ROVING SLAVE PROCESSORS FOR COMPUTER-AIDED MEASUREMENT: PRINCIPLES AND DESIGN CONSIDERATIONS

R. YOUNG

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

1979

ABSTRACT

Computer aided measurement (CAM) is a distinct branch of measurement science made possible by the fusion of discrete signal theory and real-time computing. Recent advances in these, and other, areas have increased the relevance of CAM from both the technological and economic viewpoints.

The roving slave processor (RSP) represents a novel extension of CAM which enhances the economic advantages by the use of inexpensive micro-processor systems, and removes the need for physical proximity between the CAM system and the experimental site. It thus affords the prospect of bringing real-time computing and digital signal processing to some difficult and urgent measurement tasks, which would otherwise remain neglected.

The design of an RSP is a critical balance between hardware and software realisations of measurement processes. Hardware increases speed at a cost in power consumption and reduced generality, whereas software maintains generality and conserves power at the expense of processing performance. The placing of the software-hardware boundary is thus critical. Signal processing, representing one of the more demanding CAM tasks, is used as a performance measure, and this demonstrates that many functions are still outside the capability of a practicable RSP.

The development work on the RSP, in demonstrating the need for careful optimisation of the system design, has underlined the lack of detailed knowledge of how a real-time system functions.

A basic RSP is described and demonstrated with a typical CAM application. A novel programming system is introduced, based on a high speed link to a host computer, and methods of testing, including by hierarchical connection, are suggested.

Details are given of an unorthodox programming language which has advantages of intelligibility and ease of de-assembly.

A dual processor RSP is described, and consideration is given to the difficulties of efficient synchronisation in a close-coupled system. An application of such a processor to a CAM task (pulse height analysis) is demonstrated.

The problems of efficient, programmable interfaces are discussed, and an unorthodox solution is proposed, based on a very simple, fast processor.

Finally, future developments are reviewed and the need for detailed performance analysis of the dual processor is underlined.

Benjamin Disraeli

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Factors influencing the Project

1.1.1 Availability of Semi-Conductor Technology

1.1.2 Economic and Commercial Factors

1.1.3 Division of Labour within the Project

1.2 General Points regarding Thesis Content

1.3 Layout of the Thesis

This thesis discusses techniques whereby the processing power of a programmable digital processor can be brought to bear on difficult measurement problems. It attempts to define Computer Aided Measurement as a distinct branch of measurement science, and to show how this, by the use of inherently digital methods rather than by the adaptation of conventional techniques, allows economic and flexible measurement systems to be constructed.

The realisation of CAM by a portable programmable device relying for its programming on a larger 'host' computer is discussed, and the design considerations for this device, the Roving Slave Processor, are examined in detail. The support system, both hardware and software aspects, of such an RSP are also discussed.

1.1 Factors influencing the Project

There are a number of factors which influence the form of a research/development project of this type which are not present (or are not admitted to exist) in other research work.

1.1.1 Availability of Semi-Conductor Technology

Although the need for a device of the RSP type had been seen for some time, it was the appearance of micro-processors which provided the impetus for development work. However, the rapid change in the level of processor and support-chip technology during the project (1974/78) poses severe difficulties.

Since processor construction and the writing of suitable software can take a considerable time, the engineer is bound to be many months, possibly some years, 'behind' the

current level of technology. The effects of this unavoidable lag have been mitigated in two ways: by the adoption at the project inception of two advanced 16-bit processors at a time when 8-bit devices dominated the market; and by the use of analysis and assessment of processor and system attributes rather than by a comparison of devices, which is bound soon to be outdated. The two processors, Ferranti F100L and General Instruments CP1600, have provided test beds for many ideas; and their different approaches to many problems have provided useful contrasts.

By detailed examination of the performance of processor components, the author has attempted to establish guidelines by which the suitability for RSP type applications of future processors may be assessed.

1.1.2 Economic and Commercial Factors

Since the greater part of the argument for the adoption of RSP systems is an economic one (ref. 1), the thesis considers the economic aspects of the design. In particular the author has nowhere considered either programming effort or construction effort to be free. This means that the development, particularly of the hardware, is conditioned by the availability of semi-conductor devices - a most unsatisfactory situation since no influence can be exercised over this most important aspect. It would, however, be even less satisfactory to construct an RSP of unusable proportions with no possibility of size or power reduction through the increasing use of LSI circuits, by ignoring the type of device being marketed by the semi-conductor manufacturers. The development work has thus centred around the LSI circuits that are available. Where a new device is proposed (Chapter 7), the possibility of an integrated circuit realisation is carefully considered.

As progress is conditioned by commercial developments, many of the references will be to literature released by semi-conductor manufacturers, which is necessarily not so concise or easily categorised as normal 'academic' references.

1.1.3 Division of Labour within the Project

The aspects of the RSP design/development project have been divided between two researchers in order to increase development speed. This was necessary owing to the poor level of technician support within the Department, and to attempt to keep pace with commercial developments. The division of topics was as follows:

THE PRESENT AUTHOR (described herein)

Consideration of aspects of central processor performance for RSP systems.
Development of a programming system for the RSP.
Development of programmable interfaces for use in RSP's.
Formulation of guidelines for selection of devices in RSP's.

MR. R.A. COMLEY (ref. 2)

Development on non-volatile memory.
Development of methods for linking host and slave processors.
Application of RSP to signal processing tasks (particularly bio-medical engineering).
Consideration of high level language descriptions of instrumentational processes.

Broadly speaking the present author is concerned with F100L systems, whilst CP1600 development is described in ref. 2.

1.2 General Points regarding Thesis Content

As the first thesis within the Department to concentrate on the design and performance of a measuring system rather than the physics of the measurement problem, it seeks to draw attention to the areas that need investigation and to the results that must be obtained. For this reason, the section "Suggestions for Further Work" is necessarily large and detailed.

