SOME STUDIES AND NEW RESULTS ON MULTI MICROPROCESSOR APPLICATIONS

A THESIS SUBMITTED BY

K. POULOSE JACOB

IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
UNDER THE FACULTY OF TECHNOLOGY

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF ELECTRONICS
KOCHI - 682 022, INDIA

1991

ACKNOWLEDGEMENT

I am greatly indebted to **Dr.C.S.SRIDHAR**, Professor, Department of Electronics, and Dean of Technology, for his kind hearted efforts in guiding and educating me all through. His intellectual pursuits have been a constant inspiration to me. I thank him; I thank him with love, respect and admiration.

I am very grateful to Dr. **K.G.NAIR**, Professor and Head of the department of Electronics, for providing me the facilities. I thank him for his support and encouragement.

I thank **Dr. A.K. MENON**, Professor and Head, Department of Computer Science, where I am presently occupied, for his support and for the enchanting physical environment provided.

I am thankful to my colleagues on the faculty, for their unstinted co-operation throughout. My thanks to the lab staff, research scholars, students and administrative staff.

Cochin University of
Science and Technology.

K. Poulose Jacob.

CERTIFICATE

This is to certify that the thesis entitled, "SOME STUDIES AND NEW RESULTS ON MULTI MICROPROCESSOR APPLICATIONS" is a report of the original work done by Sri. K. Poulose Jacob under my supervision and guidance in the Department of Electronics, Cochin University of Science and Technology, and that no part thereof has been presented for the award of any other degree.

[illegible]

Dr. C.S. Sridhar

Cochin 682 022,

10..9..1991.

Professor,
Department of Electronics,
Cochin University of Science and Technolgy.

DECLARATION

I hereby declare that this thesis is a report of the original work done by me under the supervision of Dr. C.S. Sridhar, in the Department of Electronics, Cochin University of Science and Technology, and that no part thereof has been presented for the award of any other degree.

Cochin 682 022,

10..9..1991.

[illegible]

K. Poulose Jacob.

CONTENTS

-:0:-

SOME STUDIES AND NEW RESULTS ON MULTI MICROPROCESSOR APPLICATIONS

ABSTRACT

One of the fastest expanding areas of computer exploitation is in embedded systems, whose prime function is not that of computing, but which nevertheless require information processing in order to carry out their prime function. Advances in hardware technology have made multi microprocessor systems a viable alternative to uniprocessor systems in many embedded application areas.

This thesis reports the results of investigations carried out on multi microprocessors oriented towards embedded applications, with a view to enhancing throughput and reliability.

An ideal controller for multiprocessor operation is developed which would smoothen sharing of routines and enable more powerful and efficient code / data interchange. Results of performance evaluation are appended.

A typical application scenario is presented, which calls for classifying tasks based on characteristic features that were identified. The different classes are introduced along with a partitioned storage scheme. Theoretical analysis is also given.

A review of schemes available for reducing disc access time is carried out and a new scheme presented. This is found to speed up data base transactions in embedded systems.

The significance of software maintenance and adaptation in such applications is highlighted. A novel scheme of providing a maintenance folio to system firmware is presented, alongwith experimental results.

Processing reliability can be enhanced if facility exists to check if a particular instruction in a stream is appropriate. Likelihood of occurrence of a particular instruction would be more prudent if number of instructions in the set is less. A new organisation is derived to form the basement for further work. Some early results that would help steer the course of the work are presented.

Chapter 1

INTRODUCTION

1.1 Background

State of the art parallel computer systems can be characterised into three structural classes [Hwang K. et al, 1985] :

Pipelined computers

Array processors

Multiprocessor systems.

A pipelined computer performs overlapped computations to exploit temporal parallelism. An array processor uses multiple synchronised arithmatic logic units to achieve spatial parallelism. Multiprocessor systems achieve asynchronous parallelism through a set of interactive processors with shared resources (memories, data bases etc.). The fundamental difference between the array processor and the multiprocessor system is that the processing elements in an array processor operate synchronously while processors in a multiprocessor system operate asynchronously.

A pipeline has several processing stages, each stage repeatedly executes the same instruction on successive pieces of data received from a preceding processor; thus the result of one processor becomes the input of the next. This according to Flynn's classification of architectures, is Multiple Instruction stream, Single Data stream (MISD).[Flynn M.J., 1966, 1972]. The pipelining technique is appropriate for applications in which the major functions are dependent on each other and the data sets are very large, for eg. Signal processing of real time data.

By Flynn's terminology, a conventional single processor system that works on one set of data may be called Single Instruction stream, Single Data stream (SISD). This class comprises serial computers in which instructions are executed sequentially but may be overlapped in their execution stages (instruction level pipelining).

In array processors, several processors under a centralized control, execute the same instruction, each on a disjoint set of data, for eg. Vector addition. Classified as Single Instruction stream, Multiple Data stream (SIMD), array processors are special purpose machines suitable for applications where data structures like arrays are natural units to operate upon. These are advantageous only for programs with a high percentage of vectored instructions.

In multiprocessor systems, each processor can execute a different sequence of instructions on a different set of data. This is named Multiple Instruction stream, Multiple Data stream (MIMD). Such asynchronous processors are the most suitable ones for performing concurrent processing which consist of a collection of co-operating or interacting processes that communicate with each other.

Another class that is found to be grouped under MIMD, is the computer network, wherein each processor is embedded in a conventional computer system, and the computers are then interconnected via communication links.

Networks, multiprocessors and array computers can be thought of as varying along a single dimension - the degree of coupling between the processors in the system. Patnaik defines this as the worst case processors minimum access time to a global data structure in the system.[Patnaik L.M., 1983]. In a multi- processor, each processor has direct access to global data stored in primary memory. Since interprocessor communication occurs by sharing primary memory, the interaction times are faster. In computer networks, global data resident in one of the computers, may have to be accessed through a hop sequence involving other computers. This takes more time. In array computers, the analog of interprocessor communication is the transfer of control information that occurs between the control unit and its associated processing elements.

The average time between interprocess interaction becomes a crucial time constant of an application and provides a good indication of the type of multiple processor organisation that will be most suitable.

1.2 Scope

Parallel processing computers are needed for large scale computations often performed in application areas like multidimensional modelling of the atmosphere/outer space, numerical weather forecasting and the like. Much work is reported in enhancing such raw computing power and state of the art approaches achieve speeds of the order of 1000 million megaflops or beyond.

The work presented in this report is oriented towards a different realm of applications namely flight/rocket-launching control, industrial instrumentation, data base applications, combat vehicle control and the like. These applications are not exclusively computing environments involving number crunching, but nevertheless call for high availability as well as adaptability. Multiprocessors, especially those constructed of relatively low cost microprocessors, offer a cost effective means of achieving the objective. A key feature of these applications is the role of computer as an information processing component within a larger engineering system. Systems for such applications have come to be known as embedded computer systems. The Oxford Dictionary of Computing defines Embedded Computer System as any system that uses a computer as a component, but whose prime function is not that of a computer. Multiprocessor embedded systems have the potential advantage of enhanced throughput and reliability. It is observed that such systems make use of time tested components and software, and changes are incorporated into them as and when new requirements arise. As a matter of fact, advanced processor architectures are not popular in those systems.

1.3 Motivation

A multiprocessor system is often described as throughput oriented, when it is designed to maximise the throughput of various jobs.[Dubois M. et al, 1988]. Jobs in such an environment are distinct from one another and execute as if they were running on different uniprocessors. Patton gives some of the fundamental characteristics of throughput oriented multiprocessing.[Patton P.C. 1985]:

* general purpose
* multiple applications
* fail soft requirements : that of recording the last operational state , in the event of failure.
* maximum number of independent jobs done in parallel
* CPU and I/O balanced workload.

The goal of throughput oriented multiprocessing appears to be obtaining high throughput at minimum cost.

Kim holds that a loosely coupled multiprocessor with either a partitioned or shared data base offers the best architecture for a highly available system.[Kim W. 1984]. For embedded applications, a small amount of shared memory would be advantageous, owing to the fact that message passing through an interconnection network run the risk of messages getting delayed, threatening the reliability of the system.

In spite of their advantages, a number of problems and open issues remain to be resolved before such systems are a practical alternative to more conventional organisations. The major problems currently facing such systems are

1. What mechanisms are appropriate that allow processors to share tasks and data.
2. How would tasks now executed on uniprocessors be decomposed so that they can be run on a set of smaller processors.
3. How should I/O devices in general and secondary storage devices in particular be used as part of the system.
4. What hardware and software structure will allow the system to realize its potential for reliability and adaptability.

1.4 Outline of the work

The current work is oriented towards a multimicroprocessor set-up involved in embedded systems. Fig.1.1 shows a typical embedded computer system. Several processors may be assigned to support the system components. On the software side, among the modules present there can be one for physically manoeuvering the devices/controls, one for monitoring the system, one for data base functions and one to interact with the operator. These modules are required to be maintained

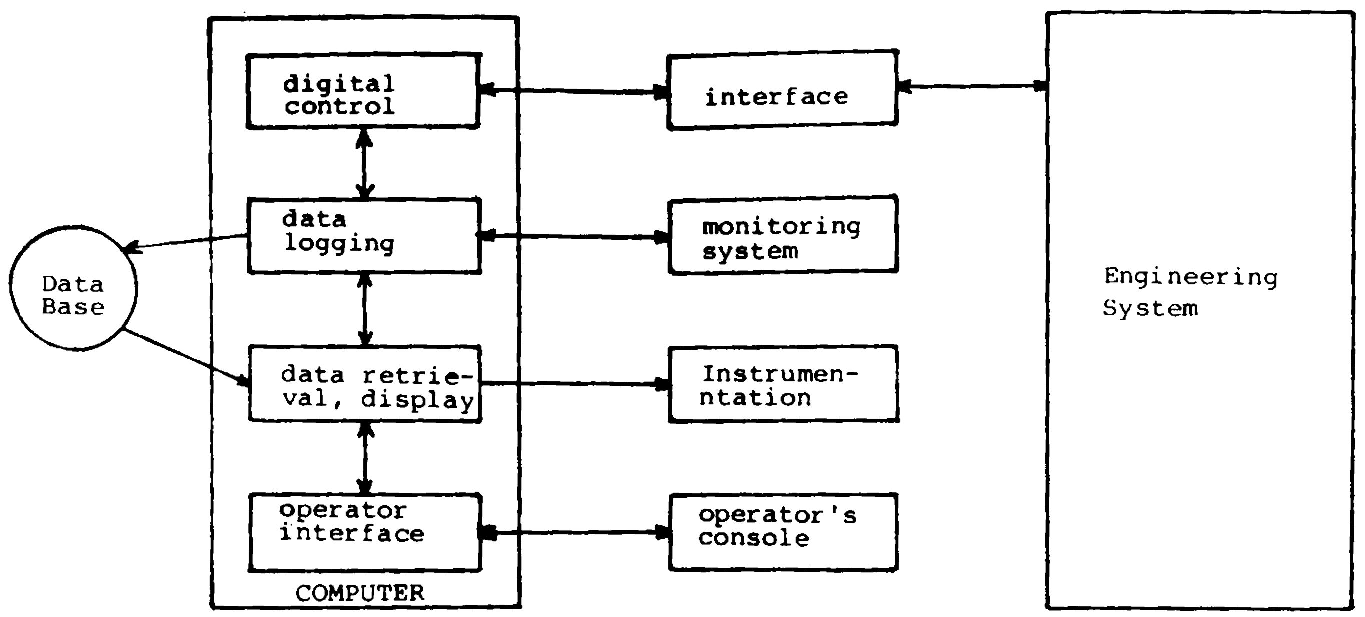


Fig 1.1 Typical Embedded Computer System

and adapted according to modifications required in the application. Changes in basic hardware, like switching over to a new family of processors, are seldom.

The various issues identified for investigation cover :

1. Impediments in DMA accession while sharing tasks/data between processors.
2. Task classification and contention problems in shared / distributed memory.
3. Storage efficiency for data base.
4. Software maintenance techniques and adaptability.
5. Sequence prediction of instruction streams.

The objective of the investigation is to improve throughput and to provide an environment for cost effective processing. The relevance of these issues has been brought out in the respective sections of Chapter 3.

Chapter 2 presents a review of the basic concepts which are relevant for this work. Current trends in modern microprocessor architectures are mentioned with a view to suggesting further work.

Chapter 3 presents current status of the various issues identified, along with their relevance. It is organised as three levels and work carried out at each level is reported in the chapters that follow.

Chapter 4 presents a new design for a family of controllers that incorporates a stack and other associated controls which would enable more powerful data/code interchange; not merely between the processor and the peripheral, but also between processors. Results of computer simulation of the design as well as a comparison with popular models currently in use are included.

Chapter 5 presents the scenario of a typical application environment. A scheme of partitioning the common resource memory into functional blocks is described, where each block is earmarked to hold predefined classes of tasks that are identified. Theoretical analysis of the scheme is also given.

Chapter 6 deals with the data base component of the embedded system and presents a review of work reported on general data base access. A new procedure that reduces disc access time is presented, along with quantitative results.

identified, along with their relevance. It is organised as three levels and work carried out at each level is reported in the chapters that follow.

Chapter 4 presents a new design for a family of controllers that incorporates a stack and other associated controls which would enable more powerful data/code interchange; not merely between the processor and the peripheral, but also between processors. Results of computer simulation of the design as well as a comparison with popular models currently in use are included.

Chapter 5 presents the scenario of a typical application environment. A scheme of partitioning the common resource memory into functional blocks is described, where each block is earmarked to hold predefined classes of tasks that are identified. Theoretical analysis of the scheme is also given.

Chapter 6 deals with the data base component of the embedded system and presents a review of work reported on general data base access. A new procedure that reduces disc access time is presented, along with quantitative results.

Chapter 7 dwells upon software maintenance as a prelude to adaptability. A maintenance folio is propounded, as part of firmware, which would express the underlying concepts clearly, bringing out all pertinent information in well structured fields. Quantitative results are also given. A case report of software reuse, namely a standard assembler adapted to support simultaneous processing in a multi microprocessor set-up is also included.

Chapter 8 attempts to extend the concept of software adaptation and suggests a well defined instruction format that would allow rapid decode through use of a consistent opcode field, with a view to assisting adaptability and sequence prediction. Some early results on instruction stream analysis, which would help predict a plausible sequence are also presented. This is intended to be a basement for further work in the field.

Chapter 9 presents the conclusions. Comments on the scope for further work in the field are also included.

Chapter 2

REVIEW OF BASIC CONCEPTS

This chapter presents a review of the basic concepts associated with this work, and subjected to the study.

2.1 Architectural Class

The multi microprocessor system comes under the class of MIMD machines in Flynn's taxonomy. Several other efforts on taxonomy are reported in [Kung, 1982], [Basu, 1987] and [Johnson, 1988]. Based on these studies Krishnamurthy has identified four attributes for a taxonomical tree. [Krishnamurthy E.V. 1989]. These attributes and the corresponding level in the case of a multiprocessor are

i.granularity : coarse grained.
ie., ratio of computation to communication is high.
[Howe C.D., et al,1987].

ii.nature of algorithm realisation module [Kung,1982] :

mixed hardware - software.

iii.topology and nature of coupling :

simple (eg.bus), shared variable, lightly coupled.

iv.control : asynchronous.

A precise classification of complex architectures require additional attributes like task allocation, routing, language issues and such other factors.

As regards nature of coupling, the general trend observed is: when the goal of the system is raw computing power, the architecture will be of the tightly coupled type. When general purpose applications are intended with numerous small modules, the coupling is necessarily of the loose type. A moderate coupling is preferred when a homogenous multiprocessor is designed with a few processors and a general application like a controller, in mind.

The notion that a loosely coupled collection of processors could function as a more powerful general purpose processing facility has existed for quite some time. [Casavant T.L.,et al,1988].

2.2 Software issues

In order to realize the strict definition of multiprocessing, three options appear to be available on the software side.

i. Design algorithm such that they take into account the parallel architecture.

ii. Specify parallelism by the user, finally scheduled by the operating system.

iii. Detect automatically the parallelism.

The throughput of a multiprocessor, multiprogrammed as per the first option has constraints to the extent that in a given program, the amount of parallelism is not uniform and it is seldom that all the processors are kept consistently busy. Memory contentions are also bound to degrade performance. Detecting parallelism during compilation/ assembly is a process which has acquired considerable importance.

The UNIX operating system makes extensive use of variants of FORK and JOIN for specifying concurrent processes.[Bourne, 1980]. The fork operation splits a parent process into two processes, each of which can run concurrently until required. The join operation recombines the two processes into one process, with a provision for waiting (delay) if needed. Fork-Join technique is appropriate for applications in which no major functions require the results of another; each major function is independent of the others. For eg. one may compute the median and the mean of the same set of data simultaneously.

Parallelism is also specified by the command

cobegin P1 P2 ..... Pn **coend**

which causes processesP1, P2,....Pn to start simultaneously and to proceed concurrently until they have all ended.

parbegin P1 P2 .... Pn **parend** is an equivalent command.

These statements provide a structured single entry, single exit control and are not as powerful as fork-join.

2.3 Language issues

One of the main characteristics of an embedded system is the need to interact with devices, all of which having their particular characteristics. The programming of such devices has traditionally been in assembly language, but of late languages like Ada and Occam attempt to provide high level support, particularly for ditributed systems.

2.3.1 Ada

Ada uses the word 'task' for a process to indicate a sequence of actions which are executed in parallel with other actions. The task has two parts : the specification and the body. Every task is written in the declarative part of some enclosing program unit called the parent. Burns et al, presents a detailed study of Ada tasking.[Burns et al, 1987]. Inter task communication can take place in either of two ways:

i. shared variable

ii. message passing mechanism called the rendezvous

In fact Ada's main innovations are its powerful rendezvous form of communication.

The main purpose of its design seem to be programming real time systems.

2.3.2 Occam

Occam was originally developed for use in a single chip computer with a processor, local RAM and four dedicated input-output links - The TRANSPUTER. Generally used in implementing parallel algorithms on a transputer type network, it lacks high level features such as recursion and the pointer data type. Burns gives a detailed comparison with Ada.[Burns, 1988].

The full potential of transputers is realized only when they are grouped together. They use point to point communication, which has the disadvantage that a message may have to be forwarded to its destination via intermediates if no direct link is available.

The PARAM computer developed by C-DAC is based on transputers. However they make extensive use of assembly level programs. Occam supports a hierarchical structure and is applicable for systems built from a large number of concurrently operating processes.

Table 2.1 summarises the facilities provided by Ada and Occam.

2.4 Communication

The general procedure to map a problem onto a parallel architecture is that the programmer must first devide the problem into segments that will run in parallel, then determine how the processors will communicate and synchronize

	Ada	Occam
Support for decomposition of large programs into modules	Yes	No
Support for concurrent programming	Yes	Yes
Support for execution in a distributed environment	With the help of tools	Yes
Facilities for fault tolerant programming	Exceptions	None
Mode of device Handling	Shared memory	Message passing

Table 2.1 Facilities provided by Ada, Occam

with one another. Shared memory and message passing, which characterise the level of coupling, are also the best form of communication between processors.

In shared communications, data written by one processor can be read by all other processors in the system. Message passing, on the other hand is point to point communication; it is more restrictive of the two because each message must go to a specified recepient.

The method used to synchronise processors depends on the approach taken to communication. Message passing is a blocking method that synchronises processes implicitly; a processor requesting data must suspend its operation until the communication process is completed. Shared memory communication is typically non blocking; a key feature is that the access time to a piece of data is independent of the processor making the request.

Hybrid systems have some of the properties of shared memory sytems and some of the properties of message passing. All memory is local to a given processor, but the operating system makes the machine look like it has a single global memory. As far as the programmer is concerned, hybrid systems are coded like shared memory systems, but have data access delays like message passing systems. Since the access time depends on the distance between the owner of the data and the requester, the data must be laid out properly.

Algorithms are easy to design for shared memory systems; one simply puts the data in memory as if running on a uniprocessor. Programs are hard to debug, an error usually occurs while picking up data from a global variable. The processor continues computing producing erroneous final result. There is no indication of time of occurrence of error.

In message passing systems, algorithm design is hard because the data must be distributed so that communication traffic is minimized. Debugging is easier because errors usually stop the system at the point of error. Thus the programmer knows the machine status at the point where the error occurred.

2.5 Trends in modern microprocessors

Manufacturers tend to integrate critical peripheral functions on chip, to better emulate system type performance.

Functions like memory management, instruction cache, data cache, pilelining are chosen for in-chip integration.

MMU(Memory Management Unit) translates the virtual addresses generated by the CPU into real addresses. The former is limited only by the maximum number of bits possible in an address (ie.,the width of the PC), while the latter is based on the physical main memory.

The cache system effectiveness depends mainly on two factors:

* the hit rate, the frequency that the desired instruction / data is actually found in the cache
* the replacement time on a cache miss, the number of clock cycles required to fetch an item from main memory into the cache, when the cache miss occurs.

Pipelining in its simple form permits the three phases of fetch, decode and execute to proceed independently without having to wait for one instruction to finish all the three.

The compromises that were required in order to incorporate these advanced features in a microprocessor, though minor compared to what they achieve, made a shift of emphasis from a complex instruction set to simplified operations. RISC architectures evolved as a result of this trend. Gimarc and others list some features commonly seen in RISC.[Gimarc C.E.,et al, 1987].

