ASPECTS OF CONTENTION IN CLOSELY COUPLED PROCESSOR SYSTEMS

P. M. MAINWARING-SAMWELL

Thesis presented for the degree of
Doctor of Philosophy
of the City University, London
1980

Abstract

This thesis describes the development of software and monitoring techniques for multiprocessor systems, and their implementation in a particular closely coupled microprocessor system designed for real time applications. The work attempts to reconcile the solutions of interdependent hardware and software problems arising in the design of such systems. It was particularly directed towards investigation of processor coupling, which was seen as a major contributory factor in the extent of contention between processors for the use of shared resources, and hence system degradation.

A non-invasive hardware monitor, controlled by a separate processor, was developed which has the capability of monitoring runtime sequences of millions of instructions in systems of concurrent processes. The accuracy and scale of the monitoring permits analysis of concurrent systems in real time units of microseconds.

A flexible software model of wide general applicability has been developed and programmed according to rigorous high level programming language constructs. The model has been particularly designed to allow variable processor coupling so that the effect of coupling on contention and degradation within the system can be assessed by monitoring.

Realisations of the software model were implemented on a dual F100-L processor system. The concept of optimal points (at which maximum potentially useful work can be extracted from the system) has been established and a method for determining such points developed. Monitor measurements allowed the extent of contention (both hardware and software) within the system to be assessed, and the software overhead due to synchronisation to be evaluated. As a result of these measurements, the percentage processor degradation is obtained by comparison of single and dual processor implementations of the software model and the suitability of the F100-L system for multiprocessing is considered.

CONTENTS

Acknowledgements

I would like to thank Dr. J. E. Brignell for his continual advice and encouragement during this project.

Particular thanks go to [redacted] who built the hardware monitor and gave untiring advice on hardware problems.

My colleagues [redacted] also contributed to this work through many useful discussions.

Finally, my thanks to the interdepartmental Microprocessor Laboratory of the City University for provision of research facilities.

Chapter 1

Introduction

The work on closely coupled processor systems reported in this thesis was carried out within a research group whose primary concern was computer aided measurement. Of particular interest were portable microprocessor based measurement systems such as the Roving Slave Processor described by Brignell et al (1976). Typical applications were intended to be high performance, real time signal processing problems; that is, problems requiring a relatively large, powerful, dedicated microprocessor for solution (Comley, 1978). Multiprocessing offered an obvious potential increase in processing power over uniprocessing, and an extension of the application range to include those problems amenable to solution by concurrent processing.

In closely coupled real time multiprocessor systems, hardware and software are very closely related and it is impossible to consider the design and implementation of either one in isolation from the other. Many fundamental characteristics of the system (such as coupling, contention in the use of shared resources, and resolution of potential deadlock) are a function of both hardware and software and their appropriate handling at one level relies on a correct implementation at the other level.

The author has been primarily concerned with implementation of software rather than hardware, other than in the design of a hardware monitor described in Chapter 6. Recent advances in LSI technology have been

extremely rapid, to the point where corresponding developments in software design and implementation have been outstripped. As far as microprocessors are concerned, the situation has been exacerbated by constantly decreasing hardware costs which increasingly emphasize the relative cost of software. If maximum use is to be made of the complex and powerful systems now potentially available at low cost it is of vital importance that software techniques keep pace with hardware developments.

In the consideration of a multiprocessor system certain basic problems, such as interprocessor communication and synchronisation, must be resolved. However, a measurement of the effectiveness of the solution is also necessary, to demonstrate that the implementation is indeed worthwhile and also to allow it to be refined.

The effectiveness of a multiprocessor system can only be discussed in the context of a specific software model. The design of a suitable model poses a difficult compromise, in that a balance must be achieved between models which are too general to represent any realistic application and those which are too specific to represent anything other than a single isolated problem.

Given the design of a model which adequately represents a set of real time concurrent processes, the question of a suitable programming language for it s implementation must be considered. Again this is governed by conflicting criteria. Critical real time problems require the rigorous solution achievable by high level programming language constructs while simultaneously demanding the maximum run time efficiency traditionally associated with assembly language programming. The situation is further complicated by the extremely limited range of available languages for most microprocessors.

The effectiveness of a system of processes must be measured by a technique which does not significantly disturb the system. Inevitably this rules out software monitoring. If real time systems are being considered, the monitoring facility must be capable of operating at a rate compatible with that of the system. It must also provide data which will not only allow an assessment of system performance to be made but will also enable specific system functions to be isolated, assessed and refined.

The problem then, was to implement a multimicroprocessor system and, in doing so, to establish general, suitable hardware and software design critera. Also required was a method for analysing a multiprocessor system's performance in terms of specific system characteristics and an evaluation of the hardware and software of the actual system developed.

A review of the background, theory and techniques of multiprocessing is given in Chapters 2 and 3. Chapter 4 describes a dual microprocessor system which had been developed at City University and Chapters 5 and 6 describe the software and monitoring systems developed by the author. Behaviour of the complete system and analyses, obtained by monitoring, of various system characteristics are presented in Chapter 7. The results are discussed, in the context of the current state of multiprocessing, in Chapter 8 and suggestions for further work are outlined in Chapter 9.

Chapter 2

Foundations of multiprocessing

One of the most exciting and challenging current areas of research in computing is that of realising and utilising the processing potential of microprocessors which, with their low cost, small size and limited power requirements can be applied in many areas where, previously, computer-aided problem solution was infeasible.

Multimicroprocessor systems appear particularly attractive both as an economical means of providing extensive raw processing power and as a solution to problems which either require distributed processing power or are amenable to solution by parallel processing. The appearance of this latter class of problem is expanding rapidly in fields such as industrial control and robotics.

Indeed, the real world functions by the execution of concurrent activities and many problems which have traditionally been solved by sequential algorithms may be discovered to be better suited to solution by concurrent algorithms.

The perceived advantages of multimicroprocessor systems are reflected in the progress of recent research in theoretical process structuring, related programming language design and the implementation, in hardware and software, of processors for the execution of concurrent programs.

This thesis describes both the development of software and hardware to implement a system of processes realised by concurrent programs running on closely coupled processors, and the subsequent analysis of the degradation induced in the system by implementation of the coupling and contention in the use of shared resources. Reviewed here are relevant developments prior to and during the earlier stages of the project. Several more recent areas of development (particularly Hoare, 1978 and Brinch Hansen, 1978 b, 1978 d) and their relation to and bearing upon this work are discussed in Chapter 8.

Prior to the advent of the microprocessor the majority of interest in parallelism was directed towards multiprogramming single processor computer systems (Lorin, 1972) and it is from operating system theory (Brinch Hansen, 1973 a) that many of the concepts which form a basis for this work come.

A definitive theory of co-operating sequential processes, provided by Dijkstra (1965, 1968, 1971) and introducing semaphores and their associated P and V operations as synchronisation primitives, is discussed in Chapter 3. Dijkstra also considered suitable programming language constructs for the expression of concurrency and suggested, in the guise of 'secretary processes', the idea of monitors for process structuring. These same ideas were developed by Hoare (1972), who recognised and discussed the problems inherent in real

time concurrent programming and also stressed the need for conceptual simplicity and provision of extensive compiler checking of code both to secure against time-dependent errors and to improve efficiency by minimising run-time checking. Hoare used conditional critical regions (which implement waiting for a data structure to satisfy certain conditions) to achieve synchronisation as opposed to Dijkstra's semaphores and critical regions. In a comparison of these mechanisms, Brinch Hansen (1972) considered Dijkstra's method (which implements waiting for a timing signal) to be suitable for synchronising heavily used resources and Hoare's conditional critical regions to be appropriate for more loosely coupled processes.

A number of attempts, based on graph theoretic and state machine methods, were made to formalise synchronisation mechanisms (for example, Holt, 1972; Gilbert and Chandler, 1972; Howard, 1976). Habermann (1972) used invariants to establish deadlock proofs (Chapter 5) for certain classes of problems, notably that of producers and consumers on a bounded buffer, but none of these attempts resulted in a standard approach.

Current work was consolidated by Brinch Hansen (1973 b) in a survey in which the concept of monitors was established. Monitors were formally presented by Hoare (1974) who discussed them in the context of operating system structuring.

A more theoretical view of multiprocessing was presented in a survey by Baer (1973) and in a comprehensive discussion (based on state machines) of process definition, combination and interaction by Horning and Randell (1973).

Dijkstra (1975) proposed guarded commands as a component of alternative and repetitive constructs which would allow the introduction of non deterministic program components.

The implementation of synchronisation in varying environments (and interrupt handling in this context) was discussed by Wettstein (1977) and, at a more specific level, implementations of hardware and software primitives were considered by Siewiorek (1975). Current multiprocessor organisations were surveyed by Enslow (1977).

New high level programming languages which emerged during this time were Concurrent Pascal (Brinch Hansen, 1975 and 1977) and Modula (Wirth, 1977 a and 1977 b). Concurrent Pascal was designed for programming operating systems and was directed towards either multiprogramming for single processors or for programming systems of multiple processors communicating via shared memory. It is an extension of Pascal (Wirth, 1971) having processes, classes and monitors implemented as abstract data types. The underlying philosophy of Concurrent Pascal is that simplicity, reliability and adaptability are the primary criteria upon which concurrent programs should be judged.

This same philosophy was endorsed by Wirth (1977 a and 1977 b) in his development of Modula as a language which directly addresses the problems of real time concurrent programming. Modula, which also employs the monitor concept, is essentially an implementation language designed for multiprogramming loosely coupled processes on a single processor. One of it's declared aims was to eliminate assembly language programming of, for example, peripheral device drivers and process control systems, by providing a construct (called a module) which allowed efficient and reliable programming of machine dependent tasks.

During the same time period the real time language now known as Ada (ACM, 1979) was being developed, on a much larger scale, to meet the specifications of the American Department of Defence.

From 1977 onwards a number of designs for multimicroprocessor systems emerged. Some of the earlier theoretical proposals were clearly impractical (for example, in the use of a load and store algorithm (due to Dijkstra, 1965) to implement synchronisation primitives (Caprani et al, 1977)) and those which were built frequently required substantial additional hardware (Hoener and Roedher, 1977; Swan et al, 1977) to organise shared memory. Probably the best known example was the Cm* system (Jones et al, 1977; Swan et al, 1977), a packet switched network of LSI-11 processors developed from the earlier Cmmp network of

PDP-11 minicomputers which relied on special purpose hardware for arbitration, process control and address mapping. CYBA-M (Dagless, 1977), a network of Intel 8080's, had also been proposed. Implementation details (Aspinall and Dagless, 1979; Dowsing, 1979) of this system which relied on high speed, multiport memory for interprocessor communication appeared later.

The multimicroprocessor systems which had been developed were not, however, designed for closely coupled, high performance, real time applications and little or no attention had been given to performance evaluation, analysis of degradation, or software structures beyond the provision of a simple synchronisation primitive.

Chapter 3

Theory and techniques of multiprocessing

3.0 Introduction

Many definitions of a 'process' are to be found in the literature. In this work a process is considered to be a function of a program and a processor; that is, a process is realised as the execution of a program by a processor. Concurrent processes are those processes which exist at the same time, and multiprocessing, with which we are concerned here, is the simultaneous execution of two or more programs on separate processors. Much of the theory of multiprocessing has developed from considering multiprogrammed systems implemented (usually) on a single processor. A multiprogrammed processor allocates successive time slices to the different programs which it is executing and so, if it is viewed at a level having a sufficiently coarse time grain, exhibits the same behaviour as a multiprocessor system. A typical example is that of high level language programs running under the time sharing operating system of a mainframe computer.

Theoretical considerations of the problems involved in systems of concurrent processes can largely ignore the differences between multiprocessing and multiprogramming. At the level of practical implementation however this is not so and discussion in this chapter concerns multiprocessor systems which are not multiprogrammed. Processes in such a system must both be able to communicate with one another and

be synchronised with respect to one another. Communication and synchronisation (and the implied possibility of deadlock) must be resolved at two distinct levels: the hardware level and the software level.

3.1 Process communication and synchronisation

The majority of closely coupled multiprocessor systems achieve inter-process communication via shared memory (as do multiprogrammed systems). More loosely coupled distributed multiprocessors may handle communication with input/output procedures and implement synchronisation with guarded commands (Brinch Hansen, 1978 b ; Hoare, 1978).

3.1.1 Shared memory systems

A mutual exclusion mechanism is an essential pre-requisite for systems in which processors communicate via shared memory. Without such a mechanism data in shared memory may be corrupted or incorrectly interpreted. For example, if two concurrent processes, P1 and P2, are operating on the (shared) data word A as follows:

P1	P2
A:= A+2	A:= A+3

the equivalent sequence of machine instructions will be of the form

P1	P2
fetch A	fetch A
add 2	add 3
store A	store A

If both processes fetch A before either of them stores A then the result of the operations will be incorrect and, more importantly, dependent upon the relative execution speeds of the processes which will vary from one execution to the next. Dijkstra (1965) showed that the problem could be solved, albeit very inefficiently, with load and store instructions and subsequently suggested P and V primitive operations as a means of resolving communication and synchronisation problems (Dijkstra, 1968).

P and V primitives operate on semaphores. Semaphores are non-negative integer variables (general semaphores) which may be restricted to the values zero and one (binary semaphores). The effect of a P operation on a semaphore is to reduce the semaphore's value by one as soon as the resulting value would be non-negative. A V operation increases the value of a semaphore by one. P and V operations on a binary semaphore (initialised to one) can effect mutual exclusion as shown:

P1	P2
P(S)	P(S)
A:= A+2	A:= A+3
V(S)	V(S)

The first process to execute P(S) will reduce S to zero and so prevent the second process from completing its P(S) operation (and hence gaining access to A) until the first process has executed a V(S) operation. Program sections protected by P and V in this way are called critical sections. If all code which accesses shared memory is implemented as a critical section then processes may safely communicate via shared memory.

Dijkstra also showed that processes can be synchronised by general semaphores. An example is that of producer and consumer processes on an unbounded buffer, where 'number-of-queueing-items' is a general semaphore initialised to zero and 'mutex' is a binary semaphore initialised to one.

```
     producer process
begin
     repeat  produce next item;
             P(mutex);
             add item to buffer;
             V(mutex);
             V(number-of-queueing-items)
     until   finished;
end;
```

```
     consumer process
begin
     repeat  P(number-of-queueing-items);
             P(mutex);
             take item from buffer;
             V(mutex);
             process item taken
     until   finished;
end;
```

The general semaphore can be replaced by a binary semaphore, 'delay', and an integer, 'number', both initialised to zero.

```
     producer process
begin
     repeat  produce next item;
             P(mutex);
             add item to buffer;
             number:= number + 1;
             if number = 0 then
             begin
                   V(mutex);
                   V(delay)
             end
             else  V(mutex)
     until   finished;
end;
```

```
     consumer process
begin
     repeat  P(mutex);
             number:= number-1;
             if number = -1 then
             begin
                   V(mutex);
                   P(delay);
                   P(mutex)
             end;
             take item from buffer;
             V(mutex);
             process item taken
     until   finished;
end;
```

Other similar primitives have been proposed, viz block and wakeup, lock and unlock, wait and signal.

There is an important distinction between the use of semaphores and signals (Wirth, 1977 b) as synchronising primitives. Signals (also called conditions (Hoare, 1972) and queues (Brinch Hansen, 1973 a)) have no memory associated with them but are used to trigger continuation after waiting. Semaphores can be expressed in terms of ordinary variables and signals, which leads to signals being considered the more primitive entity (Wirth, 1977 a). However, when signals are correctly used, the associated wait

statements must be subjected to conditions. This is demonstrated in Hoare (1972) where signals are expressed in terms of semaphores and ordinary variables.

3.1.2 Deadlock

Deadlock can occur in any system which utilises shared resources, for example shared memory. In such a system deadlock can occur on two levels: the hardware level, which is transparent to the programmer, and the software level which directly concerns the programmer. In dealing with deadlock at the software level the programmer must rely on problems at the hardware level having been correctly resolved.

Deadlock at the hardware level is illustrated in a system of two processors sharing memory which is distributed across their buses (Fig.3-1). The shared resources of interest here are the buses. If processor P1 attempts to access memory M2, and processor P2 attempts to access memory M1, then we have:

P1	P2
request bus 1	request bus 2
request bus 2	request bus 1
access M2	access M1
release bus 2	release bus 1
release bus 1	release bus 2

which can obviously lead to deadlock.

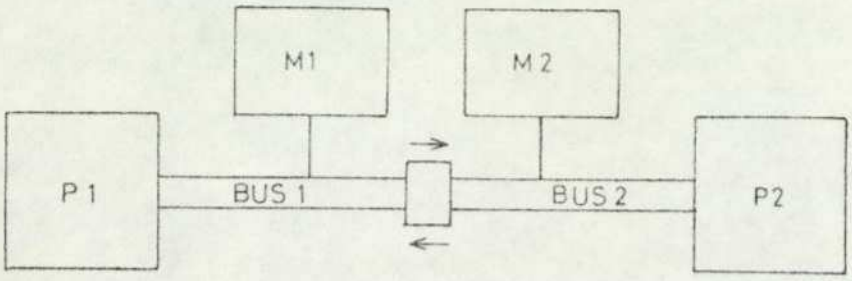


Figure 3-1 POTENTIAL DEADLOCK IN USE OF SHARED MEMORY

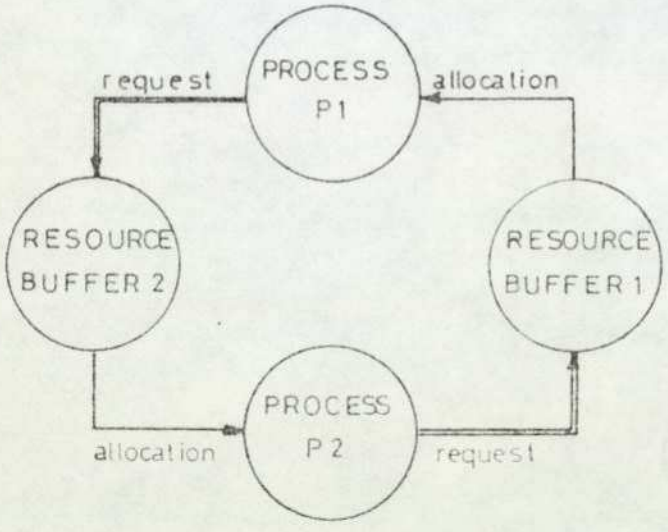


Figure 3-2 DEADLOCK IN RESOURCE ACCQUISITION

A set of processes are said to be deadlocked (Habermann, 72) when no process can proceed without accquiring a resource already held by another process in the set. This is illustrated below where processes P1 and P2 both attempt to transfer data between a pair of buffers (each protected by a separate semaphore) in shared memory.

P1

```
P(buffer1-semaphore);
P(buffer2-semaphore);
buffer2frame:= buffer1data;
V(buffer2-semaphore);
V(buffer1-semaphore);
```

P2

```
P(buffer2-semaphore);
P(buffer1-semaphore);
buffer1frame:= buffer2data;
V(buffer1-semaphore);
V(buffer2-semaphore);
```

The necessary conditions for deadlock are that resources are not simultaneously shareable and are not pre-emptable; a process must retain resources whilst accquiring further resources, and there must be a circularity in the resource requirements of processes. This is shown (Fig. 3-2) for the system above, which could deadlock if both processes executed their first P operation before either of them executed their second P operation.

The problem of deadlock can be approached by recovery, avoidance or preventive methods. Recovery methods detect deadlocks that have occurred or are about to occur and retrieve the situation by systematically destroying processes, or at least removing some of their resources, in a manner designed to disrupt the system as little as possible until the deadlock is resolved. The recovery mechanism may prove expensive to implement and such a technique may not be feasible for real time applications.

Deadlock avoidance algorithms (for example the Banker's algorithm (Dijkstra, 1968)) inspect the current state of a system to determine whether or not a request for a resource could lead to a potential deadlock. Again, such algorithms obviously involve a runtime overhead.

Preventive methods employ a static analysis of a system to ensure that a deadlock could not occur. Although there is no runtime overhead, this technique results in a system contructed from worst-case analysis.

It can be shown, however, that certain useful algorithms will not deadlock. Habermann (1972) analysed the producer/consumer algorithm (produce and consume procedures shown below) implemented with wait and signal primitives.

```
     produce
procedure deposit(d);
begin
     wait(input);
     wait(frame);
     buffer[FRONT]:=d;
     FRONT:= succ(FRONT);
     signal(message);
     signal(input)
end;
     consume
procedure accept(r);
begin
     wait(output);
     wait(message);
     REAR:= succ(REAR);
     r:=buffer[REAR];
     signal(frame);
     signal(output)
end;
```

He defined the state of synchronisation (s) of a process by the non-negative constant C[s] . The quantities nw(s), ns(s) and np(s) were defined as:

nw(s): number of times wait(s) had been executed;

ns(s): number of times signal(s) had been executed;

np(s): number of times wait(s) was passed (i.e. number of times a process was enabled to continue execution of the instruction following a wait(s)).

The effect of wait(s) was:

nw(s):=nw(s)+1:

if nw(s) < C[s] +ns(s) then np(s):=np(s)+1

and the effect of signal(s) was:

if nw(s) > C[s] +ns(s) then np(s):=np(s)+1;

ns(s):=ns(s)+1

C[s] represents a buffering facility between the number of waits and signals which have been executed.

Habermann derived the following invariant for executions of wait(s) and signal(s).

invariant

np(s) = minimum [nw(s), C[s] + ns(s)]

The producer/consumer algorithm could deadlock if producers were waiting for consumers to signal frames and consumers were waiting for producers to signal messages but the signals were never given. If no producer is going to signal a new message (np(frame) = ns(message) i.e. no producer is waiting to signal a message) and there are delayed producers (np(frame) < nw(frame)) then, from the invariant:
ns(message) = np(frame) = buffersize + ns(frame).
Similarly when no consumer is going to signal an empty frame and there are delayed consumers,
ns(frame) = np(message) = ns(message) + buffersize
As these relations cannot be true simultaneously, a deadlock cannot occur.

Habermann also showed that the buffer could not overflow or underflow, that it does not matter if a producer and consumer access the buffer simultaneously, and that all these conditions hold if the buffer is of size one.

3.1.3 Systems without shared memory

Multiprocessor systems having only non-shared memory must communicate directly with one another. For example, Brinch Hansen (1978 b) proposes the calling of common procedures, and Hoare (1978) proposes input and output commands as methods of interprocess communication. Both of these authors suggest the use of guarded regions or guarded commands (3.3) for process synchronisation. Obviously synchronism must be more tightly controlled in systems without the buffering facility of common memory.

3.2 System architectures

Concurrent processes which are multiprogrammed on a single processor can be treated in the same way as shared memory multiprocessor systems except that the problem of hardware deadlock (3.1.2) does not arise.

Multiprocessor systems may have various basic architectures: for example, the hierarchical Cm* (Swan et al, 1977 a); the ring structure of DEMOS (Dowson et al, 1979), or an array structure. The choice of architecture is determined by the envisaged

application of the system. Thus an array processor is suitable for applications where the same operation must be performed many times; hierarchical systems are suitable in cases where the memory references of a particular process are expected to be largely localised; and ring systems may be used for more generalised applications.

Resolution of deadlock problems are a matter of specific implementation. A unidirectional, packet switched ring, for example, need not deadlock over bus usage (Fig. 3-3). If, however, the ring is to be bidirectional (cf Fig. 3-1) or circuit switched (Fig. 3-4) then deadlock may occur.

Another problem to be resolved is that of arbitrating between simultaneous requests for the use of shared resources. A common method of handling this is by daisy chaining. Fig. 3-5 shows a number of devices in an F100-L processor system connected in a DMA daisy chain. If two devices are requesting a DMA transfer at the same time, the accept signal ($\overline{\text{DMAAc}}$) will be picked up by the requesting device which is electrically closest to the processor. The other requesting device will have to wait until the first DMA transfer is complete, so the daisy chain is also establishing a priority among the devices connected to it.

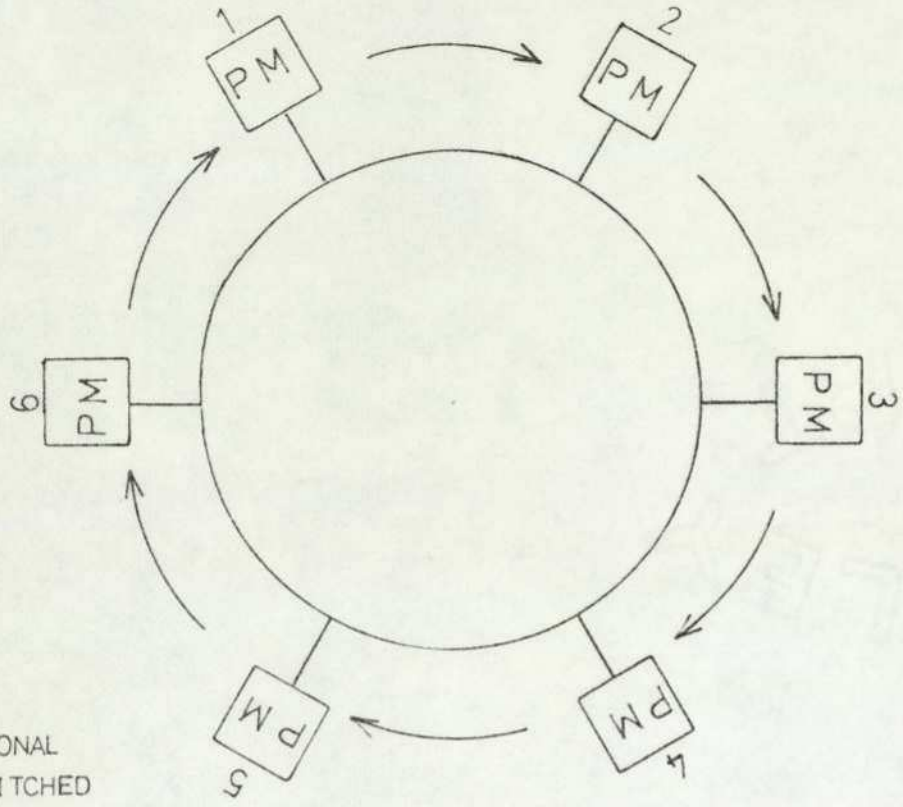


Figure 3-3
UNIDIRECTIONAL
PACKET SWITCHED
RING

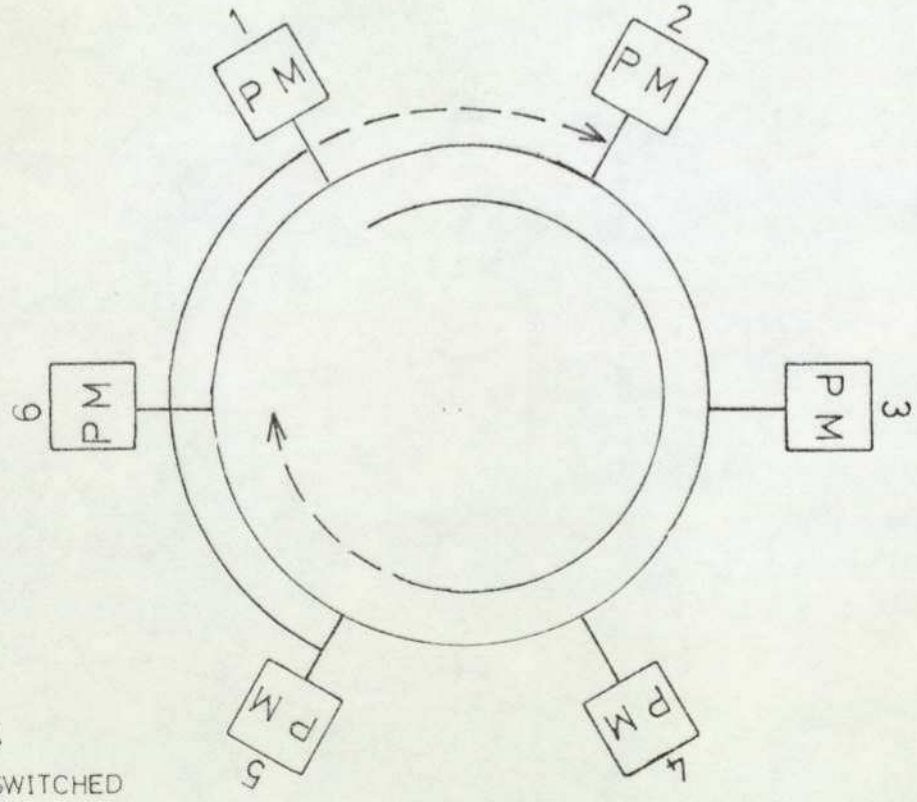


Figure 3-4
CIRCUIT SWITCHED
RING
Deadlock may occur if P1 accesses M6 as P5 accesses M2

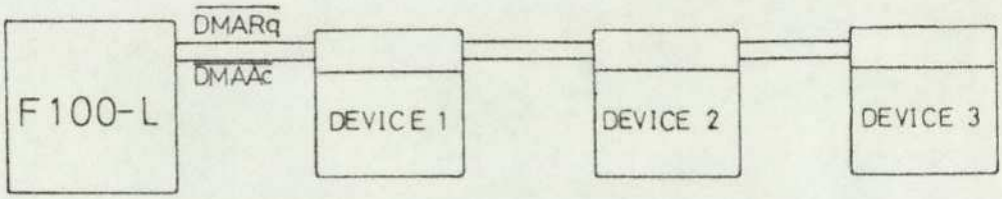


Figure 3-5 DAISY CHAIN

Other forms of arbitration, for example round robin, may impose a different discipline on devices competing for shared resources.

Distributed processors having no shared memory are frequently arranged in either a star network (with one processor acting as a central switch) or on a ring (e.g. Cambridge ring (Wilkes, 1979)). Such systems are more often termed communications networks and (particularly in the case of ring architectures) are usually packet switched.

3.3 Programming language constructs

The earliest proposals for concurrent programming facilities relied on synchronisation primitives (such as Dijkstra's P and V operations) to enforce mutual exclusion of critical sections of program from shared data areas, and a language construct (e.g. parbegin ..... parend) to permit concurrent execution of sections of program.

Hoare (1972) proposed mutually exclusive critical regions of the form

with r do C

where C is a critical region and r is a shared resource which can only be referred to from within a critical region (this was to be enforced by compiler checking). He also proposed conditional critical regions

with r when B do C

(where B is a Boolean expression) as a means of synchronising critical regions by delaying them until certain conditions are true.

A more structured concept was that of monitors (Hoare, 1974) which were adopted in Concurrent Pascal (Brinch Hansen, 1975). A monitor is an abstract data structure consisting of shared data, a set of procedures which can operate on the data, and an initialising operation. An example of this might be a bounded buffer, together with "produce data" and "consume data" procedures and an initialising operation clearing the buffer. Any process wishing to access the data structure may only do so by calling a monitor procedure (hence the term 'abstract' data structure); this is enforceable by the compiler. The monitor procedures enforce mutual exclusion amongst themselves and so the monitor is equivalent to a grouping of critical sections, together with the data they update.

In Modula (Wirth, 1977 b) synchronisation is achieved by signals together with the operations wait(signal), send(signal) and the Boolean awaited (signal). This last is a facility which allows the emission of a non-awaited signal to be trapped as an error. Communication is via shared variables with mutual exclusion of program sections accessing the shared variables guaranteed by interface modules which are similar to monitors. A module (which is a collection of constant, type, variable and procedure declarations) can define exactly the visibility of all objects within itself, and so can hide objects considered to be details of implementation but make available objects representing an intended abstraction.

Distributed processes (Brinch Hansen, 1978 b) represent a concurrent programming concept intended for multiprocessor systems which do not have shared memory. Processes can communicate only by calling common procedures defined within other processes and are synchronised by guarded regions. Guarded regions (cf guarded commands (Dijkstra, 1975)) are non-deterministic statements which can delay operations. They have the format

when B1:S1 B2:S2 ..... end

cycle B1:S1 B2:S2 ..... end

The when statement waits until one of the conditions (B) is true and executes the corresponding statement (S). The cycle statement is an endless repetition of a when statement. If several conditions (B) are true, one of the corresponding statements(S) will be (unpredictably) selected. This uncertainty is intended to reflect the non-deterministic nature of real time applications.

Similar concepts to distributed processes are suggested by Hoare (1978) whereby concurrent processes communicate by simple input and output commands and are synchronised by guarded commands.

'Tasks' are the program unit employed by Ada (ACM, 1979) for concurrent programming. A calling task (waiting to execute an entry call) must be synchronised with a called task (waiting to execute an accept statement for that entry call). When this synchronisation is achieved, a 'rendezvous' occurs. A rendezvous (which is the execution of statements between the do ..... end following the accept statement) is a critical section and during it the calling process is suspended.

```
calling task                called task
     :                           :
send(message);              entry send(message);
     :                           :
     :                      accept send(message);
     :                           do buffer:=message
     :                           end;
                                 :
                                 :
```

Ada also makes use of guarded commands as in:

when notfull ⟹ accept send .....

From the preceeding discussion it can be seen that trends within concurrent programming languages are to protect rigorously shared data structures in a manner which is enforceable by a compiler and to limit the operations that may be performed on shared data by restricting the visibility of objects within the monitor/module/task which controls the data.

3.4 Implementation of synchronisation primitives

Primitive (i.e. indivisible) operations can be implemented with suitable read/modify/write machine instructions. A read/modify/write instruction maintains control of the memory between the read and write phases so other processes cannot access the memory location being modified. 'Test-and-set' or 'exchange-memory' are the types of instruction most commonly used for implementing P operations. A P operation on a semaphore S is equivalent to:

L: if S = 1 then S:= 0 else goto L

or

when S = 1 ⟹ S:= 0

The if statement illustrates a busy waiting implementation where continual memory accesses will degrade performance. The when statement avoids this problem but, if it is to be implemented, a process must be able to 'wait' i.e. to put itself into a state where it is neither halted nor executing instructions and from which it may be 'woken' by another process.

The F100-L is an example of a processor which has a test-and-set type machine instruction but does not have a wait instruction. P(S), in a busy waiting form, may be implemented as

```
L1- JSC 0 S L2
    JMP   .L1
L2-
```

where bit 0 of location S is a semaphore. JSC (jump if set and clear) will inspect bit 0 of S and, if it is set, clear bit 0 and jump to label L2. This represents completion of the P operation. If bit 0 of S is found to be clear then the second instruction (jump to label L1) is executed and JSC repeated. V(S) may be implemented simply by setting bit 0 of S.

An important question is that of how many semaphores to utilise. If too many are used then a disproportionate amount of time may be spent in executing P and V operations. Provision of too few semaphores results in long critical sections with processes idling as they wait to gain entry to them.

In general critical sections should be kept as short as possible. The length of time a process spends in a critical section represents an equivalent potential delay in another process or processes. One question that must, therefore, be considered is that of interrupts and whether or not they should be disabled during critical sections (Wettstein, 1975).

While this is a possibility for short critical sections, it may be impractical for longer critical sections if interrupting devices require a fast response, as they are likely to in real time applications.

3.5 Applications

LSI technology has so reduced the size and cost of microcomputers that it is now practical to apply them in numbers, and to situations, that were not possible for large expensive mainframes.

Many of these new applications are naturally modelled on systems of concurrent processes (for example, the different stages of production in a manufacturing plant or the different systems operating in a ship, each of which must continue to function even if a subsystem is destroyed), and are therefore suitable candidates for control by multiprocessing.

Computerised problem solution has traditionally been (largely) achieved by sequential programming and a great deal of effort has therefore been devoted to developing sequential algorithms. However, it may be that in many cases equally suitable parallel algorithms could be developed, resulting in a further extension of the application area of multiprocessing.

Chapter 4

F100-L dual processor system

4.0 Introduction

This chapter describes a dual processor system, based on the F100-L microprocessor, developed at City University. The hardware has been described by Young (1979) and Tracey et al (1978); the software was developed by the author during this project.

4.1 Hardware

Ferranti's F100-L processor had been chosen as being suitable for the signal processing problems which were an intended application of the system for reasons which are elaborated below.

4.1.1 F100-L processor

The F100-L is a sixteen-bit, asynchronous, single chip, bipolar microprocessor manufactured using the collector diffusion isolation (CDI) process. Sixteen bits were considered to be the minimum wordlength capable of meeting the speed and accuracy requirements of real time signal processing (Reeves, 1980). Asynchronous transfers (controlled by handshakes) provide a superior operating performance over synchronous transfers by allowing the system to run at the maximum speed of memory/peripheral devices.

The processor has a single accumulator, a 7-bit condition register and a range of arithmetic, logical, shift, branch and bit-test instructions, details of which are shown in Appendix 1. Of particular interest for multiprocessing applications are the JSC (jump-if-set-and-clear) and JCS (jump-if-clear-and-set) instructions.

The address range is 32K sixteen-bit words (i.e. 64K bytes) and there are four addressing modes (direct, immediate indirect, immediate data, pointer indirect). The direct addressing range is 0 → 2K-1. Any of the first 256 locations may be used as pointers to any address in the 32K range, and this addressing mode (pointer indirect) includes an optional auto-increment/decrement of the pointer value.

There is a fully-vectored interrupt system, and DMA facilities. Multiple requests for both interrupts and DMA transfers are resolved externally by daisy chain connection. An example of an F100-L system is shown in Fig. 4-1. The special processing unit (SPU) is a feature of F100-L systems which allows special purpose hardware to be added to an F100-L to perform functions not included in the instruction set (for example, multiply/divide or fast Fourier transform). In Fig. 4-1, the SPU, peripheral device 3, and the control panel are connected into the DMA daisy chain. Peripheral devices 2 and 3 are connected into the program interrupt ($\overline{\text{PgIt}}$) daisy chain. All devices are connected to the bus by interface sets.

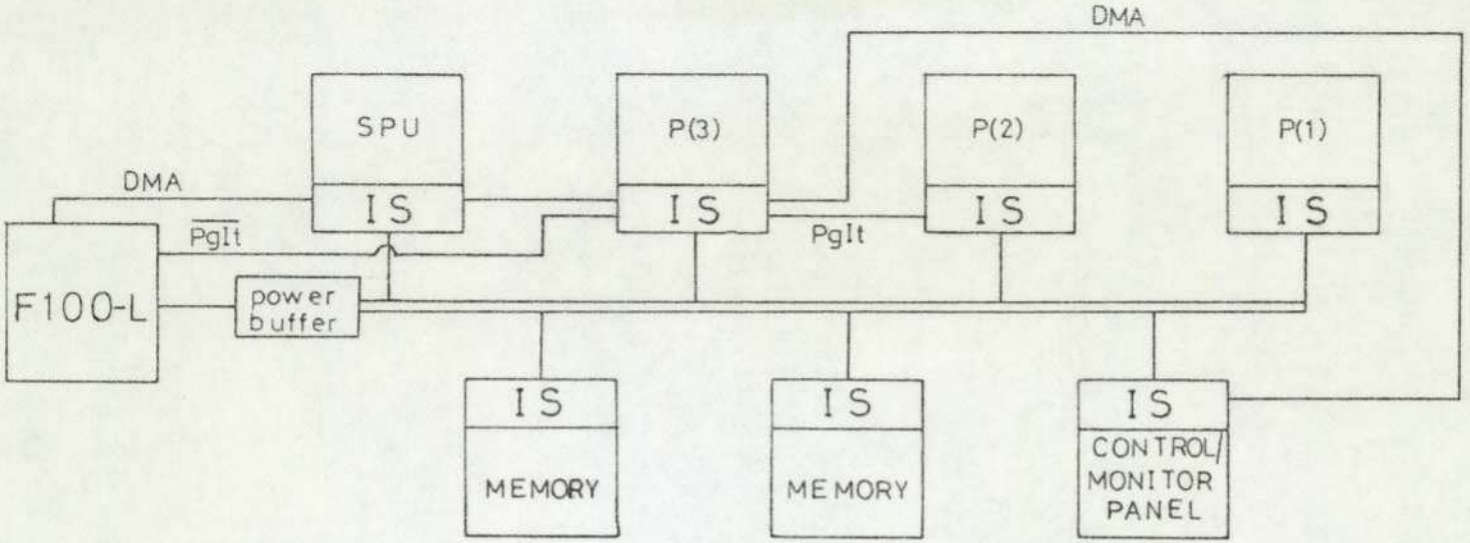


Figure 4-1 F100-L SYSTEM

4.1.2 Interface sets

An interface set (IS) comprises three LSI packages; one control chip and two eight-bit data buffers. It forms a general purpose interface by which any device may be linked to the I/O bus. By hardwiring of certain pins the IS may be configured to operate in one of five modes:

memory mode

peripheral mode

special processing unit mode

buffer mode

bus extension mode.

In memory and peripheral modes the address range or channel number respectively are selected via IS pins and the IS performs all necessary address latching and decoding operations. In peripheral mode a sixteen bit address counter is available which may be loaded by the F100-L and used during DMA transfers.

In the F100-L instruction set there is a group of instructions available to specify external functions. These instructions cause the processor to halt and are recognised and decoded by an IS in special processing unit mode and used to trigger the associated SPU. On completion of its function the SPU restarts the processor with a specially provided input ($\overline{\text{ExtFnAc}}$).

The buffer mode connection is used for bus driving purposes, while the bus extension mode uses an IS to provide a simple means of constructing multiprocessor, multibus systems. This facility was seen as an advantage in the selection of the F100-L as a suitable processor for multiprocessing.

4.1.3 Dual processor system

F100-L processor buses can be connected by means of bus extension units. A bus extension unit comprises a pair of interface sets, one configured in store mode and one configured in bus extension mode, connected 'back-to-back' (Fig. 4-2).

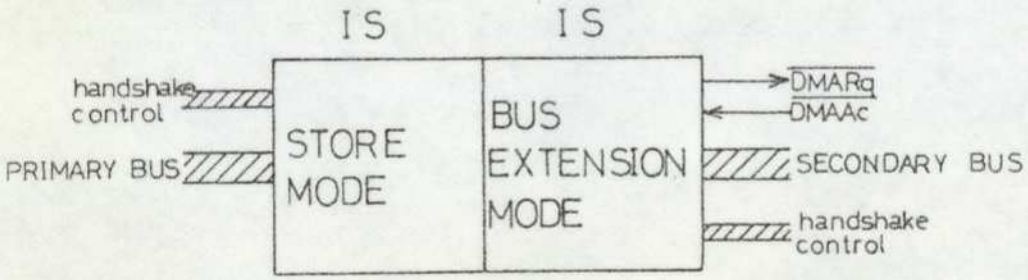


Figure 4-2 BUS EXTENSION UNIT

Buses connected by a bus extension unit are called the primary bus (store mode side) and the secondary bus (bus extension mode side). The bus extension unit provides facilities for devices connected to the primary bus to read from or write to devices connected to the secondary bus but not vice versa. That is, passage of data through the bus extension unit is bidirectional but control is unidirectional from the primary to the secondary bus.

The store mode IS is wired to decode addresses of all locations on the secondary bus which are required to be accessible from the primary bus. Such requests are passed to the bus extension mode IS and result in DMA access to the devices on the secondary bus. DMA requests are accepted at least once (during instruction decode) in each instruction cycle.

Disallowing devices on the secondary bus to access locations on the primary bus avoids the possibility of deadlock occuring with processors on each bus waiting for a DMA request to be accepted.