A project of this type involves much programming and building of hardware which, while consuming time and effort, provides few results other than tedious documentation. This thesis, therefore, concentrates on the underlying conceptual aspects of the system design rather than attempting to document a lot of detailed construction work.

The ease with which processing systems can be built and programmed, together with a rapid rise in processing power, has led many designers to the view that detailed performance analysis and optimisation are irrelevant, and that the inefficiencies of the system must be tolerated as the cost of ease of use and flexibility. In a commercial environment, where the cost of skilled manpower is the limiting factor, the practice generally is to provide more processing power by the use of an inherently more powerful machine rather than by 'fine tuning' an existing system. For example, a system whose processing power is P might actually realise 0.65P by the time high level language and operating system overheads have been allowed for. If this were insufficient for an acceptable throughput for a given application, a larger machine - still working at 65% efficiency - would be employed.

This approach is acceptable as long as there are no constraints on processor power (other than cost which is balanced by the cost of improving efficiency). However,

in the case of a processor based instrumentational system, there exist absolute limits on processor power such as size, weight and, especially, power consumption. Also, whereas in a commercial environment, poor performance means slower response, and hence inconvenience or a loss of competitiveness; in a scientific sense (particularly in sampled data systems), slow response means that a particular range of problems cannot be tackled at all. Thus it becomes important to consider in detail optimisation of processor type, system design and the acceptability or otherwise of software overheads. The author, therefore, makes no apology for the range and detail of such discussions in Chapters 4 and 5.

1.3 Layout of the Thesis

Chapter 2 reviews the published work in the field of Computer Aided Measurement techniques, and defines what CAM is and how it differs from other techniques. The extension of CAM to a portable programmable system supported by a host computer - the Roving Slave Processor - is introduced in Chapter 3.

Chapter 4 sets out the physical limitations within which the RSP must be designed, and outlines the processing performance necessary in order to realise useful measurement procedures. Chapter 6 describes aspects of the first prototype RSP while the evaluation of the designs to overcome two important problems is discussed in Chapter 5. Chapter 7 attempts to assess the suitability of various types of micro-processors for use in an RSP, by considering the effect on performance of various micro-processor attributes.

The time-consuming problem of peripheral interfacing is considered in Chapter 8, and a programmable interface converter is described. Chapter 9 outlines the RSP support system used for programming and system debugging.

CHAPTER 2

REVIEW

2.1 Computer Aided Measurement

2.2 Real-Time Processor and System Design

- 2.2.1 Central Processor and Instruction Set
  - 2.2.1.1 Instruction Set Statistics
  - 2.2.1.2 Architecture of the CPU
- 2.2.2 Input-Output Systems
- 2.2.3 Device Interfacing
- 2.2.4 Dual Processor Configurations
- 2.2.5 Support Software
  - 2.2.5.1 Program Languages and Systems
  - 2.2.5.2 Testing and Debugging Aids

2.3 Conclusions

The subjects which form the background to this thesis fall into two categories: those concerned with the use of computing power in measurement systems and those which bear on the design of the real-time computing facilities themselves.

2.1 Computer Aided Measurement

Computer Aided Measurement is defined as a distinct branch of measurement science, and much of the work in field is reviewed in 'Computer-Aided Measurement', a review written by Dr. J.E. Brignell and the present author. The paper, accepted for publication in 'Journal of Physics E', also discusses the advantages and difficulties associated with the use of computers in measurement.

‘Computer-Aided Measurement', a review written by Dr. J.E. Brignell and the present author. The paper [was] accepted for publication in ‘Journal of Physics E'

This paper (pp. 15-45) has been removed for copyright reasons

2.2 Real-Time Processor and System Design

The design of a real-time system includes such aspects as central processor architecture and instruction set, input-output capabilities and software support in the form of programming languages and testing aids. All these points will not only influence the overall performance of the system, but the existence or otherwise of certain facilities will encourage or militate against the adoption of various techniques of measurement. Thus the CAM strategy adopted may depend to a great extent on the sort of computing facilities available.

Before an attempt is made to catalogue previous work in real-time system design, it is necessary to restrict the meaning of the term 'real-time'. The following description (ref. 3) was taken from a manufacturer of military computing systems:

"This (the input-output system) reflects the basic premise of a real-time system, which is primarily a system of response, in which the traditional 'master/slave' relationship of the computer to the peripherals . . . is reversed, the computer becoming subservient to the peripherals".

Furthermore it is inherent in the term real-time as used in this thesis, that if responses are not made within a certain time, a fundamental breakdown in the system will occur. In commercial real-time systems this is not the case, since a delayed response usually means no more than an increased cost. Where breakdown does occur, the delay will have been long, minutes or tens of minutes, rather than milliseconds or tens of milliseconds. Thus much of the commercial work on real-time systems is not relevant to this work, whereas the experience of military computing is much nearer to the type of work reported here.

Chapin (ref. 4) underlines the differences between commercial and military systems, and makes some points about instruction types which tend to be used. Among the characteristics of military programs Chapin lists:

1. Inputs often cyclic with new data replacing old.

2. Input-output usually packed with variable length fields - character strings uncommon.

3. Many high data rate input-outputs require precise timing.

4. Many packed tables of multiple variable length fields require 'bit fiddling' (masking and shifting) to pack and unpack.

5. Use of logically complex algorithms for processing.

He also states that multiplication and division, as well as floating point operations, are not commonly found. With the exception of this last statement, the military experience accords well with that in a CAM laboratory.

2.2.1 Central Processor and Instruction Set Design

The starting point for any design decision is a reasonably concrete idea of the task to be performed. In this context, knowledge of the occurrence of instruction and operand types is necessary.

