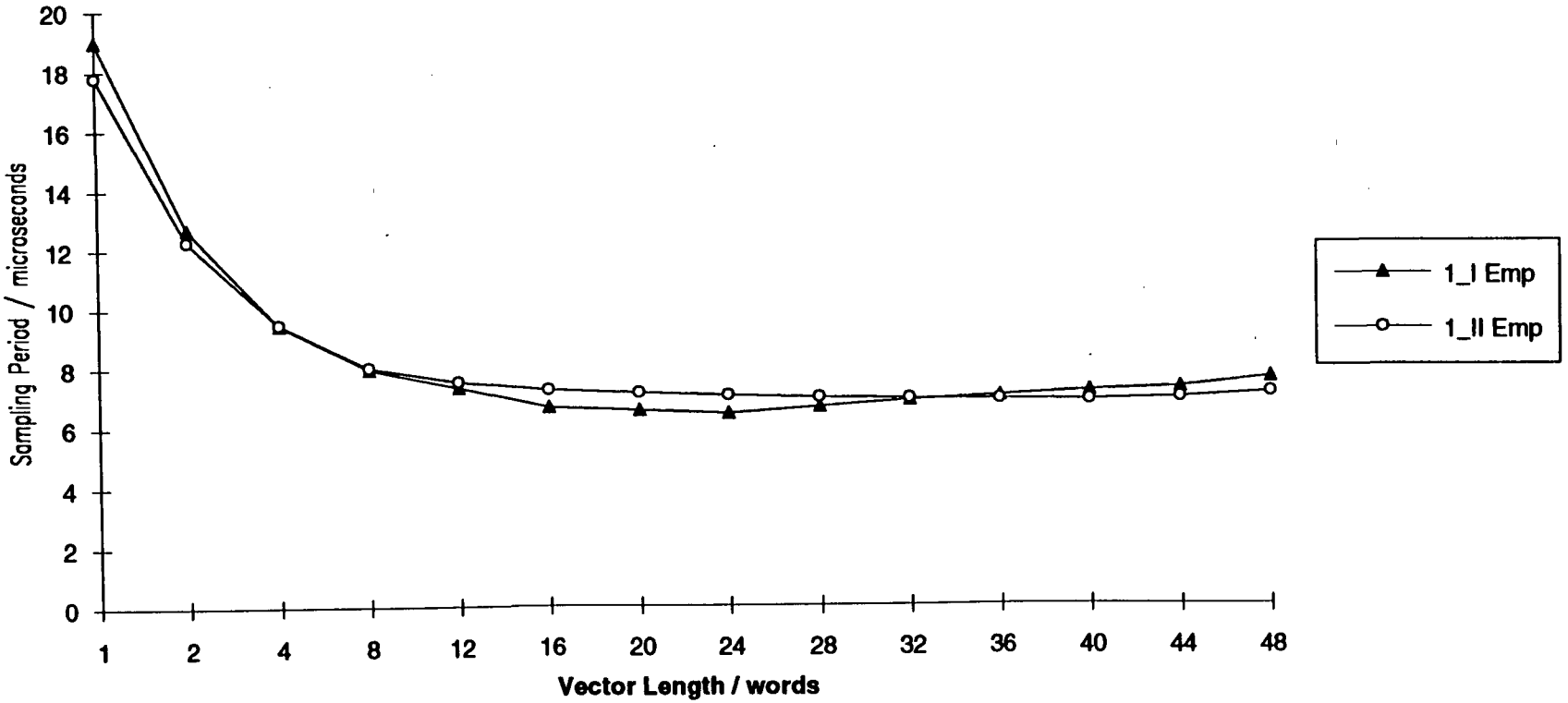


Fig 4.9b
Comparison of Empirical Results of
Each Harness for the Two Processor Mapping

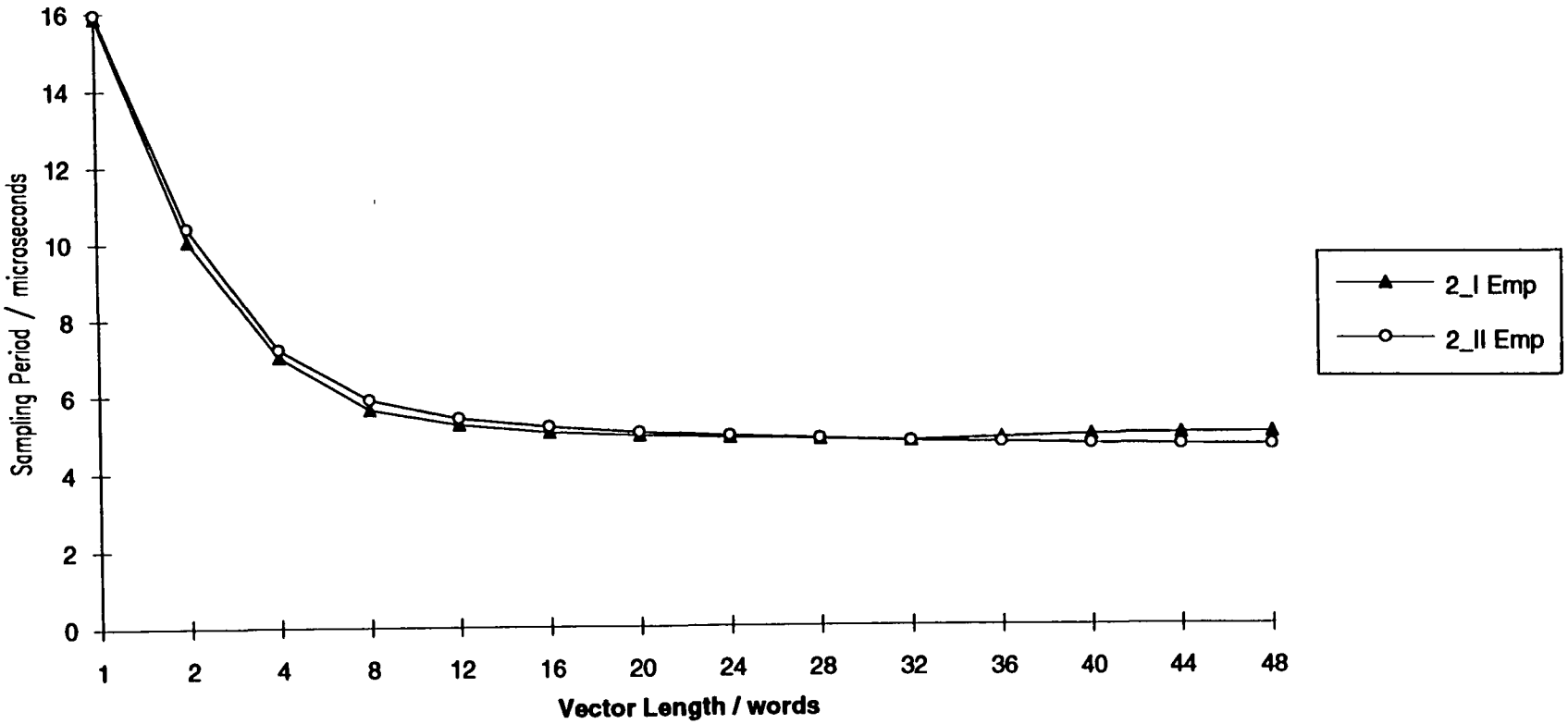


Fig 4.9c
Comparison of the Empirical Results of Each Harness of the Three Processor Mapping

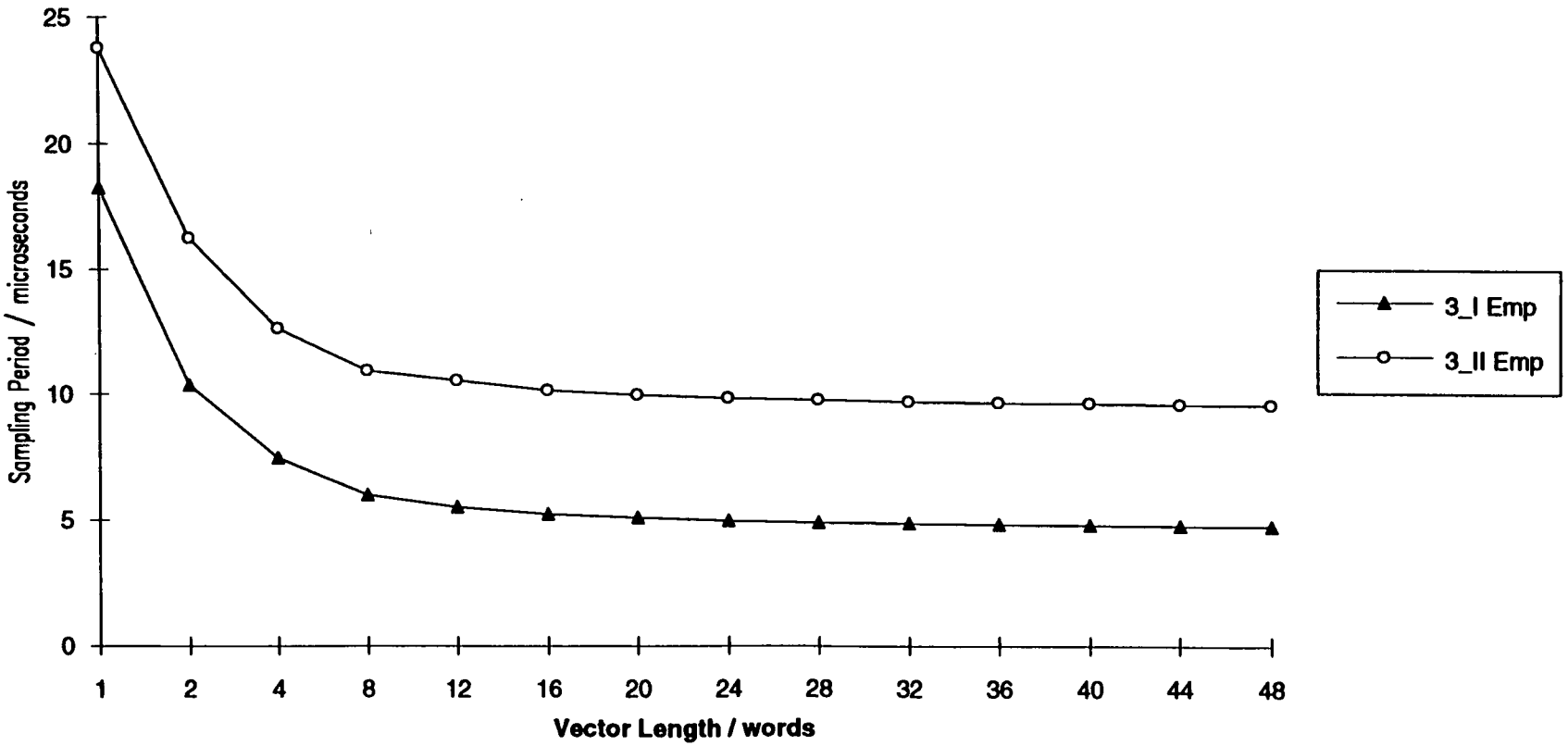


Fig 4.10a
Comparison of Theoretical and Empirical Performance for the Single Processor Mapping, Harness Type I

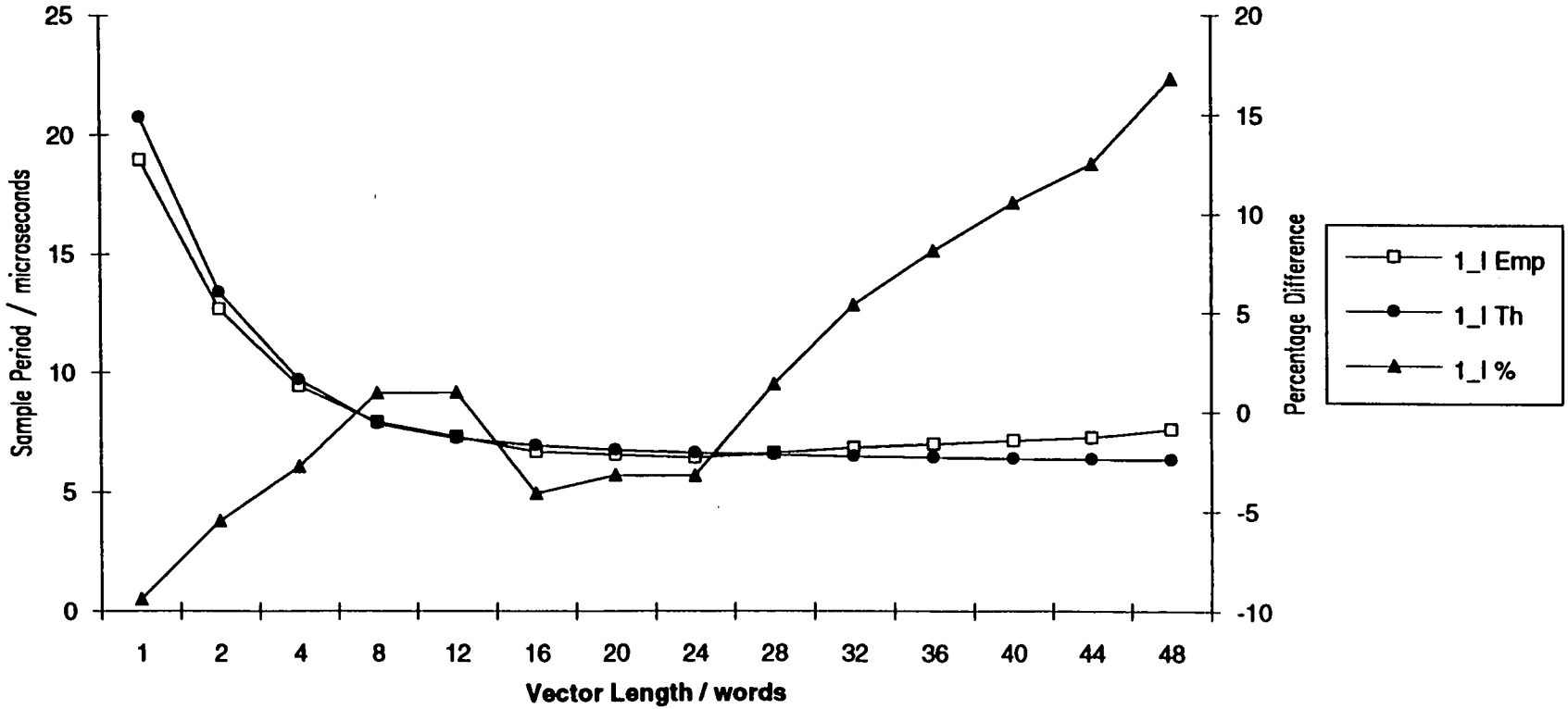


Fig 4.10b
Comparison of the Theoretical and Empirical Performance for the Two Processor Mapping, Harness Type I

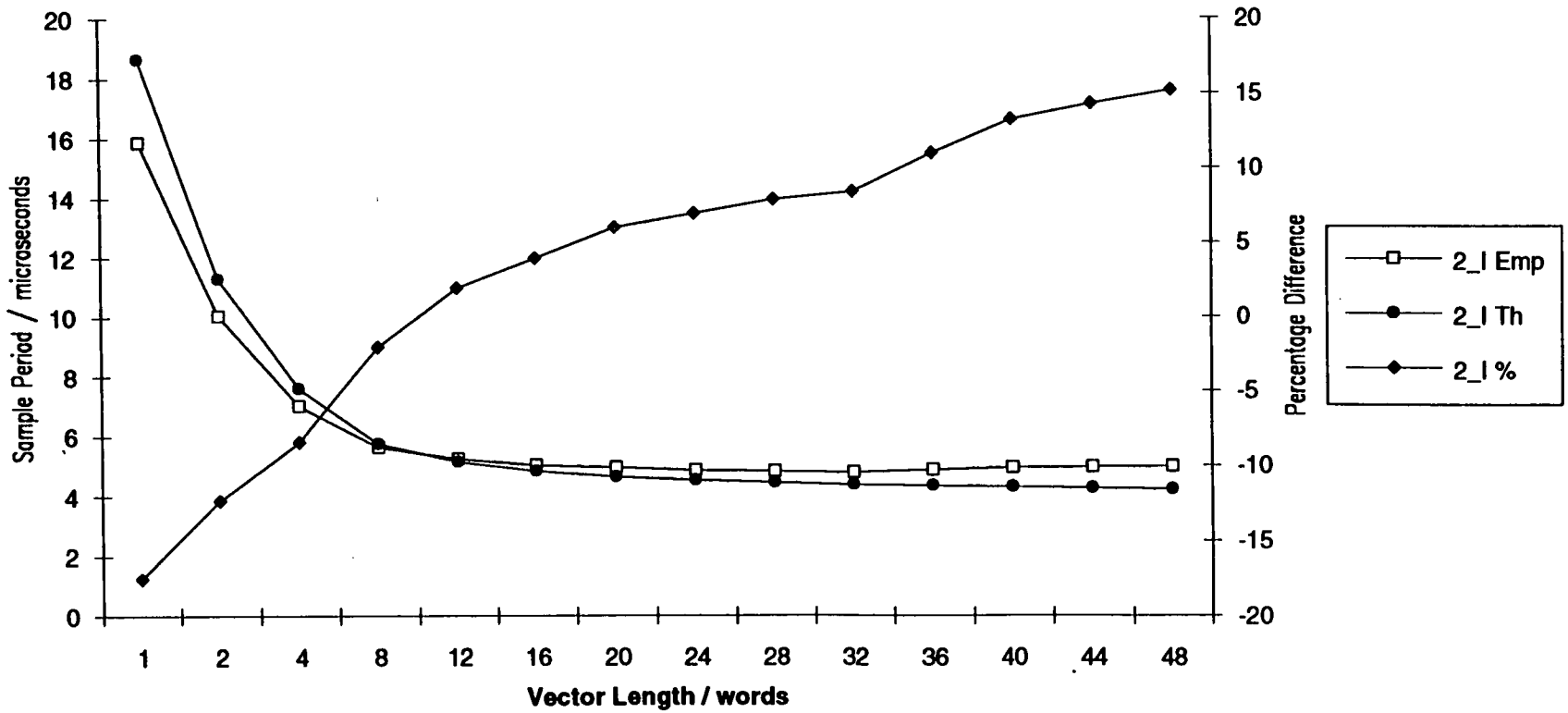


Fig 4.10c
Comparison of Theoretical and Empirical Performance for the Three Processor Mapping of Harness Type I

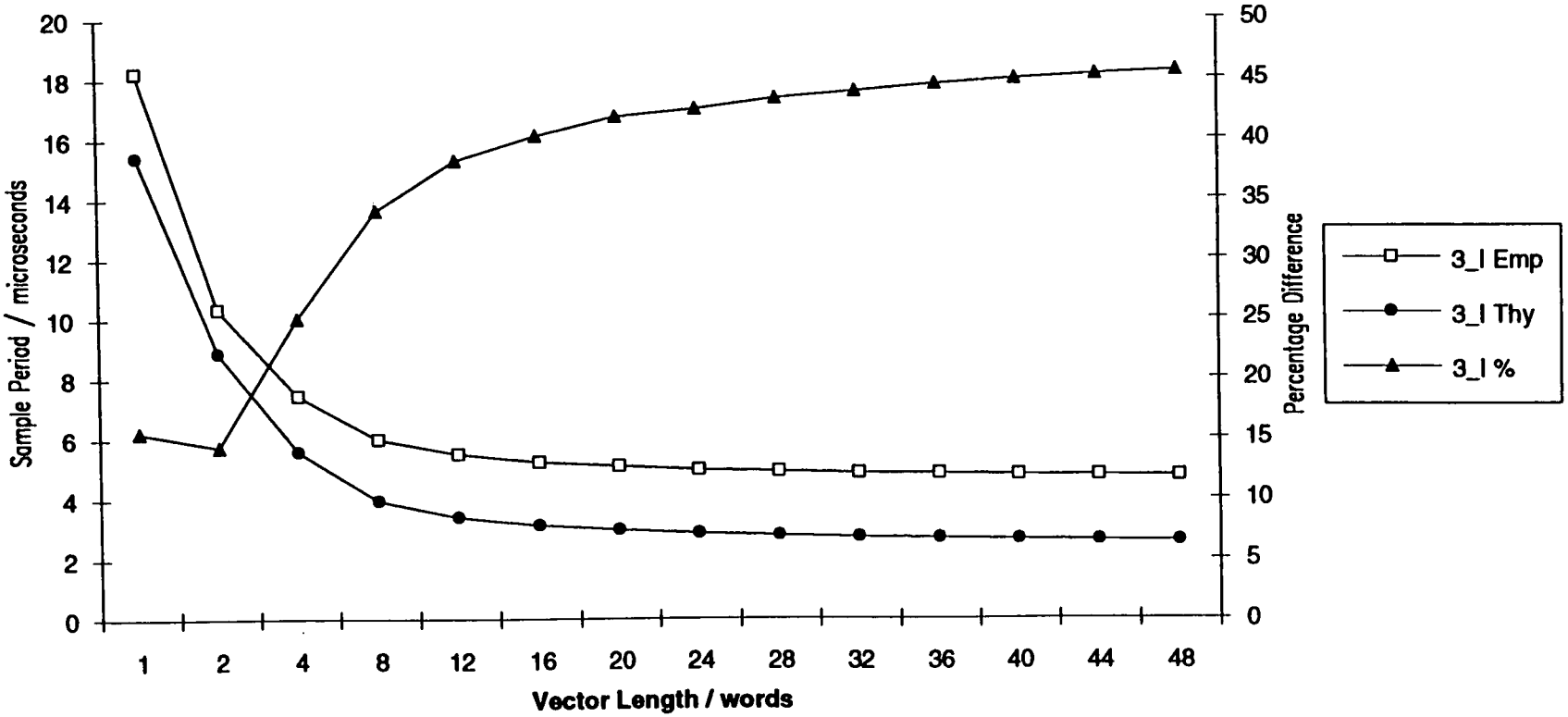


Fig 4.10d
Comparison of the Theoretical and Empirical Performance for the Single Processor Mapping, Harness Type II

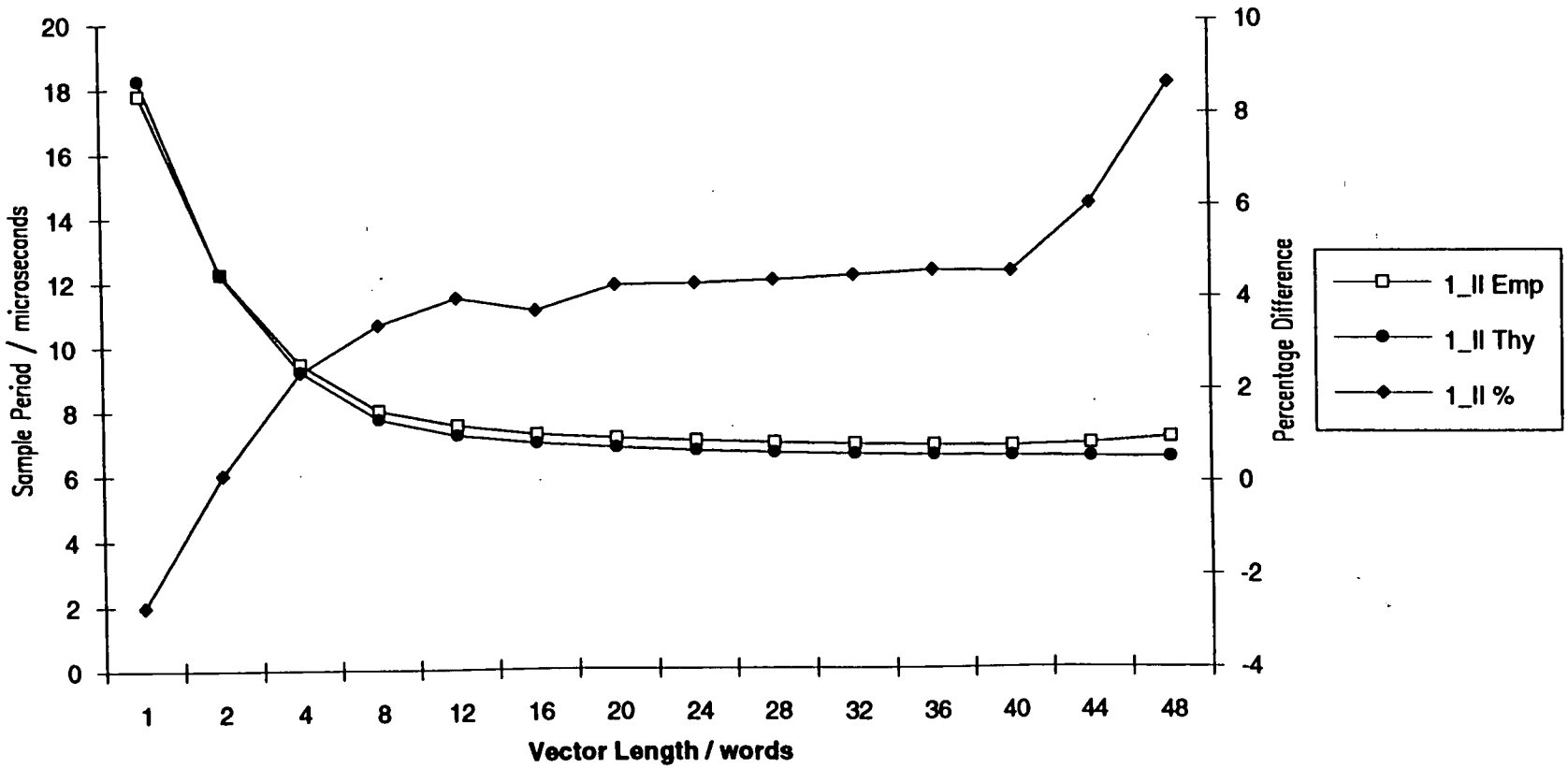


Fig 4.10e
Comparison of the Theoretical and Empirical Performance for Two Processor Mapping, Harness Type II

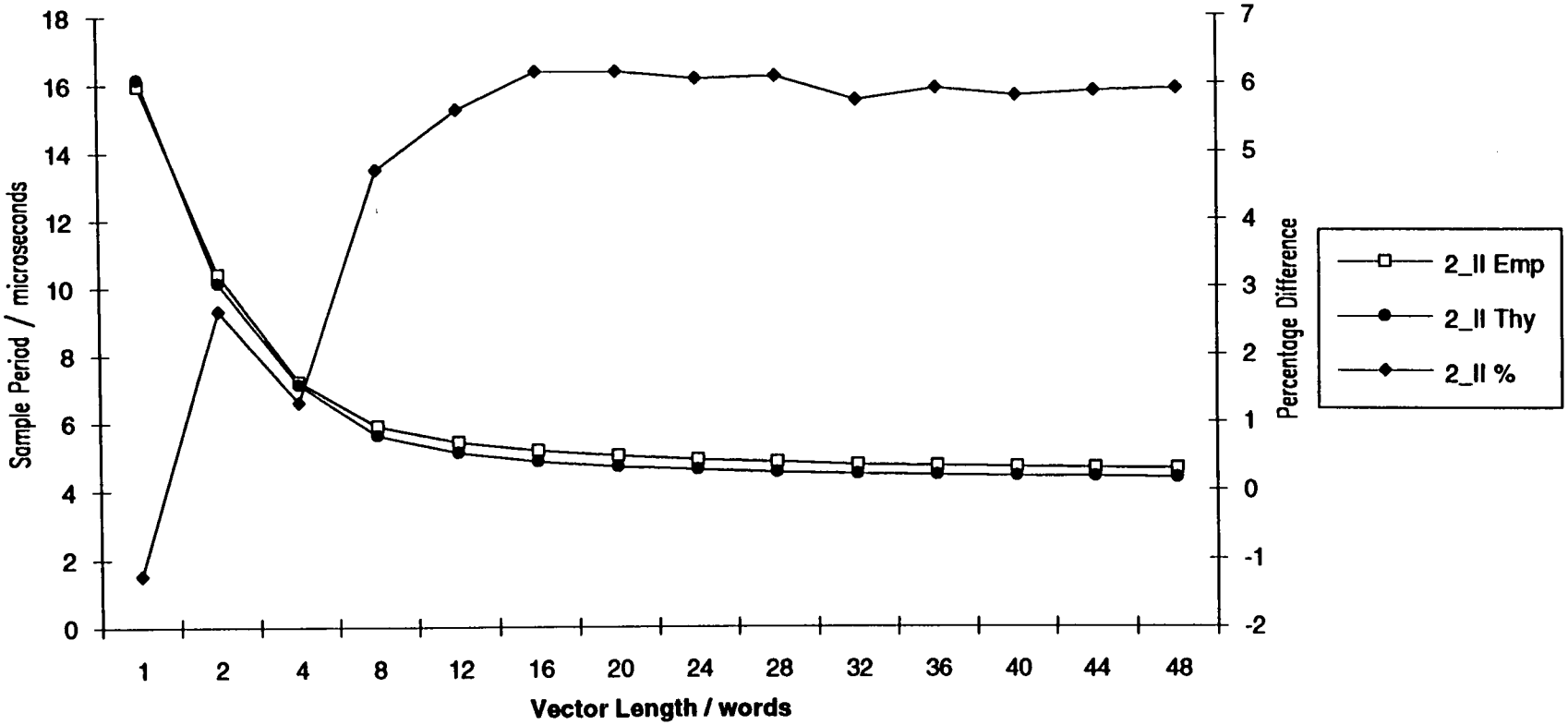
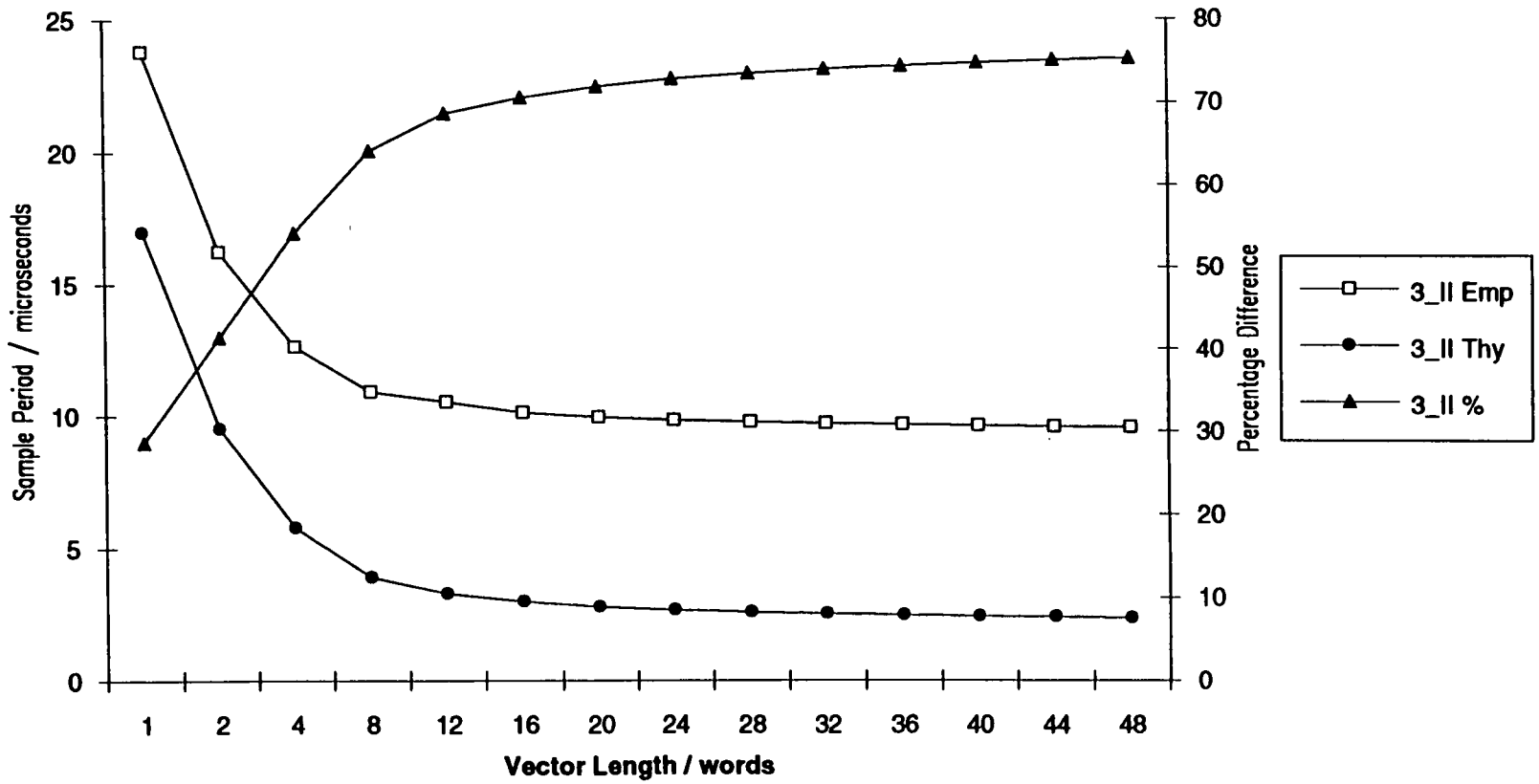


Fig 4.10f
Comparison of Theoretical and Empirical Performance for the Three Processor Mapping of Harness Type II



Chapter 5

The Motorola DSP56001

5.1 Introduction

This chapter describes the architecture and operation of the Motorola DSP56001 Digital Signal Processor (DSP) [42]. This device was the first programmable DSP marketed by Motorola, being released in 1987. The DSP56001 incorporates many design features associated with high performance DSPs — a parallel memory and multiple bus architecture, single cycle fractional multiplier and a comprehensive set of address registers [7], [40], [69]. The power of this particular device is enhanced by the modification of some of these features, and by the addition of new ones. The register addressing scheme, for instance, was the most versatile available at the time, allowing circular buffers and Fast Fourier Transforms to be easily implemented. The device also incorporates two on-chip communications peripherals which allow straightforward connection to "host" microprocessor systems and to devices such as analogue to digital (ADC) and digital to analogue (DAC) converters. A small amount of memory is incorporated on chip, and so the device may be thought of as a specialised microcomputer rather than as a microprocessor.

The architecture is based around three main parallel execution units — the Arithmetic and Logic Unit (ALU), the Address Generation Unit (AGU) and the

Program Controller (PC) — which are connected via multiple buses to parallel memory areas [70]. This results in a high degree of operational concurrency, which is reflected in the programming style.

Although an optimised C compiler is available, the highest performance may be attained only by using hand coded assembly language [71]. The assembler uses a time stationary coding method [40], which allows the programmer to maintain a high level of control over the sequence of operations. This approach contrasts with that of the "interlocking" style, used by Texas Instruments in the TMS320 range [40], which allows the programmer little direct influence on the sequence of internal operations.

The DSP56001, unlike the transputer, is not a general purpose processor; it has been designed specifically to process digital signals as efficiently as possible. The device is programmed in a far more direct and straightforward manner than the transputer, primarily because it has no facilities for supporting software defined parallel processes. The programmer may exert far greater control on the operation of the main execution units of this processor, which results in a far more hardware orientated programming style than the transputer. However, as the language represents a specialised sequential processor, programming is quite straightforward, once the architecture of the processor and time static coding are understood. This chapter is concerned primarily with the main architectural features of the DSP56001, and how they are controlled, rather than programming technique.

This particular device was chosen in preference to others for four main reasons. Firstly, it supports a 24 bit word format and so provides a larger dynamic range than 16 bit processors. Secondly, the architecture provides a high degree of operational concurrency and a single cycle non-pipelined MAC unit, which aids computational

efficiency. Thirdly, a byte wide parallel interface and both asynchronous and synchronous serial interfaces are incorporated as on-chip peripherals, allowing straightforward connection of external devices such as analogue to digital to analogue converters. Finally, a device simulator and a hardware development system were easily available [72], [73]. Floating point processors were not considered, as the devices available at the time, notably the NEC μPD77230 [74] and the AT&T DSP32 [45], could not provide performance comparable to that of fixed point DSPs, and their cost proved to be an inhibiting factor.

Section 2 provides an overview of the architecture of the device. The main functional elements are described more fully in Sections 3 to 8. The method of assembly programming is covered in Section 9, which also provides a gauge of processor performance. This chapter does not provide an exhaustive description of the DSP56001; the reader is referred to [70] and [71] for an in-depth treatment of the device. Diagrams labelled with † are taken from the Motorola literature.

5.2 Architectural Overview

The architecture of the DSP56001 is based around three main execution units — the data ALU, the address generator and the program controller — which operate concurrently, the memory areas and the interconnecting bus structure, all of which are contained on-chip. Additional units include both internal and external bus switches, a bus controller and serial and parallel communications interfaces (treated as memory mapped peripherals). A schematic representation of the architecture is given in Fig 5.1.

The memory structure is based on a modified Harvard architecture [7] — one program area and two data areas, denoted "x" and "y". The execution units and the

memory are connected via three address and four data buses. In order to reduce pin out requirements, these have been multiplexed to one address and one data bus, with appropriate control lines, for external memory access. A 16bit address word is utilised, allowing an external addressing limit of 64kwords for each of the program, x-data and y-data address spaces.

The ALU supports a 24bit fixed point fractional integer format. The accumulators impose no truncation errors after multiplication, and provide sufficient headroom for 256 consecutive overflow multiplications. Scaling and saturation arithmetic are supported.

The address generation unit allows simultaneous modification of two address registers, thus complimenting the access of the two data memory areas.

The peripherals may be configured either as general purpose i/o pins, or as a byte wide "host" interface and synchronous and/or asynchronous serial interfaces.

5.3 Buses

The DSP56001 contains three internal address and four internal data buses, see Fig 5.1, which concurrently move data and instructions between the main execution units while they are operating. Other elements of the internal bus structure include the internal bus switch and bit manipulation unit, and the external address and data bus switches. These will now be considered in turn.

5.3.1 The Data Buses

Data is passed around the chip using four bidirectional 24bit wide buses — the x-data bus (XDB), the y-data bus (YDB), the program data bus (PDB) and the global data

bus (GDB). Certain instructions cause the XDB and YDB to be concatenated, in order to produce one 48bit wide bus. The XDB and YDB connect the ALU to the x and y data memory areas. The PDB connects the program controller to the program data areas. Other data transfers, such as i/o transfers with peripherals, are carried out over the GDB.

This multiple bus structure, together with the extended Harvard architecture and execution pipelining, allow an instruction pre-fetch, two operand fetches and instruction execution to occur in parallel.

5.3.2 The Address Buses

Accesses to internal x and y data memory are addressed using the unidirectional 16 bit wide x address bus (XAB) and y address bus (YAB). Accesses to program memory are addressed using the 16 bit wide bidirectional program address bus (PAB).

5.3.3 The Internal Bus Switch

The internal bus switch allows the connection of any two data buses, without incurring a pipeline delay. This switch also incorporates a bit manipulation unit, as all data must pass through it. Bit manipulation is carried out on memory operands on the XDB, YDB and GDB.

5.3.4 The External Bus Switches

Although the DSP56001 may address each of its three internal memory areas simultaneously, allowing the same degree of access to external memory would increase the pin count of the device, resulting in a more expensive package. For this

reason only one address bus and one data bus are brought off chip. The four data buses are multiplexed into one by the external data bus switch, the three address buses are similarly multiplexed by the external address bus switch, which form part of the external memory interface (EMI). If only one bus requires access to external memory, then no performance penalty is incurred. If more than one bus requires external access, then bus arbitration must occur, and wait states inserted in the bus cycle by the bus controller.

5.4 The Memory Spaces

The DSP56001 utilises a modified Harvard architecture, accessing three separate memory spaces, the program space and the x-data and y-data spaces. These spaces may be forced into one of four configurations, controlled by the MA, MB and DE bits in the operating mode register (OMR), described in Section 5.7. The use of parallel memory areas is a typical feature of DSPs and aids performance by allowing more than one operand to be fetched in a single instruction cycle. A description of the individual memory configurations, shown in Fig 5.2, follows.

5.4.1 x-Data Memory

A maximum of 64kword of x data memory may be accessed, 256 words of which are contained on-chip. The on-chip x data static RAM area is 24 bits wide and occupies the lowest 256 locations of x memory space. An additional 256 words of internal preset ROM, containing A-Law and μ-Law expansion tables, may be mapped into locations `$100-$1FF` by setting the DE (Data Enable) bit to one in the OMR. Whenever the ROM is disabled, addresses `$100-$1FF` access external RAM locations.

The *on-chip* peripherals are mapped into external locations $FFC0-$FFFF of the x memory space, and may be accessed using the MOVEC instruction [70].

5.4.2 Y Data Memory

The y data memory is similar in size and operation to the x data memory. The ROM area contains a full sinewave look-up table, with *off-chip* peripherals being mapped into locations $FFC0-$FFFF.

5.4.3 Program Memory

The total addressable p-memory space is of similar size to the x- and y-data spaces, but differs significantly in its configuration.The on-chip program static RAM area is 24 bits wide and occupies the lowest 512 locations in p memory space. The program memory may be configured in one of four ways, shown in Fig 5.2, corresponding to the four operating modes of the device. The configuration is determined by the state of the MA and MB bits in the OMR.

Modes 0 and mode 2 utilise internal program RAM. These modes are similar, differing only in the location of the reset vector, which is placed at internal location $0 in mode 0, and at external location $E000 in mode 2.

In mode 3, the internal program memory is disabled and the processor exclusively accesses external program memory. Mode 1 is the special bootstrap mode that should be entered upon processor reset. In this mode, the special on-chip Bootstrap ROM is mapped into internal program memory space as read-only, and allows a program to be loaded either from the host interface or from a byte-wide ROM connected to the EMI.

5.5 The Address Generation Unit

The provision of multiple memory units, and their corresponding buses, in a processor architecture may well facilitate a high instruction throughput, but it also presents a problem. An instruction must specify an address for each of the memory areas that it wishes to access. For this device, then, an instruction would need to include three 16bit address fields. This would require either a long instruction word, increasing the overall cost and size of the memories and buses, or more cycles to access the instruction which would tend to decrease instruction throughput.

The solution, which is used by many processors, is to use "register indirect addressing". Special purpose address registers are used to hold the address of a word in memory. Instructions *refer* to a particular register, *indirectly* accessing a memory location. A dedicated arithmetic unit is usually incorporated to allow the address registers to be updated concurrently with bus and ALU operation. The number of registers used is comparatively small, requiring a shorter instruction. The DSP56001 possesses eight address registers, each with their associated offset and modifier registers. These allow complicated addressing schemes, such as modulo addressing (circular buffering, used in filters) and reverse carry addressing (bit reversal, used in Fast Fourier Transforms) to be implemented without incurring additional overheads.

The Address Generation Unit (AGU) is one of the main execution units on the DSP56001. This unit is used to calculate addresses used in register indirect addressing, and contains the registers used in this addressing mode. The unit is divided into two halves, and is capable of supplying two addresses every instruction cycle. This allows, for instance, two operands to be accessed, in x and y space, simultaneously. The unit consists of three main elements — the register files, the address ALU and the address

output multiplexer, Fig 5.3. These will now be considered in turn.

5.5.1 The Register Files

The AGU contains 24 registers, arranged as eight sets of register triplets. Each triplet consists of an address register, Rn, an offset register, Nn, and a modifier register, Mn. Each register is 16 bits wide and may be read or written by the GDB. When a register is read by the GDB, only the lowest two bytes are used, the most significant byte being zero extended. When the registers are written by the GDB, then only the two lowest bytes of the data word are used, the most significant byte being truncated.The eight register triplets are arranged as two banks of four. Each bank is controlled by its own address ALU.

The address registers are usually used to hold addresses that are used as pointers to memory, although they may be used to hold general data. Each address register may be used either as an input or as an output for its respective address ALU. One address register from each half of the AGU may be accessed simultaneously, allowing parallel data moves. Hence, if one half of the AGU is used to access x-data, and the other y-data, then two data operands, held in x and y data memories, may be accessed simultaneously. The manner in which any particular address register is changed depends upon the contents of its associated offset and modifier registers.

The offset registers are used to alter the contents of their respective address registers by some particular value. The offset may be applied in an incremental or a decremental fashion, and either before or after the address register is used.

The modifier register determines which of the three addressing modes the associated address register is subject to. The modes supported are linear, modulo and

reverse carry.

5.5.2 The Address ALU

The AGU incorporates two identical address ALUs, which operate on each group of register triplets. Each address ALU contains three full adders — an offset adder, a modulo adder and a reverse carry adder — each of which may act upon the contents of a specific address register and allow the three addressing modes to be implemented without additional operational overheads.

The offset adder can add plus or minus one, the contents of the associated offset register or the two's complement of the offset register, to an address register.

The modulo adder adds the output of the offset adder to a modulo value, M, or its complement, where M is the value stored in the associated modifier register.

The reverse carry adder operates in a similar fashion to the offset adder, the difference being that the carry is propagated in the reverse direction, and operates in parallel with the offset adder.

Each address ALU is capable of updating one address register in an instruction cycle. The combination of full adders allows linear, modulo or reverse carry arithmetic to be performed on the address register, depending on the contents of the associated modifier register. If the modifier register contains the value `$FFFF`, then linear arithmetic is used. If the modifier register contains `$0000`, then reverse carry arithmetic is utilised. If the modifier register contains any other value, *M*, then modulo *M-1* arithmetic is used.

5.5.3 The Address Output Multiplexer

The two banks of address registers present two 16 bit address values every instruction cycle. The address output multiplexer determines which bank is to be used to drive the XAB, YAB or PAB.

5.5.4 Address Register Indirect Modes

All main types of indirect addressing modes are available on the DSP56001, including pre-/post-increment/decrement by one or the offset value, modulo and reverse carry (bit reversal).

5.6 The Data Arithmetic and Logic Unit

The arithmetic and logic unit (ALU) is one of the three main execution units of the processor. The operation of the ALU lies at the heart of the power of the DSP56001. Incorporating a fast 24bit by 24bit multiplier with 56bit accumulation allows 256 consecutive overflows or underflows to occur with no degradation of accumulator accuracy. Furthermore, the two sets of input and output (accumulator) registers allow fast register to register or register to memory transfer, and a convenient local data store. The latter enables pipelining restrictions to be pre-empted, Section 5.11. The unit incorporates a non-pipelined multiply-accumulate (MAC) unit that is capable of operating with positive or negative accumulation, with or without convergent rounding, in a single instruction cycle.

The ALU, shown in Fig 5.4, consists of four input registers, a multiplier, an accumulator, rounding and logic units, two accumulator registers and shifting/limiting circuits. These are treated separately below.

5.6.1 The Data ALU Input Registers

The four general purpose 24 bit wide input registers, x0,x1,y0,y1 act as input buffers between the MAC unit and the data buses. They may be concatenated to form 48 bit registers, x1:x0 (x) and y1:y0 (y). The provision of these registers allows fresh data to be moved in over the data buses while the MAC unit operates on the previous data, allowing the MAC to operate continuously.

5.6.2 The Multiply Accumulator and Logic Unit

The MAC and logic unit, shown in detail in Fig 5.5, is the heart of the computational power of the DSP56001. It consists of a multiplier, an arithmetic and logic unit, convergent rounding circuitry and a data shifter. This unit operates in parallel with the bus circuitry, allowing continuous operation.

The x and y registers form the input of the multiplier, which executes 24 x 24 bit parallel fractional two's complement fixed point multiplications. The resulting full precision 48bit product is right justified and added to one of the accumulators.

The logic unit performs bitwise logic type functions on the ALU registers. There is a direct path from the output of the accumulators to the input of the MAC accumulator, incorporating a 56bit shifter that is able to perform single bit arithmetic or logical shifts to the left or right.

The convergent rounding circuitry is placed between the MAC accumulator and the accumulator registers.

5.6.3 The Data ALU Accumulator Registers

The ALU incorporates two 56 bit accumulator registers, a and b. These may

themselves be subdivided into component registers, a2:a1:a0, b2:b1:b0. The 48 bit product from the MAC unit may be stored in a1:a0 (b1:b0), whilst the additional 8 bits of a2 (b2),(the accumulator extension register), which is sign extended, allows 256 consecutive overflows or underflows to occur without loss of numerical accuracy. The individual register elements may also be accessed as unsigned registers.

5.6.4 The Shifter/Limiter Circuitry

The 56 bit accumulator registers are connected to the 24 bit data buses. Converting a 56 bit value to a 24 bit value obviously results in a loss of numerical accuracy. Usually, whenever the accumulator extension register is not in use, the 24 most significant accumulator bits (a1 or b1) are transferred to the bus. The 24 least significant bits are either truncated or rounded into the most significant portion before transfer.