* fixed instruction format for simple decoding
* relatively few instructions and address modes
* highly pipelined datapath for concurrency
* many levels of memory hierarchy
* load store instruction set
* hardwired instruction decoding
* single cycle execution of most instructions.

RISC processors have quickly moved into many different application areas indicating that RISC philosophy can be applied to embedded systems.

The i860 XP, the 64 bit microprocessor recently launched by Intel provides a peak performance rate of 100 million FLOPS at 50 mHtz. It is claimed to be ideally suited to meet the complex number crunching computing needs of scientific, engineering and graphics applications. Packaged in a 262 pin ceramic pin grid array, it has inherent support for multiprocessing. Peripheral components like the 82495 XP cache controller, the 82490 XP cache RAM and a multiprocessing interrupt controller are used in multi processor system based on the i860 XP.

Chapter 3

CURRENT STATUS OF ISSUES IDENTIFIED

This chapter presents a vivid picture of the state of the art of the various issues identified in the introductory chapter. Three distinct levels of sharing, namely control, information and storage, which are inherent in multi microprocessor systems are subjected to a critical review, with a view to enhancing the throughput.

3.1 Shared Control and DMA

Early definitions identify software interaction and shared memory as typical multiprocessor features.[Enslow P.,1974]. The memory in a multiprocessor appears to play three distinct roles :

i. Instruction and data storage for a particular processor

ii. Temporary storage for data transfers between processors

iii. Storage of instructions common to several processors.

In several cases, many identical operations are performed on different data by different processing elements, and a process often becomes a resource which may have to be shifted from the domain of one processing element to that of another. The sharing of codes and the associated transfer of data often becomes a frequent requirement in any multiprocessing system.

The current approach to handle multiprocessor systems appears to rely on the methods developed for uniprocessors and their extensions. Most systems are constructed to meet the specifications of a given environment with essentially known techniques and components.[Mohan C. et al, 1981]. A major constraint of a microprocessor in a multiprocessor structure is the fact that the internal circuits are not available to the designer and all interconnections have to be done through the external bus. Only existing processor facilities can be utilized for the structure envisaged, with no possibility of enhancement. [Paker Y.,1983]. Hence extending and modifying an available scheme, to suit the new environment is advocated.

3.1.1 Assessment of schemes available

Transfer of data between an external device and main memory has been a common occurrence, the external device being a mass storage unit or data terminal; in the current context another processing entity might be involved.

If the data transfer rate to or from an I/O device is relatively low, then the operation can be performed using either programmed or interrupt I/O. But executing instructions and performing interrupt sequences take more time than is sometimes available. Data rates for mass storage devices are often determined by the devices, and the computer must be capable of executing I/O according to the maximum speed of the device. For a disk unit data rate is determined by the speed with which data pass under the read/write head, and quite often this rate exceeds 200000 bytes per second. Thus there is less than 5 microseconds to transfer each byte to or from memory. For data rates of this magnitude, block transfers involving DMA are required.

3.1.2 DMA in Multiprocessor

The parallel bus which forms one of the main schemes used for

3.1.1 Assessment of schemes available

Transfer of data between an external device and main memory has been a common occurrence, the external device being a mass storage unit or data terminal; in the current context another processing entity might be involved.

If the data transfer rate to or from an I/O device is relatively low, then the operation can be performed using either programmed or interrupt I/O. But executing instructions and performing interrupt sequences take more time than is sometimes available. Data rates for mass storage devices are often determined by the devices, and the computer must be capable of executing I/O according to the maximum speed of the device. For a disk unit data rate is determined by the speed with which data pass under the read/write head, and quite often this rate exceeds 200000 bytes per second. Thus there is less than 5 microseconds to transfer each byte to or from memory. For data rates of this magnitude, block transfers involving DMA are required.

3.1.2 DMA in Multiprocessor

The parallel bus which forms one of the main schemes used for building multi microprocessor systems, utilizes the external bus of the microprocessors or its extension as the main transmission medium.

The microprocessor under program control often initiates the transfer of a block of data and might specify the number of words comprising the block. The transfer of individual words are however controlled by the circuitry that is separate from the microprocessor. The program might include instructions that output to the DMA control circuitry, the number of data words to be transferred and the beginning address of where they are to be located in the main memory. The program would then set a flag to commence the transfer. From that point, the program could go on to do some other function (not involving the bus) while the external control circuitry attended to the transfer. This is the conventional DMA scheme. Fig. 3.1 shows the flowchart of the controller operation. The DRAWBACK of this convention is that the processor has to initiate each transfer of a block separately. This involves notifying the location of the block and the number of words comprising the block, generally mentioned as transfer parameters.

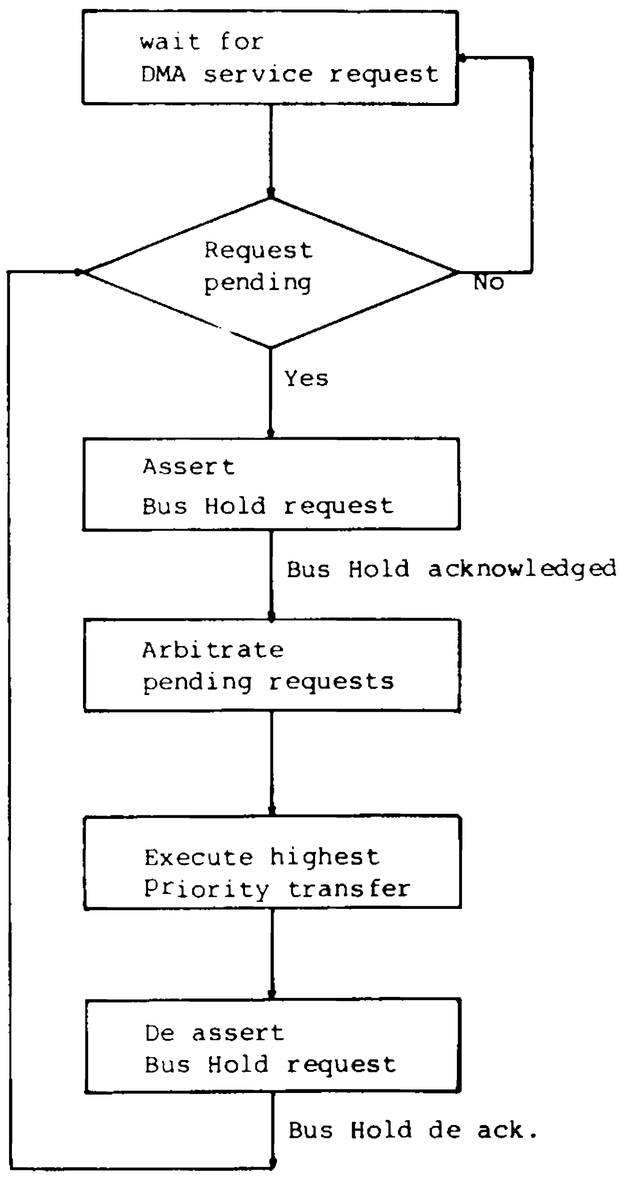


Fig 3.1 DMAC operation

3.1.3 Multichannel DMA Controller

Currently available are DMA controllers, which contain several independent channels. Contention between channel requests is resolved by resorting to one of the two programmable modes, namely fixed priority and rotating priority. After being initialized by software, these controllers can transfer a block of data between memory and a peripheral device directly, on each channel, without further intervention by the CPU. Access is limited to a block of data sequentially located in main memory. A single pair of channel, when programmed appropriately, simulates memory to memory transfer, but successive transfers involving non sequential locations are impossible.

The autoinitialize feature, when selected by program, permits another DMA service without CPU intervention upon detecting a valid DREQ, but the limitation is that transfer parameters are again the same. Further a DREQ is required to be generated to activate this.

3.1.4 Relevance

The nature of embedded systems requires the computer components to interact with the external world. They need to monitor sensors and control actuators for a variety of devices. Devices may also generate interrupts in order to signal the processor that certain operations have been performed. When multiple microprocessors exist to support such a system, block transfers are required. For instance, different modules of a program scattered in the domain of a particular processor, may have to be shifted to the domain of another processor. A typical scenario is described in Chapter 5.

The data base support to the embedded system (see fig.1.1) is another unit which requires block transfer. A typical set of data might be required for different processing units. This can be accomplished by providing direct access to an I/O buffer by concerned processing unit memory.

A new family of controllers is proposed in Chapter 4, which will take care of such transfers. The drawbacks mentioned are eliminated in the design.

3.2 Shared Information

It is essential that a proper framework should exist for concurrent processes, that are likely to run on distributed systems, to share information, in order to achieve a given application objective. The communication structure required might be quite complex; little is reported about how to optimally connect many processors together.

Hoffner mentions two important concepts of concurrent programming namely [Hoffner Y.,1983] :

i. Mutual exclusion

"...it is the abstraction of many synchronisation problems"[Ben-Ari M., 1982]. It deals with excluding any process from using a resource in use until it has been released.

ii. Communication between processes

the provision of data transfer mechanisms that allow, for example, computed results to be passed from one processor to another.[Bowen B.A.,et al, 1980].

The main feature needed in order to implement communication between processors, is the ability of passing information items from the addressing space of one processor to the addressing space of another. This can be accomplished by the shared memory scheme of writing to a particular memory location by one processor and reading from that location, by another. This is implemented by mutually exclusive accesses to mailboxes which are configured and maintained in such memory.

A limitation of the shared memory system is that if the number of processors is large, performance is found to decay, as concurrent accesses to memory, by more than two or three processors, are not feasible. This is alleviated by providing some local memory to each pocessor and simulating shared memory with caching techniques.

Another limiting factor is the memory access latency, ie., the delay between the instant of time when a processor emits an address and the time when the data is returned from memory. If the memory access latency exceeds about one instruction time, the processor must idle, until the storage cycle completes.[Athas W.C., et al, 1988]. But the merit of the scheme lies in the fact, that it allows co-operating programs on different processors to share information, as long as care is taken to ensure synchronisation.

In the case of local memory systems, if every processor attempts to communicate directly with every other processor, the complexity of the communication network would rise proportional to p^2, where p is the number of processors in the system. In such situations, the communication has to be made indirect, with information passing through intermediate processors resulting in an increase in communication delay. Software is likely to be complex, so also synchronisation procedures. This tantamounts to a computer network, where the interconnection is often via serial communication links. Usually here, the communication takes much longer than the processing time involved.

3.2.1 Synchronisation

Synchronisation problems are found to receive a great deal of attention in the literature.[Hoare C.A.R.,1980]. The main trends that can be observed are :

i. Linguistic approach, where new constructs are defined. eg. semaphore.

ii. Mechanisms such as circulating tokens to handle mutual exclusion or locks to handle concurrency.

iii. Communication approach/protocols to ensure point to point data transfer.

Solution for a problem of synchronisation of processes in a distributed system tend to differ, depending on whether the system is installed on a multiprocessor architecture, with a common memory or without. If a common memory exists, semaphores and monitors serve as useful tools whereas a controller process might be necessary on an architecture without a common memory.[Herman D.,1983]. This centralised controller would not conform much to the very aim of multiprocessing, owing to its limitations namely,

i. processes are slowed down by the exchange of messages with the controller, since in the general case, it is on a remote site.

ii. if the site containing the controller malfunctions, the entire system ceases to function.

Nevertheless, if all processors in a multiprocessor shared one single memory, access conflicts would largely neutralize the performance potential of the multiple processors.

3.2.2 Current practices

Shared memory communication is possible with ADA, although it is not the preferred method. Ada does not directly support SEMAPHORES, but the WAIT and SIGNAL procedures can be constructed from Ada synchronisation primitives. Semaphores can be criticized as too low level and error prone and hence not adequate for real time domain. Nevertheless semaphores provide a means to program mutual exclusion over a critical section.

Both Ada and Occam allow communication and synchronisation based on message passing. With Occam this is the only method available. Ada uses remote invocation with direct asymmetric naming. Occam by comparison, contains a synchronous indirect symmetric scheme.

However, Ada programming style is a little too complex and it is still under debate whether some of its features are effective; in particular it is not known whether the tasking model is a natural and effective one.[Burns A. et al, 1987].

A scheme of partitioning main memory and classifying tasks, with a view to obviating complex communication procedures is presented in Chapter 5.

3.3 Storage

Madnick and Donovan differentiate the three types of storage devices on the basis of the variation of access time T_{ij} [Madnick et al, 1974] where

T_{ij} = time to access item j given current position is item i; where T_{ij} has a large variance, it is serial access, while constant T_{ij} pertains to completely direct access. Direct access devices have only a small variance in T_{ij}. These components generally follow a hierarchical scheme where they are ranked according to their access time, storage capacity and cost per bit of capacity. Primary storage generally has the fastest access time, the smallest storage capacity and the highest cost per bit stored. Supplementing primary storage is the secondary storage which includes the on line DASDs (Direct Access Storage Devices) and the off-line storage media. Fig. 3.2 shows the hierarchy pyramid. A faster access time is obtained by moving up the pyramid. A larger storage capacity and a lower cost per bit stored are the results of moving down the pyramid.

3.3.1 Storage efficiency

The optimisation of storage as a resource warrants both program and data to move through the storage as expeditiously as possible. We have storage efficiency measured by space time product of problem usage at each level of storage hierarchy. Flynn defines the cost of storage for a particular program as [Flynn M.J., 1972]

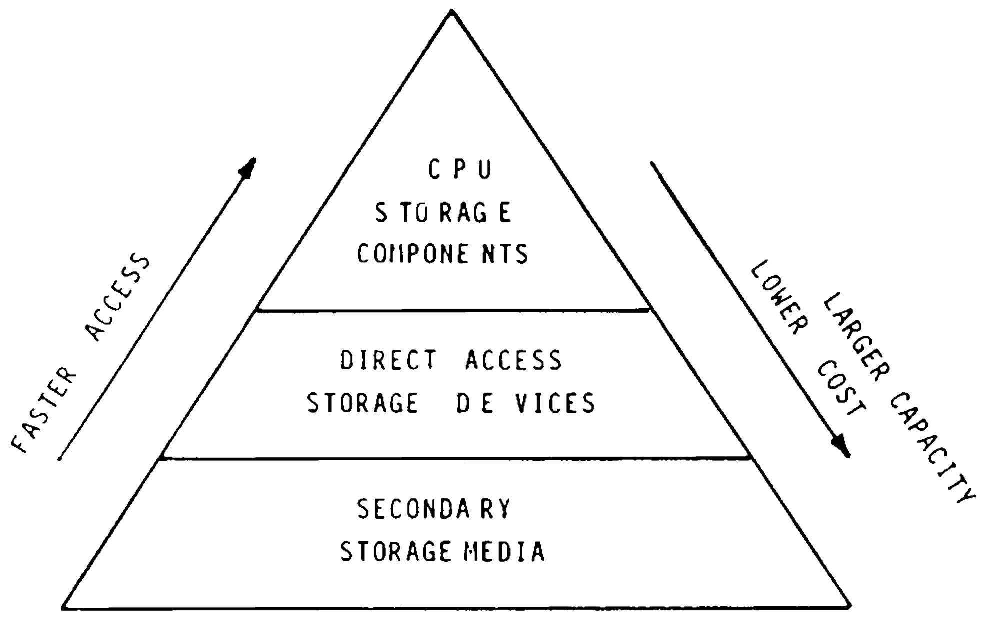


Fig. 3.2 STORAGE HIERARCHY PYRAMID

$$\text{storage cost} = \sum_i c_i s_i t_i$$

where i is the level of storage

c_i is the cost per word at that level

s_i is the average number of words the program used

t_i is the time spent at that level.

A reduction in the components of the storage cost would add to throughput.

3.3.2 Relevance

It is often the case that an embedded system is supported by a database; for eg. information relating to the passengers in a particular flight might form a database. An important attribute of the performance of a database is the disk to memory transfer rate. A new procedure that would speed up disk access time is presented in Chapter 6. This is relevant in conventional database systems as well.

Of late, these conventional systems are replaced by a dedicated machine, the Data base Machine (DBM), tailored for data processing environments and in most cases utilizing parallel processing to support some or all the functions of the database system. An important class of DBMs is the Mutiprocessor Database Machine. In this class a DBM is organised as a set of microprocessors intercommunicating through a shared memory, an interconnection network or both. These DBMs use the shared memory or a separate interconnection structure to interface the system disks where the data base is stored, to the set of microprocessors.

To sum up the discussion, it can be reasonably concluded that the issues identified for investigations, particularly the first three, are relevant in all dedicated systems, where the thrust is not so much on number crunching. The remaining two issues are considered individually in later chapters.

Chapter 4

THE NEW DESIGN

The relevance of block transfer and DMA in multiprocessor embedded systems has been brought out in the previous chapter. A family of controllers suitable for such an environment, can be designed. This chapter describes a typical architecture of the controller that is designed, simulated and tested.

4.1 Introduction

Commercially available DMA controllers normally provide access to the memory of a processor, for peripheral devices. Generally the peripheral device (which can be notified by a subsequent DMA ACK signal) makes the access request. In a multi microprocessor environment sharing of routines between processors is often required. The motivation for the design is the difficulty encountered in transfer of data/routines between processors' local memory. A typical situation might require different modules located at memory blocks that need not be contiguous, to be transferred to another processors' local memory. Earlier controllers provide a pair of channels that simulate memory to memory tranfer. But these channels can be programmed to transfer only one particular block of data. A serious drawback is that access is confined to a single block that contains sequential locations. When the block has been completely transferred, the DMA process ends and the channel becomes idle. In order that the channel be used again, it must be reprogrammed.

The new design takes care of these drawbacks. Non contiguous blocks involved in a transfer can be programmed in a single operation. This makes sharing of data / routines more efficient. It provides direct access amongst processors' local memory. The design can be easily upgraded to support larger word sizes.

4.2 The Architecture

Fig. 4.1 shows a typical architecture that was designed, simulated and tested. It includes the major logic blocks and the internal registers. The timing control block generates the internal timing and external control signals. One set of

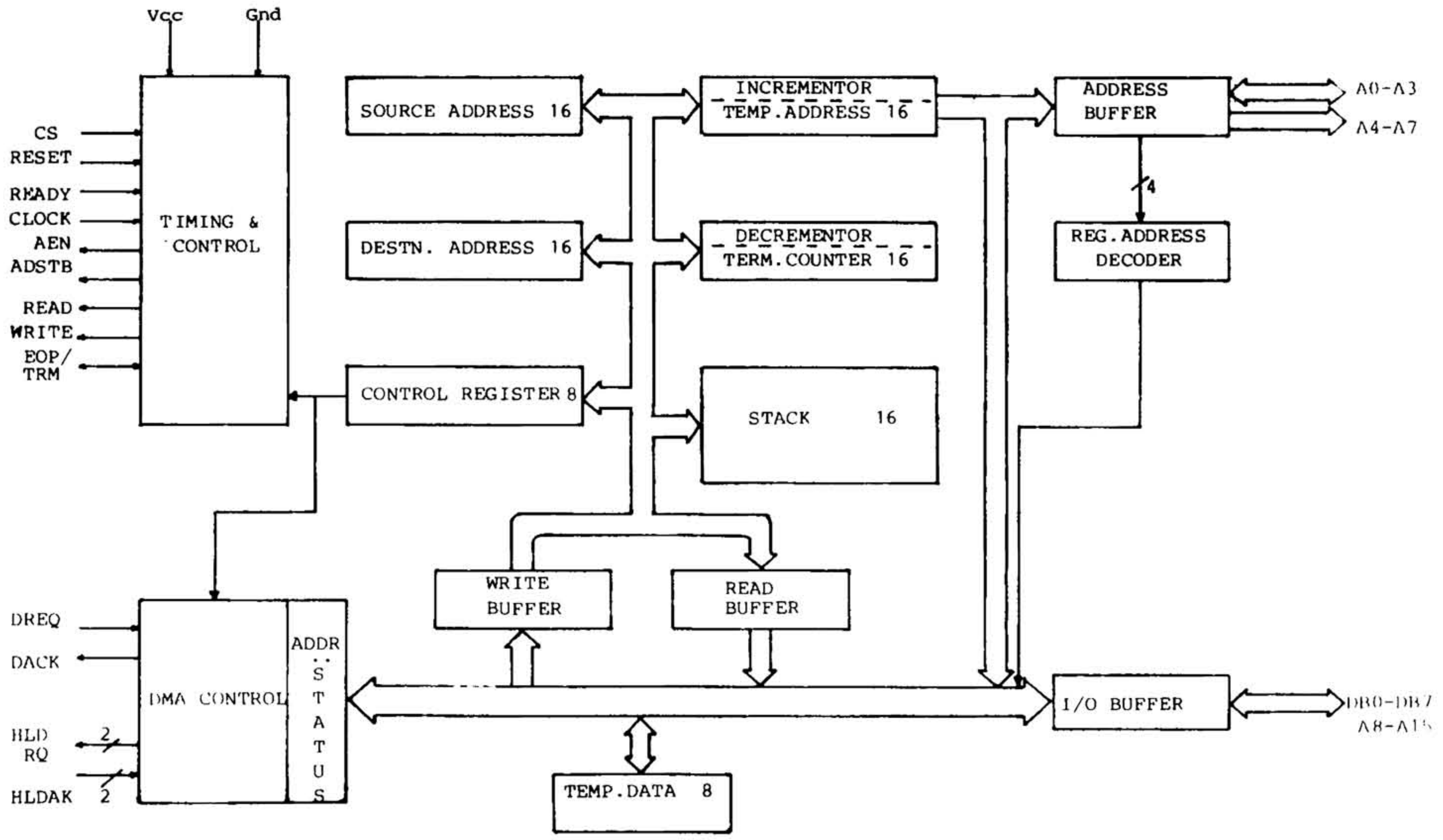


Fig 4.1 Logic block diagram

registers contain the addresses involved and data counts for individual DMA operations. The remaining are control and status registers for initiating and monitoring the operation of the controller.