2.2.1.1 Instruction Execution Statistics

Much work on instruction occurrence has been concerned with measurement of the incidence of high level language elements (e.g. Knuth, ref. 5) rather than the machine code itself. The machine code utilisation of large machines operating a variety of high level languages has been recorded. Barak & Aharani (ref. 6) collected statistics on a CDC CYBER 74 running a mixed batch of mostly (87%) FORTRAN programs with

some PASCAL, while Wirth (ref. 7) obtained similar figures on a CDC 6000 for PASCAL programs. These two reports confirm the relative unimportance of arithmetic operations (10.3% Barak, 3.5% Wirth) compared with fetch and store functions (36.2% Barak, 27.6% Wirth). Neither report distinguishes application programs from the operating system, or suggests an overall percentage for the number of operating system instructions. Sumner (ref. 8) examined machine code on a large machine in more detail and produced useful information about jump lengths and amount of code executed between jumps.

2.2.1.2 Architecture of CPU

Much of the work on the evaluation of CPU architectures has been directed either towards assessing a CPU structure for a particular high level language, or has included other performance aspects so that a computer system as a whole is evaluated. The inclusion in such work of operating system facilities (e.g. disc file handling, multi-tasking, etc.) makes it inappropriate to consideration of the RSP design.

Tannenbaum (ref. 9), by consideration of a structured language, found that the requirements of efficient operation and straightforward compilation could be met by a simple stack-orientated architecture. He claimed that such a design would be two to three times as efficient (in terms of machine code storage space) as a conventional multi-register CPU. Wichmann (ref. 10) examined in detail how a variety of machines dealt with ALGOL 60 compilation. His findings agree with Tanenbaum's claim that modern multi-register machines cannot be efficiently exploited by high level language compilers. Wichmann also demonstrated a useful method of estimating high level language execution times.

Fuller & Bar (ref. 11) describe the assessment of a number of processors for a military environment. Although the processors involved are large and powerful, many of the criteria on which the assessments are based are relevant to

RSP work. For example, two of the absolute criteria are: floating point support and multi-processor operation (explicit provision of test and set instructions). Other workers (Barbacci & Siewiorek, ref. 12) attempting to verify these evaluations by using a formalised description of architecture based on the ISP system of Bell & Newell (ref. 13 and 14), found the results very dependent on the chosen style of ISP representation.

2.2.2 Input-Output Systems

The importance of the I-O system to a real-time computer is paramount. Considering this fact, remarkably little is recorded on the subject.

Most work concentrates on large multiprogrammed computer systems, and is concerned with queuing and overlap (e.g. ref. 15) of the I-O operations and processing. Lorin (ref. 16) is a valuable source of information about I-O systems of larger machines.

Thurber et al (ref. 17) provide an exhaustive review of I-O bus structures including daisy-chaining and handshaking operations. A detailed description of an input-output controller for a military real-time computer (ref. 18) reveals the amount of I-O functions that can be realised without processor intervention. It also highlights the importance of peripheral autonomy in real-time systems. Washburn (ref. 19) describes the use of a simple sequencer on an I-O channel to give a degree of programmability to the device interface (see 2.2.3).

2.2.3 Device Interfacing

The difficulties of interfacing devices to micro-processor systems mean that this activity can take a disproportionate amount of development time. In a research development environment this may mean that new devices are not tried, so clearly the problems of interfacing must be mitigated if the claim of the RSP to be a universal research tool is to be upheld.

Most commercial work in this area has centred on the use of simple hardware with all control done in software or the use of special purpose chips to interface specific devices (ref. 20). The first method has severe speed limitations and both concentrate on program controlled transfers, which are often inherently slow and wasteful of processor effort. Attempts to improve this situation have taken the form of general purpose interface sets which provide a wide range of facilities (ref. 21), or special peripheral processors (ref. 22) which enable special interfacing functions to be delegated to the peripheral processor. The peripheral processor is a powerful concept, but again suffers a speed disadvantage due to slow (4 μS per instruction) operation. General purpose interface sets require much extra logic before a complete device interface can be realised. The idea of replacing this logic with a programmable 'state machine' leads to consideration of the structure of such machines. Simple state machines can be based on ROM's with feedback latches (ref. 23), but, although some of the size problems can be averted (ref. 24), this method is severely limited with regard to the number of input variables. Programmable logic arrays can also be used in this way and these overcome many of the problems of the ROM. Counters (ref. 25) can also be used but speed limitations due to redundant states are a drawback. Some novel approaches to the realisation of simple processors (e.g. Sweet, ref. 26) are also too slow.

The recent appearance of fast programmable logic arrays with built-in feedback paths (ref. 27) provide an ideal means of realising an interface controller.

An alternative approach to device interfacing has been the adoption of interfacing standards. The suitability of these to RSP device interfaces is restricted. The IEEE 448 Instrument Bus (ref. 28) was designed for calculator controlled instrumentation based on BCD or character information transfer over an 8-bit bus. The bus speed is limited to that of the slowest device and the control

system is inflexible. CAMAC (ref. 29 and 30) is better suited to large scale data acquisition and requires a wide (56-wire) bus for connection. It is not inconceivable that the RSP would have such an interface, but this would complement rather than replace the ordinary interfaces between the processor and its analogue I-O devices.

2.2.4 Dual Processor System Design

Much work has been done on multi-processor system design particularly with the advent of the micro-processor. However, most (e.g. ref. 31) has concentrated on the design of processor/memory modules which can be configured with many such units in a general, loosely coupled, micro-processor system. Again Lorin (see 2.2.2), in his analysis of close-coupled dual processor systems for larger machines, describes many techniques relevant to the dual processor RSP.