Whenever the extension registers are in use, then simply transferring the contents of a1 (b1) may result in serious inaccuracies. For this reason, limiting circuitry has been included on the output of each accumulator register. This circuitry substitutes the maximum or minimum value representable by 24 bits for the value held in the accumulator register.

The individual constituent registers may be transferred as unsigned values by specifying them explicitly as an instruction operand.

Provision is also made for a shifting circuit on the output of each accumulator register. This is useful for applications involving scaling, such as digital filtering.

5.7 The Program Controller

The program controller is the third of the main concurrent execution units of the DSP56001. It consists of three sub-units, the program decode controller, (PDC), the program address generator, (PAG), and the program interrupt controller (PIC). The unit contains the hardware used to control and execute both long and short interrupt routines, in addition to the main status and control registers, a system stack, and registers used in the implementation of the hardware DO loops. The controller is at the heart of the instruction pipeline, and incorporates several features which enable highly efficient program execution — in particular the implementation of interrupts and hardware DO loops.

All registers are 16 bits wide, and may be read or written over the global data bus (GDB). As this bus is 24 bits wide, only the lowest 16 significant bits are valid. The 8 most significant bits of the bus are either forced to zero or are held in a "don't care" state. The sub-units and their operation are described below.

5.7.1 The Program Decode Controller

The PDC, Fig 5.6, contains the program logic array decoders, the state machines, the instruction latch and the backup instruction latch. This unit decodes the instruction held in the instruction latch and generates all the required pipeline control signals. The backup instruction latch is used to implement the repeat (REP) and jump (JMP) instructions.

5.7.2 The Program Address Generator

This sub-unit contains the program counter (PC), the stack pointer (SP), the system

stack (SS), the operating mode register (OMR), the status register (SR), the loop counter (LC) and loop address (LA) registers. The program address controller is totally independent of the data AGU, thus allowing data and instruction addresses to be calculated simultaneously.

The SS is a 15 x 32bit separate internal memory, divided into two banks of 15 x 16bit registers (system stack high and low), referenced by the SP. The stack is used to hold the contents of the PC and SR during subroutine calls and long interrupts. The stack is also used to hold the contents of the LA and LC during execution of the DO and REP instructions.

The OMR defines the current operating mode of the processor. Hence, it determines the memory partitioning scheme, and whether or not the internal data ROM areas are mapped into internal memory.

The SR is sub-divided into two 8bit registers, the mode register (MR) and the condition code register (CCR). The MR defines the state of the system, and is affected by reset, DO loop instructions, returns from interrupt and exception processing. The CCR defines the user state of the processor and is affected by data ALU operations and data limiting on the accumulator registers.

The operation of hardware loops is also controlled by this sub-unit. The REP instruction loads the LC with the number of times that the next instruction is to be repeated. The instruction needs be fetched only once, hence reducing the bus requirement, which may be important for programs requiring multiple external bus accesses. This instruction is not interruptible.

The DO instruction represents one of the most developed low overhead looping schemes available on any processor. The instruction loads the LC with the number of

times the loop is to be iterated and the LA with the address of the last instruction of the loop, and asserts the loop flag in the SR. These registers are also stacked, together with the address of the first instruction of the loop, prior to execution, allowing DO loops to be nested and repeated with no additional overhead. During execution, the contents of LA are compared with the contents of PC in order to determine whether the end of the loop has been reached. If this is the case, then the contents of LC are tested for one. If the test fails, then LC is decremented by one and the PC updated with the address of the start of the loop. If the test succeeds, then the loop has finished. The stack is popped and used to write the LC, LA and loop flag in the SR; and the instruction fetches continue as normal. These loops are interruptible.

5.7.3 The Program Interrupt Controller

The PIC arbitrates among all interrupt requests and generates the appropriate interrupt vector address. Four external and sixteen internal interrupt sources are processed by this sub-unit. Each interrupt possesses an associated interrupt priority level, that may range from zero (the lowest level, maskable) to three (the highest level, non-maskable). Most of the interrupt request sources may be assigned priority levels between zero and two, a few sources possess a priority level of three. An interrupt of higher priority level will be serviced in preference to one of lower priority level. The interrupt mask bits in the SR define the current processor priority. No interrupts with priority less than this level will be serviced. Level three interrupts are always serviced.

Each interrupt is vectored to a two word service routine at one of 32 fixed locations occupying the lowest addresses of program memory. An interrupt begins as a short interrupt, but may develop into a long interrupt. For a short interrupt, the

instruction(s) to be executed are held in the two vectored address words. For a long interrupt, these instructions specify a jump to subroutine, which may be any length. Short and long interrupts are depicted in Fig 5.7.

When an interrupt request is received and accepted, the exception processing state is entered. The instruction presently being decoded will be allowed to execute normally. The PC is then frozen as the PIC supplies the next two fetch addresses (the two interrupt vector program words), which form a short interrupt routine. No state information is saved during a short interrupt, which eliminates any overheads incurred by stack operations; the interrupt instructions are inserted in the regular instruction stream. If the short interrupt vector contains a subroutine call, then the more standard context switching long interrupt occurs. The stack stores the system state and return address, and the instruction pipeline is flushed in order to implement the subroutine. This obviously incurs performance overheads.

The provision of short interrupts, then, allows for short sections of code, such as those required to service the on-chip peripherals, to operate with no additional instruction pipeline delay. This is a very powerful feature, as data may be transferred to/from the on-chip peripherals without interrupting the instruction pipeline. Longer interrupt routines may still make use of the more traditional context switching long interrupt routines.

Two external interrupt request pins are available on the DSP56001. These are used to indicate interrupt requests for /IRQA and /IRQB, which are maskable interrupts. The /IRQA pin is also used to signal the NMI interrupt, although this is indicated by a super-voltage of 10V, and so has not been designed for prolonged or frequent use.

5.8 The External Memory Interface (Port A)

The external memory interface (EMI), or Port A, is used to connect the DSP56001 to external memory devices such as additional RAM, ROM or EPROM. The three internal address buses are multiplexed onto one external address bus. Similarly for the internal data buses, except that the global data bus is not brought out externally. The external bus switches determine which of the buses are passed externally at any one time. Although multiplexed, the bus operates at the same rate as the internal buses, an important consideration as it allows one of the internal data areas to be extended off-chip without incurring a performance degradation.

The associated external bus control unit provides signals that indicate which of the data spaces are being accessed. This unit also provides read enable and write enable lines.

Two bifunctional lines are available, their mode being selected by the operating mode register. These are the bus request/bus grant signals, used for external DMA access, and the bus wait/bus strobe signals, which insert wait states into the present bus cycle and may be used in shared memory systems.

The external bus interface is capable of operating at full speed, and so incurs no performance penalty when only one external memory area is required per instruction. Whenever two or three external areas need to be addressed, the control logic arbitrates and orders the accesses accordingly, resulting in an overall decrease in performance.

Up to fifteen wait states may be programmed into the EMI, enabling slower (and hence less expensive) memory devices to be used. Each address space may be programmed with a different number of wait states, defined by the bus control register

5.9 Port B

Port B is implemented as one of the two on-chip peripheral communications units. It may be configured either as a general purpose input/output interface, in which the action of individual pins may be user defined, or as a byte wide "host" interface, which operates in a similar manner to a standard microprocessor interface. The port is accessed by the DSP56001 through memory mapped peripheral registers, allowing a rapid transfer of data by using short interrupts. Port B operates concurrently with the other main execution units, thus providing a powerful communications mechanism. The two operating modes will now be considered in some detail.

5.9.1 The General Purpose I/O Interface

The general purpose i/o interface consists of fifteen pins which may be individually configured to act as inputs or outputs. In this configuration, the port may be thought to consist of three memory mapped registers, residing in the internal peripheral memory area. These are the port B control register (PBC), which determines the configuration of the interface, the port B data direction register (PBDDR), which determines which pins are inputs and which outputs, and the port B data register (PBD).

Port B is a memory mapped peripheral, and so the MOVEP instruction may be used to access its locations. This instruction is slower than the normal MOVE instruction, but as it allows memory to memory transfers, it is ideal for use within fast interrupt routines.

A hardware strobe is not provided. Hence, if an external strobe signal is required, it must be generated in software by toggling one of the output pins.

The i/o pins are latched. This has the consequence that the data is not actually placed onto the output pins until an instruction cycle after the instruction appears in the code. This is an important consideration if port B is to be synchronised with port A activity.

As the port may be written or read every instruction cycle, the maximum data transfer rate using this configuration is in excess of $150Mbits^{-1}$.

5.9.2 The Host Interface

The host interface is an asynchronous, byte wide, full duplex, double buffered port, designed to be connected directly to a host microprocessor or DMA controller. It behaves, as far as the host is concerned, very much like static RAM. The configuration consists of two banks of registers, one which may be accessed by the DSP56001, the other by the host processor. The host registers are mapped into peripheral memory space.

Not only does this interface allow data transfer between the DSP56001 and a host processor, but it also allows the host processor to force interrupt routines within the DSP56001. This latter option is very powerful and allows the host to control the operation of the DSP, or to inspect its state for debugging purposes.

The interface may be configured to transfer 8, 16 or 24 bit words. The maximum burst data transfer rate is $8Mbytes^{-1}$, with an interrupt driven transfer rate of $1.71Mwords^{-1}$, the maximum allowable with a 20.5Mhz processor.

5.10 Port C

Port C consists of nine pins. Three of these pins may be configured either as a general

purpose i/o interface, or as the serial communications interface (SCI). The remaining six pins may be configured either as a general purpose i/o interface or as the synchronous serial interface (SSI). This port is therefore very versatile. The configurations will now be considered separately.

Port C is implemented by the second of the on-chip peripheral communications units, and like Port B operates independently of the main execution units and is accessed by memory mapped peripheral registers. This unit may be configured either as a general purpose input/output port or as both asynchronous and synchronous ports. Although the data transfer rate is not particularly high, this interface is useful for connection to devices such as analogue to digital converters. An additional feature of this port is that it is capable of operating in a time division multiplexed mode, allowing up to 32 DSP56001s to be interconnected.

5.10.1 The General Purpose I/O Interfaces

The two groups of pins constituting port C may be separately configured as general purpose i/o pins in much the same way as the general purpose configuration of port B. The configuration mode of the pins is controlled by the port C control register (PCC), the port C data direction register (PCDDR) determines which pins act as input and which as output, and the port C data register (PCD) is used to hold the data. These registers are memory mapped into peripheral memory space, as in the case of port B.

The MOVEP instruction may be used to transfer data in the same way as for port B. Similarly, all timing and strobe constraints applicable to the general purpose interface of port B also apply here.

5.10.2 The Serial Communications Interface

The serial communications interface (SCI) consists of three pins — transmit data (TXD), receive data (RXD) and serial clock (SCLK). The transmit and receive sections are separate and may operate asynchronously. Many synchronous and asynchronous protocols, including RS232, are supported, including a wake up on idle and wake up on address bit multi-drop modes, for use in multi-processor configurations. The transmit and receive baud rate clocks are programmable, and may act as timers.

The SCI is controlled and configured by seven registers, held in a contiguous area of peripheral memory space. These are the SCI control register (SCR), which controls all the operational features of the interface, the SCI status register (SSR) which indicates the present state of the interface, the SCI clock control register (SCCR), three data transmit and receive registers and the SCI transmit data address register.

The interface may operate at up to 320 $kbits^{-1}$ in asynchronous mode, and up to 2.56$Mbits^{-1}$ in synchronous mode (20.5Mhz device) [70].

5.10.3 The Synchronous Serial Interface

The synchronous serial interface (SSI) consists of six pins, and offers a means of high performance full-duplex serial communication. As in the SCI, the receive and transmit sections are separate and may operate asynchronously. This interface is very versatile, and many interface protocols are supported. Control is provided by means of four registers held in peripheral memory space. These are the SSI control registers (CRA and CRB), the SSI status/time slot register (SSISR/TSR) and the SSI receive/transmit

data register (RX/TX). Data wordlength is selectable as 8, 12, 16 or 24 bits.

This interface may operate in one of three modes. The normal mode is used to periodically transfer data, at the rate of one word per period. The network mode also transfers data periodically, but allows for up to 32 time slots per period. This mode may be used to build time division multiplexed systems — a very useful feature. The on-demand mode may be used to transfer data whenever it is available, and is non-periodic in nature.

The SSI is capable of transferring data at a rate of $5Mbits^{-1}$, and is ideal for connection to devices such as analogue to digital to analogue converters.

5.11 Programming

In common with most other currently available DSPs, the DSP56001 may be programmed using either a native assembly language, or a C compiler. Although the earlier versions of the C compiler produced lamentably inefficient coding, the more recent versions offer significant performance increases. Maximum operational performance may still be attained only by using assembly language, however.

The Motorola DSPs use a time stationary coding method as the basis for their assembly language, compared to the interlocking method used by Texas Instruments, and data stationary method used by AT&T for their floating point DSPs, [7], [40]. In time stationary coding, a line of code specifies the operations that are to occur simultaneously in an instruction cycle. Time stationary coding highlights the concurrent operation of the main execution units of the DSP56001, as, in comparison to the other approaches, it emphasises parallelism rather than pipelining. In other methods, the effects of pipelining, in particular delays caused by resource contention,

are largely hidden from the programmer, and performance may well suffer as a consequence. Time stationary code, although it may appear complex at first sight, allows the programmer to manipulate the instruction pipeline and interleave memory accesses to provide the highest possible performance. Any pipeline hazards or resource contentions are flagged by the assembler, allowing the programmer the opportunity to restructure the programme, or to insert a delay (a NOP instruction) between the contentious lines of code. This latter approach is automatically applied by the processor in data stationary and interlocking code.

The DSP56001 assembly code format consists of an instruction field and two parallel data move fields. An instruction pre-fetch, instruction execution and two data transfers may occur within a single instruction cycle. Furthermore, the provision of a dedicated address generation unit enables two address registers to be updated during any instruction. An example line of code is

```
MACR    x0,y0,a    a,x:(R0)+N0    y:(R4)+,y0
```

which multiplies the values held in data registers x0 and y0 and stores the result in accumulator a, simultaneously transferring the previous value held in a to the x-data memory location specified by R0, post incrementing R0 by the amount held in offset register N0, and transferring the value held in the y-data memory location specified by R4 into data register y0, post incrementing R4 by 1.

A consequence of the number of registers contained in the ALU and the control offered by time stationary coding is that pipeline hazards or resource contention may be avoided by transferring values into an ALU register many

instructions before it is required. This is an example of both the potential complexity of time stationary coding and its potential performance benefits.

In order to obtain maximum performance, it is important to ensure that most memory accesses are made to internal memory areas, and that only one external memory access is required in any one cycle. Providing that these criteria are met, then most of the instructions in the DSP56001 instruction set will operate in a single cycle. The parallel data move capability is especially useful for applications such as digital filtering and image convolution — the DSP56001 is capable of implementing a biquadratic filter section in only four cycles, which is the minimum possible for a single multiplier device.

The devices used in this work were running at 20.5MHz. More recently, 27MHz and 40MHz parts have become available.

5.12 Summary

This chapter has described the architecture of the DSP56001 and shown it to be a very powerful digital signal microcomputer. The structure of the architecture exhibits a high degree of operational concurrency, allowing the device to execute an instruction, perform an instruction pre-fetch, access two data areas and perform updates on two address registers in a single cycle. Such features enable the DSP56001 to implement stock DSP algorithms highly efficiently.

The device also incorporates a powerful address generation unit (AGU) containing eight sets of address registers and allowing circular buffering and bit reversed addressing schemes to be used with no additional loss in performance. Together with the arithmetic and logic unit (ALU), the AGU forms the computational

powerhouse of the processor.

The two on-chip peripheral interfaces — port B and port C — add to the versatility of the device. These memory mapped interfaces may be configured in a variety of ways. Port B is suited to communication with an external host processor, allowing the host to execute predefined interrupt routines on the DSP in addition to the more usual bidirectional data transfer. The two serial interfaces of port C are versatile and capable of operating at high speeds. The SSI, in particular, is capable of operating in a network mode, allowing the processor to participate in a multi-processor based time division multiplexed serial communication scheme. The SSI is also easily connected to ADC/DAC systems.

The device may be programmed either in C or in assembly language. The assembly language, which is based on time stationary coding methods, should be used if maximum performance is required. The time stationary coding method, while perhaps less "user friendly" than interlocking or data stationary methods, does allow the programmer more control over the device, producing optimal code. Assembly language coding of the DSP56001 results in very efficient and compact code.

Although the transputer is more of a general purpose processor than the DSP56001, both have been designed with efficient code execution in mind, although they incorporate different design methodologies. The transputer uses a relatively small and efficient instruction set, building up its instructions from individual bytes. Four bytes are accessed in a single bus cycle, reducing the overheads attached to instruction pre-fetching. Operations are performed on a small number of registers, rather than on elements of memory. This approach to increasing performance is typical of RISC-like architectures. An area of internal memory is provided, which may be accessed every

machine cycle.

The DSP56001 also utilises internal memory, but in the form of a modified Harvard architecture. One program and two data memory areas, together with their associated buses, are provided on-chip. This allows up to three 24bit words to be transferred in a single instruction cycle. The DSP56001 also possesses a relatively small instruction set, but this is more a consequence of the specialist nature of the device rather than any RISC based performance enhancement. Due to its high memory bandwidth, the DSP56001 does not need to utilise compound instructions to help improve operational efficiency.

There are two major differences between the two devices. The transputer incorporates a microcoded process scheduler and autonomous link engines to provide efficient implementation of parallel programs and inter-processor communications. The DSP56001 utilises a highly optimised ALU which incorporates a non-pipelined MAC unit, allowing a 24bit by 24bit multiplication (with 56bit accumulation) to be carried out every instruction cycle. Evidently, then, the DSP56001 would easily outperform the transputer when executing multiplication intensive applications, as the DSP is capable of multiplying almost forty times faster than the transputer (for equivalent clock speeds). However, as the transputer was designed to form multi-processor networks, it is far more efficient than the DSP at inter-processor communication.

The transputer may be programmed in a variety of languages, none of which, understandably, provides the performance offered by assembly programming. However, transputer assembly language is complex, as parallel processes must be defined, and performance is not easily predicted. The DSP56001 is also optimally programmed in assembly language. Although this is perhaps one of the more complex

forms of DSP assembly language, the programs produced are relatively straightforward compared to those of the transputer. Furthermore, the behaviour of the code is more straightforward, the performance of the code being easily determined by analytical means.

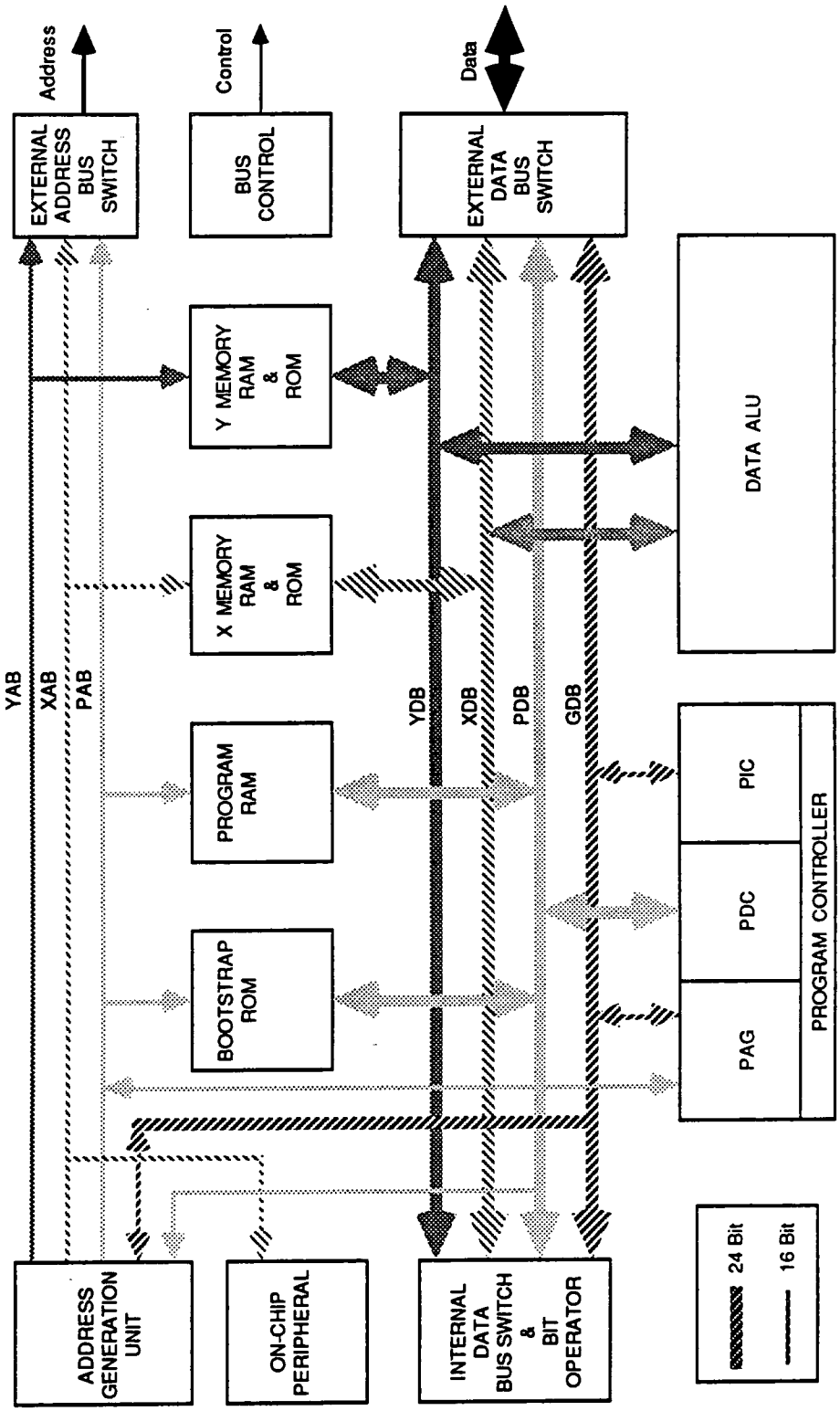


Fig 5.1 Block Diagram of DSP56001 Architecture †

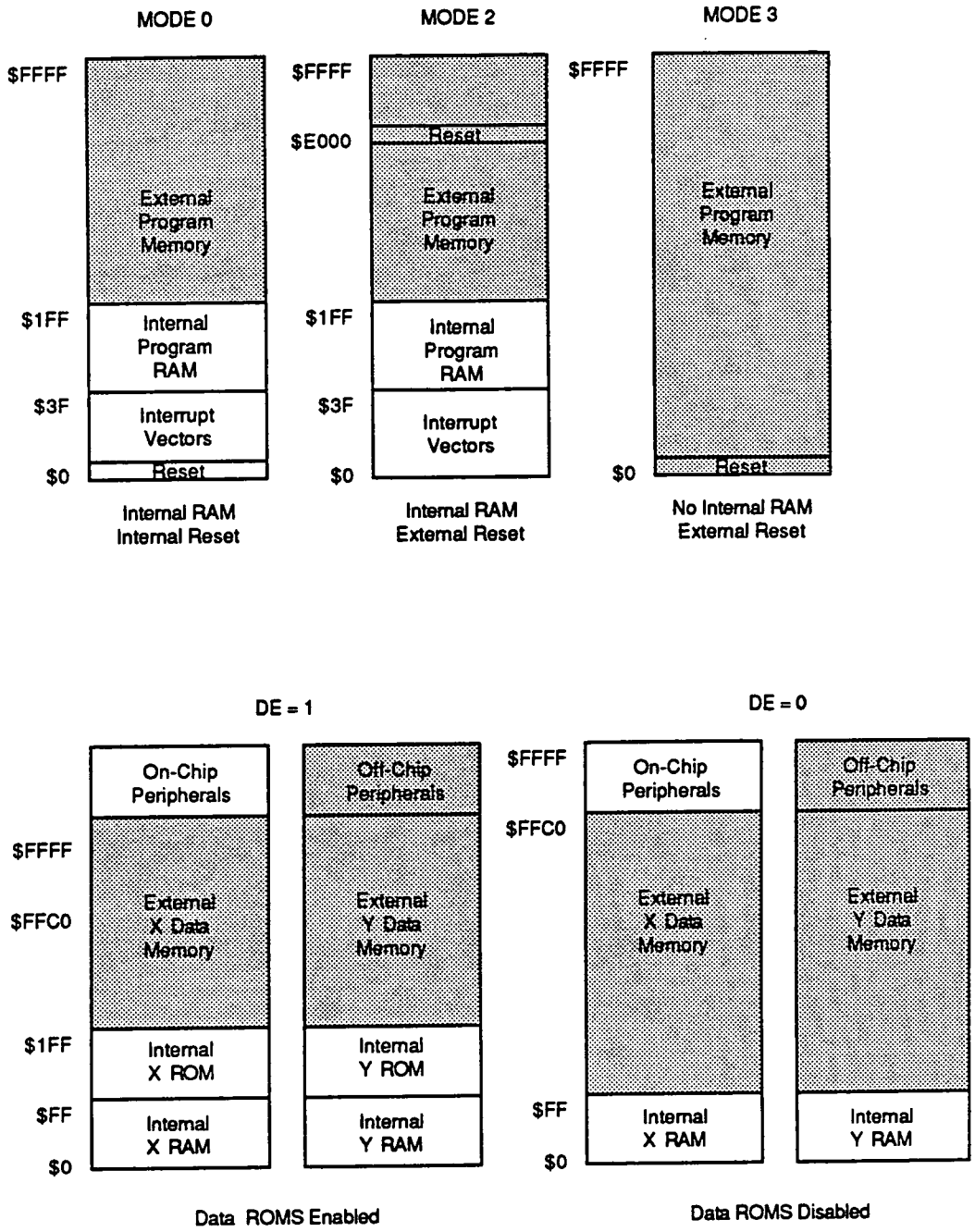


Fig 5.2 DSP56001 Memory Maps†

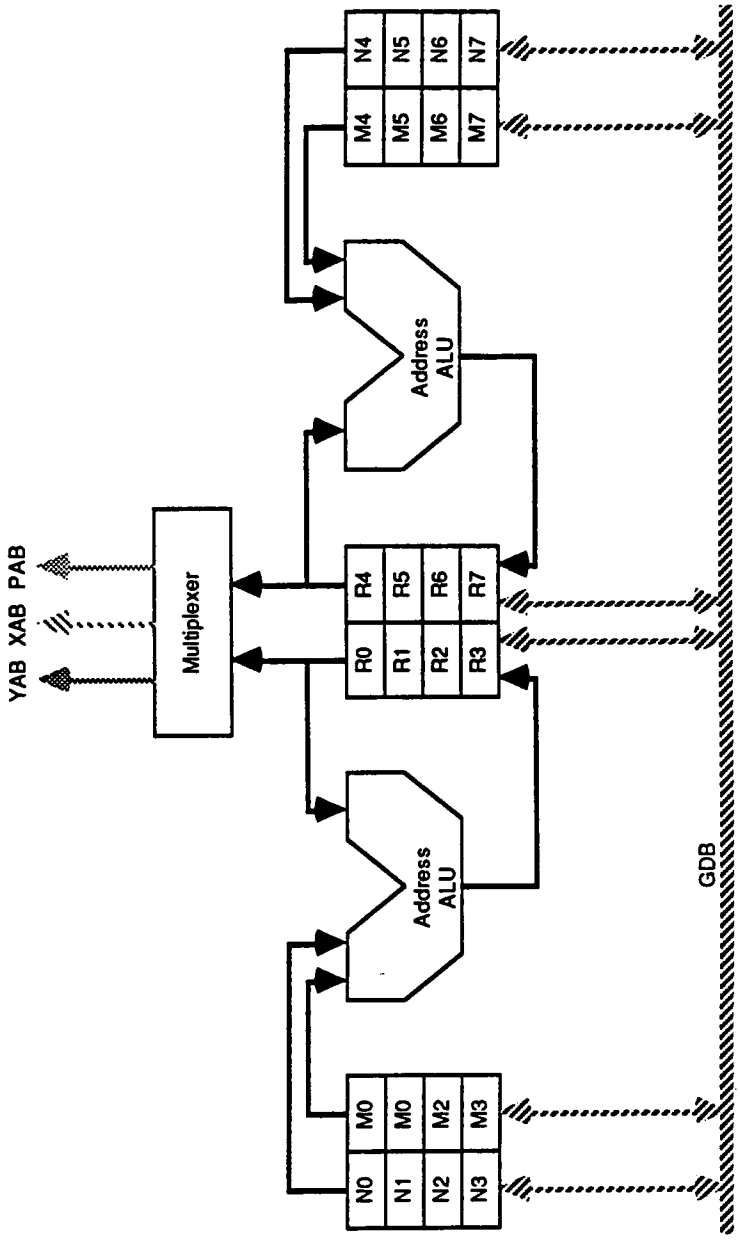


Fig 5.3 Architecture of the Address Generation Unit †

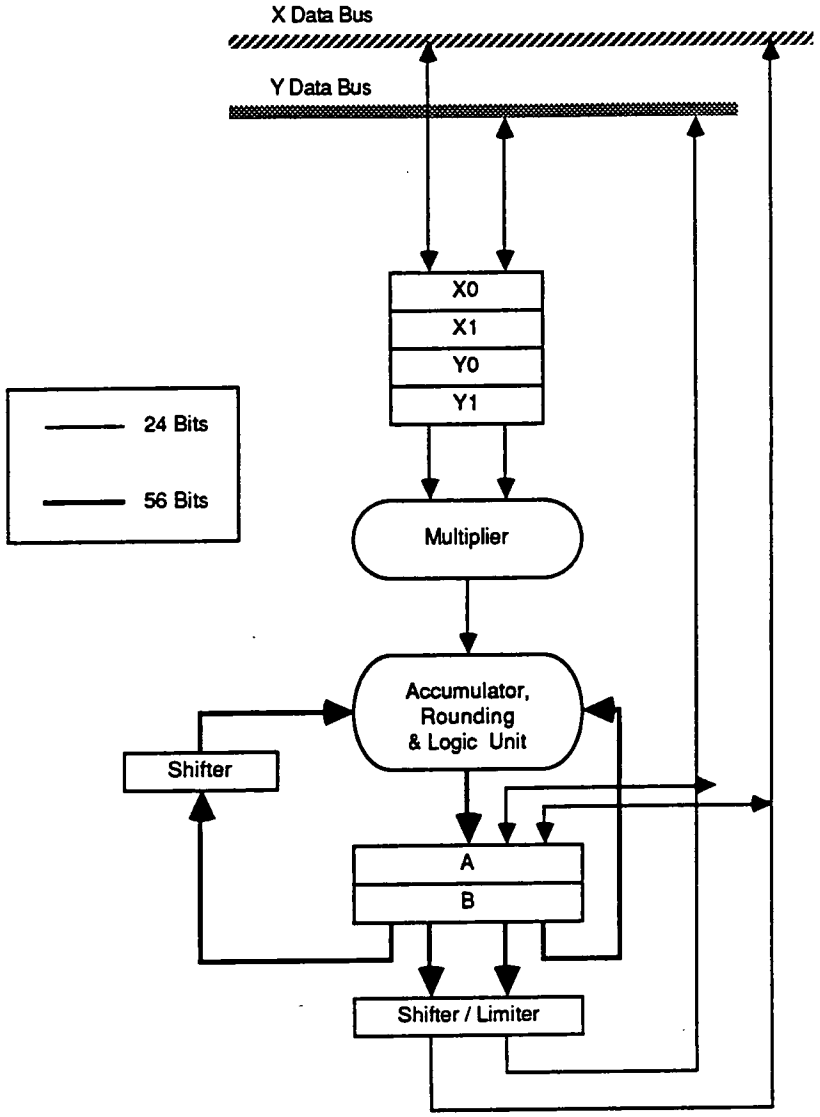


Fig 5.4 Architecture of the Arithmetic and Logic Unit †

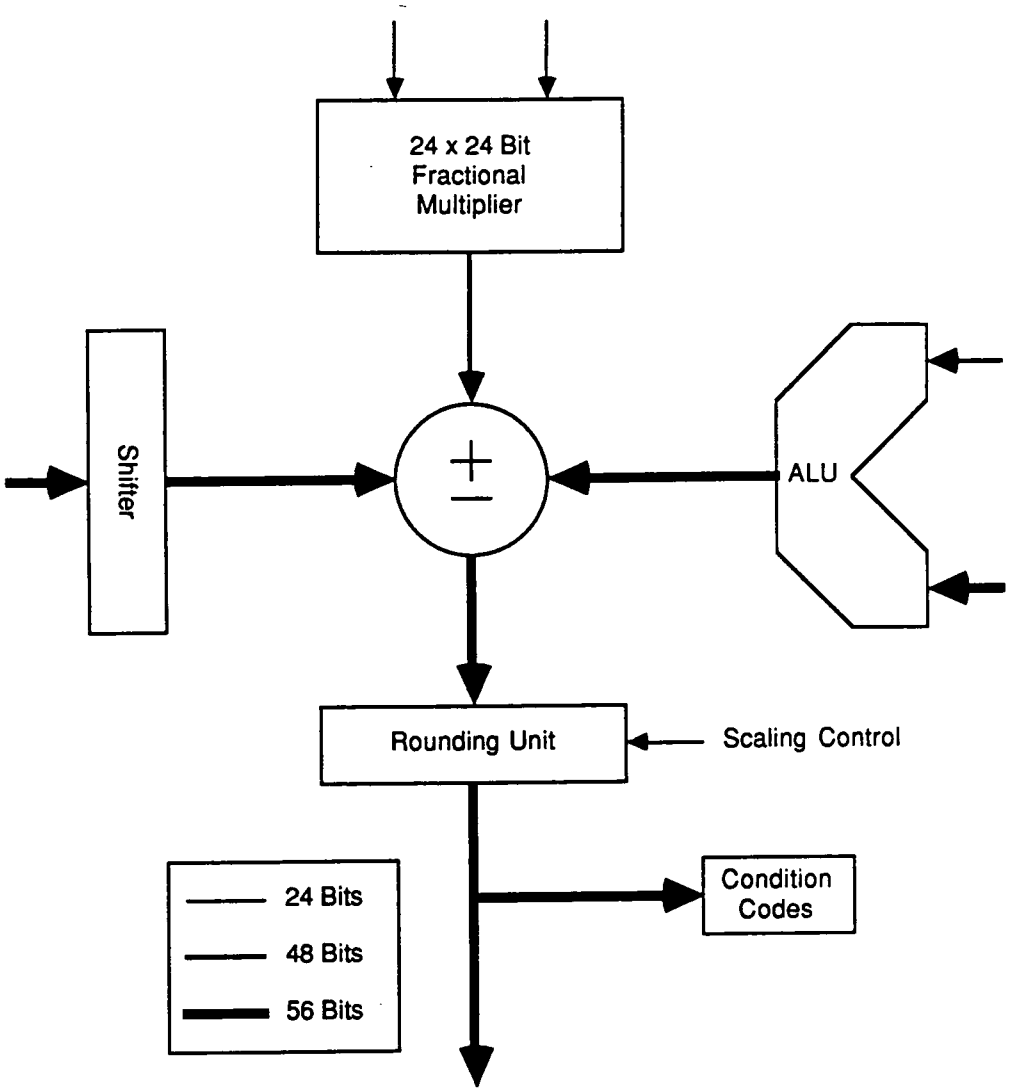


Fig 5.5 Block Diagram of the MAC Unit †

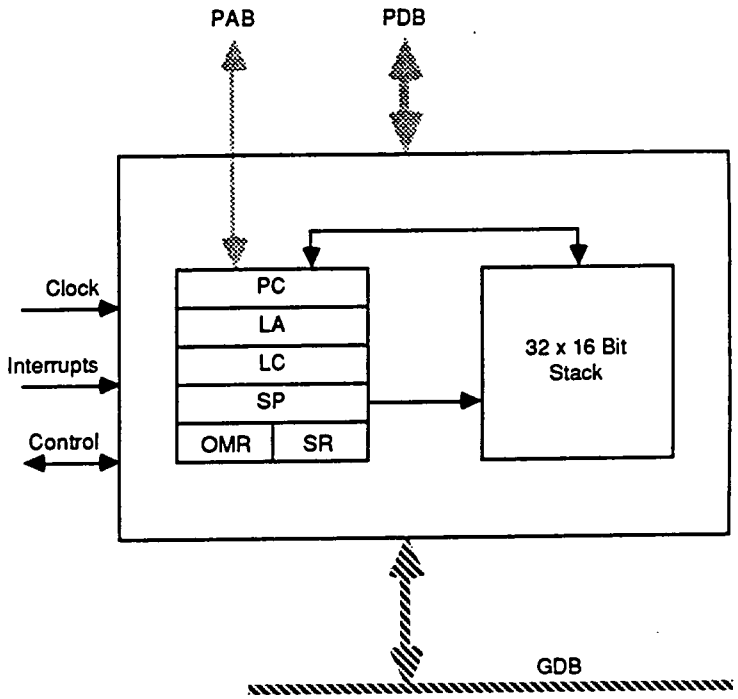
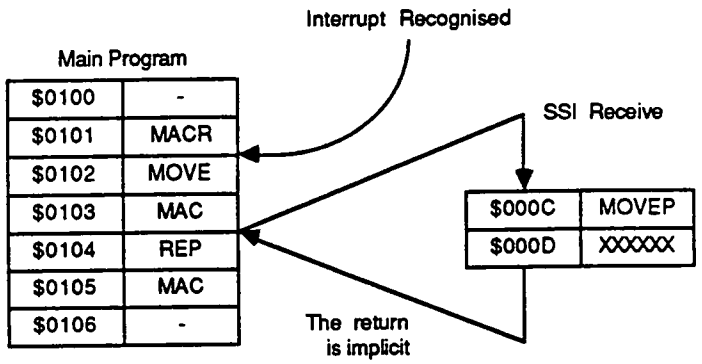
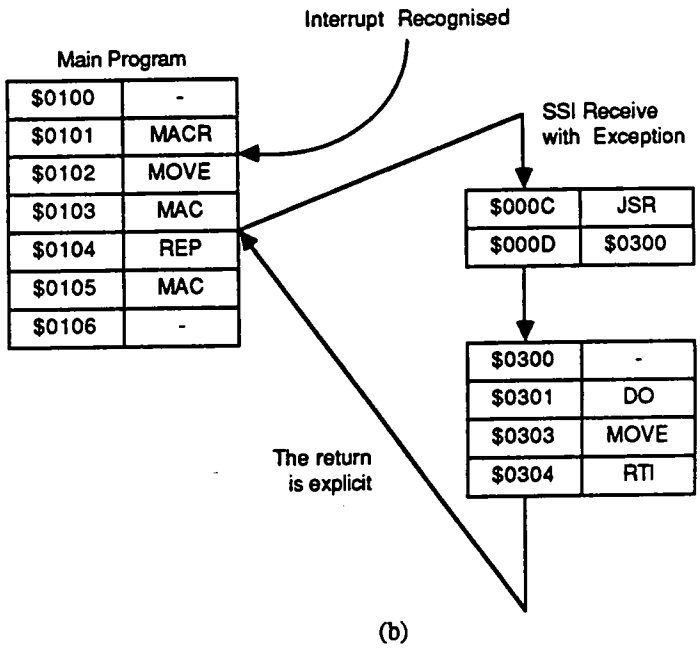


Fig 5.6 The Program Controller †



(a)



(b)

Fig 5.7 Short and Long Interrupts †

Chapter 6

Digital Filtering on the DSP56001

6.1 Introduction

This chapter is concerned primarily with demonstrating how the features of the Motorola DSP56001 may be utilised to implement efficient digital signal processing applications. The architectural features outlined in the previous chapter — indirect register addressing, extended Harvard architecture and a fast multiplier — are used in the work presented, together with illustrations of time stationary coding, to construct highly efficient infinite impulse response (IIR) filter routines. Issues relating to finite register length — quantisation noise, noise transfer functions, input scaling and node scaling — are considered only when they directly relate to implementation issues, as extensive coverage of these effects is not considered relevant to the points being made and would serve as an unnecessary complication.

Filtering is one of the most widely used digital signal processing functions. For this reason, the architecture of most digital signal processors is such that they are able to implement digital filtering algorithms very efficiently. Although there are many types of digital filtering structures and algorithms available, [37], [38], [39], [40], this chapter concentrates on the implementation of the canonic II form of the infinite

impulse response (IIR) filter since this may be optimally implemented on the DSP56001, and is suited to the implementation of the application filter outlined in Appendix A. The problems involved with the implementation of the canonic II form of the application filter are described, and a satisfactory solution presented.

The work described in this chapter was carried out using the Motorola ADS56 Development System. This system comprises a DSP56001 development board interfaced to an IBM PC and a software package including a monitor program, an assembler and a linker.

After describing the general IIR canonic II structure, together with an extension of the basic code for multi-channel filtering in section 2, section 3 goes on to investigate the implementation of the application filter on the DSP56001. It is shown in Section 4 that this filter possesses a non-standard structure, requiring a slight algorithmic modification. Furthermore, it is demonstrated why this particular filter may not be implemented in the form of a cascade of biquadratic sections on the DSP56001. Section 5 introduces an alternative structure, and extension to the multi - channel case, and provides a comparison with the more standard biquadratic approach. Section 6 provides a summary.

6.2 Realisation of the Canonic Biquadratic Filter Section on the DSP56001

The canonic form of the biquadratic filter section is widely used as the basic element in many digital filter realisations, since it incurs a minimal instruction cycle penalty. The basic biquadratic structure is shown in Fig 6.1 [75]. It may be seen from the figure that this form requires five multiplication operations. The scaling factor, which includes a factor of 0.5, and coefficient b_0, may be combined to give the structure

shown in Fig 6.2. This structure demonstrates the value of input scaling (division) and accumulator output scaling (multiplication) when coefficient values greater than those representable by the processor registers (in the case of the DSP56001, $1\text{-}2^{-23}$ and -1.0) are required, as the coefficient values used in this implementation are scaled versions of those in Fig 6 .1. Only processors incorporating this zero overhead accumulator scaling facility provide suitable platforms for this structure [76].

The transfer function for Fig 6.1 is given by

$$G_1(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}} \tag{1}$$

with difference equations given by

$$y_1(n) = b_0 w_1(n) + b_1 w_1(n-1) + b_2 w_1(n-2) \tag{2}$$

$$w_1(n) = c_0 x(n) + a_1 w_1(n-1) + a_2 w_1(n-2) \tag{3}$$

and that for Fig 6.2 is given by

$$G_2(z) = \frac{\alpha(1 + \mu z^{-1} + \sigma z^{-2})}{0.5 + \gamma z^{-1} + \beta z^{-2}} \tag{4}$$

with difference equations given by

$$y(n) = 2(0.5 w_2(n) + 0.5\mu w_2(n-1) + 0.5\sigma w_2(n-2)) \tag{5}$$

$$w_2(n) = 2(\alpha x(n) - \gamma w_2(n-1) - \beta w_2(n-2)) \tag{6}$$

Now, multiplying top and bottom of (4) by 2 gives

$$G_2(z) = \frac{2\alpha(1 + \mu z^{-1} + \sigma z^{-2})}{1 + 2\gamma z^{-1} + 2\beta z^{-2}} \tag{7}$$

and comparing like terms in (1) and (7) gives

$$2\alpha = b_0 \qquad 2\alpha\mu = b_1 \qquad 2\alpha\sigma = b_2$$

$$\gamma = a_1 \qquad 2\beta = a_2 \ .$$

The code segment for the structures of Figs 6.1 and 6.2, which are similar, differing only in their coefficient values, are given in Fig 6.3, together with a representation of

their data memory requirements. Both forms hold their coefficients in on-chip y-data space and their intermediate values, $w_j(n-i)$, in on-chip x-data space. This allows both data areas to be accessed simultaneously, since both halves of the AGU may be used. The coefficients are accessed using a cyclic addressing mode, whereas the intermediate values require only a linear mode. These code segments assume that the input and output values are accessed via a particular word in memory — in this case a peripheral i/o location. It would be a simple matter, however, to use non-peripheral locations, or even buffers, using indirect addressing. An explanation of the operation of the code is shown in Table 6.1, using nomenclature relating to the second form.

6.3 Expansion to Multiple Data Paths

It would be possible to support multiple data paths by using a different set of address registers for each channel / data path. However, as the number of address registers is limited, the number of channels which may be implemented using this method is correspondingly small. A more efficient method is available, thanks to the versatility of the addressing modes and the provision of a zero overhead DO loop, which requires only one minor change to the filter code kernel.

A possible memory structure of an N_f - channel filter is shown in Fig 6.4 ($j = 0 .. N_f - 1$). For the single channel case, the coefficients are accessed using a cyclic buffering scheme. If, in the multi-channel case, the response of each filter is to be independently controlled, then the same type of addressing scheme may be used providing that a larger buffer size is declared. The coefficient blocks would then be accessed in a cyclic sequential manner, Fig 6.4a. However, if each filter is to possess the same response, then a single coefficient block, addressed as in the single channel

case, will suffice, Fig 6.4b. The latter case obviously requires less memory.

The single channel case does not use cyclic addressing to access the intermediate values ($w(n-i)$), as there is no need to do so [75]. In the multi-channel case, however, cyclic addressing may be used to allow the intermediate value blocks to be accessed in a cyclic sequential manner. This would force R0 to point back to the first section whenever the last section had been completed. For this reason, then, M0 must be initialised so as to provide a cyclic buffer of size $2N_f$ for R0. Furthermore, at the end of the end of the j^{th} section R0 is pointing to $w_j(n-1)$. This must be modified such that R0 is pointing to $w_{j+1}(n-1)$ at the end of the section. This may be accomplished by using N0 to post increment R0 after the last reference in the section, ie by changing line 4 from

```
MAC    x0,y0,a      a,x:(R0)          y:(R4)+,y0
```

to

```
MAC    x0,y0,a      a,x:(R0)+(N0)     y:(R4)+,y0
```

For biquadratic filter sections, N0 should contain the value 2.

It should be noted that the data paths in this implementation are orthogonal — data input and output paths are not connected. A cascade filter structure may easily be implemented, however, by storing the output of one filter section in an ALU register and using it as the input to the next section.

Simply by using a modified addressing scheme, then, and a computation section embedded in hardware DO-loop, a single channel filter may be expanded to a multiple channel implementation with no additional performance overheads.

6.4 Problems in the Implementation of the Application Filter on the DSP56001

From Fig A.3, it may be seen that the application filter may be decomposed into a single pole highpass section in cascade with a biquadratic bandpass section. The single pole section may be simply implemented as half a biquadratic section, its output forming the input of the bandpass section. The transfer function of the bandpass section (Equation A.11) does not contain a term in b_0, indicating that the output of the section contains no proportion of the present input section. Furthermore, Appendix A shows that no input scaling is required, since the overall gain of the section is less than one. This results in the modified structure shown in Fig 6.5.

The code shown in Fig 6.3 is unsuitable for this structure, however, as the output would always be zero. As $b_0 = 0$, then $\alpha = 0$. Now, consider the biquadratic section output as it takes its first few input values, from (9),

$$\begin{aligned} w(0) &= 2\alpha x(0) \\ &= 0 \\ w(1) &= 2\alpha x(1) + \gamma .0 \\ &= 0 \\ w(2) &= 2\alpha x(2) + \gamma .0 - \beta .0 \\ &= 0 \end{aligned}$$

and therefore,

$$w(n) = 0, \qquad n = 0..\infty$$

From Equation (6), as $y(n)$ is a function of $w(n), w(n-1)$ and $w(n-2)$, then $y(n)$ will always take the value zero. Thus this particular implementation of the biquadratic filter section is useless when applied to those filters with $b_0 = 0$.

What is required is code that implements a filter section whose difference equation contains no proportion of $w(n)$. In the code segments of Fig 6.3, $w(n)$ is first

calculated (lines 1, 2 and 3). This value is then left in the accumulator while $y(n)$ is calculated using $w(n-1)$ and $w(n-2)$. The contribution of $w(n)$, then, may be disregarded if the accumulator is overwritten by those lines that calculate $y(n)$. In this case, $w(n)$ is used only to update the values of $w(n-1)$ and $w(n-2)$. This may be implemented by replacing the MAC instruction of line 4 with an MPY instruction, which overwrites the accumulator.