A stack is incorporated which can have as many as twelve locations of 16 bits in groups of three. This stack can be programmed to store the transfer parameters namely Source Address, Destination Address and number of bytes. In the present design, a maximum of four block transfers can be programmed, each specified by one of the four groups of three locations in the stack. Number of such groups involved is hence indicated by the depth of the stack.

4.2.1 Registers

Control Register   This 8 bit register controls the operation. It is programmed by the microprocessor in the program condition. It stores details regarding direction of data flow, destination device address and whether stack S is needed or not, along with housekeeping namely depth of stack used.

Address/Status Register : This 8 bit register provides access to the controller for processor/device. It stores information regarding the address of the calling processor or device as the case may be. It also denotes certain status conditions as shown in Fig.4.2, which will be useful in the system debugging phase.

Source Address Register : This 16 bit programmable register holds the address of the beginning of the source area in memory.

Destination Address Register : This 16 bit programmable register holds the address of the beginning of the destination area in memory.

Terminal Counter Register : This 16 bit programmable register holds the number of bytes in one block to be transferred. This is decremented by internal logic.

Temporary Address Register : This 16 bit register keeps track of the actual address associated with the source to temporary data register transfer and the temporary data register to destination transfer. This is incremented by internal logic.

Temporary Data Register : This 8 bit register is used for temporary data storage during transfers. Following transfer completion, the last word transferred can be read by the microprocessor in the program condition.

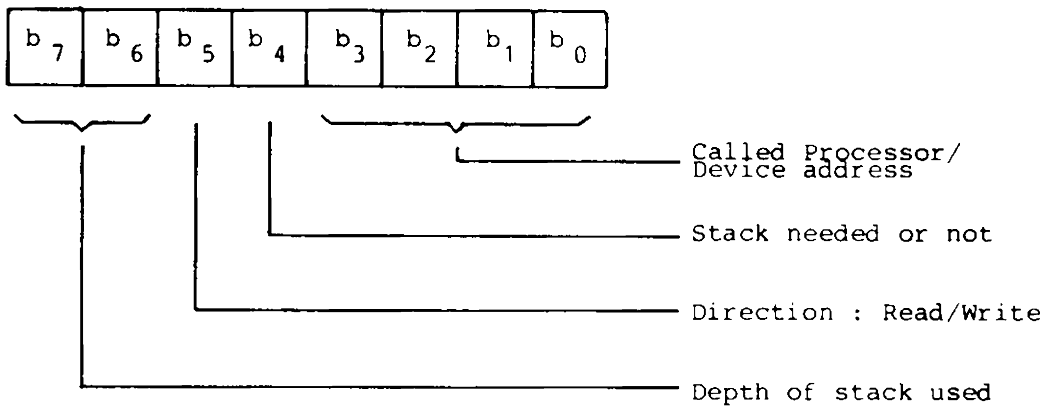
Fig. 4.2 shows the detailed bit configuration of each of the registers described.

4.2.2 Operation

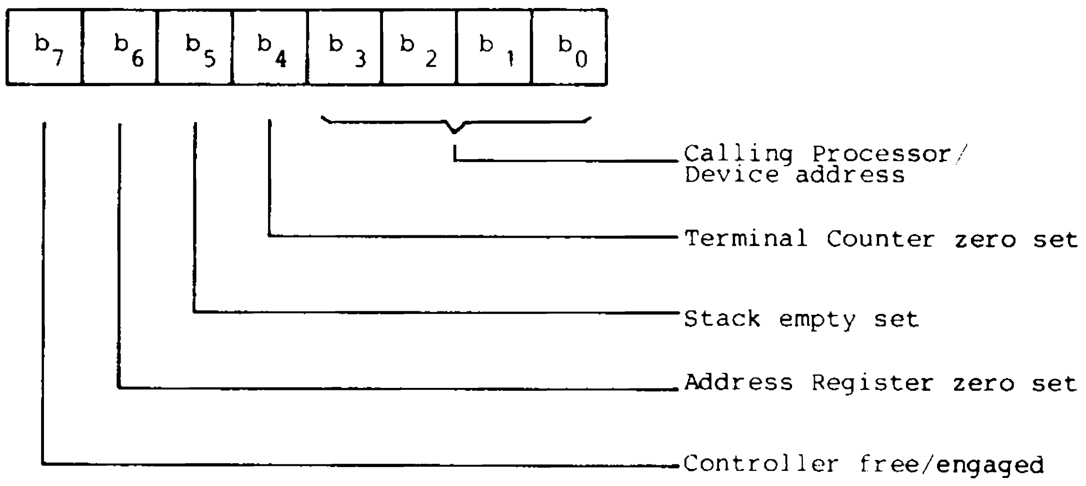
The various logic blocks together provide the necessary control signals in order

* to listen to a calling processor
* to provide access to the registers to the processor
* to raise HOLD signals and R/W signals
* to sense HOLDACK and execute transfer
* to enable all the housekeeping signals like Clear, buffer on and such other control signals.

The controller itself forms an output port for all the processors. The CALLING processor raises a 1 level at the DMARQ terminal after testing the free/engaged bit in the status register. If free, the controller records the calling address in the least significant 4 bits and routes the first byte to the control register. This byte is programmed to indicate the specifications, namely the CALLED processor address in 4 bits, data direction, stack required or not and depth of stack. The CALLING processor signals end of entry by clearing its DMARQ. The controller then executes the transfer. Two modes are permitted depending on whether stack



Control Register



b7
b6
b5
b4
b3
b2
b1
b0
Calling Processor/ Device address
Terminal Counter zero set
Stack empty set
Address Register zero set
Controller free/engaged

Address Register

Fig. 4.2 Register bit configuration

is required or not. In mode 0 which pertains to a single block transfer, stack is not involved and the address and TC registers get directly filled up. The controller then raises HOLDs on the CALLING and CALLED processors, receives HOLDAs from both and raises Read or Write signals.

The internal logic increments the address, decrements the terminal counter and on finding TC zero, generates EOP, relinquishes HOLDs and clears all registers to zero. This is important as far as the address registers are concerned, owing to the fact that certain processor families have locations starting from 0000 in the ROM area, so that no malfunction can accidently write into any memory.

In mode 1, where multilple blocks occur involving the stack, the data to all the registers, Source Address, Destination Address and Terminal Count are fetched from the first group of three locations in the stack and transfer proceeds. The sequence is repeated as many times as mentioned by the depth of the stack.

4.2.3 Terminals

DB0 - DB7	Bidirectional 8 bit data bus

A0 - A7	Output lower address byte for transfer
A0 - A3	Input register address when processor accesses internal registers for programming. Also during system debugging phase, access to the internal registers is important.
A8 - A15	Output higher address byte for transfer via data lines. Uses latch for sending address through DB
DMARQ	Input from processor/device to obtain DMA service
DMAACK	Output to acknowledge recognition of DMARQ DREQ active until corresponding DACK activated
HOLD 1	Outputs request to CALLING processor to relinquish bus lines
HOLD 2	Notifies CALLED processor for bus control
HOLD ACK 1	Inputs information that controller can acquire control of the bus.
HOLD ACK 2	From CALLED processor, inputs consent for bus control
AEN	Enables 8 bit latch containing upper 8 address bits onto system address bus
ADSTB	Strobes the upper address byte into an external latch

EOP	output when TC reaches zero, terminates DMA service
READ	Input signal to read internal registers. Output signal to access data from a selected location.
WRITE	Input signal to load information into the controller. Output signal to load data into the selected location.

Over and above these, there are the general IC control terminals like CS, READY, RESET, Vcc and GND.

Data can be transferred directly from one memory location to another, specified by the source and destination addresses, with READ and WRITE active at the same time.

Basically the DMAACK signal is to notify peripherals when it has been granted a DMA cycle. This happens when the controller has acquired control of the bus on receipt of HOLDACK from processor. In a processor to processor case, in the event of the controller not being free to entertain a REQ, it just returns a DACK to the CALLING processor.

For getting service from the controller, certain amount of software is needed for each processor. Fig.4.3 shows the flowchart of the software. The design principle enunciated may be adapted to specific requirements in large systems as well. Only the register size need be enlarged in most cases. Once the controller is initiated, a series of transfers can go without any CPU intervention.

Appendix A1 gives details of a switching unit which can be used for sytems with numerous processors.

4.3 Simulation of the design

Chip and system designers use hardware and software tools to precisely list elapsed time for every event. Evaluation by mainframes, making use of the real time clock, is a standard practice. However, obtaining access to a powerful host and expending elaborate and exhaustive efforts for a single chip design is uneconomical. Hence the simpler, but equally effective PC based simulation is undertaken with C as the simulation vehicle.

A program model of the design is developed and the time involved is estimated for the sequence of ordered events.

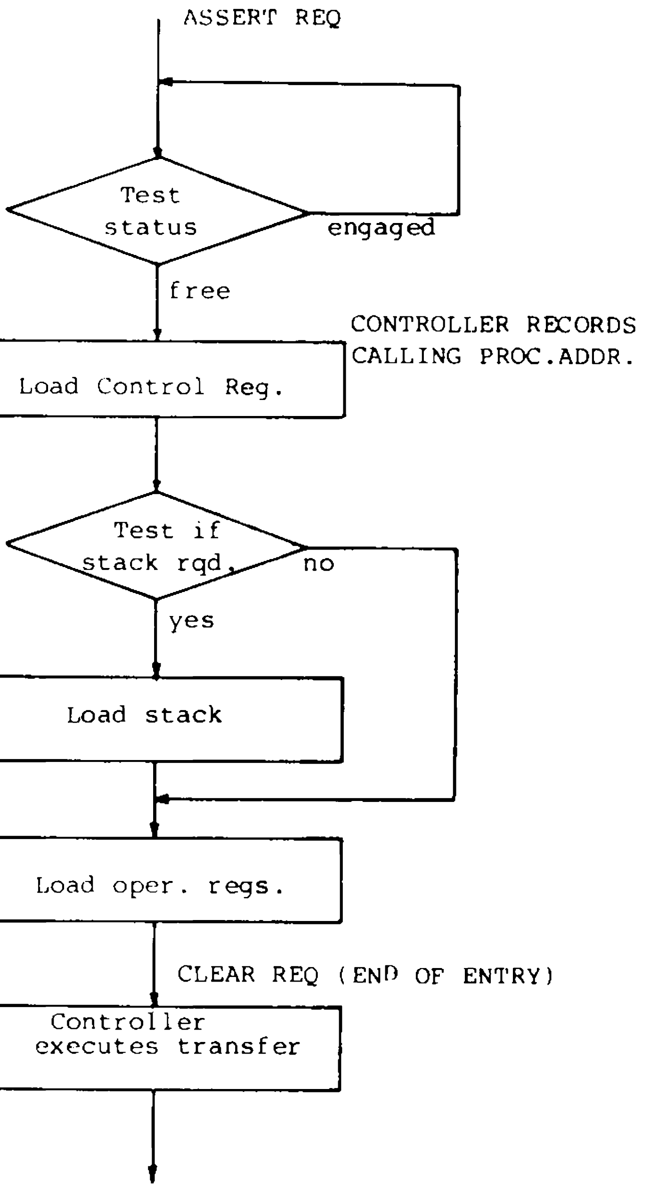
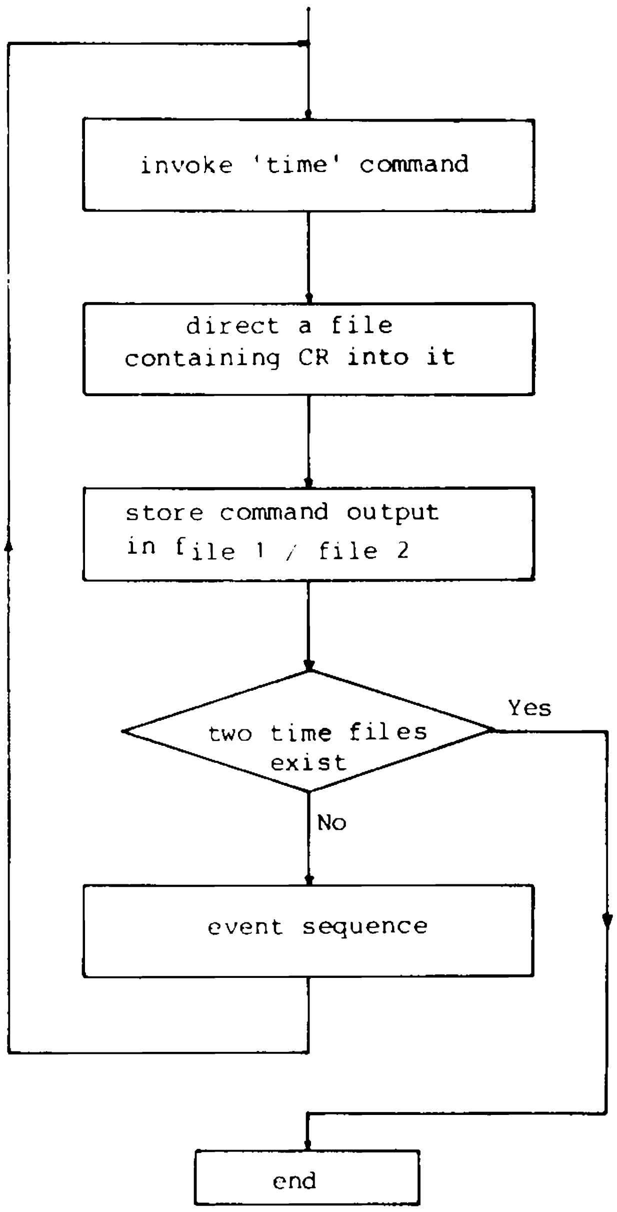


Fig.4.3 Getting service from controller

Since the state of entities remain constant between events, there is no need to account for this inactive time in any program model.

The system command 'time' is made use of. The routine waits for data or CR for exit after execution. If used directly in time estimation, this wait can cause unreliability on the result, to the extent that user action need not be consistent. This is obviated by the following procedure. A file is created with just a CR in it and is directed to the time command. The output is now directed to another file named 'timefile 1'. The time involved in one event sequence is estimated by first causing timefile 1 to be created prior to the sequence. Then subsequent to the sequence, 'timefile 2' is created along the same lines. Fig. 4.4 and 4.5 represent the flow chart. The time involved is now extracted from the two time strings, by converting the components into their respective integer equivalents and doing necessary arithmatic.

The various events that occur in a typical transfer are simulated in the program as per the flowchart shown in fig. 4.6. Transfers have been carried out involving one, two and three blocks of data for different values of byte count.



invoke 'time' command
direct a file containing CR into it
store command output in file 1 / file 2
two time files exist
Yes
No
event sequence
end

Fig 4.4 Create two time files

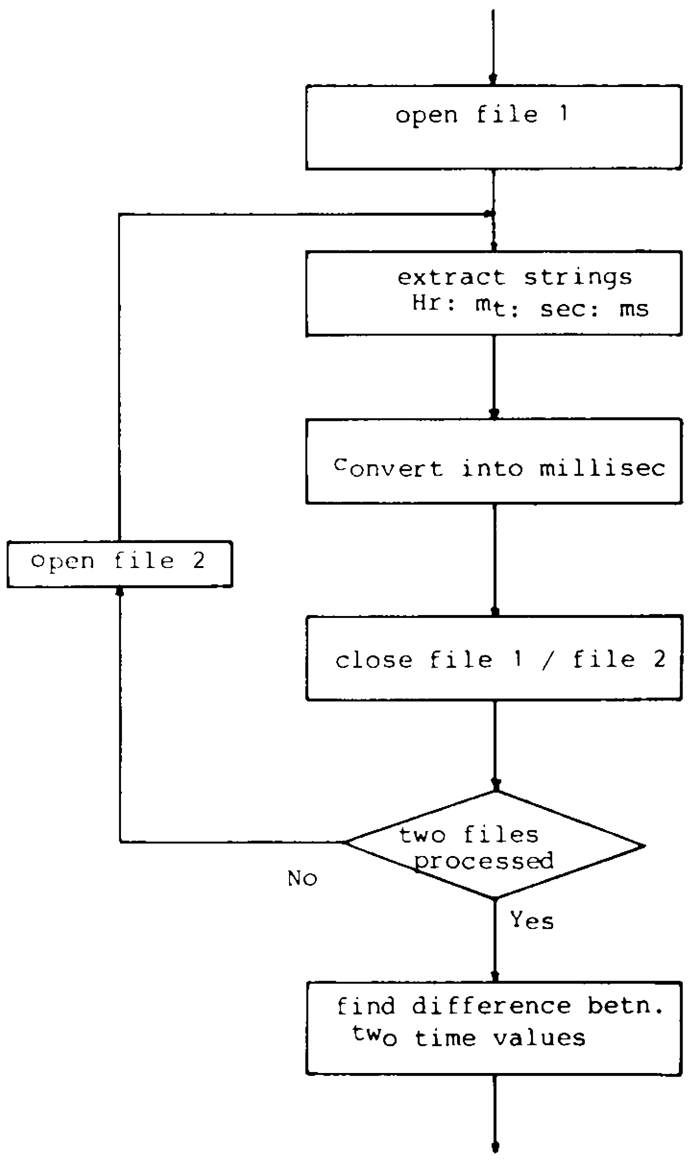


Fig 4.5 Time involved in event sequence

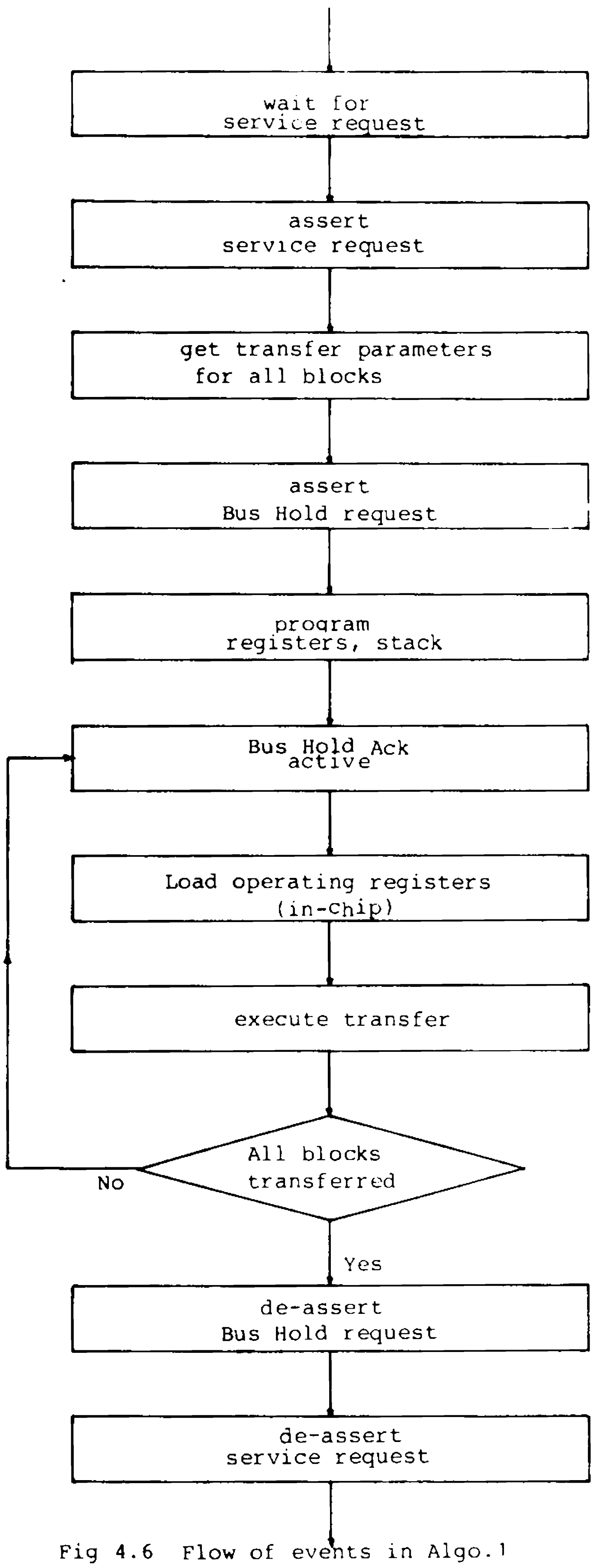


Fig 4.6 Flow of events in Algo.1

Time in milliseconds have been estimated. The results are shown in Table 4.1.

A second set of values are estimated for the transfer sequence pertaining to popular DMA controller models presently in use. This is represented by the flow chart in fig. 4.7. In a typical case where successive block transfers require reprogramming, a predefined sequence is followed in the program, so that any variation in actual programming operations is precluded.