The dual processor system which was used in this project is shown in Fig. 4-3. There is 4K of RAM on each bus. The primary processor addresses the RAM on the primary bus as 0 - 4K and it addresses the RAM on the secondary bus as 4 - 8K. The secondary processor addresses the RAM on the secondary bus as 0 - 4K. Also on the primary bus is a teletype board

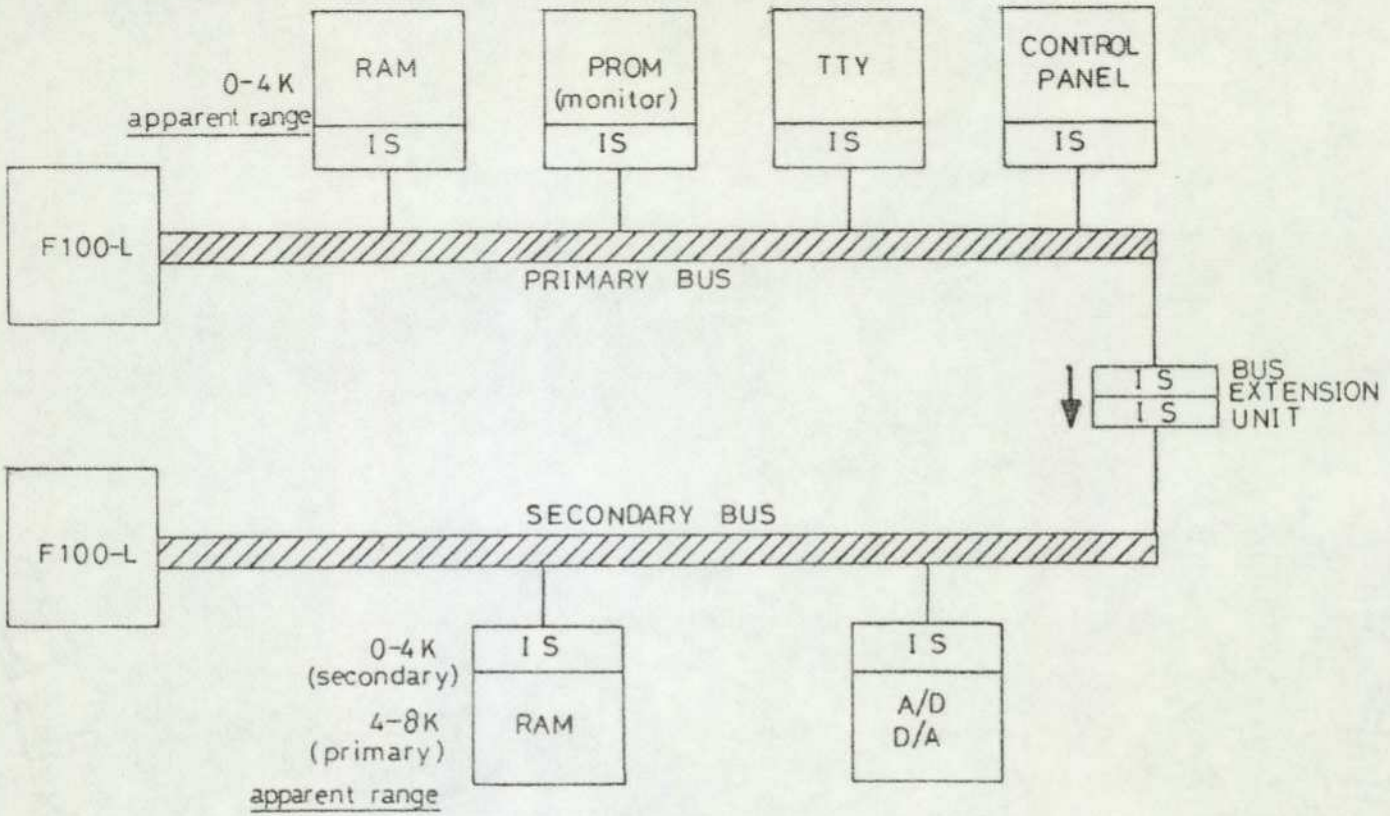


Figure 4-3 DUAL PROCESSOR SYSTEM

and 1K of PROM, starting at address 16K, holding a monitor program (4.2.2). Programs are loaded directly into memory under control of the monitor. On the secondary bus is an eight-bit analogue-to-digital/ digital-to-analogue converter board.

The secondary memory is, therefore, common to both processors. If required the secondary processor could also be provided with some private, non-shared memory by wiring the bus extension unit to allow the primary processor to address only part of the memory on the secondary bus. Fig. 4-4 shows the memory address map as it is currently configured.

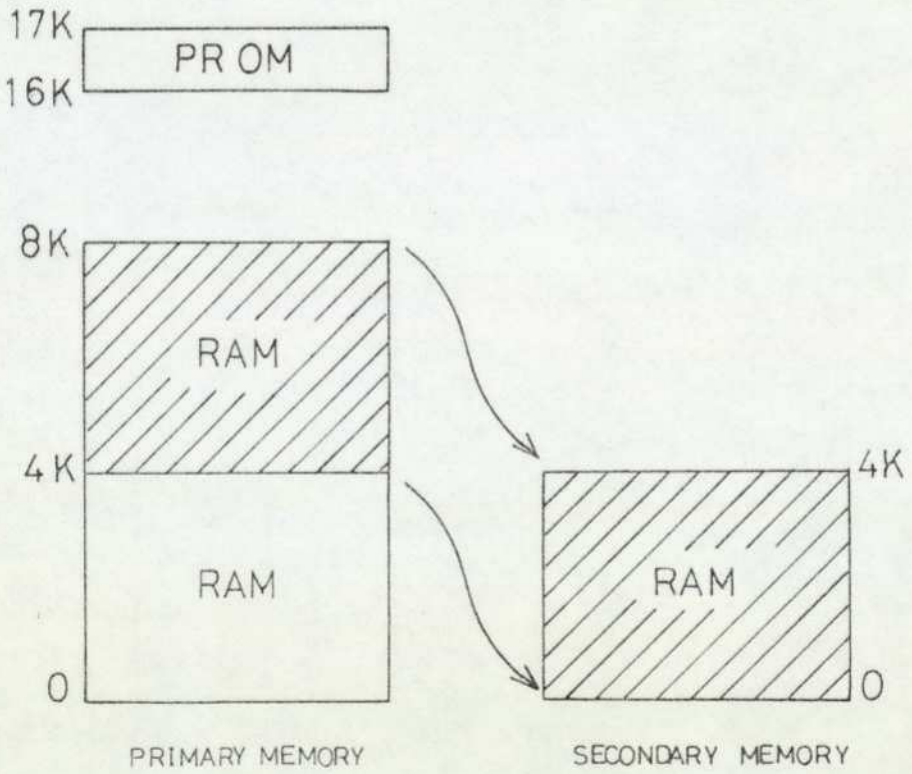


Figure 4-4 MEMORY ADDRESS MAP

4.2 Software

System software comprised cross software on an ICL 1905E system and the monitor resident in PROM.

4.2.1 Cross software

The cross software (available under the Maximop time sharing system) consisted of an assembler for F100-L assembly language, a link editor, a simulator and debug package and some library routines (Ferranti, 1976; PM150).

Programs were created as Maximop files and assembled and link edited using the cross software. The simulator and debug package proved to be of limited use because of its inability to deal with concurrent processes and real time peripheral handling.

The method adopted was to cross assemble and link edit programs and then to dump them from Maximop on papertape in F100-L loader format (see Appendix 2). The object program in loader format was then read into the dual processor system from a teletype under control of the monitor.

4.2.2 Monitor program

A controlling monitor program was written by the author to meet the particular requirements of the system. These requirements were that the monitor should be compact, that there should be comprehensive radix conversion routines (an attempt to ease the

difficult task of debugging), and that there should be a facility for displaced loading. This last requirement arose from the primary and secondary processors treating the same physical memory locations as having different addresses.

The facilities included in the monitor are: Loading of object code from paper tape (with or without displacement) in loader format; Dumping of program code in loader format on paper tape; Execution from an address (default at 2K); Copying of a block starting at address 1 and ending at address 2 (inclusively) to an area starting at address 3; Monitoring a location which may then be modified - this command also allows jumps to be made (in the general form $\pm$ n) to continue monitoring other locations.

The syntax of the commands is shown in Appendix 2 together with a program listing of the monitor.

The monitor program was burnt into PROM and located at 16K. On start up, loading of the primary processor's program counter with 16K causes the system to come under the control of the monitor.

4.3 System operation

On being reset the F100-L processors make a jump to location 2K. Both processors are reset simultaneously from the reset switch on the control panel. The control panel, being itself on the primary bus, can only monitor devices on the primary bus

directly and memory locations 0 - 4K (seen as 4 - 8K) on the secondary bus via the bus extension unit.

The method of loading programs is to load a 'jump to 2K' instruction at 2K in the secondary processor, to force it to idle, and then to cause the primary processor to jump to 16K (either by putting a 'jump to 16K' at 2K or by setting the program counter to 16K) where it enters the monitor program. Programs may then be loaded into memory from paper tape and edited by the monitor.

When the programs are ready to run the processors are halted, jumps to the correct start addresses are loaded at 2K in each processor and the processors are reset.

Areas of memory may be dumped (for subsequent reloading) in loader format on paper tape from the monitor program.

Chapter 5

Software model

5.0 Introduction

The performance of a multiprocessor system can only be sensibly discussed in the context of specific sets of programs as it is the execution of such programs by the processors which realises the concurrent processes whose behaviour is under consideration. The selection of suitable sets of programs (termed 'software models') and their subsequent implementation is the subject of this chapter.

5.1 Choice of model

The chosen models had to be of direct relevance to the envisaged high performance, real time applications of the particular hardware in question while at the same time being of sufficiently wide interest to enable useful inferences to be made about other related and/or more general systems. The models had also to be flexible enough to allow process characteristics which were thought to directly affect system performance to be varied in a systematic manner and, of prime importance in view of the emphasis placed on reliability in this work, the algorithms underlying the processes had to be rigorously defined.

5.2 Choice of algorithm

It was decided to develop models based upon the producer/consumer algorithm (3.1) for a bounded buffer. This algorithm had the advantages of great generality and of being extremely well understood, (including being known not to deadlock when correctly

implemented (3.1.2)). The basic model was viewed as a pair of concurrent processes communicating via two bounded buffers in shared memory and synchronised by their use of the buffers with each process acting as a producer for one buffer and a consumer for the other buffer (Fig. 5-1).

Within this symmetrical framework, the model could be tailored to specific requirements by appropriate definition of 'process 1' and 'process 2'.

For the purposes of this project the processes were defined as being a filter process and an i/o process, so realising a digital filter.

5.3 Coupling and flexibility

The ability to vary the coupling of the processes was essential as closeness of coupling was expected to be a major factor affecting contention within the system. Process coupling in the model was determined in part by the frequency with which data was produced for, or consumed from, the common buffers. This could be controlled by expanding or contracting the sections of processes 1 and 2 which were not involved in use of the buffer. A second factor affecting coupling was the degree of mutual exclusion to be introduced on implementation of the system of processes. If the processes were mutually excluded from accessing all of the shared memory this implied a closer degree of coupling than if they were mutually excluded from accessing each separate buffer which in turn implied a closer degree of coupling than enforcement of mutual exclusion only on subsections of each buffer.

Additional flexibility was required of the model in the sense of process activity other than inter-process communication and synchronisation. The i/o process was adaptable from the simplest case (in which input would be achieved by looping on a status flag until data was available and then reading the data from an address,and output would be achieved simply by writing to an address) to a more complex case in which input and output would be by DMA. The producer/consumer algorithm can be adapted to cope with the uncertainty introduced by DMA block transfer (Mainwaring-Samwell and Brignell, 1978) and the buffer size could be adjusted to permit DMA bursts of a size suitable to any particular application (5.5.3.4).

Variable buffer size also gave flexibility to the filter process in permitting the filter to be of a type requiring historical samples. It was considered that the actual form of the filter should, in the first instance, be extremely simple to avoid the possibility of characteristics peculiar to a particular filter influencing the behaviour of the model in general. However, the framework of the model had to accommodate both non-recursive and recursive filters of the form respectively

$$y_i = f(x_i, x_{i-j}) \text{ and } y_i = f(x_{i-j}, y_{i-k})$$
$$\text{with } o \leqslant j \leqslant n, \quad o \leqslant k \leqslant n$$

requiring the historical samples x_{i-j} or x_{i-j} and y_{i-k}

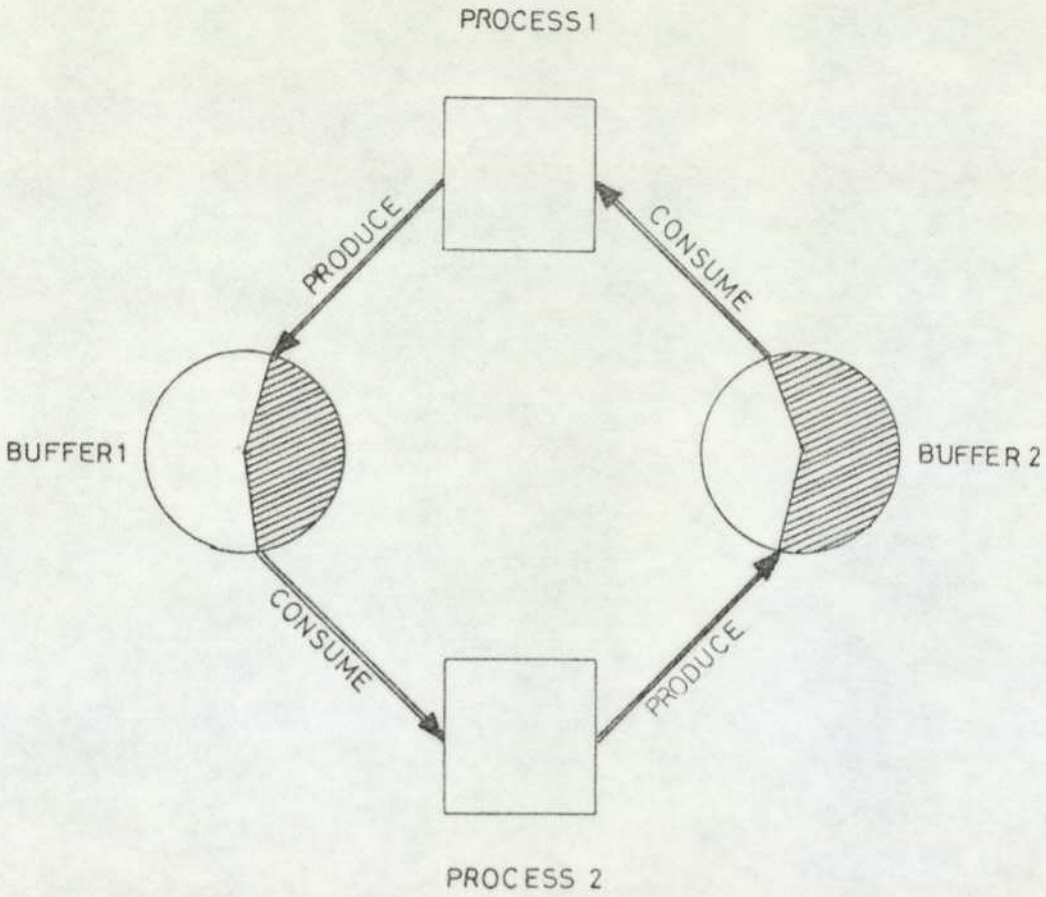


Figure 5-1 GENERAL SOFTWARE MODEL

This was, in fact, only equivalent to having an extra buffering capacity of n and so presented no problems.

Thus the model of Fig. 5-1 fulfils the criteria of generality and flexibility and exhibits a pleasing logical simplicity and symmetry.

5.4 Design of models

Within the framework described above, an i/o process and a filter process were designed as shown in Fig. 5-2. The i/o process was to accept, as input, regular samples of a known waveform from an adc/dac, to which output was also to be passed, and the filter routine was to be a simple inversion of the data (to demonstrate that data was actually being filtered). Counts i and f were variable delay loops whose purpose was to control the closeness of coupling of the processes.

These processes were used to provide three models, the first of which (model 1) was to be implemented on a single processor for comparative purposes. Model 1 is illustrated in Fig. 5-3.

Model 2 (Fig. 5-2) was to be implemented on a dual processor with the processes mutually excluded from each buffer separately. Model 3 was identical to model 2 except that mutual exclusion was to be applied to the combined buffers.

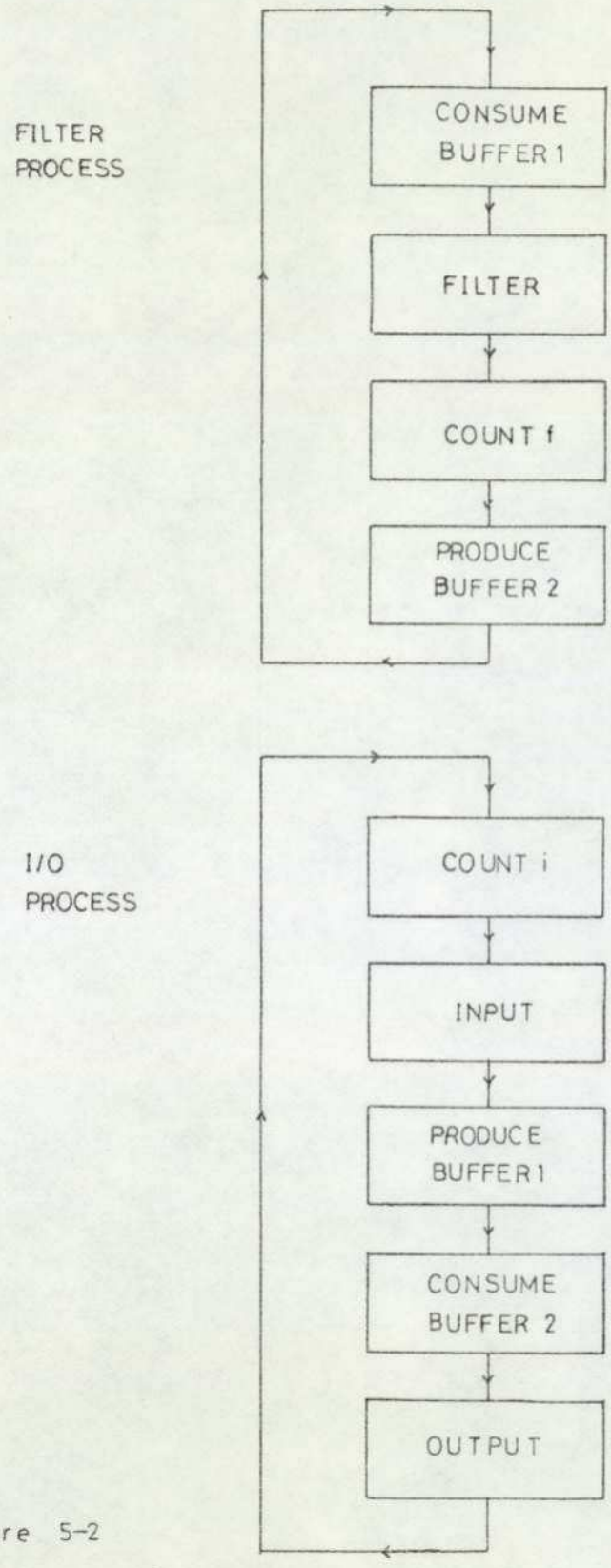

FILTER
PROCESS
CONSUME
BUFFER 1
FILTER
COUNT f
PRODUCE
BUFFER 2
I/O
PROCESS
COUNT i
INPUT
PRODUCE
BUFFER 1
CONSUME
BUFFER 2
OUTPUT

Figure 5-2

MODEL 1

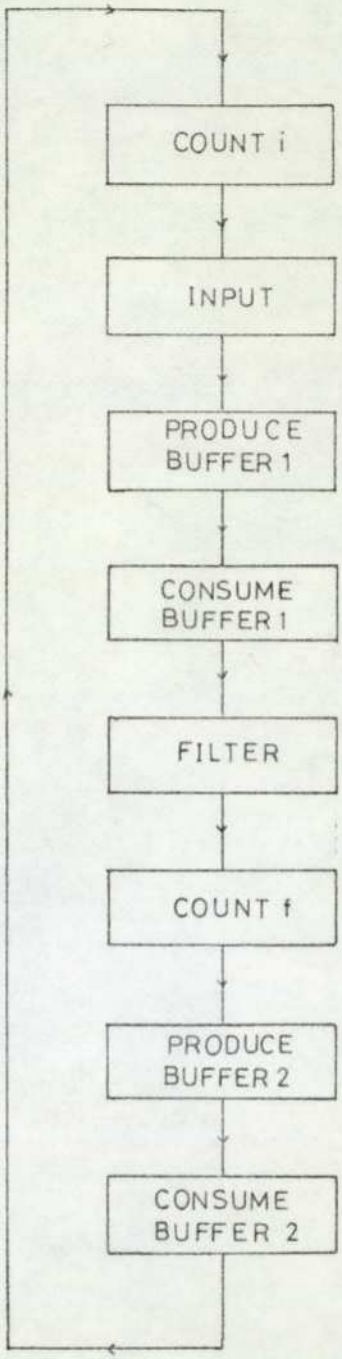


Figure 5-3

Obviously in model 1 there was no mutual exclusion requirement and, in fact, in the absence of DMA transfers or a historical filter, no buffering requirement. However, as the purpose of the model was to provide a comparison between the performance of a dual processor in executing a set of programs with the performance of a single processor achieving the same task, only the implementation of mutual exclusion was omitted. Obviously it could be argued that, for comparative purposes, model 1 could be shorter. A simpler program was, in fact, written at one stage but was discarded because it was felt to be inappropriate in that the models were designed to be extended to the general case (that is, including DMA transfers and complex filters) and this would have re-introduced the buffering requirement. It was also against the general philosophy of the project to make specific code optimisations for special cases, the special case in this instance being the extreme simplicity of model 1. Such simplicity allows the logical correctness of the model to be determined by inspection, but this would not be true of more complicated models.

5.5 Realisation

5.5.1 Programming language considerations

Integrity was considered to be of paramount importance in this system, a fact which had to be reconciled with the lack of high level language programming facilities. In the absence of a suitable compiler it was decided to write the programs in a Concurrent Pascal type notation and then to code them into F100-L assembly language. The logical clarity which resulted from this use of appropriate high level language constructs to implement algorithms of proven integrity was relied upon to generate dependable logically clean, adaptable programs. This technique has obvious drawbacks but it was found by the author to be extremely valuable as a means of producing, with confidence and comparative ease, logically complex programs.

Although efficiency must be a prime consideration in a real time system which is to be driven close to its maximum attainable speed, it was at all times considered to be secondary to reliability on the premise that an unreliable system is, for practical purposes, useless. The unavoidable software overhead incurred in ensuring the integrity of the system was minimised only to a point at which it did not compromise program reliability and simplicity. This approach precluded the use of any 'clever' programming tricks derived from peculiarities of the F100-L machine code and so the program quality should be reproducible by a good compiler.

5.5.2 Program design

The buffers of Fig. 5-1 were implemented as Concurrent Pascal monitors (3.3), that is, as a shared data structure with a set of procedures to operate on it, and an initialisation routine.

```
type buffer =
monitor
var data,frames:available;
      pointer1,pointer2:index;
      producer,consumer:queue;
      contents:buff;

procedure entry consume(var sample:byte);
begin
     if data = 0 then delay(consumer);
     data:=data-1;
     sample:=contents[pointer1];
     pointer1:=succ(pointer1);
     continue(producer);
end;

procedure entry produce(sample:byte);
begin
     if frames = 0 then delay(producer);
     frames:=frames-1;
     contents[pointer2]:=sample;
     pointer2:=succ(pointer2);
     continue(consumer);
end;
```

```
begin data:=0;
      frames:=buffersize;
      pointer1:=pointer2:=1
end;
```

The shared data structure defined by the monitor consists of

(1) a cyclic buffer whose 'contents' are of type buff;

```
type buff = array (.1.. buffersize.) of byte;
```

(2) restricted integer variables 'data' and 'frame', of type 'available', which define the number of available data samples and empty frames for a buffer;

```
type available = 0..buffersize;
```

(3) restricted integer variables 'pointer1' and 'pointer2' of type 'index', which point to the next available data sample and the next available empty frame respectively;

```
type index = 1..buffersize;
```

(4) variables 'producer' and 'consumer' of (standard) type 'queue' which are used to delay the producer and consumer processes until frames or data are respectively available.

The two procedures, 'consume' and 'produce', of the monitor are non-local (denoted by entry) i.e. they may be called from outside the monitor. 'Consume' delays the calling process until data is available, returns a sample to the process and then continues the

execution of any producer process that is waiting for frames. 'Produce' delays the calling process until frames are available, puts a sample into the buffer and continues the execution of any consumer process that is waiting for data. 'Succ' is a cyclic successor function.

The initial statement of the monitor sets the number of available data samples to zero, the number of empty frames to the constant 'buffersize' and the buffer pointers to the start of the buffer.

Two instances of the buffer can be declared and initialised as

```
var buffer1,buffer2:buffer,
init buffer1,buffer2
```

The two processes of Fig. 5-1 were declared as:

```
type ioprocess =
process(buffer1,buffer2:buffer);
var sample:byte;
begin
     cycle
          counti;
          input(sample);
          buffer1.produce(sample);
          buffer2.consume(sample);
          output(sample)
          until nosamples;
     end;
end;
```

and

```
type filterprocess =
process(bufferl,buffer2:buffer);
var sample:byte;
begin
     cycle
          bufferl.consume(sample);
          digitalfilter(sample);
          countf;
          buffer2.produce(sample)
          until nosamples;
     end;
end;
```

where 'counti' and 'countf' are delaying processes to alter the coupling, 'digitalfilter' is the actual filtering process and 'input' and 'output' are the peripheral device processes.

Instances of 'ioprocess' and 'filterprocess' can be declared and initialised as

```
var inout:ioprocess;
    filter:filterprocess;
init inout(bufferl,buffer2);
     filter(bufferl,buffer2);
```

As ordered listing of the program is shown in Appendix 3.

5.5.3 Implementation

The implemented programs input and output regular, single data samples under program control as described in 5.5.3.1 to 5.5.3.3.

This section is extended (5.5.3.4) to cover the case of irregular bursts of input and output data handled by DMA.

5.5.3.1 Monitors

Mutual exclusion was achieved by associating a binary semaphore with a buffer(s) and its associated variables ('data', 'frames', 'pointer1' and 'pointer2'). Code accessing a buffer or its variables was written as a critical section i.e. it was enclosed by P and V operations on the appropriate semaphore. In cases where mutual exclusion had to be relinquished because a process was delayed in a critical section, a V operation was performed and a branch made back to the beginning of the critical section. Thus 'delay' was implemented by polling on a shared variable ('data' or 'frames') and 'continue' by updating a shared variable (equivalent to the statements 'frames:= frames + 1' and 'data:= data + 1' in 'consume' and 'produce' respectively).

P and V primitives were coded as JSC (jump-if-set-and-clear) and SET instructions respectively. Where bit 0 of location S is a binary semaphore, P is:

```
L1 - JSC  0  S  L2
     JMP     .L1
L2 -
```

and V is:

```
     SET  0  S
```

The cyclic successor function, succ, was realised by pointer addressing of the buffers, together with a base address/buffer size check for the end-of-buffer condition.

5.5.3.2 I/O process

The delay, 'counti', introduced to control process coupling was implemented as a simple loop of (variable) size n:

```
        LDA     ,-n
        STO     X       (X = -n
L -     ICZ  X  L       (increment X until zero
                        (then continue
```

Input was achieved by inspecting a status bit, (data available), before reading a data sample from a specified location. The location was automatically cleared by the read. Data was output by writing to a specified location. In the F100-L programs (see 5.6), when data was known to be available, a critical section was entered and the presence of empty buffer frames checked before the data was actually read. In the absence of empty frames the critical section was exited from and subsequently re-entered. This was more efficient (with a single accumulator machine and input samples that were cleared by reading) than reading the data immediately it became available and having to store it again.

In the output routine, if no data was available for output, the program returned to input another sample rather than waiting for output data to become available. Obviously this path could only be taken a maximum of 'buffer size' times before the input buffer became full and the system broke down. There are two alternatives to this. One is to force the program to input and output samples alternately, but this defeats the purpose of having buffers (other than for holding a fixed number of samples for historical filters). The second alternative is to permit the input buffer to fill up but to provide a recovery mechanism whereby it is emptied again. For closely coupled processes there may be a considerable overhead associated with such a mechanism in that the program must decide when it can safely perform multiple output routines. Buffer fullness can only fluctuate (in a non-idling system) if either the input samples occur at irregular intervals or some program component (typically the filter routine) is of varying length. Under these conditions the program must choose suitable points at which to perform an 'extra' output (for example, when it knows it is taking a comparatively short path through a variable length routine).

As models 2 and 3 were sampling at a regular rate and the process relating input to output (i.e. the filter process) was of regular length it was not

worthwhile to implement a recovery mechanism such as that described above. Neither was there anything to be gained from forcing input and output to alternate strictly (although they would normally do so) when the buffers could prevent a system breakdown which might occur from a very occasional variation in program length.

5.5.3.3 Filter process

The delay 'countf' was implemented by a simple loop as described for counti (5.5.3.2). 'Digitalfilter' was simply an inversion of the current sample. This demonstrated visually that data was actually being filtered but had no characteristics which could affect a model's behaviour and was (in terms of program instructions) of regular length.

5.5.3.4 I/O by DMA

If input and output were to be performed (in the i/o process) by DMA, certain program modifications would be necessary. DMA transfers would be initiated by the program writing a positive non-zero value to a specified location. The value, which would represent the number of bytes to be transfered, would be automatically decremented. When it became zero an interrupt to the i/o processor would be generated. The transfer address would also be specified by program but incremented automatically by hardware, so an interrupt would also occur at the 'end of buffer' to allow the address pointer to be reset.

Details of such an implementation are discussed in Mainwaring-Samwell and Brignell (Appendix 6). The essential difference from the models discussed above is in the handling of the variables representing number of empty frames (NOEF) and number of queueing data (NOQD) for a particular buffer. If we consider a producer process, when a DMA transfer of n bytes is initiated then NOEF must be decreased by n i.e. n empty frames have been 'accounted for'. However, NOQD cannot be increased by n as the transfers are not known to have taken place i.e. the invariant

NOQD + NOEF = buffer size

of models 2 and 3 no longer holds. It is only when the relevant interrupt occurs that the transfers are known to have taken place and NOQD can be increased. The interrupt routine would perform the following tasks:

(1) increase NOQD i.e. signal(completed-data-transfers)

(2) reset DMA address pointer if it is at end of buffer

(3) calculate the size of the next DMA burst as the minimum of NOEF and (end-of-buffer address — address pointer)

(4) initiate DMA transfer

5.6 F100-L programs

Programs for model 1 (single processor) and models 2 and 3 (differing only in bit 0/bit 1 of S being used as a semaphore) are shown overleaf.

```
#MASTER MODEL1
#LOWER  /2000
        IPNM1,OPNM1,G,X
#UPPER  /3570
        S,INOQD,INOEF,INBUFF/128,
        ONOQD,ONOEF,OUTBUFF/128,
        F,N
#PROGRAM/2080
#COMMENT   INITIALISE  #
L1-  LDA  ,0
     STO  .F                (INITIALISATION FLAG
     LDA  ,128
     STO  .N                (BUFFER SIZE
L2-  LDA  ,INBUFF-1
     STO  33                (INPUT POINTER
     STO  35
     ADD  .N
     STO  IPNM1             (END-OF-BUFFER ADDRESS
L3-  LDA  ,OUTBUFF-1
     STO  34                (OUTPUT POINTER
     STO  36
     ADD  .N
     STO  OPNM1             (END-OF-BUFFER ADDRESS
#COMMENT  INITIALISE MONITORS  #
L4-  LDA  ,0                (INPUT BUFFER
     STO  .INOQD            (NO OF QUEUEING DATA SAMPLES
     LDA  .N
     STO  .INOEF            (NO OF EMPTY FRAMES
L5-  LDA  ,0                (OUTPUT BUFFER
     STO  .ONOQD            (NO OF QUEUEING DATA SAMPLES
     LDA  .N
     STO  ONOEF             (NO OF EMPTY FRAMES
#COMMENT  END OF INITIALISE  #
#COMMENT  MAIN LOOP          #
#COMMENT   COUNT  I          #
L8-  LDA  ,-1
     STO  X
L85- ICZ  X  L85
#COMMENT  INPUT(SAMPLE)   #
L87- LDA .8965              (DATA AVAILABLE?
     JBC  15  L87           (LOOP IF NOT
#COMMENT  BUFFER1.PRODUCE(SAMPLE)  #
L10- JZE  .INOEF  L10       (BRANCH IF NO FRAAMES
     LDA  .8964             (READ SAMPLE
     STO  /33+              (STORE IN BUFFER
     LDA  ,1
     SBS  .INOEF            (FRAMES:=FRAMES-1
     ADS  .INOQD            (CONTINUE(CONSUMER)
L11- LDA  33                (CHECK FOR
     SUB  IPNM1             (END OF BUFFFER
     JBC  Z  L23
     LDA  ,INBUFF-1
     STO  33
#COMMENT  BUFFER1.CONSUME(SAMPLE)  #
L23- JZE  .INOQD  L15 .     (BRANCH IF NO DATA
     LDA  ,1
```

```
      ADS   .INOEF           (DATA:=DATA-1
      SBS   .INOQD           (CONTINUE(PRODUCER)
      LDA   /35+             (GET DATA SAMPLE
#COMMENT   DIGITALFILTER(SAMPLE)  #
F1-   NEQ   ,*377            (INVERT
      STO   G
#COMMENT    COUNT    #
      LDA   ,-1
      STO   X
F2-   ICZ   X  F2
#COMMENT   BUFFER2.PRODUCE(SAMPLE)  #
L26-  JZE   .ONOEF  L26      (BRANCH IF NO FRAMES
      LDA   G                (GET SAMPLE
      STO   /36+             (STORE IN BUFFER
      LDA   ,1
      SBS   .ONOEF           (FRAMES:=FRAMES-1
      ADS   .ONOQD           (CONTINUE(CONSUMER)
L28-  LDA   35               (CHECK FOR END
      SUB   IPNM1            (OF BOTH
      JBC   Z  L15           (BUFFERS
      LDA   ,INBUFF-1
      STO   35
      LDA   ,OUTBUFF-1
      STO   36
#COMMENT   BUFFER2.CONSUME(SAMPLE)  #
L15-  JZE   .ONOQD  L8      (BRANCH IF NO DATA
      LDA   ,1
      SBS   .ONOQD           (DATA:=DATA-1
      ADS   .ONOEF           (CONTINUE(PRODUCER)
      LDA   /34+             (GET DATA SAMPLE
L16-  STO   .8964            (WRITE
L17-  LDA   34               (CHECK FOR
      SUB   OPNM1            (END OF BUFFER
      JBC   Z  L8
      LDA   ,OUTBUFF-1
      STO   34
      JMP   .L8
#END
#FINISH
11-17-53_
```

```
$MASTER FILTER
$LOWER  /2000
        IPNM1,G,X
$UPPER  /7666
        S,INOQD,INOEF,INBUFF/128,
        ONOQD,ONOEF,OUTBUFF/128,
        F,N
$PROGRAM/2080
$COMMENT  INITIALISE  $
L20- LDA  ,INBUFF-1
     STO  35              (INPUT POINTER
     LDA  ,OUTBUFF-1
     STO  36              (OUTPUT POINTER
L21- JBC  0  F  L21       (WAIT FOR I/O INITIALISE
     LDA  ,INBUFF-1
     ADD  .N
     STO  IPNM1           (END-OF-BUFFER ADDRESS
$COMMENT  END OF INITIALISE          $
$COMMENT  MAIN LOOP                  $
$COMMENT  BUFFER1.CONSUME(SAMPLE)    $
L22- JSC  1  S  L23       (P(S1)
     JMP  .L22
L23- JZE  .INOQD  L30     (BRANCH IF NO DATA
     LDA  ,1
     ADS  .INOEF          (DATA:=DATA-1
     SBS  .INOQD          (CONTINUE(PRODUCER)
     LDA  /35+            (GET DATA SAMPLE
     SET  1  S            (V(S1)
$COMMENT  DIGITALFILTER(SAMPLE)  $
F1-  NEQ  ,*377           (INVERT
     STO  G
$COMMENT   COUNT   $
     LDA  ,-1
     STO  X
F2-  ICZ  X  F2
$COMMENT  BUFFER2.PRODUCE(SAMPLE)  $
L25- JSC  0  S  L26       (P(S0)
     JMP  .L25
L26- JZE  .ONOEF  L31     (BRANCH IF NO FRAMES
     LDA  G               (GET SAMPLE
     STO  /36+            (STORE IN BUFFER
     LDA  ,1
     SBS  .ONOEF          (FRAMES:=FRAMES-1
     ADS  .ONOQD          (CONTINUE(CONSUMER)
     SET  0  S            (V(S0)
L28- LDA  35              (CHECK FOR END
     SUB  IPNM1           (OF BOTH
     JBC  Z  L22          (BUFFERS
     LDA  ,INBUFF-1
     STO  35
     LDA  ,OUTBUFF-1
     STO  36
L29- JMP  .L22
L30- SET  1  S            (V(S1)
     JMP  .L22            (SHORT LOOP
L31- SET  0  S            (V(S1)
     JMP  .L25            (SHORT LOOP
$END
$FINISH
18-51-16_
```

```
#MASTER IO
#LOWER  /2000
        IPNM1,OPNM1,X
#UPPER  /3570
        S,INOQD,INOEF,INBUFF/128,
        ONOQD,ONOEF,OUTBUFF/128,
        F,N
#PROGRAM/2080
#COMMENT   INITIALISE  #
L1-  LDA  ,0
     STO  .F              (INITIALISATION FLAG
     LDA  ,128
     STO  .N              (BUFFER SIZE
L2-  LDA  ,INBUFF-1
     STO  33              (INPUT POINTER
     ADD  .N
     STO  IPNM1           (END-OF-BUFFER ADDRESS
L3-  LDA  ,OUTBUFF-1
     STO  34              (OUTPUT POINTER
     ADD  .N
     STO  OPNM1           (END-OF-BUFFER ADDRESS
#COMMENT  INITIALISE MONITORS  #
L4-  LDA  ,0              (INPUT BUFFER
     STO  .INOQD          (NO OF QUEUEING DATA SAMPLES
     LDA  .N
     STO  .INOEF          (NO OF EMPTY FRAMES
     SET  1  S            (INITIALISE SEMAPHORE
L5-  LDA  ,0              (OUTPUT BUFFER
     STO  .ONOQD          (NO OF QUEUEING DATA SAMPLES
     LDA  .N
     STO  ONOEF           (NO OF EMPTY FRAMES
     SET  0  S            (INITIALISE SEMAPHORE
     SET  0  F            (SIGNAL I/O INITIALISED
#COMMENT  END OF INITIALISE  #
#COMMENT  MAIN LOOP          #
#COMMENT   COUNT  I          #
L8-  LDA  ,-1
     STO  X
L85- ICZ  X  L85
#COMMENT  INPUT(SAMPLE)   #
L87- LDA .8965            (DATA AVAILABLE?
     JBC  15  L87         (LOOP IF NOT
#COMMENT  BUFFER1.PRODUCE(SAMPLE)  #
L9-  JSC  1  S  L10       (P(S1)
     JMP  .L9
L10- JZE  .INOEF  L12     (BRANCH IF NO FRAAMES
     LDA  .8964           (READ SAMPLE
     STO  /33+            (STORE IN BUFFER
     LDA  ,1
     SBS  .INOEF          (FRAMES:=FRAMES-1
     ADS  .INOQD          (CONTINUE(CONSUMER)
     SET  1  S            (V(S1)
L11- LDA  33              (CHECK FOR
     SUB  IPNM1           (END OF BUFFFER
     JBC  Z  L14
     LDA  ,INBUFF-1
```

```
      STO   33
      JMP   .L14
L12-  SET   1  S              (V(IS)
      JMP   .L9               (SHORT LOOP
#COMMENT  BUFFER2.CONSUME(SAMPLE)  #
L14-  JSC   0  S  L15         (P(SO)
      JMP   .L14
L15-  JZE   .ONOQD  L18       (BRANCH IF NO DATA
      LDA   ,1
      SBS   .ONOQD            (DATA:=DATA-1
      ADS   .ONOEF            (CONTINUE(PRODUCER)
      LDA   /34+              (GET DATA SAMPLE
      SET   0  S              (V(SO)
L16-  STO   .8964             (WRITE
L17-  LDA   34                (CHECK FOR
      SUB   OPNM1             (END OF BUFFER
      JBC   Z  L8
      LDA   ,OUTBUFF-1
      STO   34
      JMP   .L8
L18-  SET   0  S              (V(SO)
      JMP   .L8               (SHORT LOOP
#END
#FINISH
18-54-49_
```

Chapter 6

Experimental system

6.0 Introduction

The experimental system was to consist of a set of concurrent processes and a non-invasive monitor which could provide information about the processes' behaviour without in any way influencing it. The concurrent processes were realised by the software models discussed in Chapter 5 running on the dual processor system described in Chapter 4.

6.1 Measurements required

Two categories of measurement were required. Firstly, maximum sampling rates at which the different models could operate so that their overall performance could be compared. Secondly, traces of instruction execution sequences to establish precisely the dynamic behaviour of the models and so account for the expected differences in optimal (maximum) sampling rates.

6.2 Optimal sampling rates

Measurement of optimal sampling rates (described in 7.1) was achieved as shown in Fig. 6-1. Input to the system was a series of samples of a triangular waveform (typically of frequency 120 Hz and amplitude in the range 0 - 2.4 volts) clocked by a square wave signal. Output from the system was the inverted input signal which was displayed, along with the input signal, on an oscilloscope. A method of determining the sampling rate for which the processes ceased to function properly was devised by inspecting

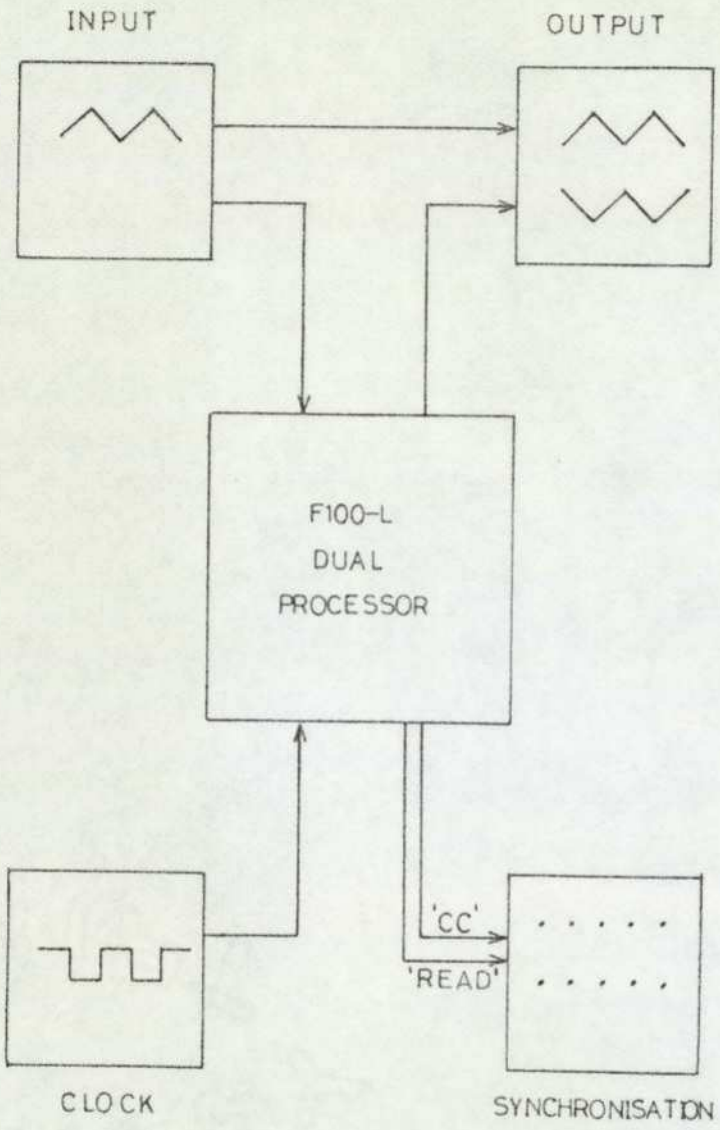
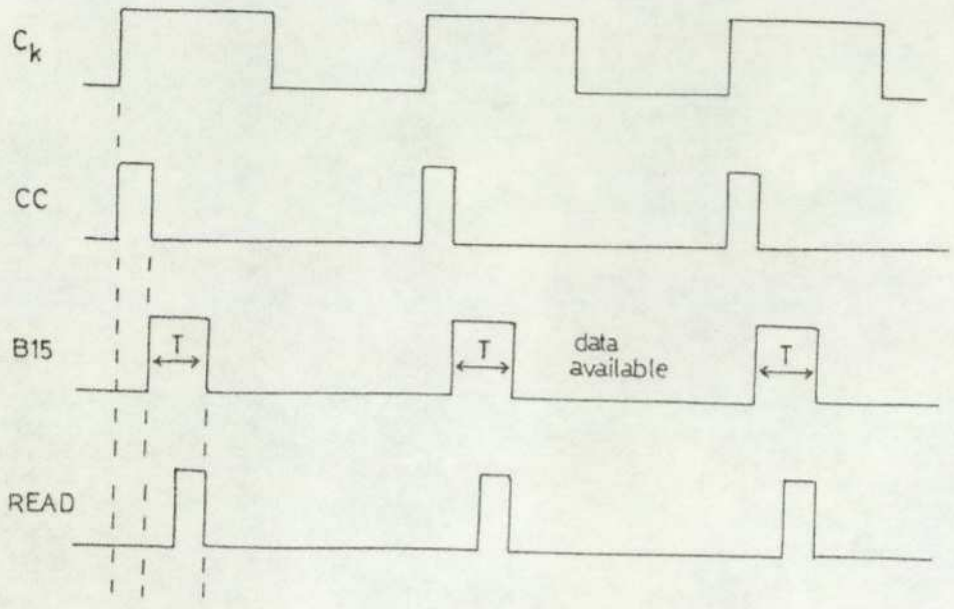


Figure 6-1 MEASUREMENT OF OPTIMAL SAMPLING RATE

the 'conversion complete' (CC) and 'read data' (READ) signals on the analog/digital conversion (ADC) board (Fig. 6-2). At each clock pulse CC goes high, a sample of the input signal is taken and analog to digital conversion begins. On completion of conversion (which takes approximately 20 μsecs) CC goes low and acts as a trigger causing bit 15 (B15) of a specified location to be set, so indicating to the processor that a data sample is available for reading. When the data sample is eventually read by program, B15 is automatically cleared. Correct operation of the system is shown in Fig. 6-2(a) with Fig. 6-2(b) illustrating the case where sampling is at a greater rate than the processes can cope with. In this figure, the second sample is not read before the third sample is converted. B15 remains set and so the second sample is lost and it may appear to the processor that data is available whilst the actual conversion is taking place. Under correct operation the processor waits for a sample to become available so READ is synchronised with CC and this synchronism can be seen on an oscilloscope. When sampling is too fast READ and CC go out of synchronisation and the non-trigger signal of the pair is seen to 'run' on the oscilloscope.