This modified code will implement the filter structure shown in Fig 6.5. However, from Equation A.17, it may be seen that for the application filter, b_2 contains a term in 2^{-30}. This value may not be represented within the 24bit registers of the DSP56001, and so the coefficient value is truncated. This truncation causes a shift in the location of the poles of the filter and hence changes its characteristic magnitude and phase response. In particular, the poles, previously a complex conjugate pair (Equation A.9) are forced onto the real axis at $z = 1 - 2^{-13}$ and $z = 1$. This problem may not be resolved by explicitly coding the direct feedback path of the structure, as the unmodified biquadratic section also contains coefficients with terms in 2^{-30}. The problem may be met by either implementing a 48bitx24bit multiplication routine [76], or by decomposing the biquadratic into two single pole sections. This latter option will now be described, as the former is computationally expensive.

6.5 A Cascade of Single Pole Sections

6.5.1 Structural Decomposition

Forming two single pole sections from the modified biquadratic section would result in two cascaded single pole sections requiring complex coefficients, which would add considerably to the computational complexity [77]. For this reason, the unmodified biquadratic section was decomposed, and the feedback path implemented explicitly, resulting in a cascade of single pole sections.

The general single pole canonic structure is shown in Fig 6.6a. However, substituting the coefficients given in Equations A.2 and A.10 for the high and low pass sections results in the structures given in Figs 6.6b and 6.6c. The high pass section uses a coefficient of -1, which may be implemented either as a subtraction or as a multiplication operation. Although each require the same amount of time to perform, the multiplication operation may also incorporate a rounding operation, and so was used in the code. These filter sections use coefficients that may be represented with 24bits and so may be safely implemented on the DSP56001.The structure of the entire filter is shown in Fig 6.7, and its code presented in Fig 6.8.

6.5.2 The Sequence of Operations

Consider a flow of operations across the filter structure from left to right. It is clear that the single pole high pass section may be completed with no problems. The summation at point A, however, may not be evaluated until the filter output, $y(n)$, has been determined, and so execution must halt at this point. The output may be determined by continuing the calculation at point C, which is separated from the

previous signal path by a delay operator, and continuing through the final single pole high pass section. The summation at point A may then be completed, followed by the signal path up to point C. It is of no consequence whether stage one or stage two is calculated first.

The DSP56001 incorporates two 56bit accumulators. Thus, the intermediate value stored at point A may be left in accumulator a, in full 56bit precision, while accumulator b is used for the second stage. The two accumulators may be added together, eliminating the rounding errors which would occur if the value at point A was stored in an intermediate memory location.

6.5.3 The Code

From Fig 6.7, and Appendix A, it may be seen that the single pole high pass section incorporates a term in $w(n)$, whereas the single pole low pass section does not. The low pass section, then, needed to make use of the $w(n)$ blocking properties of the code shown in Fig 8.5. From Appendix A, the single pole filter sections have gains of less than one, and so no external scaling is required. Consequently, the scaling factors may be assumed to be equal to one, and hence disregarded. The structure of the code may take two forms, depending upon whether the rounding operation is performed during or before the summation at point A of Fig 6.5.

Both versions of the code are shown in Fig 6.9, together with the memory usage requirements. An explanation of the code is given in Table 6.2. Address register R0 is used to point to the $w^{j}(n-1)$ values, which are stored in internal x memory and are accessed cyclically by setting M0. The offset register, N0, is used to allow a return

to the start of the block. The coefficients are held in internal y memory and are also accessed cyclically using R4 and M4.

6.5.4 Expansion to Multiple Orthogonal Data Paths

Expansion to the multi-channel case is straightforward, using methods similar to those outlined in Section 6.3. If the response of each filter is to be independently controlled, then address mode register R4 must be used to define a cyclic address range equal to N_f . *Number of coefficients per filter*. Furthermore, the address modifier register R0 must be used to define a cyclic address space equal to $3N_f$.

6.5.5 Performance

Version "a" requires 11 cycles to perform the filter computation, version "b" requires 12. The overhead for setting up a hardware DO loop is three cycles, and the instruction cycle time is 97.5ns. Let the number of channels required be represented by C, the number of cycles required to perform the computation by N_c and the required sample rate of each filter by R_f, then the following relationship must hold true for a realisable implementation

$$3 + N_c \times C < \frac{R_f^{-1}}{97.5 \times 10^{-9}} \tag{12}$$

Using this equation, the maximum sampling frequency for a single channel filter is 683.76kHz for type "a" and 732.6kHz for type "b". For a sampling frequency of 28kHz, the maximum number of channels that may be supported is 30 for type "a" and 33 for type "b".

The original filter structure, used in the transputer implementation, was also investigated. However, the single pole section alone was found to require 10 cycles to execute, and so this form would offer significantly less performance than the canonic form.

The frequency and phase responses of this filter were tested by using Hypersignal Workstation©, and found to compare with those presented in Appendix A.

6.6 Summary

This chapter has demonstrated the implementation of an infinite impulse response (IIR) filter on the Motorola DSP56001. One of the basic elements of recursive filtering, the canonic II biquadratic section, has been described and the associated DSP56001 code presented. Various coding methods may be used, depending on the coefficient values and whether scaling is required. Three variations in filter structure — standard, coefficient scaling and *w(n)* blocking — have been presented and shown to represent modifications of the same basic code. The coefficient scaling form depends for its efficiency upon the use of accumulator output scaling, available on the DSP56001.

The canonic form of the application filter has been described, with the view that this would offer the most efficient implementation on the DSP56001. However, the coefficients of the biquadratic section of this filter require wordlengths greater than those accommodated by the ALU registers of the DSP56001. For this reason, it was necessary to form a filter structure based on single pole sections. Two forms of the filter were coded, and found to operate at maximum sample frequencies of 683.76kHz

and 732.6kHz respectively.

This chapter concludes the investigation of the applicability of the Inmos Transputer and Motorola DSP56001 to digital signal processing (DSP) type algorithms (ie those requiring a high i/o bandwidth and using small, multiplication intense computation sections), in particular to the application filter.

Albeit a general purpose processor, the transputer has been shown to be capable of effectively implementing DSP type algorithms. Although this is due in part to its RISC type architecture, one of the main contributing factors to the transputer's operational efficiency is its ability to overlap communication and computation. This often enables the transputer to transfer data with minimal time penalty — the transfer appears "invisible" to the processor. However, as shown in Chapter 4, performance is likely to suffer whenever the computation execution time is short and the data transfer requirement is high (ie more links are required). Furthermore, the application filter code utilises a shifting operation in place of a multiplication, which requires approximately half as many cycles to execute than the corresponding multiplication.

The integer multiplier is the main performance limiting feature of the transputer, especially when implementing multiplication intensive algorithms. The inclusion of a concurrent floating point unit (FPU) on the T80x series does little to alleviate this problem. Other limiting features include the available memory bandwidth — only one word may be accessed at any one time — and the link transfer bandwidth. The latter results in the requirement to maintain the computation execution period above a certain limit; the time required to compute one word of data should be greater than the time required to transfer a word over a link.

The architecture of the DSP56001 has been designed around the need to ensure

that its multiplier unit is fed with data as fast as it can use it. The arithmetic and logic unit (ALU) incorporates a single cycle, non-pipelined MAC unit together with several input and accumulator registers. Combining the ALU with a comprehensive register indirect addressing scheme and an extended Harvard architecture, the DSP56001 is extremely effective at implementing DSP algorithms. Of particular note is the ability to implement zero overhead modulo and bit reversed addressing schemes and hardware DO loops.

The DSP56001 also incorporates two on-chip communication peripherals, designed to interface to "host" processors and serial devices such as modems and ADC/DACs. These perform byte wide parallel and synchronous / asynchronous serial communications, although at a slower rate than the transputer. Some facility has been given to multi-processor operation, namely DMA control lines and a "network" mode on one of the serial ports, but these are limited and involve low data transfer rates compared with the transputer.

In summary, then, the DSP56001 is highly efficient at implementing DSP algorithms due to its optimised architecture and fast multiplier. Although it incorporates three additional communications ports, these offer slower transfer bandwidth than the transputer. Limited multi-processor support is provided.

The transputer, in contrast, efficiently implements inter-processor communication due to its microcoded scheduler and concurrent link engines, having the ability to make transfers seem almost "invisible". However, the available link and memory bandwidths, and the provision of a relatively slow multiplier, limit performance when implementing DSP type algorithms.

Description of Code Segment 2	
Line Number	Comments
1	The input value is scaled and placed in accumulator a. $w(n-1)$ is placed in x0, R0 is post incremented, to point to $w(n-2)$. R4 is presently pointing at γ, which is moved into y0. R4 is post incremented to point to β.
2	x0 and y0 are multiplied and added to accumulator a, which now contains $\alpha x(n)+\gamma w(n-1)$. $w(n-2)$ is moved into x1, this time there is no change in R0. β is moved into y0, which is post incremented to point to 0.5μ.
3	x0 and y0 are multiplied and added to accumulator a, which now contains $\alpha x(n)+\gamma w(n-1)$. $w(n-2)$ is moved into x1, this time there is no change in R0. β is moved into y0, which is post incremented to point to 0.5μ.
4	x1 and y0 are multiplied and added to the accumulator, which is rounded to 24 bits and now contains $\alpha x(n)+\gamma w(n-1)+\beta w(n-2)$. x0 ($w(n-1)$), is moved into $w(n-2)$, and R0 is post decremented to point at $w(n-1)$. 0.5μ is moved into y0, which is post incremented to point at 0.5σ.
	At this point, accumulator a holds $0.5w(n)$, x1 holds $w(n-2)$, R0 points to $w(n-1)$ and R4 points to 0.5σ. The previous section, then, has calculated a value for $w(n)$. The next section will use this value to calculate a value for the output.
5	x0 and y0 are multiplied and added to the accumulator, which now contains $0.5(w(n-1)+\mu w(n-1))$. However, before the accumulation operation, the rounded contents of a are left shifted one bit (multiplied by two) and moved into $w(n-1)$, ready for the next cycle. 0.5σ is moved into y0. R4 is post incremented to point to α.
6	x1 and y0 are multiplied, and added to accumulator a, which is rounded and now contains $0.5(w(n)+\mu w(n-1)+\sigma w(n-2))$. α is moved into y0, ready for the next cycle. R4 is post incremented and forced to return to the beginning of the coefficient block by the cyclic addressing scheme.
7	The accumulator now holds a rounded value of $0.5y(n)$, which is left shifted by one bit and moved to the output location in the final instruction of the loop.

Table 6.1 Operation of the Filter Code

Description of Three Pole Filter Code	
Line No	Comments
1	Clear accumulator a and move the present input into y1.
2	Move input into a, $w^1(n-1)$ into x0 and a^1 into y0, post increment R4.
3	a = a + $a^1 \times w^1(n-1)$, rounded. Move b^1 into y0, post increment R4. a now contains the new value of $w^1(n)$.
4	Move the rounded value of a, $w^1(n)$, into $w^1(n-1)$. a = a + $b^1 \times w^1(n-1)$, post increment R0. Move b^2 into y0, post increment. At this point, a holds the output of the first stage and $w^1(n-1)$ has been updated. y0 contains b^2, R0 points to $w^2(n-1)$ and R4 points to a^3. This is point A.
5	Move $w^2(n-1)$ into x0, post increment R0 to point to $w^3(n-1)$.
6	b = b + $b^2 \times w^2(n-1)$. Move $w^3(n-1)$ into x0. Move a^3 into y1, post increment R4 to point to b^3.
7	b = b + $a^3 \times w^3(n-1)$, rounded. Move b^3 into y0, post increment R4 to point to a^1 using circular addressing.
8	b = b + $b^3 \times w^3(n-1)$. Update $w^3(n-1)$, post decrement R0 to point to $w^2(n-1)$. The order of operations now depends upon whether or not the multiplication includes a rounding operation. If so, then option 'a' is carried out, if not, then option 'b'.
9a	Add b (rounded) to a. Move $w^2(n-1)$ into x0, and b (rounded) into the output memory location.
10a	a = a + $a^2 \times w^2(n-1)$. Note that for the application filter, $a^3 = b^2$, which is already stored in y1, and so there is no need for a coefficient move at this point.
11a	$w^2(n-1)$ is updated.
9b	Add b to a, move $w^2(n-1)$ into x0.
10b	Round b.
11b	a = a + $a^2 \times w^2(n-1)$. The rounded value in b is moved to the output memory location.
12b	$w^2(n-1)$ is updated. R0 is post incremented by N0, allowing it to point to the beginning of the block, using cyclic addressing.

Table 6.2 Operation of the Three Pole Filter Code

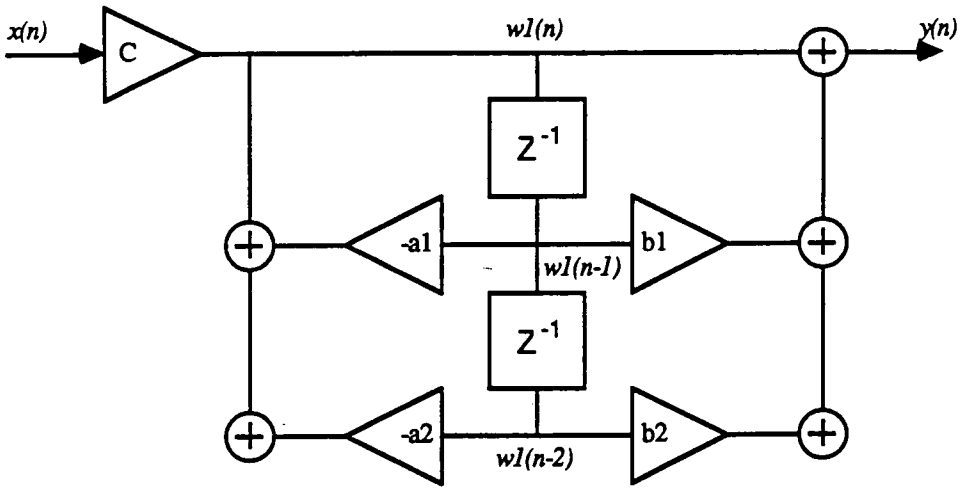


Fig 6.1 Basic Biquadratic Structure

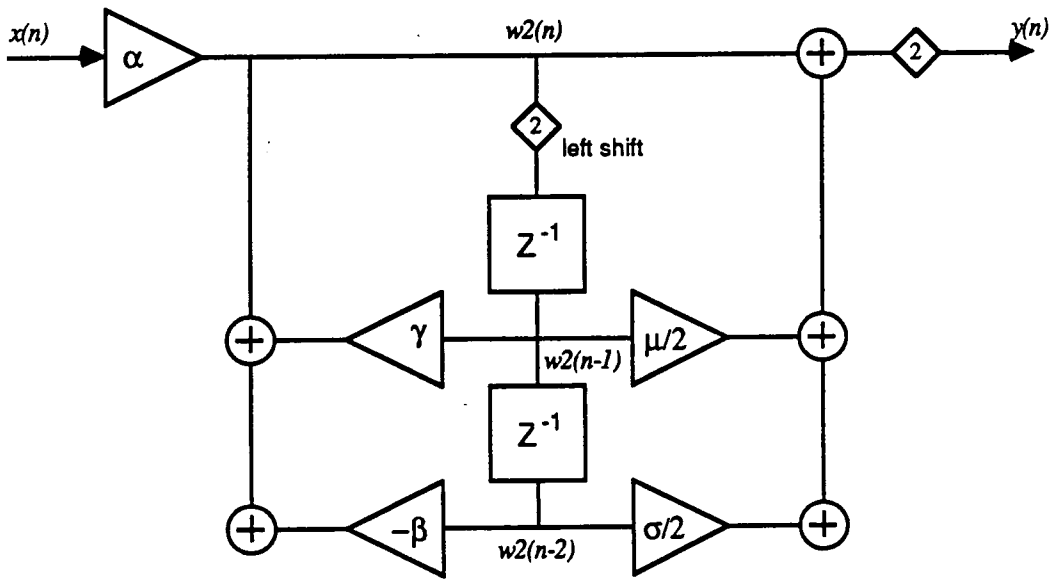


Fig 6.2 Alternative Biquadratic Structure

```
1       MPY     x0,y0,a         x:(R0)+,x0      y:(R4)+,y0
2       MAC     x0,y0,a         x:(R0),x1       y:(R4)+,y0
3       MACR    x1,y0,a         x0,x:(R0)-      y:(R4)+,y0
4       MAC     x0,y0,a         a,x:(R0)        y:(R4)+,y0
5       MACR    x1,y0,a                         y:(R4)+,y0
6       MOVE                    a,x:$FFEF
```

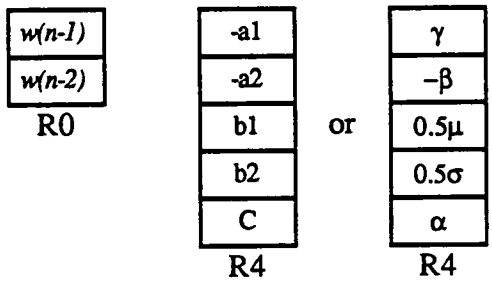


Fig 6.3 Biquadratic Section Code and Memory Requirements

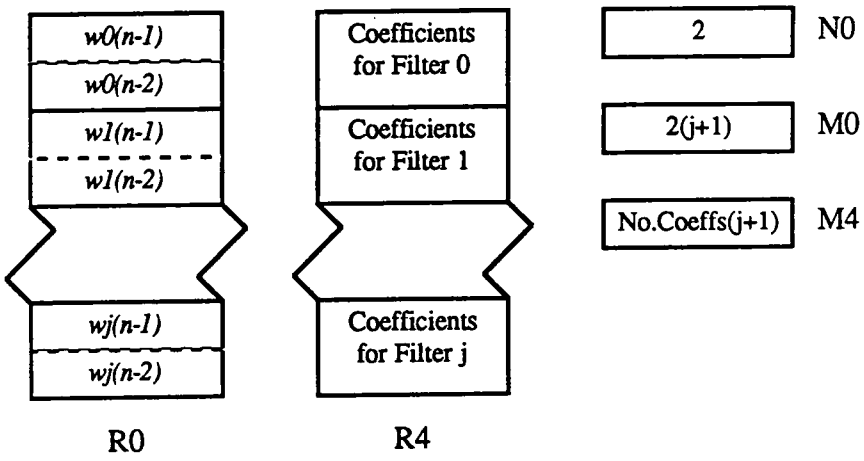


Fig 6.4a Memory Requirements for Multiple Filter Responses and Multiple Data Paths

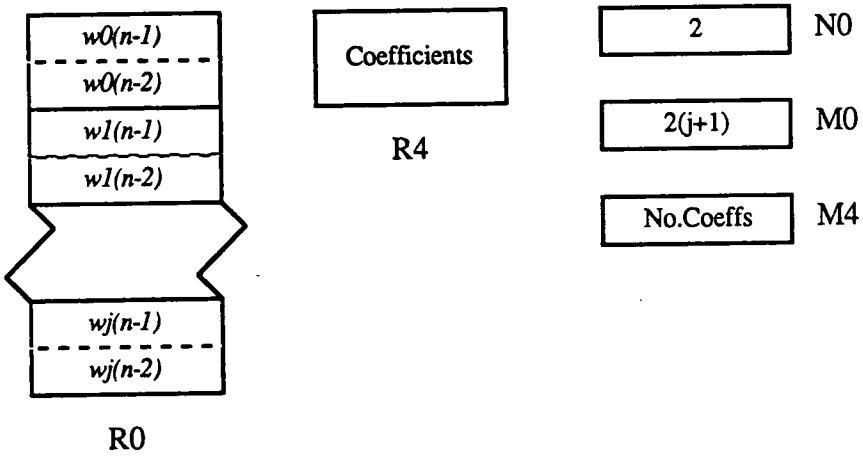


Fig 6.4b Memory Requirements for Multiple Data Paths

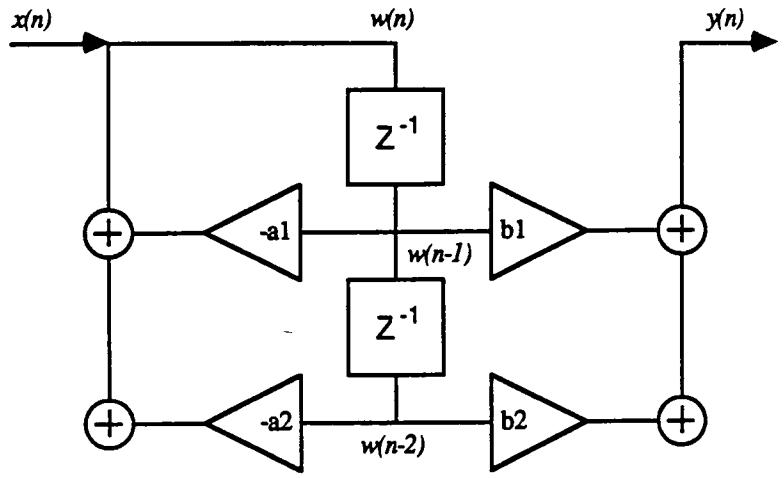


Fig 6.5 Modified Filter Structure

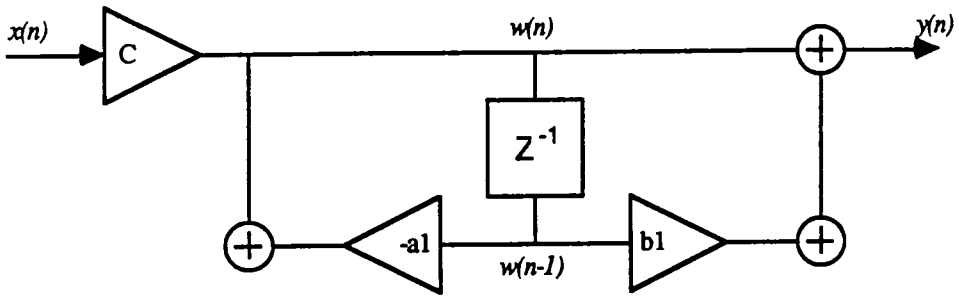


Fig 6.6a General Single Pole Section

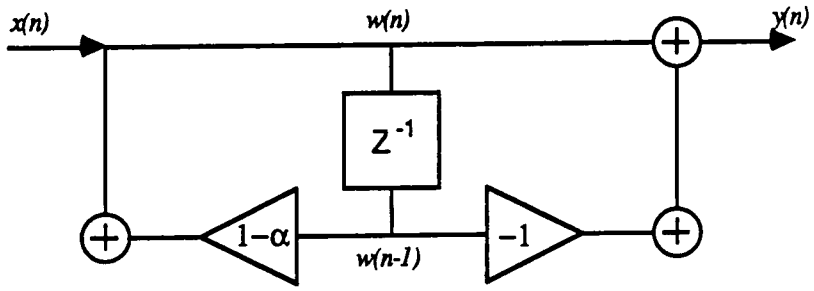


Fig 6.6b High Pass Section

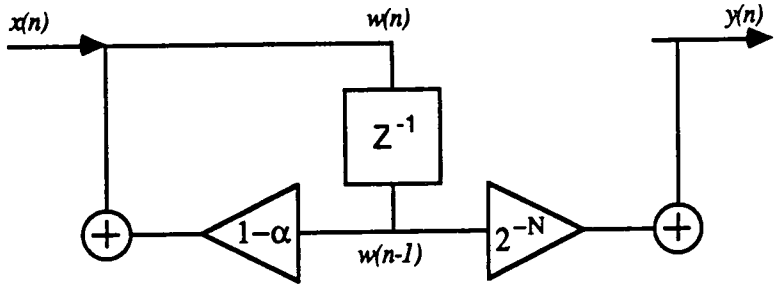


Fig 6.6c Low Pass Section

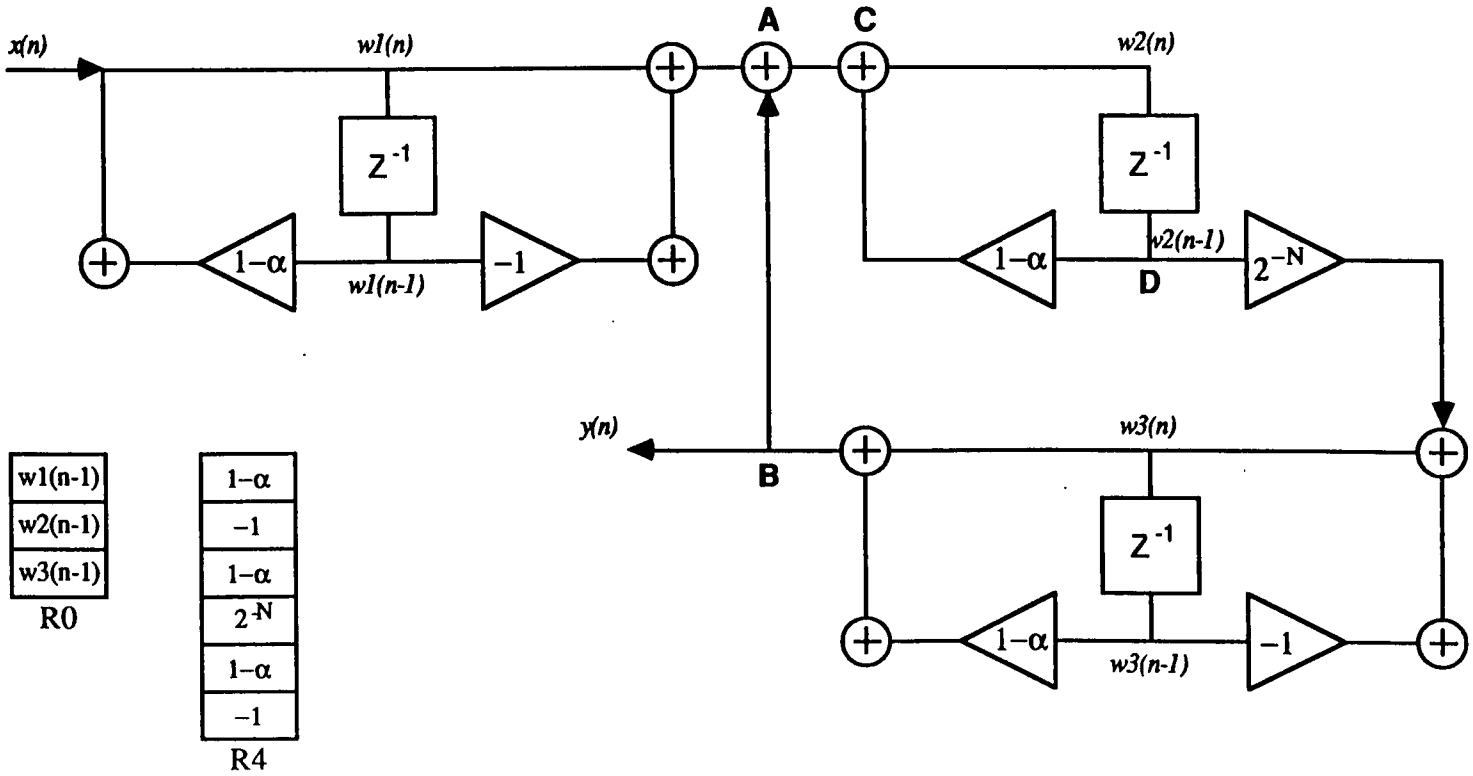


Fig 6.7 Structure of Cascaded Single Pole Filter Realisation

```
1       CLR         a
2       MOVE        y1,a            x:(R0),x0       y:(R4)+,y0
3       MACR        x0,y0,a                         y:(R4)+,y0
4       MAC         x0,y0,a         a,x:(R0)+       y:(R4)+,y0
5       MOVE                        x:(R0)+,x0
6       MPY         x0,y0,a         x:(R0),x0       y:(R4)+,y1
7       MACR        x0,y1,b
8       MAC/R       x0,y0,b         b,x:(R0)-

9a      ADD         b,a             x:(R0),x0       b,y:$output
10a     MACR        x0,y1,a
11a     MOVE                        a,x:(R0),x0

9b      ADD   b,a                   x:(R0),x0
10b     RND   b                                     b,y:output
11b     MACR  x0,y1,a
12b     MOVE                        a,x:(R0)+N0
```

Fig 6.8 Code for the Three Pole Single Stage Cascade Filter

Chapter 7

Hybrid Multiprocessor: Design Concepts

7.1 Introduction

The computational power of contemporary processors is increasing, but the most recent devices are approaching the performance limits of silicon based fabrication technology. There will always be applications, however, that require computational performance greater than that which may be provided by any single processor. In this case, there is no alternative but to move to a multi-processor system [10]. The performance of many digital signal processing applications may be improved considerably by implementing them on a multi-processor system, due to the increased overall computational power. Furthermore, many digital signal processing algorithms lend themselves to parallel partitioning, and so they may be easily and profitably mapped onto a multi-processor system.

However, although implementing an application on a number of concurrently operating processing units greatly increases the overall computational performance, these processing units must be supplied with data at a rate at least equal to their computation rate if the system as a whole is not to suffer a performance degradation [77]. Thus, the bandwidth of the inter-processor communication mechanism must

not fall below that of the computation. For any given set of tasks, or processes, the requirement for maximum computational performance will tend to decompose the application into as many parallel sub-processes as possible, running each sub-process on a separate processor. However, the requirement to reduce the overall communications bandwidth tends to favour a sequential program, running on a single processor [6], [20]. In any multi-processor architecture a compromise must be made between these two extremes.

Another important aspect of a multi-processor design is that of scalability [78], [79]. The scalability of a system is a gauge of the number of processors that may be added before system performance is unacceptably degraded. The inter-connection network greatly influences scalability.

Many high performance multi-computers are available today [80], [81], [82]. They range from small systems using relatively inexpensive and low performance interconnection mechanisms to systems utilising high performance processors and very elaborate interconnection mechanisms using dedicated communications co-processors. Such systems are expensive, however, and are not generally optimised for digital signal processing applications. One of the aims of this project was to design a multi-processor system using relatively inexpensive off-the-shelf parts and an inexpensive interconnection mechanism, which ruled out the use of complicated bus switching networks and communications co-processors.

Presented in the following chapters is a description of the architecture and performance of a multi-processor system that isolates the majority of the workload associated with computation and interprocessor communication onto separate processors. This Hybrid Multiprocessor (*Hymips*) has been designed with cost and

scalability in mind.

This chapter offers an architectural overview of such a system. The general design issues, such as the choice of processor, the interprocessor connection mechanism and the control software methodology are discussed. The following chapter deals with more specific design issues, problems encountered and their solutions.

A general specification of the requirements which the system must satisfy is given in Section 2. Section 3 discusses the processors and how they may be best utilised. The interprocessor communication mechanism is outlined in Section 4. Section 5 covers memory requirements, while Sections 6 and 7 cover system reconfiguration and reprogramming. Finally, Section 8 presents a summary of the proposed architecture.

7.2 System Requirements

The design of the multi-processor was to satisfy certain requirements. These did not constitute a technical specification as such, but did provide a guideline for the design process. The multi-processor was seen very much as a prototype system. A list of the major requirements follows.

i. The system should interface with a host system (a PC), in order to provide access to a terminal, a monitor and a file system.

ii. The system should also possess the ability to independently interface with additional peripherals such as disk storage units and graphics boards, as they provide higher performance than the host based peripherals.

iii. Digital signal processing algorithms should be efficiently implemented.

iv. The architecture should be scalable.

v. The architecture should not be complex, and make use of relatively

inexpensive off-the-shelf components.

vi. The interprocessor connection mechanism should allow high speed data transfers both into and out of the system whilst incurring minimal communications overhead.

vi. The interprocessor connection mechanism should be independent of processor type.

7.3 The Processors

The system must implement somewhat specialised applications, but still be capable of interfacing to general purpose peripherals such as disc storage devices. Digital signal processing algorithms require few instructions other than arithmetic and basic logic functions. General purpose microprocessors, be they CISC or RISC, offer many instructions that would not be required by signal processing algorithms. As a consequence of this generality, such microprocessors are inefficient at implementing this class of algorithm. As has been shown in Chapter 6, digital signal microprocessors, due to their specialised architectures and instruction sets, are capable of executing such algorithms far more efficiently than their general purpose counterparts. However, because they are specialised, then they are not suitable for managing the interfacing to external peripherals. Furthermore, managing interprocessor communication would incur a severe computation performance penalty for such devices. The Motorola DSP56001 offers a 24 bit wordlength, a high degree of operational concurrency and a number of internally based peripheral interfaces.

Although the transputer is a general purpose RISC-type processor, and so is relatively inefficient at executing DSP algorithms, it has been designed to provide efficient inter-processor communication. The transputer is capable of communication with up to four other transputers using its serial links. Furthermore, this

communication proceeds with very little cpu intervention - even when all four links are saturated, cpu performance is degraded by only 5%. The transputer may be programmed in many parallel languages and run inside a mature operating system, providing the usual peripheral interfaces.

An architecture that allows the transputer to manage communications and the DSPs to perform the computation promises to be particularly efficient, as each type of processor is allowed to perform tasks for which it has been optimised. A system architecture, consisting of nodes connected by transputer links, each of which comprise a single transputer controlling the data flow around a number of DSPs, would allow scalability both in the number of DSPs supported within a node and the number of nodes supported, Fig 7.1. Furthermore, the transputer could be easily connected to disk storage units, graphics boards or host systems, Fig 7.2.

7.4 The Interconnection Scheme

In a multi-processor, it is vital that data is transferred to the processors as quickly as possible, in order to ensure that the overall performance of the system is not impaired. The design of the interconnection network, then, is of paramount importance in the design of any multi-processor, as it is this sub-system which determines the overall scalability of the system, and hence the maximum potential performance. This section deals with the design decisions used to select the interprocessor connection sub-system of the multi-processor.

The external ports of the processors, and how they may be interfaced, are examined in this section, allowing the optimum interconnection scheme to be determined.

7.4.1 A Review of External Interfaces

7.4.1.1 The DSP56001

The DSP56001 incorporates three on-board peripheral interfaces in addition to its external memory interface (EMI), namely the serial communications interface (SCI), the synchronous serial interface (SSI) and the host interface. The SCI is capable of transferring data at a maximum of 2.56Mbits^{-1} (20.5MHz), the SSI at a maximum of 5.125Mbits^{-1} (20.5MHz). Both of these interfaces offer multi-processing or network modes, allowing for interprocessor communication in a multiprocessor system. However, the communication bandwidth, and the inherent software management overhead associated with servicing these interfaces, makes these interfaces unsuitable for use as the main communication mechanism in this system. The host interface is a synchronous byte wide interface that is capable of transferring data at a burst rate of 8Mbytes^{-1}, but more realistically at 1.71 Mwords^{-1} in interrupt mode. This is a more attractive option, but again the bandwidth and software overhead do not make this a valid option in a system that requires high data throughput. All three of the above options are suitable as a secondary communications interface, however. For instance, the host interface is suitable for receiving low bandwidth control information and the serial interfaces may be connected to an ADC/DAC or used as a debugging port. The EMI of the DSP56001 is able to transfer data at 10.25Mwords^{-1} (20.5MHz), ie one word every instruction cycle.

7.4.1.2 The Transputer

The transputer offers its four serial bi-directional links and its External Memory

Interface (EMI). The links are capable of transferring data at 1.74Mbytes^{-1} in uni-directional mode or 2.35Mbytes^{-1} in bi-directional mode.

Most transputers use multiplexed data and address lines on the EMI, resulting in a transfer bandwidth of 6.66Mwords^{-1} (20MHz), which is slower than a DSP56001 of the equivalent clock speed. The IMST801 transputer, however, uses non-multiplexed bus lines on its EMI, resulting in a transfer bandwidth comparable to that of the DSP56001. Furthermore, this part is available in a 25MHz version, providing a transfer bandwidth of 12.5Mwords^{-1}.

7.4.2 Interfacing Possibilities

The viable options for passing data between the processors would be to either utilise the transputer links to interface to the Host Port through an IMSC011 link adapter or to somehow connect the External Memory Interfaces of the processors. These options are considered in turn.

7.4.2.1 Link to Host Port

Each of the links may be connected to an IMSC011 link adapter, which converts from the serial link format to a parallel byte wide format and vice versa. It would seem feasible to connect a link to the host interface of a DSP through an IMSC011 and some glue logic. There would be three disadvantages to this method, however:

i. Although the host interface of the DSP is able to transfer data at 1.71Wwords^{-1}, the links can transfer at only 1.74Mbytes^{-1} in uni-directional mode, or 2.35Mbytes^{-1} in bi-directional mode, both of which fall well below the capabilities of the host

interface. The transfer bandwidth would be limited by the link bandwidth, which may provide a serious bottleneck for some DSP applications.

ii. In the simplest form, the IMSC011 would be connected to only one DSP. If multiple DSPs were to be connected to a single IMSC011, then what would result would effectively be a (non-buffered) shared bus architecture. Communication from the transputer to the DSP would occur in a broadcast fashion — each DSP would read and interpret a "destination" byte, and then only the designated recipient DSP would read in the following data. Communication from the DSPs to the transputer would need to be arbitrated, probably by a token passing system which would be controlled by the transputer. All this would incur a significant communications management overhead on each of the DSPs. Some of the overhead could be alleviated by the use of additional hardware [57], although even in the ideal case (zero communications idle time) the data transfer bandwidth is still limited by the transputer link. This method severely restricts scalability — the link bandwidth must be shared between a number of DSPs, which would create a tight bottleneck.

iii. Each IMSC011 would use up a link, which would limit the available inter-node connection topologies and overall inter-node communications bandwidth. This method would be more suitable for broadcasting low bandwidth control information to all of the DSP host ports simultaneously.

7.4.2.2 EMI to EMI

Both the DSP and the transputer are capable of transferring data at a rate in excess of

10Mwords^{-1} over their respective EMIs. Furthermore, both processors possess internal (on-chip) memory areas, allowing programs to be stored on-chip. Hence, the EMIs may be used to access data whilst incurring minimal hindrance to instruction pre-fetch — the transputer is able to fetch four instructions in a single instruction cycle from its internal memory, and the DSP possesses a separate internal program memory area and bus, allowing instructions and data to be fetched simultaneously.

It would seem, then, that the fastest way of transferring data between the processors would be to use their respective EMIs.The problem now remains as to how to interconnect the processors both in terms of the connection mechanisms and the network topology.

7.4.3 Interconnection Methods

The most straightforward connection method would be to use a shared bus and/or shared memory architecture, Fig 7.3. The shared bus system is prone to bus bottlenecks and severe communications overhead penalties. Incorporating a block of shared memory helps to ease the amount of idle time experienced by the processors, but bus contention is still a problem — the data bandwidth requirements of the system may easily exceed the available bus/memory bandwidth. Furthermore, only one processor may access the memory at any one time, resulting in delays due to resource contention. Not only does the bus have to handle data traffic, but also the control and test traffic associated with shared bus/memory architectures ("you have the bus" tokens, semaphore test and retry), which in turn reduce the amount of time available to transfer data and increase the communications' management overhead on each

processor [22].

Another drawback of this architecture is that the bus bottlenecking and memory access blocking problems restrict the scalability of the node architecture. The number of DSPs supported by this architecture will be low, since the communications cost is high and so the number of shared RAM accesses should be kept to a minimum. This may be achieved if more code is placed onto individual DSPs, since their internal memory may then be used as intermediate storage areas rather than the shared memory (ie map two processes onto one processor, holding the communicated data in local memory). If an additional DSP is added to a node that is already at or near to its communications bandwidth limit, then the new required communications bandwidth would exceed that available. The extra bus traffic and memory usage incurred by this extra DSP may well severely impede the performance of the node, so that rather than a performance increase, a performance decrease results. Furthermore, the individual processors do not possess any external local memory, restricting their code and data space to internal memory only.

A variation of this architecture is to use dual-ported RAM (DPR) as the shared memory resource, with the addition of local memory blocks for the transputer and DSPs, Fig 7.4. This allows both the DSPs and the transputer to access their own block of memory. Bus bottlenecking is relieved somewhat as the transputer and one of the DSPs may simultaneously access the DPR. However, the DSPs still experience bottlenecking and memory blocking.

Expanding this architecture even further results in the configuration shown in Fig 7.5. In this method, each DSP possesses its own physical block of DPR and a block of local RAM. The transputer is connected to all of the DPR blocks, and also

possesses its own block of local RAM.

The effects of bus bottlenecking are removed in this architecture. As each processor possesses its own bus, the bus access arbitration software may be removed. In fact, the communications control software may be reduced to a matter of checking whether or not a particular area of DPR contains valid data, which may be done quickly and easily. Thus more time is made available to the DSPs to compute data (rather than managing communications), allowing more computing to be carried out in unit time. It may also be seen that each processor may access an exclusive block of RAM, which it may access with no additional communications overhead.

Interprocessor communication now becomes a matter of assigning variables on the transputer. Data pertaining to a particular DSP may be placed at the relevant DPR location using the occam PLACE statement.

Thus, all interprocessor communication is dealt with by the transputer. This allows a DSP to continue computing on a dataset whilst data is being transferred to/from its DPR by the transputer — truly parallel computation and communication. The communication strategy may be defined either statically, ie defined by the transputer program, or dynamically, by specifying the source or destination of a data vector in a header. The latter obviously incurs a larger overhead than the former.This final interconnection scheme was the one chosen for the hybrid multi-processor.

7.5 Memory Requirements

The amount of memory incorporated into the system, and how it is used, is another important design factor. Include too little memory, and the communications bandwidth

could suffer in addition to the size and variety of code capable of being implemented by the processors; too much and money is wasted. It was considered that 8kword of local static RAM (SRAM) would be sufficient for each processor. As the intermediate data storage requirements of most DSP algorithms, and their code kernels, are quite small, then 8kword provides sufficient additional storage space should larger programs or data sets be required.

Dual ported memory is expensive. Furthermore, the DPR is used only to pass data, not to store intermediate data or code. For these reasons, it was considered that 2kword of DPR per DSP would be sufficient memory to test the system viability.

Although the memory size provided will be adequate for most applications, there are some applications that require more memory. Examples are image processing algorithms, which operate on a large data set, and reverberation algorithms, which require many large FIFO buffers. It would be impossible to successfully implement these algorithms with the memory available to the DSP alone. The dual domain DPR partitioning method mentioned above, however, allows the DSP to utilise a much larger memory area with no additional communications overhead. The transputer is able to transfer data from its own local memory, the DPR of other DSPs or from other nodes (over its links). Hence, the DSP is able to access a much larger memory space than it can physically address, Fig 7.6 and Fig 7.7. This simple block move method is suitable for transferring contiguous sections of memory, such as an image, but is unsuitable for algorithms requiring many buffers of different lengths, such as reverberation algorithms. There are two possible solutions to this problem, the first allows the transputer to compound piecewise contiguous areas of its local memory into a single contiguous block transferred to the DPR, Fig 7.8, the second allocates a

separate domain to each non-contiguous area, Fig 7.9.

7.6 Reconfiguration

The configuration, or network topology, of a multiprocessor system can greatly affect its overall performance. Many processor configurations, and many connection mechanisms, are used in contemporary multiprocessor systems. Certain systems possess a topology that cannot be changed either at all or while the system is running — statically configured systems. Others may alter their configuration during run time — dynamically configured systems. Dynamic systems usually incur additional costs in complexity or communication delay.

The physical configuration of the hybrid node is fixed, the only manner in which it may be changed is by adding or removing DSPs. Although this *physical* topology is fixed, however, the *logical* configuration is not. The transputer controls the flow of data around the node, and the software running on the transputer determines the manner in which the data is routed. Hence, the logical configuration of the DSPs may be defined entirely in software, and so may be changed dynamically. Complex memory mapping techniques, ie aliasing, may be used to enhance the performance of some configurations. Example topologies are shown in Fig 7.10.

7.7 Reprogramming

One of the DSP memory mapping modes maps program space into the DPR. This has been implemented to allow the transputer to download programs to the DSP. DSP programs must first be assembled and linked using an appropriate assembler package.

The resulting object files need to be stripped of their headers before they can be handled by the transputer.

The DSP programs to be downloaded by the transputer may either be stored in local transputer memory, or read in from a filing system, over a link. The transputer treats the block of object code as a data vector, and block moves it into DPR.

Once the object code has been read into the DPR, and the semaphore reset, the DSP is able to make use of the code. It is not desirable for the code to remain in the DPR for two reasons. Firstly, an area of DPR, which is a valuable resource, is used as a static store. Secondly, keeping both program and data in external memory areas reduces the performance of the DSP as only one external memory access may be made in an instruction cycle — two external accesses results in a delay in instruction execution. For these reasons, the DSP must move the code from external DPR into its internal program memory, using the MOVEM instruction (move program memory). This move does take some time, but it does ensure that subsequent execution is not impeded by additional external memory accesses.

Although the primary use of this downloading facility is expected to occur during system initialisation, this method does allow for dynamic downloading of code. Thus the code running on the DSP may be changed while the system is still in operation. The DSP will have to go "off line" while it moves the program to internal memory, but this will be a short time compared to the time taken to execute a reasonable size computation kernel.

The local memory of the DSP may also be preloaded with sections of object code at initialisation time, via the DPR, allowing the DSP to access its own local "library" of code.

7.8 Summary (Architectural Overview)

The proposed architecture of the Hymips multiprocessor consists of a node comprising a single IMST801 transputer and a number of Motorola DSP56001 devices. The transputer may communicate with other transputer based nodes via its four serial links. Data is transferred between the transputer and the DSPs through dual ported RAM.

In this architecture, the transputer controls the flow of information around the network. The DSPs are not concerned with where their input data has come from, nor where their output data is going to. This reduces their communications' overhead and allows them more time to perform what they have been designed to do — computation.

The overall communications bandwidth of the node is limited by the rate at which the controller processor, the transputer, is able to access external memory, ie the DPR blocks. The data transfer bandwidth of the node is now a function of how efficiently the transputer is able to decide whether or not an area of a particular DPR block is valid and how quickly the transputer is able to transfer external data once the decision has been made. The most efficient manner in which to transfer data on the transputer is to use its block move facility.

This architecture allows for a high degree of scalability within the node. The actual number of DSPs supported is governed by the data transfer bandwidth of the transputer. As this is not a shared bus system, the performance of each DSP is limited only by the rate at which data can be supplied to it, and is not affected by additional communication management overheads.

In summary, then, this architecture allows for efficient inter-processor communication, as the problems of bus bottlenecking and the additional overheads

associated with control in shared bus/memory systems are alleviated. The scalability of each node is limited mainly by the external transfer bandwidth capability of the transputer. The maximum simultaneous data transfer bandwidth of the node is equal to the sum of the transfer bandwidths of all the processor. This compares with the sum of the bandwidths of the transputer and one DSP using the single block of DPR, and the transfer bandwidth of either the transputer or a DSP using SRAM. As the control software overhead is greatly diminished, more time is available to the DSPs to compute data rather than manage communications.