4.3.1 Discussion of the simulation results

A close study of Table 4.1 shows that there is a considerable time gain by the proposed transfer technique. The results are classified for 1, 2 and 3 blocks. Consider a transfer of 375 bytes. In one bulk, the time taken is 378 ms by Algo.1 and 417 ms by Algo.2. If these bytes are constituted by non-sequential blocks, then the time taken by Algo.1 is 603 ms if they are in 2 locations and 791 ms if in 3 locations. This time is still of the order of 100ms less than that by Algo.2. The advantage can really be felt in multiple blocks being transferred.

No. of blocks	No. of bytes	No. of bytes in each block	Time estimated in mS Algo.1	Algo.2
1	75	75	225	253
	150	150	264	302
	225	225	307	341
	300	300	351	390
	375	375	378	417
2	225	75 + 150	509	592
	375	150 + 225	603	672
	525	225 + 300	675	750
	675	300 + 375	756	843
	825	375 + 450	829	907
3	375	75 + 125 + 175	791	902
	600	150 + 200 + 250	928	1038
	825	225 + 275 + 325	1033	1176
	1050	300 + 350 + 400	1175	1291
	1275	375 + 425 + 475	1269	1401

Table 4.1 Performance comparison

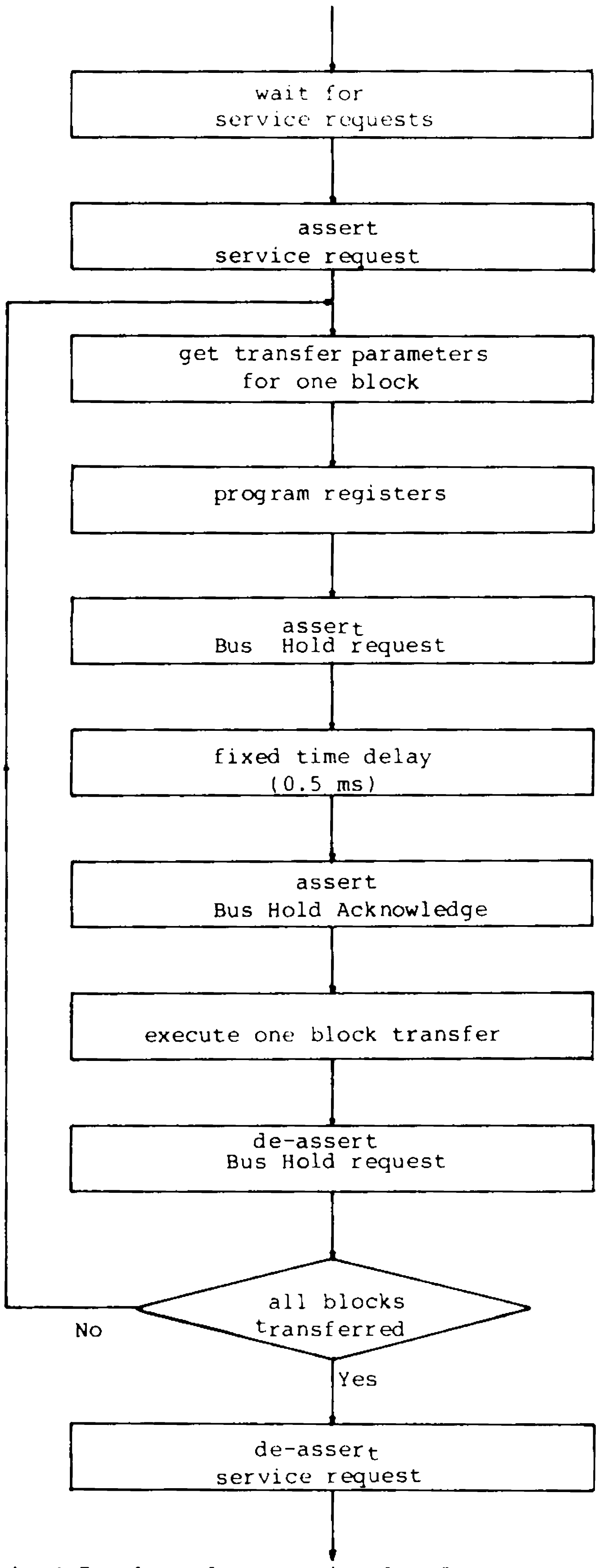

wait for service requests
assert service request
get transfer parameters for one block
program registers
assert Bus Hold request
fixed time delay (0.5 ms)
assert Bus Hold Acknowledge
execute one block transfer
de-assert Bus Hold request
all blocks transferred
No
Yes
de-assert service request

Fig 4.7 Flow of events in Algo.2

Fig. 4.8 shows the advantages of the design graphically.

The difference in time for Algo.1 and Algo.2 are found to increase with number of blocks. In a sample case of the 375 bytes transfer, the difference is seen to be highest when there are three blocks. Table 4.2 shows the time gain for the three cases. Fig. 4.9 is the variation of time gain over an increase in the number of blocks.

4.4 Comparison with other similar controllers

The design concept now enunciated was first proposed in its basic form in 1982.[in a special issue of the EUROMICRO Journal]. (Please see Appendix A2 for a Reprint). Much refinement and tuning has been done on the basic design. The principle of distinct registers for source and destination addresses has since been incorporated in the Intel product 82380, as given out in the advance information columns.[Intel Manual,1989]. Eventhough this particular controller has more powerful features and several other functions, the basic DMA transfer function cannot be regarded as efficient as our design, owing to the following limitations :

i.The 'buffer chaining process' though permits specifying a list of buffer tansfers, it requires reprogramming

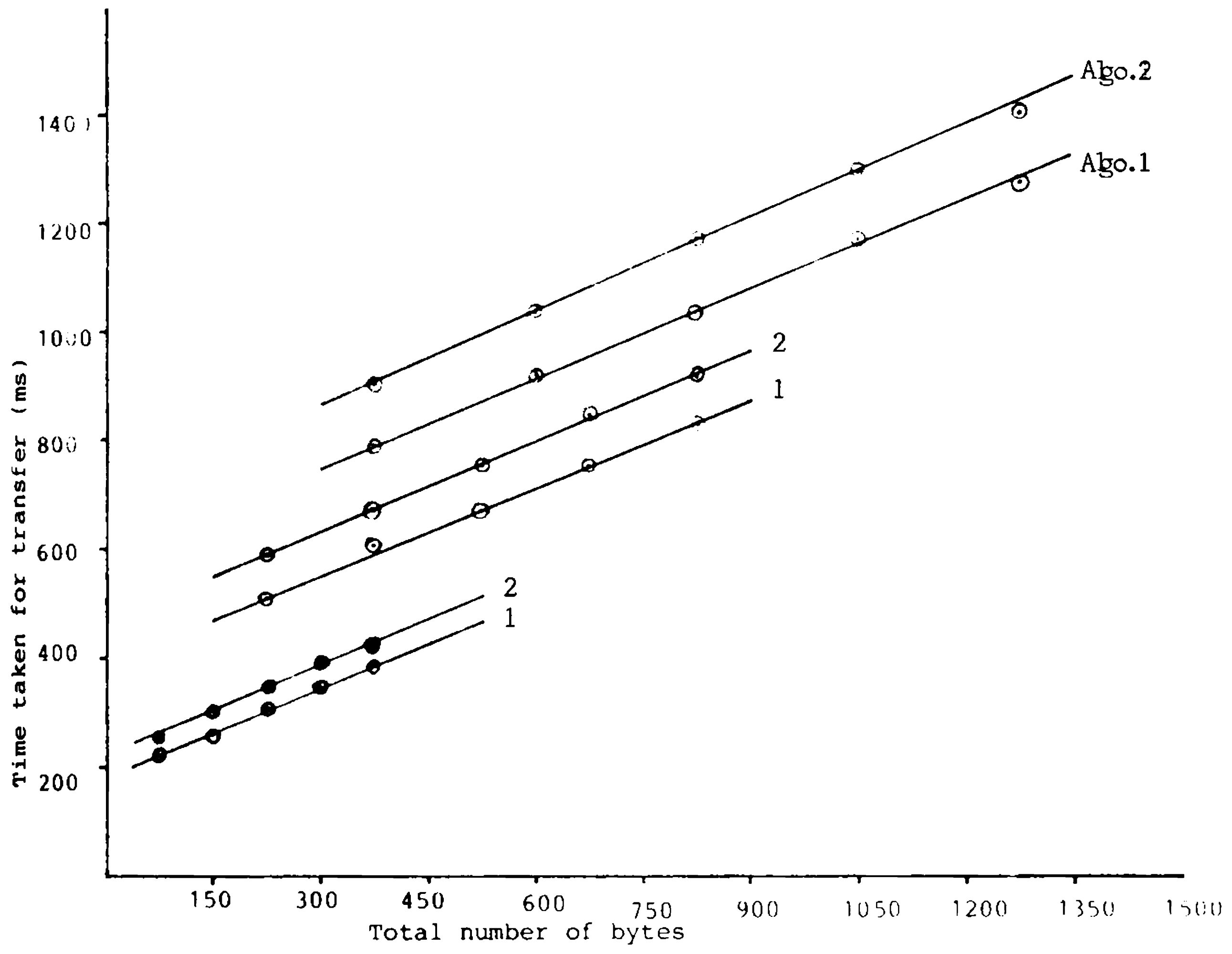


Fig.4.8 Performance comparison

No. of bytes	No. of blocks	Time in mS Algo.1	Algo.2	Time gain
375	1	378	417	39
	2	603	672	69
	3	791	902	111

Table 4.2 Time gain for different number of blocks

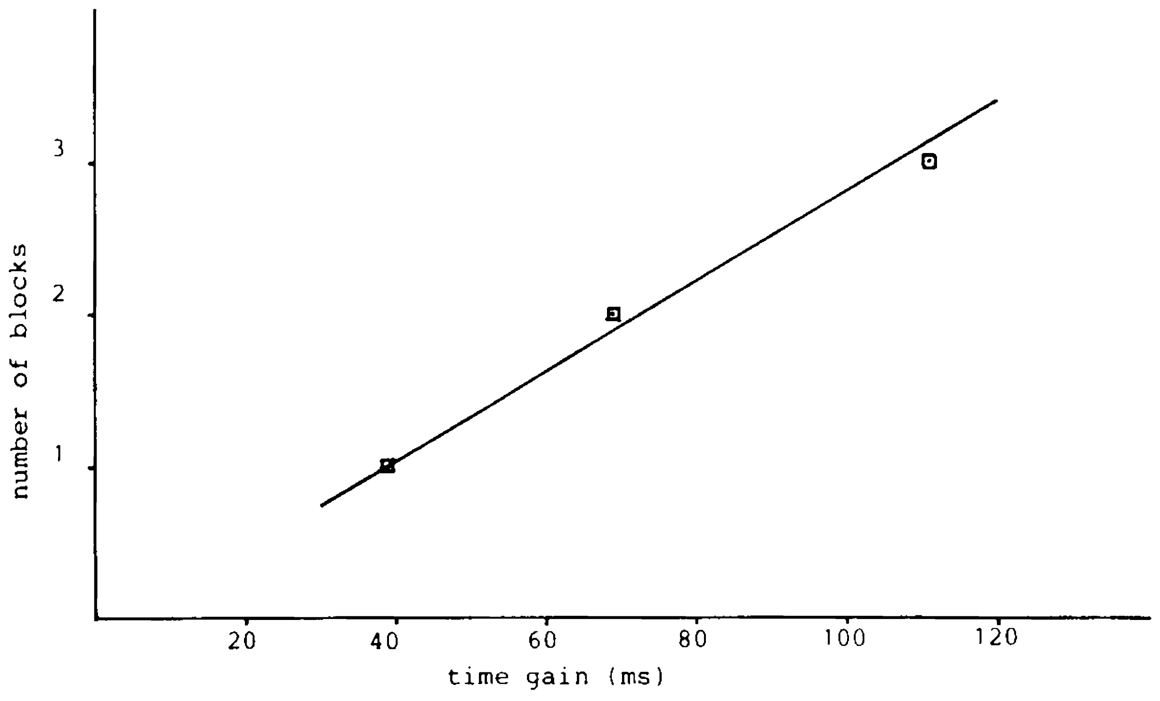


Fig.4.9 Time gain variation

controller through interrupt routines.

ii.Interrupt signals are required.

The specific improvement over earlier Intel models is that the reprogramming is done before the current transfer is complete.

The 68 pin 82258, the advanced DMA coprocessor with its extensive features and data manipulation capabilities, may not be required for applications envisaged in this work. However, it makes use of the concept of distinct source and destination pointers. Data transfer operation is specified by command blocks which reside in the coprocessor memory. Only the address of this command block is specified in the on-chip command pointer. The command block with all its parameters is to be loaded into internal channel registers before a transfer is initiated. Several such command blocks can be defined, which along with destination list chaining of data, permits gathering and scattering of data blocks. This phenomenon which tantamounts to a transfer involving non contiguous blocks, can be reasonably assumed to be an adaptation of the concepts suggested in our design. Nevertheless, its drawback is that each set of transfer parameters is to be loaded into internal registers from the memory resident command block.

Another product with powerful features notified by Intel is the 82389 Message Passing Coprocessor.[Intel Manual 1988]. It is basically a bus interface controller designed to offload the host CPU for interprocessor communication on the Parallel System Bus network. Its primary function is to support the communication protocol standard defined for the PSB (IEEE 1296). The device supports both physical and data link support. The data link includes packetisation after receiving data from the local interface, bus arbitration, burst transfer and error detection without CPU intervention. But even with all its complexities, an external DMA controller is required to support solicited message operations whereby actual data is transferred from one agent to another over the PSB bus. (Any device with an interface to the PSB bus is termed an agent).

Quite complex in its structure, the MPC is a 149 pin grid array package which requires supply voltage at 6 pins and ground connections at another 12. It provides DMA control signals like ODREQ and ODACK. The former is asserted by the MPC to enable DMA transfer of data to it. The DMAC responds by performing DMA transfers to the MPC for transfer to the receiving agent. The ODACK is an input signal asserted by the DMAC in response to the ODREQ.

The local bus of the MPC is used to interface to a host processor. The local bus interface can be categorised into three sub interfaces : register, reference and DMA. The DMA interface supports data transfer between local memory and the MPC.

The above discussion points to the fact that the controller designed is more suitable for applications that have been mentioned.

Chapter 5

TYPICAL SCENARIO

5.1 Introduction

This chapter presents a system configured as a collection of autonomous processing elements, co-operating to achieve a common goal. The communication overhead can be quite considerable in such multiple processor environments. Various schemes have been implemented in typical architectures. For eg. the Connection Machine supports two forms of communication within the processors, namely the Router and the NEWS grid.[Hillis, 1985]. In the Hypercube architecture, the processors are located at the vertices of the n-cube and the interconnections are the cube edges.[Welty L. et al, 1985].

As mentioned earlier, in a totally dedicated system it may not be economical and efficient to go in for such techniques. Though transputers have come to be used in our country recently, the number of systems using them are limited.

Moreover, since the thrust is not so much on number crunching, but on the number of operations, a different approach is suggested. In the embedded system environment, it can be observed that all processes need not get the same consideration from the system. It is essential therefore that some classification and identification of processes be done first, followed by actual implementation. The importancre of dedicated system design and the classification is described in the succeeding sections.

5.2 Application scenario

Consider a simplified flight control system. Let there be eight variables which include

Airspeed

Aircraft heading

Altitude

Angle of bank

Attitude

Throttle

Vertical speed

Weight

Appendix B shows details of these variables along with associated controls and panel instrumentation.

If four processors are provided for the control system and the variables are to be monitored and controlled every 10 ft, program modules should execute at 1 nsec. (assuming the airspeed works out to about 200 ft/sec). Here measurements interrupt the system every 1/20 of a second, some modules are common to different processors, some calculations like trajectory and banking need subroutines. Routines once set can go on for a long time till the position alters. When the aircraft is in a state of equilibrium, that is to say it continues to move forward at the same uniform rate of speed, the only critical parameter for navigation is the position.(equilibrium refers to steady motion and not to a state of rest).

5.3 System

The system suggested has been named the APT system, based on its feature of transportable processes.(A Process Transportable System). The main memory is partitioned into well defined blocks, which would enable processes to communicate and co-operate in order to achieve the application objective.

It adopts a conglomeration of shared and local memory with a view to obviating the complex communication network.

Within the APT system, a process as defined by Lister, [Lister A.M., 1979] is classified into one of the four denoted by 0, 1, 2 and 3. Class 0 tasks are non-splittable; once they start execution they run to completion, unless interrupted. An interrupt would only suspend it. The task is resumed after the higher priority task is run, by the same processor. No switching is envisaged.

Class 1 tasks have a hierarchical structure. This permits tasks to create several subtasks that can be executed in parallel asynchronously. Where the subtasks are mutually dependent, synchronism is required. Such cases are classified under Class 2. Class 3 is made up largely of functional subroutines which lie in specific areas of system memory. The main memory blocks are brought out to be in harmony with the different classes. Fig.5.1 is the pictorial representation of the different classes.

5.4 Memory partitioned

The APT System memory forms a common resource (CORE) to all

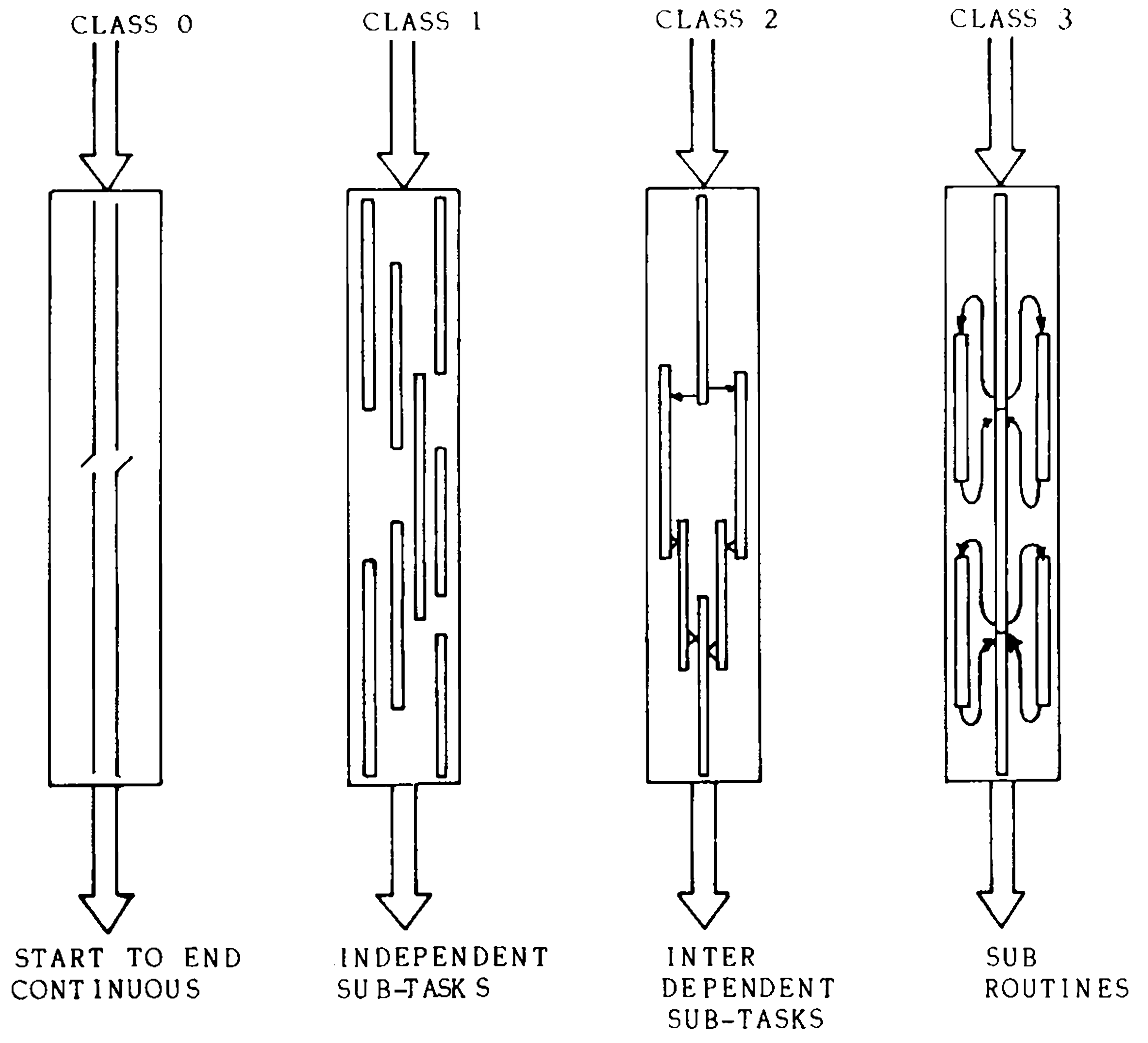


Fig. 5.1 Process / Task classes

the constituent processors. Within the core, each processor has a free hold property (FHP), which is further partitioned into functional blocks. One such block serves as the concerned processor's status indicator (PSI) which holds the saved contents of its stack pointer, pointing towards the stack, where the volatile environment [Dhamke M.,1982] of a class 0 task is available, if at all any such task remains suspended. The PSI block also holds the status of the processor - whether it is running a process or not, the class of process being run etc. This block is protected to the extent that it is read only as far as the other processors are concerned.