Searle and Frebert (ref. 32) review a number of basic configurations for multi-processor systems and point out some advantages and disadvantages. Of particular interest is their review of systems using multi-port store blocks, though little work on such stores is to be found (e.g. Ferranti FM1600 Series Stores, ref. 33). Kinnie and Maerz (ref. 34) describe a special dual-port RAM designed primarily to economise on bus use in a dual processor shared-bus system.

2.2.5 Software Support

The software part of the RSP system comprises three distinct elements: a language and programming system for the RSP, a suite of application programs for standard CAM functions, and testing software for the evaluation and proving of complete CAM systems. Since the second of these elements is dealt with largely by a colleague (ref. 2), this thesis is primarily concerned with languages, compilers/assemblers

and testing aids. Testing of real-time systems is so important and potentially difficult that the author makes no apology for regarding it as an aspect in its own right, rather than as an adjunct of the programming system.

2.2.5.1 Programming Languages and Systems

All micro-processor manufacturers offer an assembly language with their devices. These are usually (e.g. ref. 35) of the literal mnemonic type with the function names chosen for brevity rather than clarity. Such languages have attracted little attention although some moves to standardise mnemonics (ref. 36) have been made. The introduction of high level languages for micro-processors has taken the form of implementing versions of standard languages such as PL 1 (e.g. PLM for Intel 8080, ref. 37) or FORTRAN (ref. 38). Manufacturers have not been rigorous about using sub-sets of official languages, so many apparently 'standard' languages (e.f. ref. 39) have small differences when implemented on micro-processors. BASIC has been widely used because of the ease of writing an interpreter, but such a language is inherently unsuitable for real-time work. Forth (ref. 40) attempted to combine elements of interpretative and conventional languages and has made considerable inroads, particularly in process control. Some potentially interesting languages (e.g. RTL/2, ref. 41) do not have available compilers.

All these high level languages suffer from the same defects regarding efficiency of code generation. This means that programs are longer and, more importantly, slower than their assembly language counterparts. To alleviate these problems, several attempts have been made to introduce the so-called high level assemblers. These are machine dependent assemblers which incorporate many of the high level language constructs which simplify programming. The original work by Wirth (ref. 42) on a large machine prompted similar ventures on smaller ones (e.g. Bell & Wichmann, ref. 43). A simpler version is used on the GEC 4080 ('Babbage', ref. 44) for real-time process control. More recently, work on such languages for micro-processors has been reported (ref. 45 and 46).

The inability of high level languages to deal with intimate hardware interaction has recently been mitigated by the appearance of a language with specific facilities for synchronisation of multi-processor systems. PASCAL and concurrent PASCAL (ref. 47) are therefore applicable to the dual processor, and the compiling system, based on an intermediate 'P' language, is particularly easy to implement as only a 'P' language interpreter needs to be written. Berry (ref. 48) suggests that the PASCAL programs execute in the same time as an equivalent compiled ALGOL program when running on a 16-bit processor (D.G. Nova).

2.2.5.2 Testing and Debugging Aids

The sort of debugging aids provided for real-time systems are usually of the form of an interactive program which allows the user to monitor registers and insert breakpoints in the compiled code (e.g. PDP11 ODT, ref. 49). Some earlier computers incorporated breakpoint registers (e.g. IBM 650) so that a particular location or register value could initiate a transfer to a monitoring program. However these suffer a serious disadvantage: they stop program execution so in a real-time system the sequence of inputs to the computer will be lost. Program tracing must, therefore, be done while the processor is operating at full speed which implies the use of special purpose hardware. Early examples of such devices (ref. 50) were bulky but allowed brief segments of bus or store 'history' to be recorded for later analysis. The technique developed into the logic state analyser (ref. 51) which can be connected to the data/address bus of a micro-processor. It allows a trace of bus and register contents to be initiated by a selected sequence of events so that program errors (and their precursors) can be examined without interfering with program execution.

The alternative is the In-Circuit Emulator (ICE) (e.g. ref. 52) where a micro-processor in a system is emulated by a device which has debugging and tracing facilities built in. This allows powerful debugging software and hardware to be connected straight into the system being developed in place of the normal micro-processor. This is a powerful technique although it does not generally have the real-time capabilities of the logic state analyser.

2.3 Conclusions

In drawing up this review of the published work, the author has been struck by two points: firstly, the extremely large quantity of publications on the subject of hardware and software aspects of system design, and, secondly, the tendency of such work to be concentrated in certain areas, to the virtual exclusion of others. One would expect that the neglected areas are either of no importance or are sufficiently well understood not to warrant any more investigation: the author has been unable to establish that this is so.

Most of the work on computer performance evaluation has taken a 'macroscopic' approach to systems and attempted to produce overall figures for complete hardware/software system performance. Very little work is reported on attempts to isolate the elements which combine to make those figures. This makes the tracing of characteristics to individual design decisions, speculative.

Direct evaluation of CPU architectures and instruction sets is rarely found in recent literature. It is difficult to justify this neglect since, in the last analysis, the machine code constitutes the 'work load' to which the processor must be suited. Similarly assembly languages are neglected in favour of work on high level languages, despite the continuing importance of lower level programming aids, particularly in real-time systems. Input-output systems suffer a similar lack of detailed investigation.

One suspects that processor manufacturers do have such specific data in order to design their products, but they are reluctant to publish. Likewise the military experience in real-time computing would be of interest if it were available.

The overall picture is one in which the theoretical work is well ahead of practical reality, and many pressing practical problems do not attract the sort of detailed investigation they warrant. Instead much academic work is concentrated on the more esoteric aspects of computers with little thought given to the cost of the abstract approaches used.

It is therefore necessary, in trying to establish design criteria for real-time systems such as the RSP, to do a lot of investigation in areas which are not now regarded as fruitful sources of research topics.