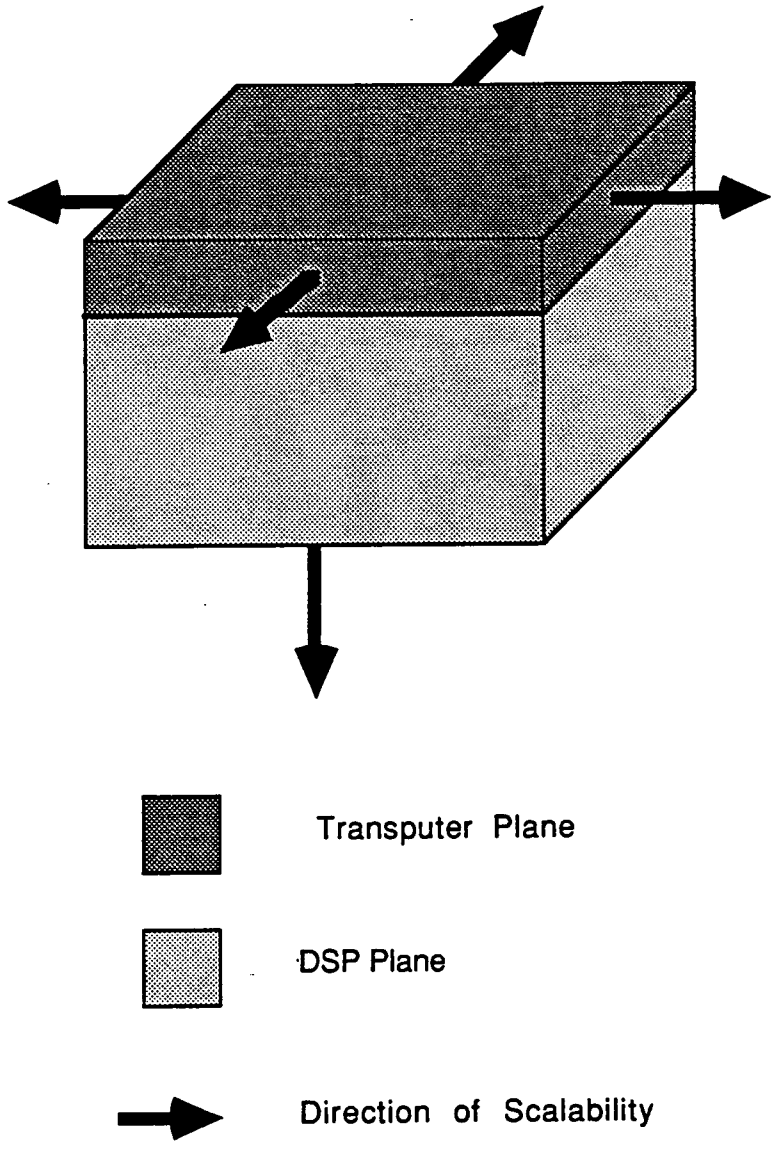


Fig 7.1 Schematic Representation of System Scalability

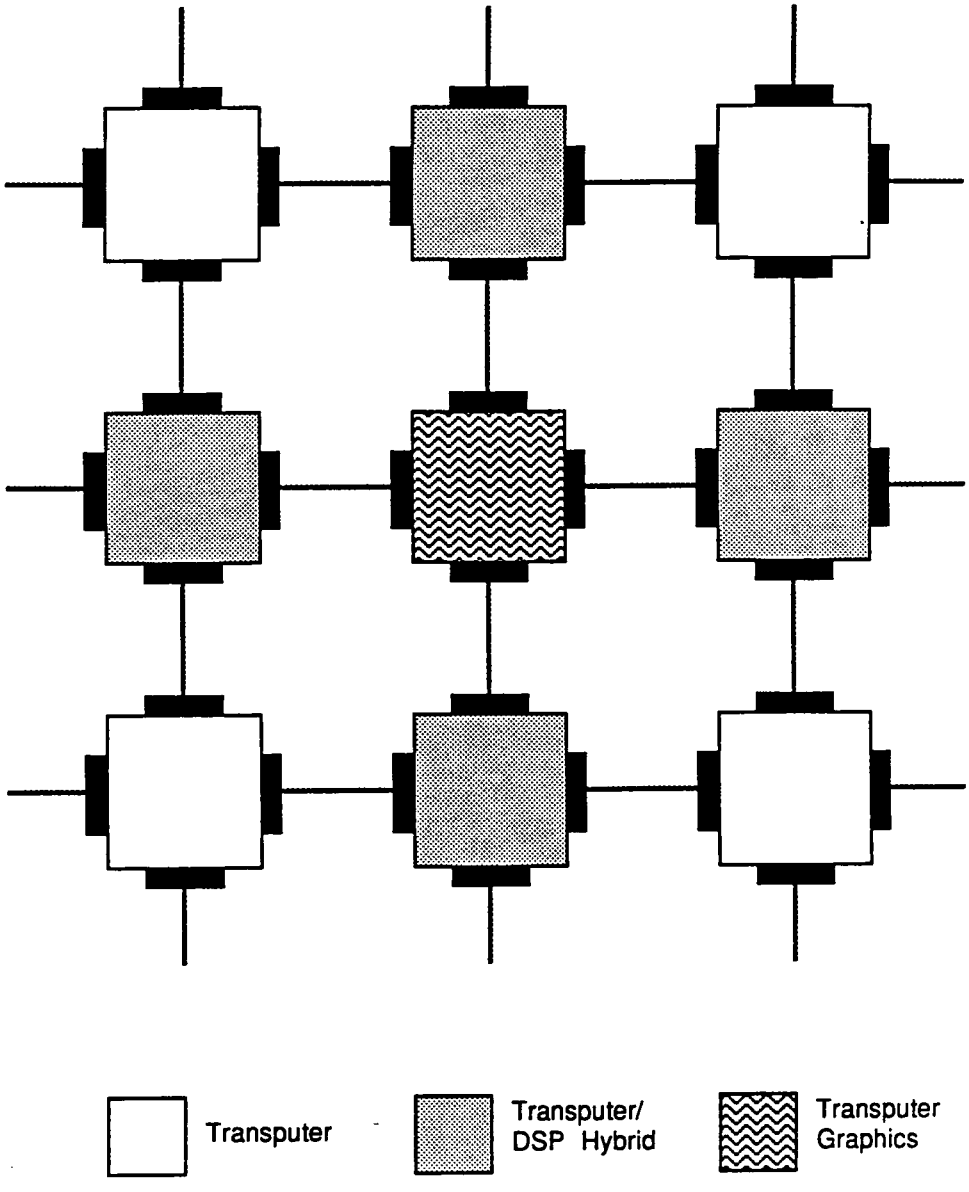


Figure 7.2 An Example Configuration of Nodes

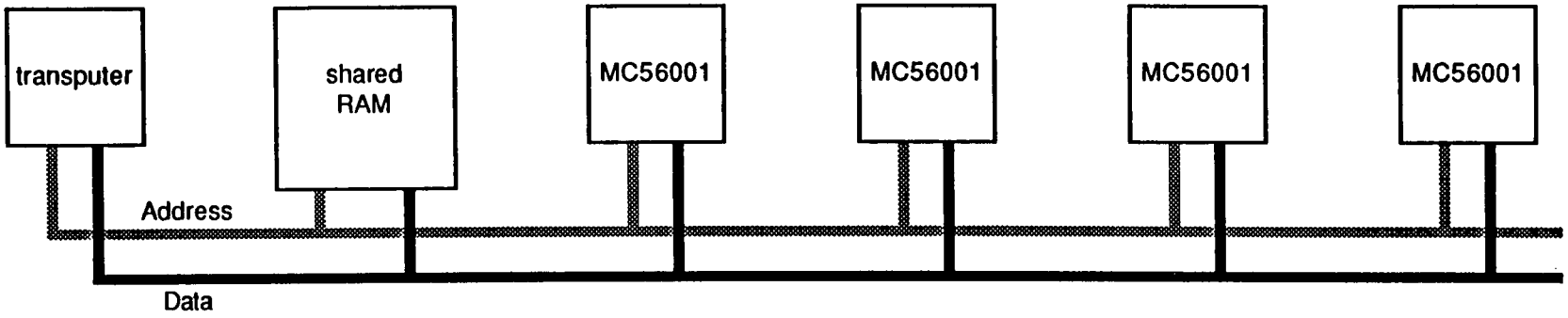


Fig 7.3 A Shared Bus, Shared Memory Architecture

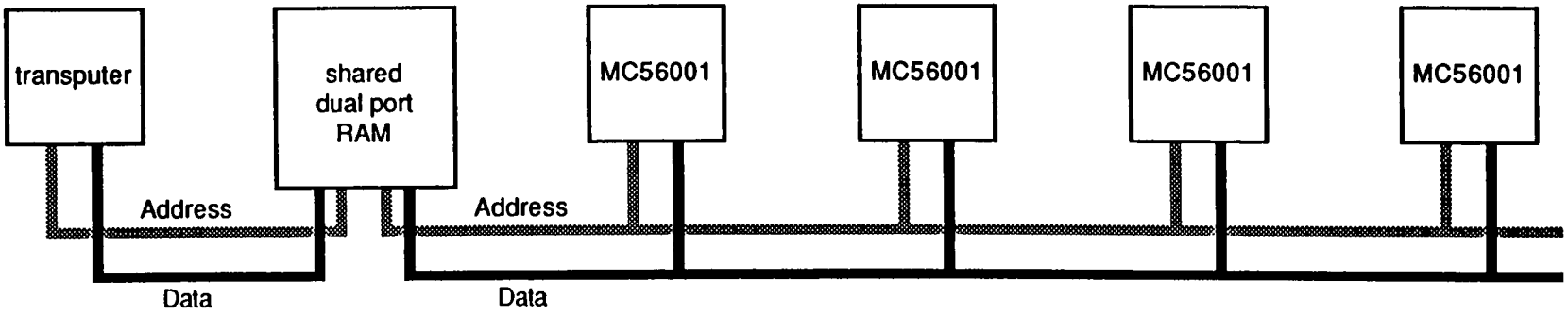


Fig 7.4 A Twin Bus, Shared Memory Architecture Using Dual Ported RAM

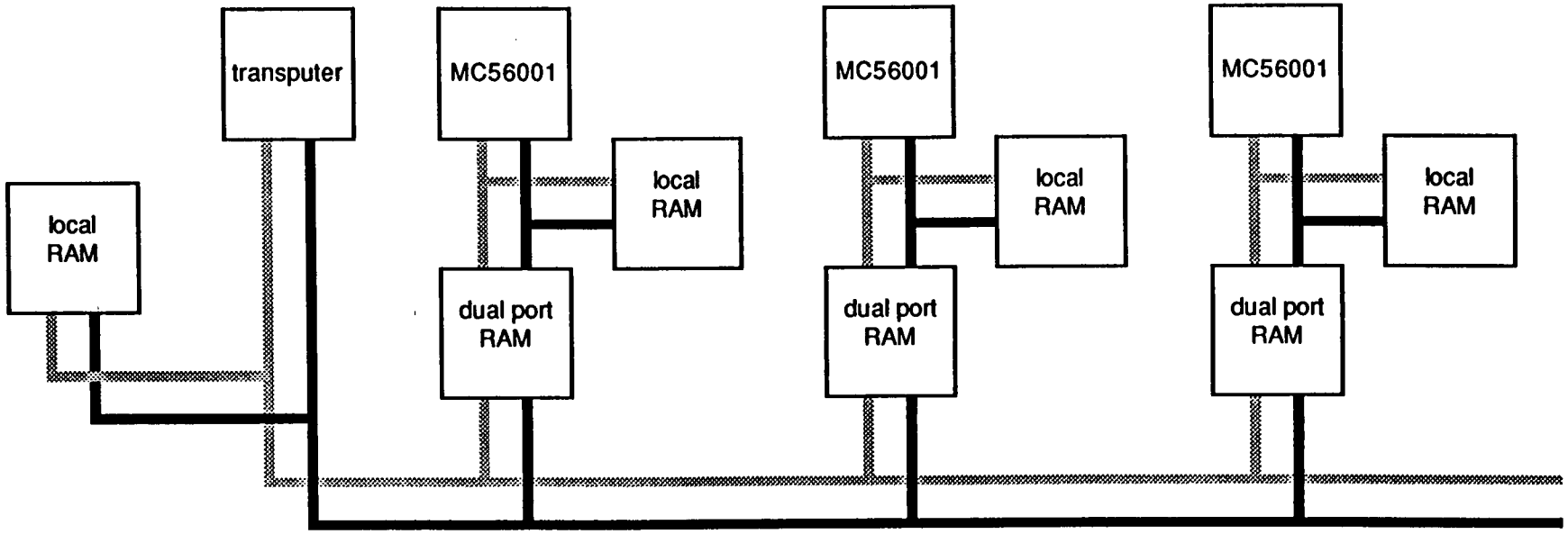


Fig 7.5 A Multiple Bus, Multiple Shared Memory Architecture Using Dual Port RAM

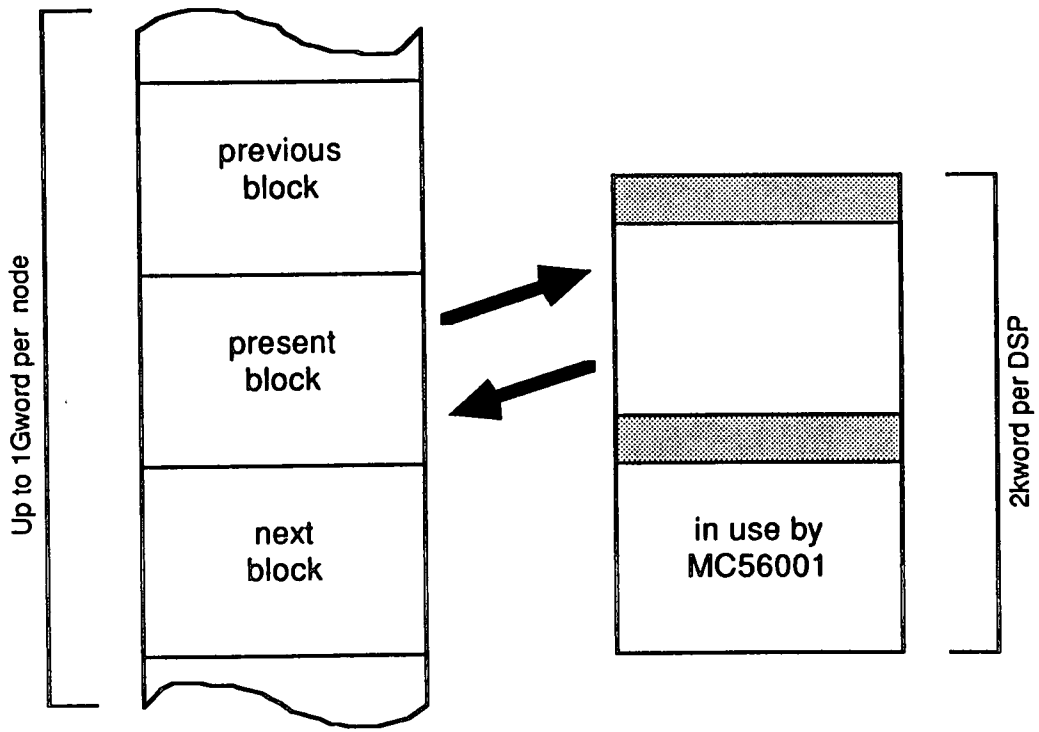


Fig 7.6 A Simple Method of Using Distributed Local memory

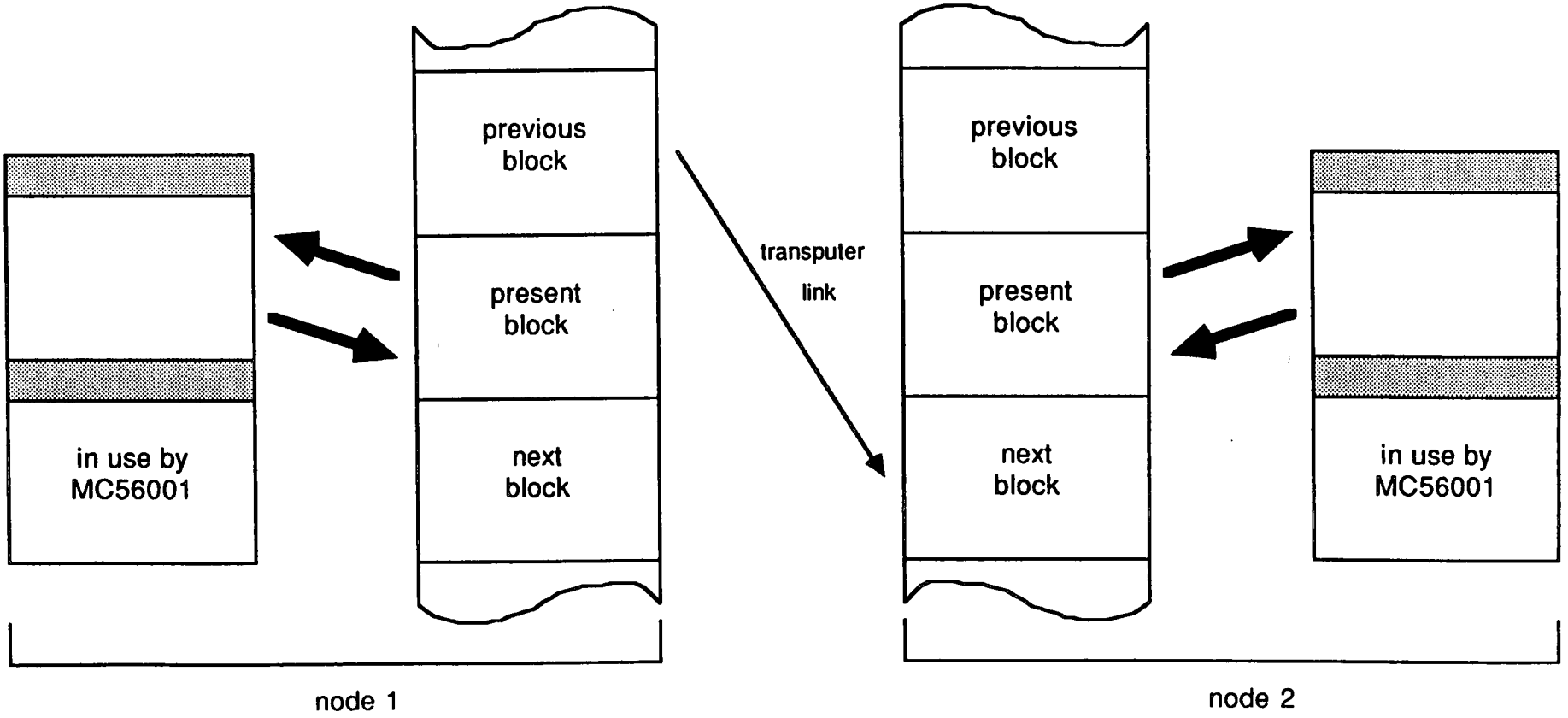


Fig 7.7 A Simple Method of Using Distributed Non-Local Memory

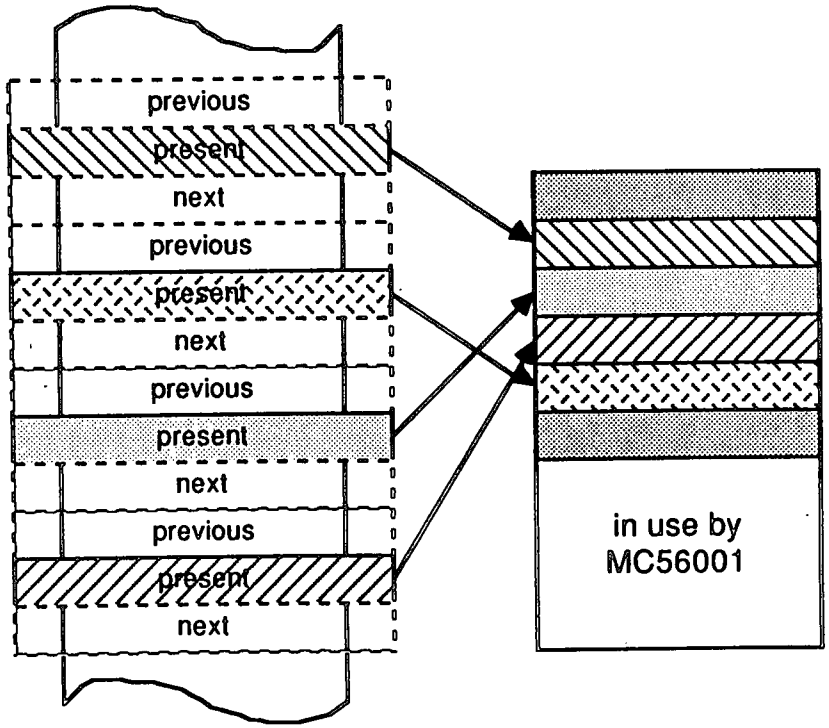


Fig 7.8 A More Complex Method of Using Distributed Local Memory - Compounding Piecewise Contiguous Areas

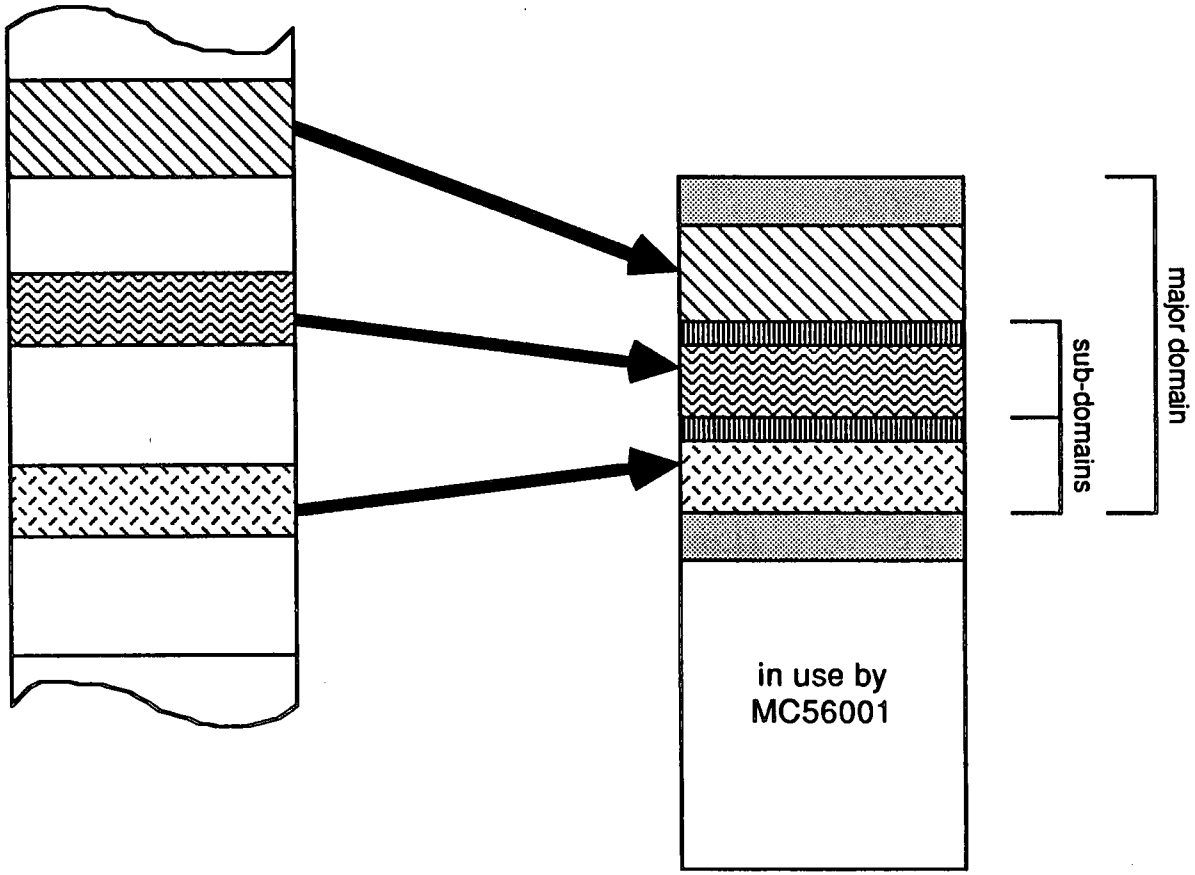


Fig 7.9 A More Complex Method of Using Distributed Local Memory - Heirarchical Semaphores

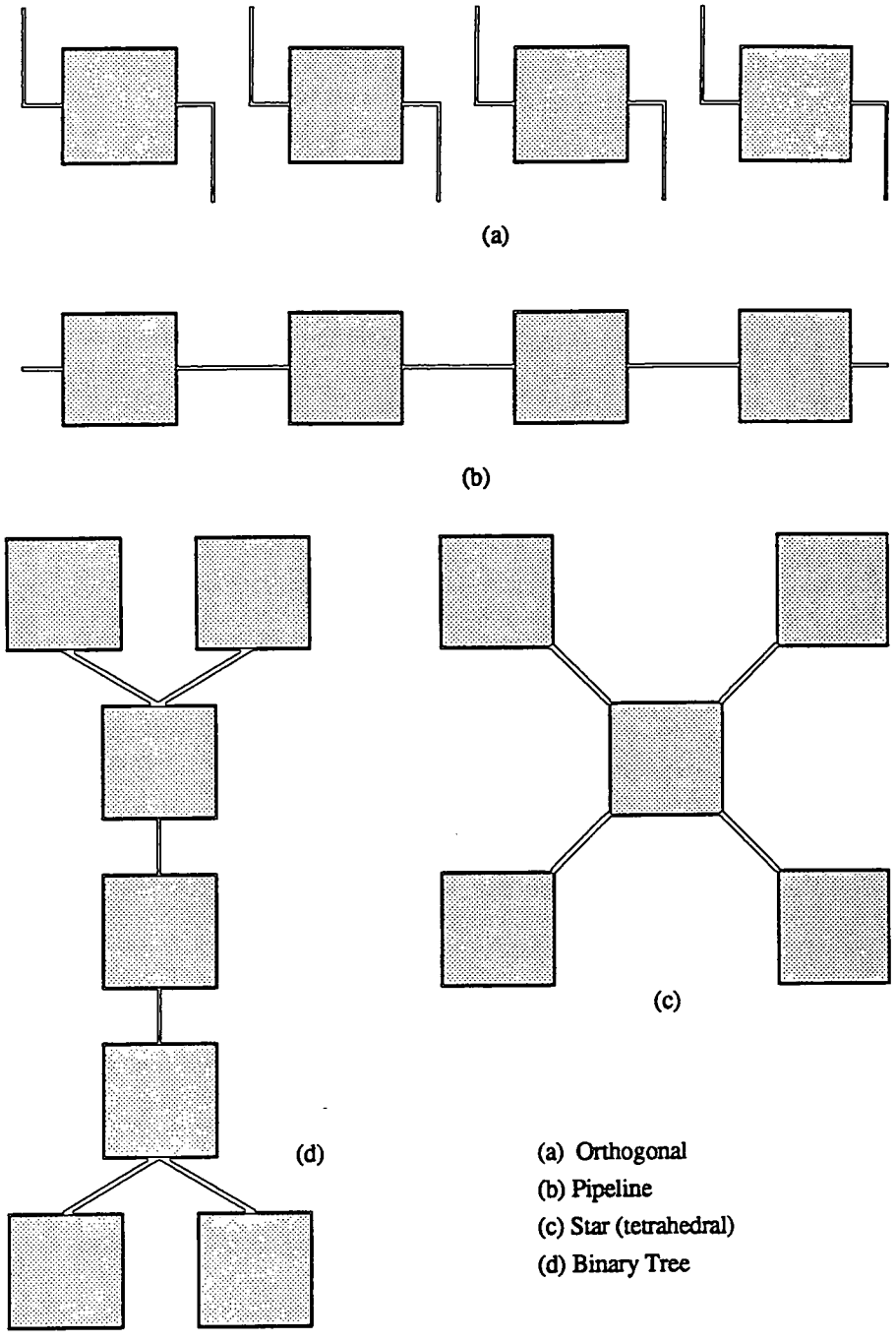


Fig 7.10 Example DSP Network Topologies

Chapter 8

Hybrid Multiprocessor: Implementation

8.1 Introduction

The previous chapter presented the design rationale and an overview of the proposed architecture for Hymips, a hybrid multiprocessor. This chapter goes on to discuss the hardware and low level control software implementation of such an architecture. Although, in principle, the architecture promises to offer high performance and a high degree of scalability, the inherent differences of the constituent processors does cause problems which threaten to reduce the potential overall performance of the multiprocessor. These problems, their causes and their solutions are outlined in this chapter.

Section 2 covers the memory map schemes used by the transputer and DSP56001. A DPR partitioning scheme that allows maximum data transfer rates to be attained is outlined in Section 3. Efficient processor synchronisation and data protection is vital to any shared memory multiprocessor architecture, the method used in Hymips being described in Section 4. Section 5 discusses possible synchronisation coding schemes. System initialisation is outlined in Section 6. Initial processor synchronisation, an important aspect of system initialisation, is covered in Section 7.

Section 8 outlines the construction of a Hymips node, Section 9 providing a Summary.

8.2 Memory Space Partitioning

It has been decided that the highest data transfer bandwidth between processors in this system may be attained through the use of a communication scheme involving blocks of dual ported (shared) memory. However, the manner in which these areas of memory are addressed by the processors, ie the processors' memory map, has significant bearing on the performance of this communication scheme. The memory mapping affects particularly the efficiency with which the transputer transfers data. Furthermore, the processors themselves are to possess an area of local memory, which must also be addressed.

This section outlines the placement of these memory areas in the address space of the processors.

8.2.1 The DSP56001

The DSP56001, with its modified Harvard architecture, may address three independent memory spaces — x-data, y-data and program. These address spaces begin in the on-chip RAM areas, allowing simultaneous access, and are continued externally, where only one space may be accessed at any given time. The processor is allowed access to 8kword of local RAM and 2kword of shared dual ported RAM, each of which need to be placed within the address space of the memory areas.

The dual ported memory is to be primarily used to transfer data. It would seem sensible to map the whole of this memory into the address space of one of the data areas. The only restriction on the placement of the memory should be that it is placed

sufficiently high up to allow the modulo addressing mode to be utilised over the whole DPR. However, in order to aid dynamic programming of the DSP network, it would be useful if a portion of the DPR was placed into the program memory address space. The DPR may thus be accessed in one of two address mapping modes, mode 1 and mode 2. The first maps the whole of the DPR into x-data space, allowing large vectors to be transferred. The second maps half of the DPR into x-data space, and half into program space, reducing the size of data vectors that may be transferred, but allowing DSP programs to be placed directly into program space by the transputer.

The local memory, at 8kword, is large enough to be partitioned between memory spaces. It is important that the addressable program area is contiguous with the on-chip area, in order to allow large programs to overflow from on-chip into off-chip memory. There need be no such restrictions placed on the positioning of the data spaces. In mode 1, then, the local memory is equally divided between y-data and program spaces. In mode 2, the program space addresses 4kword, with the x-data and y-data spaces addressing 2kword each, Fig 8.1. Both memory map maps are defined by the same PAL device.

8.2.2 The Transputer

The transputer may access a signed address space of 1Gword, with 1kword being placed on-chip. Unlike the DSP56001, the transputer stores its data and programs in a single memory space. Both the 8kword local memory and all the DPR areas must be mapped into this single address space.

It is important that the local memory is placed in an area contiguous with the on-chip memory, in order to allow the program and workspace areas to "overflow"

from internal to external memory.

There are a number of options for mapping the blocks of DPR into the address space, Fig 8.2. The most straightforward would be to map each block contiguously into the address space. Another option would be to allow double imaging (aliasing) of the same *logical* locations at two or more different *physical* DPR locations in the transputer address space. Two or more blocks may be aliased to the same address, allowing the transputer to write data to more than one DPR simultaneously, increasing the data transfer bandwidth from the transputer to the DSPs. Of course, non-aliased DPR areas must be used for the transfer from the DSPs to the transputer. Another option would be to place the input and the output sections of the DPRs at contiguous logical addresses. This would allow entire input or output vectors to be moved in a single block move. These are only three of the many possible memory map configurations, some of which are general, some of which would be specific to a particular application. Any particular memory map may be implemented by the use of a PAL.

8.3 Dual Ported Ram Partitioning Schemes

The efficient use of the DPR blocks is essential if a high communications bandwidth is to be attained throughout the node. This section examines the manner in which the individual blocks of DPR may be partitioned. Communication synchronisation occurs through the use of semaphores, which will be discussed in the next section.

The simplest partitioning scheme is shown in Fig 8.3. The DPR contains one domain, controlled by a semaphore, which contains either input or output data. This partitioning scheme requires the use of an additional block of DSP local RAM, acting

as a buffer, Fig 8.4 [21]. Transferring data to and from this additional memory incurs unacceptable overheads.

An alternative partitioning scheme is depicted in Fig 8.5. Here, the domain is split into two sections, one exclusively containing data passing from the transputer to the DSP, the other data from the DSP to the transputer. As both input and output data reside in the DPR, there is no need for the DSP to utilise local memory as a data store.

Both of the above schemes use only a single semaphore, allowing only one processor to access the DPR at any given time. The DPR is thus being used in a similar way to shared single ported memory, very little dual ported capability is being used — the only manifestation being that no access arbitration is required to read the semaphore, so that the "blocked" processor's attempts to access the semaphore do not interfere with the operation of the "unblocked" processor. An important consequence of this is that the processors experience a large amount of idle time, when they are continually testing and failing the semaphore.

A compromise solution would be to add a local data store to the second scheme outlined above, Fig 8.6. This would allow more efficient overlapped communication and computation than the first scheme. Once the DSP has moved the i/o data from the DPR into its local memory and begins its computation on that data, the transputer is free to access the DPR — thus overlapping computation and communication. However, this scheme still requires a lot of unnecessary transferring of data to and from the local store.

The solution is to partition the DPR into two domains, Fig 8.7. This partitioning scheme allows concurrent access of the DPR by both processors. There

is no need for the DSP to transfer the data to a local store. Maximum transfer bandwidth is attained if each domain is further divided into input and output areas, Fig 8.8. When the DSP is operating on the first domain, the transputer is able to operate on the second, and vice versa. While the DSP is computing on data set n, the transputer is able to transfer the input for data set n+1, and the output from data set n-1, Fig 8.9.

There may be some "idle time" experienced by the processors, depending on the number of DSPs, the length of the code segments that they are running and the size of the data vectors, but this may be reduced to a minimum by using relevant task allocation and scheduling algorithms.

8.4 Communications Synchronisation

Data is transferred through areas of shared memory in this system. In order to allow a high communications bandwidth to be attained, dual ported memory is utilised. It is important with shared memory systems, however, to ensure data integrity. This is often ensured by the use of semaphores, which control access to a particular area of memory [21], [83]. There are many semaphore protocols in use today; this system makes use of a protocol based on the test-and-set method [6], [20]. In order for these protocols to operate successfully the processor must be able to execute certain "atomic" instructions, and indeed the DSP56001 does so. However, the transputer was designed specifically to operate using a different communications mechanism, and does not support these uninterruptible instructions. The standard test-and-set protocol has been modified in order to allow the use of interruptible instructions and hence avoid data corruption. Semaphores and shared memory methods have been applied to

interprocessor communication for transputers [84], [85], [86], but these have conformed to the CSP communication model.

This section first examines the operation of the dual ported memory. The standard test-and-set semaphore protocol is then described, and the problems encountered through using the transputer instruction set are highlighted. Finally, the modified protocol is described.

8.4.1 Dual Ported Memory

It is possible to allow more than one processor to access single port memory, but this method allows only one processor access at any given time, and requires additional arbitration logic. Multiple accessed single port memory offers no performance benefits.

True dual ported memory allows two processors to access the memory at any given time. An exception to this is when both processors wish to access the same location: one of the processors is forced to wait until the other has completed its access cycle, eliminating the risk of data being spuriously overwritten. Such contention is normally flagged by a "busy" pin, which is driven by on-chip address sensing arbitration logic.

8.4.2 The Test-and-Set Semaphore Protocol

Let a semaphore value of zero indicate that the domain of the semaphore is unlocked, ie. is free to be accessed, and a value of 1 indicate that it is locked, ie. is in use. The test-and-set method is depicted in Fig 8.10. The value of the semaphore is read into a local variable (`Local_Dummy`). The semaphore is then set to one, in order to lock the domain (if it is not already locked). The original value of the semaphore is tested. If

the original value was zero, indicating that the domain is unlocked, then a section of critical code is executed, after which the semaphore is reset to zero, unlocking the domain. If however, the original value was one, indicating that the domain is locked, the process may not access the domain. Two options for continued execution in this case are firstly to retest the semaphore and secondly to enqueue the present process and dequeue another [22], [23]. For such a protocol to work correctly, it is essential that no other processor is allowed access to the semaphore between operations i and ii of Fig 8.11 — the read and set instructions must be compounded into a single uninterruptible instruction.

Consider the situation when this is not the case and that the bus is released between operations i and ii, ie the read and set operations are interruptible. The following situation could arise, depicted in Fig 10. The original value of the semaphore is zero, which is duly read in by processor 1. Consider, now, that another processor, processor 2, is allowed access to the semaphore between the read and write operations of process 1. This second process will also read the semaphore as zero, indicating that the domain is free. Thus, two processors are allowed to operate on the same domain simultaneously. Data corruption is almost a certainty in this situation, and so using the semaphore as a means of both process synchronisation and data security breaks down. This situation would probably arise very seldomly in most systems using interruptible instructions. Hence run time testing of such systems is unreliable — data corruption may not occur for quite some time. The manner in which processors test semaphores is an important consideration when porting code from one system to another.

The DSP56001 does support uninterruptible instructions, although not the test-

and-set variety. The transputer, however, supports no such instructions. For this reason, a modified approach had to be developed.

8.4.3 The Hybrid Semaphore Protocol

The problems presented in the previous section are manifest in any shared memory multiprocessor system using processors that do not possess uninterruptible read and set instructions. In the type of protocol already mentioned, the state of the semaphore indicates whether or not its particular domain is locked or unlocked. This is sensible, since many processes or processors may wish to access the domain in any given multiprocessor system. However, as any physical block of DPR is shared between only two processors in this system, a different type of protocol may be implemented.

Rather than indicate whether or not the domain is locked or unlocked, the semaphore indicates which of the two processors may access the domain. Together with the on-chip arbitration of the DPR forcing wait states when required, this protocol ensures data and synchronisation validity. The pseudo-code of this protocol is depicted in Fig 8.12. It may be seen that the two main differences between this protocol and the test-and-set protocol are firstly that the state of the semaphore determines which processor may access the domain, not whether the domain is locked or not (the domain is always "locked" in the test-and-set context) and secondly, as a consequence, there is no need to lock the domain by setting the semaphore.

Using this protocol, there is no way that the two processors can access the domain at the same time. This protocol allows processors that do not possess uninterruptible instructions to efficiently utilise dual ported memory.

8.5 Semaphore Implementation

It is important that the code running on the transputer is written as efficiently as possible, incurring minimal performance overheads, if the system is to operate at its maximum potential performance. The transputer will execute its semaphore test code N_D or $2N_D$ times for each of the N_D DSPs' one or two, and so any additional cycles will add N_D or $2N_D$ cycles to the whole of the test and transfer sequence. If the extra cycles cause the execution time of the whole loop to exceed a particular critical value then the DSPs will experience idle times.

Three possible versions of the semaphore test code are discussed below. The first is written in Occam2 and will be used as the base from which other versions may be compared. The second two versions are written in transputer assembly language.

8.5.1 Occam2 Version

This routine, shown below, uses an `IF` construct to test the value of semaphore s1. The transputer use a 32bit word, whereas the DSP56001 uses a 24bit word. In order to preserve the parity (+ve or -ve) of the DSP data the three DSP data words are mapped into the upper three bytes of the transputer's data word. Hence a value of `$1` (Hex 1) on the DSP is equivalent to a value of `#100` (Hex 100) on the transputer (the `$` prefix indicates a DSP hexadecimal value, the # prefix indicates a transputer hexadecimal value). This is the reason that s1 is tested for `#100` and not `#1`. If the semaphore is set, the relevant i/o is performed and then s1 reset. If s1 is not set, then the program comes out of the `IF` construct and continues.

```
IF
  s1 := #100 (256)
    SEQ
      ... perform input
      ... perform output
      s1 := 0
  TRUE
    SKIP
```

s1 PLACED at Occam2 word address #7FF.

The input/output and semaphore reset code is identical in all three versions presented here, and so will not be discussed further. The critical part of this code is the conditional section, which will be considered in more detail.

The assembled form of the Occam version is shown below:

```
MINT              1          Load in the value of s1
LDNLP 2047(#7FF)  2+2        using indirection.
LDNL  0           2
EQC   256 (#100)  2+2        Compare this value to
CJ    23          2/4 +1     #100 and jump if required
```

This takes 14 or 16 cycles, depending on whether or not the jump is taken. The same level of prefixing for the other semaphore addresses will be experienced only if their addresses lie between #100 and #7FF. For addresses above #7FF, which will normally be the case, an extra prefix will be used to read in the semaphores' addresses, which will add another instruction cycle.

About 50% of the time taken to run this section of code is used to generate the address of the semaphore and to read it in. This method is the most general, and is of the type typically produced by the Occam compiler as it is does not assume that the addresses of variables are known at compilation time.

8.5.2 Assembler Version 1

Whenever the address of a variable is known a priori, another method may be used. Occam2 provides no provision for this method and so transputer assembly language must be used.

```
LDC         -2147475457 (#80001FFF) 1+7 (byte address of s1)
LDNL        0                       2
EQC         256                     2+2
CJ          23                      2/4+1
```

The absolute (machine) byte address of the semaphore is loaded in directly using LDC. As the machine addressing scheme must be used, however, a small Occam address is translated into a large negative machine address. Many prefix instructions are needed to read in such a value, which is the reason that this version of the code takes more cycles to complete than the previous version.

These additional prefix instructions may be avoided by placing the semaphore addresses higher up in the memory map. If machine addresses between #0 and #F are chosen, then no prefixing is necessary to produce the address.

```
LDC         #8          1
LDNL        0           2
EQC         #100        2+2
CJ          23          2/4+1
```

This section requires ten or twelve instruction cycles. Furthermore, using a value of between #0 and #F as the operand of the LDNL instruction allows fifteen possible offsets for each of the sixteen possible values specified on the LDC instruction, allowing a total of 31 locations to be accessed with no prefixing overheads.

Although this implementation is quicker, it does require additional address mapping to map the semaphores into the DPRs. The semaphores occupy a single contiguous block of transputer memory starting at logical machine address #0. These must be mapped into individual words occupying physical DPR locations.

8.5.3 Assembler Version 2

The above method simply uses a different addressing technique to access the semaphore. The test section is essentially the same as that used in the Occam version. A different approach is used in the following code.

```
LDC         0-15        1
LB                      5
CJ          23          2/4+1
```

This method requires nine or eleven instruction cycles. Again, the semaphores are seen to reside in a contiguous block occupying the first sixteen *bytes* of positive machine address space. Each semaphore occupies a single byte in address space, necessitating the use of the "load byte" instruction. As the semaphores are treated as bytes, ie as word subsections, then the transputer no longer needs to read in a shifted version of the DSP data word. The byte values may take on boolean values. Hence, a semaphore byte may be read in by the transputer and used as the operand to the "conditional jump" instruction, which eliminates the need for the "equivalence" instruction and so saves cycles. The number of semaphores that may be accessed with zero address prefixing is limited to only 15, however. To ensure a uniform execution time for a larger number of semaphores, the semaphores themselves may be placed between byte machine addresses #10 and #100, which requires a single level of

prefixing.

The address decode scheme is the most complex. as not only does the contiguous block of bytes need to be mapped over to discrete areas of DPR, but because the transputer uses a word orientated addressing scheme on its EMI, and the lack of additional strobes on the T801 EMI, then semaphores must be placed in particular byte locations in order to avoid "overlap" on the data bus and hence semaphore corruption.

8.6 Initialisation

The initialisation of any asynchronous multiprocessor system is far from straightforward. Care must be taken to ensure that each processor executes its initialisation routine in sequence with all the other processors in order to prevent the occurrence of spurious or erroneous events. In the system discussed here, each processor possesses its own local ports and memory in addition to an area of shared memory. Hence, a processor must both initialise its own local environment and synchronise with a global initialisation routine, which involves all the processors in the system. The transputer controls the data flow around the system, and so it is logical that it should also control the global initialisation procedure.

The method of synchronisation is non-trivial and is treated in the next section. This section describes the local initialisation procedures of the IMST801 and DSP56001, binding them into a global initialisation procedure that takes the system from its boot state to a fully initialised and operational state. Firstly, however, the bootstrap routines of the processors must be described.

8.6.1 DSP56001 Bootstrap Routine

The DSP56001 possesses a special area of internal program ROM, which it maps into its memory space upon power up. This read-only routine begins to load in executable code from either the external memory interface or the host port, depending upon the state of data line D23. The code is read in byte wide sections and fills internal program memory from the lowest location upwards. When all the code has been loaded, the bootstrap ROM is mapped out of memory space, and execution jumps to the start of the loaded code.

8.6.2 The IMST801 Bootstrap Routine

When booting, the transputer may receive its code either over a link, or from a byte wide ROM placed on the external memory interface. As the transputer is to be connected to a host transputer, via a network of transputers, then the boot from link option is used.

8.6.3 DSP56001 Initialisation Procedure

The code for the DSP56001 could be stored in PROM, and loaded in during the bootstrap routine. However, if this were the case then the code running on the DSPs would be fixed by the PROM. A more versatile approach would be to allow the DSP to transfer programs held in DPR to its internal program memory area. The programs could then be transferred by the transputer from, say, a DOS based file system to the DSPs. This is the approach used in the Hymips system, and effectively constitutes a secondary bootstrap routine. The code used to initialise the DSP and to safely transfer the code section from DPR to internal memory is held in EPROM, and is listed in

Appendix E. A flow chart representation of this code is shown in Fig 8.13.

This code is placed at the bottom of the program memory space by the bootstrap program, its function being to initialise various registers within the DSP, to synchronise with the transputer, to load in code from DPR and to execute it. The code also initialises the interrupt vector space. The operating mode register is then set, and the bus control register set up to define zero wait states for all external memory accesses. The DSP synchronises with the transputer and tests a semaphore in order to determine whether or not it has access to the DPR. If so, then address registers are initialised, and the incoming code moved from external x-data space (DPR) into internal p-space. Execution then jumps to the beginning of the incoming code, which signals to the transputer that it has been successfully loaded by setting a semaphore, and then enters its main loop.

8.6.4 IMST801 Initialisation Procedure

The local transputer initialisation procedure consists of initialising its memory space to zero.

8.6.5 Global Initialisation Procedure

This procedure includes transferring DSP code segments to DPR and synchronising with the DSPs. The node transputer is connected to a host transputer, which supplies the DSP code segments and the input data. The main operations that the transputer must perform are :-

i. To initialise the DPR areas to zero.

ii. To transfer the DSP code segments, together with their associated placement information, to DPR.

iii. To transfer the first set of input data to DPR.

iv. To enter the main execution loop.

The form of the code is shown in Fig 8.14, the code being given in Appendix E. The DSP code segments are stored as DOS files on hard disk. These consist of .LOD files produced by the DSP assembler which have been stripped of their header information. Each DSP program word, of 24 bits, occupies the most significant three bytes of a transputer word as only the upper 24 bits of the transputer data word are written to DPR. The size of the code segment, and the address to which it is to be loaded in DSP internal program memory, is also placed in DPR.

Naturally, the transputer must synchronise with the DSPs at various points, in order to prevent data corruption. The transputer must prevent DPR access by the DSPs until the DPR has been fully initialised. Only when the code segments and associated information have been placed in the DPRs is it safe for the DSPs to access them.

The first synchronisation point, then, is placed after the DPR initialisation section. This is a blocking point — all DSPs must synchronise before the remainder of the code is executed — and corresponds to the first synchronisation point in the DSP EPROM code. The action of the synchronisation code is discussed in the following section.

The transputer then allows each DSP to access its DPR, and transfer its code segment, by resetting the appropriate semaphore.

A DSP indicates that it has completed the load and is running the code by setting a semaphore on one of its DPR domains. The transputer is then able to transfer the first set of input data to the DPR. This operation may be treated as a blocking (data transfers wait until all semaphores are set) or a non-blocking (a transfer takes

place on a domain as soon as the semaphore is set) synchronisation point. The transputer enters its main execution loop.

8.7 Synchronisation

As mentioned above, processor synchronisation mechanisms may be implemented either in hardware or software. Both options are available on the Hymips system.

The hardware mechanism connects one of the transputer links to the host port of each DSP through an IMSC011. The DSPs continually sense the host port, and begin execution when the transputer broadcasts the correct byte value. This method allows all of the DSPs to begin execution at the same time, or allows staggering to be performed. The problem with this method is that it ties up a link that may be required for inter-node communications.

A more general approach is to use a software routine to synchronise the processors, using the DPR to pass the synchronising "token". The most straightforward method would be to pass a token — particular value — to each DSP via the DPR. After they have booted up, the DSPs would continually monitor a particular location of DPR for this token value. Once this value was detected, the DSPs would begin execution of their main body of code. There is a problem with this method, however. The transputer initialises the relevant areas of memory as part of its initialisation routine, which it performs immediately after boot up, before it tries to synchronise the DSPs. The DSPs begin to test the DPR immediately after they have booted up. Now, even if the transputer is able to perform its initialisation routine before the DSPs have started to test the DPR, there is no guarantee that this will always be the case. The transputer may experience a delay in booting up, eg its host takes time in booting

from the link, allowing the DSPs to read the DPR before the transputer has had time to initialise it. Consider that this is indeed the case, and the DSPs are able to read the relevant DPR location before the transputer has had time to set it to a value other than the token value. As the DPR contains random data at this point, it is possible, although unlikely, that the synchronisation location does indeed contain the token value. If this is the case, then one or more DSPs will begin to execute code out of sequence, causing erroneous system behaviour. Although the probability of this happening is low, 2^{-24}, it is still possible, and so cannot be tolerated. Thus, this synchronisation method is not secure. The method used in the hybrid system is outlined below and is shown in Fig 8.13 and 8.14.

The transputer uses a `WHILE` loop to repeatedly change the value of the variable `sync`, which is used to synchronise with the DSP. The execution of the loop is governed by the value of the variable `acknowledge`, which has been reset earlier in the program. Both `sync` and `acknowledge` are placed in DPR.