One of the processor is assigned the role of Master. The Master will monitor the PSI block of each processor and serve also as system arbiter. Once the Master, which is dedicated to its role, allots a task to a processor, its execution could be from another block of FHP to which the task is transferred. This block is referred to as the working area (WA) block. The WA block is accessible to the other processors as well, so that upon switching, the new executor need scan only this pre-defined area in order to pursue the task. All subtasks of a class 1 task are bound to lie in the WA block, so that the Master can conveniently allocate processors for their asynchronous execution.

A class 2 task involves another FHP block, where intermediate results necessary for the fulfilment of another subtask, run on a different processor, are available. This block which serves as a link block, occupies a dedicated portion of the core which alone need be referred to by the other processors, for obtaining results leading to the culmination of its subtask.

Yet another block would be a free space at the disposal of the master, which may be used by itself or assigned to the constituent processors. The subroutines, which form class 3 tasks also occupy part of this space. Moreover, as the switching of processes is intended to hasten throughput, it is likely to demand high levels of I/O activity to bring in more tasks or more data. The free block provides for this demand as well.

Fig. 5.2 gives details of the various blocks.

The PSI block of the Master is distinct, in so far as it contains a set condition indicating its supervisory role and a pointer towards the core area where the

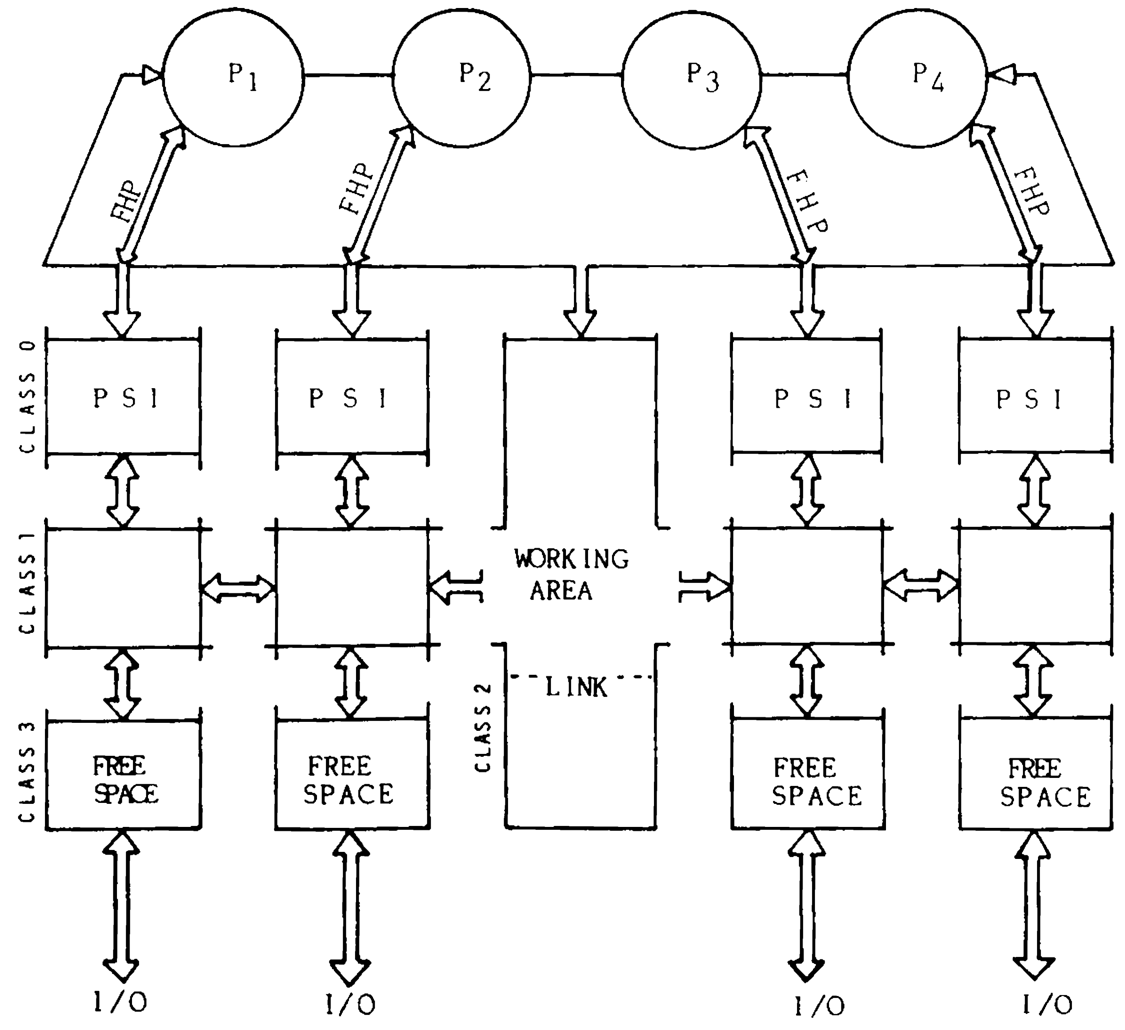


Fig. 5.2 Partitioned Memory showing location of task classes

scheduler/despatcher is located. Any of the slaves desirous of service from the Master can request and obtain it. The Master can also shift its supervisory role to a slave by transmitting its PSI block to the new Master.

5.5 Implementation.

The role of allocating access to the appropriate memory blocks to the slave processors is exclusively handled by the Master. Once the Master decides to grant such access to any one of the slaves, it will output the address code for the particular slave. This address code will link the Master Read and Slave R/W lines to the memory block via a multiplexor. At the same time the demultiplexor unit opens the appropriate blocks, to link the internal address and the data buses of the slave to the external address and data buses of the memory block. All block access requests can be generated by the Master; the Slaves need not generate such requests. This avoids the problem of concatenation of requests.

All I/O activity shall be the exclusive responsibility of the Master, so that any interrupt from a device would be dircted only towards it. The device interfaces are intelligent enough to perform the required operations and then inform the Master that the operation has been completed. Each time an interrupt occurs, the OS can trap it, set a flag indicating that an I/O event is in the offing.

5.6 Discussion

Eventually, a process in the context of the APT system, is seen as a program that occupies a relatively small directly addressable space. This is always preferable. Very large linear address spaces lead to poorly structured and difficult to debug programs as well as possible problems in protection.[Lorin H., 1982]. It would be convenient to have all vital information about each task in one place or rather a table, for easy reference. Each entry in the table can contain the task identification, its status, the entry point of the task and a save area for its stack pointer. This last entry would help a processor, taking over a new assignment, get the correct start.

The partitioned memory scheme presented requires transfer operations among the different blocks. This memory to memory transfer can be advantageously effected with the help of the controller described in an earlier chapter. The Master while performing the allocation, can load the appropriate registers in the controller, which will then manoeuvre the transfer.

Service of the controller is desired by the Master through the request terminal of the controller after testing the status bit. Each processor holds the transfer parameters in the locations assigned. The controller when free, can acquire these bytes by accessing these locations directly to fill up the registers.

In the parallel bus scheme, the BUS GNT signal will indicate the availability of the bus. The controller then gets the right to initiate one or more bus transfers; it can use the address, data and control bus lines. The BUS BUSY signal may be tied to the ENGAGED condition of the controller so that the Bus Available signal will be active only when the controller relinquishes Holds.

5.7 Theoretical analysis

Based on a model for resource usage, an analysis is done with a view to optimising throughput. Resources in this case, include memory, processors, tables and utility drives. For the sake of uniformity in this analysis, all these are grouped into one class namely 'memory'. Every process uses some 'memory' for a duration depending on the task it is performing. For the system with 4 processors, the picture of memory usage at the end of one task would look like a time matrix given by

$$\hat{\tau} = \begin{vmatrix} \tau_{11} & 0 & 0 & \tau_{14} \\ \tau_{21} & \tau_{22} & 0 & 0 \\ 0 & \tau_{32} & \tau_{33} & 0 \\ \tau_{41} & 0 & 0 & \tau_{44} \end{vmatrix} \quad \rightarrow \{1\}$$

where τ_{ii} refers to the case when a processor uses its own resource, while $j \neq i$ denotes use of a shared memory block or a resource earmarked for another. τ_{ij} includes communication as well.

τ_{ii} is a random variable, say Poisson distributed, [Musa, 1987] such that

$$p(\tau) = \frac{e^{-\mu\tau}}{\tau!}$$

where μ is the average value.

Since no two process can use the same resource at the same time, either a queue or a wait state W is generated. As a result, by the time a job is complete, several time matrices are to be summed up along with W matrices. Ultimately the zeros in the time matrix get filled up and will lead to a general τ matrix given by

$$\tau = \begin{vmatrix} \tau_{i1} & \cdots & \tau_{in} \\ . & & . \\ \tau_{n1} & \cdots & \tau_{nn} \end{vmatrix} \rightarrow \quad \{2\}$$

where τ stands for the total calender time taken by the complete job.

An n processor, n resource case is assumed, but it is not necessary that the two be equal in number. But certain operations can be performed easier, if τ is made square by padding with zeros.

It is also possible to decouple the number of times a resource is called and the time it is occupied. In that case {1} should be split into two vector equations. Adopting the same procedure, the total time occupied for each resource can be calculated and $\hat{\tau}$ can be formulated.

This model is only an extension of the model presented by Iannino.[Iannino, 1990].

$$x_Y = \theta_Y \tau + \mu_Y \quad \mu(\tau)$$

It may be noted that the leading diagonal of τ should be non zero for normal operations. In exceptional cases, like a network, some of the elements of the diagonal may almost be zero. It can also be seen that if τ is symmetric, the system will be most efficient. A bias in τ will be very clearly inefficient.

5.7.1 Resource usage model for the system

Let the FHPs and shared memory be identified separately by respective indices. The time for which each is used is denoted by τ_{ij}

where i denotes processor number

j denotes memory block of the jth processor.

Thus after any instant T, for a small number of processors

$$\tau = \begin{vmatrix} \tau_{11} & \tau_{12} & \cdots & \tau_{1n} \\ \tau_{21} & \tau_{22} & \cdots & \tau_{2n} \\ \cdot & \cdot & & \cdot \\ \tau_{n1} & \tau_{n2} & \cdots & \tau_{nn} \end{vmatrix}$$

while, at any time instant if $\tau_{ij} \neq 0$ then $\tau_{ji} \triangleq 0$

because no two processors have access to the same memory.

If the system runs for a total time T, the best τ (normalised) will be a constant

$$\tau_{ij} = \tau_{kl} \quad \text{for all } ij,\ kl$$

For a particular set of tasks, the total time taken for execution Te can be written as

$$T_{ei} = [\tau][P] \rightarrow \{3\}$$

where P is a vector. [P1......Pn] represent the processor factor ie., which processor calls which memory

{3} is an ideal situation where no failures or wait states occur. To include such an eventuality, a correction term w is to be added to {3}. Then

$$T_e = [\tau][P] + [W] \rightarrow \{4\}$$

where W is a random time variable matrix representing waits and failures. The distribution of elements of W can be found for standard tasks.

The τ matrix is a matrix made up of sum of Poisson matrices.

Order of Te of course varies with each job, but the ideal practice is to divide the job into N number of tasks such that N is the number of processors.

For maximising the throughput, Te (each element or some linear combination or the largest of Te) must be minimised.

A. Trivial Solution :

Te constant would be a trivial solution for $w = 0$ and p = constant.

B. Non trivial solution :

Since each of the component is job dependent, τ, P and W must be interrelated.

Thus if for one task, a particular processor Pr takes a time τ_{rj} and has to wait Wr for the availability, processors can be switched and Pq might take over the job, such that, switch time + Wp < Wr.

This can only be done if the class of jobs are known in the form of a table and the Master divides the task optimally.

Let $W = Q \hat{\tau} Q^{-1}$

where Q is not necessarily sparse.

Using techniques of OR, particularly scheduling it is possible to find a Q and a τ which minimises Te.

This then calls for a W which is not entirely random and independent of tasks.

C. Optimisation approach :

Using methods of variational Calculus equation {4} can be differentiated with respect to time and the minimum value of Te obtained by, say the standard Lagrangian form, given by

$$0 = [\dot{\tau}][P] + \dot{W} + C \rightarrow \{5\}$$

where C is a constant matrix.

Solution to {5} involves some transcendental equations and only guarantees an extrema. Differentiating {5} again and making the RHS non negative the matrix equation resembles the form of a Riccati equation.

It must however be remembered that τ itself is a sum of a large number of matrices, and in the limit can be equated to some

$$\underline{\tau} = \int[\tau]dx$$

Thus on a good approximation

$$\underline{\dot{\tau}} = -\lambda\dot{W} + C' \rightarrow \{6\}$$

ensures non negativeness

where $\underline{\tau}$ is the result of differentiating an integrated quantity ie., averaging out the incremental usage time.

and the solution to {5} can be written as

$$[\tau] = -\lambda W + C' \rightarrow \{7\}$$

$C' \rightarrow$ integration constant

$\lambda \rightarrow$ a constant, say Lagrangian

or $\quad I = -\lambda\ \tau^{-1} W + C'' \rightarrow \{8\}$

where C'' is another integration constant

I is a unit matrix

or in other words

$$\tau^{-1} W = \frac{C'' - I}{\lambda} \rightarrow \{9\}$$

That is average time of usage of a resource should match the average wait time for that resource and should be constant for all resources.

In practice, this may prove quite difficult to achieve without the help of a very powerful OS and expensive hardware. But as can be read from equations {4} to {9} it is fairly efficient and can be implemented in APT, since it ensures almost temporal and spatial task distribution.

Note : Both τ and **W** being result of sums of random variables, they almost always possess an inverse. If on the other hand, if only a set of resources are always used, neglecting the others, or if a row or column of τ is 0 or constant or non unique, existence of $\tau^{-1}$is not guaranteed and there cannot be a solution for {8}.

Chapter 6

THE DATA BASE COMPONENT

6.1 Introduction

Advanced weapon systems are becoming increasingly complex and rely upon computers and embedded systems to operate. For eg. the surface mobile weapon systems used by the army, rely on an embedded navigation system to provide operational direction. It incorporates a remote operating console which requires data bases and a high level of data exchange.

Other similar embedded systems also are found to have a data base component, making use of disks. With the availability of large main memories of the order of gigabytes, it has become possible to have main memory data base systems. But the problems associated with the main memory being volatile still persist.

One of the characteristic features of the disc resident data base is that the amount of information that can be accessed at one time from the disc is a block. Gupta observes that the two measures of performance of disc resident data structures are

i. the number of disc accesses that are required for common file operations (eg. search)

ii. the disc storage utilisatiion.[Gupta G.K., 1988].

It has also been observed that the performance of Data Base Machines that make use of querry processors, would be limited by their disc to memory transfer rates.[Boral H. et al, 1983].

6.2 Schemes available

This section reviews the various techniques that have been used in the past to speed up disc access time. One class of approach is to minimise mechanical delays. The IBM 3380 makes use of a cached control unit where data can be prefetched from the discs into the control unit buffer, so that a read request will not involve disk access if the data is already in the cache. However if the access pattern (ratio of reads to writes) is not appropriate, it not only does not help, but it could also hurt the average access time. This is because a cache miss may be more expensive than a native access (without cache) due to control unit overheads. A write will cause a disk access whether the data is found in the cache or not, where it need to be updated.

Another approach is the disk scheduling technique as originally described in [Denning P., 1967]. Much work has been done in trying to find scheduling policies which minimise seek time. But in order to apply a seek scheduling algorithm there must be a queue on which the algorithm may operate. It has been observed that in reasonably well tuned systems, the average length of the queue is only 0.3 which is inappropriate for a scheduling algorithm to operate.

Several propositions have explored the concept of track interleaving to achieve parallel reads. But this will require special expensive hardware. Further the interleaved bytes are to be processed to obtain a serial data stream.

6.3 Evaluating service time

The basic service time consists of seek, latency and RPS (Rotational Positioning Sensing) miss. Seek delay is the time required to position the access mechanism at the data track, rotational latency is the time required for access of the correct data or track area. It can range from 0 to a full rotation of the disk. If the channel is not free at the time device is ready, then there is an RPS miss and that causes the device to make a full rotation before it can signal again. It may be reasonably assumed that the average seek time does not vary considerably with load, and that average latency is one half of a device rotation period. We hold that the RPS miss delay is the principal load dependent component of the basic service time. This chapter presents a scheme attempting to make the channel free as and when the device is ready and hence to minimise RPS miss, thereby reducing the number of disc accesses.

6.4 Methodology

In conventional DBMs, there is the data communication subsystem, which is the terminal handler that receives user requests, invokes application program and delivers the results that it receives from the database transaction manager. An operating system exists to run these software components. The method banks on manipulating the size of the block that is accessed from the disc.

In this approach, disc sectors are grouped to form segments, the smallest made up of two sectors and called the Base Segment (BS). This segment size is programmable to the extent that one out of the several Exponential Segments (ES) can be selected. These are made up of 4,8,16... sectors as a trade off, forming respectively ES2, ES3, ES4... segments. Depending on the situation, the OS selects either BS or one of the ESs to form a single block of storage. Fig. 6.1 presents the flowchart for the related software.

The space allocation is handled by software with the aid of a segment availability table (SAT), which specifies the track number and starting sector number of every base segment -BS address- indirectly; in the sense that a free BS will be represented by a 0 and a used up BS by a 1. The physical positions of the 0s and 1s in the SAT, read on a dedicated area of memory, will be manipulated by software to determine the corresponding BS address. The address of free BSs will be stacked on a memory block, the size of which can be limited to hold say 256 bytes, so that at any given time 128 BS addresses are available if free. If number of free BSs is less, the stack size is also small, thereby reducing memory requirement. Fig.6.2 shows the flowchart for creating the BS stack.

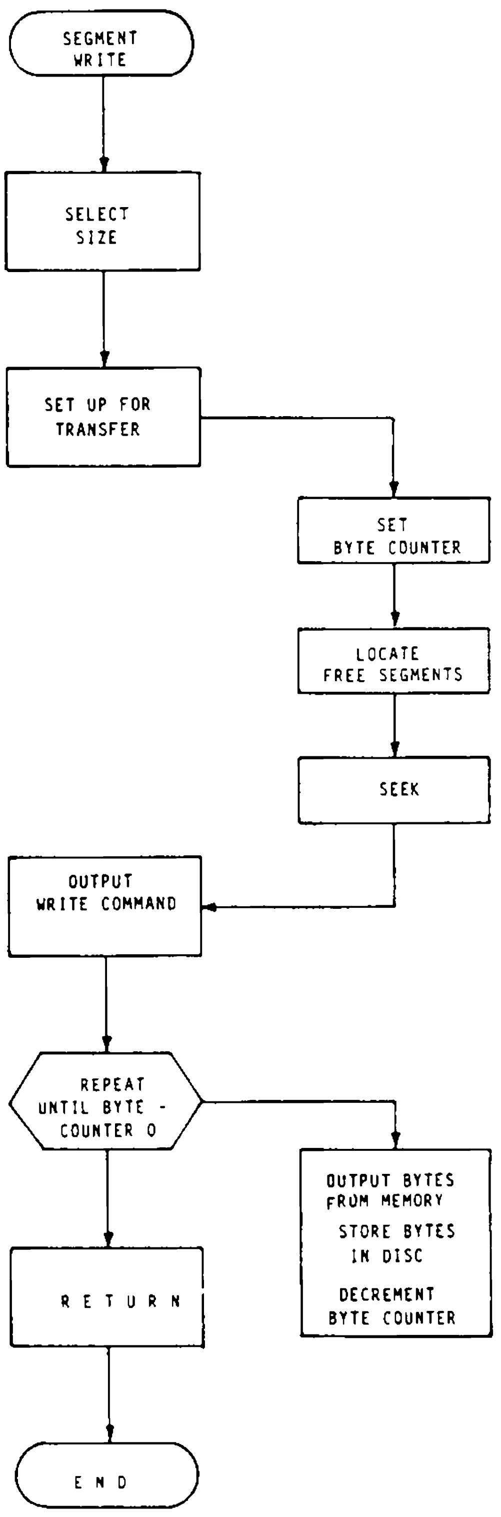


Fig. 6.1 Disc operation flow chart

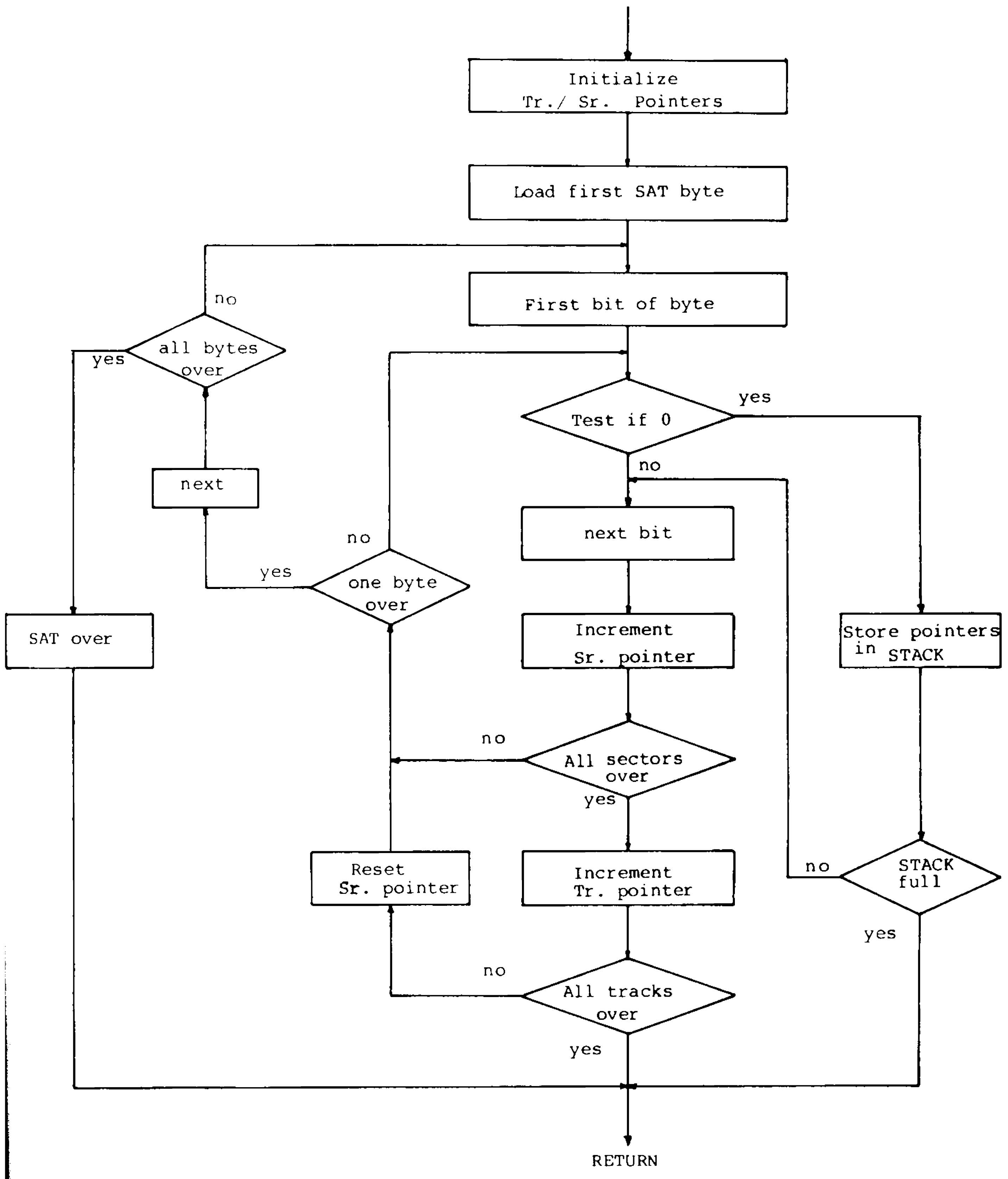


Fig.6.2 Creating BS stack

The decision regarding BS or ES will select the appropriate segments for storage. A BS will select one address, an ES2 will select four addresses, ES3 eight and ES4 sixteen addresses from the BS stack, forming corresponding single blocks. Before selecting the addresses for the next block, the control software will skip an equal number of addresses in the BS stack, thereby scattering the storage blocks and granting respite to the software for completing housekeeping tasks.