Maximum sampling rates (e.g. for software models 1, 2 and 3) were measured by increasing the sampling rate to just below the point at which READ and CC went out of synchronisation. Under these conditions

Figure 6-2(a) CORRECT OPERATION



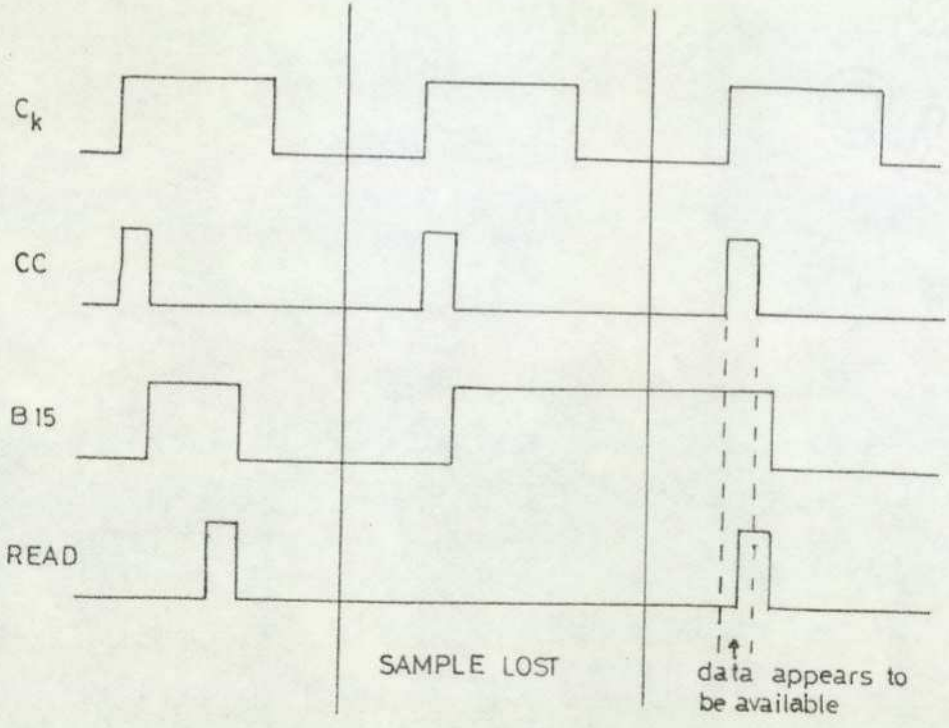


Figure 6-2(b) INCORRECT OPERATION

the processor is requesting data as it is made available.

The measurement technique does not interfere with the system in any way.

6.3 Execution sequences

Detection of instruction execution sequences presented a more difficult problem than that of measuring optimal sampling rates. The obvious way to do this was by monitoring bus activity for each of the processors in the system. F100-L instructions consist of either one, two or three sixteen-bit words. The first word of an instruction, in which the operation code is always embedded, is deemed to be the 'instruction' whilst second and third words are, if present, operands associated with the instruction. During instruction fetch cycles the signal $\overline{\text{IRd}}$ (instruction read) goes low only while the first word of the instruction being fetched is on the bus, so $\overline{\text{IRd}}$ can be used as a trigger to detect execution sequences.

A logic analyser could be used to trap instructions on the bus but available analysers had a maximum trace facility of 256 words which, while capable of providing 'snapshots' of small programs or sections of programs, was totally inadequate for the extent of monitoring required for this work.

In order to make reliable deductions concerning the general behaviour of closely coupled, concurrent processes operating at their maximum attainable rate, traces of a large number (in the region of several million) of program instructions were considered necessary. As the system was operating in real time the monitoring system had to have a data capture, processing and storage capability sufficient to keep pace with the system under test for at least the length of time necessary to complete such a trace.

Execution paths through a program are determined by conditional branches within the program text. If behaviour at such branches is known, intermediate behaviour can be assumed and it's pattern (which is likely to account for the majority of instruction executions within a process) does not need to be retained by a monitoring system.

As a result of these requirements it was decided to design a specialised hardware monitor, capable of trapping user-defined instruction sequences, which would be controlled by a separate microcomputer.

6.3.1 Design and control of hardware monitor

The computer chosen to control the monitor was General Instrument's GIMINI microcomputer (GIC, 1976) which is based on the CP1600 16-bit nMOS microprocessor. The CP1600 is clocked at 6MHz and instruction execution is pipe-lined 8-bit parallel giving an average execution time of around 6 μsecs. The instruction set includes a 'branch external' (BEXT) instruction, described later, which proved to be particularly suitable for the monitoring application.

The complete system is shown in Fig. 6-3. Either one of the 16-bit F100-L buses can be monitored at any time. The monitor unit has three 16-bit comparison stages (A, B and C) which are set by the user (by means of switches) to specify a particular instruction sequence which is to be trapped. Details of the comparison are shown in Fig. 6-4. Triggered by $\overline{IRd}$, the monitor compares the instruction currently on the F100-L bus with A and, if they are identical, the next word to appear on the bus is compared with B. A mismatch with A or B causes the monitor to be reset immediately. A second match (with B) results in the third word on the bus being ignored and the fourth word, which must be an instruction, being compared with C. A third match causes a 16-bit hardware counter to be incremented, while a mismatch at this stage sets an external branch input ($\overline{EBC3}$) to the CP1600 processor. This causes the CP1600 to increment

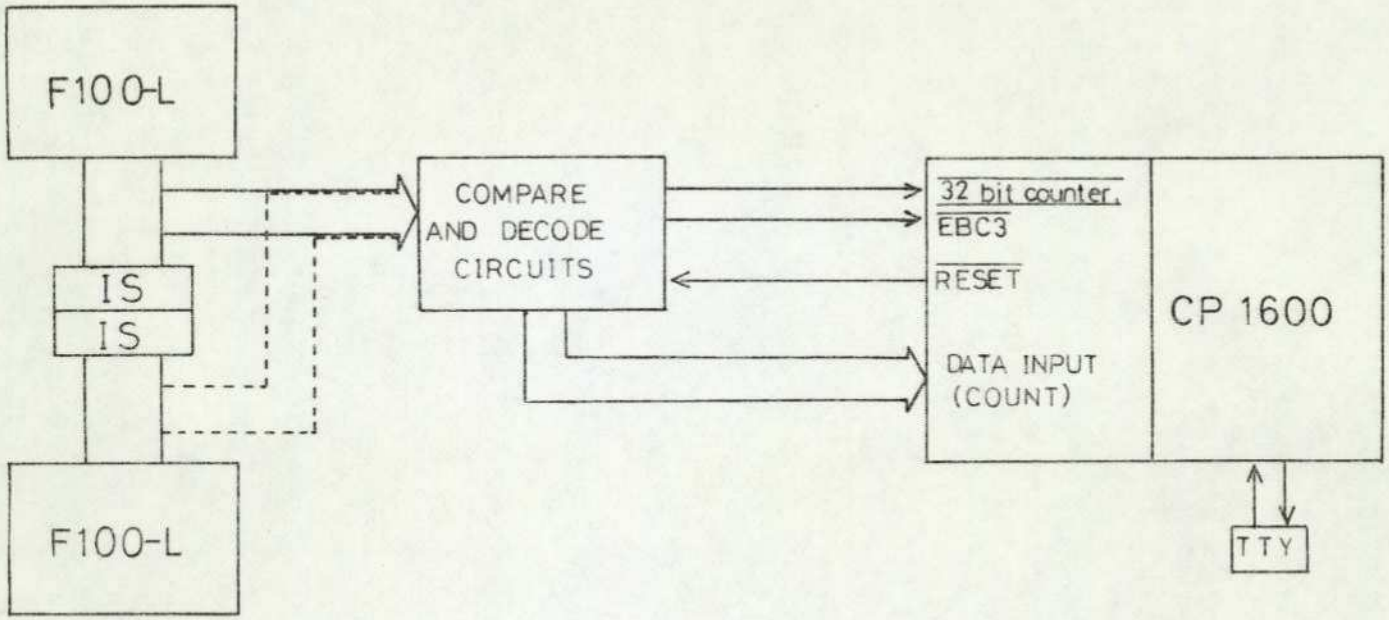


Figure 6-3 THE COMPLETE MONITOR SYSTEM

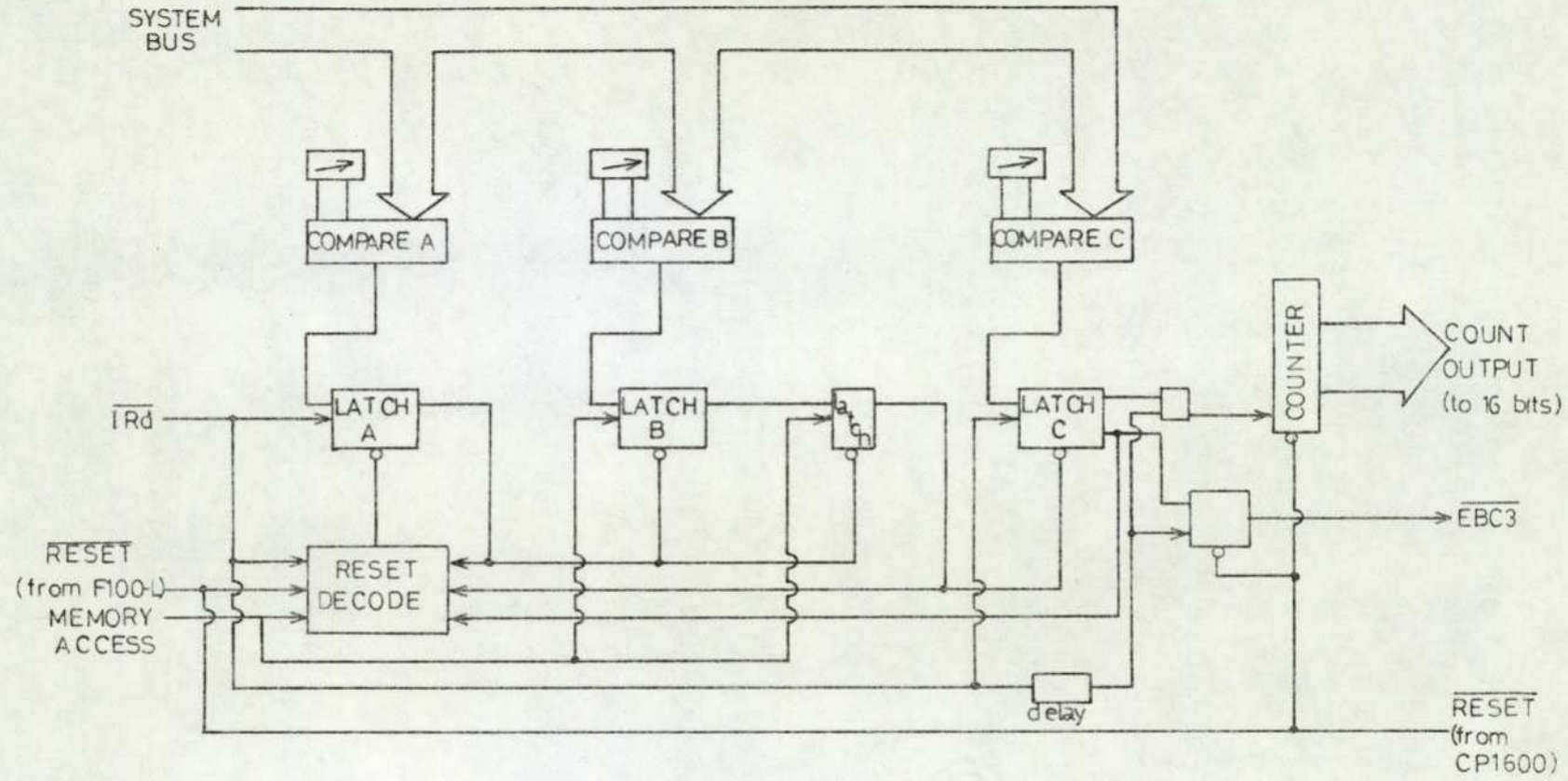


Figure 6-4 MONITOR COMPARE AND DECODE CIRCUITS

a memory location and to reset the monitor. Fig 6-5 shows details of the monitor/CP1600 interface. The address of the location to be incremented is calculated from the hardware counter plus a displacement. There is also a 32-bit hardware counter which is incremented by $\overline{IRd}$ and therefore accumulates the number of F100-L instructions executed. If this counter overflows an interrupt to the CP1600 is generated. The overall behaviour of the monitor (Fig. 6-6) is to count continuously the number of times a particular instruction sequence is repeated and to cause the CP1600 to accumulate the resulting count values. Note that matches with A and B followed by a mismatch with C will generate a zero count.

The sequences of instructions which can be trapped are as follows:

	Stage A	Stage B	Ignored	Stage C
(1)	instruction	operand	operand	instruction
(2)	instruction	instruction	operand	instruction
(3)	instruction	operand	instruction	instruction
(4)	instruction	instruction	instruction	instruction

The only exception which must be avoided by the user is the case in which the required sequence is preceded by a similar sequence which could cause a reset in the middle of the correct sequence.

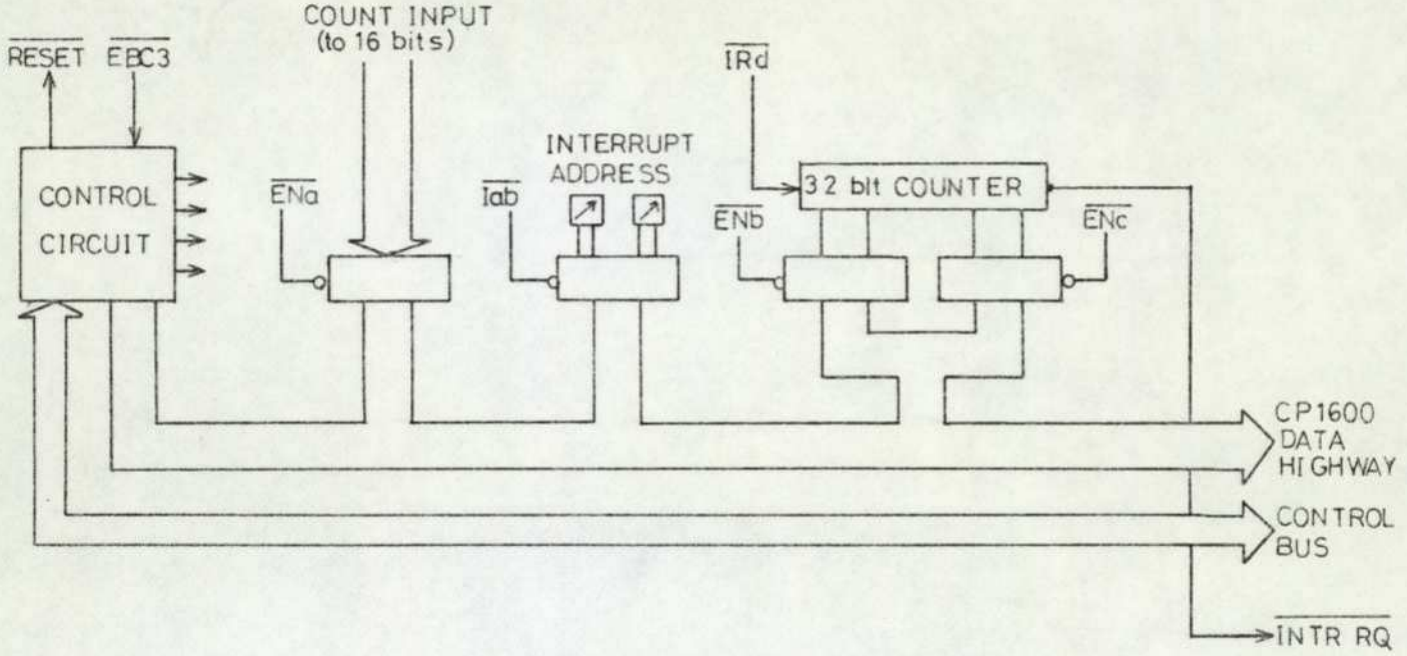
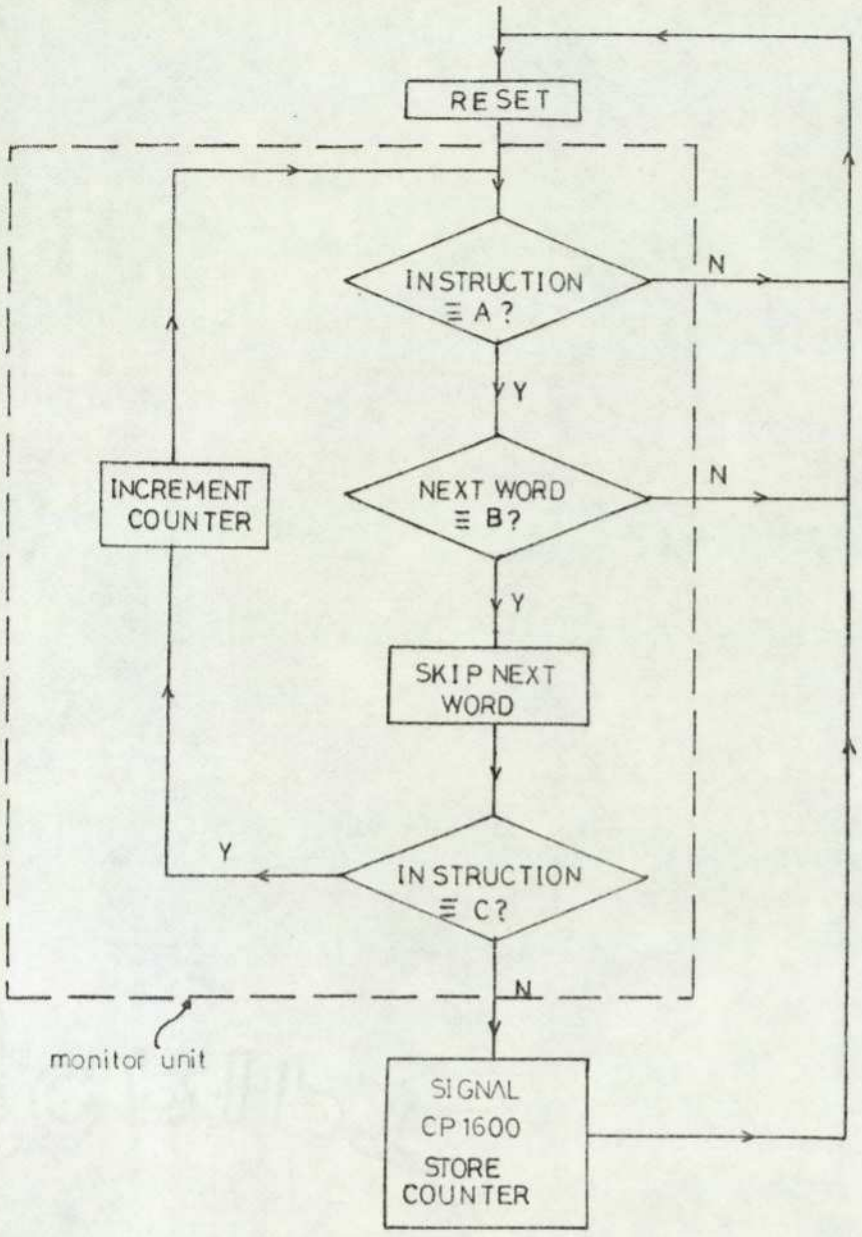


Figure 6-5 MONITOR PROCESSOR INTERFACE



OPERATION OF MONITOR

Figure 66

For example:

code being monitored

```
STO /35+)
STO /35+)  all single      A   STO /35+
LDA  X  )  word            B   LDA  X
ADD  Y  )  instructions    C   CLR  S
CLR  S  )
```

The first 'STO /35+' would match A but the second 'STO /35+' would mismatch B and so the monitor would reset for 'LDA X' and miss the correct sequence.

6.3.2 Monitor control program

The hardware monitor is controlled by a CP1600 program, details of which are given in Appendix 4. The program maintains a data buffer of user defined size in the range 1 - 2048 words. Counter values passed to it by the monitor are used as an offset to the base address of the buffer to give the actual address of the location to be incremented. Thus on receiving a counter value of zero the program would increment the first buffer location while a count of one would result in the second buffer location being incremented.

Initially the program resets the hardware monitor, asks the user to supply the buffer size and decide whether data recording or simply output of buffer contents is required, and in the former case clears the buffer. The user then specifies the number of passes required and whether or not they are to be printed. The system is now initialised and waits for the user to request enabling of the monitor. A

typical run is shown in Fig. 6-7. A single pass will cause the monitor to operate until either one of the buffer locations reaches its maximum value of 32767 or the hardware instruction count reaches 2^{32}. If printing has been requested the buffer contents, number of overflows, instruction count, loops count and computed count are printed at the end of each pass. Overflows are counter values which generate addresses outside the scope of the currently defined buffer area.

Instruction count gives the number of F100-L instructions executed during a pass. Loops count gives the sum of the values in the buffer locations and computed count gives the weighted sum of these values as:

$$\text{computed count} = \sum_{i=1}^{\text{buffersize}} i * \text{buffer}[i]$$

As many passes as have been requested are automatically performed (with or without printing) and then a set of 'average results' are printed which are average values, over the number of passes performed, for the same variables as are printed at the end of a pass.

6.3.3 Method of use of monitor

The monitor was used in two different ways to obtain the results described in Chapter 7. The first method was used to measure the percentage of times that a program took a certain execution path.

```
REC/OPT:R
NO. OF PASSES = 3:
PRINT PASSES (Y/N)?::Y
READY:
PASS 1
00000     28411   16388   32767      389        0        0        0        0        0
     OVERFLOWS = 0

     INSTR COUNT = 2677614
     LOOPS COUNT = 77955
     COMPT COUNT = 161044


PASS 2
00000     28169   16426   32767      372        0        0        0        0        0
     OVERFLOWS = 0

     INSTR COUNT = 2671917
     LOOPS COUNT = 77734
     COMPT COUNT = 160810


PASS 3
00000     28515   16398   32767      379        0        0        0        0        0
     OVERFLOWS = 0

     INSTR COUNT = 2680824
     LOOPS COUNT = 78059
     COMPT COUNT = 161128


AVERAGE RESULTS
00000     28365   16404   32767      380        0        0        0        0        0
     OVERFLOWS = 0

     INSTR COUNT = 2676785
     LOOPS COUNT = 77916
     COMPT COUNT = 160994


BUFFER SIZE = 09:
```

Figure 6-7 MONITOR CONTROL PROGRAM OUTPUT

If a program's structure is as shown in Fig. 6-8

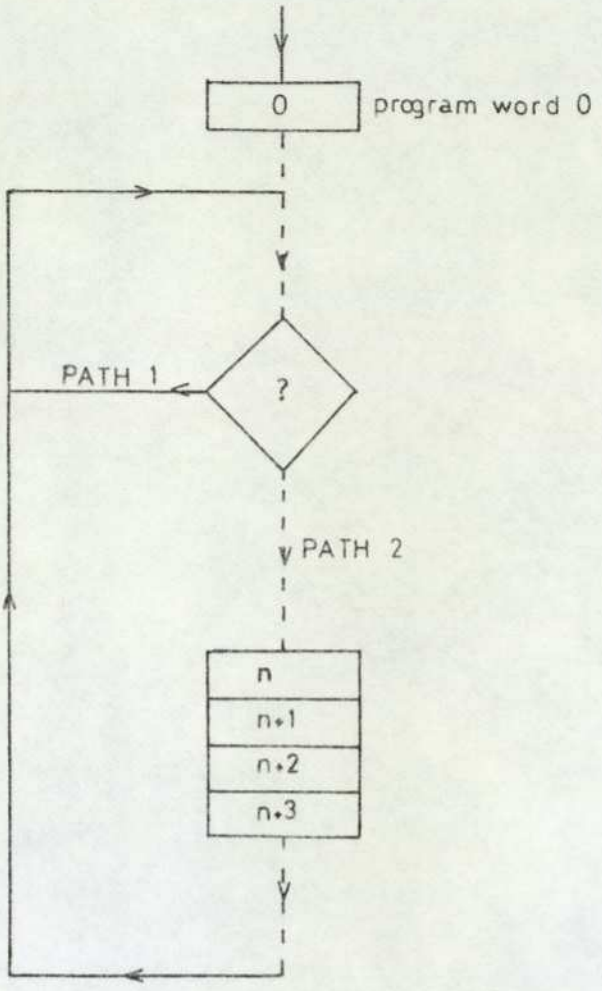


Figure 6-8

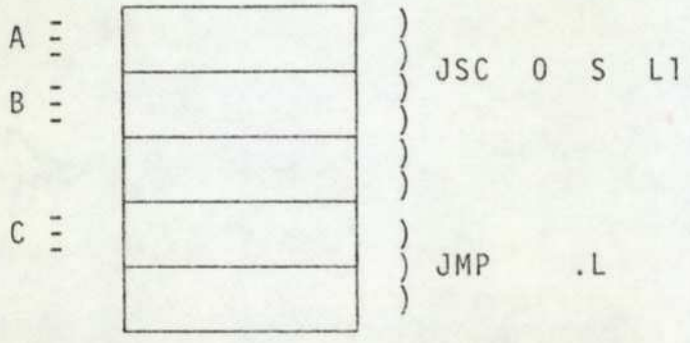
then execution of path 2 can be detected by setting monitor stages A and B to the values of program words n and n + 1 respectively with register C set to any value which is not the same as program word n + 3. The monitor will send a counter value (of zero) to the CP1600 every time this code sequence is executed.

By using the instruction count the frequency, in terms of instructions executed, with which the program took a certain run-time path can be calculated providing that words n and n + 1 of the program are not repeated elsewhere.

The second use of the monitor was to count repeated executions of a particular code pattern which is best illustrated by considering an F100-L implementation of a P operation on a semaphore i.e.

```
L-   JSC  0  S  L1
     JMP     .L
L1-  next instruction (which must not be a JMP)
```

Bit 0 of location S is the semaphore. The JSC is a three word instruction, the code pattern is shown below.



If A, B and C are set as shown then a counter value of zero will indicate that the program did not loop on the JSC instruction (i.e. the P operation was immediately successful) whilst a counter value of, say, three would indicate that it was only at the fourth execution of the JSC that the P operation was successfully completed. 'Computed count' is the number of

executions of the JSC instruction and 'loops count' represents the number of P operations completed on the semaphore.

6.3.4 Testing of monitor

The monitor and its associated program were tested on different F100-L code patterns to ensure that correct counts and buffer values were produced and that passes terminated as expected either on a buffer value reaching 32767 or on the instruction count exceeding its hardware counter range (2^{32}) - although this was a condition not expected to arise during actual use of the monitor.

The critical area of operation of the system occurs when the F100-L is executing tight program loops and there is a danger that the CP1600 will have insufficient time to increment a buffer location and reset the monitor before the next occurrence of the external branch request.

The CP1600 code performing these operations was written as efficiently as possible and an option of removing the code handling overflows was provided to cope with critical measurements. The lack of overflow detection did not prove a problem (in fact overflow was never detected in practice) as the approximate maximum counter values obtainable from monitoring of tight program loops could be calculated. The calculation was made on the basis that the only critical areas were known to occur while a program

waited for a synchronisation signal from another concurrent program. The maximum waiting time could be seen by inspection of the second program, and the buffer size set sufficiently large to prevent overflow from occuring.

The relevant CP1600 code is given below

```
   MVII  1,R1          (set R1 for increment
   MVII  DUMY,R3       (load dummy address pointer
   MVO   R1,167720     (hardware reset
```

critical loop

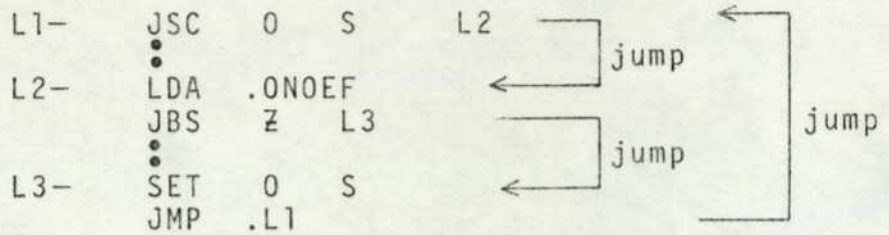
```
STRT  MVO@  R2,R3       (R2 stored in address in R3
WAIT  BEXT  CNTR,3      (branch if counter data available
      B     WAIT        (else goto WAIT
CNTR  MVI   167721,R3   (read counter/reset monitor
      MVI@  R3,R2       (read present total
      ADDR  R1,R2       (increment
      BPL   STRT        (continue if < 32K
```

The BEXT (branch external) will branch if external input 3 (from range 0-15) is false thus indicating that the monitor has counter data ready to pass to the CP1600. The alternative to BEXT would be to read from a location, test for a condition and branch accordingly. The instruction at label CNTR reads from the counter (at address #167721) and, at the end of the read cycle , generates an automatic hardware reset.

The BEXT instruction and the automatic reset were used to help minimise code, and hence execution time, for the critical loop. Execution time was further reduced by addition, in hardware, of the buffer start address to the incoming count value. This, of course, places a restriction on the maximum input count value if overlap problems are to be avoided (e.g. if the

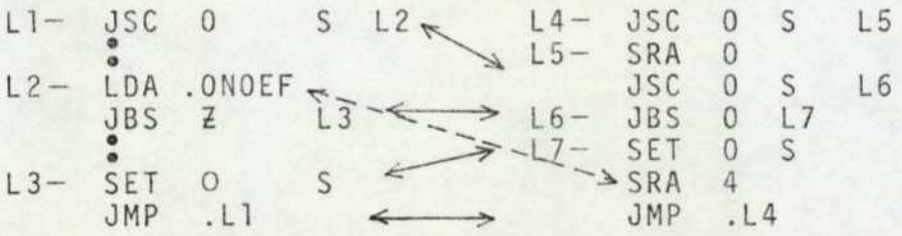
base is set to 1K then counts of 0 and 1K + 0 will map to the same buffer address as the 1K line of the counter will be disconnected). The base address was set to 2K, permitting a maximum count of 2048, which was far in excess of any expected values.

The worst-case instruction sequence which could occur in the F100-L programs was of the form:



i.e. one process waiting for the other process to provide either buffer frames or data.

Test code equivalent to the worst-case code was devised as:



The 'SRA 4' instruction in the test code was shorter than the 'LDA .ONOEF' instruction in the actual code, the premise being that if the test code could be monitored correctly then so could the actual code. In the test code, the first two words of the JSC instructions (coding 'JSC 0 S') are identical - the only difference in the instructions being in the third word holding the jump address (L5 or L6).

Monitor stages A and B were set to 'JSC 0 S' and stage C was set to the single word instruction 'SRA 0'.

After the test code had been monitored the first buffer location held the value zero and the second buffer location held the value 32767. This showed that the monitor was counting on instruction L5 and passing data on a mismatch on L6. If it had not been correctly reset by the time it returned to L4 then the count would have been missed and zero counter values returned on the mismatch at L6.

When the 'SRA 4' instruction was replaced by a (shorter) 'SRA 3' instruction the monitor could not always reset in time and some entries appeared in the first buffer location.

Timings for the instruction sequences in terms of Ra (read access), Rc (read cycle), M (read/modify/write) and L (logic cycle) are:

	instruction	timing
actual code	JSC 0 S L2	Ra + 2Rc + M + 18L
	LDA .ONOEF	2Ra + Rc + 18L
	JBS Z L3	Ra + Rc + 20L
	SET 0 S	Ra + Rc + M + 18L
	JMP .L1	Ra + Rc + 2L
	Total	6Ra + 6Rc + 2M + 76L
test code	JSC 0 S L6	Ra + 2Rc + M + 18L
	JBS 0 L7	Ra + Rc + 19L
	SET 0 S	Ra + Rc + M + 18L
	SRA 4	Ra + 8L
	JMP .L4	Ra + Rc + 2L
	Total	5Ra + 5Rc + 2M + 65L

confirming that actual code time > test code time

$$6Ra + 6Rc + 2M + 76L > 5Ra + 5Rc + 2M + 64L$$

Replacing 'SRA 4' with 'SRA 3' would reduce the test code time by a single logic beat but results showed clearly that the monitor was not functioning correctly. The actual code is longer than the test code by Ra + Rc + 11L, which provides a reasonable safety margin.

6.4 Measurements made with monitor

Monitor measurements made on models 2 and 3 (Fig. 6-9) were, firstly, to trap P operations on semaphores. In model 2, there was a single P operation on each of the two semaphores in both the filter and i/o processes. The code sequences to be trapped ('JSC 0 S' or 'JSC 1 S') were thus unique for each program so a set of four measurements (two on each bus) completely monitored process behaviour in the region of P operations for a particular instance (determined by count values Ci and Cf) of a model. The measurements were made without testing for overflows as there was a possibility of the F100-L processors executing tight loops in the monitored region (6.3.4).

As well as showing the number, on average, of JSC instructions executed to complete a P operation these results could show how many times a critical section was entered (by completion of a P operation) for a known instruction count.

In model 3, with its single semaphore, there were two 'JSC 1 S' instructions in each program so the same set of measurements as for model 2 could only show the average behaviour of the programs over their respective pairs of critical sections.

The second set of measurements were made to monitor behaviour at the remaining relevant conditional branches of which there were two in each main program loop (marked with * in Fig. 6-9). This was done by trapping the SBS instructions shown with a deliberate mismatch on stage C. The four SBS instructions are unique in models 2 and 3 and so could be trapped separately in all cases. The programs could not loop tightly around these instructions so overflow values were checked for. The results were interpreted as the number of times a program made a certain branch for a known instruction count.

Normalisation of the results for the first and second sets of measurements allowed the number of entries to a critical section to be related to the path taken within that critical section (7.4). The frequency with which a certain path was taken could then be expressed in terms of a percentage of the times that the conditional branch (marked with * in Fig. 6-9) was met.

Multiple passes with the monitor showed consistent results. For actual measurements, such as those reported in Chapter 7, three passes were made and the average results used.

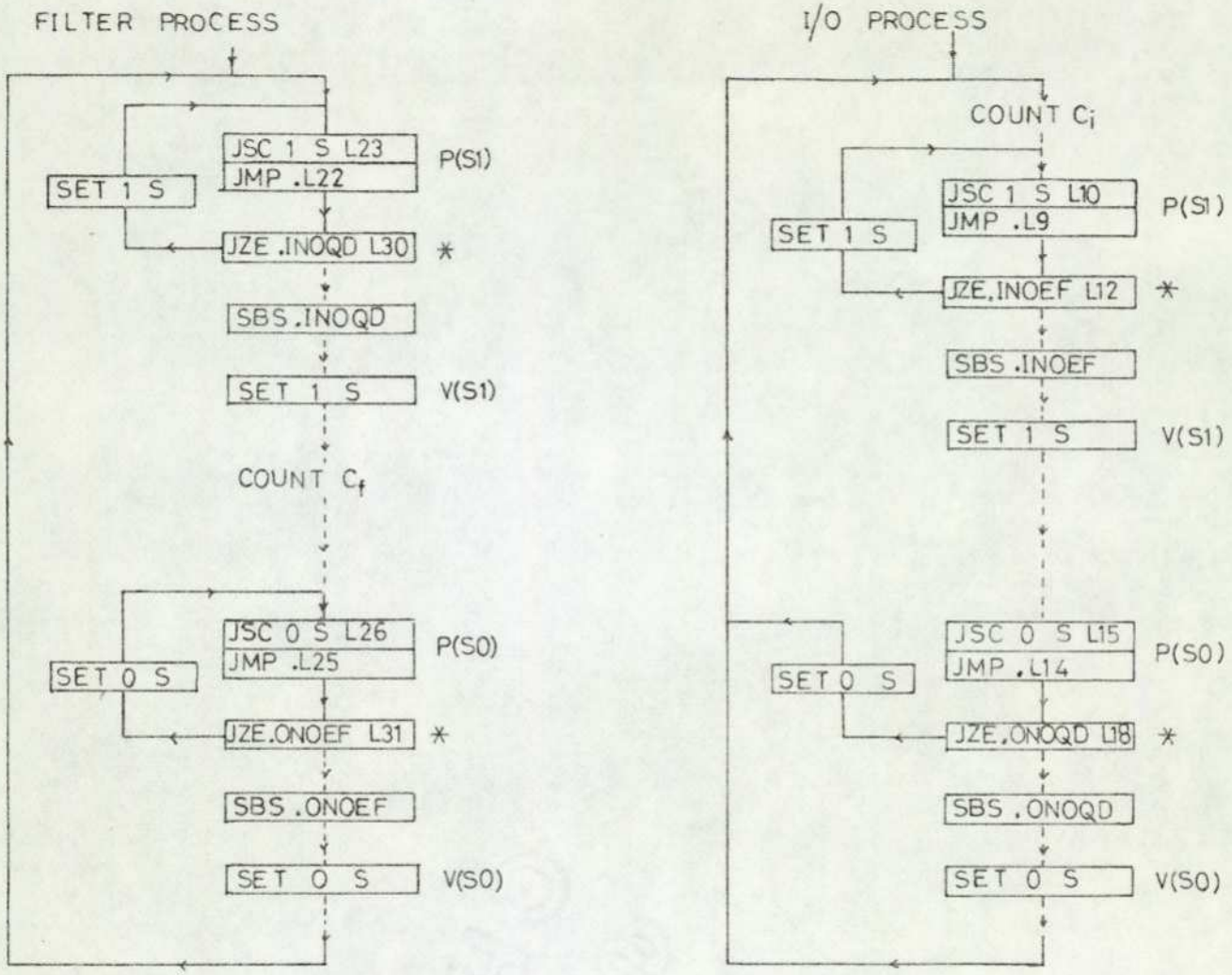


Figure 6-9 CODE SEQUENCES

Chapter 7

Results

7.1 General behaviour of models

To establish their general patterns of behaviour, models 1, 2 and 3 (as described in Chapter 5) were run to establish the maximum achievable sampling rates for varying values of counts i and f in the i/o and filter processes respectively.

7.1.1 Model 1

The single processor model was run with fixed counts (25, 50, 75, 100) in the filter process and variable counts in the i/o process and then, as a check, with fixed counts (25, 50, 75, 100) in the i/o process and variable counts in the filter process. The sampling rates for these values, the first of which is shown in Fig. 7-1, displayed the same smoothly decreasing sampling rate as the execution time of the variable count increased.

7.1.2 Model 2 (2 processors, 2 semaphores)

The dual processor model can be described in terms of it's two constituent processes.

(i) filter process

The program consists of a base loop, a count, contention and a wait.

$$P_f = b_f + c_f + x_f + w_f$$

The wait, w_f, is programmed as a loop on a 'data available' flag (INOQD) which is in shared memory and must therefore be accessed in a critical section.

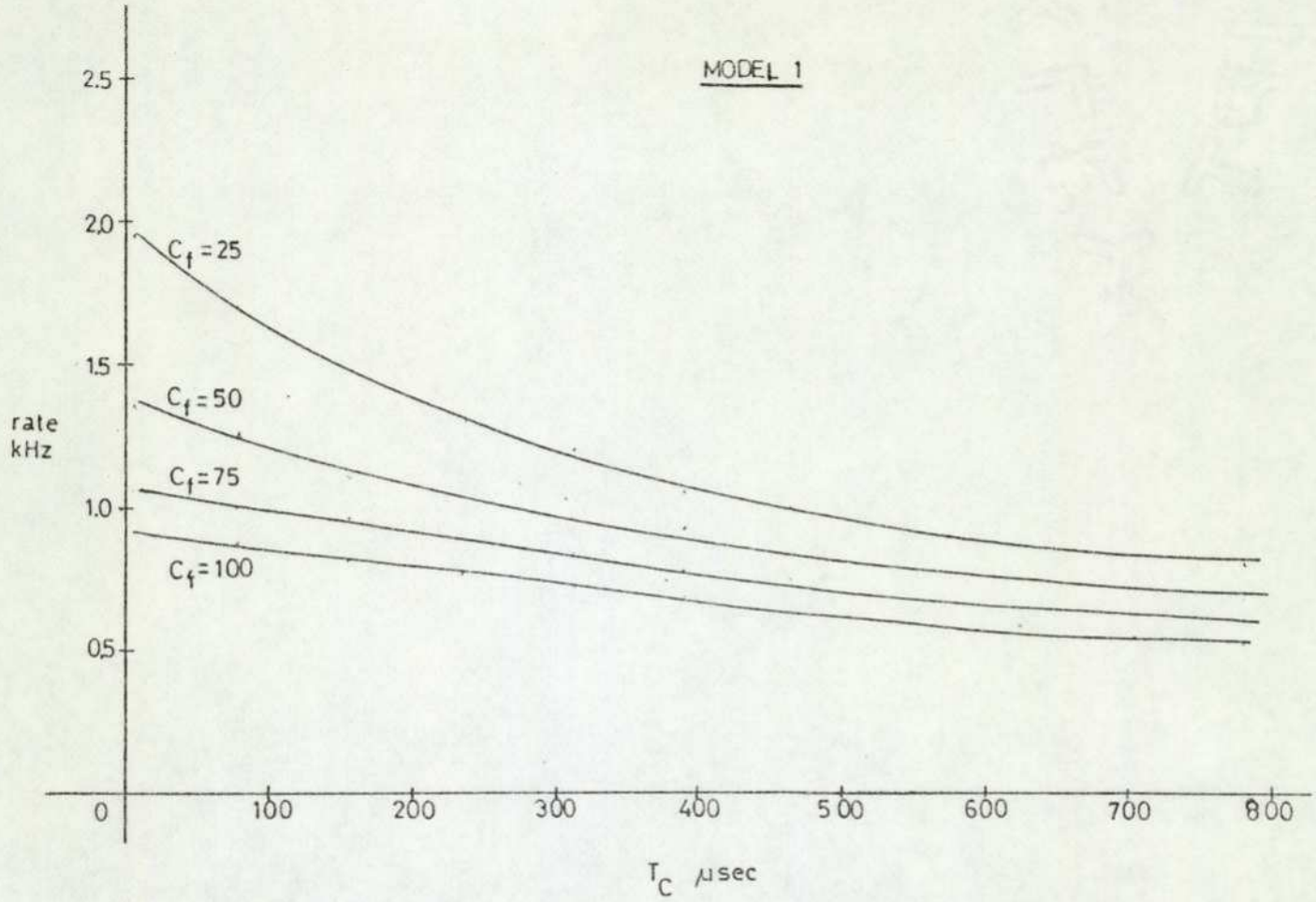

MODEL 1
2.5
2.0
1.5
1.0
0.5
0
rate
kHz
$C_f = 25$
$C_f = 50$
$C_f = 75$
$C_f = 100$
100
200
300
400
500
600
700
800
T_C μsec

Figure 7-1

(ii) i/o process

The program consists of a base loop, a count, contention and a wait.

$$P_i = b_i + c_i + x_i + w_i$$

In this case the wait, w_i, is a loop on a 'data available' flag (adc input) which is not in shared memory.