The DSP loads the initial value of `sync` into one of its accumulators. It then moves the same value into its `x0` register. The contents of the accumulator and `x0` are compared. If they are equal, then the value is loaded into `x0` again and the process repeated. If they are not equal, however, the value `ackval` is written to the location `acknowledge`. The DSP then begins execution of its main section of code.

When the transputer determines that the value of `acknowledge` is equal to `ackval`, it ends the loop and continues with the rest of its code.

The DSP detects a *change* in `sync` using this method, and so does not rely on its initial value. This method is secure.

8.8 Design and Construction

The architecture outlined in this chapter has been implemented in hardware. In order to allow additional processors to be added easily, each processor occupies its own pcb. The boards are connected via a backplane bus.

The transputer card incorporates an IMSC011 link adapter, which may be connected to link 0, in addition to local memory and link circuitry. The links and the IMSC011 interface are accessed via connectors on the front of the board. In order to save space and to allow for as much addressing flexibility as possible, the transputer address decode PAL has been located on a separate board.

The DSP56001 boards incorporate local memory, EPROM, DPR memory and an RS232/TTL level converter in addition to reset and support circuitry. The memory decode PAL is situated on the board, and may support both memory configurations. Ports B and C are accessed via connectors on the front of the board. The provision of a level converter on the board allows devices using an RS232 interface to access the serial port via another connector on the front of the board.

Circuit schematics, net lists and pcb plots were generated using proprietary software running on a PC/AT compatible. The boards themselves are 6 layer, the two innermost layers being used as the power and ground planes, the four outer layers being used to route signal lines. The backplane bus was constructed in-house using a two layer process.

8.9 Summary

This chapter has dealt with the architecture and control software of the Hymips multiprocessor in detail. Memory partitioning schemes have been discussed in relation to the requirements of each type of processor. Shared dual ported RAM has been used as an interprocessor communication buffer, and a particularly efficient method of partitioning this memory in order to allow fully overlapping communication and computation has been described.

Shared memory multiprocessor systems require some means of access arbitration, in order to protect data and allow processor synchronisation. In common with many other shared memory systems, Hymips arbitrates data access through a semaphore based protocol. However, the transputer has not been designed to communicated through shared memory, and does not support the type of instructions required to safely implement the more usual protocols. The nature of the interconnection network architecture has allowed a secure protocol to be developed.

The correct initialisation of a multiprocessor system, upon boot up or reset, is very important, and may be far from straightforward. As there are no control or interrupt signals running between the processors, then Hymips must be initialised through its shared memory. The transputer controls the initialisation sequence. and begins by setting its memory to a predefined value (0). The DSP56001s boot up from their respective EPROMs, which contain self overwriting code that synchronises with the transputer and loads in a program from the dual ported RAM, after it has been placed there by the transputer. The synchronisation is independent of memory contents and is based on a handshake protocol.

The system has been implemented using a number of 6 layer printed circuit

boards, connected over a backplane bus.

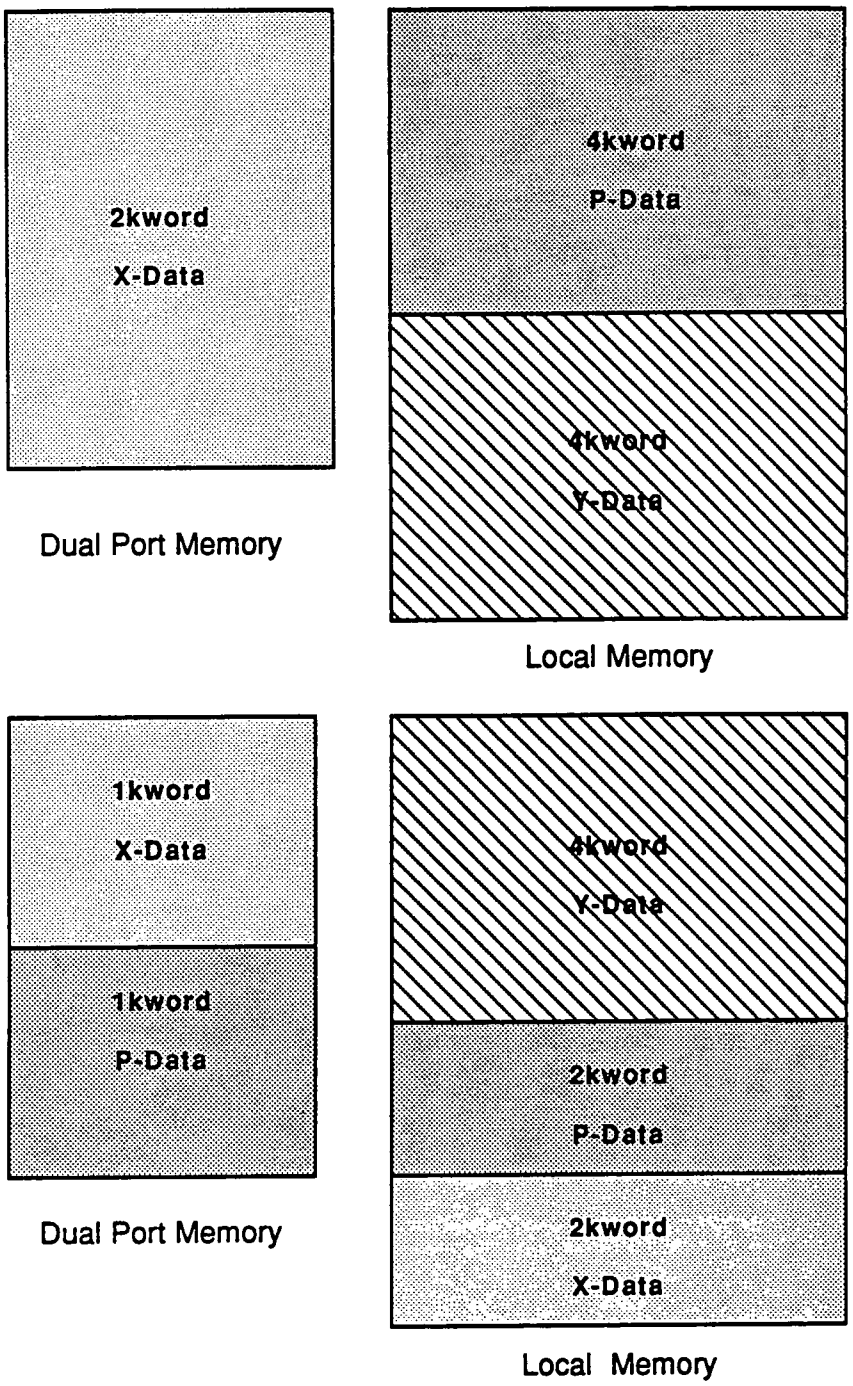


Fig 8.1 DSP Memory Partitioning Options

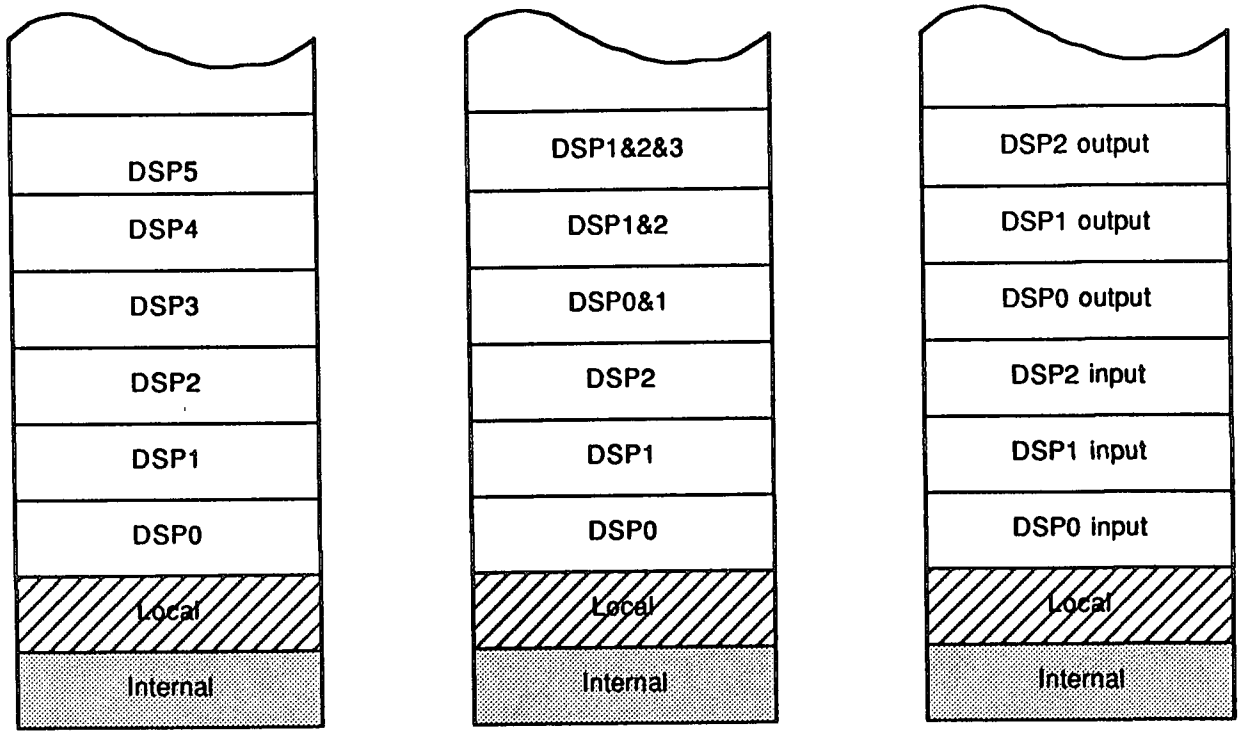


Fig 8.2 Three Possible Transputer Memory Map Options

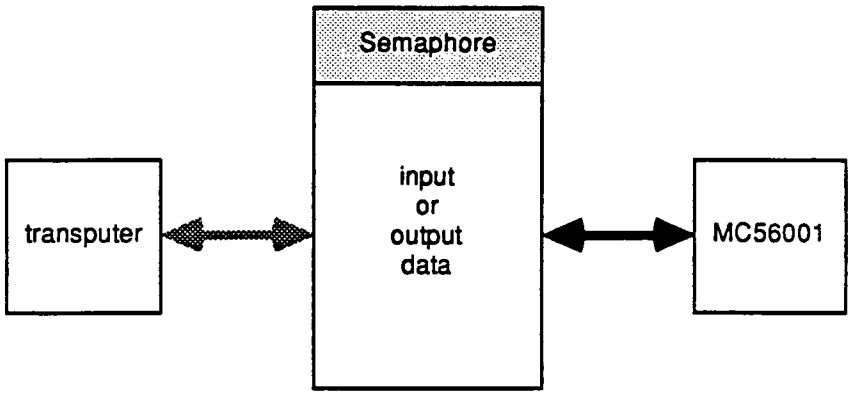


Fig 8.3 Simplest DPR Partitioning

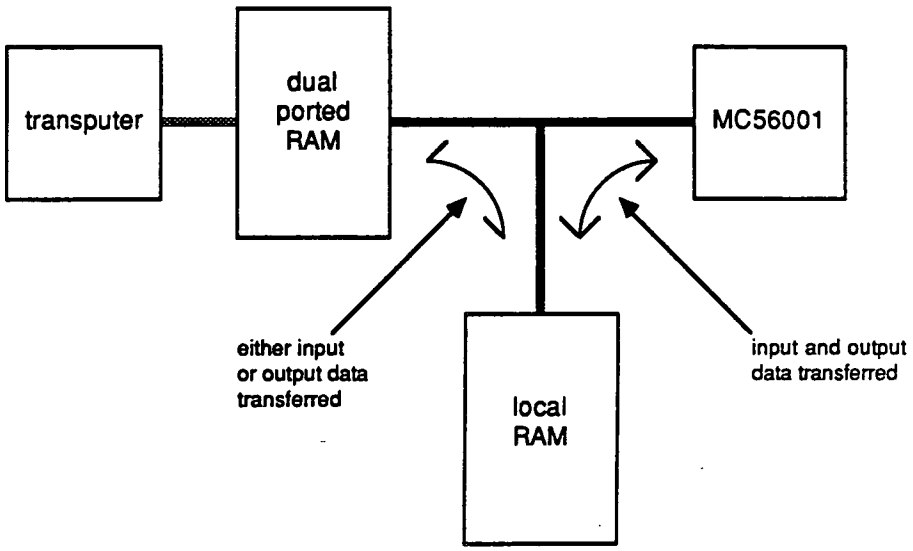


Fig 8.4 Utilisation of a Local Memory Store

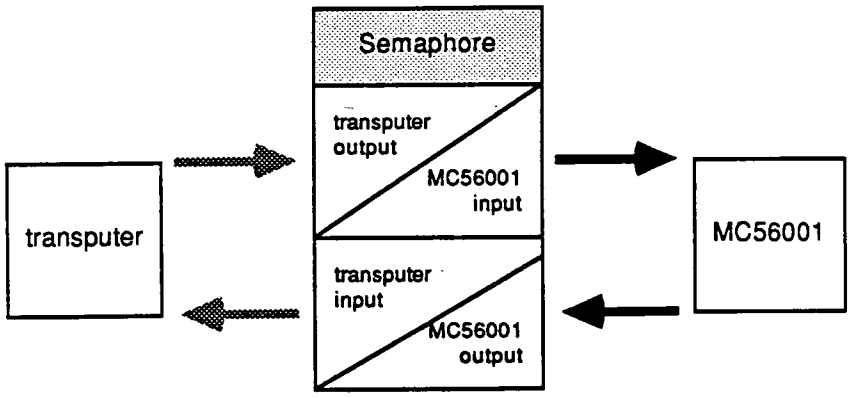


Fig 8.5 Improved DPR Partitioning Scheme

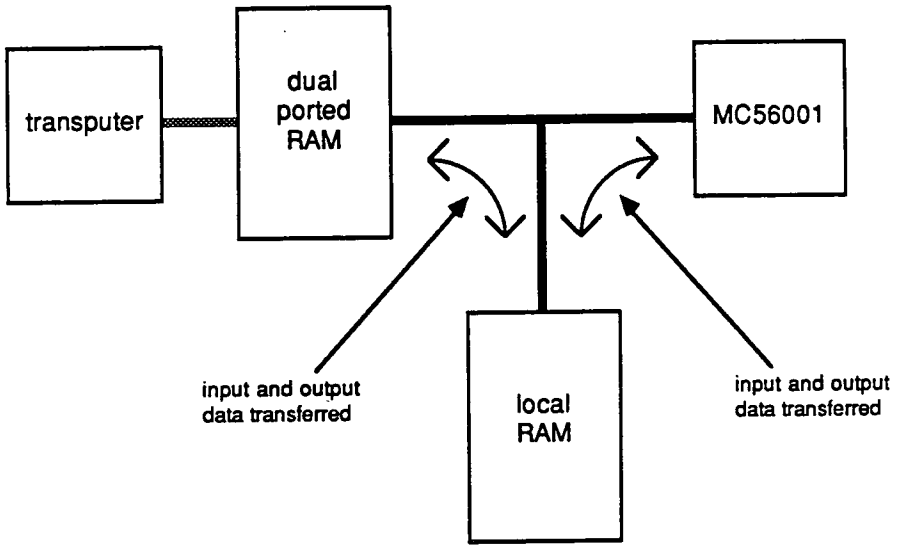


Fig 8.6 More Efficient Utilisation of a Local Memory Store

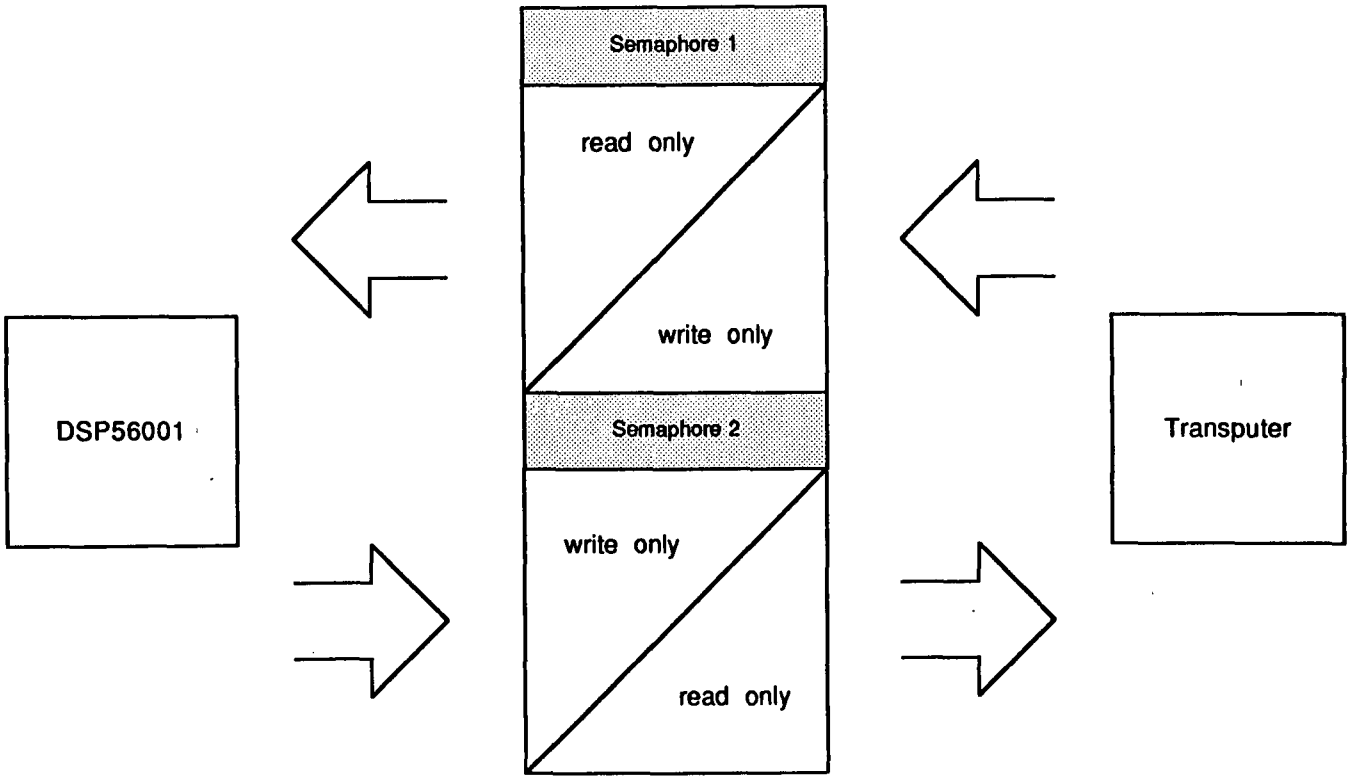


Fig 8.7 Single i/o Domain DPR Partitioning Scheme

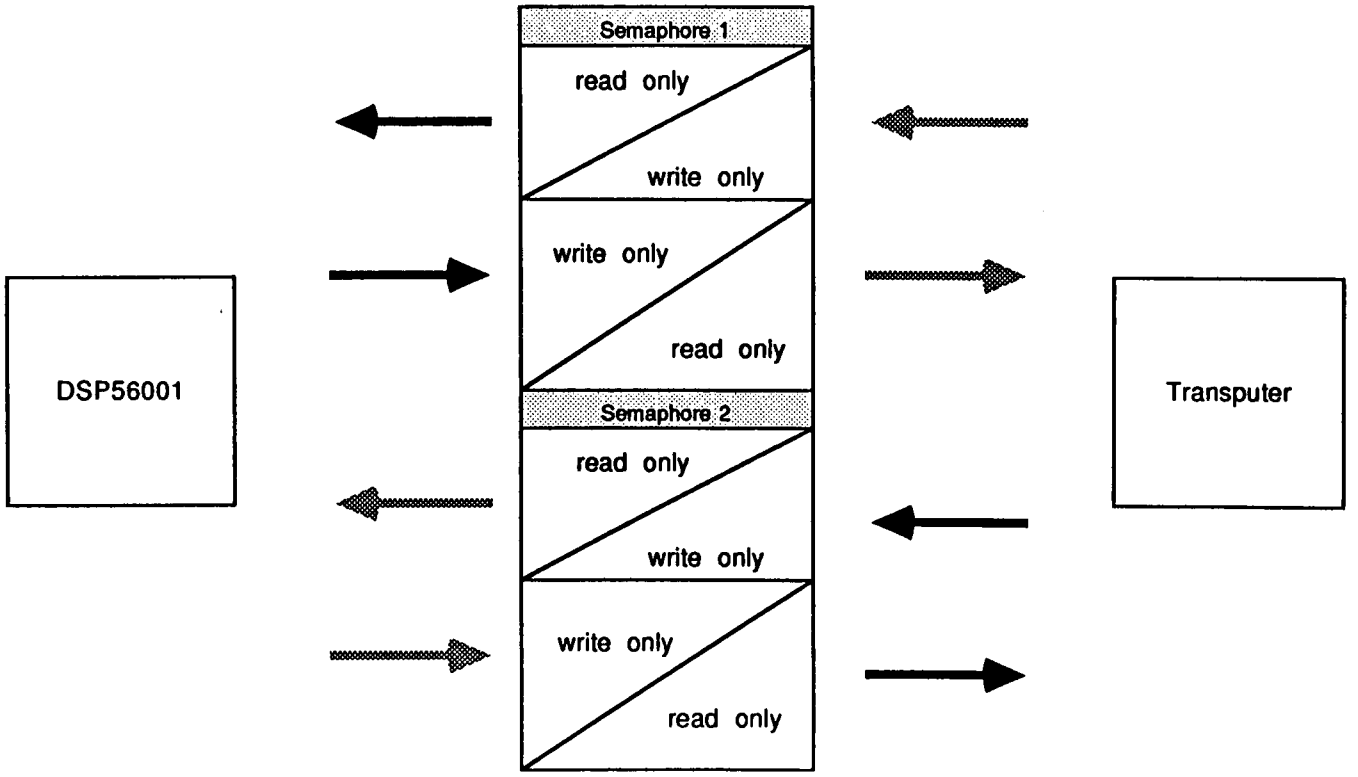


Fig 8.8 Twin i/o Domain DPR Partitioning Scheme

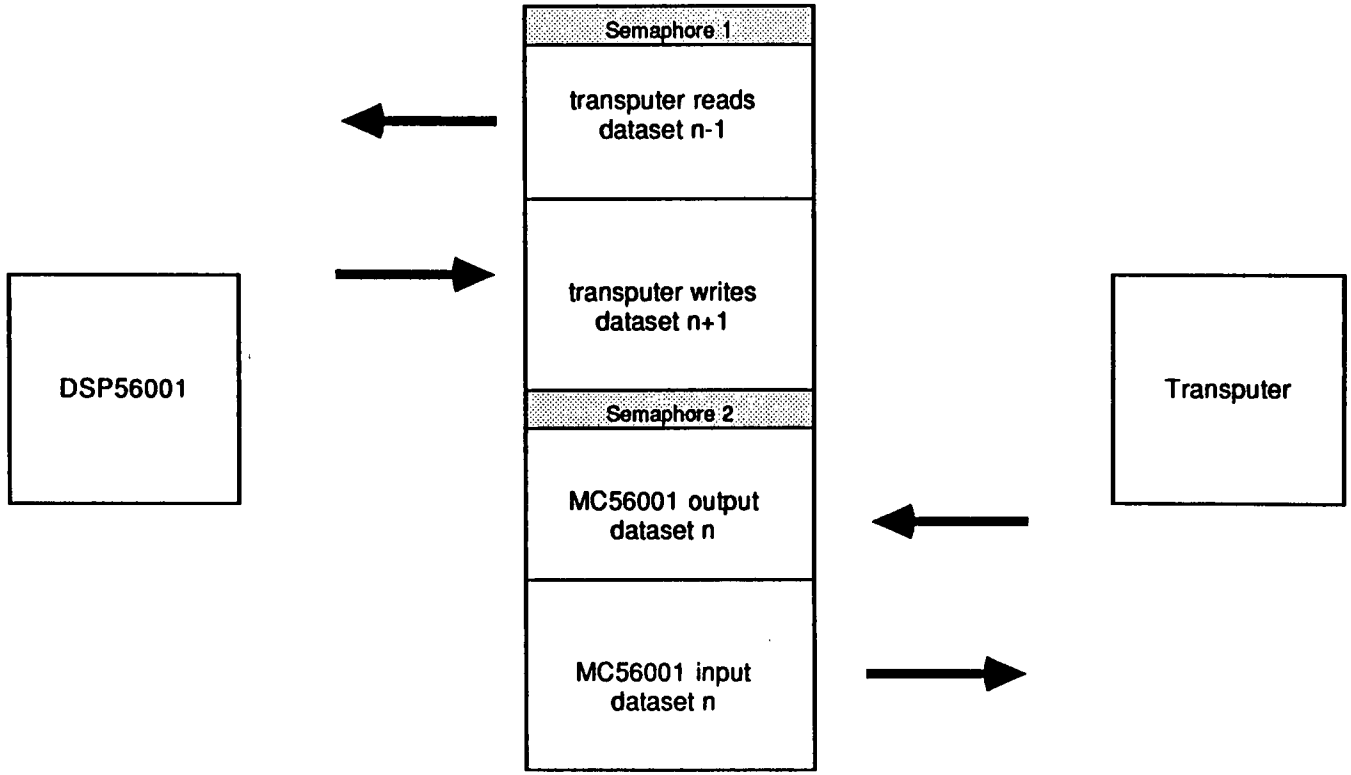


Fig 8.9 The Order in which Datasets are Transferred over Dual Port RAM

```
1     Local_Dummy := semaphore_value
2     semaphore_value := 1
3     IF
          Local_Dummy = 0
              perform action on domain
          Local_Dummy = 1
              do not take action on domain
```

Fig 8.10 Pseudo-Code for the Atomic Test-and-Set Method

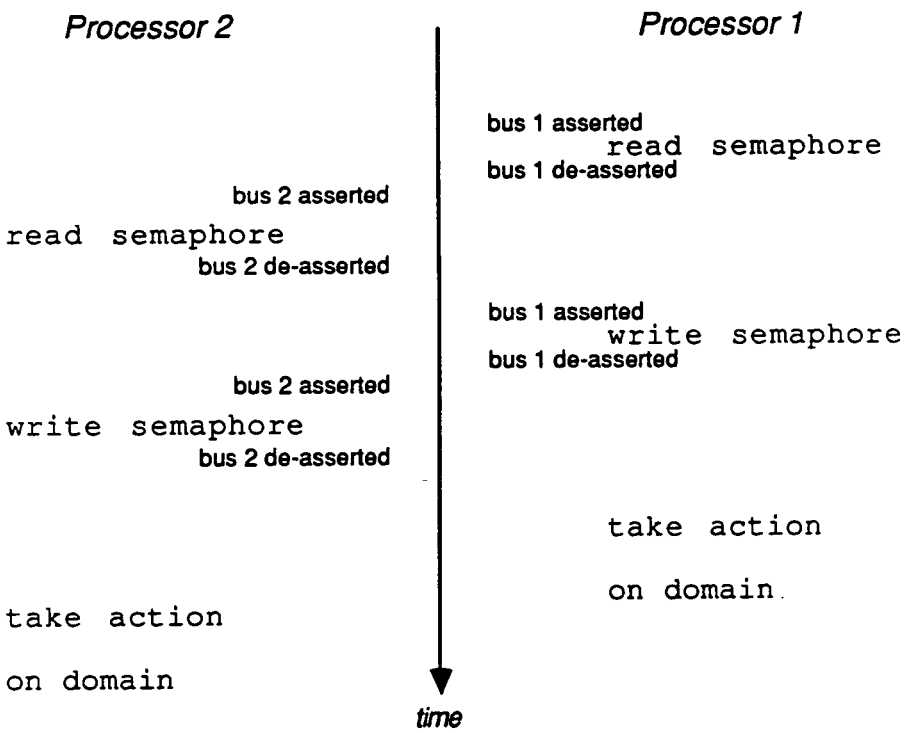


Fig 8.11 Potential Erroneous Behaviour when Non-Atomic Instructions are Used

```
1     Load semaphore_value
2     IF
         semaphore_val = processor1_go
             perform action on domain
             semaphore_val := processor2_go
         semaphore_val := processor2_go
            do not take action on domain
```

Fig 8.12 Pseudo-Code for the Hybrid Semaphore Protocol

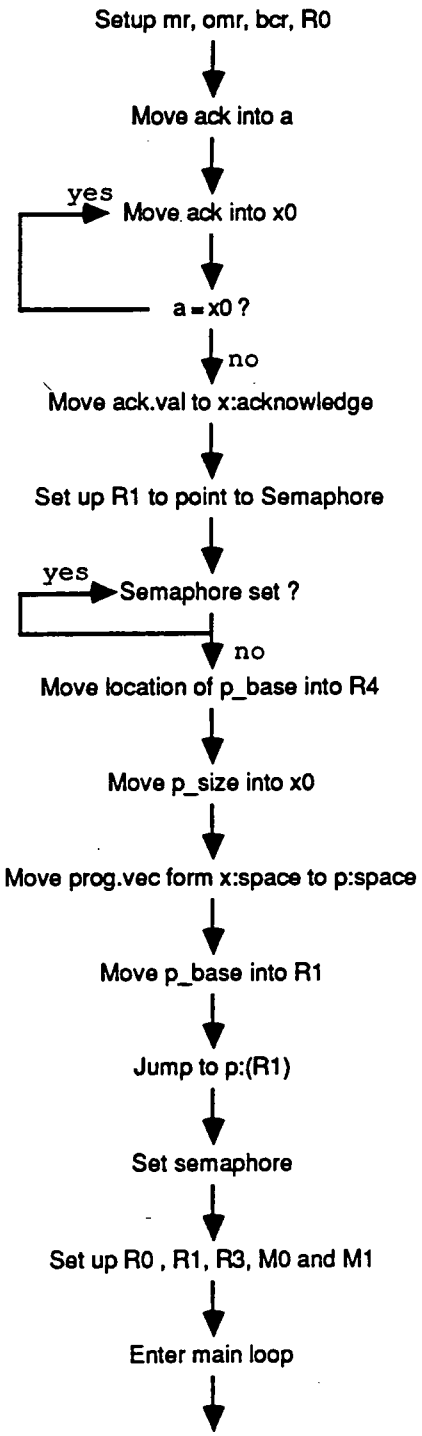


Fig 8.13 The DSP56001 Initialisation Routine

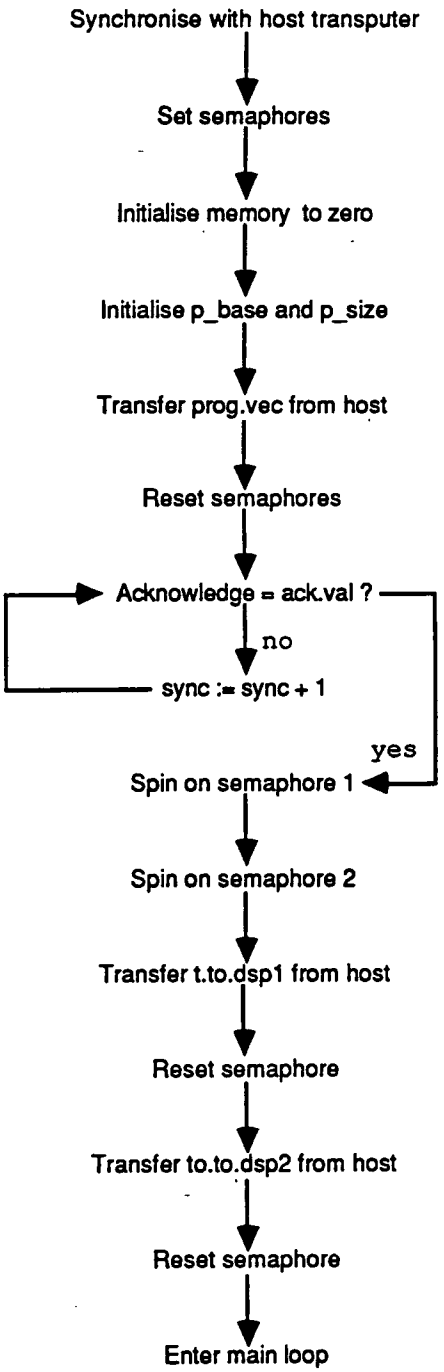


Fig 8.14 The Transputer Initialisation Routine

Chapter 9

Hybrid Multiprocessor: Performance

9.1 Introduction

The previous chapter described a hybrid multiprocessor system, which uses areas of shared dual ported RAM (DPR) to efficiently transfer data between a transputer and several digital signal processors (DSPs). The transputer and the DSPs, which are arranged in a sub-network, constitute a node. Many nodes may be connected together using transputer links. Although the potential computational performance of such a system is a linear function of the number of DSPs, the overall performance is limited by the intra- and inter-node communications bandwidths. The inter-node bandwidth is fixed by the transputer links; the intra-node bandwidth (the rate at which the transputer is able to supply data to any particular DSP) is not constant and depends upon such factors as the number of DSPs in the node, the code they are running and the transputer overheads associated with each transfer.

This system has been designed primarily to implement real-time digital signal processing algorithms, which are characterised by high data throughput, small efficient computation sections and deterministic execution periods. The operation of the system is assumed to adhere to these properties.

If the transputer communications bandwidth matches or exceeds that required by the DSP sub-network, then the overall performance of the node is proportional to the number of DSPs (linear scalability). If the transputer is unable to maintain this bandwidth, however, then the DSPs will be forced to wait for data, thus reducing the performance of the node somewhat. The point at which this happens is termed the *latency threshold*, and marks the point at which the transputer/DSP communications mechanism reaches saturation.

This chapter is concerned with the performance of the data routing code implemented on the transputer, in order to give a measure of the attainable performance of the node, and to determine the latency threshold, for a given DSP configuration.

The topology of the DSP network is defined by the data routing software, enabling arbitrary topologies to be implemented. Each different topology requires a different code structure, which modifies the performance of the inter-processor communication mechanism. This chapter considers two topologies directly applicable to a wide variety of DSP applications — the orthogonal mapping and the pipeline. In the former, data is used exclusively by a single DSP, in the latter, the output of one DSP forms the input of the next. The approach used in this chapter is aimed specifically at the Hymips architecture, and an effort has been made to provide a deterministic measure of performance. Hence the analysis may not be as general as those offered in [32], [78], [87], [88], [89], [90], [91], [92], [93], but offers a more accurate description of the system.

The performance of the routing software is parameterised in terms of the number of DSPs in the network, the execution period of the code running on the

DSPs, the data transfer rate of the transputer, the data vector length and semaphore test and set overheads. The expressions produced allow the latency threshold to be determined for any given set of conditions, thus giving a limit to the linear scalability properties of the particular configuration.

The transputer communicates according to the principles of CSP, the provision of a microcoded scheduler ensuring that it is very efficient at doing so. However, the transputer is being forced to communicate with the DSPs through shared memory, using a semaphore protocol, which is a foreign environment and departs significantly from previous methods of using either semaphores or shared memory to provide communications. The main problem arising from this different approach occurs when a number of parallel processes are created and enqueued (as is the case whenever link communications are utilised). The effect of program structure and the conditions required for valid operation are outlined in this chapter.

An operational model of the routing code is presented in Section 2. Section 3 discusses communications software using only memory to memory transfers (intra-node case). Section 4 expands on the intra-node case by including external transfers in the analysis (inter-node case). An empirical verification of the analyses is presented in Section 5. Finally, Section 6 provides a summary.

9.2 An Operational Model

In addition to testing semaphores, transferring data and resetting semaphores, the data routing code is also required to initialise the shared memory areas and synchronise with the DSPs. This chapter, however, is concerned only with data transfer and not with initialisation. There are many ways in which this may be carried out. One option

would be to test a semaphore, then transfer the data or go on to test another semaphore, depending upon whether or not the semaphore was set. This type of protocol is adequate for general purpose systems [22], but produces a non-deterministic communication scheme which may produce communication latencies unacceptable in a real-time DSP application. This system has been designed to implement digital signal processing algorithms, which possess a fixed execution time. The DSPs thus require data, and set their semaphores, at periodic intervals. A deterministic semaphore protocol, such as the blocking protocol [22], is more suitable for such applications. Using this protocol, a processor repeatedly tests a semaphore until it is set, the processor is said to be "spin locked" [22].

It has been shown in Chapter 8 that a dual domain DPR partitioning scheme provides an efficient communication mechanism, and so is used in this model. Other conditions used in the development of the performance models are:

i The code sections running on the DSPs are identical.

ii The data vectors are of constant size, *w*.

iii The transputer initialises all shared data areas and performs synchronisation with all of the DSPs before transferring data.

iv The domains are pre-loaded with data.

9.3 Data Transfer Within the Node

This section presents a performance characterisation of both the orthogonal and pipeline configurations for the case of data transfer within the node. The structure of the code is inherently sequential, data transfer being achieved by memory to memory block moves. The minimum execution period that may be tolerated by the DSPs in

order to ensure that they do not experience communication latency is

$$t_{Dcx} > N_D(t_{Tsp} + t_{Trs} + 2t_{TX}) \quad (1)$$

for the orthogonal configuration, and

$$t_{Dcx} > N_D(t_{Tsp} + t_{Trs}) + t_{TX}(N_D - 1) \quad (2)$$

for the pipeline configuration, where

N_D The number of DSPs

T_{Dcx} The time required for a DSP to successfully test a semaphore, perform computation upon the domain and reset the semaphore.

T_{Tsp} The time required by the transputer to successfully test a semaphore.

T_{Trs} The time required by the transputer to reset a semaphore.

t_{TX} The time required by the transputer to transfer a domain's input and output vectors.

From these expressions, it may be seen that the data transfer bandwidth of the pipeline configuration is significantly higher than that of the orthogonal case.

These general parameters may be expressed in terms of vector length, *w*, and instruction cycles. The transputer, running at 25MHz, operates with a 40ns instruction cycle; the DSPs,running at 20.5MHz, operate with a 97.5ns instruction cycle. The transputer utilises external memory to memory block moves, and so the time required to set up and implement the transfer of a data vector of length *w* may be expressed as [30]

$$t_{TX} = (4w + 6)40ns$$

The semaphore test routine requires 10 cycles, and a semaphore may be reset in 6 cycles. Now, the DSP requires 6 cycles to successfully test a semaphore and 2 cycles to set a semaphore. Using these execution times, Equation (1) may be re-arranged as

$$N_D < \frac{97.5(8 + wN_c)}{40(28 + 8w)} \tag{3}$$

and Equation (2) as

$$N_D < \frac{97.5(8 + wN_c) + 40(6 + 4w)}{40(28 + 4w)} \tag{4}$$

which give the maximum number of DSPs which may be supported before they begin to experience communication latency.

A typical audio processing application will utilise about 200 DSP instruction cycles [94]. If a vector length of 256 is used, then from Equation (3) 59 DSPs may be supported in an orthogonal configuration, and from Equation (4) 120 DSPs in a pipeline configuration. As the performance of these processors is not affected by communications latency, this corresponds to a node performance of 590 and 1200 MIPs respectively.

9.4 Data Transfer Outside the Node

The transputer uses its links to transfer data off the node. In order to reduce communication latency, it is important that the links are allowed to operate concurrently with the cpu, which necessitates the use of parallel constructs within the routing code. As demonstrated in Chapter 4, the operation of parallel programs is not straightforward, and care must be taken with their design in order to ensure that performance does not suffer.

9.4.1 Orthogonal Data Transfer

In this configuration, the data input and output vectors for each DSP are transferred

over transputer links. The transputer possesses four bidirectional links, limiting the number of DSPs that may be serviced in any one communication cycle to four if bidirectional link mode is used, or two if unidirectional link mode is used. A program controlling three DSPs is shown in Fig 9.1. The flow diagram and scheduling chart for this code are presented in Appendix E.

It may be seen from Appendix E that this program consists of three parallel processes — one for each DSP — which themselves contain two nested parallel processes, used to initialise the link transfers. The processes run at high priority, which eliminates the requirement of process timeslicing and makes the operation of the code more deterministic whenever excessive semaphore spinning occurs. Note that the process to be executed first is declared last, due to the manner in which the processes are scheduled.

From the scheduling chart, it may be seen that the second communications process (Ra) of this process (R) is not executed until the first parallel process of the last process (Q) is executed. This may cause excessive communications latency, as a data transfer associated with the first DSP must wait until the last DSP has set its semaphore before it is allowed to proceed.

From the appendix, for a network of one DSP, the minimum execution period that may be tolerated by the DSPs in order to ensure that they do not experience communications latency is given by

$$177 + Lw + t_{Trs} + t_{Tsp} + X \tag{5}$$

and the overhead associated with each additional DSP given by

$$88 + t_{Trs} + t_{Tsp} + X \tag{6}$$

Hence, for a four DSP system,

$$t_{Dcx} > 451 + 4(t_{Tip} + t_{Trx} + X) + Lw \tag{7}$$

which, as at the limit $X = t_{Trx}$ may be re-arranged as

$$N_c > \frac{40(515 + Lw) - 780}{97.5w} \tag{8}$$

which gives a measure of the minimum DSP computation section execution period required to alleviate DSP communication delay. For a four DSP configuration, only the bidirectional link mode is available (L=85.1), as each link is mapped to one of the DSPs. From Equation (8), for a vector length of 256, the DSPs must run a computation code section of *at least* 35 instruction cycles per data word.

An alternative program is shown in Fig 9.2. The structure of this program allows the communication transfers associated with a particular DSP to be initialised, in turn, after the semaphore has been set. The communication latency of this program is lower than the previous program, but the behaviour is not so robust. It may be seen that the operations of testing and resetting a semaphore are spread across two parallel processes. In order to avoid data corruption using this structure, the process testing the semaphore must both begin its data transfer before its paired process begins its transfer, and end its transfer before its pair has finished its transfer. If these precedence constraints are not met, then either data will be transferred before the semaphore has been set — causing the transputer to overwrite the DPR — or the semaphore will be prematurely reset — causing the DSP to overwrite the DPR. These precedence requirements are upheld if each external transfer set up by the transputer is serviced immediately. Providing these conditions are met, then this structure offers

less latency between associated link transfers, reducing overheads and increasing performance. The flow diagram and scheduling chart for this program are presented in Appendix E. For a network of one DSP, the minimum execution period that may be tolerated by the DSPs in order to ensure that they do not experience communications latency is given by

$$t_{Dcx} > Lw + t_{Trs} + 166 + X \tag{9}$$

and the overhead associated with each additional DSP given by

$$56 + t_{Trs} + X \tag{10}$$

Hence, for a four DSP system,

$$t_{Dcx} > 334 + 4t_{Trs} + Lw + 4X \tag{11}$$

which may be re-arranged to give

$$N_c > \frac{40(Lw - 382)}{97.5w} \tag{12}$$

For a vector length of 256, the DSPs must run a computation code section of at least 32 cycles per data word if they are not to experience communications delays.

9.4.2 Pipeline Data transfer

The program for a three stage pipeline is shown in Fig 9.3. The input of the first DSP in the pipe is taken from a link, and the output of the last DSP is taken to a link. All other data transfers occur through internal Occam channels. In order to maximise the overlap of external and internal data transfer, the processes utilising the links are initialised first. The internal transfers will not be made until the final semaphore has

been set, but the delay incurred is outweighed by the advantage obtained through overlapping the external communications.

Processes using internal communication set up their channels sequentially, as there is no benefit in using parallel constructs. Channel communication is used in preference to direct block move operations as the communication synchronisation capability of soft channels ensures that data is not output by a process until the semaphore of the input process has been set. Any latency caused by the queuing of processes is outweighed by the relative efficiency of soft, compared to hard, transfers. The flow diagram and scheduling chart of this program are given in Appendix E.

From this analysis, for a network of two DSPs, the minimum execution period that may be tolerated by the DSPs in order to ensure that they do not experience communications latency is given by

$$t_{Dcx} > 169 + t_{Trs} + Lw + X \tag{13}$$

which may be re-arranged to give

$$N_c > \frac{40(Lw - 595)}{97.5w} \tag{14}$$

which is a restriction applied to only the first and last DSPs in the pipeline, as only they utilise external transfers. In fact, this configuration is capable of transferring data between a number of DSPs while the external communications are taking place. These DSPs add to the computational performance of the node whilst incurring no additional communication overheads. From the appendix, the maximum number of DSPs supported is given by

$$N_{max} < \frac{w(L-8)-141}{4w+80} \tag{15}$$

For a vector size of 256, and unidirectional links, eleven additional DSPs may be supported, providing a total node performance of 130 MIPS. For the bidirectional link case, up to 17 DSPs may be supported, providing a total node performance of 190 MIPS.

9.5 Empirical Testing

The Hymips system consists of a single node supporting two DSPs at the time of writing. Although this configuration does not allow an in-depth analysis of performance, the validity of the above performance equations may still be investigated. The purpose of the empirical testing is to determine the efficiency of the semaphore based shared memory communications method by verifying the theoretically derived performance equations. In particular, the testing reported in this section determines the amount of idle time (latency) experienced by the DSPs, comparing the actual values to the predicted ones. This latency is a gauge to the efficiency of the communications scheme.

9.5.1 The Test Code

Transputer code was written for all of the configurations outlined in this chapter — orthogonal and pipeline intra-node, orthogonal types 1 and 2, and pipeline, inter-node. An additional transputer was used as the data source and sink for the inter-node configurations.

The DSPs were loaded with code comprising a modified semaphore test loop and a simple computation section held in a nested DO loop. Attempting to measure

latency directly on the DSP, using timer registers, would have significantly interfered with the operation of the code, providing misleading results. The adopted method incremented the contents of an address register by one on every semaphore spin, requiring only an additional cycle. After a pre-determined number of cycles, the DSPs came out of their "semaphore test / compute" loops and stored the contents of the address register in the DPR. This value was then read by the transputer system, and output to the screen.

9.5.2 Results

The theoretical values of N_c at which the latency threshold occurs were obtained from equations (1), (2), (4), (8) and (12) for each configuration. These values were used as the base for the empirical tests. Vector lengths of 4 and 256, both transputer link modes and $N_D = 2$ were used for the empirical comparisons. Tables 9.1 to 9.4 summarises the theoretical threshold values, and the corresponding empirically obtained latencies.

9.6 Summary

This chapter has been concerned with the performance of the Hymips multiprocessor node. The performance is determined by the intra-node communications bandwidth, which in turn is determined by the rate at which the transputer is able to transfer data over its EMI.

An operational model has been developed and used to provide theoretical estimates of the performance of the node for two configurations — orthogonal and pipeline — for both the intra- and inter-node cases. Two variations of the code for the

inter-node orthogonal transfer have been presented. The second type offers a higher performance than the first, but requires that its external transfers are always serviced in turn, with no delay. Such deterministic communications requirements are characteristic of many DSP applications. However, if such conditions are not met, then the first type may be used, which offers lower performance but is more robust. Due to the ability of the transputer to overlap link communications and cpu operation, the inter-node pipeline configuration is able to support a number of "intermediate" DSPs with no additional overheads.

An important characteristic of any DSP sub-network configuration is its latency threshold, which denotes the point at which the transputer communications mechanism becomes saturated and provides an upper limit to the linear scalability of the node. The theoretical predictions of the latency threshold, determined using a similar technique to that outlined in Chapter 4, have been compared with empirically obtained results and found to closely agree.