During deletion, the appropriate SAT bits are reset to denote a free status. The free space can then be allocated on demand, either in part or in one block. This will minimise the occurrence of unused free spaces in a "full" disk, thus reducing the cost per bit of storage.

The scheme presented is advantageous owing to the following reasons :

i. The waste of space due to some blocks remaining partly filled is reduced.

ii. The waste of space due to chain pointers is reduced Small blocks would require more pointers.

The system is adaptable to environments calling for 'large files only', simply by enlarging the size of the base segment. This will reduce the size of the SAT as also limit the file name appendix, providing more space to the user and reducing housekeeping time. At the other end, the base segment can be retained to have the size of one sector, and selection of block size can be from among multiples of base segment.

6.5 Block transfer and DMA

Controllers for block devices can conveniently make use of DMA concepts. Here the CPU gives two items of information, in addition to the block address - the memory address associated and the number of bytes to transfer. After the controller has read the entire block from the device into its buffer and verified the checksum, it copies the first byte into the main memory, at the specified DMA memory address. Then it increments the DMA address and decrements the DMA count. The process is repeated until the DMA count becomes zero, at which time the controller causes an interrupt.

The Controller described in Chapter 4 will be overtly useful for vectored transfers from the disk. This is particularly significant in the light of the demarcated primary storage, mentioned in Chapter 5. On line data/program can be transferred over to the appropriate blocks, by the OS which will augment the file name/identity of the data block in the disk with the appropriate destination memory address. These addresses can directly go to the DMAC stacks, which will manoeuvre the transfer.

6.5.1 Transfer efficiency

In the environment of transfer discussed, we define transfer efficiency as the ratio of the time involved in the actual operative part of the job, to the total time which comprises the supportive part as well. Accordingly, we have

$$T_{io} = T_{rw} + T_{hk}$$

where Tio represents the total time involved in the transfer operation

Trw represents the actual time of read/write operation and

Thk represents the time taken for associated housekeeping.

Table 6.1 gives quantitative results which indicate that the new design cuts down T_{hk}. This is because, the number of

Record size in bytes	Thk for fixed block size (ms.)	Thk for pro. block size (ms.)
256	65	65
512	140	90
768	210	140
1024	290	210
1280	380	280

Table 6.1 T_{hk} for the two schemes

seek operations is reduced as a result of the predefined block size; the drive head need not have to go each and every time to the housekeeping track to locate free sectors. Again, since the segments themselves are scattered, there are instances when one full revolution time is eliminated. Consequently a high efficiency of transfer can be achieved.

6.6 Relevance

The enhanced transfer efficiency is advantageous in data base applications. It has been observed that the service time is less. The basic measurement statistics used is the response time of requests that are processed by the data base systems. The response time of a request is the time between the initial issuance of the request by the user and the final receipt of the request set. Usually a data base system processes information from the secondary memory using a fixed size block. If the block size is allowed to change, then the response time would vary. It is observed that there is considerable improvement over response time with the concept of programmable block size. Table 6.2 shows the values of response time in seconds for different requests. Since the majority of the work done in a database is to retrieve data, we limit our measurements to only retrieve requests.

Data base request size	Response time fixed	Response time programmable	Improvement in response time Secs
2 records	21.5	21.5	0
4 records	38.7	26.2	12.5
6 records	52.5	37.4	15.1
8 records	67.2	47.1	20.1
10 records	80.0	54.9	25.1

Table 6.2 Response time improvement

The response time is arrived at after five repetitions of each request. A typical block size of 512 bytes was assigned and records were of size 256 bytes. For the programmable block size, the same size as that of the request, is assigned.

Chapter 7

MAINTENANCE AND ENHANCEMENT

7.1 Introduction

Embedded systems, by their definition, must respond to real world events. The cost of redesigning or rewriting software to respond to the continuosly changing requirements of the real world is prohibitive. Therefore, such dedicated systems undergo constant maintenance and enhancements during their life times. Several studies reveal that software maintenance consumes about 50% of data processing budgets and programmers spend 50 to 80 % of their time doing maintenance.[Parikh G., 1986]. The cost factor involved is growing over the years, as given by Boehm.[Boehm, B.W., 1981]. Fig 7.1 shows the plot of cost factors over a specified period, composed by Parikh.

In spite of the cost and time involved in maintenance, very little research is reported on maintenance techniques and associated tool development. This chapter presents a review of the work reported in software maintenance and presents a

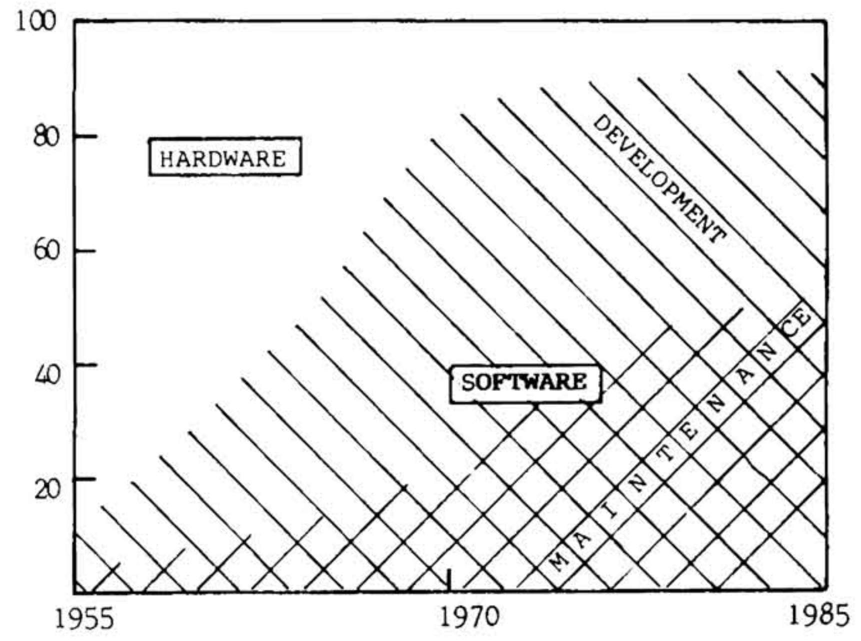


Fig. 7.1 Cost factors

- G·5042 -

scheme to augment systematic maintenance and enhancement.

Martin J. et al, define maintainability as the ease with which a software system can be expanded or contracted to satisfy new requirements.[Martin J., et al,1983]. The problem for most maintainers is the loss of traceability. This is defined as the ability to identify the technical information which pertains to a software error detected during the operational phase.[Kline M.B., et al,1981]. This is applicable as well to post requirement changes. Belady believes that "old" software is an important asset, embodying a wealth of experience, and constitutes an inventory of ideas for identifying the building blocks of future systems.[Belady L.A., 1979]. Lehman suggests that change is intrinsic in software, and must be accepted as a fact of life. [Lehman M.M., 1980]. He further states that large programs are never completed, they just continue to evolve.

Software maintenance authors have made many suggestions for improving the maintainability of software. These suggestions can be classified into three categories :

design approach

maintenance practices

management

Silverman and others suggest a design approach with a part list and connection list. The part list shows functions and the data associated with the functions. The connection list shows the input output relationship between functions and data.[Silverman J., et al, 1983].

Some maintenance practices suggested include change management, guidelines for modifying and retesting, and the like, which are generally based on local information. But Letovsky et al point out that making assumptions about the plan of a program on the basis of local information could lead to an inaccurate understanding of the program as a whole.[Letovsky S. et al, 1986].

Some guidelines offered by Mcclure for improving management of maintenance are to involve maintainers in design and testing and to provide design documentation to maintainers.[McClure C.L., 1981].

Structure chart, Data trace and Control trace are some of the tools available which address various areas of maintenance. [Martin J., et al, 1983].

Automated documentation tools are now available; using source program as the input, the package gives program documentation as the output. A typical example is the package that generates flowcharts for existing COBOL programs. Such software packages often are found to have the following flaws

i. over documentation

ii. complex parts of the program may still remain incomprehensible

iii. the kind of graphical tool used, for instance a flowchart output may affect the understanding of the documentation.

7.2 The Problem

A crucial point with respect to maintenance is that an engineer making changes is not necessarily the implementor. In many cases maintenance is performed on software systems for which very little or no documentation is available. An important phase is to understand the existing program before proceeding to modify. Understanding of the software pertains to the type of information that we wish to find out about it. The ideal environment would require every variable used to be annotated, every subroutine accompanied by a description of what it is intended to do, and what algorithm it uses.

This understanding of instruction streams is also the heart of automatic programming, which is projected for further work in the later chapter. However, a host of fundamental problems remain to be chipped away from many directions. An approach to assist programmers, rather than replace them is envisaged.

7.3 Motivation

A major motivation is the reusability factor which is most significant in improving software development productivity and quality. One of the problems identified by Ramamoorthy and others which limit reusability is the problem of understanding someone else's program.[Ramamoorthy C.V., et al, 1988]. If the original author of the program is not available to answer questions about the program, then this problem becomes quite critical. A typical situation would be the environment of a training lab where multiprocessor system development is on the cards, making use of available uniprocessor systems. Low level systems like kits, trainers, intelligent VDUs and the like, are likely to be adapted during the various exercises on interfacing and/or integration. It often happens that while handling several ROM chips, one loses track of which is which; the visual identification mark having worn out. A reading of the contents would give the codes, which obviously are not directly revealing.

7.4 Observation

Experience gleaned from analysing many monitors and system software indicates that they are largely composed of four types of modules : Actions, Decisions, Loops and Transfers of Control.

Actions are the actual things the program does, such as incrementing a counter or outputting a character from a buffer; whereas the others are mostly supportive. A host of distinct memory addresses also appear which would be pointers, subroutine / loop branching addresses and/or flags. Forward and reverse jumps occur in cases where relative mode of addressing is permitted. Altogether, it presents an intricate picture, when a hard copy listing of the software is made out, the problem of overall identity not withstanding. Understanding could be made easier if it is supported by systematic documentation. The basic suggestion is to include these systematic tips as part of firmware itself. Software size may become less of a constraint these days as memory densities rise and costs fall.

This work incorporates a 'Maintenance Folio' in system firmware. A series of locations at either end of the ROM may be devoted to hold the MF; which shall be called the MF block. This matches a directory in a file storage system, where all pertinent information is made available. What all information in the MF block could be helpful, is a poser which requires trade-offs as well as an awareness of the maintenance bottlenecks, brought out earlier in this chapter.

7.5 The MF structure

The MF starts with an ID field, which carries the identity of the software, expressed as a fixed number of characters, represented in ASCII hex, like for example, codes for SB MICRO OS, indicating Operating System for Single Board Microcomputer. This is followed by a list of subroutines/modules available, giving their addresses as well as a short legend code which would characterise their functions. This is referred to as the MD field. Then comes the AD field which comprises a list of addresses; each address with a coded information about the one out of its several possible roles mentioned in the previous section. Depending upon the complexity of the particular software, its MF can have more information fields to give indications about Actions, Decisions or the like. This is where tradeoffs are required as to the degree of 'friendliness' offered.

Each field is demarcated by a predetermined number of characters. Since the MF lies in a well defined location, either at start or at end of ROM, reading and converting to ASCII could be a software function; giving 'very friendly results'. It is argued that for any firmware, a scrutiny of the MF block proposed would give an insight into it and smoothen out maintenance procedures.

7.6 Scope

Parallel computers generally provide subroutine libraries containing functions that can be invoked by the programmer.[Karp A.H., 1987]. The MD field suggested, though not a library in the strict sense, accomplishes its role at a lower level reasonably well. The pointers which constitute the AD field help parameter passage as well as communication between processors, by providing a shared location. It may be noted that the pointers are necessarily in the RAM area. Any interested processor can have access to the MF block without interfering with other processor's activity.

Operations on multiple processors do not necessarily occur in precisely the same order from run to run. Some work is reported in developing a parallel debugger that will record the order of events in a program, so that events can be replayed in sequence to help diagnose errors.[Howe C.D., et al, 1987]. It would be worth while to incorporate this sequence information as part of the MF block.

7.7 Quantitative results

In a maintenance environment, it is usually extremely difficult to measure productivity.[Parikh G., 1984]. Some quantitative aspects of measuring maintenance programming productivity are :

i. Number of change requests handled by a programmer in a period of time may indicate productivity. However, some change requests may be large and complex, while others may be simple.

ii. Lines of code added/changed/deleted.
Automated auditors are now available which help identify automatically, program changes in the new source code compared to the old code.

iii. Bebugging technique
Bebugging means artificial insemination of bugs in a program. The time consumed by a programmer to correct a bebugged program is an indication of his productivity.

Quantitative measure of the benefit derived out of using the scheme has been arrived at by the following procedure. Various attributes were taken care in adopting the procedure, namely :

1. Programmer experience
Two graduate students were selected as programmers, who had the same exposure to microprocessor systems.

Their familiarity and programming experience may be reasonably assumed to be the same.

2. Program modularity

   The same program, two copies were given to both of them.

3. Quantity of change

   Represented by the number of lines changed, added and deleted per unit of time; for eg. programmer hours.

4. Quality of change

   This is subjective. This is assessed in a review.

5. Quality of documentation

   This is also subjective and assessed in review.

Table 7.1 shows test results obtained for a drive controller software, modified with the aid of, and without the MF block.

P1 and P2 are the two programmers who were assigned the job. The change required to be incorporated is the same, namely to change the size of the block involved in a single read/write operation. P1 who was aided by the MF block had a very clear advantage.

The incidence of change in block size occurred in four different modules in the ROM supplied.

Programmer Team	Programmer hours for P1	Programmer hours for P2
T1	3.2	5.8
T2	3.6	5.4
T3	3.1	5.0
T4	4.0	6.2
T5	3.0	5.8

Table 7.1 Test results for modification

Values for programmer hours were arrived at after repetitions with different set of programmers.

Almost identical results were obtained when maintenance was undertaken after bebugging. P1 had distinct advantage over P2 in identifying the bug and to corect it. The results are shown in Table 7.2

Column 1 and column 2 show programmer hours spent by P1 and P2 to fulfil the task, repeated for five different instances in each case.

7.8 Reuse - A case report

This section presents a sample case of reuse of typical software.

Studies show that reusability is the most significant factor in improving software development productivity and quality.[Bitar I. et al, 1985]. In the multi microprocessor context, available uniprocessor software can advantageously be tailored to the specific realisation. This section mentions a case in context.

Number of codes bebugged	Programmer hours for P1	Programmer hours for P2
2 (in same module)	0.5	1.5
	0.45	1.55
	0.7	1.65
	0.8	2.0
	0.65	1.9
2 (in separate module)	1.2	2.45
	1.0	2.3
	1.15	2.2
	1.3	2.4
	1.4	2.65

Table 7.2 Test results for debugging.

In this work, a program is supposed to be formed by different modules and the total set of programs form the system. In other words, a unit of software that performs a small useful task would constitute a module. Fig 7.2 shows the hierarchy.

Myers mentions some characteristic features of modules [Myers G.J., 1978] :

* it is a closed subroutine
* it can be called from any other module in the program
* it has the potential of being independently assembled.

Yourdon and others define modules as a contiguous sequence of statements bounded by boundary elements, having an aggregate identifier.[Yourdon E., et al, 1979]. This definition means that

* the statements in a module follow one another from a logical viewpoint
* there are distinct initial and terminal points in a module representation.

A scheme which permits reuse of a low level assembler by adapting it to support parallel processing activity by the different processing elements, is presented. Most assemblers contemplate a modular structure of source program with

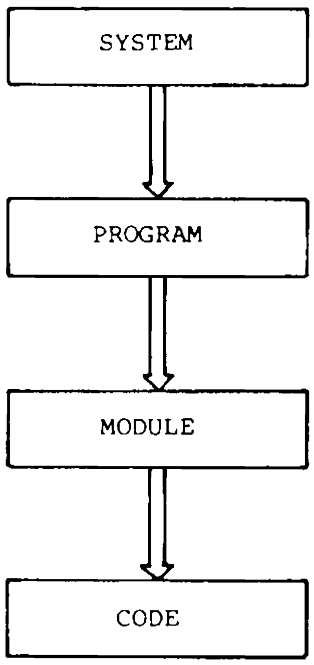


Fig. 7.2 Software hierarchy

appropriate labels. Each line in the source program usually contains a statement which is split into fields like LABEL field, OPCODE field, OPERAND field and COMMENT field. The comment field, although ignored by the assembler, is essential to good programming style and should contain a clear description of the function of each line of code. This will simplify maintenance procedure as well as help create the MF structure.

The primary function of the assembler comprises evaluating the fields, processing labels and assigning addresses. In a typical pass, the assembler identifies the labels and forms a label table with the addresses corresponding to each appearing in the next two bytes. In the modified version, a module pile is formed, which can be a first-in first-out stack; its elements being the label ID of the program subsets. The address counter in the assembler, which keeps track of each instructions address can be made use of to get the address of the last instruction in any module, indicating its size.

This linear list of modules may be termed a 'pipe' where all insertions, removals and accesses are made at the ends only. The pipe which is formed as an offshoot of the label table during assembly, holds module names, their addresses and size information; the last helping housekeeping of appropriate resource blocks. Fig 7.3 shows the flowchart of the modified assembler (MODAS).

7.8.1 Relevance

Chapter 5 has introduced functional blocks in the resource memory and each block is earmarked to hold predefined classes of tasks which have been identified. In the modular approach discussed, classification is simpler, particularly when confronted at the assembler stage. The specific class to which a module belongs is also determined. For instance, a subroutine which forms a distinct class, can be identified by the entry label and the subsequent return instruction; an iterative loop will be characterised by a counter and a conditional branching. Presence of a memory address used as a pointer, would indicate reference to a value content, and hence dependence on another task. As a matter of fact, the MF structure introduced, carries such information as subroutine labels, counter and pointer addresses etc.