Two sets of measurements were taken for the model. The first set obtained the maximum sampling rates for fixed values of c_i (1, 15, 25, 50, 75, 100, 150, 200) with c_f varying and is shown in Fig. 7-2. As c_f increases, the sampling rate for a fixed value of c_i rises to a peak and then decreases.

The second set of measurements obtained the maximum sampling rates for fixed values of c_f (1, 15, 25, 50, 75, 100, 150, 200) with c_i varying and is shown in Fig. 7-3. As c_i increases, the sampling rate for a fixed value of c_f is steady initially and then decreases.

In Fig. 7-4 the measurements for $c_i = 25$ with c_f varying (case 1) and $c_f = 25$ with c_i varying (case 2) are isolated for discussion. The behaviour of the model in these cases was, with the aid of the hardware monitor (Chapter 6), established to be as follows.

Case 1 (c_f varying)

(i) The maximum sampling rate occurs at time M_f when neither process is waiting ($w_i = w_f = 0$). There is minimal contention in the system.

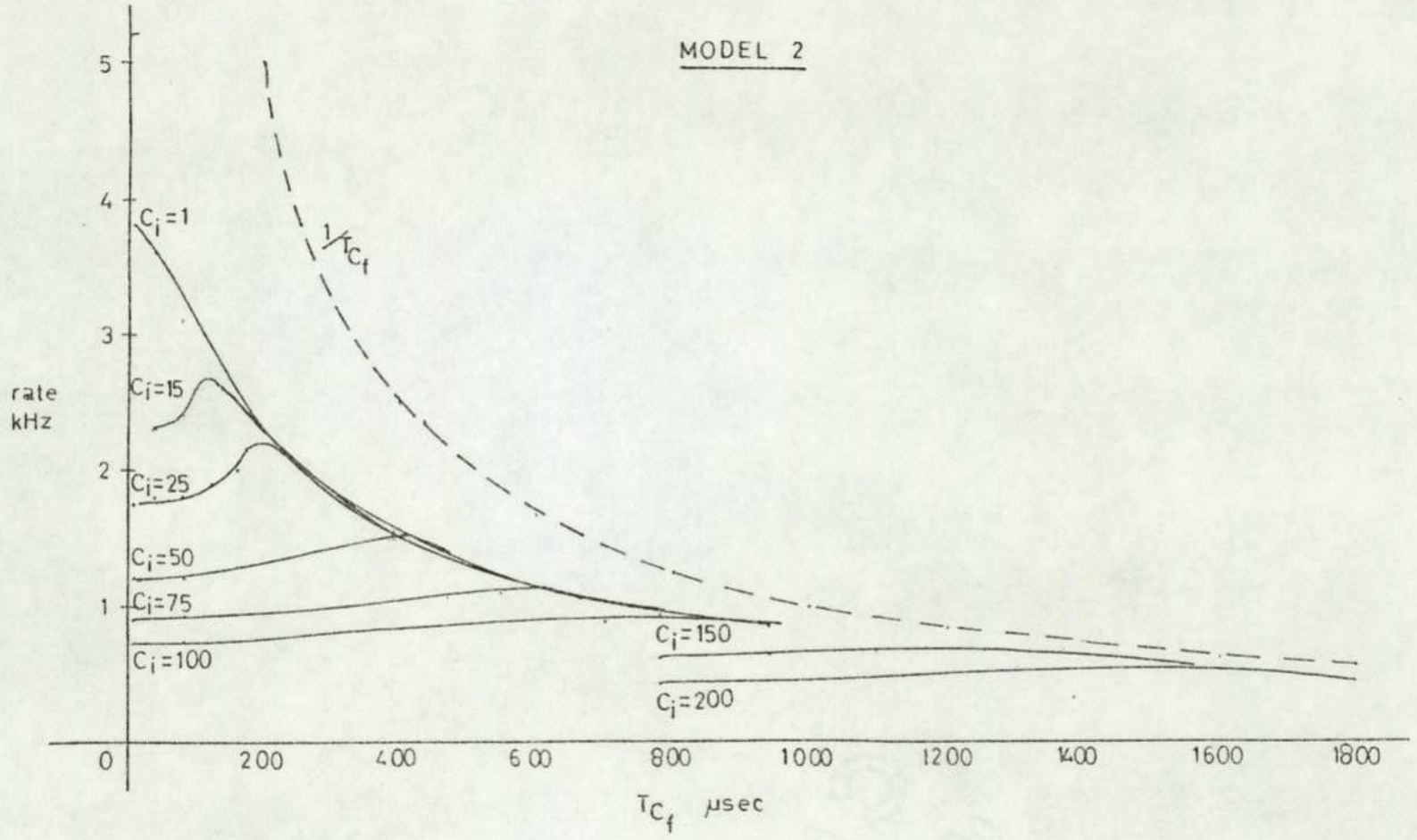

MODEL 2
5
4
3
2
1
0
rate
kHz
C_i=1
$1/T_{C_f}$
C_i=15
C_i=25
C_i=50
C_i=75
C_i=100
C_i=150
C_i=200
200
400
600
800
1000
1200
1400
1600
1800
T_{C_f} μsec

Figure 7-2

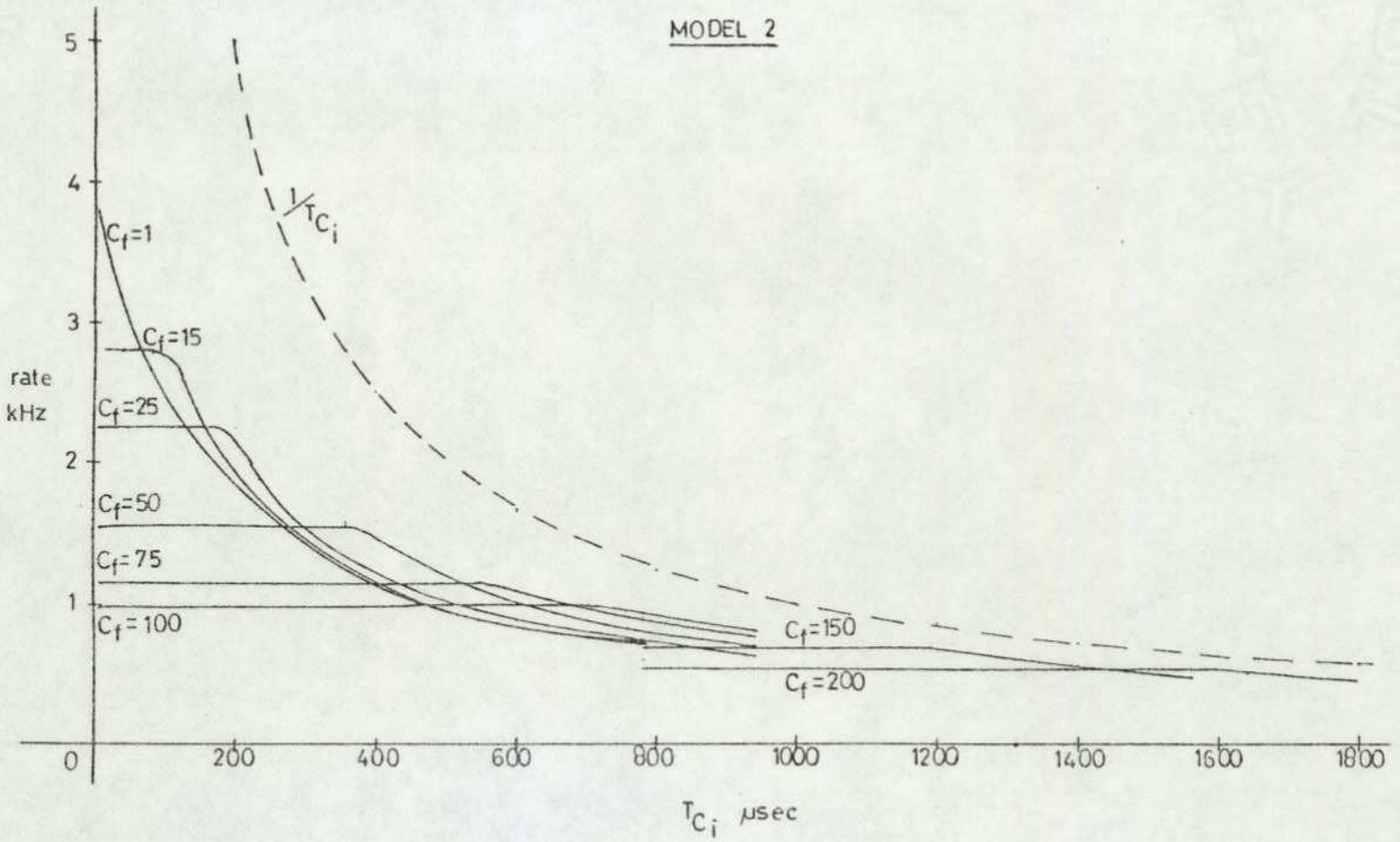
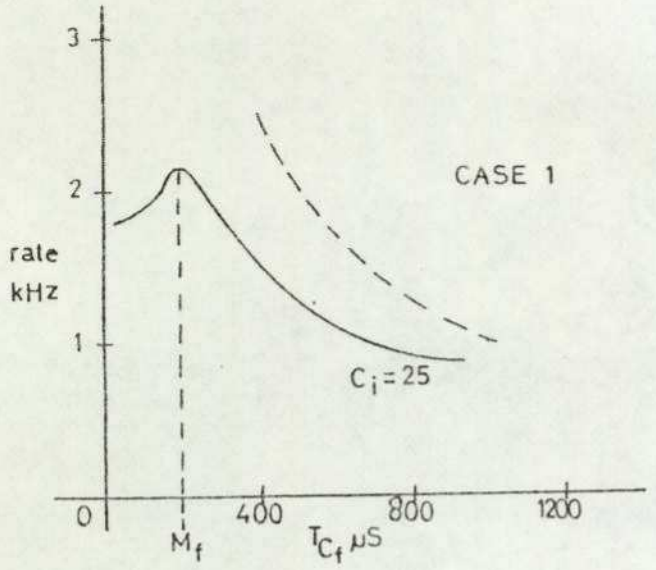
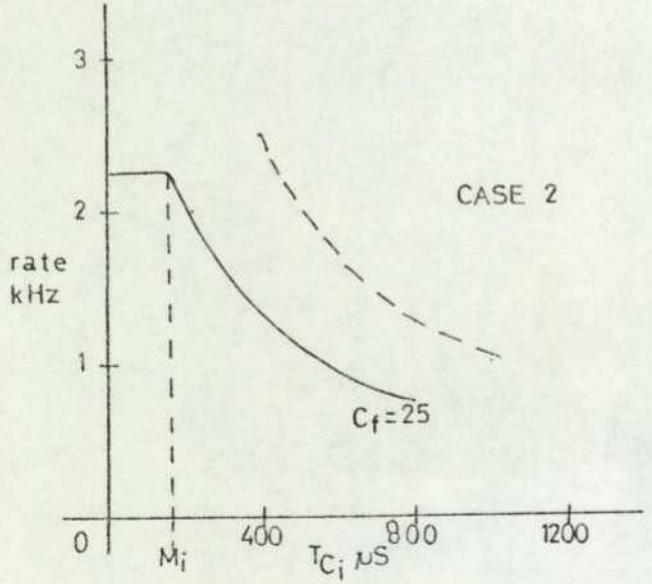

MODEL 2
5
4
3
2
1
0
rate
kHz
$1/T_{C_i}$
C_f=1
C_f=15
C_f=25
C_f=50
C_f=75
C_f=100
C_f=150
C_f=200
200
400
600
800
1000
1200
1400
1600
1800
T_{C_i} μsec

Figure 7-3


3
2
rate
kHz
1
CASE 1
$C_i = 25$
0
400
800
1200
M_f
T_{C_f} µS


3
2
rate
kHz
1
CASE 2
$C_f = 25$
0
400
800
1200
M_i
T_{C_i} µS

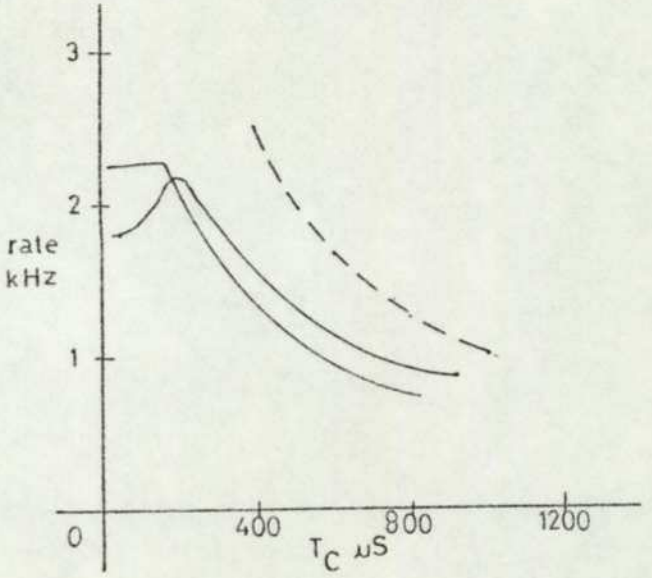

3
2
rate
kHz
1
0
400
800
1200
T_C µS

Figure 7-4

(ii) For $T_{c_f} > M_f$, rate $(T_{c_f}) \rightarrow {}^{1}/_{T_{c_f}}$ as $T_{c_f} \rightarrow \infty$

In this region, as c_f increases the rate (which is now governed by P_f as c_i is constant) decreases. P_i is idling so w_i increases but as it involves no accesses on shared memory no additional contention is introduced.

(iii) For $T_{c_f} < M_f$, rate (T_{c_f}) < rate (M_f) as $c_f \rightarrow 1$.

As $c_f \rightarrow 1$, the rate is governed by P_i with $w_i = 0$.

The rate would be constant but, as c_f decreases, w_f increases. w_f involves repeated accesses on shared memory and therefore causes additional contention in P_i, which in turn causes additional contention in w_f. As a result, the sampling rate decreases as c_f decreases. This is therefore a very complex circular process.

Case 2 (c_i varying)

(i) The maximum sampling rate occurs up to time M_i. At M_i, $w_i = w_f = 0$ i.e. neither process is waiting.

(ii) For $T_{c_i} > M_i$, rate $(T_{c_i}) \rightarrow {}^{1}/_{T_{c_i}}$ as $T_{c_i} \rightarrow \infty$.

In this region, as c_i increases the rate, which is governed by P_i, decreases. P_f is idling and so w_f increases and causes additional contention in P_i as in case 1. This causes the rate to decrease further and it can be seen in Fig. 7-4 that for times greater than M_f and M_i, the rate for case 2 is lower than that for case 1.

(iii) For $T_{c_i} < M_i$, as $c_i \rightarrow 1$ the rate remains steady. It is now governed by P_f with $w_f = 0$ and although w_i increases as c_i decreases, w_i does not involve accesses on shared memory and so does not introduce any additional contention into the system.

Measurements with the hardware monitor (described in 6.4) confirmed this pattern of behaviour. The total amount of contention in the system (compounded of contention on both semaphores in both processes) was measured in the region of $c_f = 25$ and $c_i = 25$ and is shown in Table 7.1. These points were chosen as being most likely to exhibit any inconsistent behaviour (7.1.4).

Contention in the region of optimal points

Pt	c_f	c_i	rate in kHz	contention	
a	25	18	2.25	0.94	
b	25	20	2.25	0.71	
c	25	23*	2.25	0.78	case 2
d	25	25	2.16	1.78	
e	25	28	2.02	3.44	
f	20	25	1.98	3.84	
g	22	25	2.11	3.64	
h	25	25	2.16	1.78	case 1
i	28	25*	2.16	0.70	
j	30	25	2.07	0.65	

* optimal

Table 7.1

It can be seen that points a, b and c show reasonably steady contention corresponding to the steady sampling rate. Points d and e show contention increasing. Points h, g and f also show increasing contention similar to that of d and e.

7.1.3 Model 3 (2 processors, 1 semaphore)

Model 3 is identical to model 2 apart from relying on a single semaphore to control access to the input and output buffers instead of each buffer having a separate semaphore associated with it.

The behaviour of model 3 was expected to follow the same pattern as that of model 2 but with lower sampling rates due to increased contention caused by the use of a single semaphore.

This proved to be the case and measurements for model 3 corresponding to those already described for model 2 are shown in Figures 7-5, 7-6 and 7-7.

7.1.4 Irregularities in sampling rate

In model 3, an irregularity can be seen running through the lines c_i = 1, 15, 25, 50 where the rate fluctuates near its optimum (Figs. 7-8 and 7-9). This is most marked when c_i = 25. Investigation of the same region in model 2 (Fig. 7-10) shows a small flat region at the optimum of c_i = 25.

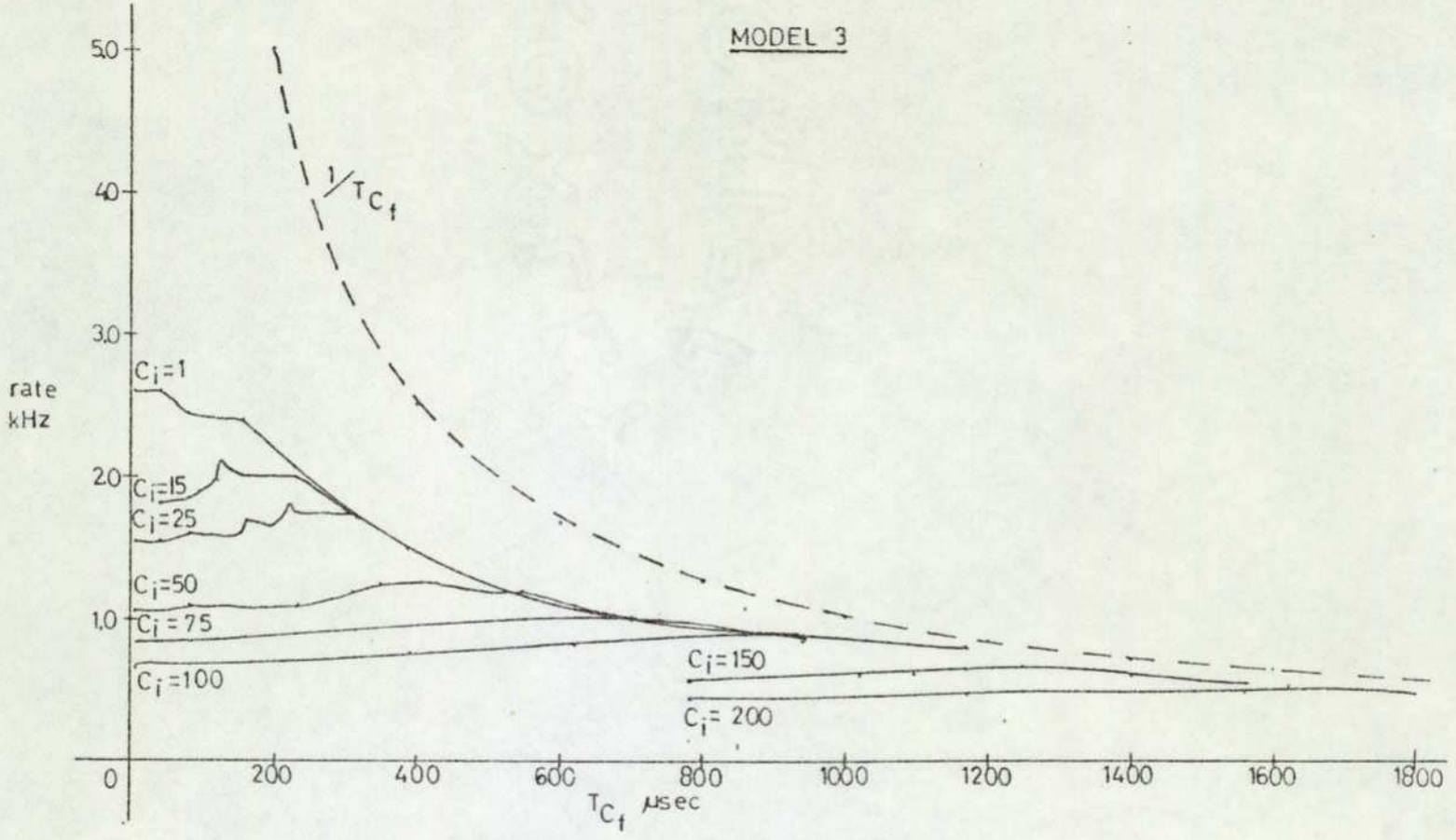

MODEL 3
5.0
4.0
3.0
2.0
1.0
0
rate
kHz
1/T_Cf
C_i=1
C_i=15
C_i=25
C_i=50
C_i=75
C_i=100
C_i=150
C_i=200
200
400
600
800
1000
1200
1400
1600
1800
T_Cf μsec

Figure 7-5

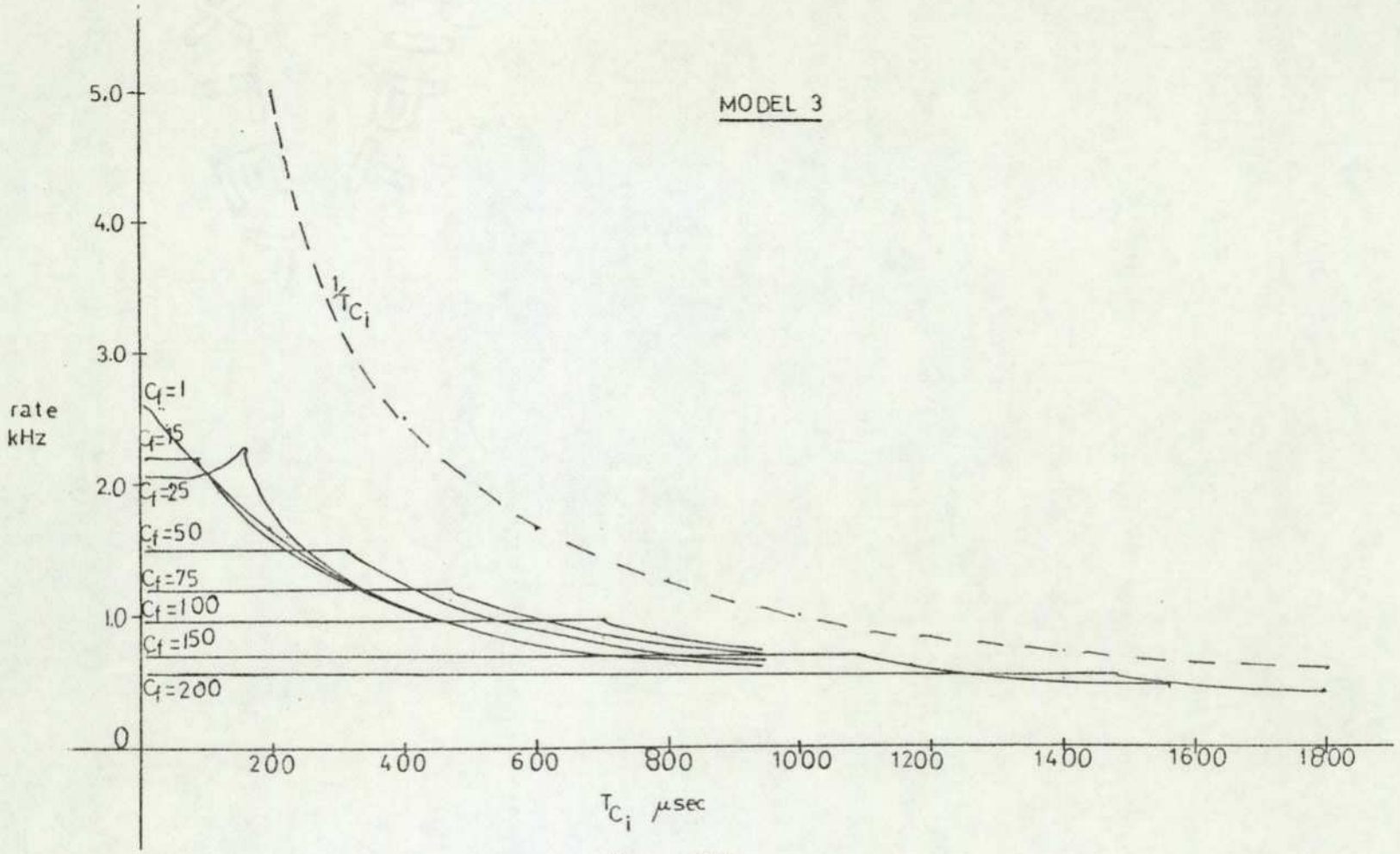
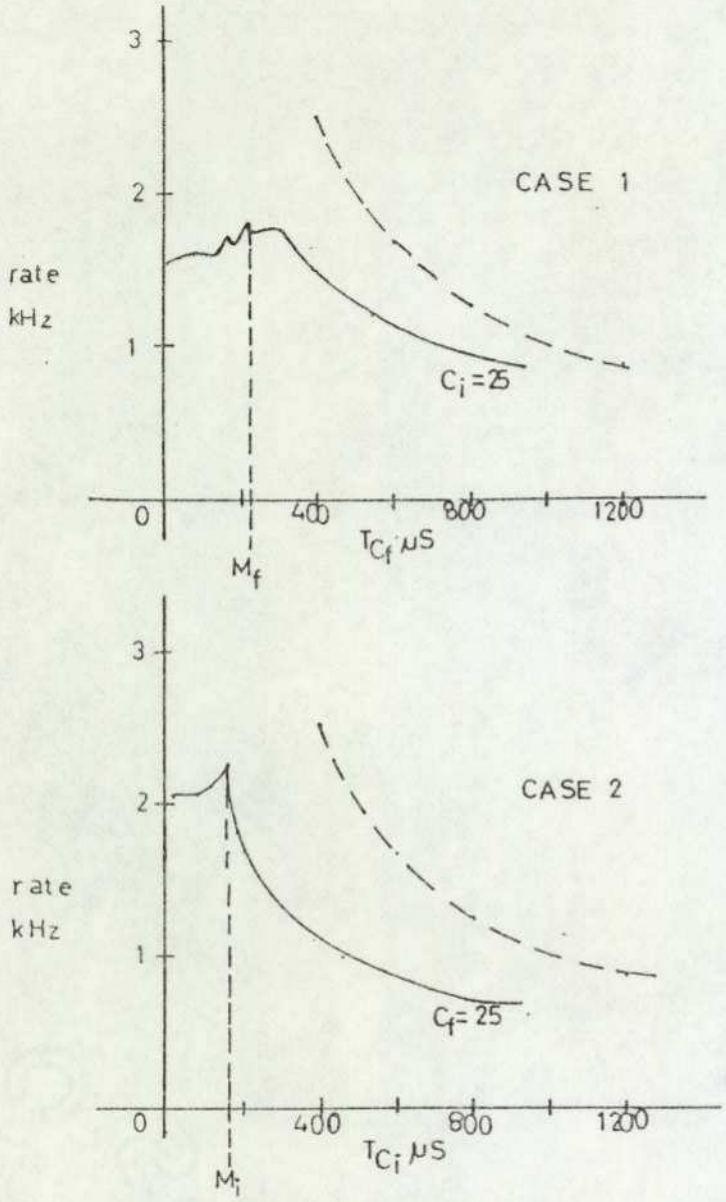

MODEL 3
5.0
4.0
3.0
2.0
1.0
0
rate
kHz
1/TCi
Cf=1
Cf=25
Cf=50
Cf=75
Cf=100
Cf=150
Cf=200
200
400
600
800
1000
1200
1400
1600
1800
TCi μsec

Figure 7-6


CASE 1
rate
kHz
3
2
1
0
400
800
1200
T_{Cf} µS
C_i=25
M_f
CASE 2
rate
kHz
3
2
1
0
400
800
1200
T_{Ci} µS
C_f=25
M_i

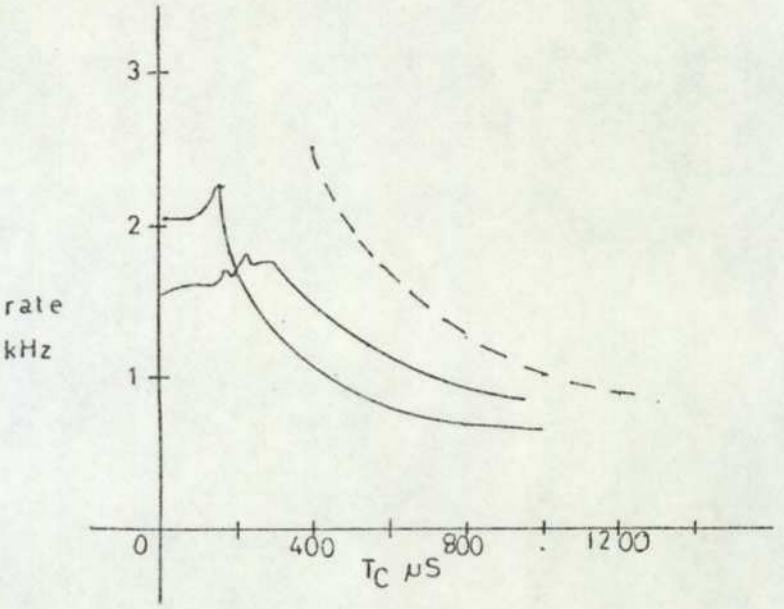

rate
kHz
3
2
1
0
400
800
1200
T_C µS

Figure 7-7

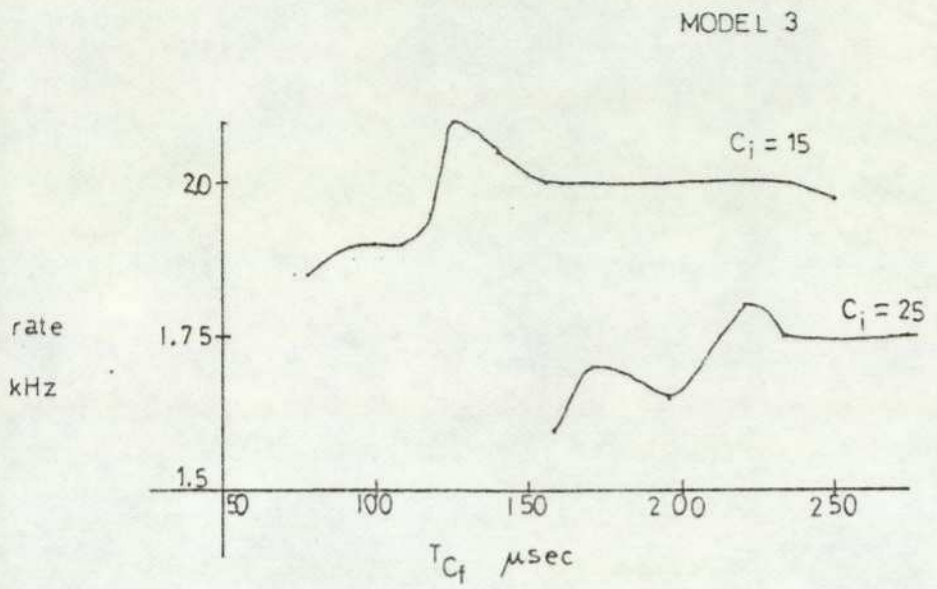


Figure 7-8

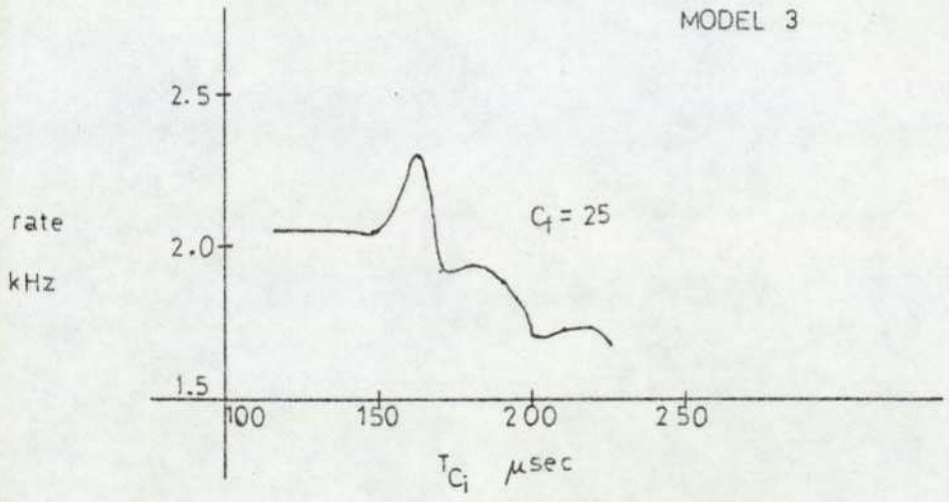


Figure 7-9

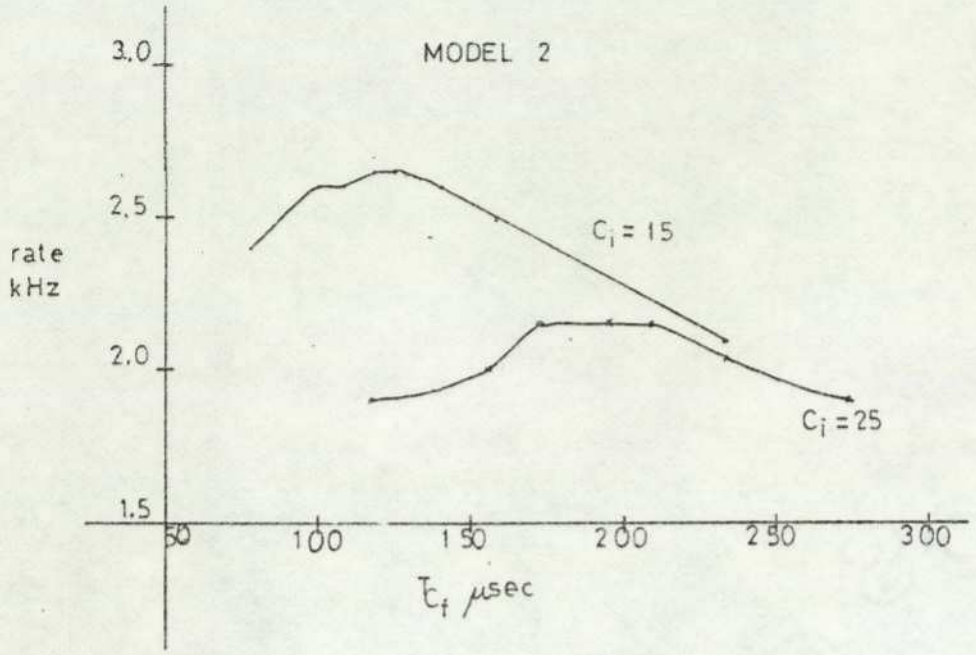


Figure 7-10

Also in model 3 it can be seen that the line $c_f = 25$ does not exhibit a steady initial state but increases to a maximum value and is thus out of step with its neighbours $c_f = 15$ and $c_f = 50$.

There is obviously some non-uniformity in the region $c_f = c_i = 25$. This may be accounted for by the fact that the sum of the number of instructions in the base loop, contention and wait is approximately

equal to twenty five for both processes. Thus the region where the count is equal to twenty five is a transition region where the count is becoming the dominant factor in the processes. At the point where the variable element of the process (count) is equal to the fixed element (base loop plus contention plus wait) it is likely that there will be additional synchronisation of the processes, causing one of them to exert a 'pulling' influence on the other.

7.2 Optimal points

Maximum sampling rates for a particular value of c_f or c_i are obtained when the corresponding c_i or c_f value is such that neither process is idling. It was decided that measurements of system behaviour would be taken at these points and they are referred to as 'optimal points' in the work that follows. Obviously one would not expect a real system to operate only at such points but they represent the optimal case of useful work done and are therefore unique among an infinite choice of other points at which measurements could be made.

Optimal points for models 2 and 3 corresponding to c_f fixed (arbitrarily) at 1, 15, 25, 50, 75, 100, 150, 200 were established by means of an initial approximation from the graphs shown in Fig. 7-2, 7-3, 7-5, 7-6 and refining these by trial and error. The results are shown in Table 7.2.

optimal point	Model 2			Model 3		
	c_f	c_i	optimal rate	c_f	c_i	optimal rate
1	1	1	3.8	1	1	2.6
2	15	13	2.8	15	13	2.25
3	25	23	2.25	25	23	1.9
4	50	47	1.55	50	47	1.4
5	75	72	1.2	75	72	1.2
6	100	95	0.98	100	95	0.98
7	150	148	0.72	150	148	0.72
8	200	198	0.55	200	198	0.55

Table 7.2

It can be seen that optimal points occur at similar values of c_f and c_i in each model and, at every optimal point, c_f is very close to c_i. This is because, at optimal points, both w_i and w_f should be zero and thus, in terms of instructions,

$$P_f = b_f + c_f + x_f = P_i = b_i + c_i + x_i$$

By chance, b_f is approximately equal to b_i and thus an uneven spread of contention, x, is the only factor which should cause c_f to differ greatly from c_i.

Point 1 is not, in fact, a true optimal point but is included for reference. The true c_i value for $c_f = 1$ should lie between zero and one. Obviously it is impossible to consider part of an instruction and this fact gives rise to a certain amount of error in establishing the optimal rates.

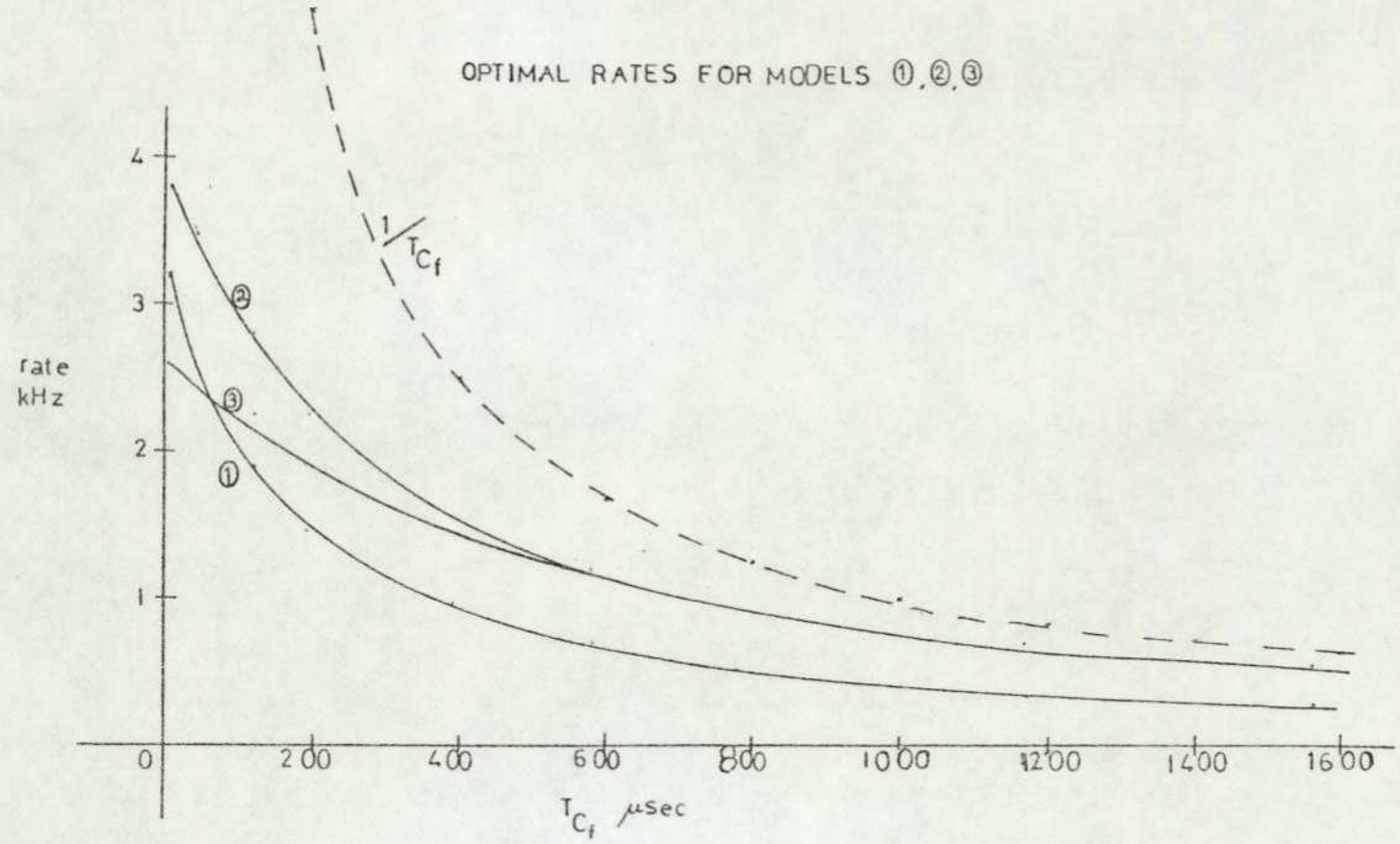

OPTIMAL RATES FOR MODELS ①,②,③
rate
kHz
4
3
2
1
0
1/T_Cf
②
③
①
200
400
600
800
1000
1200
1400
1600
T_Cf μsec

Figure 7-11

Some of the points, particularly of model 3, proved to be very difficult to measure with a number of 'beating' effects being apparent.

At points 5, 6, 7 and 8 it was impossible to detect any difference in the optimal rates of models 2 and 3. This implied that the effects causing the difference in rates for earlier points had either disappeared or had become swamped as c_f and c_i increased. This called for further investigation as discussed in 7.3.

Optimal rates for the three models are shown in Fig. 7-11.

7.2.1 Degradation/improvement in dual processor models

The optimal rates for models 2 and 3 were compared with optimal rates for model 1 (using the same c_f and c_i values) to measure the degradation/improvement of the dual processor systems over the single processor system. This is shown in tabular form in Table 7.3

Sampling rates at optimal points					
c_f	c_i	Optimal rate	2 * model 1 rate	Percentage degradation	Percentage improvement
Model 1					
1	1	3.2			
15	13	1.9			
25	23	1.45			
50	47	0.95			
75	72	0.69			
100	95	0.55			
150	148	0.38			
200	198	0.29			
Model 2					
1	1	3.8	6.4	40.6	18.75
15	13	2.8	3.8	26.3	47.37
25	23	2.25	2.9	22.4	55.17
50	47	1.55	1.9	18.4	63.16
75	72	1.2	1.38	13.0	73.91
100	95	0.98	1.1	10.9	78.18
150	148	0.72	0.76	5.3	89.47
200	198	0.55	0.58	5.2	89.66
Model 3					
1	1	2.6	6.4	59.4	-18.75
15	13	2.25	3.8	40.8	18.42
25	23	1.9	2.9	34.5	31.03
50	47	1.4	1.9	26.3	47.37
75	72	1.2	1.38	13.0	73.91
100	95	0.98	1.1	10.9	78.18
150	148	0.72	0.76	5.3	89.47
200	198	0.55	0.58	5.2	89.66

Table 7.3

Fig. 7-12 shows the percentage degradation of the dual processor system which is calculated as

$$\frac{(2 * \text{single processor rate}) - (\text{dual processor rate})}{(2 * \text{single processor rate})} * 100$$

Fig. 7-13 shows the percentage improvement of the dual processor system which is calculated as

$$\frac{(\text{dual processor rate}) - (\text{single processor rate})}{\text{single processor rate}} * 100$$

In terms of the sampling rate attainable, model 2 shows a consistent improvement over model 1, ranging from approximately nineteen per cent for the most closely coupled case to approximately ninety per cent for the least closely coupled case.

Model 3, however, attains a lower sampling rate than model 1 for the most closely coupled case: the dual processor has been so degraded that it is achieving less than the single processor. As the process coupling becomes looser, model 3 approaches and becomes identical with model 2 in performance.

Model 3, with it s single semaphore is inherently more closely coupled than model 2 and Fig. 7-12 demonstrates the relationship between degradation and coupling.