	Theoretical N_c	Average No. Spins at Theoretical N_c
Orthogonal (Intra)	7	1
Pipeline (Intra)	2	2
Orthogonal 1 (Inter)	24	10
Orthogonal 2 (Inter)	24	8
Pipeline (Inter)	24	3

Table 9.1 Threshold Values for $L = 57$ $w = 256$ $N_D = 2$

Type	Theoretical N_c	Average No. Spins at Theoretical N_c
Orthogonal (Intra)	7	1
Pipeline (Intra)	2	2
Orthogonal 1 (Inter)	36	12
Orthogonal 2 (Inter)	36	9
Pipeline (Inter)	34	2

Table 9.2 Threshold Values for $L = 85$ $w = 256$ $N_D = 2$

Type	Theoretical N_c	Average No. Spins at Theoretical N_c
Orthogonal (Intra)	11	2
Pipeline (Intra)	5	3
Orthogonal 1 (Inter)	57	15
Orthogonal 2 (Inter)	48	12

Table 9.3 Threshold Values for $L = 57$ $w = 4$ $N_D = 2$

Type	Theoretical N_c	Average No. Spins at Theoretical N_c
Orthogonal (Intra)	11	2
Pipeline (Intra)	5	3
Orthogonal 1 (Inter)	69	16
Orthogonal 2 (Inter)	60	12

Table 9.4 Threshold Values for L = 85 w = 4 N_D = 2

```
PRI PAR
  PAR

    SEQ                              -- P
      ... spin on sem2a
      PAR
        in2 ? in2a
        out2 ! out2a
      sem2a := sem.set

    SEQ                              -- Q
      ... spin on sem3a
      PAR
        in3 ? in3a
        out3 ! out3a
      sem3a := sem.set

    SEQ                              -- R
      ... spin on sem1a
      PAR
        in1 ? in1a
        out1 ! out1a
      sem1a := sem.set

... Likewise for data set "b"
```

Fig 9.1 Orthogonal Control Program Type I

```
PRI PAR
  PAR

    SEQ                                 -- P
      out ! out1a
      sem1a := sem.set

    SEQ                                 -- Q
      ... spin on sem2a
      in2 ? in2a

    SEQ                                 -- R
      out ! out2a
      sem2a := sem.set

    SEQ                                 -- S
      ... spin on sem3a
      in3 ? in3a

    SEQ                                 -- T
      out ! out3a
      sem3a := sem.set

    SEQ                                 -- U
      ... spin on sem1a
      in1 ? in1a

...  Likewise for data set "b"
```

Fig 9.2 Orthogonal Control Program Type II

```
PRI PAR
  PAR

    SEQ                                -- P
      ... spin on sem3a
      PAR
        2.to.3 ? in3a
        out ! out3a
      sem3a := sem.set

    SEQ                                -- Q
      ... spin on sem2a
      PAR
        2.to.3 ! out2a
        1.to.2 ? in2a
      sem2a := sem.set

    SEQ                                -- R
      ... spin on sem1a
      PAR
        1.to.2 ! out1a
        in ! in1a
      sem1a := sem.set

... Likewise for data set "b"
```

Fig 9.3 Pipeline Control Program

Chapter 10

Conclusion

The rapid growth of silicon device technology over the past two decades has resulted in the production of increasingly powerful processors. This growth has allowed the development of many varieties of high performance computer systems, such as multiprocessors which offer increased performance by executing tasks in parallel. The variety of multiprocessor architectures is wide, ranging from small SIMD systems employing parallel processing on a single silicon die, to large MIMD systems comprising many thousands of autonomous inter-communicating microprocessors. The corresponding increase in computer power has broadened the application area of such systems, including CAD, mathematical modelling, image processing, database systems and real-time digital signal processing.

Digital signal processing applications tend to require high data throughput and the ability to perform efficiently a small set of arithmetic operations (primarily multiplication and addition). These requirements are especially acute if the application is to be implemented in real time, when strict timing constraints must be met. Early microprocessors, being general purpose, were not optimised for arithmetic throughput, and so had limited use within real time signal processing systems. The low

performance of general purpose microprocessors, together with the apparent advantages of using digital rather than analogue processing methods, resulted in the development of specialised high speed multiplier and associated support chips which were used in dedicated systems. Although these systems provided a much higher operating bandwidth, they required a large number of dedicated devices (requiring a large amount of board space and high power consumption) and were difficult to program.

The continuing advances in microprocessor design and fabrication allowed the development of the first programmable digital signal processors, in the early 1980s. These devices incorporated a hardware multiplier within the datapath. Other architectural characteristics included a double wordlength accumulator (at least), multiple memory areas (Harvard architecture) and a number of input / output registers. These devices were relatively straightforward to program, whilst offering a high performance. Subsequent generations of signal processors have enhanced or added to these characteristics — contemporary devices incorporate larger multiple memory areas, instruction caches, hardware floating point multipliers and additional peripherals on chip. Many of these devices may be programmed using optimised C compilers in addition to their native assembly languages, and run inside mature operating systems. Although these devices offer very high performance (33 MFLOPS is typical), the need for higher bandwidths, or increased overall processing power, is leading to the development of multiple signal processor systems.

Large arrays of transputers have been successfully used in such application areas as radar processing , as even though their computing power is no higher than any other general purpose processor, they may be easily interconnected to provide

large parallel systems. Some digital signal processors do offer limited multiprocessor support, but this generally amounts to the provision of a number of DMA control lines. Elaborate interprocessor communication mechanisms, such as those used in the latest general purpose multiprocessor systems, could be used, but these are expensive. A digital signal multiprocessor needs to offer an efficient interprocessor communications bandwidth, whilst incurring minimal additional hardware costs. The more constrained behaviour of digital signal processing algorithms, compared to their general purpose counterparts, allows more efficient, and less complex architectures to be developed.

This thesis has examined the performance of two different types of processor, the Inmos transputer and the Motorola DSP56001, when used to implement a typical signal processing application, a multiple channel digital filter. The resulting characteristics of these devices has been used in the design of a hybrid MIMD multiprocessor system that is optimised to implement DSP applications. A node of this hybrid multiprocessor (Hymips) has been constructed, and is currently running performance test software.

This chapter is divided into two parts. The first summarises the work carried out on the processors and the multiprocessor system, the second discusses possible continuing work.

10.1 The Transputer

The transputer represents an ideal building block with which to construct large multiprocessor systems. The devices in this family incorporate up to four bidirectional serial links, which are used to interconnect them. Although the data transfer rate is

quite high, the main advantage of link transfer over more conventional communications methods is that once they have been initialised, the transfers occur simultaneously with cpu operation. Although based around a von Neumann architecture, the native language of the transputer, Occam, is a parallel language. This language allows parallel constructs to be defined, and directly supports the asynchronous unbuffered message passing communications protocol used by the transputer. Parallel programs may be developed on a single transputer, then easily mapped on to a transputer network to provide increased performance. With the exception of external communications, all logical parallel processes running on a single transputer are executed "pseudo-concurrently". This is achieved through the use of a microcoded scheduler, which keeps track of which processes are awaiting a communications or timer input (*inactive* processes), which processes are able to run (*active* processes) and for how long the present process has been running. Two priority levels may be defined. High priority processes run in preference to low priority processes, and are generally used to instigate link transfers. Low priority processes are timesliced by the scheduler, in order to ensure that each process is allocated its fair share of cpu time. Performance optimisation techniques, using Occam, are well documented. The Occam compiler also supports assembly language inserts, which may be used for time critical sections of code.

Transputer networks have been successfully used to implement DSP applications. As the transputer has not been optimised for DSP operation, these systems gain their power from the high degree of parallelism which they exhibit. The suitability of smaller transputer systems to DSP applications has been investigated in this thesis. The application consisted of a three pole Butterworth bandpass filter,

implemented in a multi-channel configuration. The filter utilised shifting operations rather than multiplication operations in order to increase computational throughput.

In order to investigate the effects of parallelism, the filter was mapped onto one, two and three transputers. The two processor mapping constituted a simple pipeline structure, whereas the three processor mapping incorporated an additional feedback link. Unlike sequential languages, the use of a parallel language allows the same logical program to be implemented using a number of different program structures. Two such structures, or *harnesses*, were used to gain an insight into the performance implications of program structure. Both harnesses used high priority communications processes and a low priority sequential computation process. Harness type I used the decoupled construct recommended in the literature, whereas type II used internal channels to pass data between the communications and computation processes. These harnesses incurred different types of overheads, the effects of which were analysed from the results.

In order to provide maximum performance, the computation section was coded in assembly language. The computation section associated with each data channel was coded explicitly, and the data elements accessed directly. This approach was memory intensive but supplied the maximum performance.

Link communications proceed more efficiently if blocks of data, rather than a single item, are transferred, as initialisation overheads are reduced to negligible levels. The effect of transfer block size (vector length) upon performance was investigated. This has been shown to be a flexible approach, allowing a data channel to use either one vector element or a number of elements, and allowing multiple rate filters to be implemented.

A theoretical model of program behaviour was developed, in order to allow the overall performance to be investigated and the effects of overheads and vector lengths to be assessed. The theoretical performance predictions were compared with the empirical results.

The empirical performance of each mapping of each harness for a range of vector lengths was measured using a system comprising in-house transputer boards. As expected, performance increased with vector length in all cases. The two processor mappings exhibited higher performance than the single processor mappings (but not twice as high), whereas the three processor mappings exhibited some unexpected behaviour. The three processor mapping of harness type I provided similar performance to the two processor mapping, whereas that of harness type II provided the lowest performance of all. This was probably due to the low computation code size, resulting in the dominance of the communications overheads.

Harness type I required almost twice the amount of memory space than type II. The effects of external memory access were seen as the drop in performance of the single processor mappings at higher vector sizes. This was also seen in the two processor mapping of harness type I. All other mappings used internal memory exclusively within the given range of vector size.

The theoretical model performed well for the one and two processor mappings of both harnesses. The model estimated a slightly higher performance than that obtained for low vector sizes, since it assumed a vector size of at least sixteen. The estimated performance at high vector sizes was slightly higher than that obtained, since the model did not take into account the effects of operand prefixing and external memory usage. The model provided much higher performance estimates than those

obtained for the three processor mappings, however. This was probably caused by the assumption of the model that the processor running the largest section of computation code dictated the overall performance. The additional link between the second and third processors caused additional complexity which the model did not take into account.

Harness type II offers the best performance for vector lengths below 12, whereas type I provides the best performance up to those vector sizes at which external memory accessing causes performance degradation.

10.2 The DSP56001

In contrast to the transputer, the Motorola DSP56001 has been designed specifically to implement DSP algorithms. This was the first digital signal processor marketed by Motorola, incorporating a 24bit wordlength and operating at clock frequencies of 20.5, 27 and 40MHz.

The heart of the processor is the arithmetic unit (ALU) which contains a single cycle non-pipelined MAC unit, a number of 24bit input and 56bit output registers and assorted shifter units. The device contains three independent simultaneously accessible memory spaces on chip, which together with a versatile register based indirect addressing scheme allow the MAC unit to be invoked every instruction cycle.

The address generation unit (AGU) contains eight sets of 16bit register triplets, divided into two banks of four. Each bank possesses its own arithmetic unit. The register triplets consist of an address register together with associated offset and modifier registers. An address register is used to access an operand in one of the memory spaces, and may be pre- / post- incremented / decremented by one or the

its offset register. An address may also be generated by adding / subtracting the contents of the offset register to the address register. A modifier is used to define one of three addressing modes — linear, modulo and bit reversed. Linear mode is the usual addressing scheme used in all processors. Modulo addressing allows circular buffers to be implemented with zero overhead. The bit reversed mode allows FFT algorithms to be implemented, also with zero addressing overheads.

The device also incorporates a byte wide interface in addition to asynchronous and synchronous serial interfaces, which are treated as memory mapped peripherals. These may be used for connection to another processor, or an ADC/DAC system.

Multiprocessor support is limited. The serial port may be configured in a "network" mode, with 32 time slots, but the communications bandwidth is limited. Bus request / grant pins are also included, in order to support DMA or shared memory access.

In common with other programmable digital signal processors, the DSP56001 optimally implements canonic II form filter difference equations. The difference equations of the application filter were derived from the shift and add algorithms used by the transputer. The final analytic form consisted of a single order high pass section in series with a bandpass biquadratic section. However, coefficient quantisation problems required that the filter be implemented as a cascade of single pole sections, with an additional feedback path. Extension to the multichannel case was achieved using address offset and modifier registers.

As the DSP56001 is a sequential processor, and interrupts were suspended during filter kernel operation, performance analysis simply became a matter of counting instruction cycles. A Motorola ADS56 Development System was used to

implement the code, and its monitor used to determine the number of cycles required to do so. Understandably, the DSP56001 provided significantly higher performance than the transputer.

The nature of the frequency response of the application filter precluded the use of on-line methods to test it. Instead, data was stored on disk and processed off-line using a proprietary signal analysis package.

10.3 The Hybrid Multiprocessor

It may be concluded from the characterisation of the above processors that, in a signal processing environment, the transputer is more efficient at communicating data than it is at computation, whereas the DSP56001 possesses the inverse properties. Based on these observations, the design of a digital signal multiprocessor was proposed. The architecture of this hybrid multiprocessor (Hymips) consists of nodes, interconnected by transputer links. Each node contains a number of DSP56001s, connected to a transputer through areas of dual ported RAM (DPR). The transputer controls the flow of data both within the node (using memory to memory block moves) and outside the node (using its links). This interconnection scheme allows the DSPs to continue processing data with minimal interruptions caused by communications. The logical configuration of the DSPs is software defined, and may be dynamically modified. Special memory maps may be used to speed up data transfer, and DSP programs may be downloaded from the transputer on the fly.

However, implementation problems associated with the interprocessor communications mechanism were encountered. Interprocessor communication occurs through shared memory, using a semaphore based protocol in order to ensure data

validity. The transputer has been designed to communicate over its links, using a message passing paradigm, and does not support the "atomic" instructions required to implement most semaphore protocols safely. A modified test and set semaphore protocol has been developed, which may be safely implemented on devices such as the transputer. The execution time of the semaphore test routine was decreased using assembly language and an addressing scheme which mapped the semaphore locations into positive address space.

The Hymips system has been designed to offer scalability both in the number of nodes (the transputer plane) and in the number of DSP56001s supported per node (the DSP plane). The upper limit to the number of DSPs which may be supported by a node with zero communications latency (the scalability limit) occurs whenever the data bandwidth required by the DSPs reaches the maximum capability of the transputer. Beyond this limit, the DSPs will be forced into idle periods as they await data transfer. This has been used as a gauge to the maximum attainable performance of the node, since in real-time systems performance is limited by i/o bandwidth, not overall computational performance.

Using experience gained in theoretically analysing the transputer filter code, programs were written in Occam which controlled data routing both inside and outside the node. Two of the many possible DSP configurations were highlighted — the orthogonal and pipeline configurations.

The internal control programs were straightforward, implemented sequentially and utilising block moves. The external control programs were more complex, incorporating parallel processes to initialise link transfers. Two versions of the external orthogonal code were implemented, one being more robust than the other but

providing higher latencies.

These programs were analysed using the transputer performance model. Using the results of these analyses, expressions for the scalability limit of the node were derived for each configuration in terms of the number of DSPs, the vector length and the amount of DSP code executed per datum. These theoretical performance predictions show that the node is capable of sustaining a useful amount of DSPs, for a general set of given conditions, which results in high node performance. The external pipeline configuration, in particular, is able to sustain additional DSPs whilst incurring no additional overheads.

A node of the proposed Hymips system, incorporating two DSPs, has been constructed. Performance testing is limited with such a small number of processors, but the tests which have been carried out have aligned the theoretical performance figures with the empirical results.

10.4 Suggestions for Further Work

Further work lies mainly with Hymips, although an extension to the transputer performance model to include multiprocessor link transfer synchronisation would be useful.

Suggestions for further work on Hymips may be divided into two sections — development work carried out on the existing system, and extensions to the architecture.

Presently, if the DSPs are to be reprogrammed, then the ADS56 system (and host PC) must be connected to one of the in-house transputer boards (and host PC), via a DPR prototype board. The DSP object code is then transferred from the ADS56,

through DPR and the transputer and finally into a DOS file on the transputer host PC, which may then be accessed by Hymips. Quite obviously, this is a laborious and inconvenient process. A routine could be easily written to strip a DSP object code file of its header information, and to convert it to a form which would be directly readable by Hymips. Routines could then be assembled and converted on the same PC, greatly easing reprogramming.

The DSPs in the node may currently be accessed only through DPR. This is sufficient for high speed data transfer, but does leave the DSPs somewhat isolated, severely restricting debugging support. The additional communications ports could be used in a similar fashion to those on the development system board. This would allow system debugging and monitoring to be implemented,and will be an essential tool whenever the system is programmed with real application software. A straightforward method of interconnection would be to connect a PC serial port to the RS232 level converters on the DSP boards, and configuring the DSP serial ports in multidrop mode.

An extension to the above improvements would be to design an integrated development and application environment. This would be a major project, but would greatly ease application development on Hymips. Such a system might be based around a windowing type environment, allowing DSP and transputer code to be edited and compiled from the same screen. Graphical output windows could also be supported in addition to processor state windows.

The most obvious architectural extension to the present system would be to add more DSPs to the present node, and to produce more nodes. This would then allow the performance of the system to be assessed more fully, and allow a wide range of

applications to be implemented.

The architecture itself has been designed to be as open as possible. Devices such as ADC/DACs may be easily connected to the DSPs, the transputer or onto the backplane bus.

The backplane bus supports 32bit data wordlength, and so allows other processors, such as floating point DSPs, to be incorporated into the system. These components tend to be more expensive than their fixed point counterparts, however.

Increased node performance may be obtained by using a higher bandwidth controller processor. This could be achieved using an ASIC, but this would limit reprogrammability. Only one currently available high performance processor (the TMS320C40) supports high bandwidth interprocessor communication. The new generation transputer, the T9000, also offers higher bandwidth and an expanded instruction set which supports semaphores and allows the scheduler operation to be modified from software, but has not yet been released. These processor are very expensive, and their use would be dependent upon a cost / performance trade-off.

References

[1] Mitchell H J (Ed.), *32bit Microprocessors*. London: Collins, 1986.

[2] Freer J R, *Systems Design with Advanced Microprocessors*. London: Pitman, 1987.

[3] Dasgupta S, *Computer Architecture: A Modern Synthesis (Volume 1)*. NY: Wiley, 1989.

[4] Jouppi N P, "The Non-Uniform Distribution of Instruction Level and Machine Parallelism and its Effect on Performance", *IEEE Trans. Comp.*, vol. 38 No. 12, pp 1645-1658, Dec. 1989.

[5] Hwang and Briggs, *Computer Architecture and Parallel Processing*. USA: MacGraw - Hill, 1987.

[6] Lee E A, "Programmable DSP Architectures: Part 1", *ASSP Magazine*, Oct. 1988, pp4-19.

[7] Dasgupta S, *Computer Architecture: A Modern Synthesis (Volume 2)*. New York: Wiley, 1989.

[8] Patterson D A, "Reduced Instruction Set Computers", *Comms. ACM*, vol. 28 No. 1, pp 8-21, Jan. 1885.

[9] Wilson R, "Higher Speeds Push Embedded Systems to Multiprocessing", *Computer Design*, July 1989, pp72-83.

[10] Hwang K, "Multiprocessor Supercomputers for Scientific/Engineering Applications", *IEEE Computer*, June 1985, pp57-73.

[11] Flynn M J, "Very High Speed Computer Systems", *Proc. IEEE*, vol. 54 No. 12, pp 1901-1909, Dec. 1966.

[12] Duncan R, "A Survey of Parallel Computer Architectures", *IEEE Computer*, Feb. 1990, pp5-24.

[13] Patton P C, "Multiprocessors: Architectures and Applications", *IEEE Computer*, June 1985, pp29-42.

[14] Schindler M, "Multiprocessing Systems Embrace Both New and Conventional Architectures", *Electronic Design*, March 1984, pp97-130.

[15] Hockney R, *Introduction to Parallel Computers*, Tutorial Lecture Notes presented at Conpar '88, UMIST, Manchester, UK. Sept. 1988

[16] Hockney and Jesshope, *Parallel Computers 2: Architectures, Programming and Algorithms*. Bristol, UK: Adam Hilger, 1988.

[17] Kung H T, "Why Systolic Architectures?", *IEEE Computer*, Jan. 1982, pp37-46.

[18] Kung H T, "VLSI Array Processors", *IEEE ASSP Magazine*, July 1985, pp4-22.

[19] Stone H S, *High-Performance Computer Architecture (2nd ed.)*. Reading, Mass.: Addison Wesley, 1990.

[20] Jagadish N et al, "An Efficient Scheme for Interprocessor Communications using Dual-Ported RAMs", *IEEE Micro*, Oct. 1989, pp10-19.

[21] Anderson T E, Lazowska E D and Levy H M, "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors", *IEEE Trans. Comp.*, vol. 38 No. 12, pp 1631-1644, Dec. 1989.

[22] Graunke G and Thakkar S, "Synchronisation Algorithms for Shared Memory Multiprocessors", *IEEE Computer*, June 1990, pp60-69.

[23] Dubois M and Thakkar S, "Cache Architectures in Tightly Coupled Multiprocessors", *IEEE Computer*, June 1990, pp9-11.

[24] Stenström P, "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, June 1990, pp12-24.

[25] Chaiken D et al, "Directory-Based Cache Coherence in Large Scale Multiprocessors", *IEEE Computer*, June 1990, pp49-58.

[26] Thakkar S et al, "Scalable Shared-Memory Multiprocessor Architectures", *IEEE Computer*, June 1990, pp71-83.

[27] Seitz C L, "The Cosmic Cube", *Comms. ACM*, vol. 28 No. 1, pp 22-29, Jan. 1989.

[28] Zhiang X, "System Effects of Interprocessor Communications Latency in Multicomputers", *IEEE Micro*, April 1991, pp12-55.

[29] Inmos Ltd, *The Transputer Databook (2nd Ed)*. London: Prentice Hall International, 1989.

[30] Yassaie H and Bramley R, "Vectram", *Parallelogram International*, Sept. 1990, pp6-10.

[31] Gelenbe E, *Multiprocessor Performance*. UK: Wiley and Sons, 1989.

[32] Inmos Ltd, *Transputer Technical Notes: Lies, Damn Lies and Benchmarks*. London: Prentice Hall International, 1989.

[33] Conte T M, Hwu W W, "Benchmark Characterisation", *IEEE Computer*, Jan. 1991, pp48-56.

[34] Kaiser J F,*Design Methods for Sampled Data Filters*. Proc. First Allerton Conf. on Circuits and Systems, Nov. 1963, pp221-236.

[35] Cooley J W,Tuckey J W, "An Algorithm for the Machine Computation of Complex Fourier Series", *Math. Comp.*, vol. 19 No. 4, pp 297-301, April 1965.

[36] DeFatta D J, Lucas J G and Hodgkiss W S, *Digital Signal Processing: A Systems Design Approach*. New York: John Wiley and Sons Inc., 1988.

[37] Rabiner and Gold, *Theory and Application of Digital Signal Processing*. NJ: Prentice Hall, 1975.

[38] Terrel T, *Introduction to Digital Filters*. UK: Macmillan, 1988.

[39] Lee E A, "Programmable DSP Architectures II", *IEEE ASSP Magazine*, Jan. 1989, pp4-14.

[40] Frantz G A et al, "The Texas Instruments TMS320C25 Digital Signal Microcomputer", *IEEE Micro*, Dec. 1986, pp10-28.

[41] Kloker K L, "The Motorola DSP56000 Digital Signal Processor", *IEEE Micro*, Dec. 1986, pp29-48.

[42] Roesgen J P, "The ADSP-2100 DSP Microprocessor", *IEEE Micro*, Dec. 1986, pp49-69.

[43] Papamichalis P and Simar R, "The TMS320C30 Floating Point Digital Signal Processor", *IEEE Micro*, June 1988, pp13-29.

[44] Fuccio M L et al, "The DSP32C: AT&T's Second Generation Floating Point DSP", *IEEE Micro*, June 1988, pp30-48.

[45] Sohie G R L and Kloker K L, "A Digital Signal Processor with IEEE Floating Point Arithmetic", *IEEE Micro*, June 1988, pp49-67.

[46] LH9124 Digital Signal Processor, Advanced Product Brief, Sharp Corporation, 1991.

[47] Raja P V R and Ganesan S, "An SIMD Multiple DSP Microprocessor System for Image Processing", *Microprocessors and Microsystems*, vol. 15 No. 9, pp 493-503, Sept. 1991.

[48] Kingswood N et al, "Image Reconstruction using the Transputer", *Proc. IEE (E)*, vol. 133 No. 3, pp 139-144, May 1986.

[49] Beton R D, Turner S P, Upstill C, "Hybrid Architecture Paradigms in Radar ESM Data Processing Applications", *Microprocessors and Microsystems*, vol. 13 No. 3, pp 160-164, April 1989.

[50] Gass W A et al, "Multiple Digital Signal Processor Environment for Intelligent Signal Processing", *Proc. IEEE*, vol. 75 No. 9, pp 1246-1259, Sept. 1987.

[51] Hesson J H, Gallagher F A and Harrington D R, "A 32 bit Programmable Signal Processor for a MultiProcessor System Environment", *IEEE Trans. ASSP*, vol. ASSP-31 No. 4, pp 912-921, Aug. 1983.

[52] Bolch G et al, "MUPSI: A Multiprocessor for Signal Processing", *Proc. IEEE*, vol. 75 No. 9, pp 1211-1219, Sept. 1987.

[53] Santos J, Parera J and Veiga M, "A Hypercube Multiprocessor for Digital Signal Processing Algorithm Research", *Proc. ICASSP*, May 1988, pp1698-1701.

[54] Gaudiot J-L, "Data Driven Multicomputers in Digital Signal Processing", *Proc. IEEE*, vol. 75 No. 9, pp 1220-1234, Sept. 1987.

[55] Multinovic V, Fortes J A B and Jamieson L H, "A Multiprocessor Architecture for Real-Time Computation of a class of DFT Algorithms", *IEEE Trans. ASSP*, vol. ASSP-34 No. 5, pp 1301-1309, Oct. 1986.

[56] Sandler M B, "Interfacing the Transputer to the TMS320 in an Image Processing Environment", *Microprocessors and Microsystems*, vol. 12 No. 11, pp 490-496, Nov. 1988.

[57] Ching P C and Wu S W, "Real-Time Digital Signal Processing using a Parallel processor Architecture", *Microprocessors and Microsystems*, vol. 13 No. 10, pp 653-658, Oct. 1989.

[58] Zhon S, Sandler M B and Bergman G D, "A Switched Memory Decoding System for a Multiprocessor System", *Microprocessors and Microsystems*, vol. 15 No. 9, pp 493-503, Sept. 1991.

[59] Sandler M B, Hayat L and Casta L D F, "Benchmarking Processors for Image Processing", *Microprocessors and Microsystems*, vol. 14 No. 9, pp 583-588, Sept. 1990.

[60] Lang G R et al, "An Optimum Parallel Architecture for High Speed Real-Time DSP", *IEEE Computer*, Feb. 1988, pp47-58.

[61] Sung W, Mitra S K and Jeren B, "Multiprocessor Implementation of Digital Filtering Algorithms using a Parallel Block Processing Method", *IEEE Trans. Parallel and Distributed Computing*, vol. 3 No. 1, pp 110-120, Jan. 1992.

[62] Pountain R, *A Tutorial Introduction to OCCAM Programming*. UK: Inmos Ltd, 1987.

[63] Hoare C A R, "Communicating Sequential Processes", *Comms. ACM*, vol. 8 No. 21, pp 666-677, Aug. 1988.

[64] Inmos, *The Transputer Instruction Set - A Compiler Writer's Guide*. UK: Inmos Ltd, 1987.

[65] Atkins P, Performance Maximisation, Transputer Technical Note No. 17, Inmos Ltd, 1987.

[66] Anderson A J, "A Performance Evaluation of Microprocessors, DSPs and the Transputer for Recursive Parameter Estimation", *Microprocessors and Microsystems*, vol. 15 No. 3, pp 131-140, April 1991.

[67] Lee P J, Design of a Transputer Evaluation System, MSc Project Report, University of Durham, 1986.

[68] Inmos Ltd, *The Transputer Development System*. UK: Inmos Ltd, 1988.

[69] Motorola, *DSP56001 Users Manual*. USA: Motorola, 1989.

[70] Motorola, *The DSP56001 Technical Data Sheet*, USA: Motorola, 1988

[71] Motorola, *DSP56000 Assembler Manual*. USA: Motorola, 1988.

[72] Motorola, *The DSP56000 Simulator*. USA: Motorola, 1988.

[73] Motorola, *ADS56 User's Manual*. USA: Motorola, 1988.

[74] Eichen W, "NEC's PD77230 Digital Signal Processor", *IEEE Micro*, Dec. 1986, pp60-69.

[75] Lane J, Hillman G, *Implementing IIR / FIR Digital Filters with Motorola's DSP56001*. USA: Motorola, 1990.

[76] Chrysafis A, Lansdwne S, *Fractional and Integer Arithmetic using the DSP56000 Family of General Purpose Digital Signal Processors*. USA: Motorola, 1990.

[77] Bhuyun N L, Yang Q and Agrawal D P, "Performance of Multiprocessor Interconnection Networks", *IEEE Computer*, Feb. 1989, pp25-37.

[78] Cheriton D R and Goosen H A, "Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture", *IEEE Computer*, Feb. 1991, pp33-46.

[79] Vranesic Z G et al, "Hector: A Hierarchically Structured Shared-Memory Multiprocessor", *IEEE Computer*, Jan. 1991, pp72-79.

[80] Allan R and Purvis B, "Exercising the FX2800", *Parallelogram International*, April 1991, pp8-10.

[81] Sanders J, "Intel Scientific Wows Users with 7GFLOP i860 Based Hypercube", *Parallelogram International*, Jan. 1990, pp8-10.

[82] Hastings H, "Power Per Processor", *Parallelogram International*, Sept. 1989, pp10-11.

[83] Molesky L D et al, "Predictable Synchronisation Mechanisms for Multiprocessor Real-Time Systems", *The Journal of Real Time Systems*, vol. 2 No. 3, pp 163-180, Sept. 1990.

[84] Howeister D, "Semaphores at the Transputer Instruction Level", *Occam User Group Newsletter*, July 1990, pp46-50.

[85] De Pietro G and Vaccaro R, "Asynchronous Communication Primitives for Occam Programs", *Occam User Group Newsletter*, Jan. 1992, pp43-48.

[86] Boianov L K and Knowles A E, "Higher Speed Transputer Communication Using Shared Memory", *Microprocessors and Microsystems*, vol. 15 No. 2, pp 67-72, Feb. 1991.

[87] Gustafson J L, "Re-Evaluating Amdahl's Law", *Comms. ACM*, vol. 31 No. 5, pp 532-533, May 1988.

[88] Holliday M A and Vernon M K, "Exact Performance Estimates for Multi-Processor Memory and Bus Interference", *IEEE Trans. Comp.*, vol. C-36 No. 1, pp 76-85, Jan. 1987.

[89] Dubois M and Scheurich C, "Memory Access Dependencies in Shared-Memory Multiprocessors", *IEEE Trans. Software Engineering*, vol. 16 No. 6, pp 660-673, June 1990.

[90] Mahgoub I O and Elmagarmid A K, "Performance Analysis of a Generalised Class of m-Level Hierarchical Multiprocessor Systems", *IEEE Trans. Parallel and Distributed Systems*, vol. 3 No. 2, pp 129-138, Feb. 1992.

[91] Menasce D A and Barroso L A, "A Methodology for Performance Evaluation of Parallel Applications on Multiprocessors", *Journal of Parallel and Distributed Computing*, vol. 14 No. 1, pp 1-4, Jan. 1992.

[92] Dolter J W, Ramanathan P and Shin K G, "Performance Analysis of Virtual Cut-through Switching in HARTS: A Hexagonal Mesh

Multicomputer", *IEEE Trans. Comp.*, vol. 40 No. 6, pp 669-680, June 1991.

[93] Chiang M C and Sohi G S, "Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput Oriented Environment", *IEEE Trans. Comp.*, vol. 41 No. 3, pp 297-317, March 1992.

[94] Linton N L, Terepin S and Purvis A, "Parallel Digital Signal Processing for Audio Engineering", *88th Audio Engineering Society Convention*, Montreux, March 1990.

Appendix A

Filter Analysis

A.1 Introduction

This section deals with the analysis of the filter structures used by the transputer and DSP56001. The structures of the highpass and lowpass single pole sections are given in Fig A.1, together with the overall block structure.

The difference equations of the two basic filter types, and their associated transfer functions, are developed in section 2. These transfer functions are used to determine the difference equation for a lowpass/highpass cascade in section 3. The difference equation for the "modified" cascade, and its associated transfer function, is developed in section 4. As cascaded filter sections (either single pole or biquadratic) are usually used in digital filter implementations, it was not felt necessary to develop the characteristic equations of the filter any further. The overall filter, then, may be decomposed into either three single pole sections or a single pole highpass stage followed by a modified cascade stage. The location of the poles and zeroes of the various filter elements are determined in section 5.

A.2 The Single Pole Sections

A.2.1 The Lowpass Section

From Fig A.1a, it may be seen that

$$y(n)_{lp} = \frac{x(n-1)-y_{lp}(n-1)}{2^N} + y_{lp}(n-1) \tag{1}$$

re-arranging forms the difference equation,

$$y_{lp}(n) = 2^{-N}x(n-1) + (1-2^{-N})y_{lp}(n-1) \tag{2}$$

corresponding to a transfer function of

$$G_{lp}(z) = \frac{2^{-N}z^{-1}}{1-(1-2^{-N})z^{-1}} \tag{3}$$

which, in pole—zero form, becomes

$$G_{lp}(z) = \frac{2^{-N}}{z-(1-2^{-N})} \tag{4}$$

The frequency and phase response of this filter are shown in Fig A.4. The filter exhibits a first order Butterworth response (maximally flat, 20dB per decade cut off rate), with a low cut off frequency. The amplitude has not been normalised, and so it may be seen that the gain of this filter is never more than unity.

A.2.2 The Highpass Section

From Fig A.1b,

$$y_{hp}(n) = x(n) - u(n) \tag{5}$$

re-arranging,

$$u(n) = x(n) - y_{lp}(n) \tag{6}$$

From Fig A.1b and equation 2,

$$u(n) = 2^{-N}x(n-1) + (1-2^{-N})u(n) \tag{7}$$

Substituting 4 into 5,

$$x(n) - y_{lp}(n) = 2^{-N}x(n-1) + (1-2^{-N})(x(n-1) - y_{lp}(n-1)) \tag{8}$$

re-arranging,

$$y_{lp}(n) = x(n) - 2^{-N}x(n-1) - (1-2^{-N})(x(n-1) - y_{lp}(n-1) \tag{9}$$

and simplifying, to give the difference equation

$$y_{lp}(n) = x(n) - x(n-1) + (1-2^{-N})y_{lp}(n-1) \tag{10}$$

which corresponds to a transfer function given by

$$G_{lp}(z) = \frac{1 - z^{-1}}{1 - (1-2^{-N})z^{-1}} \tag{11}$$

which, in pole—zero form, becomes

$$G_{lp}(z) = \frac{z - 1}{z - (1-2^{-N})} \tag{12}$$

The frequency and phase response of this filter are shown in Fig A.5. As for the low pass section, this also filter exhibits a first order Butterworth response, at the same cut off frequency.

A.3 Cascaded Sections

The highpass/lowpass cascade is represented in Fig A.2a. Now,

$$G_c(z) = G_{lp}(z)G_{hp}(z) \tag{13}$$

Hence

$$G_c(z) = \frac{2^{-N}}{z-(1-2^{-N})}\frac{z-1}{z-(1-2^{-N})} \tag{14}$$

expanding,

$$G_c(z) = 2^{-N}\frac{z-1}{z^2-2(1-2^{-N})z+(1-2^{1-N}+2^{-2N})} \tag{15}$$

dividing by Z*Z,

$$G_c(z) = 2^{-N}\frac{z^{-1}-z^{-2}}{1-2(1-2^{-N})z^{-1}+(1-2^{1-N}+2^{-2N})z^{-2}} \tag{16}$$

which corresponds to a difference equation of

$$y_c(n) = 2^{-N}(x(n-1)-x(n-2))+2(1-2^{-N})-(1-2^{1-N}+2^{-2N})y(n-2) \tag{17}$$

The frequency and phase response of this compound filter are shown in Fig A.6.

A.4 Modified Cascade

The modified cascade structure is represented in Fig A.2b. It may be seen from this figure that the modification takes the form of a feedback path from the output directly into the input.

By inspection,

$$v(n) = x(n)+y_m(n) \tag{18}$$

and from 17,

$$y_m(n) = 2^{-N}(v(n-1)-v(n-2))+2(1-2^{-N})y_m(n-1)-(1-2^{1-N}+2^{2N})y_m(n-2) \tag{19}$$

substituting 18 into 19,

$$y_m(n) = 2^{-N}(x(n-1)+y_m(n-1)-x(n-2)-y_m(n-2)) \\ +2(1-2^{-N})y_m(n-1)-(1-2^{1-N}+2^{-2N})y_m(n-2) \quad (20)$$

simplifying, to give the difference equation,

$$y_m(n) = 2^{-N}(x(n-1)-x(n-2))+(2-2^{-N})y_m(n-1)-(1-2^{-N}+2^{-2N})y_m(n-2) \quad (21)$$

corresponding to a transfer function given by

$$G_m(z) = 2^{-N}\frac{z^{-1}-z^{-2}}{1-(2-2^{-N})z^{-1}+(1-2^{-N}+2^{-2N})z^{-2}} \quad (22)$$

which may also be written

$$G_m(z) = 2^{-N}\frac{z-1}{z^2-(2-2^{-N})z+(1-2^{-N}+2^{2N})} \quad (23)$$

The frequency and phase responses of this filter are shown in Fig A.7. The effect of the feedback is to sharpen the frequency response.

A.5 The Whole Filter

The whole filter may be thought of as being composed of a single pole highpass section in series with a biquadratic bandpass section, Fig A.3b, with a transfer function given by

$$G(z) = 2^{-N}\frac{(z-1)^2}{(z-\alpha)(z^2-(2-2^{-N})z+(1-2_{-N}+2^{-2N})} \quad (24)$$

The frequency and phase responses for the whole filter are given in Fig A.8. Note the slight change in gain and cut off frequency, and the second order high pass response, caused by the additional high pass section.

A.6 Location of Poles and Zeroes

A.6.1 Highpass and Lowpass Sections

From equation 4, it is apparent that the lowpass section does not possess a zero. It does, however, possess a pole which lies at

$$p_{lp} = 1 - 2^{-N} \tag{25}$$

As N=15, then, the pole, which is real, lies at

$$p_{lp} = 1 - 2^{-15} \tag{26}$$

From equation 12, it may be seen that the highpass section possesses a pole in the same position as the lowpass section, but that it also possesses a zero at z=1.

As the cascade section is composed of highpass and lowpass sections, equation 13, then it possesses a zero at z=1 and two poles, both of which occur at

$$p_{1c} = p_{2c} = 1 - 2^{-15} \tag{27}$$

The modified cascade section has a zero in the same place as the highpass/lowpass cascade. In order to determine the position of its poles, however, it is necessary to find the roots of the denominator of equation 23, ie

$$z^2 - (2 - 2^{-N})z + (1 - 2^{-N} + 2^{-2N}) = 0 \tag{28}$$

Using the quadratic formula,

$$z = \frac{(2 - 2^{-N}) \pm \sqrt{(2 - 2^{-N})^2 - 4(1 - 2^{-N} + 2^{-2N})}}{2} \tag{29}$$

Hence roots are given by

$$z = \frac{2^{N+1} - 1 \pm j\sqrt{3}}{2^{N+1}} \tag{30}$$

and so the poles of the filter lie at

$$p_{1m} = (1 - 2^{-(N+1)}) + j2^{-(N+1)}\sqrt{3} \tag{31}$$

and

$$p_{2m} = (1 - 2^{-(N+1)}) - j2^{-(N+1)}\sqrt{3} \tag{32}$$

Compared to the cascade, the poles of the modified cascade have a higher real component, and an imaginary component (albeit a small one). The poles form a conjugate pair, as they must since the coefficients of the filter are real.