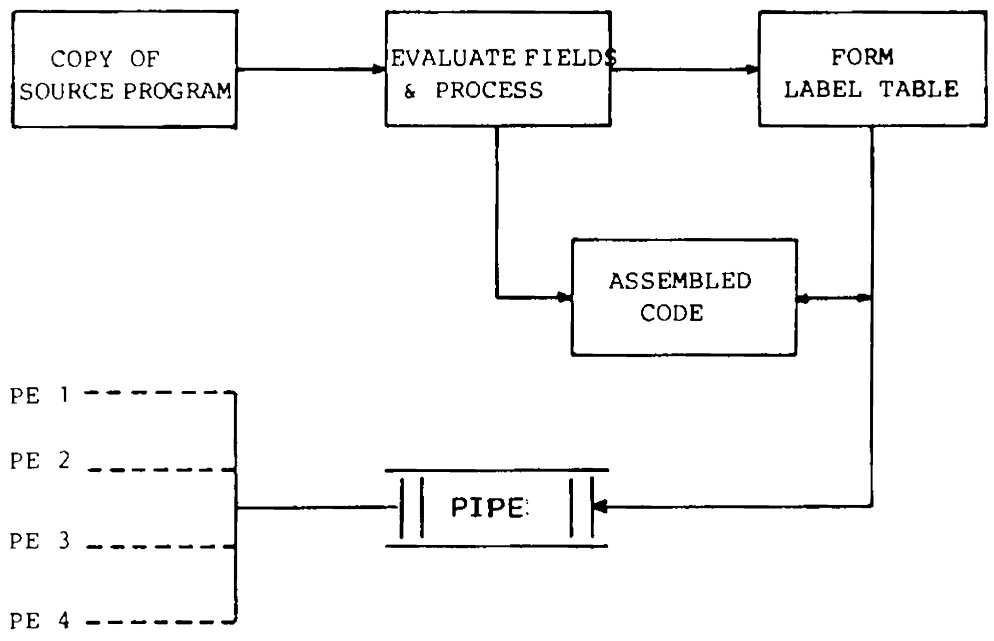


Fig. 7.3 MODAS flowchart

7.8.2 Conclusion

The pipe can be conveniently located in the block accessible to the various processing elements, as envisioned in Chapter 5. Communication inter-module and intra-module, can be effected by writing to and reading from the block. As soon as an entry appears in the pipe, execution can proceed with, if resources permit. The MODAS pursues its task, making fresh entries in the stack. Each processor whenever free, looks into the pipe for tasks pending. Alternately, interrupt concepts can also be utilised advantageously.

In the case presented, it was observed that only marginal changes were required to make it support a new structure. This purports to the significance of maintenance. The typical assembler was one developed in-house, for MC 6800 processor.

Chapter 8

TOWARDS IMPROVING PROCESSING RELIABILITY

8.1 Introduction

This chapter attempts to extend the concept of software adaptation / maintenance and presents the ground work for improving processing reliability. Maintenance is often mentioned as the twin sister of reliability.[Parikh G., 1984]. The ultimate objective would be to make the processor comprehend, improve and correct programs, by itself during execution. This is suggested for further work; broad guidelines likely to help steer the work are presented.

Current day sequential computers which are more popular than any other architectural class, are guided entirely by the instruction codes which constitute a given program. Processing proceeds by fetching these codes as they appear in the program. No facility exists to verify if a particular code being decoded is the correct one. Empirical observations indicate a pattern of likelihood for instruction occurrence in a stream, which would help predict a plausible sequence. Relevant information about the codes/stream is made available to the processor, based on which a built in smart logic checks the appropriateness of a code in a stream.

A new format of instruction, based on RISC ideology is formulated with a view to facilitating rapid decode, through use of a consistent opcode field; which would simplify the process. The corresponding organisation for the processor is derived. If there are too many instructions as with the case of CISC processors, level of complexity of the smart logic may be too high. The format presented is also helpful in moving programs made out of such instructions into new environments by a simple pre-processing.

8.2 Current Status

A review of instruction codes of different microprocessors would reveal that the pattern of bits do not follow any hard and fast rule within one processor or without. Eventhough clearly identifiable fields are present to mention opcodes and operands in some processors, their specific positions are never standardised. For example, in the case of the INTEL 8085 microprocessor model, if two MS bits alone specify opcodes in some instructions, there are others where two more LS bits are utilized; in such cases the operand field occurs in their midst. Again there are codes where operands are mentioned by LS bits alone. In another case of the MC 6800 microprocessor, codes do not appear to have any distinct fields at all. It has been observed that codes devoid of well defined fields are unattractive; particularly when attempts are made to predict their sequence, as also to adapt them to a new environment.

Traditionally the general trend in designing microprocessors has been towards increasingly complex instruction sets and associated architectures. But recently, a shift is seen towards a simple set of instructions judiciously chosen, and corresponding simple machine organisation. The focus has been in increasing the reliability of individual instruction streams, rather than going in for large instruction repertoires.[Toong H.D. et al, 1981]. In fact, a whole spectrum of VLSI special purpose and general purpose processors were developed based on the philosophy of simplicity and replication.[Trleaven P.C., 1984].

8.3 Methodology

In the new format presented, codes form decisive groups based on the number of operand bytes involved in the particular instruction. Group 0 instructions have no operands, its function being that of a controller; while group 1 will have a single operand. A mention of two operands which are exclusively data bytes, like the addend and augend in an ADD operation, constitute group 2. If mention is made of a two byte address, it is group 3. Thus, in a sample case of an 8-bit instruction and 16-bit address, two MS bits are reserved to demarcate one of the four groups identified.

Next, we propose to specify the location of opcodes in the format. This warrants a scrutiny of the register organisation as well. Adverting to the concept of a small register file, which would help to have only a few simple instructions, an organisation is arrived at based on analysis of various register profiles. The analysis has been conducted by assigning Register Credit Values (RCV) as given out in Table 8.1. The registers selected for the new organisation are an ACC, an IX, an SP and a status register, apart from the PC, which is indispensable in the Von Neumann scheme. It may be noted that an all memory architecture

TABLE 8.1 REGISTER CREDIT VALUE

Features CPU	6800						8085							Z-80														
registers	PC	SP	IX	CCR	AC1	AC2	PC	SP	AC	BC	DE	HL	ST	PC	SP	IX	IY	IPR	MRR	AC1	AC2	BC	DE	HL	BC'	DE'	HL'	FLAG
1. Architectural Indispensability	°				°		°		°					°						°								
2. Variable Location Reference			°													°	°											
3. 16-bit address handling		°	°					°		°	°	°	°		°	°	°											
4. Fast Interrupt response		°		°				°							°										°	°	°	
5. Programming ease		°	°	°		°		°		°	°	°	°		°	°	°	°	°		°	°	°	°	°	°	°	°
6. Subroutine Call		°		°				°					°		°													°

would have about 47% of data references for temporary storage of intermediate results.[Flynn M.J., et al, 1987]. Use of registers is resorted to, in view of this contingency.

In the 8-bit instruction word example, the next two MS bits specify the register associated with the operation. The remaining bits characterise the logic. Table 8.2 depicts a few bit patterns in the new format. The use of consistent opcode field, a RISC phenomenon, allow rapid decode and swift adaptation. Decoding of the instruction proceeds by identifying the group to which it belongs and then looking for operand specification as well as opcodes. The operand location is also well defined with respect to the groups. For a group 1 instruction, it lies invariably in the ACC. For group 2, one operand lies in the ACC or other register specified by bits b5 and b4, and the second lies in a memory location dedicated for the purpose, designated "MEMOP" - Memory Operand. It would be desirable to locate MEMOP next to the stack area, so that whenever stack is initialised this address is set, which is pointed to by the SP of a virgin stack.

In group 3, the two-byte address, obviously may be held by MEMOP together with its adjacent higher address location.

TABLE 8.2

CODE FORMAT

b7	b6	b5	b4	b3	b2	b1	b0

b7	b6	: group
0	0	0
0	1	1
1	0	2
1	1	3

GROUP 0 :

b5	b4	b3	OP	b2	b1	b0	
0	0	0	HLT	x	x	x	don't care
0	0	1	NOP				
0	1	0	EI				
0	1	1	DI				
1	0	0	SI				
1	0	1	RtS				
1	1	0	RtCn				

GROUP 1 :

b5	b4	Reg.Indr.	b3	b2	b1	b0	OP
0	0	ACC	0	0	0	0	CLR
0	1	IX	0	0	0	1	COMPLT
1	0	SP	0	0	1	0	INCR
1	1	Stat.	0	0	1	1	DECR
			0	1	0	0	ROTLT
			0	1	0	1	ROTRT
			0	1	1	0	PUSHD
			0	1	1	1	PULLD
			1	0	0	0	BITST
			1	0	0	1	DCMLAD
			1	0	1	0	LDACC
			1	0	1	1	STACC

GROUP 2 :

b5	b4		b3	b2	b1	b0	
0	0	ACC	1	1	0	0	ADD
0	1	IX	1	1	0	1	ADDwCy
1	0	SP	1	1	1	0	AND
1	1		1	1	1	1	OR

compare
with [Reg] specified by b1 b0

GROUP 3 :

b5	b4	b3	OP	b2	b1	b0
0	0	0	LDIX	x	x	x
0	0	1	STIX			
0	1	0	LDSP			
0	1	1	STSP			
1	0	0	JTSub			
1	0	1	JSCn			
1	1	0	LDOPIM			
1	1	1	STOPIM			

Addressing is just direct or indexed, again a RISC feature namely few addressing modes. The excessive use of indexed computed mode of addressing, though deviates from the RISC scheme, is advocated as it enables shifting of routines to a different area of memory, by altering the initialisation part. This is particularly significant in the light of the partitioned resource memory, and the different classes of tasks proposed in Chapter 5. It also facilitates an excellent memory expansion technique using indexed mapping.

8.4 Adaptation

The well defined format presented, enables swift identification of the different attributes in an instruction. With few addressing modes and the prescribed operand locations, a simple pre-processing would enable migration of programs to new environments. Conversely customised instructions can be rebuilt after identifying opcodes and determining operands, by resorting to a table look-up or other standard cross assembler procedures. The sequence proceeds by first fixing the group based on the number of operands involved and assigning the respective bits. The operands are then put in the appropriate locations specified.

The opcode field is checked and the appropriate code/codes generated. As a matter of fact, additional codes may be required to accomplish the operation.

Implementation with fewer instruction types obviously requires additional instruction memory traffic. This is alleviated by providing a small instruction cache as an on-chip buffer. This would be kept full by a prefetch logic.

8.5 Smart Assistance

Rapid progress in semiconductor process technology enables the increasing integration of systems on chip. Functions previously exclusive to minicomputers or large scale machines are now appearing in microprocessors. This latter motivation led to suggest an on-chip Smart Processing Support (SPS) which would promote a high level of system reliability. Two significant aspects of an intelligent assistant [Kaiser G.E. et al, 1988] have been identified :

* it should provide an insight into the process.
* it should be a participant in the processing by providing suitable instructions as and when required.

The insight is provided by the MF structure proposed in the previous chapter. The SPS looks for the MF and off loads its content to a dedicated buffer provided on chip. This will serve as a ready reference as regards the functional role of various entities identified.

Implementing a small set of instruction would result in a substantial saving in control logic. Also, it reduces the range of probable instructions in a stream. The saved area on chip can be used to hold storage facility, which in turn is used to implement the SPS. The SPS participates in the processing and strives to provide suitable instructions as and when required.

8.6 Observation

Church has given the probability of occurrence of each different class of instructions [Church C.C., 1970] as shown in Table 8.3. As a preliminary attempt, only the first two groups which together constitute almost 75 % of all programs, are subjected to scrutiny.

It is often the case that when an instruction is executed,

TABLE 8.3

Functions	Rate of occurrence
Data transfers (LOAD, STORE, MOVE)	45 %
Program Control (BRANCH, CALL, RETURN)	29 %
Arithmetic	10 %
Compare	6 %
Logical	4 %
Shift / Rotate	3 %
Bit Manipulation	2 %
Input/Output and Control	1 %

not all possible instructions are equally likely to follow. For example, a LOAD ACC instruction is not followed by a STORE ACC with reference to the same address. Taking advantage of these facts, opcodes are grouped into clusters, where the members of a cluster are likely to follow one another. In another instance, an increment instruction would never be followed by a decrement instruction. But when it occurs in a counter routine which makes use of a memory location, then it may be followed by a store instruction. Another code might compare the value of the counter to decide on a branch operation. The SPS foresees a branch code in such a situation and verifies the appropriateness of the instruction in the stream.

A typical case of comprehension by the SPS might occur like this: an instruction refers to a memory location which is specified as a counter flag in the MF structure. The likely sequence would be

```
LOAD ACC
INCR/DECR
STORE ACC
BRANCH CN
```

8.7 Design Approach

The organisation of a typical processor, incorporating the SPS logic is suggested, which comprises a set of tacks. This is shown in fig. 8.1. These stacks provide for replacement of codes at instruction level, module level or supply results of program depending on the level of complexity. The structure of the logic is to be worked out.

8.8 Conclusion

It is argued that the new organisation provides a novel technique at improving reliability. It serves as an alternative to software replication, which by no means resolves the problem completely. The design is intended to be a basement; further work is suggested in modelling and evaluating it. The objective would be to arrive at a smart logic that comprehends, improves and corrects programs. Systems become more reliable when this is achieved.

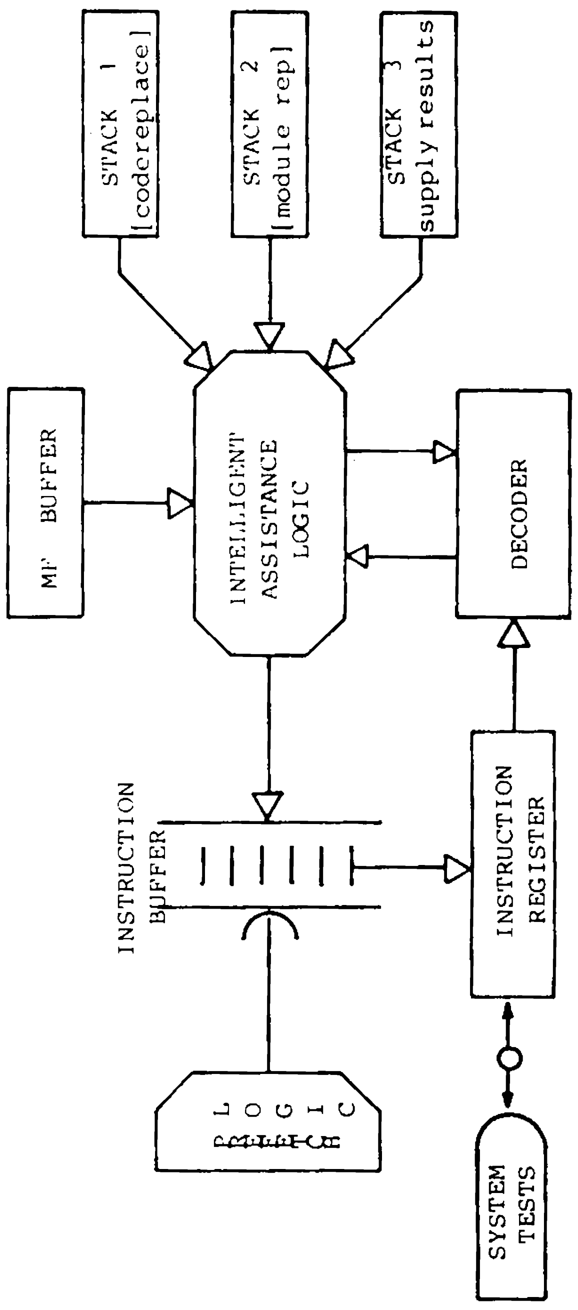


Fig. 8.1 Design approach

Chapter 9

CONCLUSION AND SUGGESTION FOR FURTHER WORK

It is very probable that future attempts at performance improvement in embedded applications should include multiple microprocessor approach. Resource sharing and adaptation/ enhancement are key features for such systems. Three distinct levels of sharing occur at Control, Information and Storage.

A new controller has been designed, simulated and tested. The test results indicate that the design supports powerful and efficient interchange of control codes, as well as data. It provides ideal controllers for multi microprocessor systems. This is an improvement over earlier controllers.

A typical application scenario has been thoroughly explored. It has been observed that processes rank differently for system service. A classification of such processes has been done and the system subjected to analysis, with a view to optimising throughput. Ideal condition for resource usage has been arrived at. This obviates the need for powerful operating system and expensive hardware.

The storage level purports to the data base component of embedded systems, whose performance is limited by the disc to memory transfer rates. A new scheme is suggested which would optimise disc operations. Test results indicated that transfer efficiency, that has been defined, would be better with this scheme. Storage cost also is less.

Dedicated systems are found to undergo continuous changes during their span of life. Enormous amount of effort as well as cost is saved by resorting to certain maintenance tips suggested. Quite different from conventional documentation, this is made part of firmware. Adapting such software is easier and more efficient, as indicated by experimental results.

Groundwork for more reliable processing units, quite different from simple replication, has been laid. A new organisation has been arrived at with a view to ascertaining the appropriateness of any given code in a stream.

The problem of predicting the sequence of an instruction stream, based on the likelihood of occurrence, may be pursued to the level of integrating the necessary logic on chip. This'promotes the level of system reliability. System level solutions rather than functional redundancy is adopted to accomplish this. It would turn out that a little hardware enables to support a highly reliable system.

The CPU architecture is to be tailored to ensure certain degree of autonomous or involuntary operations based on statistical studies and previous history. The logic/ intelligence added to the CPU should "gauge" the mind of the programmer. Such intelligent processors can assist the user by making the program writing process itself interactive.

The concepts are bound to have their ramifications on LANs by fully exploiting their potential advantages. One of the characteristic features of LAN is the facility whereby files can be transferred between any machine on the network. The controller developed, with its unique features , proves to be very effective here. More over, the disc/file server is able to serve more number of clients by adopting the programmable block size cocnept.

-:0:-

APPENDIX A1

SWITCHING UNIT

Fig. A1 shows a matrix switch that can be used if more than two processors are involved. This supports the controller designed, to the extent of decoding CALLING and CALLED processor addresses and providing a Bus Switching Network.

The CALLING decoder selects the appropriate processor for HOLD and buffers the source address from the controller ADDR. register. The data which reaches the data buffer will be switched to the destination processor buffers which have been enabled by the CALLED decoder and passed to the destination address indicated by the controller register.

It is assumed that all the processors involved provide address and data buses, R/W control, HOLD and HOLDACK lines at connector. The address decoding is done by the incumbent processor support hardware.

Since only one pair of buffers are activated at a time, to take part in a transfer, the problem of interlock will not arise in these operations. The controller itself provides access only if it is not engaged by any processor that has initiated a transfer.

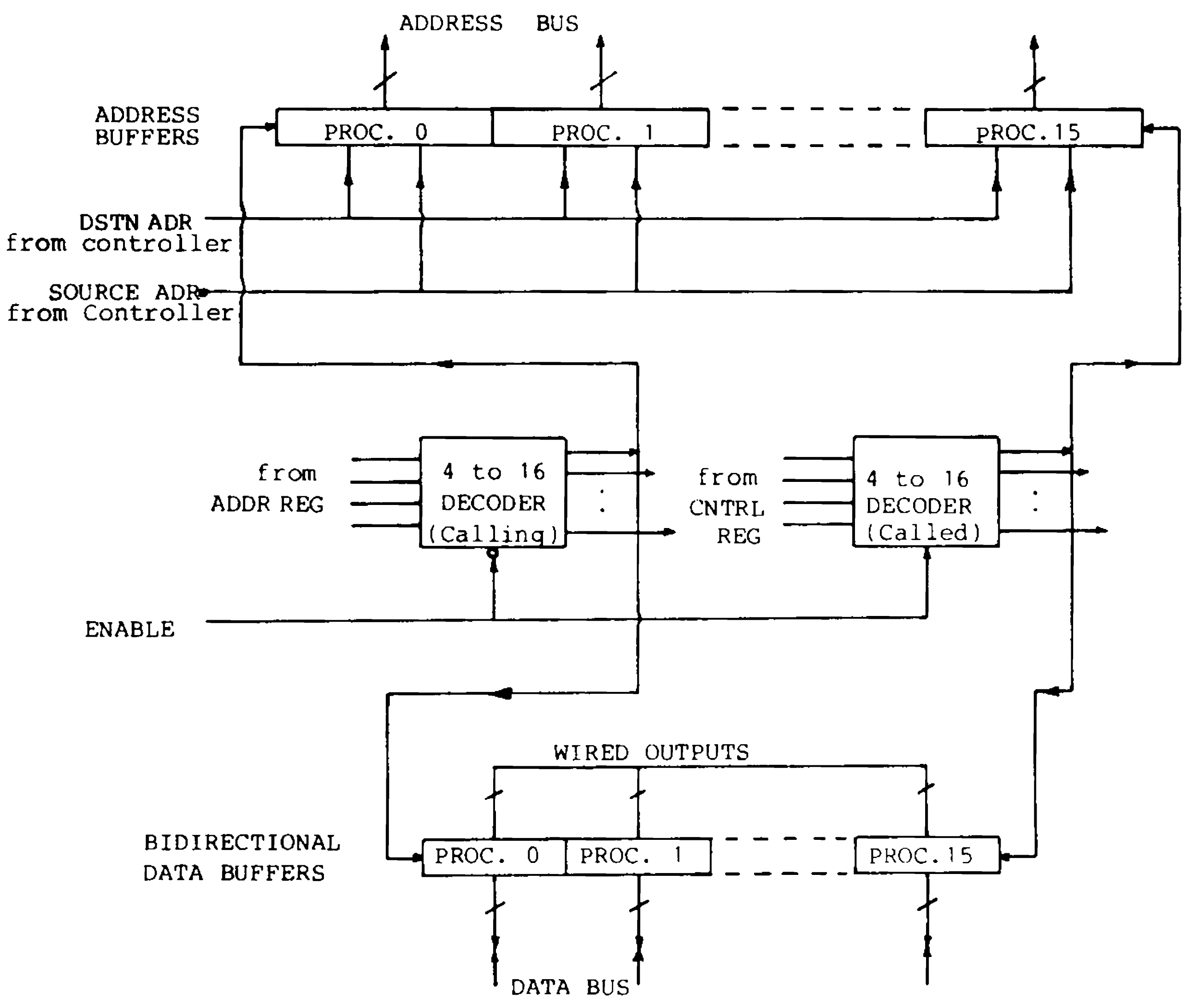


Fig A.1 Switching unit

MICROSYSTEMS: Architecture, Integration and Use
C.J. van Spronsen and L. Richter (eds.)
North-Holland Publishing Company
© EUROMICRO, 1982

A NOVEL ARCHITECTURE FOR DMA CONTROLLER

K. Paulose Jacob — O/E/N India Ltd, Cochin, India

C.S. Sridhar — Department of Electronics, University of Cochin, Cochin, India

A new DMA Controller architecture is proposed, combining the features of available controller and peripheral interfaces. It is argued that provision of a stack in the controller enhances its power by enabling access to non-sequential locations also.