7.2.2 Effect of buffer size

The models were run with different sizes of input and output buffer but there proved to be very little difference in optimal sampling rates other than for very small buffers. Rates for model 2 at optimal points 2 (c_f = 15, c_i = 13) and 4 (c_f = 50, c_i = 47) are shown below in Table 7.4.

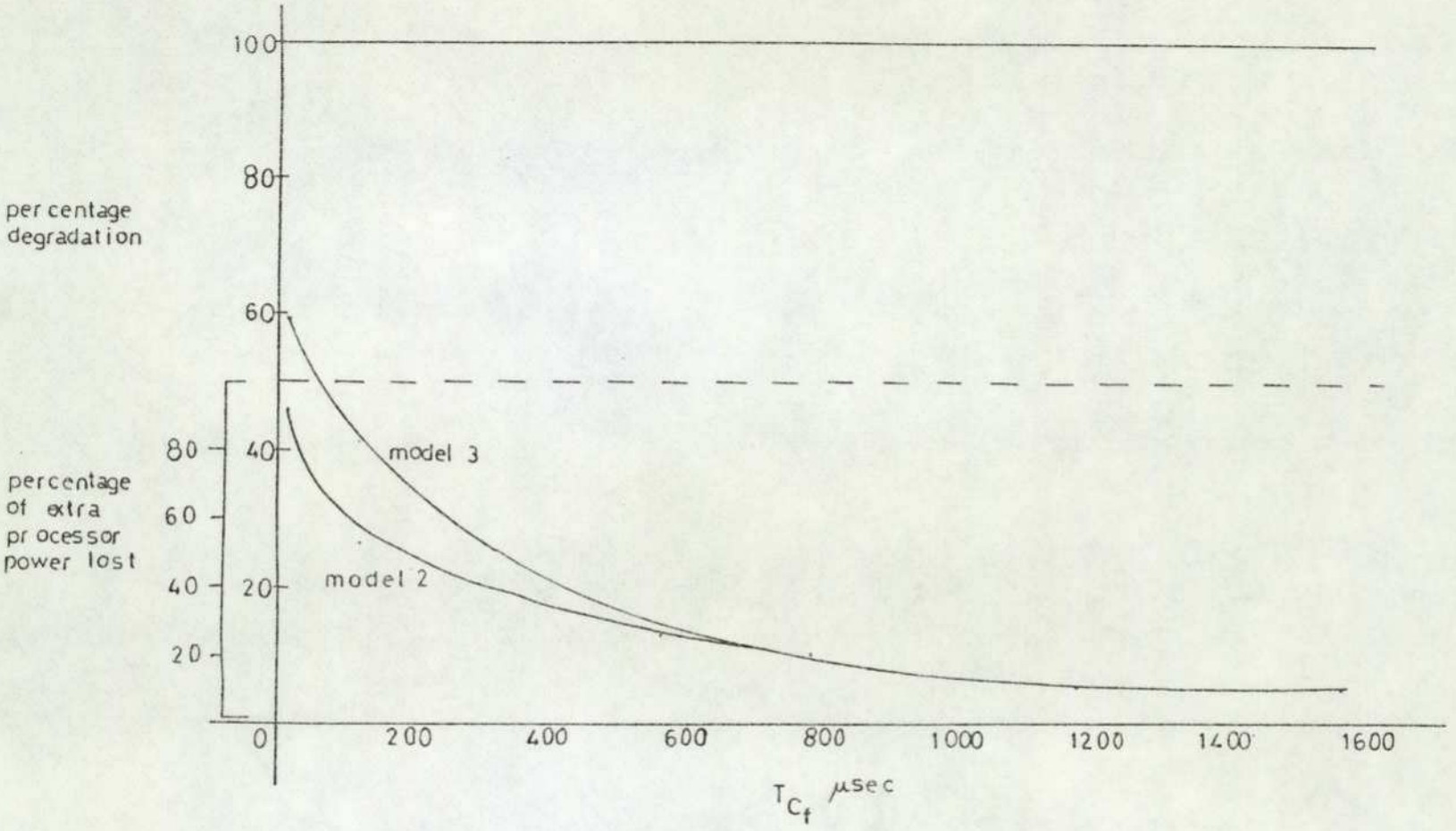

100
80
60
40
20
0
percentage degradation
percentage of extra processor power lost
80
60
40
20
model 3
model 2
200
400
600
800
1000
1200
1400
1600
T_{C_f} μsec

Figure 7-12

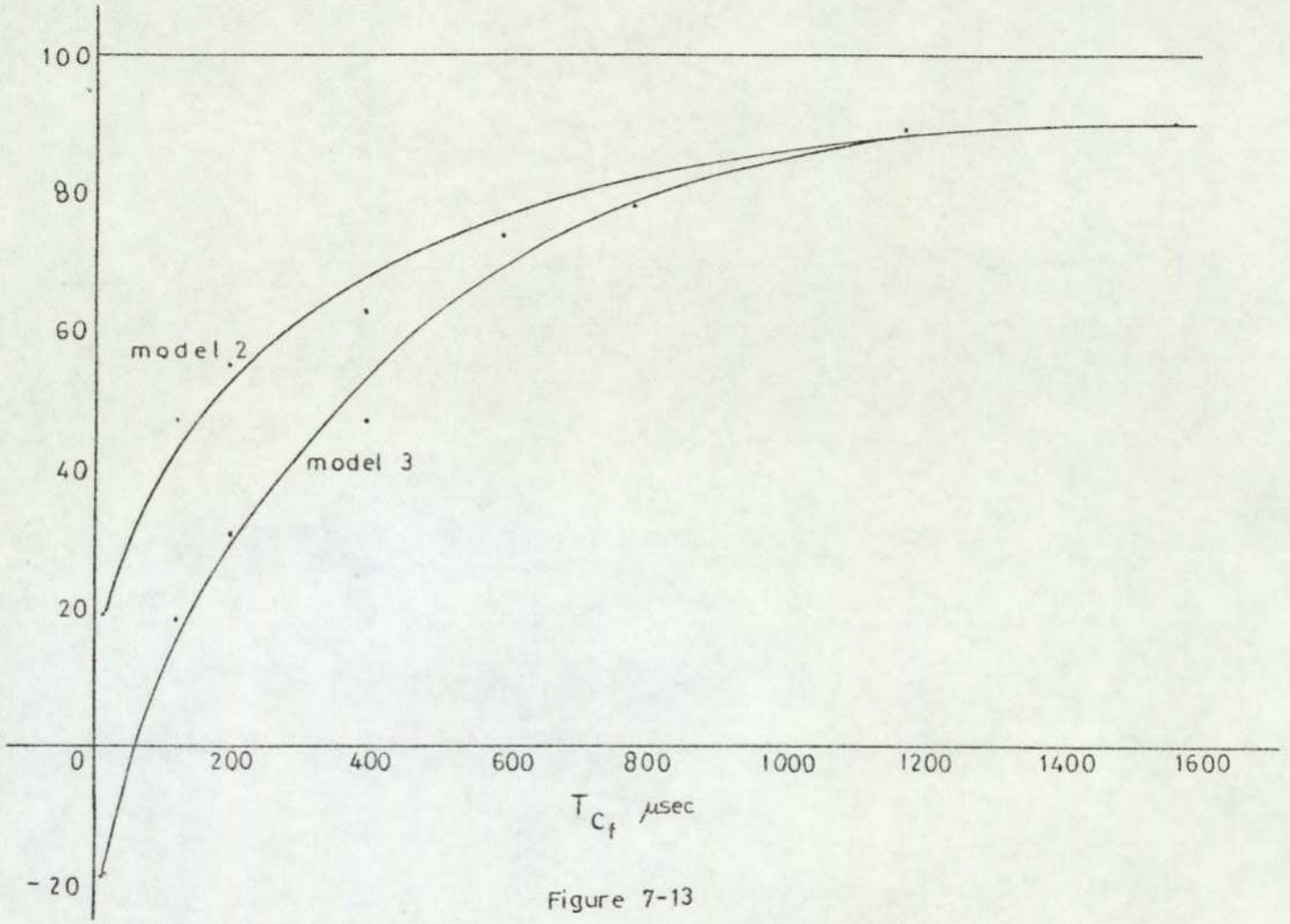

100
80
60
40
20
0
-20
model 2
model 3
percentage improvement
200
400
600
800
1000
1200
1400
1600
T_{C_f} μsec

Figure 7-13

buffer size	rate in kHz at point 2	rate in kHz at point 4
1	2.64	1.50
4	2.76	1.54
8	2.78	1.55
16	2.80	1.55
32	2.80	1.55
64	2.80	1.55
128	2.80	1.55
256	2.80	1.56

Table 7.4 Variation of sampling rate with buffer size

The lack of effect of buffer size was confirmed by the repeated halting of the processors, and inspection of the buffers, while the models were running. They were found to be either empty or holding a single sample on all occasions.

This is to be expected in a synchronised model having regular input, and regular program length, where the buffers have no smoothing effect to exercise.

7.3 Measurement of software contention

Contention occurs when more than one process attempts to access an area of shared memory at the same time. A measure of the contention occuring in a process may be obtained from the number of 'unsuccessful' P operations on the semaphores which enforce mutual exclusion on the critical sections of code accessing shared memory.

By means of the hardware monitor such code sequences can be trapped and counted. In model 2 there were four such P operations, two each on the semaphores S1 (controlling the input buffer) and S0 (controlling the output buffer). The following measurements (Tables 7.5 and 7.6) were taken in which, for example, a zero value would indicate that P operations were immediately successful (and so there was no contention) and a one value would indicate that, on average, a P operation was only successful at the second attempt i.e. the sequence of instructions executed would be

```
L1-  JSC   0   S   L2  ┐
     JMP               .L1 ┘ unsuccessful P operation
L1-  JSC   0   S   L2      successful P operation
L2-
```

7.3.1 Contention in model 2

optimal point	filter process		i/o process	
	S1(input)	S0(output)	S1(input)	S0(output)
1	0.74	0	0	0
2	0.74	0	0	0
3	0.88	0	0	0
4	0.66	0	0	0
5	0.69	0	0	0
6	0.69	0	0	0
7	0.85	0	0	0
8	0.77	0	0	0
average:	0.75	0	0	0

Table 7.5

It can be seen from Table 7.5 that there is only contention associated with the semaphore controlling access to the input buffer from the filter process.

The contention represented, generally, a single retry on a P operation. For example, the results for point 2 for the S1 semaphore in the filter process were:

number of retries	0	1	2	3	4	5	6	7	8
occurrence	11333	32767	42	0	0	0	0	0	0

This is also illustrated in Fig. 7-14.

7.3.2 Contention in model 3

In model 3, it was impossible for the hardware monitor to distinguish between the contention associated with the input buffer and that associated with the output buffer, as access to both buffers was controlled by the same semaphore, and therefore by identical code. Thus the equivalent results to those obtained for model 2 gave the average contention for both P operations in a process.

optimal point	filter process	i/o process
	Sl(input + output)	Sl(input + output)
1	3.15	3.13
2	2.15	2.26
3	2.17	2.23
4	2.25	2.28
5	2.27	2.28
6	1.95	2.61
7	2.20	2.29
8	2.08	2.29

Table 7.6

The results showed a distribution (Fig. 7-14) in which the majority of P operations were either immediately successful or else needed to retry four or five times.

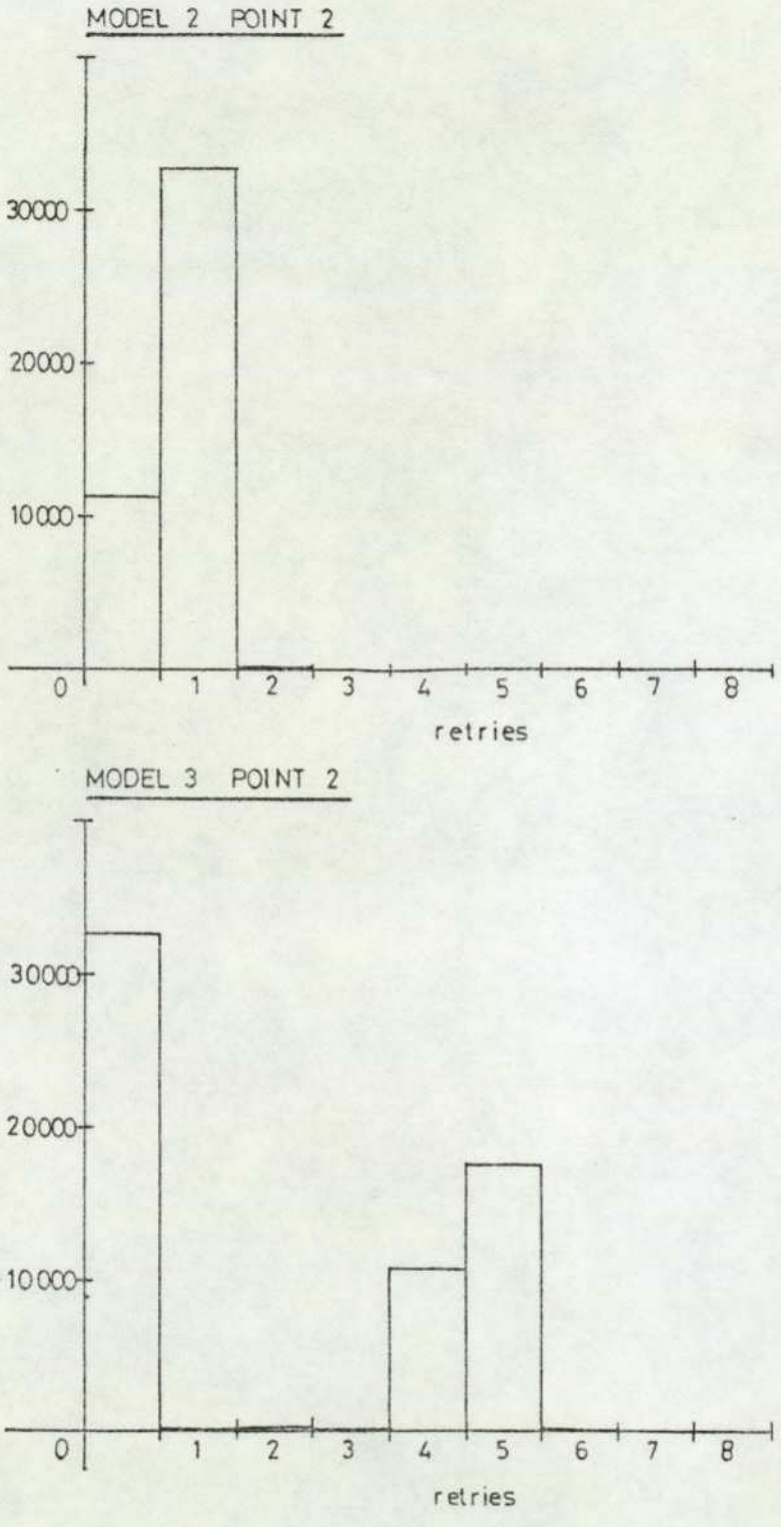
For example, the results for point 2 were as follows:

filter process

number of retries	0	1	2	3	4	5	6	7	8
occurrence	32767	5	165	22	10801	17651	3	1	0

i/o process

number of retries	0	1	2	3	4	5	6	7	8
occurrence	32767	121	1197	129	18001	15658	10	2	0



contention on S1 in filter process

Figure 7-14

Repeated measurements with a logic analyser (maximum trace 256) on the buses indicated that in the filter process there was no contention associated with the output buffer but that it took five or six attempts to enter a critical section associated with the input buffer. On the other hand, in the i/o process there was no contention associated with the input buffer but it took five or six attempts to enter a critical section associated with the output buffer.

As the number of successful semaphore operations must be evenly split between control of the input buffer and control of the output buffer (assuming a straight forward path through the program with no 'short loops' - see 7.4) on the basis that samples entering the process must emerge from it, the following assumptions were made about the processes' behaviour.

Using the filter process results, as above, as an example:

(i)total successful P operations = $\sum_{\substack{\text{retry} \\ \text{values}}}$ occurrence

=61420

(ii)assuming that there is no contention associated with the input buffer then this accounts for the 30710 (= $61420/_2$) lowest occurrence values - in this case all zero number-of-retries. All remaining values i.e.

number of retries	0	1	2	3	4	5	6	7	8
occurrence	2057	5	165	22	10801	17651	3	1	0

which total 30710, are associated with the output buffer.

(iii) for the i/o process, low contention values are associated with the output buffer instead of the input buffer.

Working on this basis, Table 7.7 below was obtained from Table 7.6 above.

Contention in model 3

optimal point	filter process		i/o process	
	Sl(input)	Sl(output)	Sl(input)	Sl(output)
1	1.99	4.32	4.49	1.77
2	0	4.3	4.45	0.06
3	0	4.34	4.45	0.01
4	0	4.5	4.34	0.21
5	0	4.55	4.48	0.07
6	0	3.9	4.51	0.72
7	0	4.4	4.49	0.09
8	0	4.16	4.54	0.23
average	0.24	4.31	4.47	0.40

Table 7.7

7.3.3 Factors affecting contention

Taking into account the fact that point 1 is known to be slightly off-optimal, it can be seen from the results that

(i) protecting the shared memory with fewer semaphores gives rise to greater contention.

(ii) the absolute contention in a particular model is not affected by the closeness of coupling of the processes in that model but is a function of their synchronism.

(iii) results for model 2 suggest that the processes are synchronised in such a way that the filter process is requesting data just before the i/o process has finished its 'input' phase (Fig. 7-15).

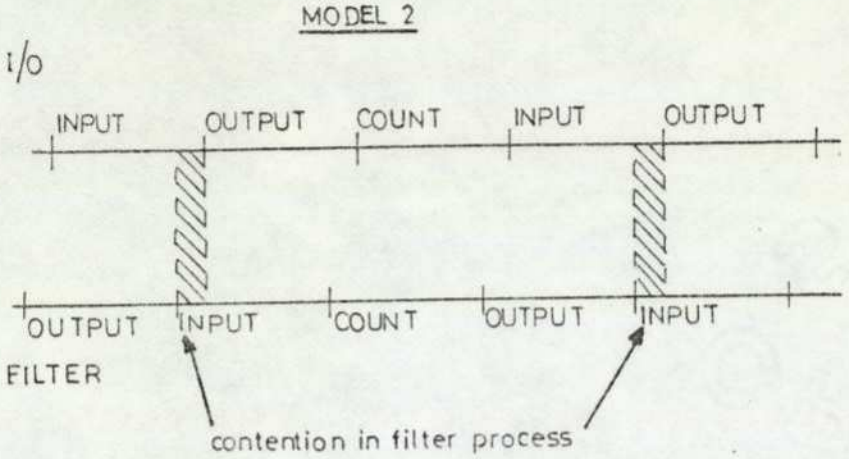


Figure 7-15

(iv) results for model 3 suggest that the model is synchronised so that the i/o process attempts to start it's 'input' phase as the filter process is accepting a previous sample and that the filter process attempts to store a filtered sample as the i/o process is outputting a previous sample (Fig. 7-16).

MODEL 3

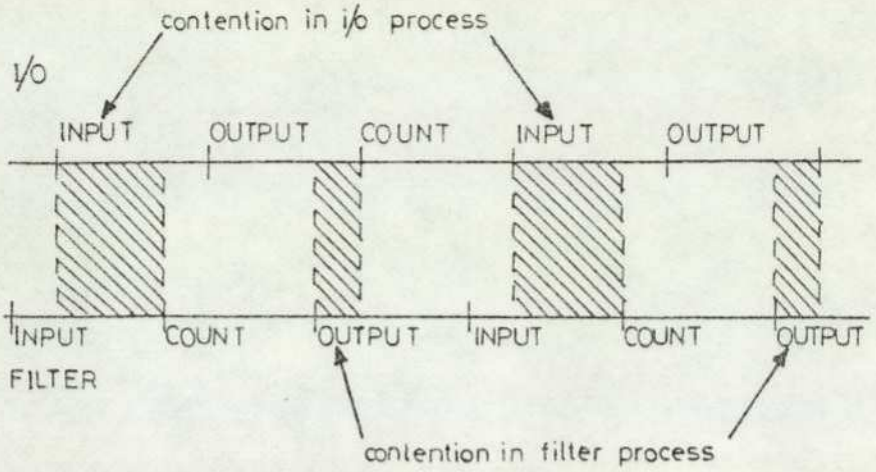


Figure 7-16

As the input and output phases are now controlled by the same semaphore, any overlap will cause contention, unlike model 2 where input must overlap with input or output with output to cause contention.

(v) As the relative durations of the input, output and count phases alter, the processes may slip out of synchronism which will affect the level of contention in the system. The process is however circular in that the contention also contributes to the relative durations, so the relationship is complicated. Irregularities such as those discussed in 7.1.4 are therefore to be expected.

7.4 Program execution paths

The possible execution paths through the programs of models 2 and 3 are illustrated by Fig. 7-17. Execution of a 'short loop', by branching on a JZE instruction, occurs either when a process is waiting for data (filter process input, i/o process output) or when it is waiting for buffer frames (filter process output, i/o process input). These short loops, embedded in their corresponding 'long loops', are the only possible variation in program path other than that caused by looping on P operations.

Logic analyser traces taken to establish contention measurements (7.3.2) had already indicated that short loops were not usually executed at optimal points. The hardware monitor was used to confirm this by trapping instruction sequences (SBS's) to show the proportion of 'long' loops being executed. The results are shown in Table 7.7. Their method of calculation is described in 6.4.

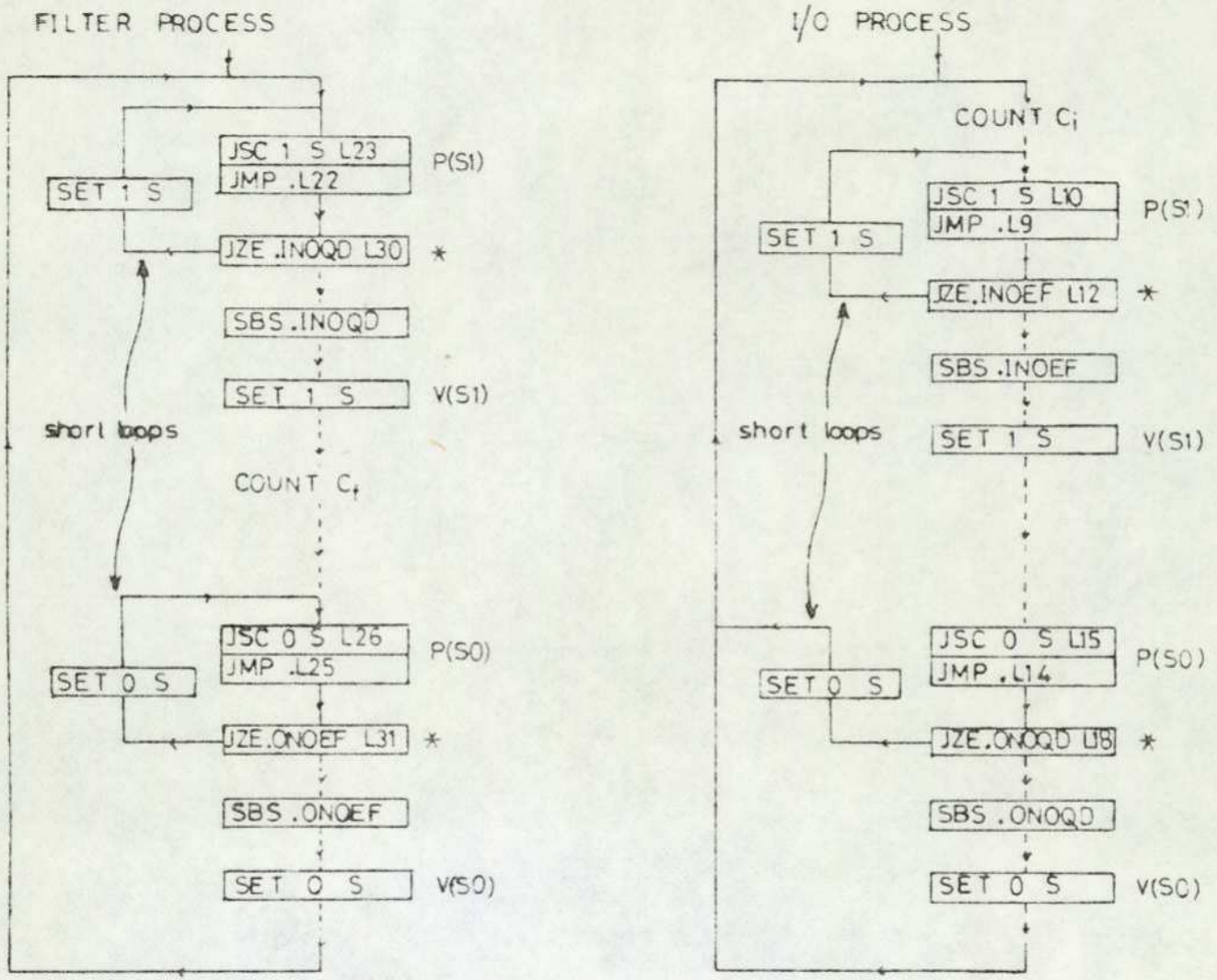


Figure 7-17 EXECUTION PATHS

Model 2

percentage of times execution path is not through

optimal point	filter process: first short loop	filter process: second short loop	i/o process: first short loop	i/o process: second short loop
1	67.5*	99.9	101.0	102.0
2	99.0	99.4	99.3	101.0
3	97.0	100.0	102.0	110 †
4	99.4	99.7	99.9	98.6
5	98.7	99.4	100.0	100.0
6	101.0	99.9	99.7	98.6
7	82.7*	99.8	101.0	101.0
8	71.6*	100.0	101.0	108 †

Model 3

percentage of times execution path is not through

optimal point	filter process: first or second short loop	i/o process: first or second short loop
1	97.4	97.8
2	96.3	101.0
3	95.9	100.3
4	98.2	103.0
5	97.5	102.0
6	85.6*	102.4
7	91.4	96.8
8	88.0*	99.6

Table 7.7

With the exception of those values marked with an asterisk, the results were interpreted as showing that the programs never executed short loops (although there is a large error in two values, †).

Point 1 is known not to be quite optimal and so some short loops are to be expected as the filter process waits for data.

Points 7 and 8 in model 2 and points 6 and 8 in model 3 are to a lesser extent also exhibiting short loops indicating that, for these points, the values chosen for c_i are in error by, on average, a fraction of one instruction.

7.5 Process coupling

The behaviour of the models having been established, in terms of contention and program path, the dynamic coupling of the processes can be calculated in terms of the percentage of the instructions which access shared memory.

optimal point	percentage of instructions accessing shared memory	
	Model 2	Model 3
1	43	48
2	29	34
3	23	28
4	16	20
5	12	15
6	9	12
7	7	9
8	5	7

Table 7.8

Using Brinch Hansen's (1978 d) figure of 10% as the lower bound of close coupling it can be seen that the models lie largely within the range (Fig. 7-18).

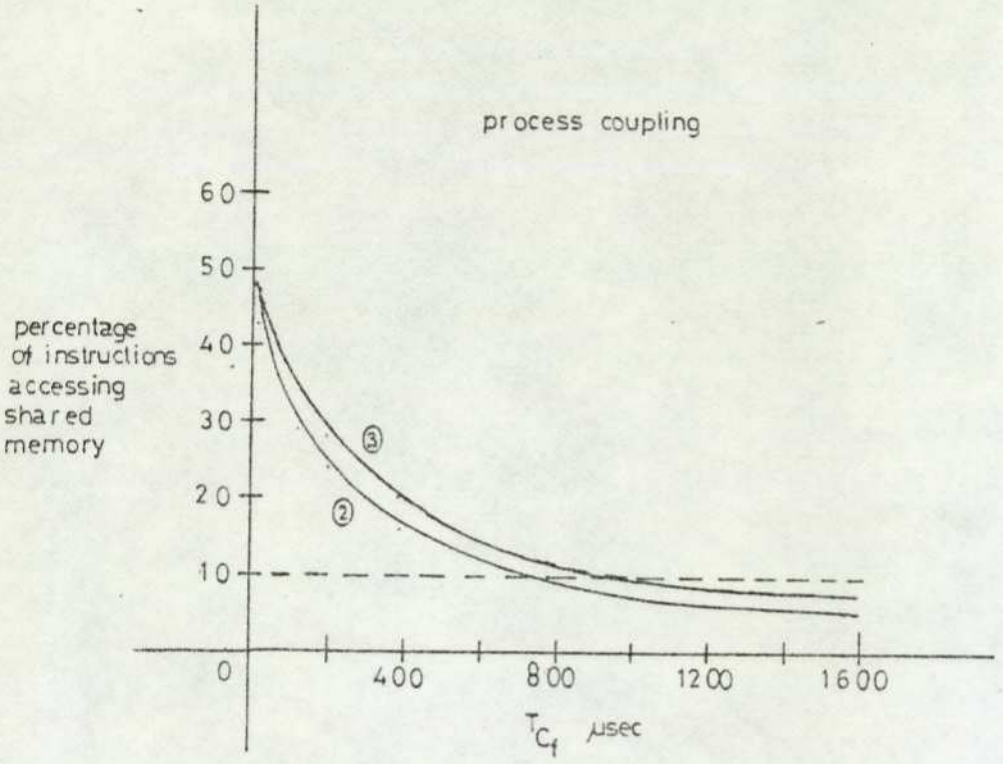


Figure 7-18

As model 3 is controlled by one less semaphore than model 2 it is obviously more closely coupled. This is manifested in the higher number of shared memory accesses resulting from greater contention caused by the additional degree of synchronisation.

7.6 Analysis of degradation

From Figs. 7-12 and 7-13 earlier, it can be seen that beyond the region where the models cease to be closely coupled (around T_{c_f} = 780 to 1170 μsecs) the degradation/improvement graphs show little change.

Once the exact pattern of dynamic behaviour of the programs had been established, the next step was to analyse the degradation present in the system and attribute it to its constituent causes.

Three distinct causes of degradation were identified in the system:

(1) instructions executed to perform successful semaphore operations.

(2) software contention i.e. instructions executed in performing unsuccessful semaphore operations.

(3) hardware delays arising from bus contention in access of shared memory.

To attach a value to these factors required specific knowledge of individual instruction times.

7.6.1 Instruction times

An average instruction time was calculated from the static sum of the times of the instructions in the main loops of the filter and i/o processes.

average instruction time = 6.5 μsec

Counts c_f and c_i consist of repeated execution of an '1CZ' instruction which takes 7.8 μsec. When the model was running with high counts the average instruction time was increased by the preponderance of these longer instructions as shown in Table 7.9.

optimal point	model 2	model 3
1	6.5	6.5
2	6.9	6.9
3	7.1	7.0
4	7.3	7.2
5	7.4	7.4
6	7.5	7.4
7	7.6	7.5
8	7.6	7.6

Table 7.9 Average instruction times (μsec)

Delays in instruction execution are caused by the filter process (primary processor) accessing shared memory (which is on the secondary bus). The buses are connected by a pair of interface sets (4.1.2) and a primary processor read/write on shared memory appears as a DMA request on the secondary bus (Fig. 7-19).

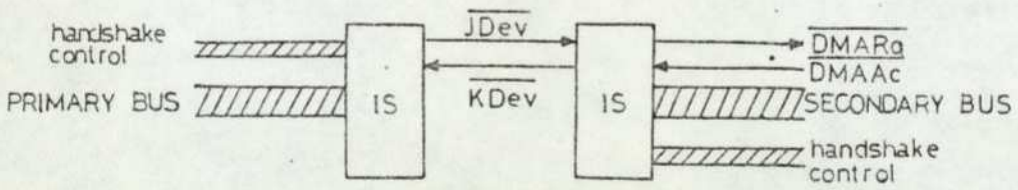


Figure 7-19 PRIMARY PROCESSOR ACCESSING SHARED MEMORY

The DMA request will be serviced by the secondary processor, either during instruction decode or, for certain instructions, during instruction execution. Thus, two variable length delays are engendered in the system (Fig. 7-20):

(1) the primary processor waits for it s DMA request to be accepted.

(2) when the DMA request has been accepted, the primary processor will hold the secondary bus for the duration of it s instruction execution, so causing the secondary processor to wait.

In order to establish bounds on these waits, measurements on $\overline{JAcv}$ and KPas for best and worst cases were made with a transient recorder. This was a Bromation 8100 sampling at 100 nsecs per sample, so all edges were $\pm$ 50 ns.

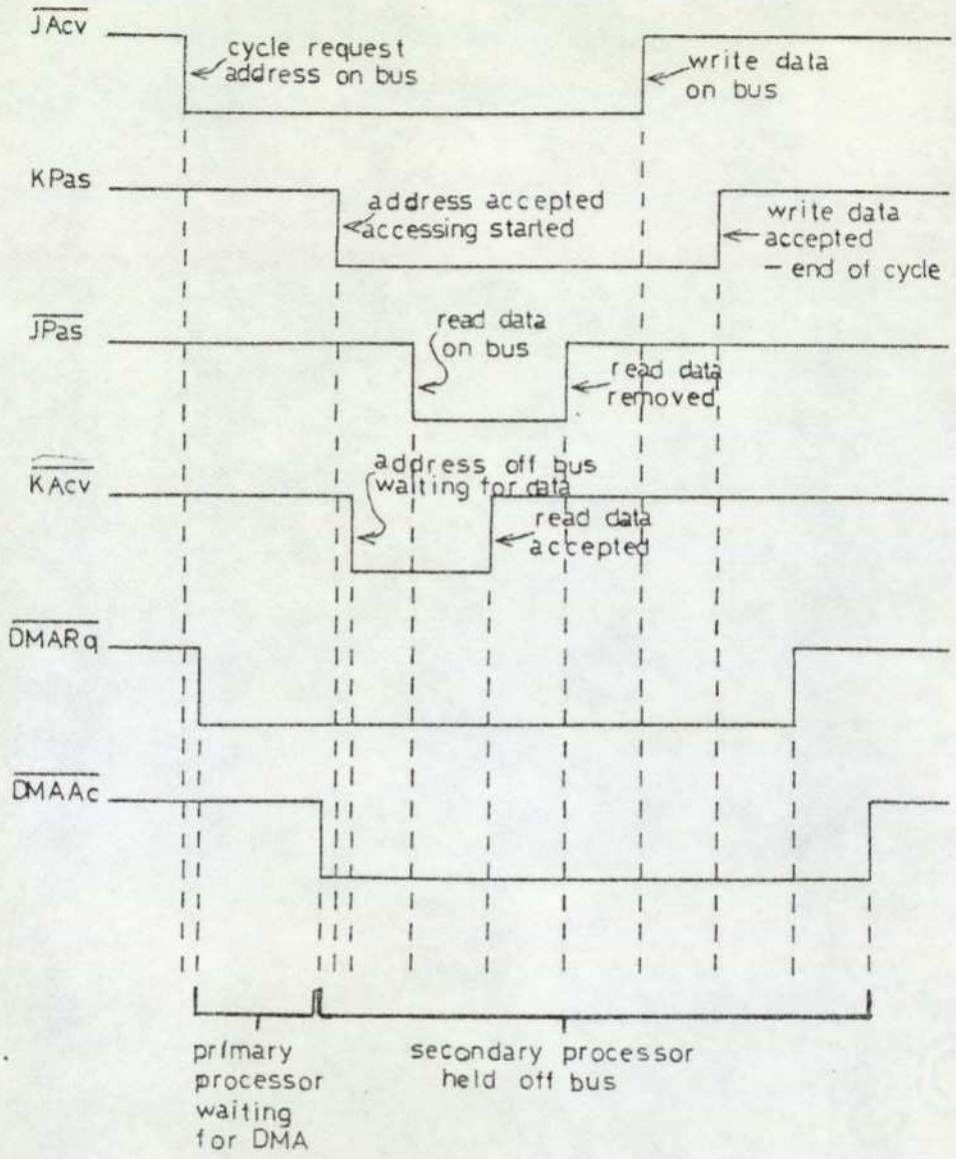
7.6.1.1 DMA delay in primary processor

Three cases were measured:

(1) an instruction which does not access shared memory

primary code	secondary code
L- STO .N	HLT
JMP .L	

STO .N was measured where N was not in shared memory. The secondary processor was halted so that it could not affect the primary processor.



R/M/W cycle for primary processor accessing shared memory

Figure 7-20

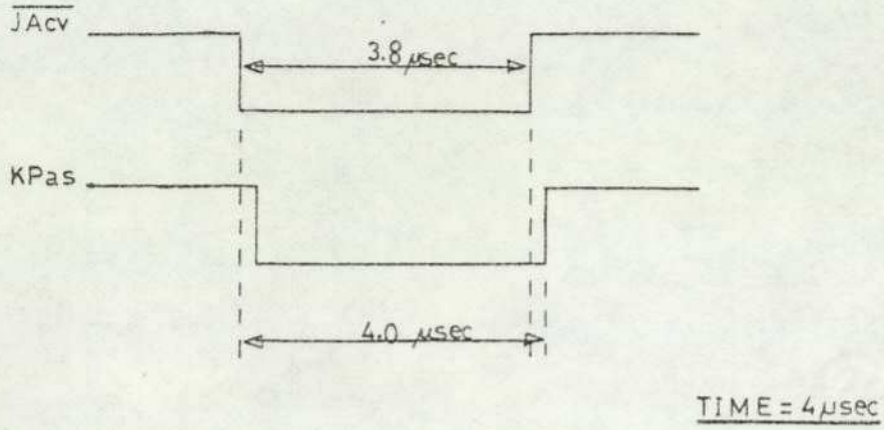


Figure 7-21

(2) an instruction which does access shared memory

primary code	secondary code
L- STO .NN	HLT
JMP .L	

STO .NN was measured where NN was in shared memory. The secondary processor was halted so this gives the best case, where the DMA request should be accepted immediately.

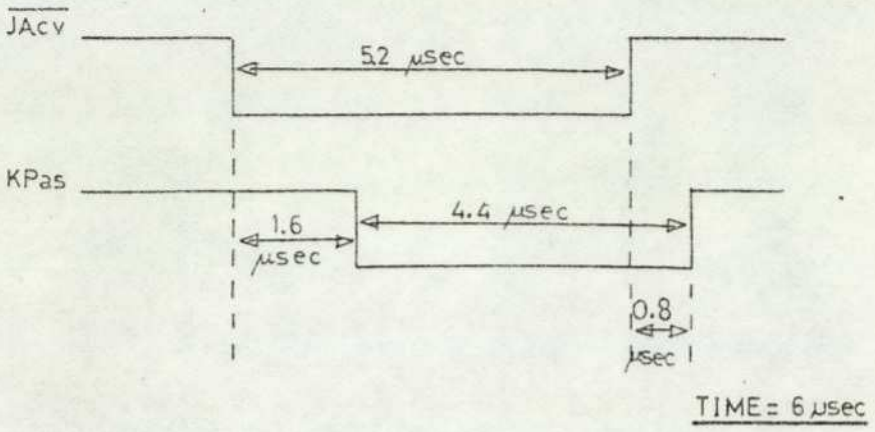


Figure 7- 22

The delay caused by the DMA was 1.6 μsec and the remainder of the instruction was extended by 0.4 μsec.

(3) an instruction which accesses shared memory with the secondary processor running.

primary code		secondary code			
L-	STO .NN	L1-	JSC	O N	L2
	JMP .L	L2-	SET	O N	
			JMP	.L1	

STO .NN was measured where NN was in shared memory. A long (3-word) JSC instruction was chosen for the secondary code to find a worst case. By inspection, the longest cycles were identified and measured.

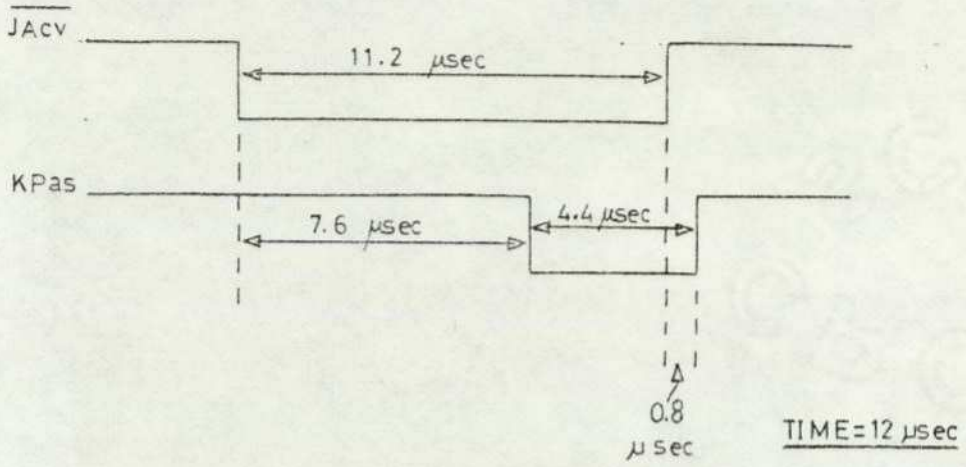


Figure 7-23

This case differed from (2) above only in that the DMA time was increased from 1.6 μsec to 7.6 μsec. Therefore, the execution time for a store operation in private memory = 4 μs. The average execution time for a store operation in shared memory = (6.0 + 12.0)/2 = 9 μs.

Accessing shared memory added, on average, 5 μs to instruction execution time.

Fig. 7-24 shows an actual plot of signals for the case where the primary processor is executing the code

address of instruction	instruction
2048	STO .3072
2050	JMP .2048

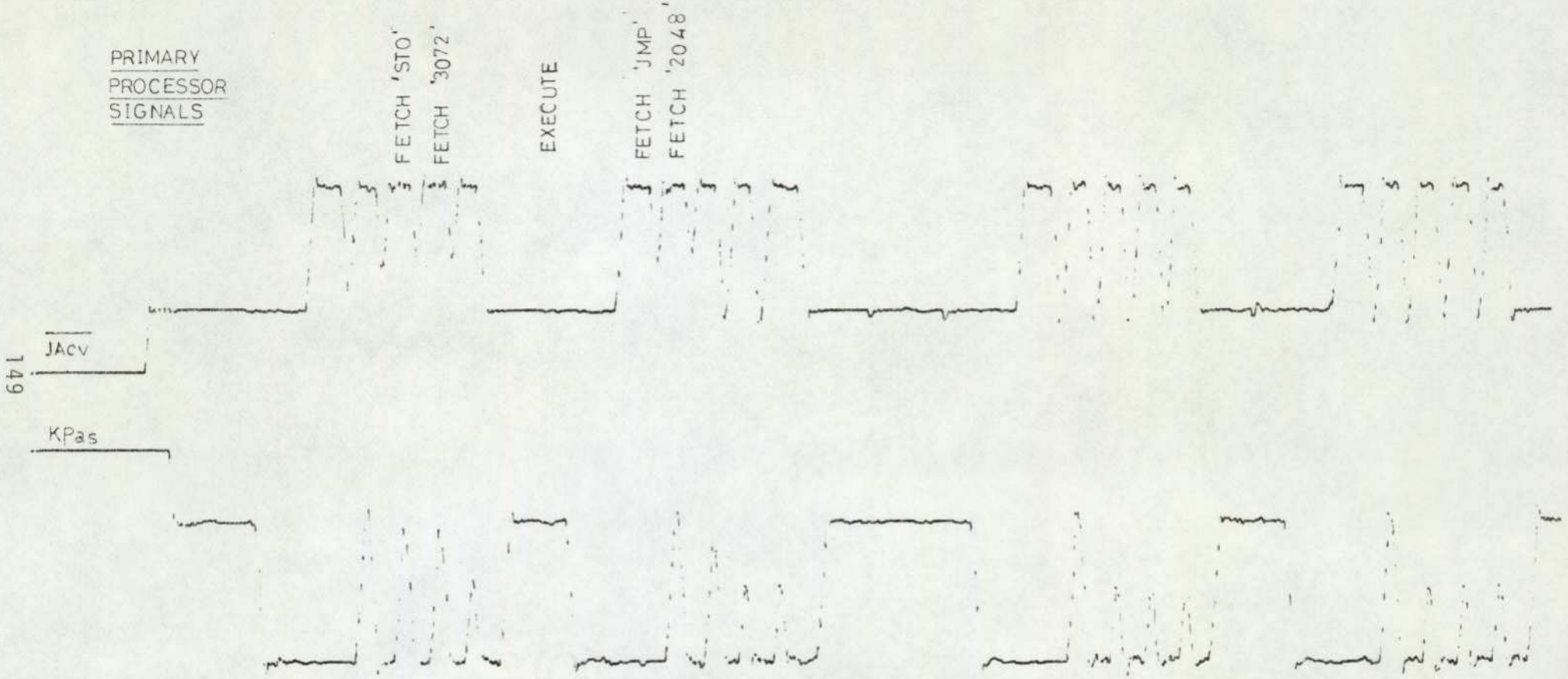

PRIMARY
PROCESSOR
SIGNALS
FETCH 'STO'
FETCH '3072'
EXECUTE
FETCH 'JMP'
FETCH '2048'
JAcv
149
KPas

Figure 7-24

7.6.1.2 DMA delay in secondary processor

When the primary processor is holding the secondary bus, the secondary processor will be delayed unless it is executing an instruction which does not use the bus. Such instructions (e.g. logical shifts) are, however, very short so the following instruction fetch would then be delayed. The extent of the delay depends largely upon the type of instruction, 'read' (R) or 'read/modify/write' (RMW), being executed by the primary processor.