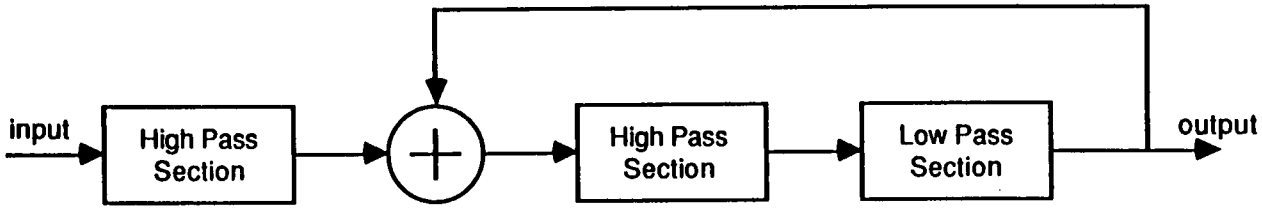


Fig A.1 a Overall Structure of the Filter

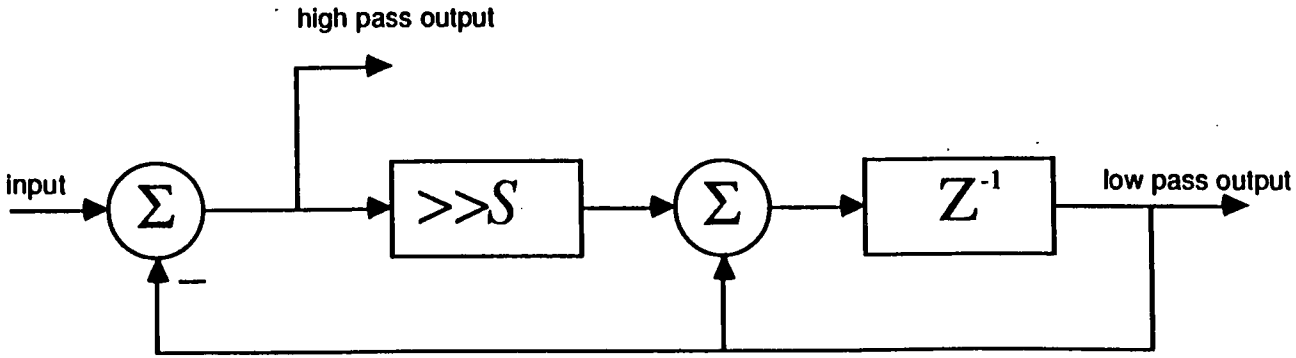


Fig A.1b & c Structure of the High Pass and Low Pass Sections

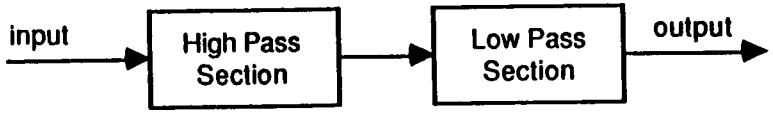


Fig A.2a Structure of the High Pass / Low Pass Cascade

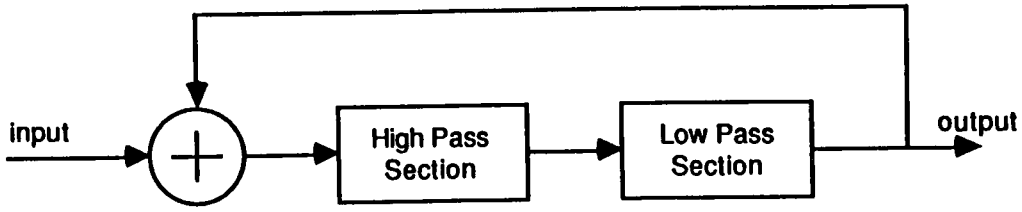


Fig A.2b Structure of the Modified High Pass / Low Pass Cascade

Fig A.3 Structure of the Whole Filter

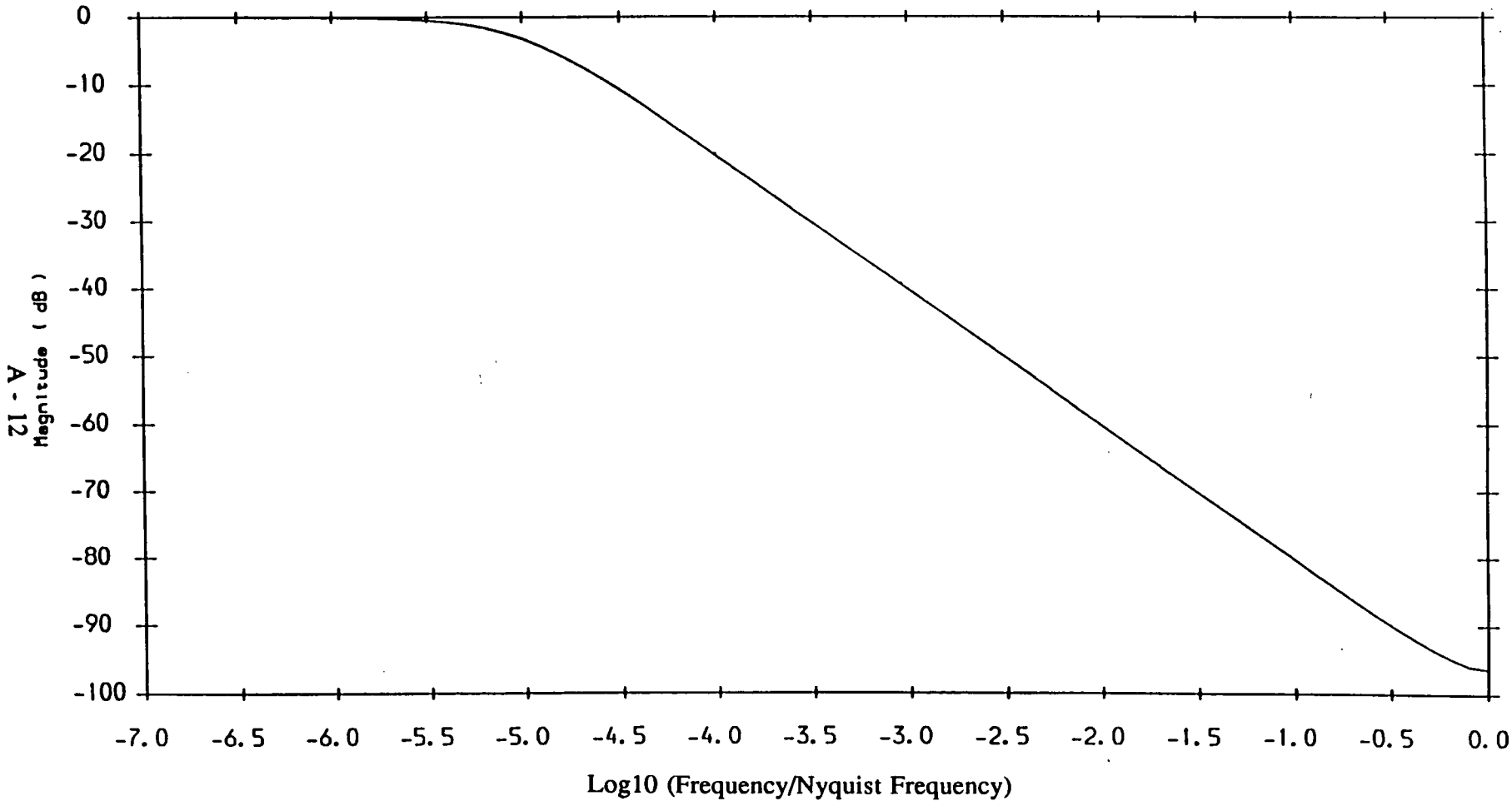


Fig A.4a Magnitude Response of the Low Pass Section

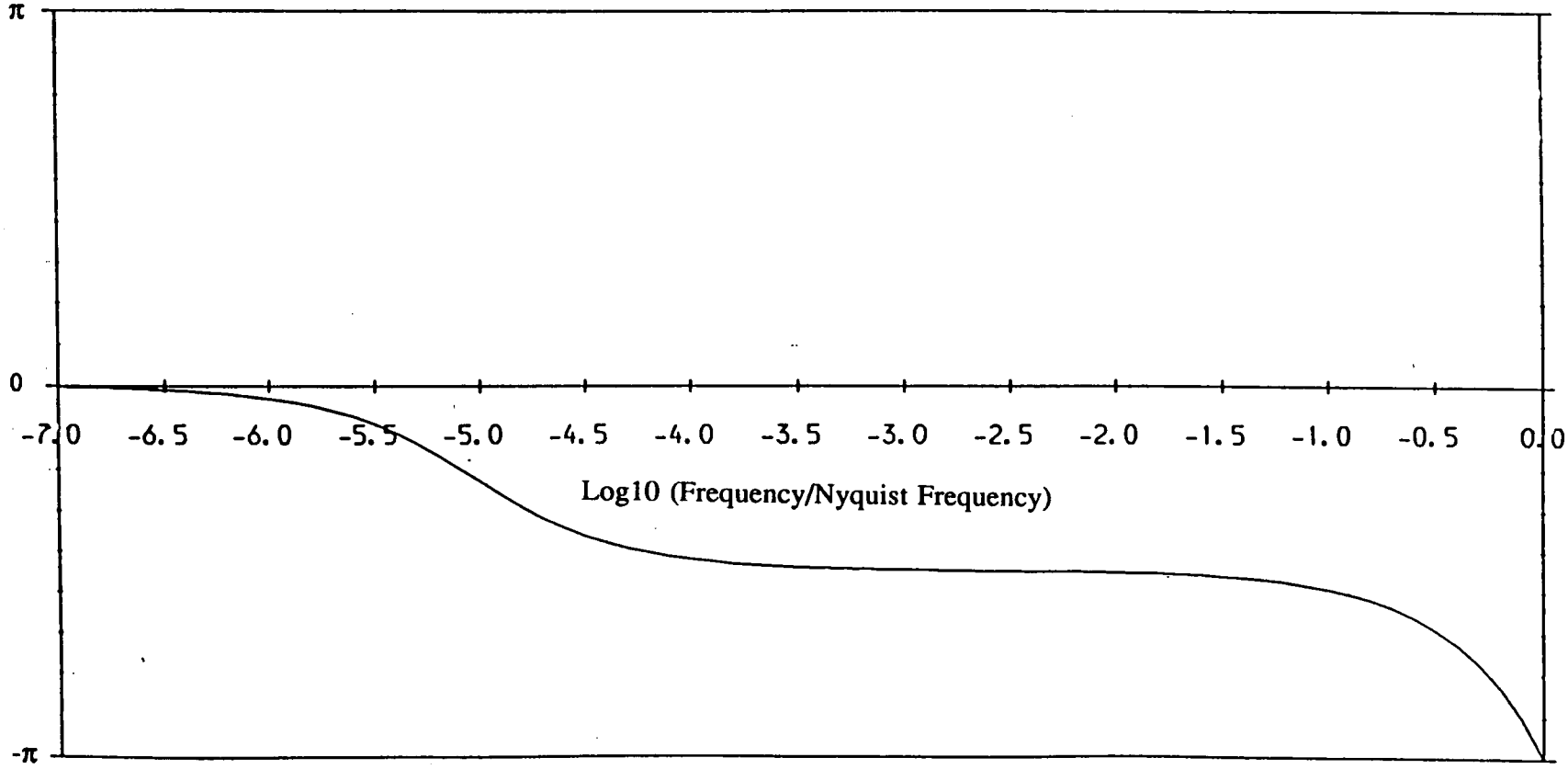


Fig A.4b Phase Response of the Low Pass Section

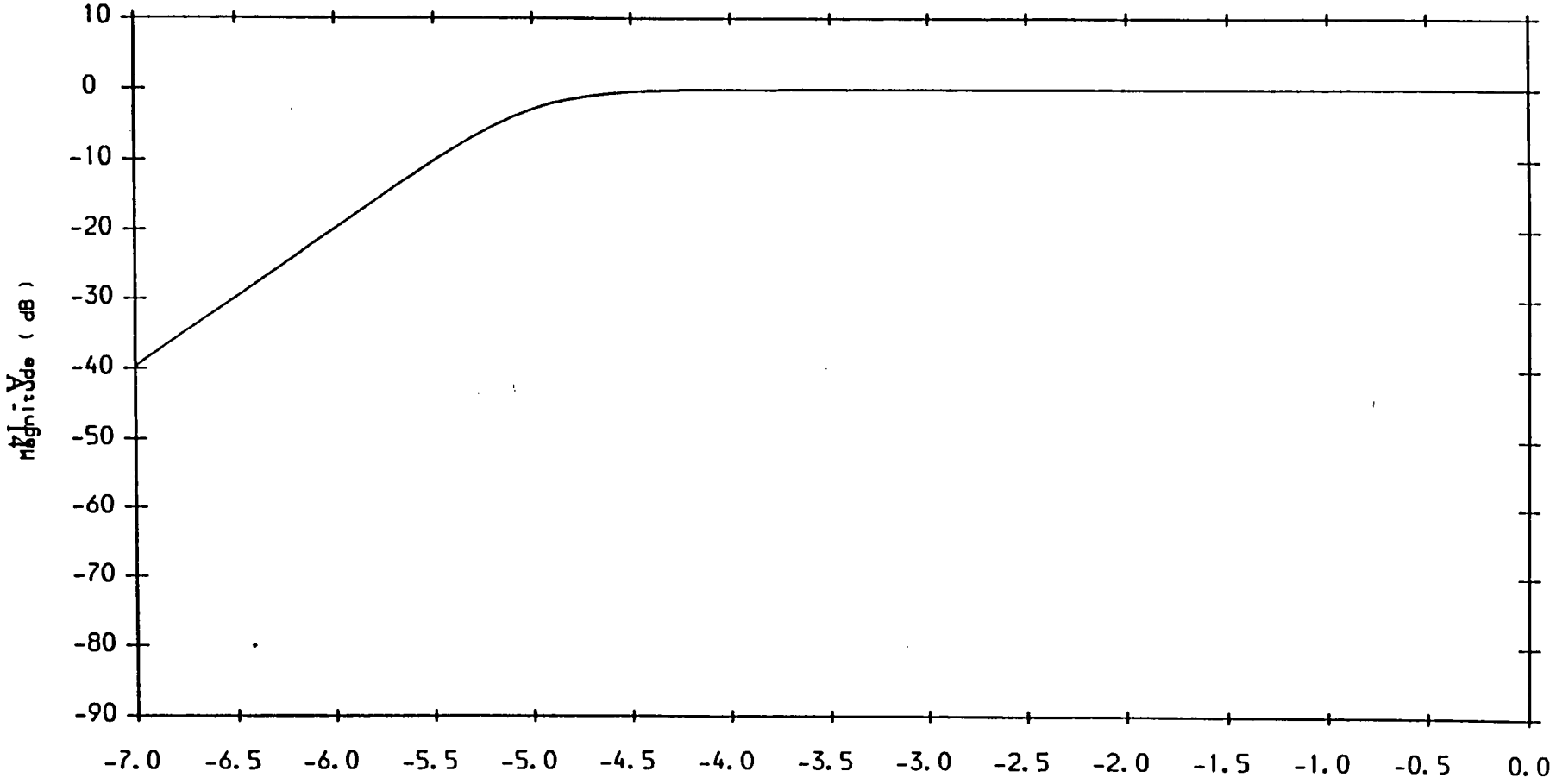


Fig A.5a Magnitude Response of the High Pass Section

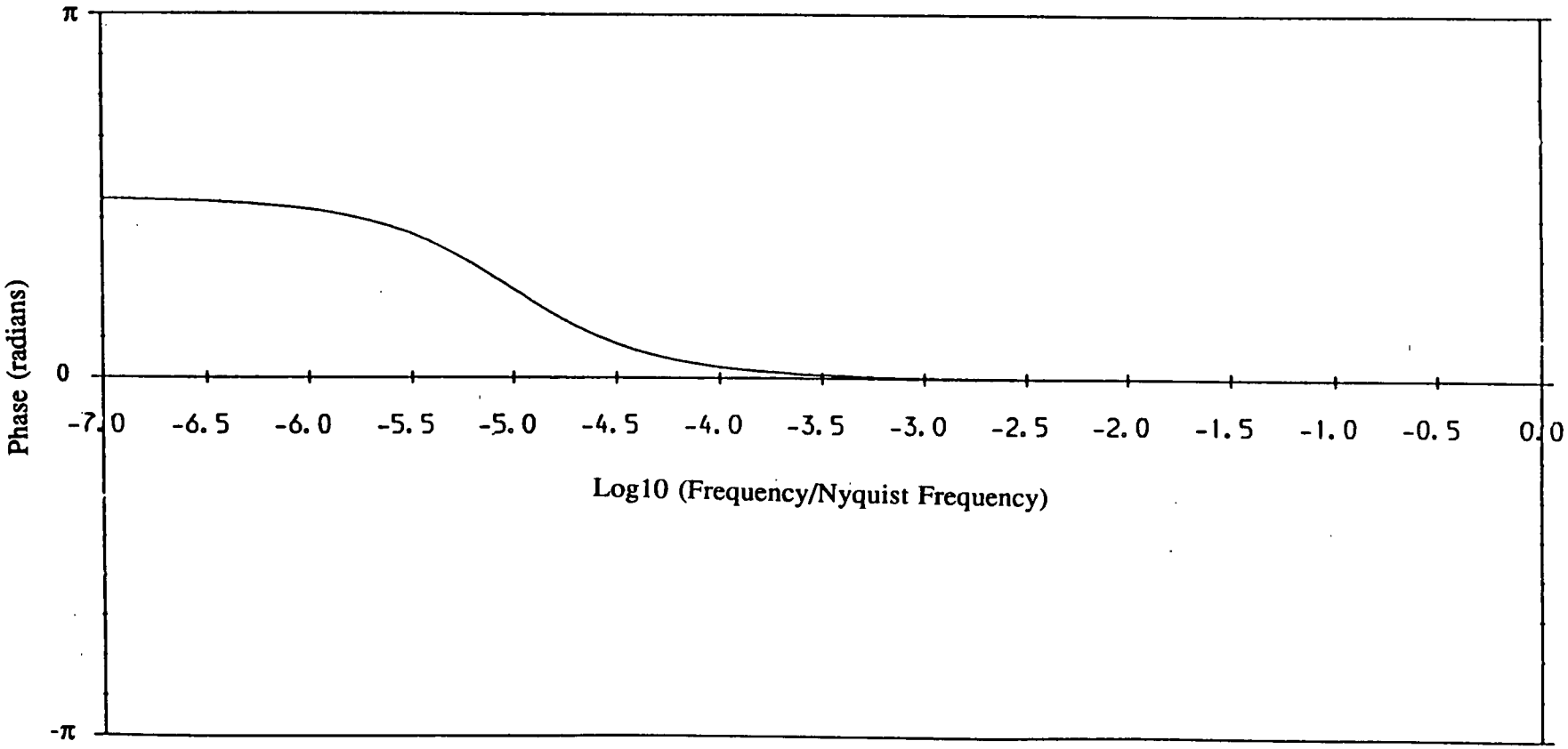


Fig A.5b Phase Response of the High Pass Section

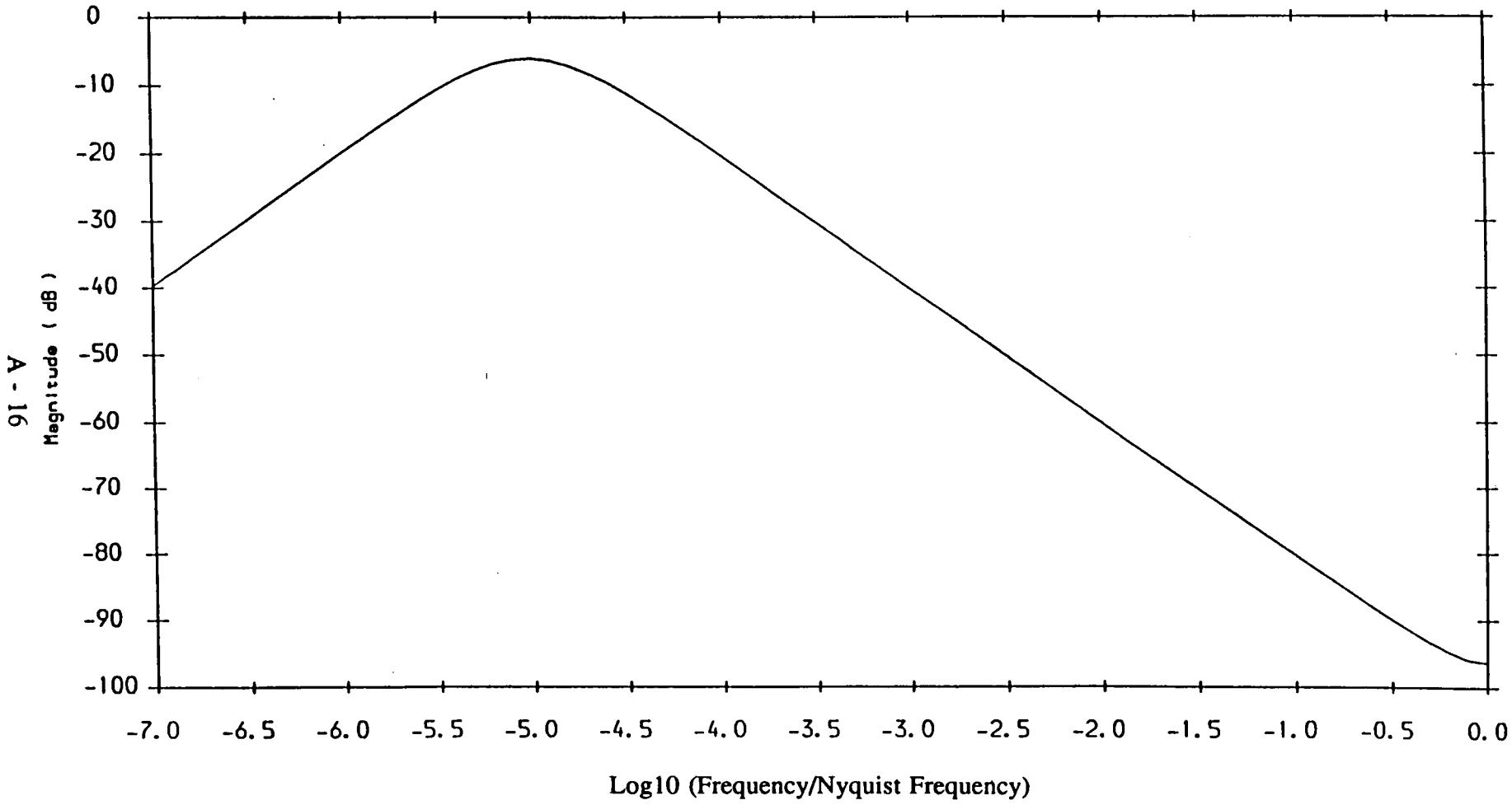


Fig A.6a Magnitude Response of the Cascade Section

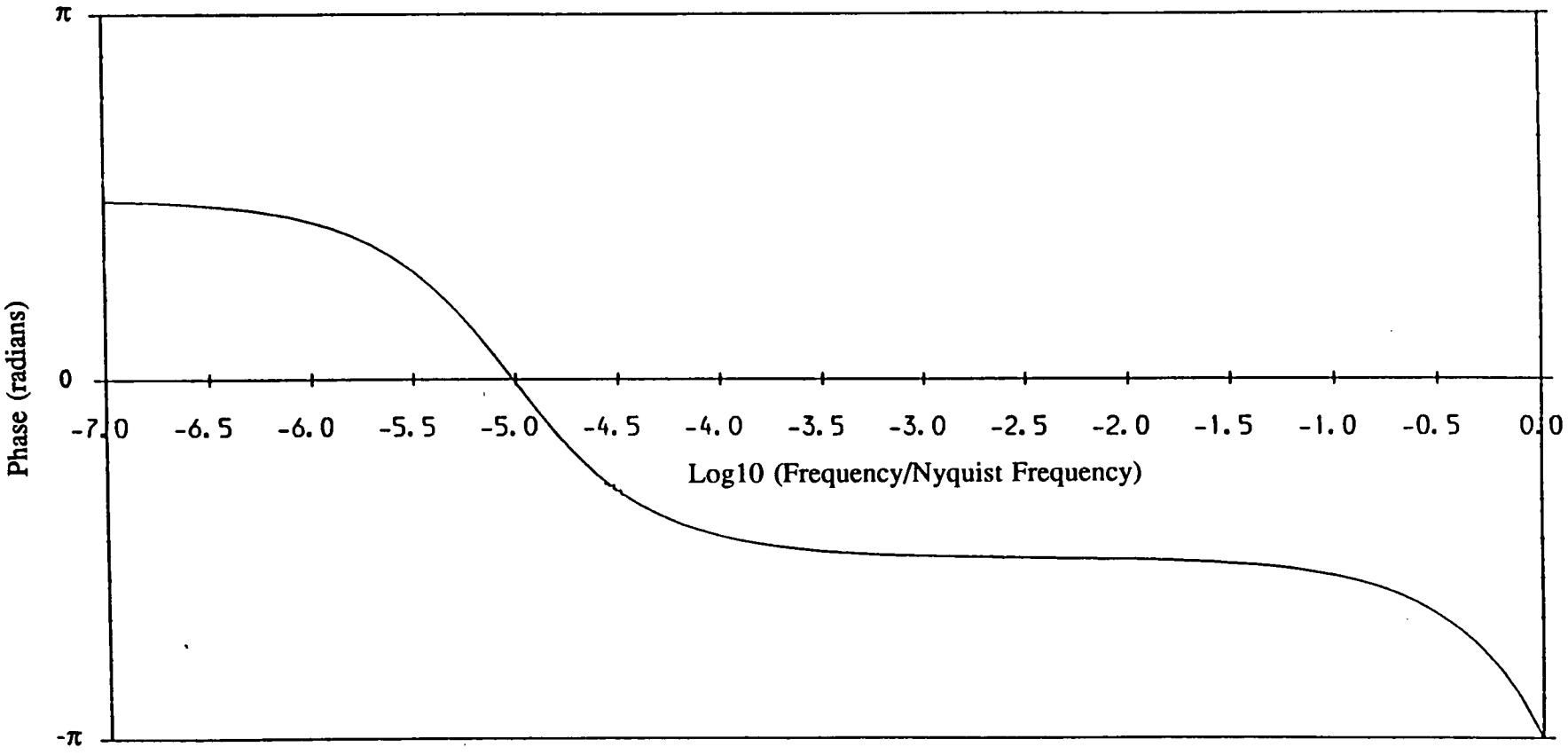


Fig A.6b Phase Response of the Cascade Section

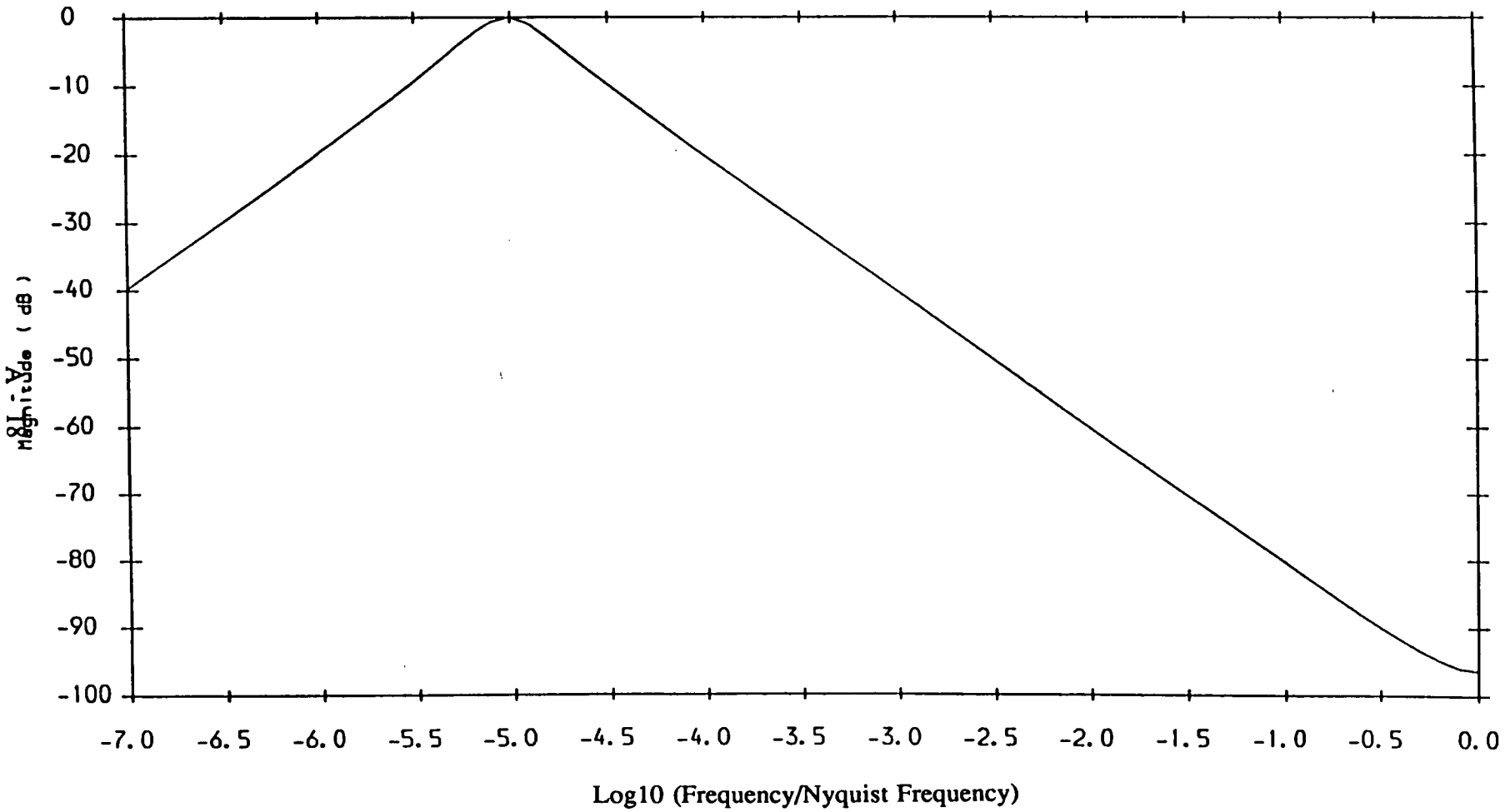


Fig A.7a Magnitude Response of the Modified Cascade Section

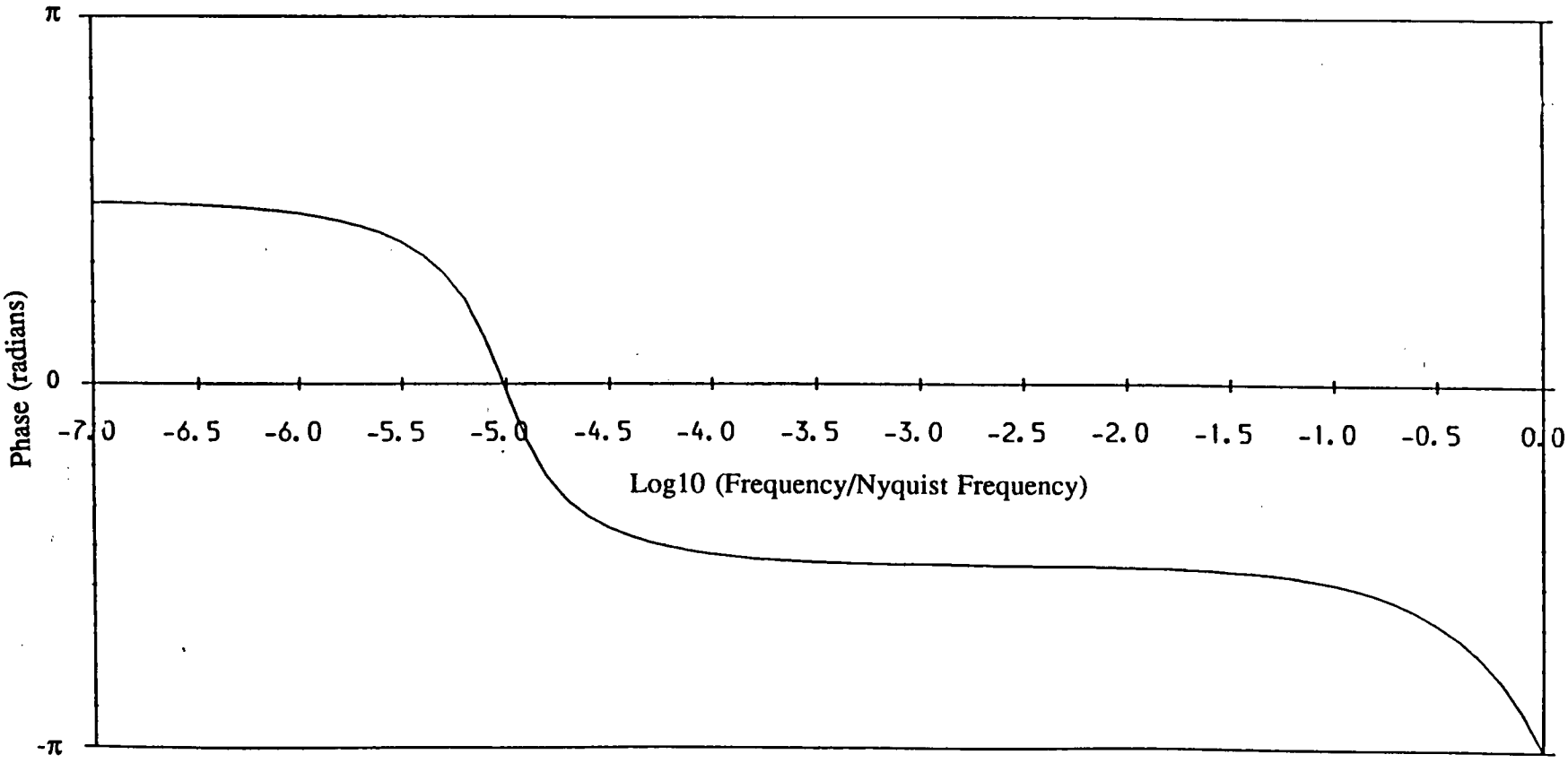


Fig A.7b Phase Response of the Modified Cascade Section

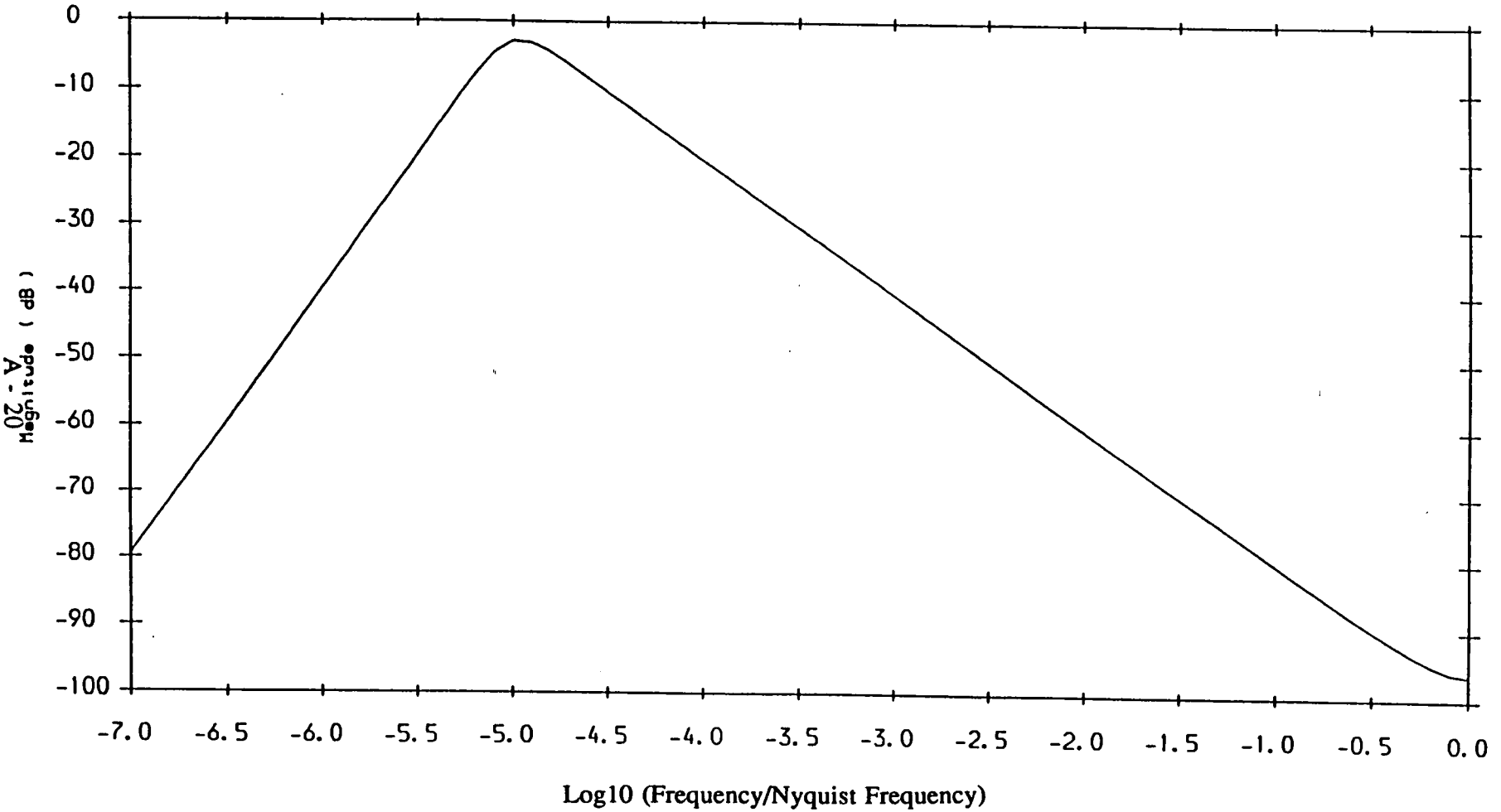


Fig A.8a Magnitude Response of the Whole Filter

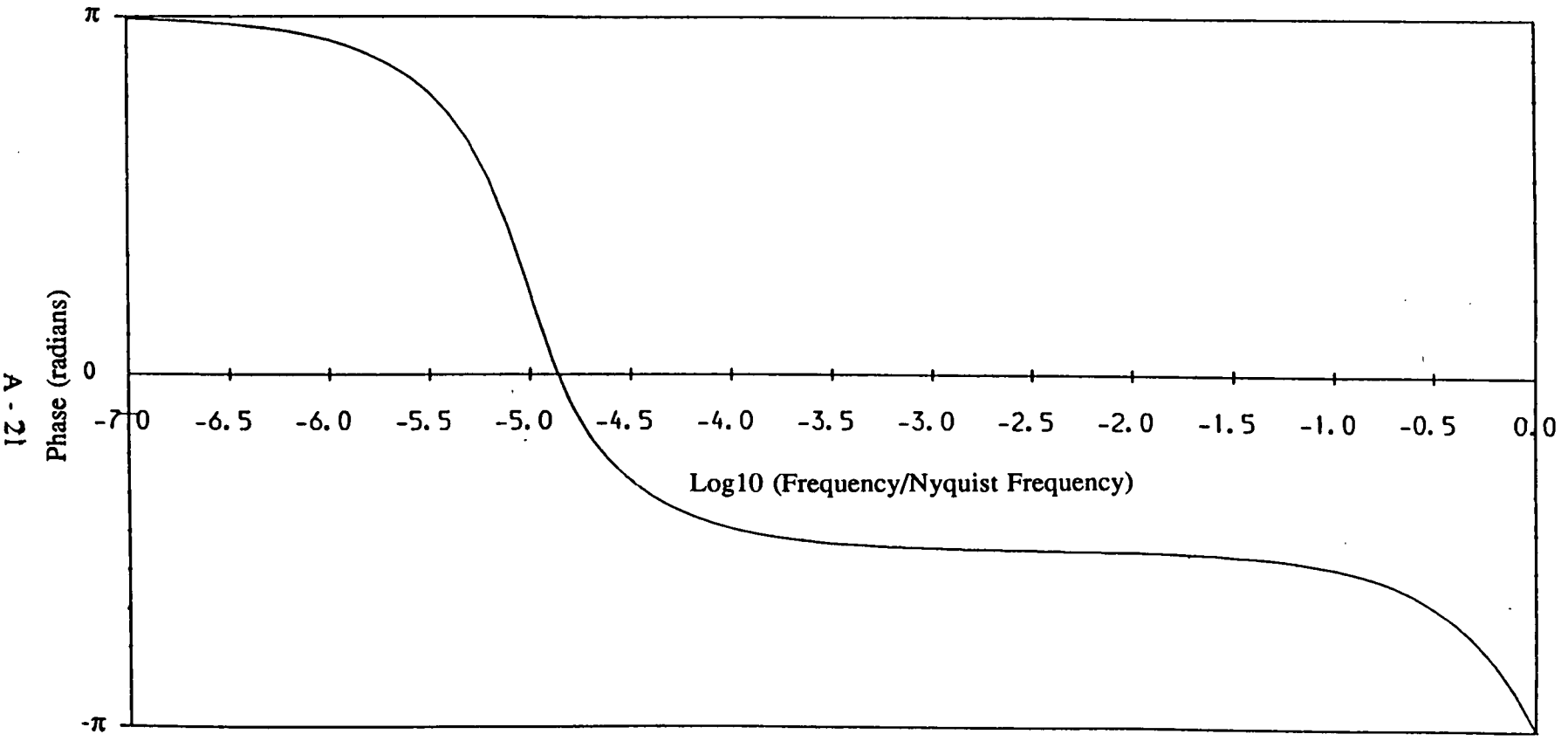


Fig A.8b Phase Response of the Whole Filter

Appendix B

Occam2 Filter Code

```
        PROC I.1(CHAN OF ANY input,output)
  VAL size IS 1 :
  {{{  declarations
  CHAN OF ANY input :
  PLACE input AT #05 :                -- link1 input
  INT i :
            [     s     i     z     e     ]     I     N     T
input.val1,output.val1,first.hpo1,second.hpo1,first.lpo
1,second.lpo1 :
            [     s     i     z     e     ]     I     N     T
input.val2,output.val2,first.hpo2,second.hpo2,first.lpo
2,second.lpo2 :
  }}}
  SEQ
    {{{  initialise variables
    SEQ i = 0 FOR size
      SEQ
        input.val1[i] := 0
        input.val2[i] := 0
        first.lpo1[i] := 0
        first.lpo2[i] := 0
        second.lpo1[i] := 0
        second.lpo2[i] := 0
        output.val1[i] := 0
        output.val2[i] := 0
    }}}
    WHILE TRUE
      SEQ
        PRI PAR
          {{{  comms 0
          PAR
            input ? input.val1
            output ! output.val1
          }}}
          {{{  comp 1
          SEQ
            {{{  [0] - [15]
            {{{  [0]
            LDL input.val2[0]
            LDL first.lpo2[0]
            SUB
            STL first.hpo2[0]
            LDL first.hpo2[0]
            XDBLE
            LDC #0F
            LSHR
            LDL first.lpo2[0]
            ADD
            STL first.lpo2[0]
            LDL first.hpo2[0]
            LDL output.val2[0]
            ADD
            LDL second.lpo2[0]
            SUB
            STL second.hpo2[0]
            LDL second.hpo2[0]
            XDBLE
            LDC #0F
            LSHR
            LDL second.lpo2[0]
            ADD
            STL second.lpo2[0]
            LDL second.hpo2[0]
            LDL output.val2[0]
            SUB
            XDBLE
            LDC #0F
            LSHR
            LDL output.val2[0]
            SUM
            STL output.val2[0]
            }}}
  }}}
PRI PAR
  {{{  comms 1
  PAR
    input ? input.val2
    output ! output.val2
  }}}
  {{{  comp 0
  SEQ
    {{{  comp0
```

```
    {{{  [0] - [15]
    {{{  [0]
    LDL input.val1[0]
    LDL first.lpo1[0]
    SUB
    STL first.hpo1[0]
    LDL first.hpo1[0]
    XDBLE
    LDC #0F
    LSHR
    LDL first.lpo1[0]
    ADD
    STL first.lpo1[0]
    LDL first.hpo1[0]
    LDL output.val1[0]
    ADD
    LDL second.lpo1[0]
    SUB
    STL second.hpo1[0]
    LDL second.hpo1[0]
    XDBLE
    LDC #0F
    LSHR
    LDL second.lpo1[0]
    ADD
    STL second.lpo1[0]
    LDL second.hpo1[0]
    LDL output.val1[0]
    SUB
    XDBLE
    LDC #0F
    LSHR
    LDL output.val1[0]
    SUM
    STL output.val1[0]
    }}}
  }}}
```



```
PROC II.1(CHAN OF ANY input,output)
  CHAN OF ANY comms.to.comp,comp.to.comms :
  VAL array.len IS 1 :
  PRI PAR
    {{{  comms
    PAR
      WHILE TRUE
        [array.len]INT output.val :
        SEQ
          comp.to.comms ? output.val
          output ! output.val
      WHILE TRUE
        [array.len]INT input.val :
        SEQ
          input ? input.val
          comms.to.comp ! input.val
    }}}
    [array.len]INT first.lpo,second.lpo,output.val :
    [array.len]INT input.val,first.hpo,second.hpo :
    SEQ
      SEQ i=0 FOR array.len
        SEQ
          output.val[i] := 0
          first.lpo[i] := 0
          second.lpo[i] := 0
      WHILE TRUE
        SEQ
          {{{  comp
          comms.to.comp ? input.val
          {{{  assembly comp [16] new
          {{{  computation
          LDL input.val[0]
          LDL first.lpo[0]
          SUB
          STL first.hpo[0]
          LDL first.hpo[0]
          XDBLE
          LDC #0F
          LSHR
          LDL first.lpo[0]
          ADD
```

```
STL first.lpo[0]
LDL first.hpo[0]
LDL output.val[0]
ADD
LDL second.lpo[0]
SUB
STL second.hpo[0]
LDL second.hpo[0]
XDBLE
LDC #0F
LSHR
LDL second.lpo[0]
ADD
STL second.lpo[0]
LDL second.hpo[0]
LDL output.val[0]
SUB
XDBLE
LDC #0F
LSHR
LDL output.val[0]
SUM
STL output.val[0]
}}}
}}}
comp.to.comms ! output.val
}}}
```



```
PROC II.2a(CHAN OF ANY input,inter1)
  VAL vector.size IS 1 :
  CHAN OF ANY comms.to.comp,comp.to.comms :
  PRI PAR
    PAR
      WHILE TRUE
        {{{  input buffer
        [vector.size]INT input.val :
        SEQ
          input ? input.val
          comms.to.comp ! input.val
        }}}
      WHILE TRUE
        {{{  output buffer
        [vector.size]INT output.val :
        SEQ
          comp.to.comms ? output.val
          inter1 ! output.val
        }}}
    [vector.size]INT lpo :
    SEQ
      SEQ i = 0 FOR vector.size
        SEQ
          lpo[i] := 0
      WHILE TRUE
        [vector.size]INT hpo,comp.input :
        SEQ
          comms.to.comp ? comp.input
          {{{  computation
          {{{  [0] - [15]
          {{{  [0]
          GUY
            LDL comp.input[0]
            LDL lpo[0]
            SUB
            STL hpo[0]
            LDL hpo[0]
            XDBLE
            LDC #0F
            LSHR
            LDL lpo[0]
```

```
      ADD
      STL lpo[0]
    }}}
    comp.to.comms ! hpo
:
```

```
PROC II.2b(CHAN OF ANY inter1,inter2)
  VAL vector.size IS 1 :
  CHAN OF ANY comms.to.comp1,comp.to.comms :
  PRI PAR
    PAR
      WHILE TRUE
        {{{  input buffer
        [vector.size]INT input.val :
        SEQ
          inter1 ? input.val                    -- inter1 ?
input.val
          comms.to.comp1 ! input.val            -- feedback
? feedback.val
        }}}
      WHILE TRUE
        {{{  output buffer
        [vector.size]INT output.val :
        SEQ
          comp.to.comms  ?  output.val                  --
comp.to.comms ? output.val
          inter2 ! output.val                   -- inter2 !
output.val
        }}}
    [vector.size]INT hpo,lpo,hlpo :
    SEQ
      SEQ i = 0 FOR vector.size
        SEQ
          hpo[i] := 0
          lpo[i] := 0
          hlpo[i] := 0
      WHILE TRUE
        [vector.size]INT comp.input :
        SEQ
          comms.to.comp1 ? comp.input
          {{{  computation
          {{{  [0] - [15]
          {{{  [0]
          GUY
            LDL comp.input[0]
            LDL hpo[0]
            ADD
```

```
    LDL lpo[0]
    SUB
    STL hpo[0]
    LDL hpo[0]
    XDBLE
    LDC #0F
    LSHR
    LDL lpo[0]
    ADD
    LDL hlpo[0]
    SUB
    STL hpo[0]
    LDL hpo[0]
    XDBLE
    LDC #0F
    LSHR
    LDL hlpo[0]
    ADD
    STL hlpo[0]
  }}}
  comp.to.comms ! hpo
```



```
PROC I.3b(CHAN OF ANY inter1,inter2,feedback)
  VAL vector.len IS 1 :
  [vector.len]INT lpo1,hpo1,x2,lpo2,fb1,fb2 :
  SEQ
    {{{  initialise values
    SEQ i = 0 FOR vector.len
      SEQ
        hpo1[i] := 0
        x2[i] := 0
        lpo1[i] := 0
        lpo2[i] := 0
        fb2[i] := 0
    }}}
    WHILE TRUE
      [vector.len]INT x1,hpo2 :
      INT internala,internalb :
      SEQ
        PRI PAR
          {{{  first pass
          PAR
            inter2 ! hpo1
            inter1 ? x1
            feedback ? fb1
          SEQ
            {{{  comp array2
            {{{  [0]
            GUY
              LDL x2[0]
              LDL fb2[0]
              ADD
              LDL lpo2[0]
              SUB
              STL hpo2[0]
              LDL hpo2[0]
              XDBLE
              LDC #0F
              LSHR
              LDL lpo2[0]
              ADD
              STL lpo2[0]
            }}}
```

```
    }}}
  }}}
PRI PAR
  {{{  second pass
  PAR
    inter2 ! hpo2
    inter1 ? x2
    feedback ? fb2
  SEQ
    {{{  comp array1
    {{{  [0]
    GUY
      LDL x1[0]
      LDL fb1[0]
      ADD
      LDL lpo1[0]
      SUB
      STL hpo1[0]
      LDL hpo1[0]
      XDBLE
      LDC #0F
      LSHR
      LDL lpo1[0]
      ADD
      STL lpo1[0]
   }}}
   }}}
 }}}
```



```
PROC I.3c(CHAN OF ANY inter2,feedback,lpout)
  VAL vector.len IS 1 :
  [vector.len]INT lpo1,hpo1,x2,lpo2,fb1,fb2 :
  SEQ
    {{{  initialise  values array
    SEQ i = 0 FOR vector.len
      SEQ
        hpo1[i] := 0
        x2[i] := 0
        lpo1[i] := 0
        lpo2[i] := 0
    }}}
    WHILE TRUE
      [vector.len]INT x1,hpo2 :
      SEQ
        PRI PAR
          {{{  first pass
          PAR
            inter2 ? x1
            lpout ! lpo1
            feedback ! lpo1
          SEQ
            {{{  comp array2
            {{{  [0]
            GUY
              LDL x2[0]
              LDL lpo2[0]
              SUB
              XDBLE
              LDC #0F
              LSHR
              LDL lpo2[0]
              ADD
              STL lpo2[0]
          }}}
        PRI PAR
          {{{  second pass
          PAR
            inter2 ? x2
            lpout ! lpo2
            feedback ! lpo2
```

```
SEQ
  {{{  comp array1
  {{{  [0]
  GUY
    LDL x1[0]
    LDL lpo1[0]
    SUB
    XDBLE
    LDC #0F
    LSHR
    LDL lpo1[0]
    ADD
    STL lpo1[0]
  }}}
  }}}
}}}
:
```



```
PROC II.2b(CHAN OF ANY inter1,inter2,feedback)
  CHAN OF ANY comms.to.comp1,comms.to.comp2,comp.to.comms
:
  VAL vector.len IS  1 :
  PRI PAR
    PAR
      WHILE TRUE
        {{{  input buffer
        [vector.len]INT input.val :
        SEQ
          inter1 ? input.val                    -- inter1 ?
input.val
          comms.to.comp1 ! input.val            -- feedback
? feedback.val
        }}}
      WHILE TRUE
        {{{  feedback input buffer
        [vector.len]INT feedback.val :
        SEQ
          feedback  ?  feedback.val                    --
comms.to.comp1 ! input.val
          comms.to.comp2   !   feedback.val           --
comms.to.comp2 ! feedback.val
        }}}
      WHILE TRUE
        {{{  output buffer
        [vector.len]INT output.val :
        SEQ
          comp.to.comms  ?  output.val                 --
comp.to.comms ? output.val
          inter2 ! output.val                   -- inter2 !
output.val
        }}}
    [vector.len]INT lpo :
    SEQ
      SEQ i = 0 FOR vector.len
        lpo[i] := 0
      WHILE TRUE
        [vector.len]INT hpo,comp.input,fb :
        INT lpoa :
        SEQ
```

```
comms.to.comp1 ? comp.input
comms.to.comp2 ? fb
{{{  assembly  computation
{{{  [0]
GUY
  LDL comp.input[0]
  LDL fb[0]
  ADD
  LDL lpo[0]
  SUB
  STL hpo[0]
  LDL hpo[0]
  XDBLE
  LDC #0F
  LSHR
  LDL lpo[0]
  ADD
  STL lpo[0]
}}}
}}}
comp.to.comms ! hpo
```



```
PROC II.3c(CHAN OF ANY inter2,lpout,feedback)
  CHAN OF ANY comms.to.comp,comp.to.comms,fb :
  VAL vector.len IS 1 :
  PRI PAR
    [vector.len]INT fb.setup :
    SEQ
      SEQ i = 0 FOR vector.len
        SEQ
          fb.setup[i] := 0
          feedback ! fb.setup
      PAR
        WHILE TRUE
          {{{  input buffer
          [vector.len]INT input.val :
          SEQ
            inter2 ? input.val
            comms.to.comp ! input.val
          }}}
        WHILE TRUE
          {{{  output buffer
          [vector.len]INT output.val :
          SEQ
            comp.to.comms ? output.val
            lpout ! output.val
          }}}
        WHILE TRUE
          {{{  feedback buffer
          [vector.len]INT fb.val :
          SEQ
            fb ? fb.val
            feedback ! fb.val
          }}}
      [vector.len]INT lpo :
      SEQ
        SEQ i = 0 FOR vector.len
          lpo[i] := 0
        WHILE TRUE
          [vector.len]INT comp.input :
          INT lpoa :
          SEQ
            comms.to.comp ? comp.input
```

```
{{{  assembly computation
{{{   [0]
{{{   [0]
GUY
  LDL comp.input[0]
  LDL lpo[0]
  SUB
  XDBLE
  LDC #0F
  LSHR
  LDL lpo[0]
  ADD
  STL lpo[0]
}}}
}}}
comp.to.comms ! lpo
fb ! lpo
```

```
PROC inputter(CHAN OF ANY to.filter)
  VAL vector.size IS 1 :
  CHAN OF ANY from.host :
  PLACE from.host AT #05 :    --link1 input
  {{{  declarations
  INT len,error,val,char,any :
  }}}
  SEQ
    {{{
    WHILE TRUE
      [vector.size]INT output.val :
      SEQ
        SEQ i = 0 FOR vector.size
          output.val[i] := 1
        {{{  100 outputs
        {{{  100
        {{{  10
        to.filter ! output.val
        }}}
        }}}
        }}}
    }}}
:
```

```
PROC watch(CHAN OF ANY in)
  VAL array.len IS 1 :
  INT i :
  WHILE TRUE
    SEQ
      SEQ i = 0 FOR 20
        PRI PAR
          {{{  DECS
          [20]INT start.time,end.time :
          CHAN OF ANY tohost,fromhost :
          TIMER clock :
          [array.len]INT any :
          PLACE tohost AT #02 :
          PLACE fromhost AT #06 :
          }}}
          SEQ
            clock ? start.time[i]
            {{{  100 inputs
            {{{  100 inputs
            {{{
            in ? any
            }}}
            }}}
            }}}
            clock ? end.time[i]
            tohost ! start.time[i] / array.len
            tohost ! end.time[i] / array.len
          SKIP
:
```

Appendix C

Occam2 Filter Program Scheduling Charts and Results Table

SCHEDULING CHART FOR THE TWO PROCESSOR MAPPING OF HARNESS TYPE I										
Note ref.	Processor Cycles	Label	PROCESS STATUS							
			SCHEDULING						COMMUNICATION	
			Execute	Active		Inactive			Q	R
				P0	P1	P0	P1		in	out
1	12	1	P'							
2	2	2	P'							
3	6	3	P'							
4	4	4	P'							
5	73w	5	P'							
6	6/11	6	P'							
7	2	7	P'							
8	6	8	P'							
9	6	9	P'							
10	7	10	P'							
11	13	11	P	OQ						
12	18		OQ		P					
13	2	15	OQ		P					
14	6	16	OQ		P					

SCHEDULING CHART FOR THE TWO PROCESSOR MAPPING OF HARNESS TYPE I										
Note ref.	Processor Cycles	Label	PROCESS STATUS							
			SCHEDULING						COMMUNICATION	
			Execute	Active		Inactive			Q	R
				P0	P1	P0	P1		in	out
15	3	17	OQ		P					
16	12	18	OQ	R	P					
17	6	19	Q	R	P					
18	18	20	R		P	Q			xfer	
19	6	22	R		P	Q			"	
20	18	23	P			Q,R			"	xfer
21	2	12	P			Q,R			"	"
22	46w-26	13	P	Q		R			fin	"
23	22		Q		P	R				"
24	16	21			P	R				"
25	18		P			R				"
26	4	13	P	R						fin
27	18		R		P					
28	16	24			P					

SCHEDULING CHART FOR THE TWO PROCESSOR MAPPING OF HARNESS TYPE I										
Note ref.	Processor Cycles	Label	PROCESS STATUS							
			SCHEDULING						COMMUNICATION	
			Execute	Active		Inactive			Q	R
				P0	P1	P0	P1		in	out
29	25	25	DP		P					
30	16	26	DP		P					
31	18		P							
32	29w+30	13	P							
33	16	14	P							

Two Processor Mapping of Harness Type I	
Note ref.	Comments
1	OP begins by jumping to the first instruction.
2	OP claims its workspace.
3	The location of the 'jump' block (at the head of the program) is stored in workspace location 15.
4	The vector initialisation loop is set up.
5	One iteration of the initialisation loop is performed.
6	OP executes a LEND. If the loop has completed, then execution will continue. If not, then execution returns to the start of the loop. The total number of cycles taken to perform the loop (excluding initialisation) is 82+(w-1)87.
7	OP begins to set up the PRI PAR by storing the number of parallel processes in workspace location 1...
8	...and storing the instruction pointer to the successor process (the next PRI PAR) in workspace location 0.
9	The current priority is checked to make sure that it is low.
10	OP stores the instruction pointer of the child process in (what will be) the new workspace location -1.
11	OP defines the process descriptor of the child process (which implicitly defines its priority - high) and places this process on the high priority queue. OP is de-activated.
12	P is interrupted in deference to the high priority process OQ.
13	OQ begins by setting the number of parallel processes it will produce.
14	OQ stores the instruction p[ointer of its successor process.
15	OQ sets up a child parallel process...
16	... and initialises it at the current priority level. The new process, R, is placed on the high priority queue.
17	Q continues by setting up a communications transfer.
18	Q executes an external 'in' and so is descheduled. R is executed in preference to P.
19	R begins by setting up a communications transfer.
20	R executes an external 'out' and so is descheduled. As P is the only remaining active process, it is re-executed.
21	P continues by claiming workspace for the computation section.
22	P enters its computation section. However, after a further 46w-26 cycles, Q completes its external transfer and so is rescheduled.