I. INTRODUCTION

Commercially available Direct Memory Access (DMA) Controllers normally provide access to the memory of a processor for peripheral devices. These are programmable to the extent that start address and number of bytes to be accessed, are written into the controller which in turn, by raising a HOLD or such signal, captures the busses and executes the transfer of data [1]. Also available are PPI's (Programmable Peripheral Interface) which incorporate a large number of powerful modes for data transfer, interrupt etc.[1,2]. One drawback of the former, namely DMA Controller in that access is limited to a block of data sequentially located in the main memory. For large systems very complex BIM (Bus Interconnection Module)[3] may be required for data exchange.

. is proposed that features of DMA Controllers and PPIs can be combined in a single chip to enable more powerful and efficient data interchange; not merely between the processor and the peripherals but also between processors. It is also shown that the proposed architecture can easily be upgraded to provide ideal controllers for multiprocessor operation.

The motivation for the design is the difficulty encountered in DMA accession of outputs in batch processing and sharing of routines between processors, particularly when non-sequential locations have to be accessed.

The scheme has been oriented towards a twin-processor operation using 8080-8085 pair. It is felt, however, that the design can be extended to more than two processors, with equal efficiency, provided the individual CPUs are of the same family, e.g., 8080, 8085, Z80, etc. Section II describes the architectural details of the proposed DMA Controller. Section III describes the software required to implement the DMA Controller, while Section IV covers discussions and conclusions.

II. THE ARCHITECTURE

The architecture of the proposed controller is shown in Figure 1.

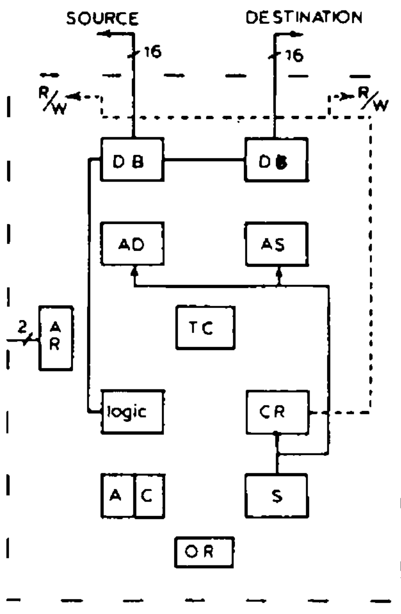


Fig. 1
THE PROPOSED ARCHITECTURE

A in figure is a control section which generates the necessary control signals for the operation. Clock C times the operation. AR is an address register which provides access to the controller for external devices. This register stores data regarding the address of the calling processor or device, as the case may be. Also contained in this, is a one bit flag to indicate 'ENGAGED' condition if more than one processor/device wishes to access the controller. A Control Register CR, written into by the calling processor, stores the details regarding direction of data flow, destination device address, and whether stack S is needed or not. TC is a terminal counter register which stores the number of bytes to be transferred. AD and AS are the address registers buffered through DB tristate buffers, to the address lines of the processor.

The control and logic units provide the necessary control signals

- to listen to a calling processor.
- to provide access to the control register, address buffers etc. to the processor.
- to raise HOLD signals and R/W signals.
- to sense HOLDA and execute transfer.
- to enable all the housekeeping signals like clear, buffer on and such other control signals.

III. SOFTWARE

The operation of the proposed controller is as follows. The controller itself forms an output port for all the processors. The 'CALLING' processor raises a one level at the REQ terminal after testing the MSB to determine 'ENGAGED' or 'FREE' condition. If free, the controller first records the 'CALLING' address and routes the first byte to the control register. The data direction and the destination address are recorded here. A '1' or '0' at b4 in control register, is interpreted as 'STACK REQUIRED' and 'NO STACK' respectively. b0 to b3 form the destination address. b5 is allotted for data direction, namely Read or Write into called memory. b6 and b7 are reserved for housekeeping. The next bytes are routed to AD, AS, TC respectively, if b4 is zero; or to the stack, if it is a one. b6 and b7 indicate depth of stack used. The controller accepts six bytes on mode 0, and a maximum of 24 bytes on mode 1.

The CALLING processor signals end of entry by clearing its Req output. The controller then depending on the mode executes transfer. For eg.,in mode zero, the address registers and TC are filled up and hence the controller will directly raise HOLDs on the CALLING (source), and CALLED (destination) processor, receive the HOLDA from both and raise the R or W signal. The internal clock increments the address, decrements the terminal counter and on finding TC zero, relinquishes the HOLDs and clears all registers to zero. This is necessary, since for 8080 family systems, normally locations starting from 0000 are in the ROM area; hence the registers are set to start from 00 level, so that no malfunction can accidentally write into any memory.

In mode 1 however, the data to all the registers AD, AS and TC is fetched from the first five locations of the stack and repeated b7 b6 times before which the clearing operations are executed.

IV. DISCUSSIONS AND CONCLUSIONS

The proposed DMA Controller is mainly designed towards a multiprocessor-based system. No attempt was made in the present work to minimise or optimise the pin count of the controller chip. For example, if the controller is implemented for two processors, 32 address bus pins, 8 calling data port pins, 4 Hold and Hold Ack, and 2 R/W pins are a minimum requirement. Over and above these, there will be power supply pins.

For getting service from the controller, certain amount of software is needed for each processor. Appendix I shows this software for test processors 8080 and 8085. It should also be pointed out that data lines were tightly coupled. In a true multiprocessor set up, additional hardware in the form of bus switching network will be necessary. The authors can only hypothesize on such a support device and hence only a visualisation is presented in Appendix II.

The routing of bytes for loading the various registers was possible in the present case because they are allotted to the output ports of the processor and nearly 10 cycle time is available for switching. For faster output rates, some more thought should be devoted for implementation. Alternately, a few locations are assigned by the processor for such operations, and the DMA Controller itself acquires these bytes by accessing the memory of the calling processors. For example locations 8F00 to 8F05 may be filled up by the 8085 processor in its own memory and it may raise a Req on the controller. The controller, when it is free, and subject to priority may access these locations by

DMA and fill up the registers CR, AD, AS and TC.

It can be seen that problems of interlock will not arise in these operations. It might also be made clear that the actual system described is based on combining discrete 8255 PPI and 8257 DMA Controller. As a matter of fact, no attempt was made to fabricate a single chip incorporating the features described.

Also, WAIT conditions were not simulated since the RAMs used were compatible with the discrete chips described and the clock rate was not too high.

REFERENCES:

[1] MCS 80 Users Manual, Intel Corporation, September 1975.

[2] 6800 Application Manual, Motorola Corporation, 1976.

[3] Hoener, S. and Roehder, W., Modular Multi-microprocessor architecture with virtual Memory, Proceedings of Euro Micro Symposium, Venice, October 1976.

APPENDIX I

```
SERVE:  LXI H, ADDR    Locations for
        MVI D, STACK   memory address
        MVI C, DESTN   of source and
                       destination and
                       number of bytes
                       for transfer.
                       One if no stack,
                       depth+1 if stack
                       is needed.
                       Load into C
                       destination pro-
                       cessor address
                       (b0 to b3) and
                       direction (b5).

TSFR:   IN, 04         Test port 4
        ORAA           If DMA Control-
                       ler is not free,
                       b0 will be 0.
        JZ, TSFR       Continue testing.
        CALL,FILUP
        RET

FILUP:  MOV A,D        Test if stack is
                       required.
        ORAA           If not required,
        DCRA           directly trans-
        JZ, NSTCK      fer 5 bytes from
                       SERVE.
        DCR D
        MOV A,D

NSTCK:  RAR
        RAR            Form control word
        ORA,C
        OUT,06         Send control word to
                       DMA

DEEP:   MVI,B,05       Load into B 5

CHART:  MOV A,M
        OUT,06         Transfer 5 bytes
        INX H
        DCR B
        JNZ, CHART
        DCR D
        JNZ,DEEP       If stack depth not
        RET            zero, repeat D times.
```

APPENDIX II

A matrix switch can be visualised if more than two processors are involved. It is assumed that all the processors involved are providing their addresses and data buses, R/W control lines, HOLD and HOLDA, at some connectors and that the controller only HOLD's the CPUs. Consequently a mere setting up of address is required on the part of the controller; the address decoding being done by the incumbent processor support hardware. This is presented in Figure 2.

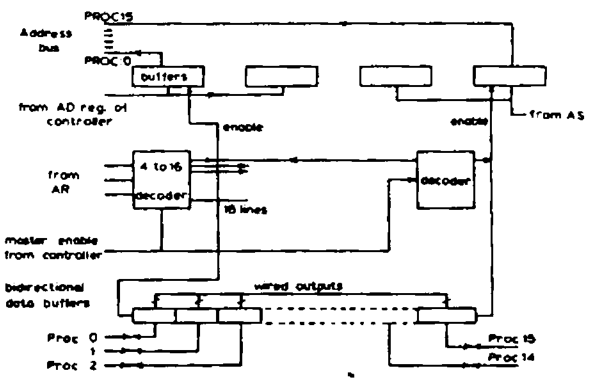


Fig 2. SWITCHING UNIT

APPENDIX B

GLOSSARY OF AVIATION PARAMETERS AND PANEL INSTRUMENTATION

1.	Airspeed	Speed at which an aircraft is moving. Airspeed in knots(nautical miles per hour) is indicated on panel
2.	Altitude	Height of the aircraft above sea level. Altimeter is the instrument that displays altitude in feet.
3.	Aircraft Heading	Direction in which the aircraft flies through the air. 0 degree for North. 180 degrees for South. Directional Gyro is the instrument that indicates aircraft heading.
4.	Attitude	Pitch and roll of the aircraft as compared to the horizon in degrees. Artificial Horizon is the instrument that indicates attitude of the aircraft.
5.	Angle of Bank	Aircraft bank angle from vertical in degrees. Angle of bank is indicated by Aileron deflection on panel.

6.	Vertical Speed	Rate of climb or descent of the aircraft in feet per minute. Vertical speed indicator on panel displays the rate. Elevator deflection is also indicated on panel.
7.	Throttle	Throttle position varies from a computed minimum idle value to a maximum value of 1.0 for full throttle. Position indicated on panel.
8.	Weight	Influences load factor given by n = L/W where L is lift(lbs) and W is weight(lbs). Indicated on panel.
9.	Mach	Mach number is the ratio of the speed of motion to the speed of sound. Mach 1 = speed of sound in feet/second.
10.	Rudder	The primary control surface attached to the vertical stabilizer, deflection of which causes the tail of the aircraft to swing left or right.
11.	Elevator	The primary control surface attached to the horizontal stabiliser, that can be deflected up or down to control the pitch of the aircraft.
12.	Ailerons	The primary control surfaces located at the trailing edges of the outer wing panels, which, when deflected up or down, cause the airplane in flight to bank.

-:0:-

REFERENCES

1. Athas, William C., Seitz, Charles L., "Multicomputers : Message Passing Concurrent Computers", Computer, August, 1988.

2. Basu A., "Parallel Processing Systems : a nomenclature based on their characteristics", Proceedings IEE (UK), 1987.

3. Belady L.A., "Evolved Software for the 80's", Computer, Feb., 1987.

4. Ben-Ari M., "Principles of Concurrent Programming", Prentice-Hall NJ, 1982.

5. Bitar I., Penedo M.H., Stuckle E.D., "Lessons Learned in Building the TRW Software Productivity System", Proceedings COMPCON, San Francisco CA., Spring 1985.

6. Boehm, Barry W.
"Software Engineering Economics", Englewood Cliffs, New Jersey; Prentice-Hall, 1981.

7. Boral H., Dewitt D.J.,
"Data Base Machines : An Idea Whose Time Has Passed? A Critique of the Future of Data Base Machines",
Data Base Machines, Springer-Verlag, 1983.

8. Bourne S.R.,
"The UNIX System", Wokingham : Addison-Wesley, 1980.

9. Bowen B.A., Buhr R.J.A.,
"The Logical Design of Multiple Microprocessor Systems",
Prentice-Hall, Englewood Cliffs, NJ, 1980.

10. Burns A.,
"Programming in Occam 2", Wokingham:Addison Wesley, 1988.

11. Burns A., Lister A., Wellings A.,
"Ada Tasking", Lecture Notes in Computer Science, 262, 1987.

12. Casavant, Thomas L., Kuhl, John G.,
"A Taxonomy of Scheduling in General Purpose Distributed Computing Systems", IEEE Transactions on Software Engineering, February, 1988.

13. Church C.C.,
"Computer Instruction Repertoire, Time for a Change", AFIPS Conference Proceedings, Atlantic City, May, 1970.

14. Denning P.,
"Effect of Scheduling on File Memory Operations" Proceedings AFIPS Spring Joint Computer Conference, AFIPS Press, 1967.

15. Dhamke, Mark,
"Microcomputer Operating Systems", McGraw Hill, BYTE, 1982.

16. Dubois, Michel, Scheurich, Christoph, Briggs, Faye A.,
"Synchronisation, Coherence and Event Ordering in Multi-Processors", Computer, February 1988.

17. Enslow, Philip H.,
"Multiprocessors and Parallel Processing", John Wiley & Sons, 1974.

18. Flynn, Michael J.,
"Some Computer Organisations and Their Effectiveness",
IEEE Transactions on Computers, September, 1972.

19. Flynn, Michael J., Mitchell, Chad L., Mulder, J. M.,
"And Now a Case for More Complex Instruction Sets",
Computer, September, 1987.

20. Gimarc, Charles C., Milutinovic, Veljko M.,
"A Survey of RISC Processors and Computers of the Mid
1980s", Computer, September, 1987.

21. Gupta, Gopal K.,
"Access Methods for Main Memory Data Bases",
Modern Trends in Information Technology, P.V.S.Rao,
P.Sadanandan (Eds.), Tata McGraw Hill, 1988.

22. Herman D.,
"Towards a Systematic Approach to Implement Distributed
Control of Synchronisation", Distributed Computing
System, Y.Paker, J.P.Verjus (Eds.), Academic Press, 1983.

23. Hillis W.D.,
"The Connection Machine", Cambridge MA: MIT Press, 1985.

24. Hoare C.A.R.,
"Synchronisation of Parallel Processors", Advanced Techniques for Microprocessor Systems, F.H.Kanna (ED.) Peter Perigrinus, London, 1980.

25. Hoffner Y.,
"A Reconfigurable Multiprocessor Development System", Microcomputers; Developments in Industry, Business and Education, D.R.Wilson et al (Eds.), North Holland, 1983.

26. Howe, Carl D., Moxon, Bruce,
"How to Program Parallel Processors", IEEE Spectrum, September, 1987.

27. Hwang Kai, Briggs, Faye A.,
"Computer Architecture and Parallel Processing", McGraw Hill, NY, 1985.

28. Iannino, Anthony, Musa J.D.,
"Software Reliability", Advances in Computers, Vol.30, Academic Press, 1990.

29. Johnson E.E.,
"Completing an MIMD Multiprocessor Taxonomy", Computer Architecture News, 16, 1988.

30. Kaiser, Gail E., Feiler, Peter H., Popovicn S.S., "Intelligent Assistance for Software Development and Maintenance", IEEE Software, May, 1988.

31. Karp, Alan H., "Programming For Parallelism", Computer, May, 1987.

32. Kim W., "Highly Available Systems for Database Applications", Computing Surveys, March, 1984.

33. Kline M.B., Schneidewind N.F., "Life Cycle Comparisons of Hardware and Software Maintainability", Proc. Third National Reliability Conference, Birmingham, Apr-May, 1981.

34. Krishnamurthy E.V., "Parallel Processing - Principles and Practice", Addison Wesley, 1989.

35. Kung H.T., "Why Systolic Architectures", IEEE Computer, 15, 1987.

36. Lehman M.M.,

"Programs, Life Cycles and Laws of Software Evolution",

Proceedings IEEE (Vol.68), September, 1980.

37. Letovsky S., Soloway E.,

"Delocalized Plans and Program Comprehension",

IEEE Software, May, 1986.

38. Lister A.M.,

"Fundamentals of Operating Systems", Macmillan, 1975.

39. Madnick, Stuart, Donovan, John,

"Operating Systems", McGraw Hill, 1974.

40. Martin J., McClure C.,

"Software Maintenance : The Problem and its Solutions",

Englewood Cliffs, NJ : Prentice-Hall, 1983.

41. McClure C.L.,

"Managing Software Development and Maintenance",

Van Nostrand, NY, 1981.

42. Mohan C., Silberchatz A.,

"Advances in Distributed Processing Management", Vol.II

Heyden, London, 1981.

43. Musa J.D., Iannino A., Okumoto K.,
"Software Reliability : Measurement, Prediction, Application", McGraw Hill, 1987.

44. Myers, Glenford J.,
"Composite/Structured Design", New York : Van Nostrand Reinhold Co., 1978.

45. Paker Y.,
"Multi-microprocessor Systems", Academic Press, 1983.

46. Parikh G.,
Handbook of Software Maintenance,
John Wiley, 1986.

47. Patnaik L.M.,
"An Overview of Parallel Computer Architecture",
IEEE Short Course on Parallel Processing,
Hyderabad, September, 1983.

48. Patton, Peter C.,
"Multiprocessors : Architecture and Applications",
Computer, June, 1985.

49. Ramamoorthy C.V., Garg, Vijay, Prakash, Atul,
"Support for Reusability in Genesis", IEEE Transactions on Software Engineering, August, 1988.

50. Silverman J., Giddings N., Behane J.,
"An Approach to Design-for-maintenance", Proceedings Software Maintenance Workshop, R.S.Arnold (Ed.) IEEE CS Press, December, 1983.

51. Toong H.D., Gupta A.,
"An Architectural Comparison of Contemporary 16-bit Microprocessors", IEEE Micro, May, 1981.

52. Treleaven P.C.,
"Decentralized Computer Architecture", New Computer Architectures, J.Tiberghien (Ed.), Academic Press, 1984.

53. Welty L., Patton, P.C.,
"Hypercube Architectures", Proc. AFIPS., 1985.

54. Yourdon, Edward, Constantine, L.L.,
"Structured Design : Fundamentals of a Discipline of Computer Progaram and System Design", Englewood Cliffs, NJ, Prentice-Hall, 1979.

55. Microprocessor and Peripheral Hand book,
Volume I, Intel Corporation, 1989.

56. Microprocessor and Peripheral Hand book,
Volume II, Intel Corporation, 1988.

57. Oxford Dictionary of Computing,
Oxford University Press, NY, 1983.

58. Encyclopaedia of Science and Technology,
Volume 5, enf-fns,
McGraw Hill, 1977.

59. MC 6800 Application Manual,
Motorola Corporation, 1976.

-:0:-

PUBLISHED WORK OF THE AUTHOR

1. K.Poulose Jacob and C.S.Sridhar, "Multiprocessor Operation", Proceedings National Symposium, National Academy of Sciences India, 1981.

2. K.Poulose Jacob and C.S.Sridhar, "A Novel Architecture for DMA Controller", MICROSYSTEMS Architecture, Integration and Use, C.J.Van Spronsen and L. Richter (Eds.) North Holland Publishing Co., 1982.

3. K.Poulose Jacob and C.S.Sridhar, "Data Storage in Floppy - An Optimised Format", Proceedings INFORMATICS - 85, International Conference, Trivandrum, 1985

4. K.Poulose Jacob and C.S.Sridhar, "The APT System", Proceedings IEEE ACE-86, Conference, Madras, 1986.

5. K.Poulose Jacob and C.S.Sridhar, "A Software Adaptation Towards Parallel Processing", Modern Trends in Information Technology, P.V.S.Rao, P.Sadanandan (Eds.) Tata McGraw Hill, 1988.

6. K.Poulose Jacob and C.S.Sridhar,
"An Approach Towards Insight into Instruction Streams", Artificial Intelligence in Industry and Government, E. Balagurusamy (Ed.) McMillan, 1989.

7. K.Poulose Jacob and C.S.Sridhar,
"An Approach Towards Improving Processing Reliability", Proceedings SEARCC-90, Manila, Philippines, 1990.

-:0:-