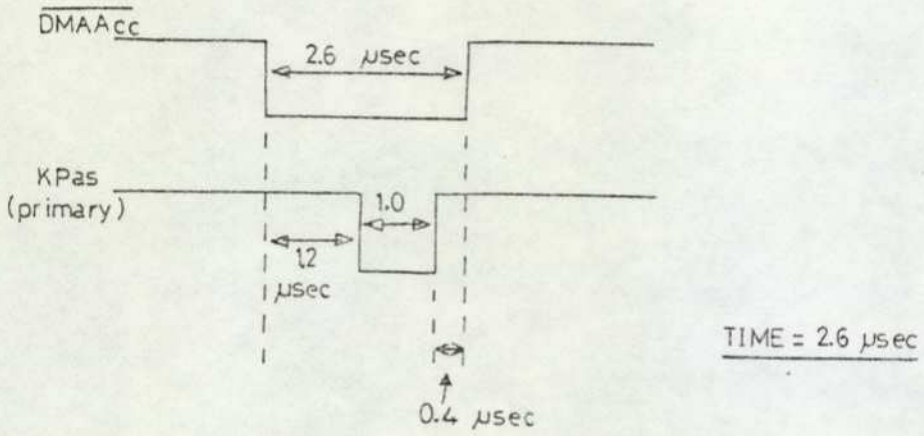


Figure 7-25

Theoretical worst cases (derived from 7.6.1.1) are shown in Fig. 7-25, where the primary processor is executing a 'read' instruction, and in Fig. 7-26, where the primary processor is executing a 'read/modify/write' instruction. The potential delays in these cases are 2.6 μs and 6.0 μs respectively.

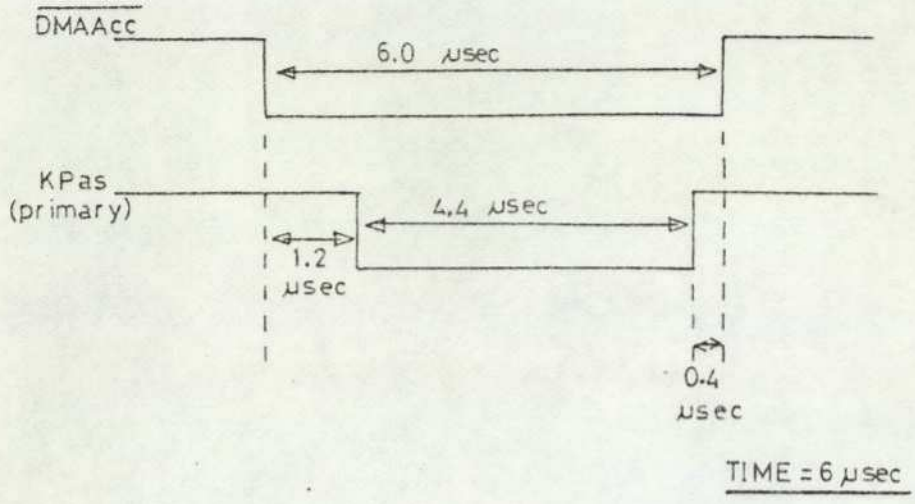


Figure 7-26

When measurements were taken for a read (LDA .NN) instruction, and a long read/modify/write (JSC) instruction, the actual delays generated were found to be 2.2 μs and 5.6 μs respectively.

In models 2 and 3, the majority of instructions in the filter process which accessed shared memory were read/modify/write instructions. The ratio (approximately 1:10) between read and read/modify/write instructions was used to calculate an average delay of 5.2 μs.

Thus the total average delay generated in the system by an access of shared memory by the filter process was 10.2 μs (5μs in filter process and 5.2 μs in the i/o process).

It should be noted that there will be errors in these figures caused by both inaccuracies in the transient recorder and the variation in the delay due to DMA cycles.

7.6.2 Degradation in models 2 and 3

An analysis of the degradation in the models was made by establishing, for each optimal point, the number and type of instructions executed in a typical program cycle. This was made possible by knowledge of the execution paths (7.4) and measurement of the contention on each semaphore (7.3). By use of the instruction execution times (7.6.1) it was possible to estimate the time per program cycle devoted to

(a) implementations of P (JSC B S LABEL) and V (SET B S) on semaphores

(b) contention in the form of busy waiting during P operations

```
LABEL1-  JSC  B  S  LABEL2
         JMP       .LABEL1
```

(c) other instructions

(d) instructions (of (a), (b) and (c) above) involving access to shared memory from the filter process, and hence the additional DMA cycle delay incurred (which had been ignored in (1), (2) and (3) above)

Times for (a), (b) and (d) were calculated as a percentage of the total execution time and as a percentage of the total degradation time (see Appendix 5 for details). The results are summarised in Fig. 7-27 to 7-30.

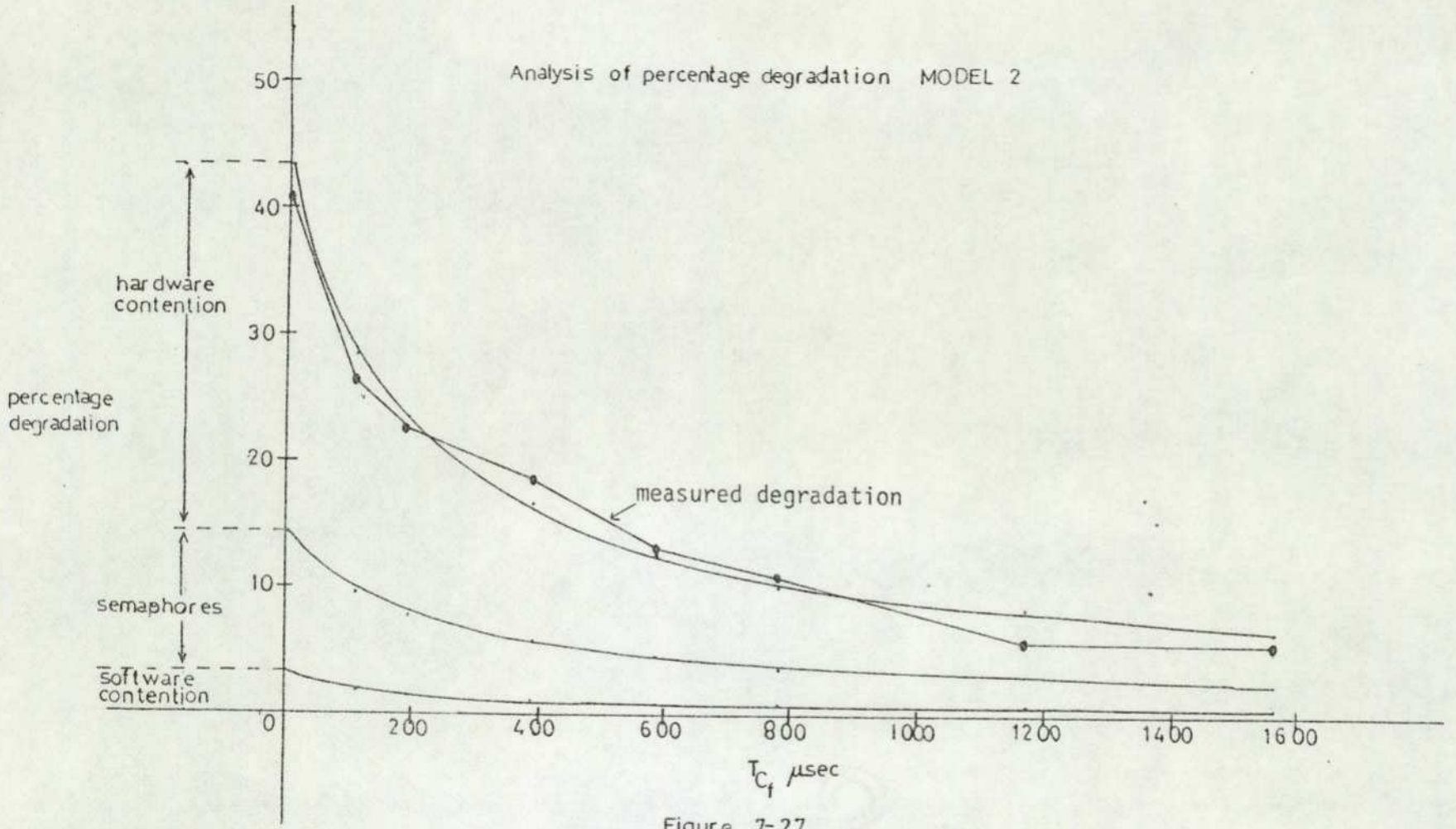

Analysis of percentage degradation MODEL 2
50
40
30
20
10
0
hardware contention
percentage degradation
semaphores
software contention
measured degradation
200
400
600
800
1000
1200
1400
1600
T_{C_1} μsec

Figure 7-27

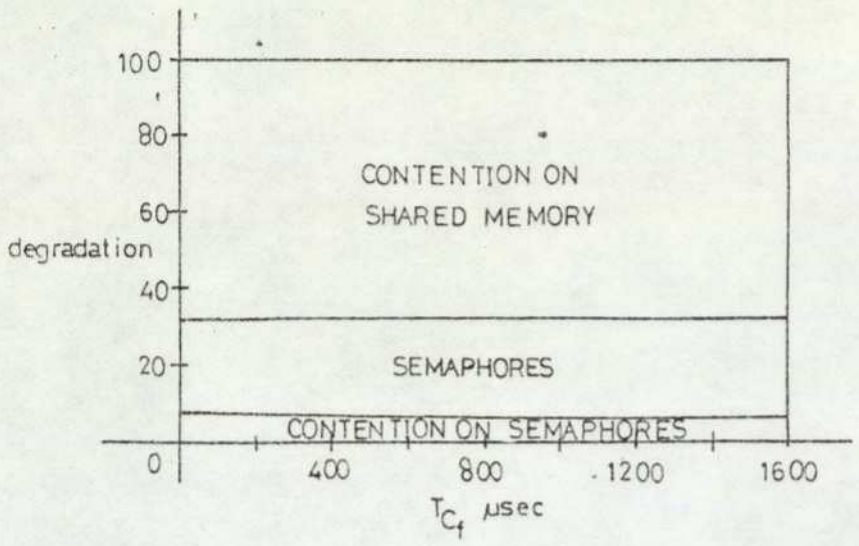


Figure 7-28 ANALYSIS OF DEGRADATION — MODEL 2

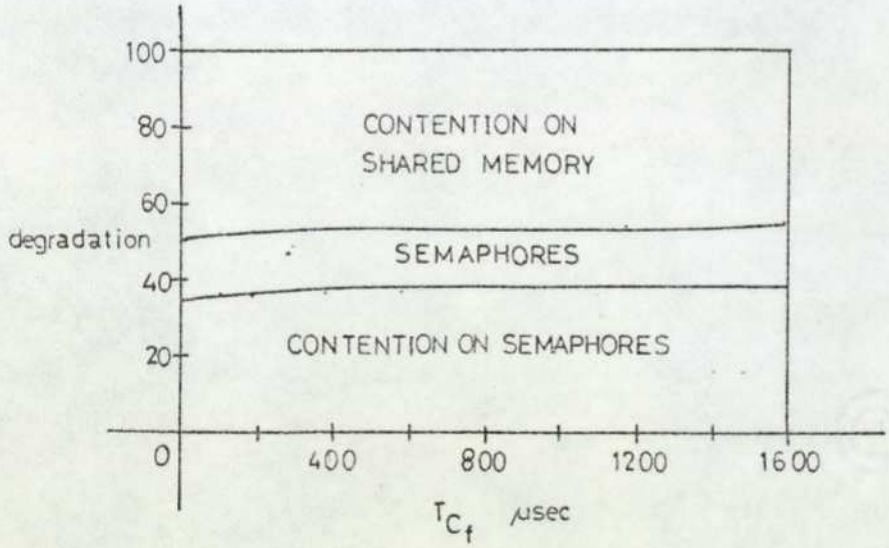


Figure 7-29 ANALYSIS OF DEGRADATION — MODEL 3

Fig. 7-28 shows that the degradation in model 2 is approximately 6% due to contention on semaphores, 29% due to semaphore (P and V) operations and 65% due to contention in accessing shared memory. Degradation calculated on this basis (Fig. 7-27) agrees extremely well with the measured degradation (7.2.1), which suggests that an adequate model of the system has been achieved.

Figs. 7-29 and 7-30 show corresponding results for model 3. Contention on semaphores now accounts for approximately 37% of the degradation, semaphore operations for 15% and contention in accessing shared memory for 47% of the degradation. The calculated degradation, however, agrees less well with the measured degradation than was the case for model 2.

We would, in general, expect model 3 to be less accurate than model 2 as the higher level of contention on semaphore operations generates a greater number of variable length DMA cycles. When expressed as a percentage however one would expect this variation in accuracy to be more evident for the more closely coupled optimal points, which is not the case.

The accuracy of the measured sampling rates is obviously dependent upon the equipment used. At optimal points 5-8 of models 2 and 3 the measured sampling rates were identical, although subsequent investigation with the hardware monitor showed that model 3 was in fact performing longer program cycles

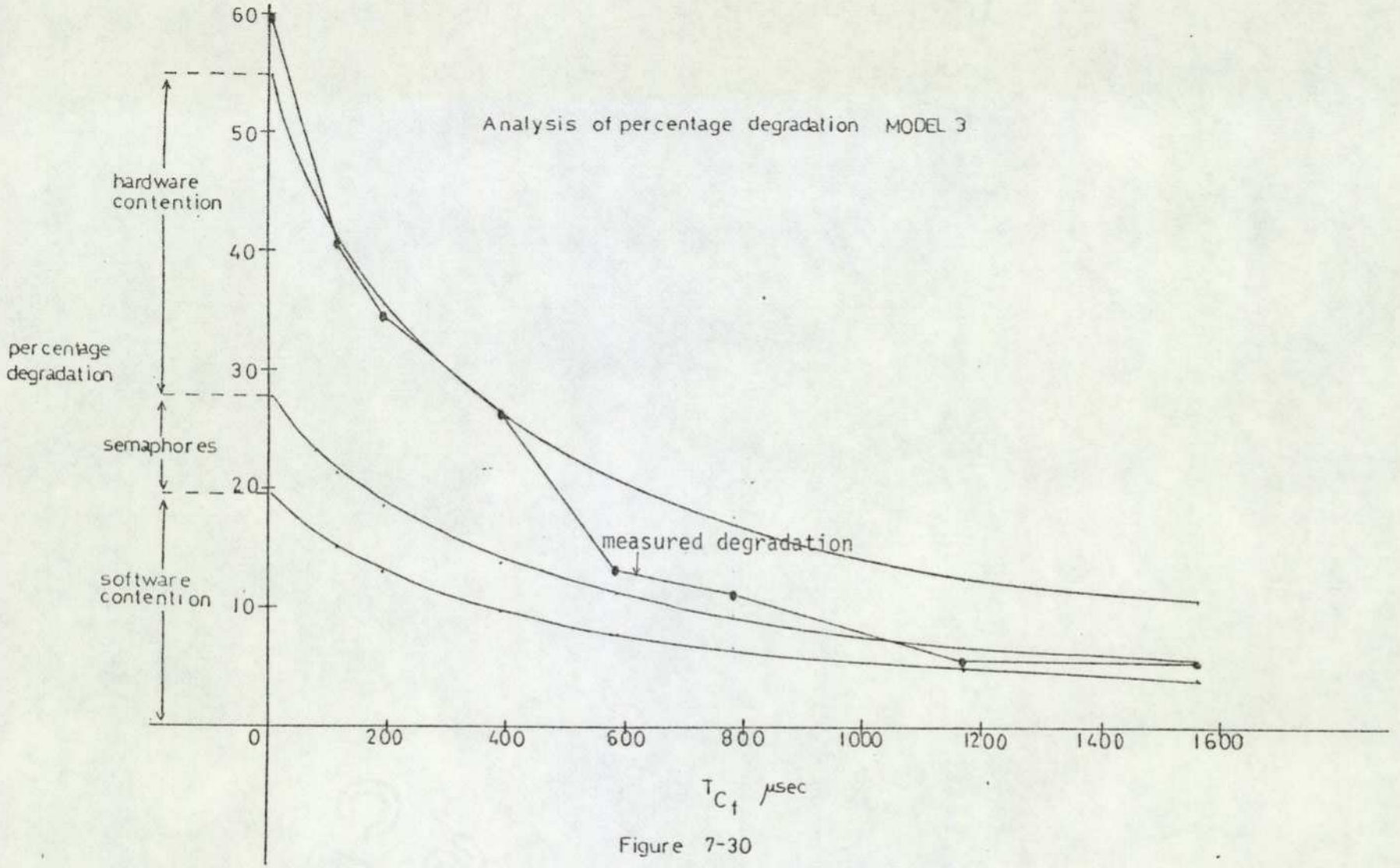

Analysis of percentage degradation MODEL 3
60
50
40
30
20
10
0
200
400
600
800
1000
1200
1400
1600
hardware contention
percentage degradation
semaphores
software contention
measured degradation
T_{C_f} μsec

Figure 7-30

(due to additional contention) than model 2. This suggests that the measured degradation for model 3 should be higher (and therefore closer to the calculated degradation) than that of model 2. It also provides an estimate of the accuracy of the method used to measure optimum sampling rates. If the number of instructions executed per cycle (of both processes) is compared (Table 7.10) for models 2 and 3 it is apparent (with reference to Table 7.2) that differences of less than approximately 10% are not detected.

optimal point	average number of instructions per cycle		Percentage difference
	Model 2	Model 3	
1	57.5	71	18.73
2	83.75	101	17.1
3	103.75	121	14.3
4	153.75	171	10.1
5	203.75	221	7.8
6	253.75	274	7.4
7	357.5	372	3.9
8	457.5	475	3.7

Table 7.10

Finally it should be noted that some of the optimal points for model 3 were difficult to fix because of the presence of beating effects (7.2) and that it may be that the results for model 2 are, by chance, better than the methods and equipment used could reasonably be expected to produce.

7.7 Summary of results

The results reported in this chapter were the outcome of investigation by the non-invasive methods described in Chapter 6 of the behaviour of the software models of Chapter 5 when implemented on a dual F100-L processor (Chapter 4).

The typical behaviour of the models under varying degrees of close coupling was established and optimal points, at which a maximum amount of useful work could be obtained, were isolated.

The performances of the dual processor systems were compared with that of the single processor system in terms of degradation of the additional processor power. A model of the composition of the degradation (consisting of hardware contention in bus usage, software contention in access of critical sections protecting shared memory, and software overhead for implementation of synchronisation primitives) was proposed and shown to agree, within the bounds of reasonable experimental error, with the observed behaviour of the system.

Chapter 8

Discussion

Real time signal processing power is of immense practical importance in many fields such as image processing, robotics and artificial intelligence, which will play an influential role in determining the direction of development of our society.

Raw processing power is now available at a cost which, until recently, was inconceivable and this has resulted in an explosive increase in the range and scale of applications which are economically and technically feasible. If the maximum benefit is to be derived from technological advance it is of vital importance that we learn to manage the complexity of the potential which it provides.

Multiprocessors lend themselves naturally to systems in which the processing requirements are interdependent although physically or conceptually distributed: for example in providing the 'eyes' and 'hands' of an industrial robot, realising a scheduler and a memory management unit in an operating system or supporting the filter and i/o processes of the models described in Chapter 5. However, the potential complexity of even comparatively simple multiprocessing systems (particularly those which are closely coupled) is such that unless a rigorous approach is adopted towards their design they rapidly become incomprehensible and therefore unreliable.

The necessity for structuring has been recognised by authors primarily concerned with design of both software (Dahl, Dijkstra and Hoare, 1972) and hardware (Lawson and Magnhagen, 1975) and the twin attributes

of simplicity and reliability have been motivating criteria in the development of programming language design (Brinch Hansen, 1975; Wirth, 1977; ACM, 1979). In many microprocessor applications hardware and software are very closely interrelated and it is inappropriate to consider their designs in isolation. In such a situation where there is no universally applicable programming (in the sense of both hardware and software) language it is convenient and productive to consider a system as a set of interdependent abstract processes but treat their subsequent realisation in either hardware or software at a more practical architectural level. The concept of 'levels of abstraction' is, in the author's opinion, the key to management of complexity.

The need for rigour that is apparent in the design of any system of concurrent processes is considerably more marked when those processes are coupled to the real world, for example: life support systems in hospitals, missile and space guidance systems and industrial control of dangerous machinery. It was with such critical, real time applications in mind that this project was carried out, and the emphasis on association with the real world is reflected in the concrete nature of the results presented. In the literature concerning systems of multiple processes (referenced in Chapter 12) there is little consideration of specific software models,

less of the question of process coupling and virtually none of the analysis of performance in terms of the passage of time in microseconds in the real world. In the design of systems which are expected to operate at the limit of their performance, behaviour must be considered in absolute terms and hence the results presented in this thesis are, besides being expressed comparatively, given in absolute units of time. To the author's knowledge, there is little published work which refers directly to system capabilities in terms of microseconds.

A summary of graphical results is shown in Fig. 8-1. Graphs 1 and 2 show the sampling rates of the dual processor system varying with the process coupling as counts Cf and Ci in the filter and i/o processes respectively are altered. Graph 3 shows the interrelation of 1 and 2 and the manner in which the sampling rate tends to $^{1}/_{T}$ as the coupling becomes looser. Graph 4 shows comparatively the optimal sampling rates achievable for models 1, 2 and 3 for arbitrarily fixed values of Cf. In reality, we would not expect to achieve such rates consistently for an actual implementation. In general the operating point of a process would not be fixed at an optimal point but would vary along the horizontal axis of the curves shown in graphs 1 and 2 as the degree of process coupling was affected by such factors as erratic input rates, variable DMA cycles to access

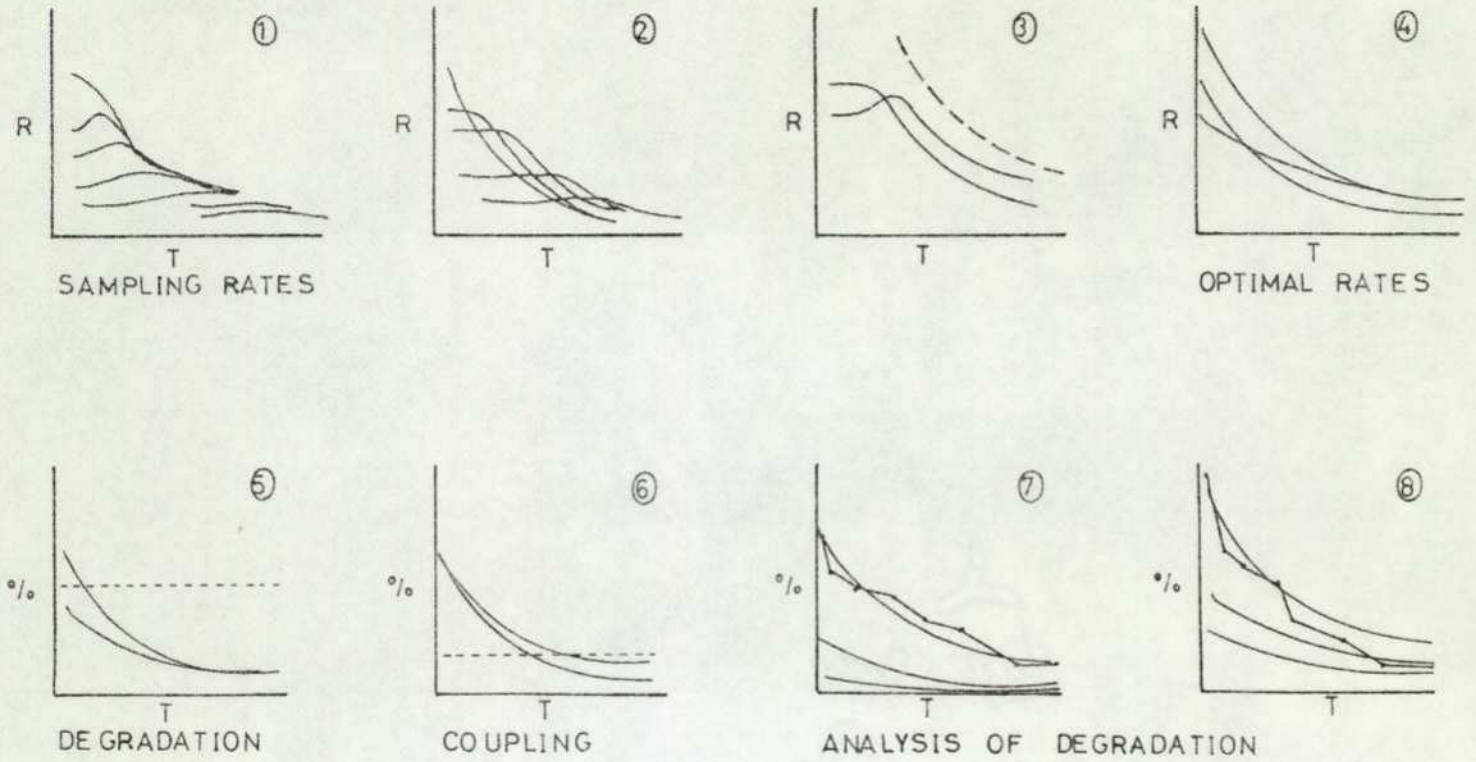


Figure 8-1. GRAPHICAL RESULTS

shared memory and external interrupts. What the graphs do show is the trend and limit of process behaviour which in turn leads to an estimation of the expected degradation.

Measurement of degradation at optimal points (Graph 5) shows its dependence on coupling and emphasizes the danger of closely coupled multiprocessor systems being degraded to an extent such that their performance is worse than that of a single processor system.

The nature of the processes and the coupling mechanism determine the extent of process interaction within a system, the absolute degradation being dependent upon the actual implementation of the coupling. Analysis of software (programs) and the hardware (processors) on which it runs to realise processes gives a static measure of the potential coupling of a system. The actual coupling can only be measured dynamically while the processes are in existence (i.e. the programs are being executed by the processors). Such a measurement is only feasible with a hardware monitor, as described in Chapter 6, which negligibly disturbs the balance of the system.

System degradation will be an increasingly important aspect of future designs. If it is to be minimised, then its constituents must be isolated. Graphs 7 and 8 summarise the analyses of the systems described here, in which degradation was found to be

caused by contention in the use of shared resources (memory and bus) and the overheads of implementing hardware and software synchronisation. The same factors would be present in any multiprocessor system based on similar design principles (shared memory on a common bus), but the extent of their influence depends upon the inherent suitability, or lack of it, of the hardware and software provision for handling synchronisation and the methods used for controlling and resolving contention.

The asynchronous F100-L processor achieves hardware synchronisation through handshake protocols and software synchronisation by a test-and-set type machine instruction (Chapter 4). The standard interface sets, which are an excellent concept in themselves, actually proved to be a weakness in the system because the delays associated with them accounted for the greatest part of the degradation displayed by models 2 and 3 (that caused by the resolution of bus contention). As the F100-L processor is asynchronous it should be able to optimise bus usage, and thus minimise contention in the use of what is the most critical resource in the system. Design of a more efficient replacement for the interface set would therefore be an obvious potential improvement to system performance (and to the author's knowledge this has been done in more than one instance).

Software design controls the contention, and hence the degradation, exhibited by a system. Design factors which exercise this control are the choice of number of semaphores and consequent extent of critical section code dependent upon a particular semaphore (illustrated in graphs 7 and 8). In a more complex system the size of critical sections, design of data structures (e.g. choice of buffer size), and order of execution of process tasks could all be expected to influence system behaviour.

For high performance, real time applications, such as digital filtering, towards which this system was oriented, the F100-L processor proved to be suitable in terms of the precision given by its sixteen-bit wordlength and its hardware and software support for synchronisation.

The shared memory dual processor architecture reflects currently proposed designs (Brinch Hansen, 1978 d) of multiprocessors for concurrent programming, but it's implementation with standard interface sets, although convenient, could be improved in terms of efficiency.

Other proposed systems at the time this decision was made (Caprani et al, 1977; Hoener and Roehder, 1977; Swan et al, 1977; Dagless, 1977) required additional, frequently extensive, hardware; and were based, with the exception of Cm* (Swan et al, 1977), on eight bit synchronous processors without machine instructions

specifically intended for multiprocessing. Degradation due to contention in accessing shared memory in these systems was either ignored, disregarded because of the looseness of coupling of the processors or deemed to be reduced to insignificance by the use of shared memory with a shorter cycle time than that of the non-shared memory in the system. Contention was, in fact, likely to be particularly severe in synchronous systems as gaining control of the bus is the worst potential bottleneck in a shared memory multiprocessor.

Recent multiprocessor designs, for example the DEMOS system (Dowson, Collins and McBride, 1979) which was based originally on Ferranti's Argus 700F sixteen bit minicomputer and later on Intel's 8086 sixteen bit microprocessor, have test-and-set machine instructions and a software bus locking facility. DEMOS however is a packet switched unidirectional ring network and exhibits the architecture of a more loosely coupled distributed processor than the closely coupled multiprocessor with which we are concerned.

The dual F100-L processor described here is seen as a particular, minimal instance of the multiprocessor case. Solution of the two-processor problem is a necessary but not sufficient condition for solution of the multiprocessor problem. It is also a highly convenient vehicle for the application under immediate consideration, namely, signal processing. For the general case a ring architecture is an obviously

attractive possibility, the key questions being the minimum contention levels and hence the real time response levels achievable.

In performance measurements, the processor (CP1600) controlled hardware monitor proved to be a satisfactory method of analysing bus activity, having the virtues of both avoiding the distortion of a system that is caused by software monitoring and being able to cope with a volume of data greatly in excess of the capacity of a conventional logic analyser, in addition to the ability to present the data in a pre-processed form.

If meaningful statements are to be made about the coupling of processors or contention between processes in a system, then they must be made in the context of a particular software model: it is non-constructive to say that "when n processors were connected the system exhibited m% degradation". On the other hand, specific measurements taken of a highly specialised system are, while undoubtedly meaningful, of little general use or interest.

As this work was undertaken in a research group involved in signal processing, a digital filter was an obvious potential software model which had the advantage of partitioning neatly for a dual processor into a filter process and an i/o process. To retain generality and avoid effects caused by the characteristics of a particular filter, the filter was kept to a

minimally simple form (5.5.3.3) and the model then reduced to two instances of the producer/consumer algorithm on a bounded buffer. Thus a very well understood and widely applicable algorithm was being dealt with. The algorithm is particularly interesting in the light of recent work on communicating sequential processes (Hoare 1978) in which input and output are regarded as programming primitives and the use of concurrent communicating sequential processes is regarded as a basic program structuring method so that concurrent programs consist of sequential processes which communicate with one another by means of i/o commands. Similar ideas are expressed by Brinch Hansen (1978 a and 1978 b) who concerns himself more directly with real time applications. Both of these authors, however, are primarily concerned with distributed systems having distributed (non-shared) memory and implementing interprocessor communication by means of guarded i/o commands.

The model was further generalised by having variable coupling, with the majority of cases analysed conforming to Brinch Hansen's definition (Brinch Hansen, 1978 d) of closely coupled processors as being those in which ten per cent or more of instructions access shared memory. Clearly this must be a dynamic measurement: otherwise a factor obviously influencing the coupling, such as the reduction in number of semaphores from model 2 to model 3, would go unnoticed.

Thus in addition to a suitable software model, a non-invasive monitor (6.3.1) is also shown to be a necessity in detecting and analysing contention.

Production of reliable and efficient concurrent real time programs is accepted to be one of the most difficult of programming tasks (Wirth, 1977 a). Sequential programs are characterised by their reproducible behaviour, so if they are written in a well-structured high level language, such as Pascal, and systematically tested the programmer can have confidence in their reliability. Concurrent programs however exhibit context-dependent behaviour and testing will not necessarily detect or isolate time-dependent errors. The reliability of such programs depends upon the availability to the programmer of high level, time independent programming language constructs which can be checked by a compiler to eliminate time dependent errors. Establishing the reliability of real time concurrent programs presents a further order of difficulty, in that the real time sections of a program may rely on specific timing constraints to establish their correctness.

On account of these difficulties, many real time programs or sections of programs have been written at assembly level where it has been felt that the programmer has the greatest degree of control and can achieve the greatest efficiency. In the author's view it is almost impossible to have confidence in the

reliability of a concurrent real time program of any complexity which has been written directly in assembly language. Unreliable programs of this type are particularly disastrous in many of the specialised applications, for example medical and aerospace systems, in which they commonly occur.

Some implementations of real time programming languages e.g. CORAL 66 (H.M.S.O., 1970) adopted a hybrid approach to the problem by allowing low level inserts to be introduced into an otherwise high level language program. This technique however is directly contrary to the current practice in programming language design of attempting to achieve reliability by strong typing, rigorous control structures and compiler-enforced restricted and standardised access to data structures. The question is one of level: all programs are translated to machine code before execution but many details and facilities, which are visible to the 'implementor', are transparent to the 'user'. In real time programming a task which is, strictly speaking, one of implementation (typically relating to i/o which is usually grammatically undefined in high level languages) may commonly fall to the lot of the user and, unless he recognises the difference, any degree of reliability achieved by the use of high level language constructs and compiler checking of their use may be nullified.

Modula (Wirth, 1977 b) was a direct approach to this problem, the 'modules' which give the language its name being language constructions encapsulating machine dependent items and restricting their validity to a specific, small area of program. A machine's device registers, and operators on them, would be available within a peripheral device module but invisible outside. This was designed to provide the same efficiency as assembly language programming could, while at the same time maintaining the rigour of a high level language.

The price that is paid for high level language constructs is a certain amount of redundancy to enable the compiler to make extensive checks on the source code. The efficiency of the generated machine code will depend upon the calibre of the compiler but any compiler performing comprehensive compile time checks should be able to minimise the overhead of run time checking. Other advantages which accrue from the use of high level languages are greater simplicity at the source program level, greater adaptability, a degree of portability, and reduced software costs.

The most recent high level languages directed towards real time concurrent programming, concurrent Pascal, Modula and Ada, are all based on Pascal and their authors are at pains to stress (Brinch Hansen, 1977; Wirth, 1977) their priorities of simplicity, reliability and efficiency. Concurrent Pascal has been

used to write operating systems (Brinch Hansen, 1977) and to implement networks of distributed processes (Brinch Hansen, 1978 a and 1978 b) whereas Modula (Wirth, 1977) has been used primarily as an implementation language for multiprogramming on a single processor.

Ada (ACM, 1979; Wegner, 1980) aims to cover a much wider application area by providing an extensive range of language features. It is directed at both shared memory and distributed process implementations although initial implementations will be for multiprogrammed single processors. Of particular interest is the tasking mechanism for concurrent programming. This is termed a rendezvous and occurs when a calling task, T1, which is waiting to execute an 'entry' call on a task T2 protecting a shared resource, is synchronised with the called (T2) task waiting to execute an 'accept' statement for that entry call. Thus a rendezvous is equivalent to a critical section (3.1) and the calling process is suspended during execution of the rendezvous. 'Accept' statements may appear in guards (3.3), so, instead of the passive monitor of Concurrent Pascal, in Ada an active task is used to achieve synchronisation.

It is an unfortunate fact that, at the time of writing, available software for microprocessors is usually extremely limited and has little point of contact with current language developments. The programmer must choose between either implementing a

suitable existing language (Dowson, Collins and McBride, 1979), developing a language himself (Schutz, 1979), using an available language irrespective of its suitability (Aspinall and Dagless, 1979) or relying on assembly language. The first two of these options imply a degree of manpower not available to the author and, in view of the initial lack of anything but assembly language and the relatively small amount of programming involved, the programs described in this thesis were coded in assembly language. The programs were actually written, however, in concurrent Pascal-like notation and were then hand translated and checked. Although this is obviously far from satisfactory, the method proved surprisingly worthwhile in terms of the ease of expression, clarity of logical ideas and confidence of time-dependent correctness given at the programming stage, with the coding stage reduced to a largely automatic exercise.

Meaningful measurement of degradation in multimicroprocessor systems is an area in which there is little published information, although it is of great importance for closely coupled systems, where, as was seen in Chapter 7, it may account for losses in excess of one hundred per cent of additional processing power. Some degradation will always be present in closely coupled systems with any degree of integrity: if it is to be minimised, its causes and factors which influence them must be determined.

In this thesis, the causes of degradation in the F100-L system have been shown to be implementation of synchronisation primitives, software contention in accessing shared memory and hardware delays (i.e. bus contention) in accessing shared memory (7.6). For any particular software model the extent of the software contention, and hence of the total hardware delay, will not be affected, in absolute terms, by the closeness of coupling of a system of synchronised, non-idling process (7.5) unless (as in reducing the number of semaphores from two in model 2 to one in model 3) the coupling is altered in a manner which also affects the synchronisation pattern. If a process is idling, and in doing so is accessing shared memory, then this is likely to increase contention, although the synchronisation pattern will not alter. A problem presented for solution by concurrent programming will normally have an inherent synchronisation requirement. The exact pattern of synchronisation will be determined by the number of synchronisation controls (e.g. semaphores) employed which in turn affects the form of the mutual exclusion in terms of size and distribution of critical sections.

Given a fixed synchronisation pattern, software contention may be minimised by preventing processes from destructively idling and forcing them instead to idle without requesting shared resources. At an optimal point there is no destructive idling. As the

processors under consideration are dedicated, and so not multiprogrammed, no useful work is lost by the introduction of a fixed length idling loop. If however the processes were irregular, in the sense of input occurring in DMA bursts or the filtering routine being of variable length, then the problem of idling is not so easily solved as the introduction of a fixed length loop is no longer adequate. Processors having the ability to suspend themselves (which the F100-L does not) could rely on a system of signals rather than semaphores but, for reliability, these would need to be used in guarded constructs to detect potential errors such as the issuing of non-awaited signals.

Delays due to bus contention are a function of the system hardware. It has been clearly demonstrated here (7.6.2) that resolution of such contention is a critical factor in the design of an efficient system.

The actual overhead of implementing synchronisation primitives is not high (the time for a P plus V pair is 19.5 μsecs) and is, in any case, indispensible for problems of non-trivial complexity. This overhead is affected only by the number of primitives employed and the size, and hence number, of critical sections. If a good compiler is available, abstract data structures and high level language constructs to manipulate them should not introduce any overhead which is not greatly outweighed by the advantages of logical simplicity, reliability and adaptability. The software model which has been discussed has extremely wide applicability,

particularly when concurrent programs are viewed (Hoare, 1978) as sequential processes with the interprocess communication mechanism being i/o commands on a 'device', an instance of which could as well be a data structure (e.g. a buffer) as a conventional peripheral device.

Finally, the work presented here, as well as constituting a detailed analysis of the behaviour of a particular system, may serve as a model for other closely coupled systems both to predict behaviour and as a method for analysing and optimising performance.

Chapter 9

Suggestions for further work

The results reported in Chapter 7 show clearly that F100-L system performance could be improved by replacing the standard interface sets, particularly those forming the bus extension unit (4.1.3), by special purpose logic.

An obvious area for future development is the extension of the current dual processor system to a multiprocessor system. A unidirectional ring appears to be an attractive architecture for such a system (Comley, 1980 b). The outstanding problem is that of resolving bus allocation. If the system was packet switched then bus interface hardware must be designed to handle the packets. If the system was circuit switched then the possibility of deadlock must be catered for (3.1.2).

The hardware monitor described in Chapter 6 could also be extended to enable it to trap a wider range of instruction sequences and so increase it s flexibility. It might also prove useful to allow the controlling CP1600 processor to perform additional processing on collected data in cases when it is not monitoring very tight program loops.

Implementation of a software model handling data transfers by DMA, such as that described in (5.5.3.4), requires some modification of the F100-L processor's DMA mechanism. DMA transfers are initiated by writing an address (at which data is to be written or read) into a control word. The address is automatically

incremented and transfers proceed under the control of the device. This would be modified by the addition of a counter to specify the number of transfers which were to occur. Transfers would be initiated by a program supplying both an address and a count (which would be automatically decremented). When the count was exhausted an interrupt would occur enabling the program to initiate the next transfer. In this way a process could control DMA transfers with the interrupt acting as a signal of a given number of completed transfers.

A system in which input and output transfers are achieved by DMA (and input is generated in random bursts by a specially designed data burst generator) would allow the effect of buffering on system performance to be investigated. Choice of buffer sizes, DMA burst sizes and strategies for emptying buffers which have been filled by rapid input bursts are all factors which should be considered. Statistical analysis of buffer operation may also prove to be useful. However, this would require a statistical description of the data source as, for example, a stochastic process of prescribed properties.

Apart from variations on the models described in this project, there are many other general models which could be analysed. In particular, more complex models with a greater number of shared data structures would demonstrate the effects of critical section length versus the overhead of handling larger numbers of synchronisation primitives.

Other important questions are those of suitable techniques (detection, prevention, avoidance) of dealing with software deadlock, and of the allocation of concurrent programs to processors. It seems probable that for multimicroprocessor systems such allocation will be static, as the overhead of dynamic allocation is likely to prove prohibitive. Development of parallel algorithms is itself a wide area for future research.

Although much effort is being devoted to designing real time concurrent programming languages (such as Modula and Ada) very few, if any, such languages are yet available for microprocessors. Portability techniques seem to offer the best hope for relatively quick and cheap provision of language compilers, or indeed any other software development aid. There is much scope for further work here as many of the current techniques in use are inappropriate for high performance applications in, for example, that they rely on unsuitable implementation languages (and are therefore inefficient) or they are interpretive and therefore too slow for real time work.

The potential benefits of learning to manage the complexity inherent in advanced multiprocessor systems are very great indeed. Consequently, areas of research in this field are expanding rapidly and are likely to prove to be particularly challenging and rewarding.

Chapter 10

Conclusion

This thesis has described the hardware and development of software for a dual microprocessor system. The system was designed to solve, for a particular case, the technical problems of multi-processing and to act as a vehicle to establish design principles for such systems. The design was directed towards portable microprocessor based systems intended for real time applications.

A generalised software model of wide applicability was developed, and the question of suitable language constructs for real time concurrent programming was considered. The software model reflected the type of high throughput signal processing problem, the solution of which was a motivating factor for this project. The system was programmed according to a high level language philosophy adopted to provide the rigour necessary in critical high performance applications.

The model was used to determine general behaviour patterns for a class of problems based on producer/consumer algorithms, and to isolate sets of conditions under which optimal system performance could be achieved.

A non-invasive computer-controlled hardware monitor has been designed and used to obtain precise extensive data, tracing program behaviour when the system of concurrent processes was operating at its maximum attainable speed. The hardware monitor proved

to be a very satisfactory technique for performing measurements at the speed, and of the scope, required for analysis of closely coupled, real time concurrent processes.

The precision of the measurements made it possible to analyse program behaviour in real terms (that is, in absolute time units - microseconds). As a result, degradation exhibited within the system could be attributed to it's various causes. These causes were found to be contention in gaining control of shared resources, namely the system bus (hardware level contention) and shared memory (software level contention), and the software overhead resulting from the use of synchronising primitives. The extent of degradation attributable to the various causes was determined for systems of varying closeness of coupling (caused both directly and as a result of a differing synchronisation pattern due to the number of semaphores employed).

This demonstrated the causes of degradation likely to be present in multiprocessor systems and allowed precise comment to be made on the behaviour of the F100-L system. The F100-L processor proved to be suitable for multiprocessing but use of the interface sets, although ideal in theory, proved in reality to incur an extremely high overhead. The use of high level language constructs in designing programs greatly simplified the production of reliable programs,

and the overhead incurred was considered to be minimal in view of the rigour and security provided by this approach. The monitoring system was a very satisfactory solution to a difficult measurement problem and its flexibility and adaptability will allow it to be applied in many other situations.