CHAPTER 3

CONCEPT OF THE ROVING SLAVE PROCESSOR

3.1 Introduction

This chapter seeks to explain how the idea of a slave processor grew out of the conventional CAM system. The extension of this idea to the Roving Slave Processor (RSP) is shown to be a simple step but one offering considerable advantages. The aspects of CAM systems in general, and the RSP in particular, that must be investigated, are discussed.

3.2 The Operation of A Typical CAM System

Fig. 31 shows, in block diagram form, the CAM arrangements used in the author's laboratory for the study of charge transport in liquid dielectrics. In this case the interfaces between processor and experiment consist of a high-voltage programmable power supply to cause near-breakdown stresses in the dielectric, and charge amplifiers and photomultipliers to indicate, via analogue-digital converters, charge transfers and light emissions within the test cell. Although this is obviously a highly-specialised application, the peripherals are basically common to all CAM operations in that they provide the means to apply stimuli and monitor responses. As nearly all physical phenomena are time-dependent, an accurate real-time clock is incorporated within the system.

The sequence of operations during a CAM exercise fall into three distinct parts which may be termed preparation, on-line and post mortem. The activities of these phases, and their characteristics, are listed below:

1. Preparation

Activities	Program writing, assembly, compilation, editing and correction, simulation, etc.
Characteristics	Low CPU utilisation (could be time-shared), high demand for peripherals (e.g. printers, disc backing store, etc.), no time criticality.

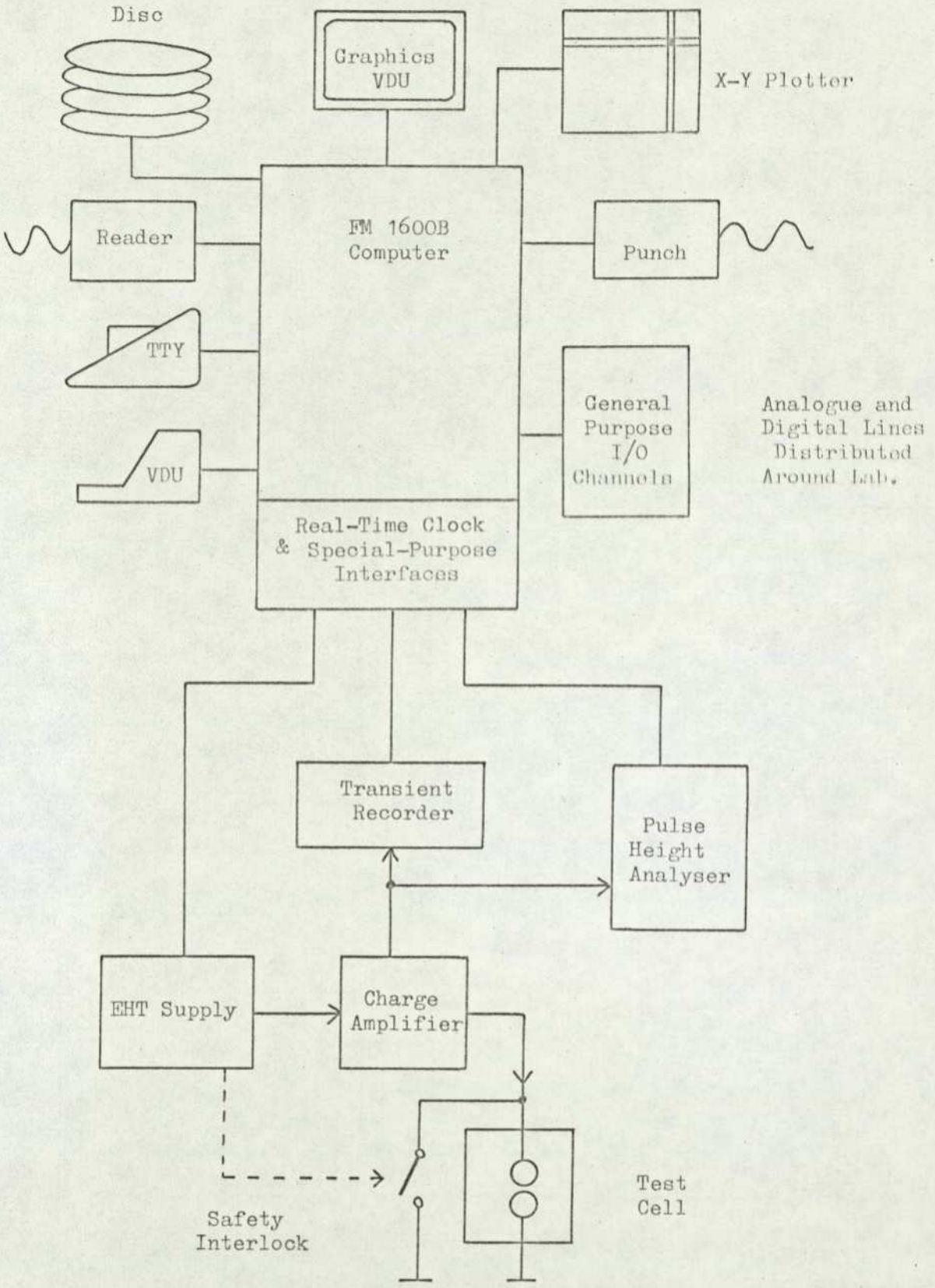


Fig. 31. The CAM Laboratory used for Dielectric Studies.

2. On-line

Activities	Experimental operation, control of stimuli to test-object, monitoring and recording of responses, detection of unsafe conditions.
Characteristics	High CPU demand, time-critical operations, no use of standard peripherals, use of 'signal' peripherals (e.g. A-D, D-A converters).

3. Post Mortem

Activities	Display and analysis of experimental results, calculation of secondary results.
Characteristics	Low CPU demand, no time criticality, use of standard peripherals for priority or plotting results.