Two Processor Mapping of Harness Type I	
Note ref.	Comments
23	P is interrupted in deference to Q. The particular instruction on which P is interrupted varies bioth with w and with time, hence the average instruction length of 4 cycles is used.
24	Q continues by pointing to its parent (OQ) and ending itself. Hence Q is taken off the queue.
25	R has still not completed, and so P is re-executed.
26	P continues with its computation section. However, R completes its external transfer during the context switch, and so P is allowed to execute just one instruction...
27	...before being interrupted.
28	R continues by pointing to its parent (OQ) and ending itself. Hence R is taken off the queue. Both the child processes of OQ have now completed, and so OQ is free to invoke its successor process.
29	The de-prioritising code is invoked by OQ...
30	... and then, having pointed to its parent (OP), ends itself.
31	As P is the only remaining process, it is rescheduled.
32	P completes its computation section.
33	P points to its parent (OP) and ends itself. As its child processes have completed, OP is free to invoke its successor process, the next PRI PAR structure.

SCHEDULING CHART FOR THE TWO PROCESSOR MAPPING OF HARNESS TYPE II													
Note ref.	Processor Cycles	Label	PROCESS STATUS										
			SCHEDULING					COMMUNICATION					
			Execute	Active		Inactive		P		Q		R	
				P0	P1	P0	P1	in	out	in	out^E	in^E	out
1	32	1	P'										
2	11	2		Q'	P'								
3	8	14	Q'		P'								
4	4	15	Q'		P'								
5	12	16	Q'	R	P'								
6	7	17	Q	R	P'								
7	19	18	R		P'	Q				H			
8	6	24	R		P'	Q				"			
9	19	25	P'			Q,R				"		xfer	
10	7	3	P			Q,R				"		"	
11	41(w-1)+36	4	P			Q,R				"		"	
		5	P			Q,R				"		"	
12	6	6	P			Q,R				"		"	

SCHEDULING CHART FOR THE TWO PROCESSOR MAPPING OF HARNESS TYPE II													
Note ref.	Processor Cycles	Label	PROCESS STATUS										
			SCHEDULING					COMMUNICATION					
			Execute	Active		Inactive		P		Q		R	
				P0	P1	P0	P1	in	out	in	out[E]	in[E]	out
13	19	7				Q,R	P	H		"		"	
14	5w - 8 5w + 31			R		Q	P	"		"		F	
15	5	26	R			Q	P	"		"			
16	2w + 19	27	R		P	Q		F		"			C
17	4	28	R		P	Q				"			
18	6	24	R		P	Q				"			
19	19	25	P			Q,R				"		xfer	
20	46w-19	8	P	R		Q				"		F	
21	23	8	R		P	Q				"			
22	5	26	R		P	Q				"			
23	19	27	P			Q,R				"			H
24	29w+19	8	P			Q,R				"			"
25	6	9	P			Q,R				"			"
26	2w+19	10	P	Q		R			C	F			"

SCHEDULING CHART FOR THE TWO PROCESSOR MAPPING OF HARNESS TYPE II														
Note ref.	Processor Cycles	Label	PROCESS STATUS											
			SCHEDULING					COMMUNICATION						
			Execute	Active		Inactive		P		Q		R		
				P0	P1	P0	P1	in	out	in	out^E	in^E	out	
27	18		Q		P	R							"	
28	6	19	Q		P	R							"	
29	19	20	P			R,Q					xfer		"	
30	4	11	P			R,Q					"		"	
31	6	6	P			R,Q					"		"	
32	2w+19	7	P	R		Q		C			"		F	
33	18		R		P	Q					"			
34	4	28	R		P	Q					"			
35	6	24	R		P	Q					"			
36	19	25	P			Q,R					"	xfer		
37	44w-76	8	P	Q		R					fin	"		
38	23		Q		P	R						"		
39	4	21	Q		P	R						"		
40	5	17	Q		P	R						"		
41	19	18	P			R,Q				II		"		

SCHEDULING CHART FOR THE TWO PROCESSOR MAPPING OF HARNESS TYPE II													
Note ref.	Processor Cycles	Label	PROCESS STATUS										
			SCHEDULING					COMMUNICATION					
			Execute	Active		Inactive		P		Q		R	
				P0	P1	P0	P1	in	out	in	out^E	in^E	out
42	2w+25	8	P	R		Q				"		fin	
43	23		R		P	Q				"			
44	5	26	R		P	Q				"			
45	19	27	P			Q,R				"			H
46	32w+51	8	P			Q,R				"			"
47	6	9	P			Q,R				"			"
48	2w+19	10	P	Q		R			C	F			"
49	18		Q		P	R							"
50	5	19	Q		P	R							"
51	19	20	P			R,Q					xfer		"
52	4	11	P			R,Q					"		"
53	6	6	P			R,Q					"		"
54	2w + 19	7	P	R		Q		C			"		F
55	18		R		P	Q					"		
56	4	28	R		P	Q					"		

SCHEDULING CHART FOR THE TWO PROCESSOR MAPPING OF HARNESS TYPE II													
Note ref.	Processor Cycles	Label	PROCESS STATUS										
			SCHEDULING					COMMUNICATION					
			Execute	Active		Inactive		P		Q		R	
				P0	P1	P0	P1	in	out	in	out^E	in^E	out
57	6	24	R		P	Q					"		
58	19	25	P			R,Q					"	xfer	
59	44w-76	8	P	Q		R					fin	"	
60	23	8	Q		P	R						"	
61	4	21	Q		P	R						"	
62	5	17	Q		P	R						"	
63	19	18	P			R,Q				H		"	
64	2w+25	8	P	R		Q				"		fin	
65	23		R		P	Q				"			
66	5	26	R		P	Q				"			
67	19	27	P			Q,R				"			H
68	32w+51	8	P			Q,R				"			"
69	6	9	P			Q,R				"			"
70													

Two Processor Mapping of Harness Type II	
Note ref.	Comments
1	P' claims its workspace, initialises the internal channels, checks the priority, points to its successor process and sets up the high priority process.
2	P' executes a RUNP, activating the high priority process, Q'. P' is de-activated in preference to Q'.
3	Q' defines the number of 'child' processes, points to its successor (the de-prioritising code).
4	Q' sets up its child process, R.
5	Q' executes a STARTP, which places R on the active high priority queue.
6	Q' enters Q by adjusting the workspace pointer and setting up a communications transfer.
7	Q executes an internal 'in'. However, the channel is empty and so Q is descheduled. R is executed in preference to P'.
8	R begins by setting up a communications transfer.
9	R executes an external 'in' and so is descheduled. P' is the only remaining active process and so is re-executed.
10	P' enters P by adjusting the workspace pointer and initialising a control block for a replicated SEQ structure.
11	P enters a replicated SEQ loop. The time taken to execute this loop is 41(w-1) + 36 cycles. The next high priority process to become active is R, after 46w -7/+32 cycles from the beginning of the replicated SEQ. Hence, the loop completes before the transfer finishes and so P is not forced on to the queue by R.
12	P sets up a communications transfer.
13	P executes an internal 'in'. However, the channel is empty and so P is descheduled. There are no currently active processes.
14	There is now a delay until R, the only process not awaiting internal channel rescheduling, completes its external transfer. The delay is 5w - 8 (min), 5w + 31 (max).
15	R continues by setting up a communications transfer.
16	R executes an internal 'out', corresponding to the 'in' of P. Hence the transfer takes place and P is rescheduled.
17	R jumps to the top of its WHILE TRUE loop.
18	R sets up a communications transfer.
19	R executes an external 'out' and so is descheduled. P is the only remaining active process and so is re-executed.
20	P continues by entering its computation section. During this period, R completes its transfer after a further 46w-19 cycles and so is rescheduled.

Two Processor Mapping of Harness Type II	
Note ref.	Comments
21	This rescheduling causes P to be interrupted during the computation section. [the instruction that this rescheduling occurs on is dependent upon w. It is possible to calculate the instruction, but use the average instruction length of the computation section here, abs(3.54)=4]. Thus, the interrupt latency is 22 cycles.
22	R continues by setting up a communications transfer.
23	R executes an internal 'out'. The channel is empty and so R is descheduled. P is the only remaining active process and so is re-executed.
24	P continues by completing its computation loop. The high priority processes are currently awaiting soft channel communications, and so there is no further interruption of the computation section.
25	P sets up a communications transfer.
26	P executes an internal 'out', corresponding to the 'in' of Q. Hence the transfer takes place and Q is rescheduled.
27	P is interrupted in deference to Q.
28	Q continues by setting up a communication transfer.
29	Q executes an external 'out' and so is descheduled. P is the only remaining active process and so is re-executed.
30	P continues by jumping to the top of its "WHILE TRUE" loop.
31	P sets up a communications transfer.
32	P executes an internal 'in', corresponding to the 'out' of R. Hence the transfer takes place and R is rescheduled.
33	P is interrupted in deference to R.
34	R continues by jumping to the top of its "WHILE TRUE" loop.
35	R sets up a communications transfer.
36	R executes an external 'in' and so is descheduled. P is the only remaing active process and so is re-executed.
37	P continues by entering its computation section. After a further 46w-[2w+76] cycles, Q completes its external transfer, and so is rescheduled.
38	P is interrupted in deference to Q.
39	Q continues by jumping to the top of its "WHILE TRUE" loop.
40	Q sets up a communications transfer.
41	Q executes an internal 'in'. The channel is empty and so Q is descheduled. P is the only remaining active process and so is re-executed. [not enough cycles yet for R to have completed for largish w]

Two Processor Mapping of Harness Type II	
Note ref.	Comments
42	P continues its computation section. However, after a further 2w+25 cycles{46w-[44w-76+51]}, R completes its external transfer and so is rescheduled. Hence w>=11.
43	P is interrupted in deference to R.
44	R continues by setting up a communications transfer.
45	R executes an internal 'out'. The channel is empty and so R is descheduled. P is the only remaining active process and so is re-executed.
46	P continues by completing its computation section.{78w-[44w+2w-76+25]}
47	P sets up a communications transfer.
48	P executes an internal 'out', corresponding to the 'in' of Q. The transfer takes place and Q is rescheduled.
49	P is interrupted in deference to Q.
50	Q continues by setting up a communications transfer.
51	Q executes an external 'out' and so is descheduled. P is the only remaining active process and so is re-executed.
52	P continues by jumping to the top of its "WHILE TRUE" loop.
53	P sets up a communications transfer.
54	P executes an internal 'in', corresponding to the 'out' of R. The transfer takes place and R is rescheduled.
55	P is interrupted in deference to R.
56	R continues by jumping to the top of its "WHILE TRUE" loop.
57	R sets up a communications transfer.
58	R executes an external 'in' and so is descheduled. P is the only remaining active process and so is re-executed.
59	P enters its computation loop. After a further 46w-[2w+76] cycles, Q completes its external transfer and so is rescheduled.
60	P is interrupted in deference to Q.
61	Q continues by jumping to the top of its "WHILE TRUE" loop.
62	Q sets up a communications transfer.
63	Q executes an internal 'in'. The channel is empty and so Q is descheduled. P is the only remaining active process and so is re-executed.
64	P continues its computation section. After a further 2w+25 cycles, {46w-[44w-76+23+4+5+19]}, R completes its external communication and so is rescheduled.
65	P is interrupted in deference to R.

Two Processor Mapping of Harness Type II	
Note ref.	Comments
66	R continues by setting up a communications transfer.
67	R executes an internal 'out'. The channel is empty and so R is descheduled. P is the only remaining active process and so is re-executed.
68	P continues by completing its computation section.
69	P sets up a communications transfer

SCHEDULING CHART FOR THREE PROCESSOR MAPPING OF HARNESS TYPE I											
Note ref.	Processor Cycles	Label	PROCESS STATUS								
			SCHEDULING								
			Execute	Active		Inactive		P	Q	R	
				P0	P1	P0	P1	out	in	in	
1	6	1	OM								
2	43w	2	OM								
3	6/10	3	OM								
4	22	4	OM								
5	13	5	HM		OM						
6	8	9	HM		OM						
7	16	10	HM	Q	OM						
8	15	11	HM	Q,R	OM						
9	5	12	P	Q,R	OM						
10	19-58	13	Q	R	OM	P		xfer			
11	6	15	Q	R	OM	P		"			
12	19-58	16	R		OM	P,Q		"	xfer		
13	5	18	R		OM	P,Q		"	"		
14	19-58	19	OM			P,Q,R		"	"	xfer	

SCHEDULING CHART FOR THREE PROCESSOR MAPPING OF HARNESS TYPE I											
Note ref.	Processor Cycles	Label	PROCESS STATUS								
			SCHEDULING								
			Execute	Active		Inactive		P	Q	R	
				P0	P1	P0	P1	out	in	in	
15	2	6	OM			P,Q,R		"	"	"	
16	42w	7	S			P,Q,R		"	"	"	
17	14	8	S			P,Q,R		"	"	"	
18	4w-143 (min) 4w-65 (max)			P		Q,R		fin.	"	"	
19	14	14	P			Q,R			"	"	
20	11/49			Q		R			"	"	
21	14	17	Q			R			fin.	"	
22	-4/34			R						fin.	
23	14	20	R								
24	25	21	DP								
25	14	22	DP								

Empirical and Theoretical Performance Figures for Transputer Filter

Vector Siz	1_I Emp	1_I Th	2_I Emp	2_I Th	3_I Emp	3_I Thy	1_II Emp	1_II Thy	2_II Emp	2_II Thy	3_II Emp	3_II Thy
1	18.97	20.75	15.87	18.65	18.23	15.4	17.781	18.25	15.951	16.15	23.784	16.95
2	12.68	13.375	10.04	11.275	10.33	8.85	12.25	12.225	10.401	10.125	16.227	9.5
4	9.43	9.688	7	7.588	7.44	5.575	9.443	9.213	7.205	7.112	12.614	5.775
8	7.92	7.844	5.63	5.744	5.97	3.9375	7.982	7.706	5.885	5.606	10.913	3.9125
12	7.3	7.229	5.23	5.129	5.49	3.391667	7.506	7.204	5.408	5.104	10.514	3.291667
16	6.65	6.922	5.02	4.822	5.22	3.11875	7.225	6.953	5.173	4.853	10.115	2.98125
20	6.53	6.738	4.935	4.638	5.085	2.955	7.109	6.803	5.013	4.703	9.937	2.795
24	6.41	6.615	4.85	4.514	4.95	2.845833	7.006	6.702	4.9	4.602	9.818	2.670833
28	6.62	6.527	4.805	4.427	4.89	2.767857	6.935	6.63	4.825	4.53	9.7435	2.582143
32	6.83	6.461	4.76	4.361	4.83	2.709375	6.886	6.577	4.751	4.477	9.669	2.515625
36	6.975	6.41	4.84	4.31	4.8	2.663889	6.849	6.535	4.715	4.435	9.621	2.463889
40	7.12	6.369	4.92	4.269	4.77	2.6275	6.812	6.501	4.673	4.401	9.585	2.4225
44	7.24	6.335	4.94	4.235	4.75	2.597727	6.89	6.474	4.648	4.374	9.549	2.388636
48	7.58	6.307	4.96	4.207	4.73	2.572917	7.061	6.451	4.624	4.35	9.522	2.360417

Difference Between Empirical and Theoretical Performance Figures

1_I Diff	1_I %	2_I Diff	2_I %	3_I Diff	3_I %		1_II Diff	1_II %	2_II Diff	2_II %	3_II Diff	3_II %
-1.78	-9.38324	-2.78	-17.5173	2.83	15.52386		-0.469	-2.63765	-0.199	-1.24757	6.834	28.7336
-0.695	-5.48107	-1.235	-12.3008	1.48	14.3272		0.025	0.204082	0.276	2.653591	6.727	41.4556
-0.258	-2.73595	-0.588	-8.4	1.865	25.0672		0.23	2.435667	0.093	1.29077	6.839	54.21754
0.076	0.959596	-0.114	-2.02487	2.0325	34.04523		0.276	3.45778	0.279	4.740867	7.0005	64.14826
0.071	0.972603	0.101	1.931166	2.098333	38.22101		0.302	4.023448	0.304	5.621302	7.222333	68.69254
-0.272	-4.09023	0.198	3.944223	2.10125	40.25383		0.272	3.764706	0.32	6.185966	7.13375	70.52645
-0.208	-3.1853	0.297	6.018237	2.13	41.88791		0.306	4.304403	0.31	6.183922	7.142	71.8728
-0.205	-3.19813	0.336	6.927835	2.104167	42.50842		0.304	4.339138	0.298	6.081633	7.147167	72.79656
0.093	1.404834	0.378	7.866805	2.122143	43.3976		0.305	4.397981	0.295	6.11399	7.161357	73.49882
0.369	5.402635	0.399	8.382353	2.120625	43.90528		0.309	4.487366	0.274	5.767207	7.153375	73.98257
0.565	8.100358	0.53	10.95041	2.136111	44.50231		0.314	4.584611	0.28	5.938494	7.157111	74.39051
0.751	10.54775	0.651	13.23171	2.1425	44.91614		0.311	4.565473	0.272	5.820672	7.1625	74.72613
0.905	12.5	0.705	14.27126	2.152273	45.311		0.416	6.037736	0.274	5.895009	7.160364	74.98548
1.273	16.7942	0.753	15.18145	2.157083	45.6043		0.61	8.639003	0.274	5.925606	7.161583	75.21092

Appendix D

Hybrid Multiprocessor Code

```
{{{  LIBRARIES
#USE userio :
#USE interf :
#USE userhdr :
#USE filerhdr :
#USE msdos :
}}}
{{{  CONSTANTS
VAL prog.size IS #100 :
VAL data.size IS #100 :
}}}
{{{  VARIABLES
INT len.in,error,any,char,dummy :
[prog.size]INT prog.vec :
[data.size]INT t.to.dsp,dsp.to.t :
[abs.id.size]BYTE infile :
CHAN OF ANY file.in,to.net,from.net :
}}}
{{{  PLACEMENTS
PLACE to.net AT #02 :
PLACE from.net AT #06 :
}}}
SEQ
  {{{  file set up
  char := 1
  len.in := 10
  [infile FROM 0 FOR len.in] := "dcode2.dat"
  }}}
  PAR
    {{{  file service routine
    keystream.from.server(from.filer,to.filer,file.in,len.in,infile,error)
    }}}
    SEQ
      {{{  read in dsp program
      SEQ i = 0 FOR prog.size
        SEQ
          read.int(file.in,prog.vec[i],char)
      read.int(file.in,dummy,char)
      write.full.string(screen,"+    DSP  program  loaded  onto  host
transputer*c*n")
      }}}
      {{{  initialise to.to.dsp
      SEQ i = 0 FOR data.size
        SEQ
          t.to.dsp[i] := #1000000
      write.full.string(screen,"+ First input vector initialised*c*n")
      }}}
      {{{  synchronise with T801
      write.full.string(screen,"+  Press any key to synchronise with
T801*c*n")
      keyboard ? any
      to.net ! any
      }}}
      {{{  transfer DSP code to T801
      to.net ! prog.vec
      write.full.string(screen,"+     DSP   program   transferred   to
T801*c*n")
      }}}
      {{{  signal completion of acknowledgement routine
      from.net ? any
      write.full.string(screen,"+        Acknowledgement    routine
completed*c*n")
      }}}
      {{{  signal transferrence of first input vector to T801
      to.net ! t.to.dsp
      write.full.string(screen,"+    First   input   vector   has   been
transferred*c*n")
      }}}
      {{{  results to screen
      SEQ i = 0 FOR 8
        SEQ
          PAR
            to.net ! t.to.dsp
            from.net ? dsp.to.t
          write.full.string(screen,"*c*ndone transfer  ")
          write.int(screen,i,5)
              }}}
  {{{  terminate
  write.full.string(screen,"*c*n+  Press any key to terminate")
  keyboard ? any
  }}}
```

```
PROC pcode1(CHAN OF ANY in,out)
  PRI PAR
    {{{  constants
    VAL data.size.val IS 256 :
    VAL semaphore.val IS #11100 :
    VAL one IS #100 :
    VAL ack.val IS #10000000 :
    VAL prog.size.val IS #100 :
    VAL prog.base.val IS #60 :
    }}}
    {{{  variables
    [data.size.val]INT t.to.dsp,dsp.to.t :
    [prog.size.val]INT prog.vec :
    INT sync,s1,any,acknowledge,prog.base,prog.size :
    }}}
    {{{  hybrid placements
    PLACE s1 AT #27FF :
    PLACE sync AT #27FD :
    PLACE acknowledge AT #27FC :
    PLACE prog.base AT #27FB :
    PLACE prog.size AT #27FA :
    PLACE dsp.to.t AT #2000  :
    PLACE t.to.dsp AT #2200  :
    PLACE prog.vec AT #2001  :
    }}}
    SEQ
      {{{  initialise memory space to zero
      in ? any  --synchronise with host transputer
      s1 := 0
      acknowledge := 0
      {{{  zero data areas
      SEQ i = 0 FOR data.size.val
        SEQ
          t.to.dsp[i] := 0
          dsp.to.t[i] := 0
      }}}
      {{{  zero dsp prog area
      SEQ i = 0 FOR prog.size.val
        prog.vec[i] := 0
      }}}
      {{{  initialise prog parameters
      prog.size := prog.size.val
      prog.base := prog.base.val
      }}}
      }}}
      {{{  synchronisation
      s1 := semaphore.val
      in ? prog.vec
      prog.size := prog.size.val << 8
      prog.base := prog.base.val << 8
      s1 := 0
      WHILE acknowledge <> ack.val
        SEQ
          sync := sync PLUS one
        out ! any
      }}}
      {{{  wait for semaphore to be set, then in t.to.dsp
      WHILE s1 <> semaphore.val
        SEQ
          SKIP
      in ? t.to.dsp
      s1 := 0
      }}}
      {{{  main loop
      WHILE TRUE
        SEQ
          WHILE s1 <> semaphore.val
            SKIP
          PAR
            in ? t.to.dsp
            out ! dsp.to.t
          s1 := 0
    }}}
  SKIP
:
```

```
;*****************************************************************
;*                                                               *
;*                    EPROM CODE ver2   6-2-92                   *
;*  To read the size of code, and its destination base address   *
;*  from DPR, and to transfer the code vector to internal p space *
;*                                                               *
;*****************************************************************

sem1    EQU     $17FF
sync    EQU     $17FD
ack     EQU     $17FC
pbase   EQU     $17FB
psize   EQU     $17FA
ackval  EQU     $100000
pvec    EQU     $1001
bcr     EQU     $FFFE

        ORG     p:$40

        JMP     setup

boot    MOVE    #sem1,R1
        NOP
boot2   JSET    #0,x:(R1),boot2
        MOVE    x:pbase,R4
        MOVE    x:psize,x1
;----------------------------
        DO      x1,endloop
        MOVE    x:(R0)+,x0
        MOVE    x0,p:(R4)+
endloop
;----------------------------
        MOVE    x:pbase,R1
        NOP
        JMP     (R1)
;----------------------------
setup   ORI     #03,mr
        MOVEC   #0,omr
        ORI     #80,omr
        MOVEP   #0,x:bcr
        MOVE    #pvec,R0
;----------------------------
        MOVE    x:sync,a
label1  MOVE    x:sync,x0
        CMP     x0,a
        JEQ     label1
        MOVE    #ackval,x1
        MOVE    x1,x:ack
;----------------------------
        JMP     boot
;----------------------------
        END
```

```
;*****************************************************************
;*                                                               *
;*                DSP code transferred over DPR                  *
;*                                                               *
;*                Source location : #800/p:$1000                 *
;*                   Destn  location : p:$60                     *
;*                                                  LG 9-5-92    *
;*****************************************************************

size    EQU     $100
reps    EQU     $100
wrvec   EQU     $1000
rdvec   EQU     $1200
s1      EQU     $17FF
semval  EQU     $0111
out1    EQU     $FFFF
out2    EQU     $0000
setest  EQU     $10
bcr     EQU     $FFFE
ack     EQU     $17FC

        ORG     p:$60

        MOVE    #semval,y0              ;Signals xfer to T
        MOVE    y0,x:s1                 ;the code and is running it.
        MOVE    #rdvec,R0               ;Set up address registers
        MOVE    #wrvec,R1
        MOVE    #ack,R2
        MOVE    #s1,R3
        MOVEP   #$0,x:bcr
        MOVE    #>0,R4
        MOVE    #>setest,x1
        MOVE    R4,x:(R2)

        DO      #7,end1
label1  MOVE    (R4)+

        JSET    #0,x:(R3),label1        ;semaphore test
        MOVE    #wrvec,R1
        DO      #size,endloop1
        DO      #reps,endloop2
        NOP
endloop2
        NOP
endloop1
        MOVE    y0,x:s1                 ;set semaphore
        NOP
end1
        MOVE    R4,x:(R2)
        END
```

Appendix E

Hybrid Multiprocessor Performance Test Code Scheduling Charts

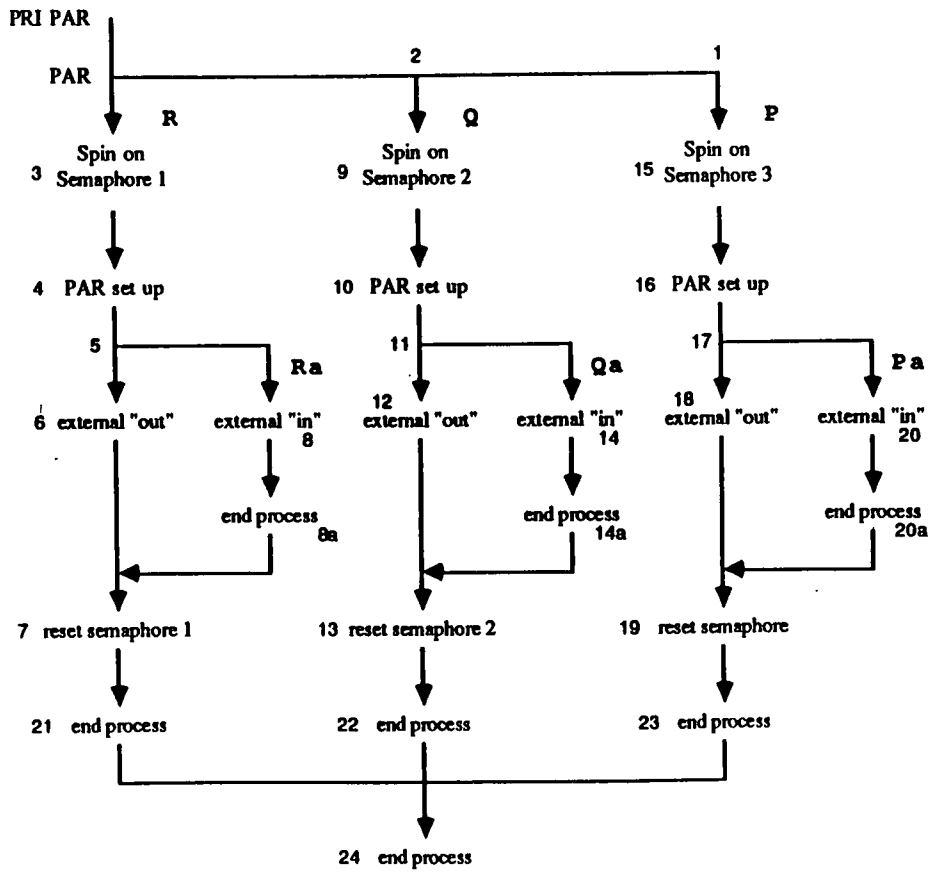


Fig E.1 Schematic Representation of Hymips Control Program Type I

SCHEDULING CHART FOR HYMIPS CONTROL TYPE I												
Note ref.	Processor Cycles	Label	PROCESS STATUS									
			SCHEDULING			COMMUNICATION						
			Execute	Active	Inactive	P		Q		R		
				P0	P0	in2	out2	in3	out3	in1	out1	
1	43											
2	11											
3	13	1		P								
4	16	2		P,Q								
5	X1	3	R	P,Q								
6	11	4	R									
7	13	5	R	P,Q,Ra								
8	24	6	P	Q,Ra	R						xfer	
9	X2	15	P	Q,Ra	R						"	
10	11	16	P	Q,Ra	R						"	
11	13	17	P	Q,Ra,Pa	R						"	
12	24	18	Q	Ra,Pa	R,P		xfer				"	
13	X3	9	Q	Ra,Pa	R,P		"				"	
14	11	10	Q	Ra,Pa	R,P		"				"	
15	13	11	Q	Ra,Pa,Qa	R,P		"				"	

SCHEDULING CHART FOR HYMIPS CONTROL TYPE I												
Note ref.	Processor Cycles	Label	PROCESS STATUS									
			SCHEDULING			COMMUNICATION						
			Execute	Active	Inactive	P		Q		R		
				P0	P0	in2	out2	in3	out3	in1	out1	
16	24	12	Ra	Pa,Qa	R,P,Q		"		xfer		"	
17	24	8	Pa	Qa	R,P,Q,Ra		"		"	xfer	"	
18	24	20	Qa		R,P,Q,Ra,Pa	xfer	"		"	"	"	
19	24	14			R,P,Q,Ra,Pa ,Qa	"	"	xfer	"	"	"	
20	Lw-(168+X2+X3)				R,P,Q,Ra,Pa ,Qa	"	"	"	"	"	"	
21				R	P,Q,Ra,Pa,Q a	"	"	"	"	"	fin	
22	X2+48				P,Q,Ra,Pa,Q a	"	fin	"	"	"		
23				P	Q,Ra,Pa,Qa	"		"	"	"		
24	X3+48				Q,Ra,Pa,Qa	"		"	fin	"		
25				Q	Ra,Pa,Qa	"		"		"		
26	24			Ra	Pa,Qa	"		"		fin		
27	16	8a	Ra		Pa,Qa	"		"				
28	t_{Tm}	7	R		Pa,Qa	"		"				

SCHEDULING CHART FOR HYMIPS CONTROL TYPE I												
Note ref.	Processor Cycles	Label	PROCESS STATUS									
			SCHEDULING			COMMUNICATION						
			Execute	Active	Inactive	P		Q		R		
				P0	P0	in2	out2	in3	out3	in1	out1	
29	16	21			Pa,Qa	"		"				
30				Pa	Qa	fin		"				
31	16	20a	Pa		Qa			"				
32	t_{Tm}	19	P		Qa			"				
33	16	23		Qa				fin				
34	16	14a	Qa									
35	t_{Tm}	13	Q									
36	16	22	Q									
37	16	24										

HYMIPS CONTROL TYPE I	
Note ref.	Comments
1	The PRI PAR is set up, and the high priority process initiated with runp.
2	The parallel processes witin the high priority process are set up.
3	Process P is placed on the high priority queue by executing a STARTP.
4	Process Q is also placed on teh high priority queue.
5	Process R cintinues by spinning on semaphore 1a.
6	R sets up a PAR construct.
7	Process Ra is placed on teh high priority queue.
8	R continues by executing an external communication, and so is descheduled. P is taken off the active queue.
9	P continues by spinning on semaphore 2a.
10	P sets up a PAR construct.
11	Process Pa is placed on the high priority queue.
12	P continues by executing an external communication and so is descheduled. Q is taken off the active queue.
13	Q begins by spinning on semaphore 3a.
14	Q sets up a PAR construct.
15	Process Qa is placed on teh active queue.
16	Q continues by executing an external communication, and so is descheduled. Ra is taken off the active queue.
17	Ra begins by executing an external communicxation and so is descheduled. Pa is taken from the active queue.
18	Pa begins by executing an external communication and so is descheduled. Qa is taken from the active queue.
19	Qa begins by executing an external communication and so is descheduled. There are no processes currently active.
20	There is now a delay of Lw-(168+X2+X3) cycles until R completes its external communication.
21	R is rescheduled, but may not continue until its sub-process has completed.
22	P completes its external communication after a further x2+48 cycles...
23	... and behaves similarly.
24	Q completes its external transfer after a further X3+48 cycles...
25	... and behaves similarly.
26	Ra completes its external transfer after a further 24 cycles.

HYMIPS CONTROL TYPE I	
Note ref.	Comments
27	Ra points to its parent process (R) and ends.
28	This allows R to continue by resetting semaphore 1a.
29	R points to its successor and ends.
30	Pa completes its external transfer and is rescheduled.
31	Pa points to its successor (P) and ends.
32	This allows P to continue by resetting semaphore 2a.
33	P points to its successor and ends.
34	Qa, which is rescheduled, ppoints to iots successor (Q) and ends.
35	This allows Q to continue by resetting semaphore 3a.
36	Q points to its successor and ends.
37	The main high priority process is now able to point to its succesor, the next PRI PAR construct, and end.

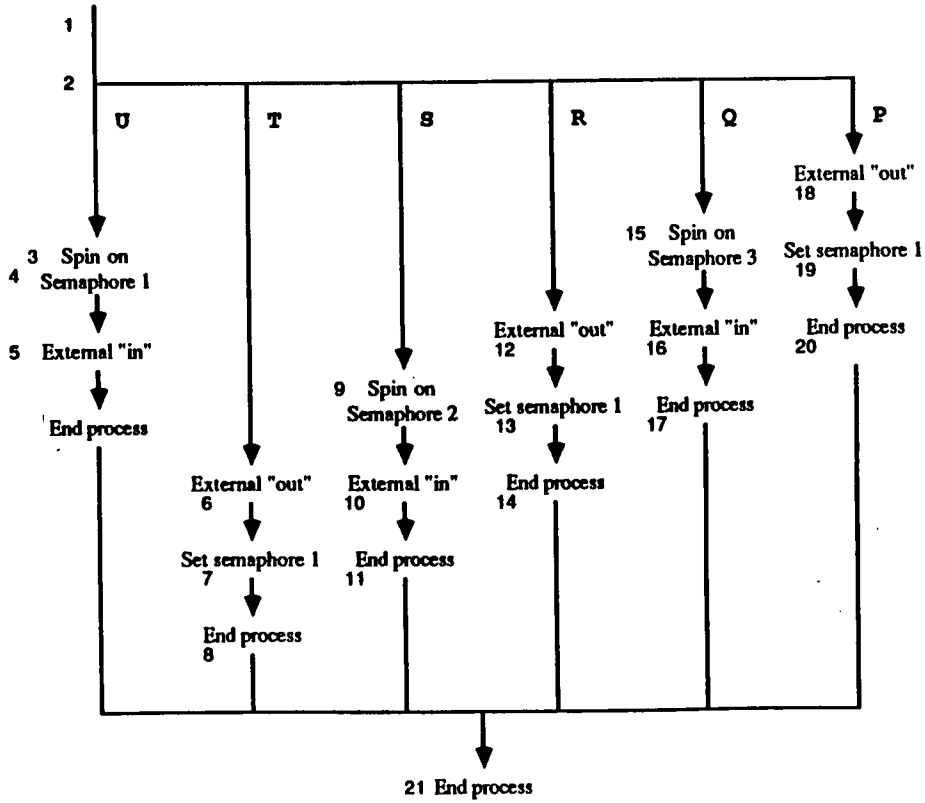


Fig E.2 Schematic Representation of Hymips Control Program Type II

SCHEDULING CHART FOR HYMIPS CONTROL TYPE II												
Note ref.	Processor Cycles	Label	PROCESS STATUS									
			SCHEDULING			COMMUNICATION						
			Exec	Active	Inactive							
				P0	P0	U	P	Q	R	S	T	
1	43	1										
2	11	2										
3	13	2a		P								
4	13	2b		P,Q								
5	13	2c		P,Q,R								
6	13	2d		P,Q,R,S								
7	13	2e		P,Q,R,S,T								
8	X1	3	U	P,Q,R,S,T								
9	24	4	P	Q,R,S,T	U	xfer						
10	24	18	Q	R,S,T	U,P	"	xfer					
11	X2	15	Q	R,S,T	U,P	"	"					
12	24	16	R	S,T	U,P,Q	"	"	xfer				
13	24	21	S	T	U,P,Q,R	"	"	"	xfer			
14	X3	9	S	T	U,P,Q,R	"	"	"	"			
15	24	10	T		U,P,R,S	"	"	"	"	xfer		

SCHEDULING CHART FOR HYMIPS CONTROL TYPE II												
Note ref.	Processor Cycles	Label	PROCESS STATUS									
			SCHEDULING			COMMUNICATION						
			Exec	Active	Inactive							
				P0	P0	U	P	Q	R	S	T	
16	24	6			U,P,Q,R,S,T	"	"	"	"	"	xfer	
17	Lw-(120+x2+x3)			U	P,Q,R,S,T	fin	"	"	"	"	"	
18	16	5	U		P,Q,R,S,T		"	"	"	"	"	
19	8			P	Q,R,S,T		fin	"	"	"	"	
20	t_{Trs}	19	P		Q,R,S,T			"	"	"	"	
21	16	20		Q	R,S,T			fin	"	"	"	
22	16	17	Q		R,S,T				fin	"	"	
23	8			R	S,T					"	"	
24	t_{Trs}	13	R		S,T					fin	"	
25	16	14		S	T						"	
26	16	11	S		T						fin	
27	8			T								
28	t_{Trs}	7	T									
29	16	8										

SCHEDULING CHART FOR HYMIPS CONTROL TYPE II											
Note ref.	Processor Cycles	Label	PROCESS STATUS								
			SCHEDULING			COMMUNICATION					
			Exec	Active	Inactive						
				P0	P0	U	P	Q	R	S	T
30	16	21									

HYMIPS CONTROL TYPE II	
Note ref.	Comments
1	The PRI PAR construct is initialised, and the high priority (main) process is set up.
2	The high priority PAR construct is set up.
3	Process P is placed on the high priority queue with a STARTP instruction.
4	Similarly for Q...
5	...R...
	...S...
7	...T...
8	U continues by spinning on semaphore 1a.
9	U executes an external communication and so is descheduled. P is taken from the active queue.
10	P continues by executing an external communication and so is descheduled. Q is taken from the active queue.
11	Q continues by spinning on semaphore 2a.
12	Q executes an external communication and so is descheduled. R is taken from the active queue.
13	R executes an external communication and so is descheduled. S is taken from the queue.
14	S continues by spinning on semaphore 3a.
15	S executes an external communication and so is descheduled. T is taken from the active queue.
16	T executes an external communication and so is descheduled. There are no remaining active processes.
17	There is now a delay of Lw-(X2+X3+120) while U completes its transfer.
18	U continues by pointing to its successor and ending.
19	After a further 8 cycles, P completes its communication and is rescheduled.
20	P continues by resetting semaphore 1a.
21	P points to its successor and ends. Q is rescheduled.
22	Q continues by pointing to its successor and ending.
23	R is rescheduled after a further 8 cycles.
24	R continues by resetting semaphore 2a.
25	R points to its successor and ends. S is rescheduled.
26	S points to its successor and ends.

HYMIPS CONTROL TYPE II	
Note ref.	Comments
27	T is rescheduled after a further 8 cycles.
28	T continues by resetting semaphore 3a.
29	T points to its successor and ends.
30	The main high priority process may now point to its successor (the next PRI PAR construct) and end.

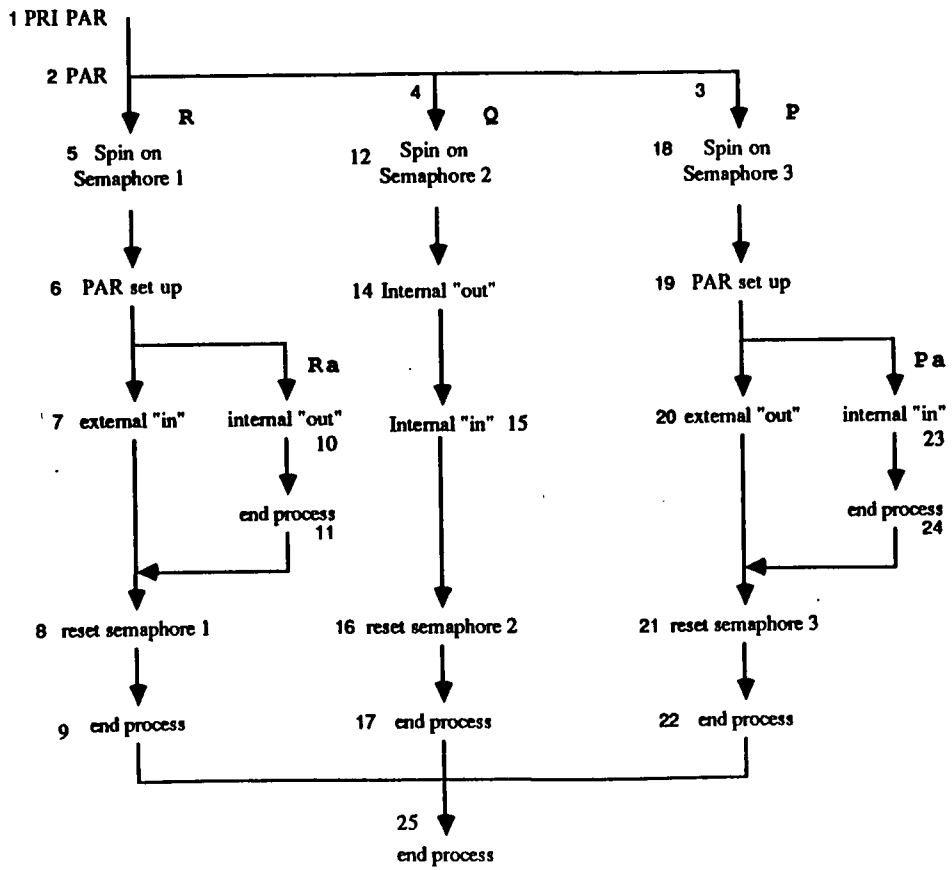


Fig E.3 Schematic Representation of Hymips Pipeline Control Program

SCHEDULING CHART FOR HYMIPS PIPELINE CONTROL												
Note Ref	Processor Cycles	Label	PROCESS STATUS									
			SCHEDULING			COMMUNICATION						
			Execute	Active	Inactive	P		Q		R		
				P0	P0	in	out	in	out	in	out	
1	43	1										
2	11	2										
3	13	3		P								
4	16	4		P,Q								
5	X1	5	R	P,Q								
6	11	6	R	P,Q,Ra								
7	24	7	P	Q,Ra						xfer		
8	X3	18	P	Q,Ra	R					"		
9	11	19	P	Q,Ra,Pa	R					"		
10	24	20	Q	Ra,Pa	R,P		xfer			"		
11	X2	12	Q	Ra,Pa	R,P		"			"		
12	24	14	Ra	Pa	R,P,Q		"	H		"		
13	4w+24	10	Ra	Pa,Q	R,P		"	F		"	C	
14	16	11	Ra	Pa,Q	R,P		"			"		
15	24	23	Q		R,P,Pa	H	"			"		

SCHEDULING CHART FOR HYMIPS PIPELINE CONTROL												
Note Ref	Processor Cycles	Label	PROCESS STATUS									
			SCHEDULING			COMMUNICATION						
			Execute	Active	Inactive	P		Q		R		
				P0	P0	in	out	in	out	in	out	
16	4w+24	15	Q	Pa	R,P	F	"		C	"		
17	t_{Tn}	16	Q	Pa	R,P		"			"		
18	16	17	Pa				"			"		
19	16	24	Pa		R,P		"			"		
20			R	P			"			fin		
21	t_{Tn}	8	R				"					
22	16	9	R		P		"					
23	43+X3-t_{Tn}		P				fin					
24	t_{Tn}	21	P									
25	16	22	P									
26	16	25										

HYMIPS PIPELINE CONTROL	
Note ref.	Comments
1	PRI PAR construct is initialised.
2	The high priority process is set up.
3	Process P is placed on the high priority queue by executing a STARTP instruction.
4	Similarly for process Q.
5	R continues by spinning on semaphore 1a.
6	Process Ra is placed on the high priority queue.
7	R executes external communication and so is descheduled. P is taken from the queue.
8	P continues by spinning on semaphore 3a.
9	Pa is placed on the queue.
10	p executes an external communication and so is descheduled. q is taken from the queue.
11	Q continues by spinning on semaphore 2a.
12	Q executes an internal communication . However, the channel is empty and so Q is descheduled. Ra is taken from the queue.
13	Ra continues by executing an internal communication, corresponding to that of Q. The transfer takes place and Q is rescheduled.
14	Ra points to its successor and ends. Pa is taken from the queue.
15	Pa continues by executing an internal communication. The channel is empty and so Pa is descheduled. Q is taken from the queue.
16	Q continues by executing an internal communication, corresponding to that of Pa. The transfer takes place and Pa is rescheduled.
17	Q continues by resetting semaphore 2a.
18	Q points to its successor and ends. Pa is taken from the queue.
19	Pa points to its successor and ends.
20	There is now a delay until R completes its link transfer.
21	As Ra has completed, R is allowed to continue by resetting semaphore S1a.
22	R points to its successor and disappears.
23	P completes its link transfer and is rescheduled.
24	P is allowed to continue by setting semaphore 3a.
25	P points to its successor and disappears.

HYMIPS PIPELINE CONTROL	
Note ref.	Comments
26	The high priority process points to its successor (the next PRI PAR construct) and ends.
27	
28	

Appendix F

Background References

Further background on the work presented in this thesis is presented in the following conference papers:

Gould G L, Bowler I and Purvis A, *"Real-Time, Multi-Channel Digital Filtering on the Transputer"*, IEE Symposium on Computer Architectures and Digital Signal Processing, Hong Kong, September 1989.

Gould G L, Linton K N, Terepin S and Purvis A, *"Multiprocessor Architectures and Allocation Strategies for Digital Audio Mixing Consoles"*, Reproduced Sound 6, Windermere, Great Britain, November 1990.

Linton K N, Gould G L, Terepin S and Purvis A, *"Real-Time, Multi-Channel Digital Audio Processing: Scalable Parallel Architectures and Taskforce Scheduling Strategies"*, 1991 IEEE Conference on Acoustics, Speech and Signal Processing, Toronto, Canada, May 1991.

Linton K N, Gould G L, Terepin S and Purvis A, *"Optimising Massive Parallel Architectures for Real-Time Digital Audio"*, 89th Audio Engineering Society Convention, Los Angeles, USA, September 1990.