In conclusion, multiprocessing does offer a feasible solution for the problems occurring in complex measurement applications. Suitable implementation and assessment techniques have been developed here and the major critical design criteria have been isolated.

Chapter 11

Appendices

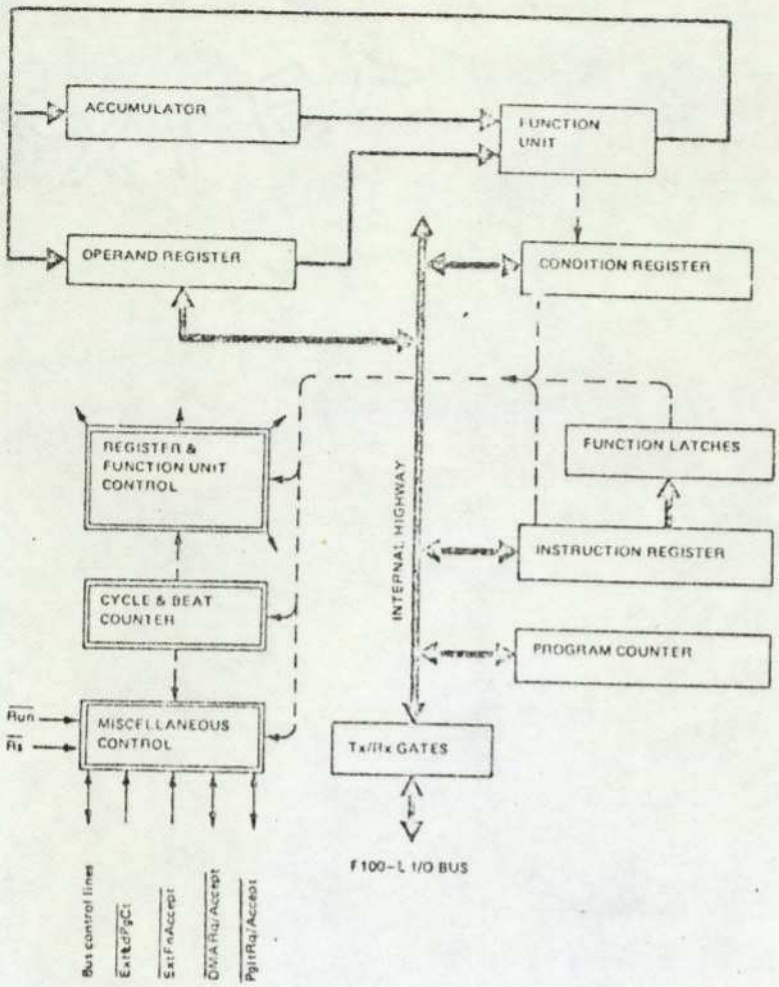
Appendix 1

F100-L block diagram

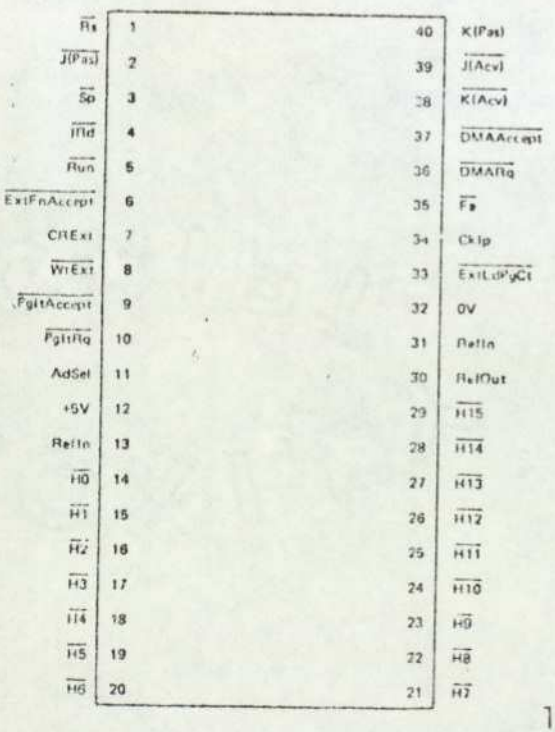
F100-L pin allocation

F100-L instruction set

F100-L system timing



F100-L Block Diagram



F100-L Pin Allocation

F100 ASSEMBLER INSTRUCTIONS

This table contains a list of all Assembly Language instructions. The function of each form of instruction is given under six headings as described below:

MNEMONIC

For each opcode mnemonic all permitted operands are shown using the following notation:

B - a bit number or the number of places involved in a shift instruction.
D - denotes immediate data held as a 16-bit quantity in the second word of a multi-word instruction.
H - a 10-bit halt number.
K - denotes any of the condition register staticiser mnemonics.
N - is an address in lower memory (1 to 2047).
W - is an address anywhere in memory (0 to 32767).
W1 - is an address anywhere in memory (0 to 32767).
P - is a pointer address (1 to 255).

WORDS

The number of 16-bit F100 words occupied by any given form of an instruction is defined.

MACHINE CODE

This column defines each bit of the machine code generated as the first word of the given instruction. The individual bits of an address or bit reference are designated by the letter used in the mnemonic column. X denotes 'don't care' bits, which are in fact set to zero by the assembler.

STATICISERS

The Condition Register staticisers affected by any instruction are defined using the following symbols:

0 - the staticiser is cleared
1 - the staticiser is set
- the staticiser may be set or cleared by the instruction
* - resultant state is meaningless
- set if the most significant two bits (bits 14 and 15) of the original data in the relevant register or store location were unequal, otherwise cleared
? - only the staticiser specified in the instruction may be changed.

TIMING

The execution times of each instruction are given in terms of the logic cycle time (L = twice the period of the input clock), the memory read access time (Ra), the memory read cycle time (Rc), the memory read/modify/write cycle time (M) and the memory write cycle time (Wc). Normal algebraic notation is used.

DESCRIPTION

The action of each instruction is briefly described using the following notation:

A - Accumulator
PC - Program Counter
CR - Condition Register
OR - Operand Register

Enclosing parentheses are used to denote 'contents of'. Thus if P is a pointer address, (P) denotes the contents of P, i.e. the address pointed to by P, and ((P)) denotes the contents of that address.

MNEMONIC	WORDS	MACHINE CODE F	I	X	N R	N P S	N P J	N P B	STATICISERS F 6	M 5	C 4	S 3	V 2	Z 1	I 0	TIMING	DESCRIPTION
ADD N	1	1 0 0 1	0	T	N N N N	N N N	N N N	N N N			✓	✓	✓	✓		2Ra+18L	Add (N) into A. C is used if M is set.
ADD ,D	2	1 0 0 1	0	0	0 0 0 0	0 0 0	0 0 0	0 0 0			✓	✓	✓	✓		2Ra+18L	Add D into A. C is used if M is set.
ADD /P	1	1 0 0 1	1	X	0 0	P P P	P P P	P P P			✓	✓	✓	✓		2Ra+M+19L	Add ((P)) into A. C is used if M is set.
ADD /P+	1	1 0 0 1	1	X	0 1	P P P	P P P	P P P			✓	✓	✓	✓		2Ra+M+34L	Increment (P). Add ((P)) into A. C is used if M is set.
ADD /P-	1	1 0 0 1	1	X	1 1	P P P	P P P	P P P			✓	✓	✓	✓		2Ra+M+34L	Add ((P)) into A. C is used if M is set. Decrement (P).
ADD .W	2	1 0 0 1	1	X	X X	0 0 0	0 0 0	0 0 0			✓	✓	✓	✓		2Ra+1Rc+18L	Add (W) into A. C is used if M is set.
ADS N	1	0 1 0 1	0	N	N N N	N N N	N N N	N N N			✓	✓	✓	✓		Ra+M+18L	Add A into (N). C is used if M is set.
ADS ,D	2	0 1 0 1	0	0	0 0 0 0	0 0 0	0 0 0	0 0 0			✓	✓	✓	✓		Ra+M+18L	Add A into next word containing D, overwriting D. C is used if M is set.
ADS /P	1	0 1 0 1	1	X	0 0	P P P	P P P	P P P			✓	✓	✓	✓		Ra+2M+19L	Add A into ((P)). C is used if M is set.
ADS /P+	1	0 1 0 1	1	X	0 1	P P P	P P P	P P P			✓	✓	✓	✓		Ra+2M+34L	Increment (P). Add A into ((P)). C is used if M is set.
ADS /P-	1	0 1 0 1	1	X	1 1	P P P	P P P	P P P			✓	✓	✓	✓		Ra+2M+34L	Add A into ((P)). C is used if M is set. Decrement (P).
ADS ,W	2	0 1 0 1	1	X	X X	0 0 0	0 0 0	0 0 0			✓	✓	✓	✓		Ra+Rc+M+18L	Add A into (W). C is used if M is set.
AND N	1	1 1 0 0	0	N	N N N	N N N	N N N	N N N			1	✓	*	✓		2Ra+18L	AND (N) into A.
AND ,D	2	1 1 0 0	0	0	0 0 0 0	0 0 0	0 0 0	0 0 0			1	✓	*	✓		2Ra+18L	AND D into A.
AND /P	1	1 1 0 0	1	X	0 0	P P P	P P P	P P P			1	✓	*	✓		2Ra+M+19L	AND ((P)) into A.
AND /P+	1	1 1 0 0	1	X	0 1	P P P	P P P	P P P			1	✓	*	✓		2Ra+M+34L	Increment (P). AND ((P)) into A.
AND /P-	1	1 1 0 0	1	X	1 1	P P P	P P P	P P P			1	✓	*	✓		2Ra+M+34L	AND ((P)) into A. Decrement (P).
AND .W	2	1 1 0 0	1	X	X X	0 0 0	0 0 0	0 0 0			1	✓	*	✓		2Ra+Rc+18L	AND (W) into A.
CAL N	1	0 0 1 0	0	N	N N N	N N N	N N N	N N N		0						Ra+M+2W+19L	Stack PC and CR. Clear M. Load PC with N.
CAL ,D	1	0 0 1 0	0	0	0 0 0 0	0 0 0	0 0 0	0 0 0		0						Ra+M+2W+19L	Stack PC and CR. Clear M. PC not altered so D obeyed as an instruction
CAL /P	1	0 0 1 0	1	X	X X	P P P	P P P	P P P		0						Ra+Rc+M+2W+18L	Stack PC and CR. Clear M. Load PC with (P).
CAL .W	2	0 0 1 0	1	X	X X	0 0 0	0 0 0	0 0 0		0						Ra+Rc+M+2W+18L	Stack PC and CR. Clear M. Load PC with W.
CLR B	1	0 0 0 0	0	0	0 0	1 1	1 1	B B B								Ra+19L	Clear bit B of A.
CLR K	1	0 0 0 0	0	0	0 1	1 1	1 1	B B B	?	?	?	?	?	?	?	Ra+20L	Clear appropriate staticiser.
CLR B W	2	0 0 0 0	0	0	1 1	1 1	1 1	B B B								Ra+Rc+M+18L	Clear bit B of (W).
CMP N	1	1 0 1 1	0	N	N N N	N N N	N N N	N N N			✓	✓	✓	✓		2Ra+18L	Set staticisers on result of (N) - A. C is used if M is set.
CMP ,D	2	1 0 1 1	0	0	0 0 0 0	0 0 0	0 0 0	0 0 0			✓	✓	✓	✓		2Ra+18L	Set staticisers on result of D - A. C is used if M is set.
CMP /P	1	1 0 1 1	1	X	0 0	P P P	P P P	P P P			✓	✓	✓	✓		2Ra+M+19L	Set staticisers on result of ((P)) - A. C is used if M is set.
CMP /P+	1	1 0 1 1	1	X	0 1	P P P	P P P	P P P			✓	✓	✓	✓		2Ra+M+34L	Increment (P). Set staticisers on result of ((P)) - A. C is used if M is set.
CMP /P-	1	1 0 1 1	1	X	1 1	P P P	P P P	P P P			✓	✓	✓	✓		2Ra+M+34L	Set staticisers on result of ((P)) - A. C is used if M is set. Decrement (P).
CMP .W	2	1 0 1 1	1	X	X X	0 0 0	0 0 0	0 0 0			✓	✓	✓	✓		2Ra+Rc+18L	Set staticisers on result of (W) - A. C is used if M is set.
HLT H	1	0 0 0 0	0	1	H H H	H H H	H H H	H H H									Stop processor.
ICZ N W	2	0 1 1 1	0	N	N N N	N N N	N N N	N N N								Ra+Rc+M+18L	Increment (N). Jump to W if (N) ≠ 0.
ICZ ,D W	3	0 1 1 1	0	0	0 0 0 0	0 0 0	0 0 0	0 0 0								Ra+Rc+M+18L	Increment D. Jump to W if D ≠ 0.
ICZ /P W	2	0 1 1 1	1	X	0 0	P P P	P P P	P P P								Ra+Rc+2M+19L	Increment ((P)). Jump to W if ((P)) ≠ 0.
ICZ /P+ W	2	0 1 1 1	1	X	0 1	P P P	P P P	P P P								Ra+Rc+2M+34L	Increment (P). Increment ((P)). Jump to W if ((P)) ≠ 0
ICZ /P- W	2	0 1 1 1	1	X	1 1	P P P	P P P	P P P								Ra+Rc+2M+34L	Increment ((P)). Decrement (P). Jump to W if ((P)) ≠ 0.
ICZ .W W1	3	0 1 1 1	1	X	X X	0 0 0	0 0 0	0 0 0								Ra+2Rc+M+18L	Increment (W). Jump to W1 if (W) ≠ 0.
JBC B W	2	0 0 0 0	0	0	0 0	1 0	0 0	B B B								Ra+Rc+19L	Jump to W if bit B of A clear.
JBC K W	2	0 0 0 0	0	0	0 1	1 0	0 0	B B B								Ra+Rc+20L	Jump to W if appropriate staticiser clear.
JBC B W W1	3	0 0 0 0	0	0	1 1	1 0	0 0	B B B								Ra+2Rc+M+18L	Jump to W1 if bit B of (W) clear.
JBS B W	2	0 0 0 0	0	0	0 0	1 0	0 1	B B B								Ra+Rc+19L	Jump to W if bit B of A set.

MNEMONIC	WORDS	MACHINE CODE F	I	N T	N R	N P S	N P J	N P B	STATICISERS F 6	M 5	C 4	S 3	V 2	Z 1	I 0	TIMING	DESCRIPTION
JBS K W	2	0 0 0 0	0	0	1	1 0	0 1	B B B B								Ra+Rc+20L	Jump to W if appropriate staticiser set.
JBS B W W1	3	0 0 0 0	0	0	1	1 0	0 1	B B B B								Ra+2Rc+M+18L	Jump to W1 if bit B of (W) set.
JCS B W	2	0 0 0 0	0	0	0	1 0	1 0	B B B B								Ra+Rc+19L	Jump to W if bit B of A clear and set bit B.
JCS K W	2	0 0 0 0	0	0	1	1 0	1 0	B B B B	?	?	?	?	?	?	?	Ra+Rc+20L	Jump to W if staticiser K clear and set K.
JCS B W W1	3	0 0 0 0	0	0	1	1 0	1 0	B B B B								Ra+2Rc+M+18L	Jump to W1 if bit B of (W) clear and set bit B.
JMP N	1	1 1 1 1	0	N	N	N N	N N	N N N N								Ra+3L	Load PC with N, i.e. jump to N.
JMP ,D	2	1 1 1 1	0	0	0	0 0	0 0	0 0 0 0								Ra+3L	Load PC with address of next word (containing D).
JMP /P	1	1 1 1 1	1	X	0	0 P	P P	P P P P								Ra+M+4L	Load PC with (P).
JMP /P+	1	1 1 1 1	1	X	0	1 P	P P	P P P P								Ra+M+19L	Increment (P). Load PC with (P).
JMP /P-	1	1 1 1 1	1	X	1	1 P	P P	P P P P								Ra+M+19L	Load PC with (P). Decrement (P).
JMP .W	2	1 1 1 1	1	X	X	X 0	0 0	0 0 0 0								Ra+Rc+2L	Load PC with W.
JSC B W	2	0 0 0 0	0	0	0	0 1	0 1	B B B B								Ra+Rc+19L	Jump to W if bit B of A set anc clear bit B.
JSC K W	2	0 0 0 0	0	0	0	1 1	0 1	B B B B	?	?	?	?	?	?	?	Ra+Rc+20L	Jump to W if staticiser K is set and clear K.
JSC B W W1	3	0 0 0 0	0	0	1	1 1	0 1	B B B B								Ra+2Rc+1M+18L	Jump to W1 if bit B of (W) is set and clear bit B.
LDA N	1	1 0 0 0	0	N	N	N N	N N	N N N N				✓	0	✓		2Ra+18L	Load A with (N).
LDA ,D	2	1 0 0 0	0	0	0	0 0	0 0	0 0 0 0				✓	0	✓		2Ra+18L	Load A with D.
LDA /P	1	1 0 0 0	1	X	0	0 P	P P	P P P P				✓	0	✓		2Ra+M+19L	Load A with ((P)).
LDA /P+	1	1 0 0 0	1	X	0	1 P	P P	P P P P				✓	0	✓		2Ra+M+34L	Increment (P). Load A with ((P)).
LDA /P-	1	1 0 0 0	1	X	1	1 P	P P	P P P P				✓	0	✓		2Ra+M+34L	Load A with ((P)). Decrement (P).
LDA .W	2	1 0 0 0	1	X	X	X 0	0 0	0 0 0 0				✓	0	✓		2Ra+1Rc+18L	Load A with (W).
NEQ N	1	1 1 0 1	0	N	N	N N	N N	N N N N			0	✓	*	✓		2Ra+18L	Non-equivalence (N) into A.
NEQ ,D	2	1 1 0 1	0	0	0	0 0	0 0	0 0 0 0			0	✓	*	✓		2Ra+18L	Non-equivalence D into A.
NEQ /P	1	1 1 0 1	1	X	0	0 P	P P	P P P P			0	✓	*	✓		2Ra+1M+19L	Non-equivalence ((P)) into A.
NEQ /P+	1	1 1 0 1	1	X	0	1 P	P P	P P P P			0	✓	*	✓		2Ra+1M+34L	Increment (P). Non-equivalence ((P)) into A.
NEQ /P-	1	1 1 0 1	1	X	1	1 P	P P	P P P P			0	✓	*	✓		2Ra+1M+34L	Non-equivalence ((P)) into A. Decrement (P).
NEQ .W	2	1 1 0 1	1	X	X	X 0	0 0	0 0 0 0			0	✓	*	✓		2Ra+Rc+18L	Non-equivalence (W) into A.
RTC	1	0 0 1 1	1	X	X	X X	X X	X X X X								Ra+Rc+1M+18L	Subroutine exit not affecting CR.
RTN	1	0 0 1 1	0	X	X	X X	X X	X X X X		✓	✓	✓	✓	✓	✓	Ra+2Rc+1M+18L	Subroutine exit restoring CR to entry state (except F staticiser).
SBS N	1	0 1 1 0	0	N	N	N N	N N	N N N N			✓	✓	✓	✓		1Ra+1M+18L	Subtract A from (N). C is used if M is set.
SBS ,D	2	0 1 1 0	0	0	0	0 0	0 0	0 0 0 0			✓	✓	✓	✓		1Ra+1M+18L	Subtract A from D, result overwriting D. C is used if M is set.
SBS /P	1	0 1 1 0	1	X	0	0 P	P P	P P P P			✓	✓	✓	✓		1Ra+2M+19L	Subtract A from ((P)). C is used if M is set.
SBS /P+	1	0 1 1 0	1	X	0	1 P	P P	P P P P			✓	✓	✓	✓		1Ra+2M+34L	Increment (P). Subtract A from ((P)). C is used if M is set.
SBS /P-	1	0 1 1 0	1	X	1	1 P	P P	P P P P			✓	✓	✓	✓		1Ra+2M+34L	Subtract A from ((P)). Decrement (P). C is used if M is set.
SBS .W	2	0 1 1 0	1	X	X	X 0	0 0	0 0 0 0			✓	✓	✓	✓		Ra+Rc+1M+18L	Subtract A from (W). C is used if M is set.
SET B	1	0 0 0 0	0	0	0	0 1	1 1	1 0 B B B B								1Ra+19L	Set bit B of A.
SET K	1	0 0 0 0	0	0	0	1 1	1 1	1 0 B B B B	?	?	?	?	?	?	?	1Ra+20L	Set staticiser K in CR.
SET B W	2	0 0 0 0	0	0	1	1 1	1 1	1 0 B B B B								Ra+Rc+1M+18L	Set bit B of (W).
SJM	1	0 0 0 1	X	X	X	X X	X X	X X X X								1Ra+20L	Add A into PC, i.e. switch jump.
SLA B	1	0 0 0 0	0	0	0	0 1	0 0	B B B B				✓	✓	*		1Ra+(B+4)L	M=0: Shift A left arithmetic B places (B ≤ 15)
		0 0 0 0	0	0	0	0 1	0 B	B B B B				✓	✓	*		Ra+(B+4)L	M=1: Shift A and OR left arithmetic B places (B ≤ 31).
SLA B ECR	1	0 0 0 0	0	0	0	1 1	0 0	B B B B	✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=0: Shift CR left arithmetic B places (B ≤ 15).
		0 0 0 0	0	0	0	1 1	0 B	B B B B	✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=1: Shift A and CR left arithmetic B places (B ≤ 31).
SLA B W	2	0 0 0 0	0	0	1	1 0	1 0	B B B B				✓	✓	*		Ra+Rc+M+(B+3)L	M=0: Shift (W) left arithmetic B places (B≤ 15).
		0 0 0 0	0	0	1	1 0	1 B	B B B B				✓	✓	*		Ra+Rc+1M+(B+3)L	M=1: Shift A and (W) left arithmetic B places (B ≤ 31).
SLE B	1	0 0 0 0	0	0	0	0 1	1 1	B B B B				✓	✓	*		Ra+(B+4)L	M=0: Shift A left end-around B places (B ≤ 15).
SLE B ECR	1	0 0 0 0	0	0	0	1 1	1 1	B B B B	✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=0: Shift CR left end-around B places (B ≤ 15).

MNEMONIC	WORDS	MACHINE CODE F	I	N T	N R	N P S	N P J	N P B	STATICISERS F 6	M 5	C 4	S 3	V 2	Z 1	I 0	TIMING	DESCRIPTION
SLE B W	2	0 0 0 0	0 0	1 1	0 1	1 1	B B B B				✓	✓	*			Ra+Rc+M+(B+3)L	M=0: Shift (W) left end-around B places (B ≤ 15).
SLL B	1	0 0 0 0	0 0	0 0	0 1	1 0	B B B B				✓	✓	*			Ra+(B+4)L	M=0: Shift A left logical B places (B ≤ 15).
		0 0 0 0	0 0	0 0	0 1	1 B	B B B B				✓	✓	*			Ra+(B+4)L	M=1: Shift A and OR left logical B places (B ≤ 31).
SLL B ECR	1	0 0 0 0	0 0	0 1	0 1	1 0	B B B B		✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=0: Shift CR left logical B places (B ≤ 15).
		0 0 0 0	0 0	0 1	0 1	1 B	B B B B		✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=1: Shift A and CR left logical B places (B ≤ 31).
SLL B W	2	0 0 0 0	0 0	1 1	0 1	1 0	B B B B				✓	✓	*			Ra+Rc+M+(B+3)L	M=0: Shift (W) left logical B places (B ≤ 15).
		0 0 0 0	0 0	1 1	0 1	1 B	B B B B				✓	✓	*			Ra+Rc+M+(B+3)L	M=1: Shift A and (W) left logical B places (B ≤ 31).
SRA B	1	0 0 0 0	0 0	0 0	0 0	0 0	B B B B				✓	+	*			Ra+(B+4)L	M=0: Shift A right arithmetic B places (B ≤ 15).
		0 0 0 0	0 0	0 0	0 0	0 B	B B B B				✓	+	*			Ra+(B+4)L	M=1: Shift A and OR right arithmetic B places (B ≤ 31).
SRA B ECR	1	0 0 0 0	0 0	0 1	0 0	0 0	B B B B		✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=0: Shift CR right arithmetic B places (B ≤ 15).
		0 0 0 0	0 0	0 1	0 0	0 B	B B B B		✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=1: Shift A and CR right arithmetic B places (B ≤ 31).
SRA B W	2	0 0 0 0	0 0	1 1	0 0	0 0	B B B B				✓	+	*			Ra+Rc+M+(B+3)L	M=0: Shift (W) right arithmetic B places (B ≤ 15).
		0 0 0 0	0 0	1 1	0 0	0 B	B B B B				✓	+	*			Ra+Rc+M+(B+3)L	M=1: Shift A and (W) right arithmetic B places (B ≤ 31).
SRE B	1	0 0 0 0	0 0	0 0	0 0	1 1	B B B B				✓	+	*			Ra+(B+4)L	M=0: Shift A right end-around B places (B ≤ 15).
SRE B ECR	1	0 0 0 0	0 0	0 1	0 0	1 1	B B B B		✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=0: Shift CR right end-around B places (B ≤ 15).
SRE B W	2	0 0 0 0	0 0	1 1	0 0	1 1	B B B B				✓	+	*			Ra+Rc+M+(B+3)L	M=0: Shift (W) right end-around B places (B ≤ 15).
SRL B	1	0 0 0 0	0 0	0 0	0 0	1 0	B B B B				✓	+	*			Ra+(B+4)L	M=0: Shift A right logical B places (B ≤ 15).
		0 0 0 0	0 0	0 0	0 0	1 B	B B B B				✓	+	*			Ra+(B+4)L	M=1: Shift A and OR right logical B places (B ≤ 31).
SRL B ECR	1	0 0 0 0	0 0	0 1	0 0	1 0	B B B B		✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=0: Shift CR right logical B places (B ≤ 15).
		0 0 0 0	0 0	0 1	0 0	1 B	B B B B		✓	✓	✓	✓	✓	✓	✓	Ra+(B+5)L	M=1: Shift A and CR right logical B places (B ≤ 31).
SRL B W	2	0 0 0 0	0 0	1 1	0 0	1 0	B B B B				✓	+	*			Ra+Rc+M+(B+3)L	M=0: Shift (W) right logical B places (B ≤ 15).
		0 0 0 0	0 0	1 1	0 0	1 B	B B B B				✓	+	*			Ra+Rc+M+(B+3)L	M=1: Shift A and (W) right logical B places (B ≤ 31).
STO N	1	0 1 0 0	0	N N N N N N N N N N							✓	0	✓			Ra+M+18L	Replace (N) by A.
STO ,D	2	0 1 0 0	0	0 0 0 0 0 0 0 0 0 0							✓	0	✓			Ra+M+18L	Replace D by A.
STO /P	1	0 1 0 0	1	X 0 0		P P P P P P P P					✓	0	✓			Ra+2M+19L	Replace ((P)) by A.
STO /P+	1	0 1 0 0	1	X 0 1		P P P P P P P P					✓	0	✓			Ra+2M+34L	Increment (P). Replace ((P)) by A.
STO /P-	1	0 1 0 0	1	X 1 1		P P P P P P P P					✓	0	✓			Ra+2M+34L	Replace ((P)) by A. Decrement (P).
STO .W	2	0 1 0 0	1	X X X		0 0 0 0 0 0 0 0					✓	0	✓			Ra+Rc+M+18L	Replace (W) by A.
SUB N	1	1 0 1 0	0	N N N N N N N N N N							✓	✓	✓	✓		2Ra+18L	Subtract A from (N), result in A. C is used if M is set.
SUB ,D	2	1 0 1 0	0	0 0 0 0 0 0 0 0 0 0							✓	✓	✓	✓		2Ra+18L	Subtract A from D, result in A. C is used if M is set.
SUB /P	1	1 0 1 0	1	X 0 0		P P P P P P P P					✓	✓	✓	✓		2Ra+M+19L	Subtract A from ((P)), result in A. C is used if M is set.
SUB /P+	1	1 0 1 0	1	X 0 1		P P P P P P P P					✓	✓	✓	✓		2Ra+M+34L	Increment (P). Subtract A from ((P)), result in A. C is used if M is set.
SUB /P-	1	1 0 1 0	1	X 1 1		P P P P P P P P					✓	✓	✓	✓		2Ra+M+34L	Subtract A from ((P)), result in A. C is used if M is set. Decrement (P).
SUB .W	2	1 0 1 0	1	X X X		0 0 0 0 0 0 0 0					✓	✓	✓	✓		2Ra+Rc+18L	Subtract A from (W), result in A. C is used if M is set.

SYSTEM TIMING

F100-L processor:

Ra	1162 ns
Rc	1743 ns
Wc	1328 ns
M	1909 ns
L	166.66 ns

RAM memory:

Ac	420 ns
W	120 ns

Appendix 2

Syntax of monitor program dialogue

Program listing

F100-L object program loader format

Syntax of monitor program dialogue

<decimal digit>::=0|1|2|3|4|5|6|7|8|9

<octal digit>::=0|1|2|3|4|5|6|7

<hexadecimal digit>::=0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

<terminator>::=NL

<sign>::=+|-

<decimal>::=$\{$<decimal digit>$\}_{1}^{5}$

in the range 0 to 32767

<octal>::=$\{$<octal digit>$\}_{1}^{6}$

in the range 0 to 177777

<hexadecimal>::=$\{$<hexadecimal digit>$\}_{1}^{4}$

in the range 0 to FFFF

<number>::=<decimal>|<octal>|<hexadecimal>

<signed number>::=<number>|-<decimal>

<radix>::=D|H|O

<breakin>::=SOH

<error warning>::=?NL→

<copy command>::=C<number>,<number>,<number><terminator>

<dump command>::=D<number>,<number><terminator>

with second number greater than first number

<execute command>::=E<terminator>|E<number><terminator>

<load command>::=L<terminator>|L<number><terminator>

<modify command>::=M<number><terminator>

<modify reply>::=•|NL|<sign><number><terminator>|

A<signed number><terminator>

<user command>::=<copy command>|<dump command>|

<execute command>|<load command>|

<modify command>|<radix><NL>

Notes

(1) the monitor program prompts with ' → '

(2) errors are flagged with the sequence '?NL → '

(3) the default radix is decimal:the radix remains set until altered by the user.

(4) load commands may be displaced:'L n' will cause n to be added to the base address given in the loader format.

(5) as a result of a modify (M) command, the monitor prints the address being inspected (in the appropriate radix) followed by the contents of the address: a modify reply may either alter the data (A<signed number><terminator>), inspect the following location (NL), inspect a relative location (<sign><number>) or terminate the modify command (.).

```
#MASTER MON
#LOWER/1024
   FLAG,STADDR,DATA/72,CT,WORD,LSTACK,DISP
   CHECK,TERM,BINARY,CT1,AD1,AD2,AD3,TEMP,CT3
   WAIT,TEMP1,SUM,ADSUM,PSTADR,CT2,RADIX,NEG
#LOWER
TABLE- 10000
       1000
       100
       10
EX- *55
    *63
    *62
    *67
    *66
    *70
#PROGRAM/2080
#COMMENT   INTERRUPT ROUTINES  #
      JBC  1  FLAG  IR1
      CLR  1  FLAG           (FROM TT1
      LDA  .32757            (STATUS WORD
      STO  CHECK
      RTN
IR1-  JBC  2  FLAG  IR3
      CLR  2  FLAG           (FROM TT2
      LDA  .32757            (STATUS WORD
      STO  CHECK
      JBC  10  CHECK  IR2    (NOT NL
      SET  10  FLAG          (NL
IR2-  RTN
IR3-  CLR  3  FLAG           (FROM TT3
      LDA  .32757            (STATUS WORD
      RTN
#COMMENT  READ CHAR                    #
#"COMMENT  CHARACTER INTO 'CHECK'      #
TT1-  SET  1  FLAG
      LDA  .32757
      LDA  ,*100000
      STO  .32758            (CONTROL WORD
TT11- JBS  1  FLAG  TT11     (WAIT FOR INTERRUPT
      CLR  I
      LDA  CHECK             (CHARACTER
      AND  ,*177
      CMP  ,1
      JBS  Z  MAIN           (BREAK IN
      RTN
#COMMENT  READ BLOCK                   #
#COMMENT  BLOCK AT ADDRESS 'DATA'      #
TT2-  SET  2  FLAG
      LDA  .32757            (STATUS WORD
      LDA  ,DATA-1
      STO  .32759            (ADDRESS FOR DATA
      LDA  CT
      ADD  ,*110000
      STO  .32758            (CONTROL WORD
TT21- JBS  2  FLAG  TT21     (WAIT FOR INTERRUPT
```

```
      CLR  I
      RTN
#COMMENT  WRITE 'CT' CHAR/S FROM ADDRESS 'DATA'  #
TT3-  SET  3  FLAG
      LDA  .32757            (STATUS WORD
      LDA  ,DATA-1           (ADDRESS-1
      STO  .32759            (ADDRESS FOR DATA
      LDA  ,*100000
      NEG  CT
      STO  .32758            (CONTROL WORD
TT31- JBS  3  FLAG  TT31     (WAIT FOR INTERRUPT
      CLR  I
      RTN
#COMMENT   ERROR ROUTINE                              #
#COMMENT  PRINTS '?NL PROMPT' AND RETURNS TO MAIN  #
ERR-  LDA  ,3
      STO  CT
      LDA  ,*77              ('?')
      STO  DATA
      LDA  ,*12              ('NL')
      STO  DATA+1
      LDA  ,*137             ('PROMPT')
      STO  DATA+2
      JMP  .MAIN2
#COMMENT   RUN OUT PAPER TAPE LEADER  #
ROUT- LDA  ,-100
      STO  CT1
      LDA  ,1
      STO  CT
      LDA  ,0                (BLANK
      STO  DATA
RU1-  CAL  .TT3
      ICZ  CT1  RU1
      RTN
#COMMENT  CONVERT 'BINARY' TO  2'S COMPLEMENT  #
TWSC- LDA  ,-16
      STO  CT2
      LDA  ,0
      STO  NEG
TWS1- SRE  1  BINARY
      JBS  S  TWS2
      ICZ  CT2  TWS1
      JMP  .TWS5
TWS2- LDA  ,*100000
      ADS  NEG
      ICZ  CT2  TWS3
      JMP  .TWS5
TWS3- SRL  1  NEG
      SRE  1  BINARY
      JBC  S  TWS4
      ICZ  CT2  TWS3
      JMP  .TWS5
TWS4- LDA  ,*100000
      ADS  NEG
      ICZ  CT2  TWS3
TWS5- LDA  NEG
```

```
       STO  BINARY
       RTN
#COMMENT   PACK PROGRAM FROM LOADER FORMAT        #
#COMMENT  ON ENTRY LOCATION 254 HOLDS ADDRESS     #
#COMMENT  OF DATA-1 WHERE DATA - DATA+3 HOLDS     #
#COMMENT  4 CHARACTERS TO BE PACKED TO ONE WORD   #
#COMMENT  ON EXIT 'WORD' HOLDS 16-BIT RESULT      #
PACK- LDA  ,0
       STO  WORD
       LDA  ,-4
       STO  CT                  (4 CHARACTERS PER DATA WORD
P1-    SLL  4  WORD
       CAL  .TT1
       LDA  CHECK
       JBS  10  P2              ('NL'
       AND  ,*17
       ADS  ADSUM               (CHECK SUM
       ADS  WORD
       ICZ  CT  P1
P2-    RTN
#COMMENT  READ OCTAL NUMBER FROM TTY    #
#COMMENT  STORE IN 'DATA ' - 'DATA+5'   #
OCT-   LDA  ,DATA-1
       STO  255
       LDA  ,-6                 (MAXIMUM 6 DIGITS
       STO  CT
       LDA  ,0
OCT1-  STO  /255+
       ICZ  CT  OCT1
       LDA  ,DATA-1
       STO  255
       JSC  15  FLAG  OCT3
OCT2-  CAL  .TT1
OCT3-  LDA  CHECK
       AND  ,*177
       JBS  14  FLAG  OCT4
       CMP  TERM                (CHARACTER STRING TERMINATOR
       JBS  Z  OCT6
OCT4-  STO  TEMP
       LDA  ,-6
       CMP  CT
       JBS  Z  ERR              (TOO MANY DIGITS
       LDA  ,-1
       ADS  CT
       LDA  TEMP
       CMP  ,*67
       JBS  S  ERR              (>7  NON OCTAL CHARACTER
       CMP  ,*56
       JBS  S  OCT5             (<0  NON OCTAL CHARACTER
       JBC  14  FLAG  ERR
       STO  TERM
       JMP  .OCT6
OCT5-  STO  /255+
       LDA  ,*177720            (-*60  ASCII DISPLACEMENT
       ADS  /255
       JMP  .OCT2
```

```
OCT6- LDA  ,DATA-1
      CMP  255
      JBS  Z  OCT9
      LDA  ,-6
      CMP  CT
      JBS  Z  OCT9
      LDA  ,DATA+5
      STO  254
OCT7- LDA  /255-
      STO  /254-
      ICZ  CT  OCT7
OCT8- LDA  ,0
      STO  /254-
      LDA  ,DATA-1
      CMP  254
      JBC  Z  OCT8
OCT9- RTN
#COMMENT  READ DECIMAL NUMBER FROM TTY  #
#COMMENT  SRORE IN 'DATA' - 'DATA+5'    #
DEC-  LDA  ,DATA-1
      STO  255
      LDA  ,-6              (MAXIMUM NO DIGITS PLUS SIGN
      STO  CT
      LDA  ,0
DEC1- STO  /255+
      ICZ  CT  DEC1
      LDA  ,DATA-1
      STO  255
      LDA  ,*40
      STO  /255+
      JSC  15  FLAG  DEC2
      CAL  .TT1
DEC2- JBS  10  CHECK  DEC11
      LDA  CHECK
      AND  ,*177
      CMP  ,*55             (NEGATIVE
      JBC  Z  DEC5
DEC3- STO  /255
DEC4- CAL  .TT1
      LDA  CHECK
      AND  ,*177
      JBS  14  FLAG  DEC6
DEC5- CMP  TERM
      JBS  Z  DEC8
DEC6- STO  TEMP
      LDA  ,-5
      CMP  CT
      JBS  Z  ERR           (TOO LONG
      LDA  ,-1
      ADS  CT
      LDA  TEMP
      CMP  ,*71             (>9  NON DECIMAL CHARACTER
      JBS  S  ERR
      CMP  ,*56
      JBS  S  DEC7          (<0  NON DECIMAL CHARACTER
      JBC  14  FLAG  ERR
```

```
        STO   TERM
        JMP   .DEC8
DEC7-   STO   /255+
        LDA   ,*177720          (-*60  ASCII DISPLACEMENT
        ADS   /255
        JMP   .DEC4
DEC8-   LDA   ,-5
        CMP   CT
        JBS   Z  DEC11
        LDA   ,DATA+5
        STO   254
DEC9-   LDA   /255-
        STO   /254-
        ICZ   CT  DEC9
DEC10-  LDA   ,0
        STO   /254-
        LDA   ,DATA
        CMP   254
        JBC   Z  DEC10
DEC11-  RTN
*COMMENT  READ HEX NUMBER FROM TTY    *
*COMMENT  STORE IN 'DATA' - 'DATA+5'  *
HEX-    LDA   ,DATA-1
        STO   255
        LDA   ,-4
        STO   CT                (MAXIMUM NO DIGITS
        LDA   ,0
HEX1-   STO   /255+
        ICZ   CT  HEX1
        LDA   ,DATA-1
        STO   255
        JSC   15  FLAG  HEX3
HEX2-   CAL   .TT1
HEX3-   LDA   CHECK
        AND   ,*177
        JBS   14  FLAG  HEX4
        CMP   TERM
        JBS   Z  HEX8
HEX4-   STO   TEMP
        LDA   ,-4
        CMP   CT
        JBS   Z  ERR
        LDA   ,-1
        ADS   CT
        LDA   TEMP
        CMP   ,*106             (>F  NON HEX CHARACTER
        JBS   S  ERR
        CMP   ,*56
        JBS   S  HEX5           (<0  NON HEX CHARACTER
        JBC   14  FLAG  ERR
        STO   TERM
        JMP   .HEX8
HEX5-   CMP   ,*71
        JBC   S  HEX6
        CMP   ,*77
        JBC   S  ERR
```

```
        LDA   ,*177711
        JMP   .HEX7
HEX6-   LDA   ,*177720
HEX7-   STO   /255+
        LDA   TEMP
        ADS   /255
        JMP   .HEX2
HEX8-   LDA   ,DATA-1
        CMP   255
        JBS   Z  HEX11
        LDA   ,-4
        CMP   CT
        JBS   Z  HEX11
        LDA   ,DATA+3
        STO   254
HEX9-   LDA   /255-
        STO   /254-
        ICZ   CT  HEX9
HEX10-  LDA   ,0
        STO   /254-
        LDA   ,DATA-1
        CMP   254
        JBC   Z  HEX10
HEX11-  RTN
#COMMENT  CONVERT OCTAL TO BINARY          #
#COMMENT  MINUS LENGTH OF OCTAL IN 'CT1'  #
#COMMENT  ADDRESS-1 OF DECIMAL IN TEMP    #
OTOB-   LDA   TEMP1
        STO   255
        LDA   ,0
        STO   BINARY
OB1-    LDA   BINARY
        CMP   ,*17777
        JBS   S  ERR              (O/FLOW
        SLL   3  BINARY
        LDA   /255+
        ADS   BINARY
        JBS   C  ERR              (O/FLOW
        ICZ   CT1  OB1
        RTN
#COMMENT  CONVERT DECIMAL TO BINARY         #
#COMMENT  MINUS LENGTH IN OF DECIMAL 'CT1'#
#COMMENT  ADDRESS-1 OF DECIMAL IN TEMP1    #
DTOB-   LDA   TEMP1
        STO   255
        LDA   ,0
        STO   BINARY
DB1-    LDA   ,-9
        STO   CT
        LDA   BINARY
DB2-    ADS   BINARY
        JBS   S  ERR
        ICZ   CT  DB2
        LDA   /255+
        ADS   BINARY
        JBS   S  ERR
```

```
        ICZ   CT1  DB1
        LDA   DATA
        CMP   ,*55                (-
        JBC   Z  DB3
        CAL   .TWSC               (TWO'S COMPLEMENT
DB3-    RTN
#COMMENT  CONVERT HEX TO BINARY          #
#COMMENT  MINUS LENGTH OF HEX IN 'CT1'#
#COMMENT  ADDRESS-1 IOF HEX IN TEMP1   #
HTOB-   LDA   TEMP1
        STO   255
        LDA   ,0
        STO   BINARY
HB1-    LDA   ,-15
        STO   CT
        LDA   BINARY
HB2-    ADS   BINARY
        JBS   C  ERR
        ICZ   CT  HB2
        LDA   /255+
        ADS   BINARY
        JBS   C  ERR
        ICZ   CT1  HB1
        RTN
#COMMENT  CONVERT BINARY TO OCTAL        #
#COMMENT  ON ENTRY NUMBER IN 'BINARY'    #
#COMMENT  ADDRESS-1 FOR OCTAL IN 255     #
BTOO-   LDA   BINARY
        STO   TEMP
        LDA   ,*60                (ASCII DISPLACEMENT
        STO   /255+
        JPZ   TEMP  BO1
        LDA   ,1
        ADS   /255
BO1-    SLL   1  TEMP
        LDA   ,-5
        STO   CT
BO2-    SLE   3  TEMP
        LDA   TEMP
        AND   ,*7
        ADD   ,*60                (ASCII DISPLACEMENT
        STO   /255+
        ICZ   CT  BO2
        RTN
#COMMENT  CONVERT BINARY TO DECIMAL      #
#COMMENT  ON ENTRY NUMBER IN 'BINARY'    #
#COMMENT  ADDRESS-1 FOR DECIMAL IN 255   #
BTOD-   LDA   BINARY
        CMP   ,*100000
        JBC   Z  BD2
        LDA   ,-6
        STO   CT
        LDA   ,EX-1
        STO   255
        LDA   ,DATA-1
        STO   254
```

```
BD1-   LDA  /255+
       STO  /254+
       ICZ  CT  BD1
       RTN
BD2-   JBC  S  BD3
       CAL  .TWSC              (TWO'S COMPLEMENT
       STO  TEMP
       LDA  ,*55
       JMP  .BD4
BD3-   STO  TEMP
       LDA  ,*60               (ASCII DISPLACEMENT
BD4-   STO  /255+
       LDA  ,-4
       STO  CT
       LDA  ,TABLE-1
       STO  254
BD5-   LDA  ,1
       ADS  255
       ADS  254
       LDA  ,*60               (ASCII DISPLACEMENT
       STO  /255
BD6-   LDA  /254
       SBS  TEMP
       JBS  S  BD7
       LDA  ,1
       ADS  /255
       JMP  .BD6
BD7-   ADS  TEMP
       ICZ  CT  BD5
       LDA  TEMP
       STO  /255+
       LDA  ,*60               (ASCII DISPLACEMENT
       ADS  /255
       RTN
#COMMENT  CONVERT BINARY TO HEX          #
#COMMENT  ON ENTRY NUMBER IN 'BINARY'    #
#COMMENT  ADDRESS-1 FOR HEX IN 255       #
BTOH-  LDA  BINARY
       STO  TEMP
       LDA  ,-4
       STO  CT
BH1-   SLE  4  TEMP
       LDA  TEMP
       AND  ,*17
       CMP  ,9
       JBS  S  BH2
       ADD  ,*60               (ASCII DISPLACEMENT
       JMP  .BH3
BH2-   ADD  ,*67
BH3-   STO  /255+
       ICZ  CT  BH1
       RTN
#COMMENT  READ NUMBER FROM TTY AND CONVERT TO   #
#COMMENT  BINARY WRT RADIX (DEC/OCT/HEX)        #
NUMBIN- LDA  RADIX
       SLL  4
```

```
        SJM
        LDA   ,DEC              (DECIMAL
        STO   253
        LDA   ,DTOB
        STO   252
        LDA   ,-5
        STO   .CT1
        LDA   ,DATA
        STO   .TEMP1
        JMP   .N1
        LDA   ,HEX              (HEX
        STO   253
        LDA   ,HTOB
        STO   252
        LDA   ,-4
        STO   .CT1
        LDA   ,DATA-1
        STO   .TEMP1
        JMP   .N1
        LDA   ,OCT              (OCTAL
        STO   253
        LDA   ,OTOB
        STO   252
        LDA   ,-6
        STO   .CT1
        LDA   ,DATA-1
        STO   .TEMP1
N1-     CAL   /253
        CAL   /252              (CONVERT
        RTN
$COMMENT  CONVERT FROM BINARY WRT RADIX    $
$COMMENT  (DEC/OCT/HEX) AND PRINT          $
NUMBOUT- LDA  ,DATA-1
        STO   255
        LDA   RADIX
        SLL   3
        SJM
        CAL   .BTOD
        LDA   ,6
        STO   .CT
        JMP   .N2
        CAL   .BTOH
        LDA   ,4
        STO   .CT
        JMP   .N2
        CAL   .BTOO
        LDA   ,6
        STO   .CT
N2-     CAL   .TT3
        RTN
$COMMENT  UNPACK TO LOADER FORMAT FROM PROGRAM          $
$COMMENT  WORD POINTED AT BY 255 UNPACKED TO 4 CHARS    $
$COMMENT  IN WORDS POINTED AT BY 254                    $
UNPACK- LDA   ,-4
        STO   CT
U1-     LDA   /255
```

```
        SLE   4
        STO   /255
        AND   ,*17
        ADS   SUM
        STO   /254+
        JBC   Z  U2
        LDA   ,*60
        JMP   .U3
U2-     LDA   ,*100
U3-     ADS   /254
        ICZ   CT  U1
        RTN
#COMMENT  HEADER STRIP FOR LOADER FORMAT  #
HEADER- LDA   ,*12
        STO   DATA
        STO   DATA+2
        LDA   ,0
        STO   DATA+1
        STO   DATA+3
        LDA   ,4
        STO   CT
        CAL   .TT3                (WRITE
        RTN
#COMMENT  COPY A BLOCK (AD1 TO AD2) FROM AD1 TO AD3  #
COPY-   LDA   ,*54                (COMMA
        STO   TERM
        CAL   .NUMBIN
        LDA   BINARY
        STO   AD1
        CAL   .NUMBIN
        LDA   BINARY
        STO   AD2
        LDA   ,*12                (NL
        STO   TERM
        CAL   .NUMBIN
        LDA   BINARY
        STO   AD3
        LDA   AD2
        SUB   AD1
        STO   CT
        LDA   ,-1
        ADS   CT
        LDA   AD3
        SUB   AD2                 (AD2-AD3
        JBS   S  C7
        LDA   AD1
        CMP   AD3
        JBC   S  C9
C7-     LDA   AD1                 (FORWARDS
        STO   253
        LDA   AD3
        STO   252
        LDA   ,1
        SBS   252
        SBS   253
C8-     LDA   /253+
```

```
        STO   /252+
        ICZ   CT  C8
        JMP   .MAIN1
C9-     LDA   AD2                 (BACKWARDS
        STO   253
        LDA   AD3
        STO   252
        LDA   AD1
        SUB   AD2
        ADS   252
C10-    LDA   /253-
        STO   /252-
        ICZ   CT  C10
        JMP   .MAIN1
*COMMENT DUMP BLOCK (AD1 TO AD2) TO TTY IN LOADER FORMAT  *
DUMP-   LDA   ,*54          (COMMA
        STO   TERM
        CAL   .NUMBIN
        LDA   BINARY
        STO   AD1
        LDA   ,*12          (NL
        STO   TERM
        CAL   .NUMBIN
        LDA   BINARY
        STO   AD2
        LDA   AD1
        SUB   AD2                 (AD2-AD1
        JBS   S  ERR
        STO   CT2
        LDA   ,0
        STO   CT3
        LDA   ,1
        ADS   CT2                 (NO OF WORDS
        LDA   ,1
        STO   CT
        LDA   ,*137
        STO   DATA
        CAL   .TT3
        CAL   .ROUT
        CAL   .HEADER
D1-     LDA   ,*101               ( 'A' LINE
        STO   DATA
        LDA   ,1
        STO   SUM                 (CHECK SUM
        ADS   CT3
        LDA   ,15
        CMP   CT2
        JBC   S  D3
        LDA   CT2                 (PART LINE
        STO   CT1
        SLL   2
        ADD   ,12
        STO   TEMP1
        LDA   ,0
        STO   CT2
        JMP   .D4
```

```
D3-   LDA  ,-15
      ADS  CT2
      LDA  ,72
      STO  TEMP1
      LDA  ,15
      STO  CT1
D4-   LDA  CT1
      AND  ,*17
      ADS  SUM
      STO  DATA+1
      LDA  ,*100
      ADS  DATA+1
      LDA  ,AD1                  (DESTINATION
      STO  255
      LDA  ,DATA+5
      STO  254
      CAL  .UNPACK
D6-   LDA  AD1
      STO  255
      CAL  .UNPACK
      LDA  ,1
      ADS  AD1
      SBS  CT1
      JBC  Z  D6
      LDA  CT2
      JBC  Z  D7
      LDA  ,*12
      STO  /254+
      LDA  ,0
      STO  /254+
D7-   LDA  SUM                   (INSERT CHECKSUM
      STO  TEMP
      LDA  ,TEMP
      STO  255
      LDA  ,DATA+1
      STO  254
      CAL  .UNPACK
      LDA   ,*12
      STO  DATA+70
      LDA  ,0
      STO  DATA+71
      LDA  TEMP1                 (WRITE LINE
      STO  CT
      CAL  .TT3
      LDA  CT2
      JBC  Z  D1
      LDA  ,*102                 ('B'LINE
      STO  DATA
      LDA  ,5
      STO  SUM
      LDA  ,*103
      STO  DATA+1
      LDA  ,*60
      STO  DATA+18
      STO  DATA+19
      STO  DATA+20
```

```
        LDA   CT3
        ADD   ,*100
        ADS   SUM
        STO   DATA+21
        LDA   ,LSTACK
        STO   255
        LDA   ,DATA+5
        STO   254
        CAL   .UNPACK
        LDA   ,STADDR
        STO   255
        CAL   .UNPACK
        LDA   ,PSTADR
        STO   255
        CAL   .UNPACK
        LDA   SUM
        STO   TEMP
        LDA   ,TEMP
        STO   255
        LDA   ,DATA+1
        STO   254
        CAL   .UNPACK
        LDA   ,*12
        STO   DATA+22
        STO   DATA+24
        STO   DATA+26
        LDA   ,0
        STO   DATA+23
        STO   DATA+25
        STO   DATA+27
        LDA   ,28
        STO   CT
        CAL   .TT3                  (WRITE LINE
        CAL   .ROUT
        JMP   .MAIN1
#COMMENT  EXECUTE AT 2K OR GIVEN ADDRESS  #
EXEC-   LDA   ,*12
        STO   TERM
        CAL   .NUMBIN
        LDA   BINARY
        JBS   Z  2048
        LDA   ,-1
        STO   252
        LDA   /252+                 (LINK STACK POINTER
        STO   252
        LDA   BINARY
        STO   /252+
        LDA   ,-1
        STO   252
        LDA   ,2
        ADS   /252+
        RTC
#COMMENT  LOAD PROGRAM FROM PAPER TAPE  #
LOAD-   LDA   ,0
        STO   DISP
L1-     LDA   ,*12
```

```
        STO   TERM
        CAL   .NUMBIN
        LDA   BINARY
        JBS   Z  L2
        LDA   BINARY
        STO   DISP
L2-     CAL   .TT1                  (READ CHARACTER
        LDA   CHECK
        AND   ,*177
        CMP   ,0                    (BLANK
        JBS   Z  L2
        CMP   ,*12                  (NL?
        JBS   Z  L2
        CMP   ,*101                 (A?
        JBC   Z  L6
        AND   ,*17
        STO   TEMP
        CAL   .TT1
        LDA   CHECK
        AND   ,*17
        ADS   TEMP
        CAL   .PACK
        LDA   WORD
        STO   SUM
        LDA   TEMP
        STO   ADSUM
        CAL   .PACK
        LDA   ,1
        SBS   WORD
        LDA   WORD
        ADD   DISP
        STO   254                   (ADDRESS POINTER
L4-     CAL   .PACK
        JSC   10  CHECK  L5         (END OF LINE
        LDA   WORD
        STO   /254+                 (STORE DATA WORD
        JMP   .L4
L5-     LDA   ADSUM                 (CHECKSUM CORRECT
        CMP   SUM
        JBC   Z  ERR
        JMP   .L2
L6-     CMP   ,*102                 (B?
        JBC   Z  ERR
        AND   ,*17
        STO   TEMP
        CAL   .TT1
        LDA   CHECK
        AND   ,*17
        ADS   TEMP
        CAL   .PACK
        LDA   WORD
        STO   SUM
        LDA   TEMP
        STO   ADSUM
        CAL   .PACK                 (LINK STACK POINTER
        LDA   WORD
```

```
        CMP   ,0
        JBC   Z  L8
        LDA   ,*3001              (DEFAULT =1.5K+1
L8-     STO   LSTACK
        CAL   .PACK               (USER RESET ADDRESS
        LDA   WORD
        CMP   ,0
        JBC   Z  L9
        LDA   ,2048               (DEFAULT =2K
L9-     STO   STADDR
        CAL   .PACK               (PROG START ADDRESS
        LDA   WORD
        CMP   ,0
        JBC   Z  L10
        LDA   ,2048               (DEFAULT =2K
L10-    STO   PSTADR
        CAL   .PACK
        LDA   ADSUM
        CMP   SUM
        JBC   Z  ERR
        JMP   .MAIN1
#COMMENT  MODIFY  #
MOD-    LDA   ,*12
        STO   TERM
        CAL   .NUMBIN
        LDA   BINARY
        STO   AD1
M1-     LDA   ,*100000            (WAIT
        STO   WAIT
M2-     ICZ   WAIT  M2
        CAL   .NUMBOUT            (PRINT ADDRESS
        LDA   ,1
        STO   CT
        LDA   ,*40
        STO   DATA
        CAL   .TT3                (PRINT SPACE
        LDA   AD1
        STO   255
        LDA   /255
        STO   BINARY
        CAL   .NUMBOUT            (PRINT DATA
M4-     LDA   ,1
        STO   CT
        LDA   ,*72
        STO   DATA
        CAL   .TT3                (PRINT COLON
        CAL   .TT1                (READ CHAR
        LDA   CHECK
        AND   ,*177
        CMP   ,*56                (FULL STOP
        JBS   Z  M10
        CMP   ,*53                ('+'
        JBC   Z  M7
        CAL   .NUMBIN
        LDA   BINARY
        ADS   AD1
```