It can be seen that phases 1 and 3 have much in common in their demands for CPU and peripheral resources, whereas the needs for high CPU utilisation and 'signal' peripherals are peculiar to phase 2.

3.2.1 A Modified System Using A Slave Processor

This division suggests that while phases 1 and 3 can be carried out on any normal computer system (single user, multi-access or even 'batch'), phase 2 may best be executed on an entirely different machine. This phase calls for the complete dedication of CPU time during the experiment and for the attachment of a variety of 'signal' peripherals. If phases 1 and 3 are run on any interactive type of system, then a hierarchical arrangement may be used as in fig. 32.

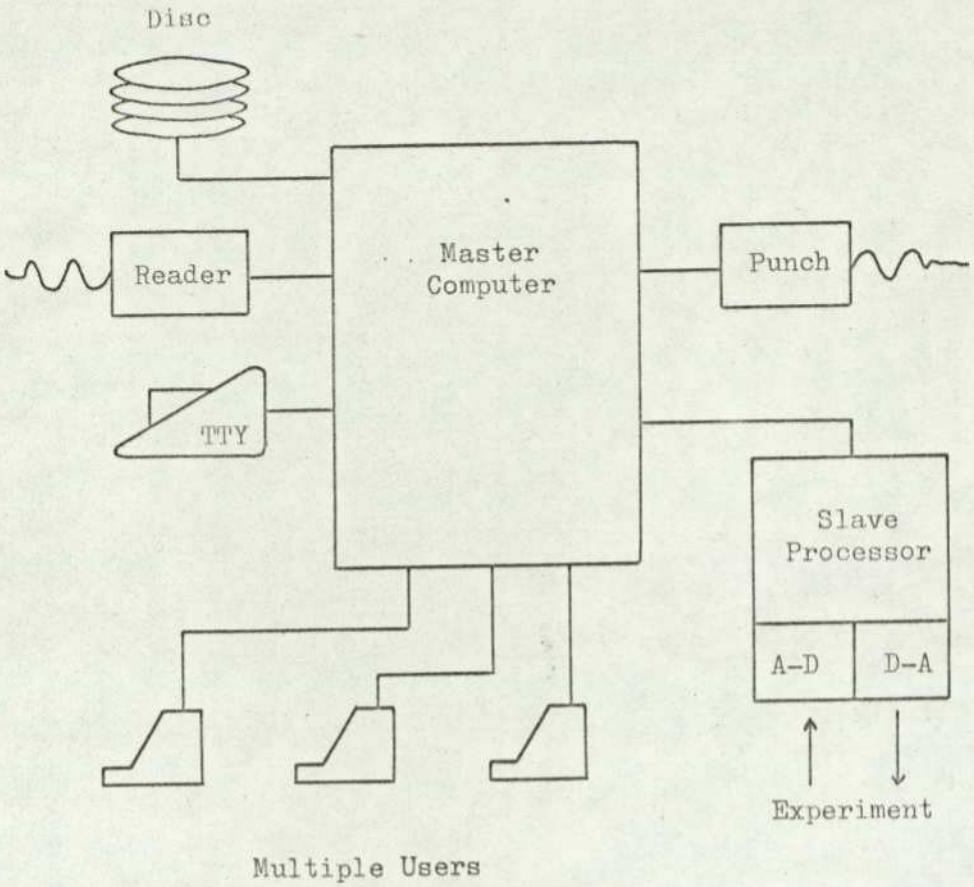


Fig. 32. Hierarchical Arrangement with 'Slave' Processor.

Thus the master computer is freed during phase 2, which may need to be an extremely long time, and can operate in a normal multi-user mode.

A further advantage accrues from this arrangement: none of the specialised peripherals used in phase 2 need be interfaced to the main computer. This is important, since the input-output channels that exist on a general purpose computer are likely to be slow and inflexible, being designed for teletypes or printers, and the construction of special high throughput channels may require the modification of the operating system. The only new interface to the master computer is that joining it to the slave processor. This can be an ordinary serial character-based link with no stringent time limitations, although as is shown later (Chapter 9) there can be advantages in the use of a high speed bi-directional link.

3.2.2 The Roving Slave Processor

It is now a small step to provide the slave processor with a non-volatile memory and make the master-slave link breakable. Since this link is used only at the beginning and end of phase 2, there is no need for it during the experimental 'run'. There is now no requirement for the test-object to be located close to the master computer, because the slave processor can be removed to a remote experimental site. Thus the concept of the Roving Slave Processor, as it has been termed, embodies not only the economic advantages associated with freeing the master computer and avoiding special interfaces, but also the technical advantage of being able to take computing power to a test-object instead of vice versa. Fig. 33 shows an RSP system. The slave processor is realised as a portable micro-processor system with a semi-conductor memory and battery back-up to provide non-volatility.

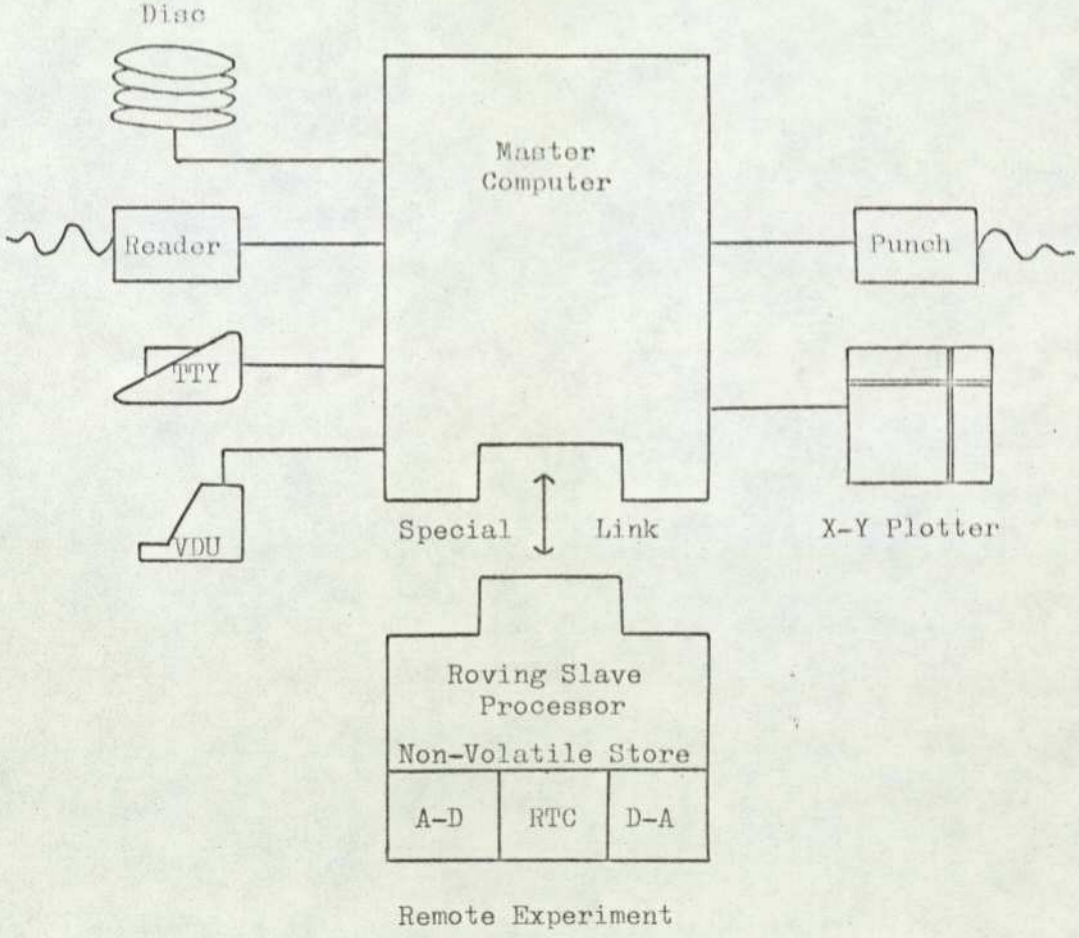


Fig. 33. A Roving Slave Processor System.

The micro-processor has no bulky standard peripherals and so can be a truly portable device. It is plugged into the master computer and 'charged' with its program and data, then removed to the experimental site. It controls the experiment while the master computer is free for other tasks, and is then taken back to the master computer where it 'dumps' its results. Phase 2 might include a procedure for testing the operating of the slave's program by exercising it with the master computer monitoring its performance. This technique (9.5.2) is used in the author's laboratory to provide the micro-processor with a testing facility.

3.3 Lines of Investigation

The lines of investigation which must be followed in order to bring about the successful development of the RSP and its support system, fall into two categories: those concerned with CAM systems in general and those which bear on the problems of the RSP design.

3.3.1 Problems of CAM Systems

CAM systems seek to replace instruments with instrumentational processes defined in software, operating on standard hardware. This poses two important questions: the best way of providing a high level description of the processes required, and the extent to which the hardware design can be optimised for these processes without losing its generality. There is also the problem of device interfacing which can take a disproportionate amount of development time.

3.3.1.1 High Level Process Description

It is clearly desirable to be able to specify a complete process by linking modules of software which each represent a distinct operation, e.g. input block, output block, filter, extract RMS value, smooth, etc. Description at such a high level obviously implies that many options and parameters must be specified, either by the user to obtain the process he requires, or by the linking program to maintain consistency throughout the chain of processes. The linking program must perform some error checking between modules, and also has the task of dividing operations between two processors in a dual processor system. Automatic synchronisation of the processors must also be achieved.

3.3.1.2 Optimising the Hardware

The great majority of instrumentational tasks are real-time, which is interpreted in this thesis as time-critical. This means that the programs which represent the processes must execute as quickly as possible. The crude approach to speeding up running time, by adopting a more powerful processor and faster memory, is of limited value; since, in an RSP context, size, power consumption and the presently available level of technology, set a limit to processing speed. Hence the processor type and the system architecture must be carefully optimised with proper regard to the type of processing involved in CAM. A great difficulty is in setting the border-line between hardware and software, i.e. between speed on the one hand and versatility on the other. Chapter 4 considers this point and the fundamental constraints on an RSP design, while Chapter 7 reviews various processor characteristics with particular reference to real-time systems. Chapter 6 demonstrates how two specific performance bottlenecks can be removed with the aid of additional hardware, without spoiling the generality of the system.

3.3.1.3 Easing the Interface Design Problems

The task of interfacing devices and instruments to a processor system is not difficult, but it is time-consuming. This implies that during a program of development, where time is at a premium, it is often impossible to investigate alternatives to tried devices. In order to overcome this limitation, programmable interfaces have been investigated (Chapter 8) and a novel approach suggested.

3.3.2 RSP Design Problems

The main problems specifically related to the RSP system are the difficulty of getting enough processing power and memory into a portable (roving) equipment, and the choice of programming system for the RSP itself.

3.3.2.1 Size and Weight Problems for an RSP

The portability requirements of the RSP pose severe problems even with the present generation of processing and storage devices. Power consumption is by far the greatest problem, both because of the bulk of suitable power supplies when the equipment is operating, and the weight of batteries needed to maintain memory retention during transportation. Reducing power consumption and physical size both lead to reductions in processing performance. Thus these physical limitations influence the decisions as to where to draw the hardware/software boundary, as mentioned earlier (in general a software solution uses limited hardware very effectively thereby realising a complex process on smaller hardware, whereas the hardware solution gains high operating speed by the use of dedicated but less highly utilised circuitry). Chapter 4 discusses these constraints and comes to some quantitative and qualitative conclusions about processor performance for an RSP.