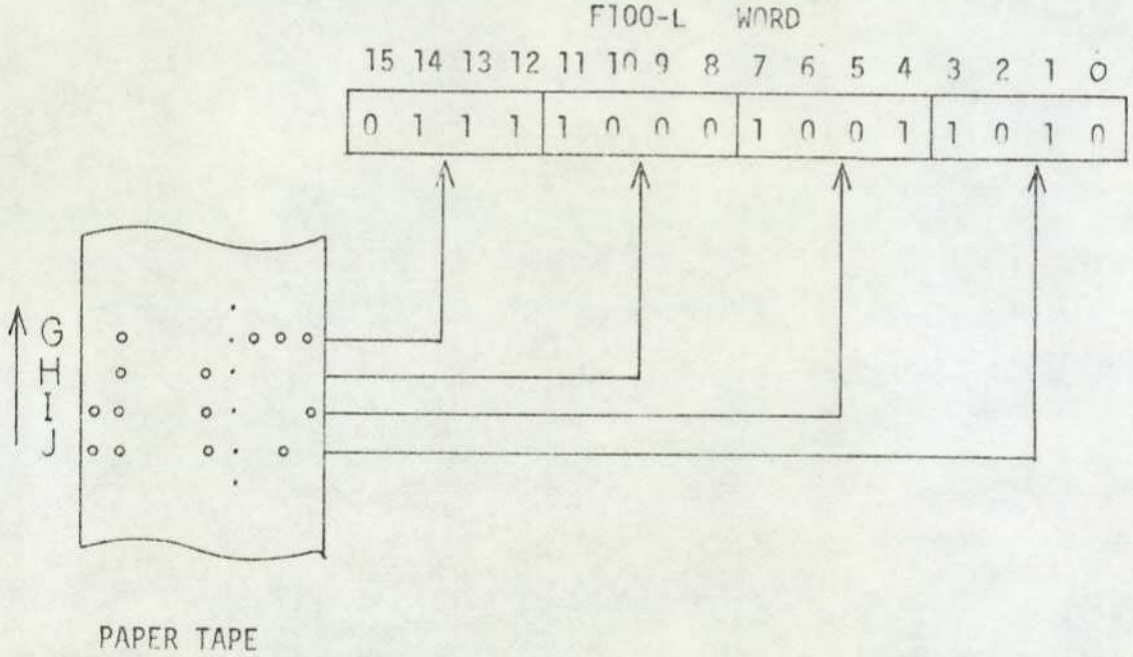
```
        LDA   AD1
        STO   BINARY
        JMP   .M1
M7-     CMP   ,*55                    ('-'
        JBC   Z  M8
        CAL   .NUMBIN
        LDA   BINARY
        SBS   AD1
        LDA   AD1
        STO   BINARY
        JMP   .M1
M8-     CMP   ,*12
        JBC   Z  M9
        LDA   ,1
        ADS   AD1
        LDA   AD1
        STO   BINARY
        JMP   .M1
M9-     CMP   ,*101                   (A?
        JBC   Z  M4
        LDA   ,*12
        STO   TERM
        CAL   .NUMBIN
        LDA   AD1
        STO   252
        LDA   BINARY
        STO   /252
        LDA   AD1
        STO   BINARY
        JMP   .M1
M10-    LDA   ,2
        STO   CT
        LDA   ,*12
        STO   DATA
        LDA   ,*137
        STO   DATA+1
        JMP   .MAIN2
#COMMENT    SET RADIX  0:DEC,1:OCT,2:HEX  #
RADI-   CAL   .TT1                    (READ CHAR
        LDA   CHECK
        AND   ,*177
        CMP   ,*104                   (DEC
        JBC   Z  R1
        LDA   ,0
        STO   RADIX
        JMP   .R3
R1-     CMP   ,*110                   (HEX
        JBC   Z  R2
        LDA   ,1
        STO   RADIX
        JMP   .R3
R2-     CMP   ,*117                   (OCT
        JBC   Z  ERR
        LDA   ,2
        STO   RADIX
R3-     CAL   .TT1
```

```
       JBC   10  CHECK  ERR
       JMP   .MAIN1
#COMMENT  MAIN PROGRAM LOOP  #
MAIN-  LDA   ,0
       STO   FLAG
       STO   RADIX
       STO   LSTACK
       STO   PSTADR
       STO   STADDR
MAIN1- LDA   ,1
       STO   CT
       LDA   ,*137
       STO   DATA
MAIN2- CAL   .TT3
       CAL   .TT1
       LDA   CHECK
       AND   ,*177
       CMP   *103                  (COPY
       JBS   Z  COPY
       CMP   ,*104                 (DUMP
       JBS   Z  DUMP
       CMP   ,*105                 (EXECUTE
       JBS   Z  EXEC
       CMP   ,*114                 (LOAD
       JBS   Z  LOAD
       CMP   ,*115                 (MODIFY
       JBS   Z  MOD
       CMP   ,*122                 (RADIX
       JBS   Z  RADI
       JMP   .ERR
#END
#FINISH
17-19-49_
```

F100-L Object Program Loader Format

The object program produced by the link editor is output as a series of object program records on paper tape. These records contain information specifying the contents of F100-L 16-bit words. Each F100-L word is represented by the last four bits of a set of four characters taken in order. The character set used consists of the number 0 (zero) and the letters A to O. Thus the four letters OOOO represent an F100-L word with all the bits set, i.e. the number -1.

The illustration below shows how the integer value 30874 is obtained from the four characters GHIJ which are assumed to be punched on the paper tape in the ISO Eight-Hole Tape Code.



The object program records are of two types. Type 1 is a data record of variable length and the last record is a Type 2 record. An object program consists of a number of Type 1 records followed by a terminating Type 2 record. The structure of these records is shown below. The check sum consists of the sum of the last 4 bits of every character in a record, excluding the four characters comprising itself and any record terminators. The record terminator is generated by the Fortran output system of the host computer.

Character	Description
1	Block type = 1 (i.e. character A)
2	Number of data words in record
3	Check sum
4	
5	
6	
7	Destination address
8	
9	
10	
11-70	Up to 15 data words, each represented by 4 chars

Object Program Record - Data Type

Character	Description
1	Block type = 2 (i.e. character B)
2	Number of words in record = 3 (i.e.character C)
3	
4	
5	Check sum
6	
7	
8	
9	Initial value of the link stack pointer (zero if not specified)
10	
11	
12	
13	Address of user's reset routine (zero if not specified)
14	
15	
16	
17	Processor start address (zero if not specified)
18	
19	
20	
21	Total number of preceeding data type records
22	

Object Program Record - Terminator Type

Appendix 3

General software model

```
'TYPE' BUFF = 'ARRAY'(.1..BUFFERSIZE.) OF 'BYTE';

'TYPE' AVAILABLE = 0..BUFFERSIZE;

'TYPE' INDEX = 1..BUFFERSIZE;

'TYPE' BUFFER =
'MONITOR'
'VAR' DATA,FRAMES:AVAILABLE;
      POINTER1,POINTER2:INDEX;
      PRODUCER,CONSUMER:QUEUE;
      CONTENTS:BUFF;

'PROCEDURE''ENTRY' CONSUME('VAR' SAMPLE:BYTE);
'BEGIN'
     'IF' DATA = 0 'THEN' DELAY(CONSUMER);
     DATA:=DATA-1;
     SAMPLE:=CONTENTS[POINTER1];
     POINTER1:=SUCC(POINTER1);
     CONTINUE(PRODUCER);
'END';

'PROCEDURE''ENTRY' PRODUCE(SAMPLE:BYTE);
'BEGIN'
     'IF' FRAMES = 0 'THEN' DELAY(PRODUCER);
     FRAMES:=FRAMES-1;
     CONTENTS[POINTER2]:=SAMPLE;
     POINTER2:=SUCC(POINTER2);
     CONTINUE(CONSUMER);
'END';

'BEGIN' DATA:=0;
        FRAMES:=BUFFERSIZE;
        POINTER1:=POINTER2:=1
'END';
```

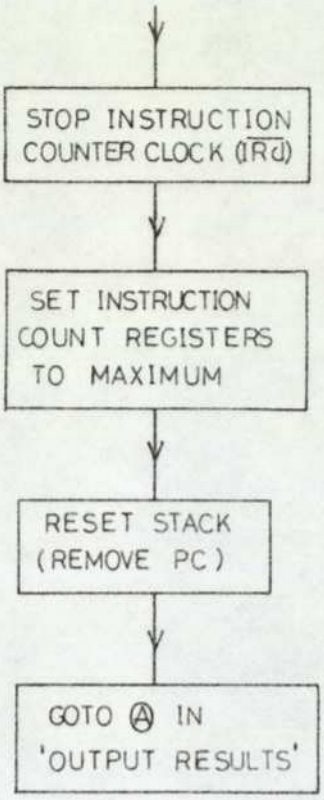
```
'TYPE' IOPROCESS =
'PROCSS'(BUFFER1,BUFFER2:BUFFER);
'VAR' SAMPLE:BYTE;
'BEGIN'
     'CYCLE'
          COUNTI;
          INPUT(SAMPLE);
          BUFFER1.PRODUCE(SAMPLE);
          BUFFER2.CONSUME(SAMPLE);
          OUTPUT(SAMPLE)
          'UNTIL' NOSAMPLES;
     'END';
'END';



'TYPE' FILTERPROCESS =
'PROCESS'(BUFFER1,BUFFER2:BUFFER);
'VAR' SAMPLE:BYTE;
'BEGIN'
     'CYCLE'
          BUFFER1.CONSUME(SAMPLE);
          DIGITALFILTER(SAMPLE);
          COUNTF;
          BUFFER2.PRODUCE(SAMPLE)
          'UNTIL' NOSAMPLES;
     'END';
'END';



'VAR' BUFFER1,BUFFER2:BUFFER;
      INOUT:IOPROCESS;
      FILTER:FILTERPROCESS;



'INIT' BUFFER1,BUFFER2,
       INOUT(BUFFER1,BUFFER2),
       FILTER(BUFFER1,BUFFER2);
```

Appendix 4

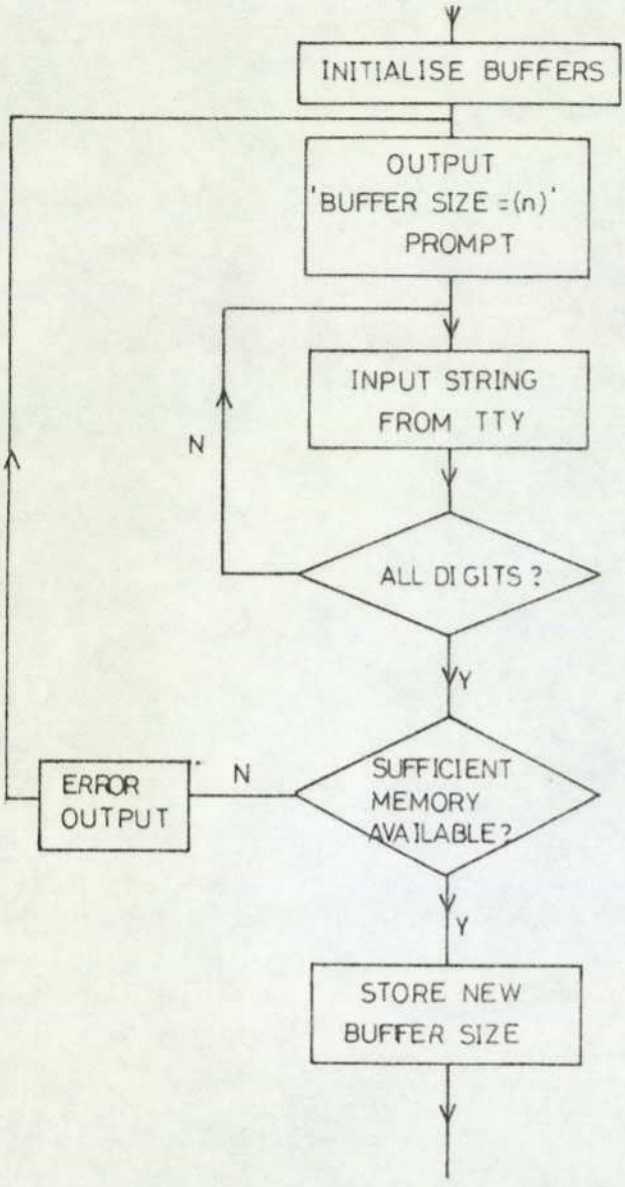
Hardware monitor control program block diagram

INTERRUPT ROUTINE

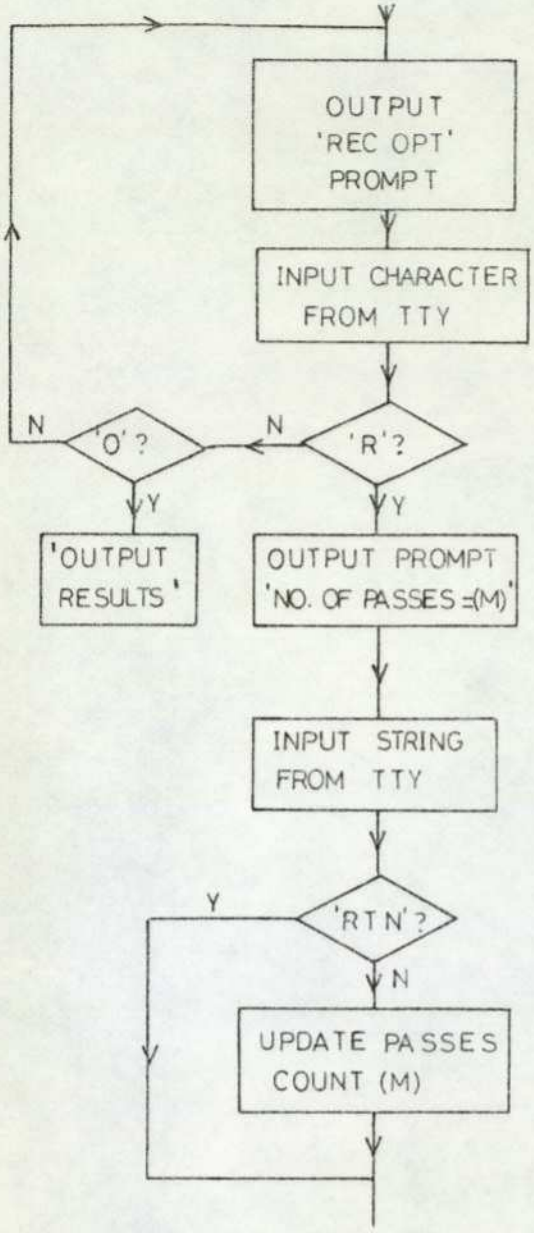
(on instruction counter overflow)

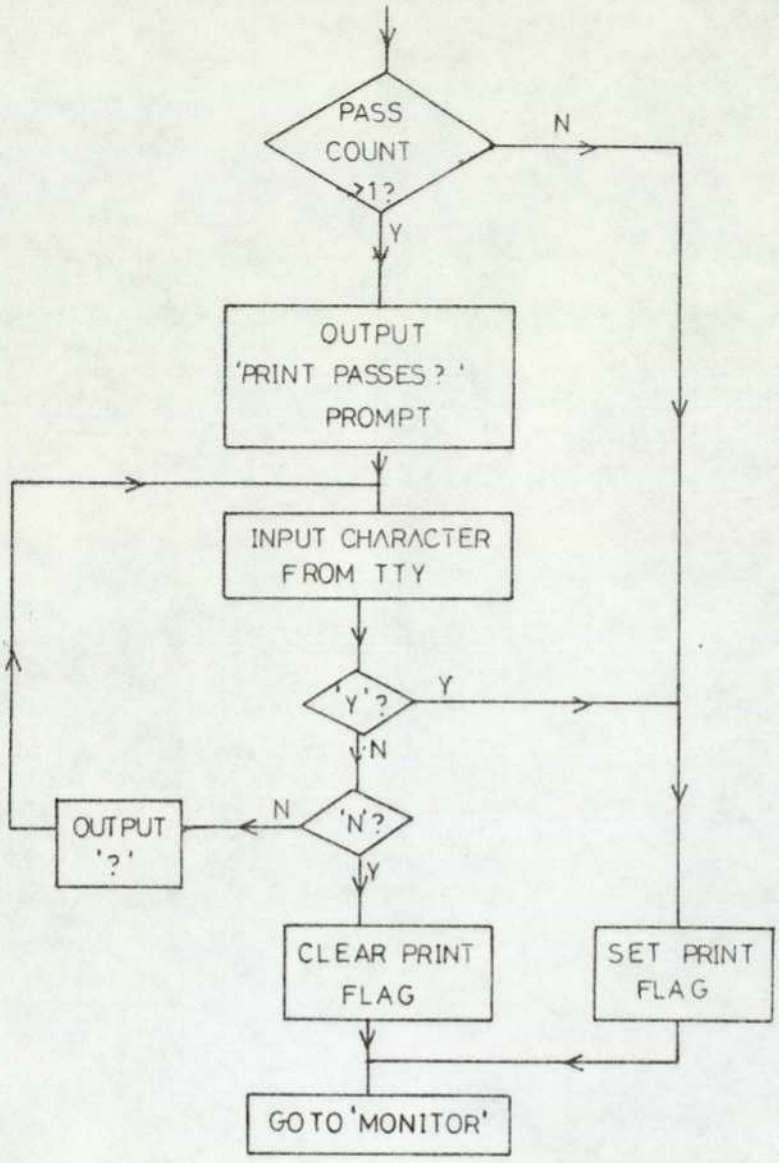


INITIALISE AND SELECT BUFFER SIZE

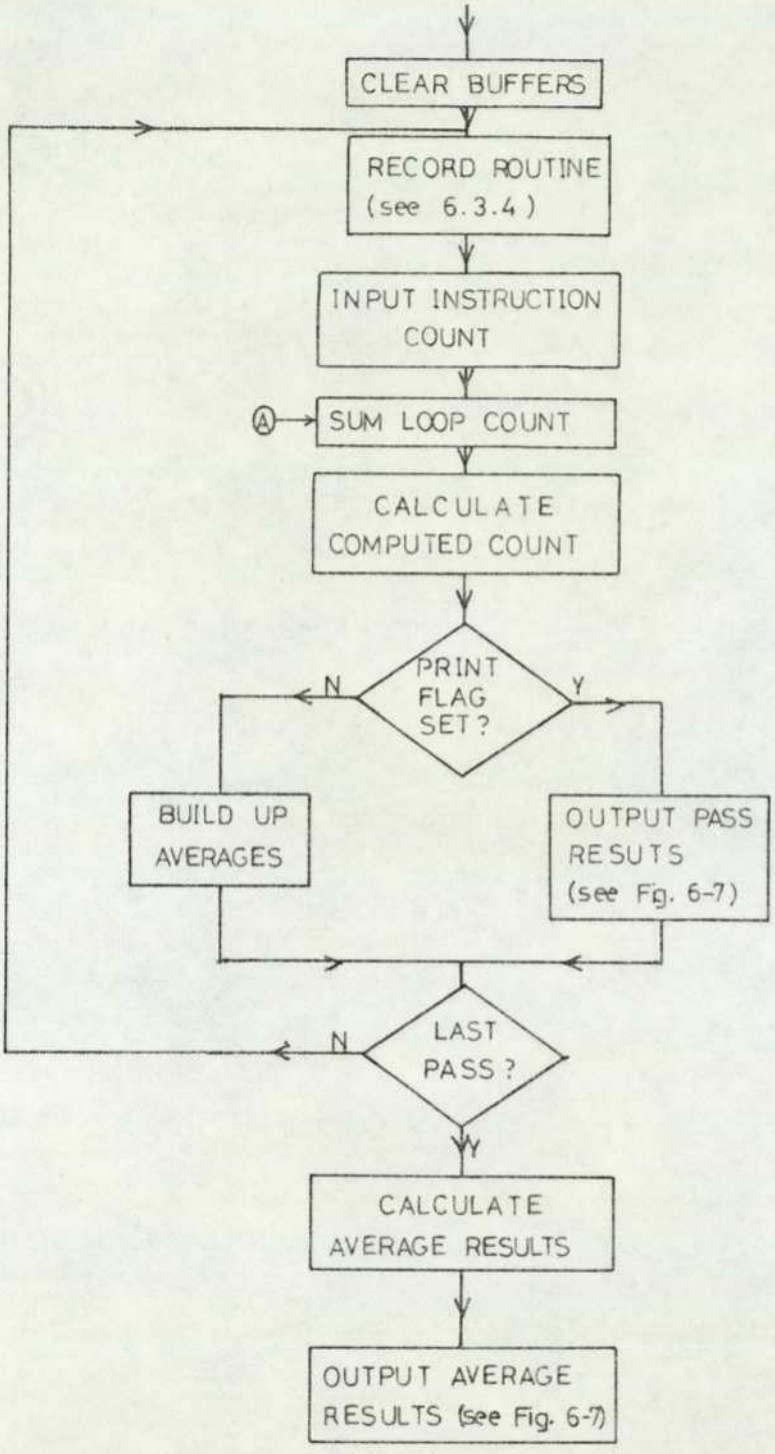


SELECT OPTIONS




PASS COUNT >1?
N
Y
OUTPUT 'PRINT PASSES?' PROMPT
INPUT CHARACTER FROM TTY
'Y'?
Y
N
'N'?
N
OUTPUT '?'
Y
CLEAR PRINT FLAG
SET PRINT FLAG
GOTO 'MONITOR'

MONITOR PASSES AND OUTPUT RESULTS



Appendix 5

Degradation analysis

Model 2

optimal point	instructions per cycle					instruction times μs		
	software contention	semaphore	remainder	total	DMA accesses	average	total	DMA cycle delay
1	2.5	9	46	57.5	15	6.5	374	153
2	1.75	8	74	83.75	13	6.9	578	133
3	1.75	8	94	103.75	13	7.1	737	133
4	1.75	8	144	153.75	13	7.3	1122	133
5	1.75	8	194	203.75	13	7.4	1508	133
6	1.75	8	244	253.75	13	7.5	1903	133
7	2.5	9	346	357.5	15	7.6	2717	153
8	2.5	9	446	457.5	15	7.6	3477	153

Model 2

optimal point	percentage degradation due to				measured degradation	percentage of total degradation due to		
	software contention	semaphores	hardware contention	total		software contention	semaphores	hardware contention
1	3.1	11.1	29.0	43.2	40.6	7.2	25.7	67.1
2	1.7	7.8	18.7	28.2	26.3	6.0	27.7	66.3
3	1.4	6.5	15.3	23.2	22.4	6.0	28.0	65.9
4	1.0	4.7	10.8	16.5	18.4	6.1	28.5	65.5
5	0.8	3.6	8.1	12.5	13.1	6.4	28.8	64.8
6	0.64	2.9	6.5	10.0	10.9	6.4	29.0	65.0
7	0.66	2.4	5.3	8.4	5.3	7.9	28.7	63.4
8	0.5	1.9	4.2	6.6	5.2	7.6	28.8	63.6

Model 3

optimal point	instructions per cycle					instruction times μs		
	software contention	semaphore	remainder	total	DMA accesses	average	total	DMA cycle delay
1	19	8	44	71	17	6.5	462	173
2	19	8	74	101	17	6.9	697	173
3	19	8	94	121	17	7.0	847	173
4	19	8	144	171	17	7.2	1231	173
5	19	8	194	221	17	7.4	1635	173
6	20	9	246	274	18	7.4	2028	184
7	19	8	344	372	17	7.5	2790	173
8	20	9	446	475	18	7.6	3610	184

Model 3

optimal point	percentage degradation due to				measured degradation	percentage of total degradation due to		
	software contention	semaphores	hardware contention	total		software contention	semaphores	hardware contention
1	19.4	8.2	27.2	54.8	59.4	35.4	15.0	49.6
2	15.1	6.0	19.9	41.0	40.8	37.8	14.6	48.5
3	13.0	5.5	17.0	35.5	34.5	36.6	15.5	47.9
4	9.7	4.1	12.3	26.1	26.3	37.2	15.7	47.1
5	7.8	3.3	9.6	20.7	13.0	37.7	15.9	46.4
6	6.7	2.6	8.3	17.6	10.9	38.1	14.8	47.1
7	4.8	2.0	5.8	12.6	5.3	38.1	15.9	46.0
8	4.0	1.8	4.8	10.6	5.2	37.7	17.0	45.3

Appendix 6 (pp. 231-243)
removed for copyright reasons
Mainwaring-Samwell, P. and Brignell, J.E. (1978)
Dual processor software for signal processing.
Paper presented at IMMM '78,
Geneva

Chapter 12

References

1. ACM (1979).
Preliminary Ada Reference Manual. Part A.
ACM SIGPLAN Notices 14 (6)

2. Aspinall, D. and Dagless, E. (1979).
Overview of a development environment.
Microprocessors and Microsystems 3 (7).

3. Baer, J. L. (1973).
A survey of some theoretical aspects of multiprocessing.
Computing Surveys 5 (1).

4. Brignell, J., Comley, R. and Young, R. (1976).
The roving slave processor.
Microprocessors 1 (2)

5. Brinch Hansen, P. (1972).
A comparison of two synchronising primitives.
Acta Informatica 1 : 190-199.

6. Brinch Hansen, P. (1973 a).
Operating system principles. (Englewood Cliffs, N. J. : Prentice-Hall).

7. Brinch Hansen, P. (1973 b).
Concurrent programming concepts.
Computing Surveys 5 (4).

8. Brinch Hansen, P. (1975).
The programming language Concurrent Pascal.
IEEE Transactions on Software Engineering 1 (2).

9. Brinch Hansen, P. (1977).
The architecture of concurrent programs.
(Englewood Cliffs, N. J. : Prentice-Hall).

10. Brinch Hansen, P. (1978 a).
Network: a multiprocessor program.
IEEE Transactions on Software Engineering 4 (3).

11. Brinch Hansen, P. (1978 b).
Distributed processes: a concurrent programming concept.
Comm. ACM 21 (11).

12. Brinch Hansen, P. (1978 c).
Reproducible testing of monitors.
Software Practice and Experience 8 : 721-729.

13. Brinch Hansen, P. (1978 d).
Multiprocessor architectures for concurrent programs.
In: ACM '78, Washington D. C., 1978. Proceedings

14. Caprani, O., Jensen, K. H. and Ougaard, U. (1977).
Microprocessors connected to a common memory.
In: Microcomputer Architectures, ed. by Nicoud, Wilmink and Zaks. (North-Holland).

15. Comley, R. (1978).
Portable computers for real time signal processing: EEG analysis as a case study.
PhD. thesis: The City University, Department of Electrical and Electronic Engineering.

16. Comley, R. (1980 a).
Dual processor monitor. Technical Document.
The City University.

17. Comley, R. (1980 b).
Private communication.

18. Dagless, E. L. (1977).
A multimicroprocessor - CYBA-M.
In: Information Processing 77, ed. by B. Gilchrist. (North-Holland).

19. Dahl, O. J., Dijkstra, E. W. and Hoare, C. A. R. (1972).
Structured programming. (London: Academic Press).

20. Dijkstra, E. W. (1965).
Solution of a problem in concurrent programming control.
Comm. ACM 8 (9).

21. Dijkstra, E. W. (1968).
Cooperating sequential processes.
In: Programming languages, ed. by F. Genuys. (New York: Academic Press).

22. Dijkstra, E. W. (1971).
Hierarchical ordering of sequential processes.
Acta Informatica 1 : 115-138.

23. Dijkstra, E. W. (1975).
Guarded commands, non-determinacy and formal derivation of programs.
Comm. ACM 18 (8).

24. Dijkstra, E. W. (1976).
A discipline of programming. (Englewood Cliffs, N. J. : Prentice-Hall).

25. Dowsing, R. (1979).
Software for CYBA-M.
Microprocessors and Microsystems 3 (7).

26. Dowson, M., Collins, B. and McBride, B. (1979).
Software strategy for multiprocessors.
Microprocessors and Microsystems 3 (6).

27. Enslow, P. H. (1977).
Multiprocessor organisation - A survey.
Computing Surveys 9 (1).

28. Ferranti Ltd. (1976).
HSM 150 F100-L Hardware and System Manual.

29. Ferranti Ltd. (1976).
PM150 F100 Software Manual.

30. Ferranti Ltd. (1978).
FC8 F100 CORAL66 Language.

31. General Instrument Corporation.(1976).
Series 1600 Microprocessor System documentation.

32. Gilbert, P. and Chandler, W. J. (1972).
Interference between communicating parallel processes.
Comm. ACM 15 (6).

33. Habermann, N. (1972).
Synchronisation of communicating processes.
Comm. ACM 15 (3).

34. H.M.S.O. (1970).
Official definition of CORAL66.
London.

35. Hoare, C. A. R. (1972).
Towards a theory of parallel programming.
In: Operating systems techniques, ed. by Hoare and Perrott. (New York: Academic Press).

36. Hoare, C. A. R. (1974).
Monitors: an operating system structuring concept.
Comm. ACM 17 (10).

37. Hoare, C. A. R. (1978).
Communicating sequential processes.
Comm. ACM 21 (8).

38. Hoener, S. and Roehder, W. (1977).
Efficiency of a multimicroprocessor system with time shared buses.
In: Microcomputer Architectures, ed. by Nicoud, Wilmink and Zaks. (North-Holland).

39. Holt, R. C. (1972).
Some deadlock properties of computer systems.
Computing Surveys 4 (3).

40. Horning, J. J. and Randell, B. (1973).
Process structuring.
Computing Surveys 5 (1).

41. Howard, J. H. (1976).
Proving monitors.
Comm. ACM 19 (5).

42. Jones, A. K. et al. (1977).
Software management of Cm* - a modular multimicroprocessor.
In: AFIPS Conference, 1977. Proceedings.

43. Lawson, H. W. and Magnhagen, B. (1975).
Advantages of structured hardware.
Paper presented at 2nd Annual Symposium on Computer Architecture, Huston, Texas, 1975.

44. Lorin, H. (1972).
Parallelism in hardware and software: real and apparent concurrency. (Prentice-Hall).

45. Mainwaring-Samwell, P. and Brignell, J. E. (1978).
Dual processor software for signal processing.
In: IMMM Conference, 2nd, Geneva 1978. Proceedings.

46. Reeves, D. W. (1980).
Quantization and other digital noise sources.
Electron No.220.

47. Schutz, H. A. (1979).
On the design of a language for programming real time concurrent processes.
IEEE Transactions on Software Engineering 5 (3).

48. Siewiorek, D. P. (1975).
Process coordination in multimicroprocessor systems.
In: Microarchitecture of computer systems, ed. by Hartenstein and Zaks. (North-Holland).

49. Swan, R. J., Fuller, S. H. and Siewiorek, D. P. (1977 a).
Cm* - a modular multimicroprocessor.
In: AFIPS Conference, 1977. Proceedings.

50. Swan, R. J. et al. (1977 b).
The implementation of the Cm* multimicroprocessor.
In: AFIPS Conference, 1977. Proceedings.

51. Tracey, R., Young, R. and Brignell, J. E. (1978).
Dual processor hardware for signal processing.
In: IMMM Conference, 2nd, Geneva 1978. Proceedings.

52. Wegner, P. (1980).
Programming with Ada: an introduction by means of graduated examples. (Englewood Cliffs, N. J.: Prentice-Hall).

53. Wettstein, H. (1977).
The implementation of synchronising operations in various environments.
Software Practice and Experience 7 : 115-126.

54. Wilkes, M. V. and Wheeler, D. J. (1979).
The Cambridge digital communication ring.
In: Local Area Communications Network Symposium, Boston, 1979. Proceedings.
U.S. National Bureau of Standards Special Publication.

55. Wirth, N. (1971).
The programming language Pascal.
Acta Informatica 1 : 25-63.

56. Wirth, N. (1977 a).
Towards a discipline of real time programming.
Comm. ACM 20 (8).

57. Wirth, N. (1977 b).
Modula: a language for modular multiprogramming.
Software Practice and Experience 7 : 3-35.

58. Young, R. (1979).
Roving slave processors for computer-aided measurement: principles and design considerations. PhD. thesis: The City University, Department of Electrical and Electronic Engineering.