3.3.2.2 RSP Programming System

The choice of programming system for an RSP involves a number of questions as to the type of assembly system (cross-assembler or self-assembler), applicability or otherwise of high level languages, usefulness of simulation programs and choice of debugging systems. The last point is crucial, since fault-finding in real-time programs demands more powerful tools than the normal trace and breakpoint routines. The added difficulties imposed by hardware/software interaction must also be catered for. Chapter 9 discusses support systems, and describes a standard cross-software arrangement and the novel RSP host/slave system.

3.4 Summary

Much of the foregoing is summarised in the article 'The Roving Slave Processor' published in 'Microprocessors', Vol. 1, No. 2, December 1976.

'The Roving Slave Processor' published in 'Microprocessors', Vol. 1, No. 2, December 1976

This article (pp. 68-71) has been removed for copyright reasons

CHAPTER 4

DESIGN CONSTRAINTS ON AN RSP SYSTEM

4.1 Introduction

The constraints which confront the designer of an RSP system fall into two categories: physical limitations of size, weight, power consumption, etc., and performance limitations of processing speed, accuracy, etc.

4.2 Physical Limitations

Since the RSP is a portable device, its usefulness is dependent on its bulk. While in general a size equivalent to that of a portable oscilloscope and a weight of 5 - 10 Kg poses no problems, there are some potential applications where size and weight are extremely critical. The permanent attachment of an RSP to a patient for ECG or EEG monitoring is an obvious case, and one which is well outside the presently-available technology. It serves, however, as a useful target for this development programme.

4.2.1 Size and Weight

The size and weight of a piece of electronic equipment depends not so much on the number of its components as on secondary factors, such as power supply bulk and the need or otherwise for forced air cooling devices.

Switching mode power supplies have increased the power supplied per unit volume from about 8 Kw/cubic metre to 70 - 100 Kw/cubic metre, while the power per unit weight has increased from 5 - 7 W/Kg to 80 - 120 W/Kg. Furthermore, since they are much more efficient than continuously regulated power supply units (PSU's), they dissipate less power which has to be removed as heat.

Comparison of two 5A 5V power supplies:

	CONTINUOUS	SWITCHING
Volume	0.00035 m	0.000044 m
Weight	4.8 Kg	0.5 Kg
Efficiency	40%	70%
Power Dissipated	37 W	10.5 W

It can be seen that for all cases where the power consumption is greater than a few watts, true portability is only possible by use of the switching mode PSU. For present technology, a signal processing capability implies such a consumption.

Since a non-volatile memory is inherent in the design of an RSP (so that information entered via the host computer is not lost as the RSP is transferred to an experimental site), it is relevant to assess the size and weight penalty paid for non-volatility. A detailed analysis of storage methods suitable for an RSP will be found elsewhere (ref. 2), but in general the conclusion is that, of the inherently non-volatile media, core stores require too much power when operating due to the high drive currents required for fast core switching, while magnetic bubbles are too slow due to their serial nature. The latter were not an available and proven technology anyway.

Thus one is left with the conclusion that at least part of the RSP's semi-conductor memory must be capable of retaining information by means of back-up batteries. Thus the specific power of the battery is also a limiting factor in RSP design.

4.2.2 Power Consumption

The following quantities must be estimated before the design of the system can begin:

1. The operating power consumption.
2. The standby (memory information retention) power.
3. The length of time standby must be maintained.
4. Whether battery operation is to be allowed, and if so, for how long.
5. The maximum permissible temperature rise within the cabinet.

Thus it can be seen that power consumption of memory and processor circuits is crucially important to the successful design of a compact RSP.

There are only two ways of cutting down the power consumption of a digital system: by the use of fewer gates or by the use of gates whose individual power requirements are less. These both amount to the same thing - cutting down system speed.

4.2.2.1 Reducing the Gate Count

If one assumes that the processing system has a task, or range of tasks, of a given complexity, then reducing the number of gates requires that more use be made of the remaining devices. That is to say that what was realised as an array of logic elements must now be carried out in a sequential manner on one or two devices. This transfer from parallel to serial processing inevitably slows down the response of the system.

The most obvious example of this in micro-processor systems is the difference in approach to I-O systems. At one extreme the simplest data path is provided and all I-O handling is performed in software (Chapter 8), while the other extreme involves the use of specialised hardware so that the interfacing functions can be realised quickly. The first arrangement makes greater use of the processor gates and keeps the hardware at a minimum but at a greater cost in performance. The second design achieves maximum performance but at a considerable cost in increased hardware and hence power consumption. A similar situation arises over the question of hardware or software multiplications.

Hence the goals of reducing the gate count and increasing the system performance are seen to be fundamentally incompatible.

4.2.2.2 Reducing the Individual Gate Power

Gate power and propagation delay are interrelated so that the lower the power, the longer the delay. Nevertheless, as fig. 41 shows, some logic families approach the ideal (high speed, low power) point 'A' more closely than others. In all cases the internal gate performance (e.g. in MSI circuits) is better than the individually packaged gate. CMOS is represented by a line because its power consumption depends entirely on the frequency of operation as it dissipates virtually no power in either state. This low power standby capability of CMOS is attractive where power consumption is important, but the speed penalty is large. Also the packing density is poor, e.g. in RAM's the number of cells per chip is a quarter of that obtainable by NMOS. So reducing power consumption by the choice of a different logic family leads to a slower system, and, in the case of CMOS, many more chips.

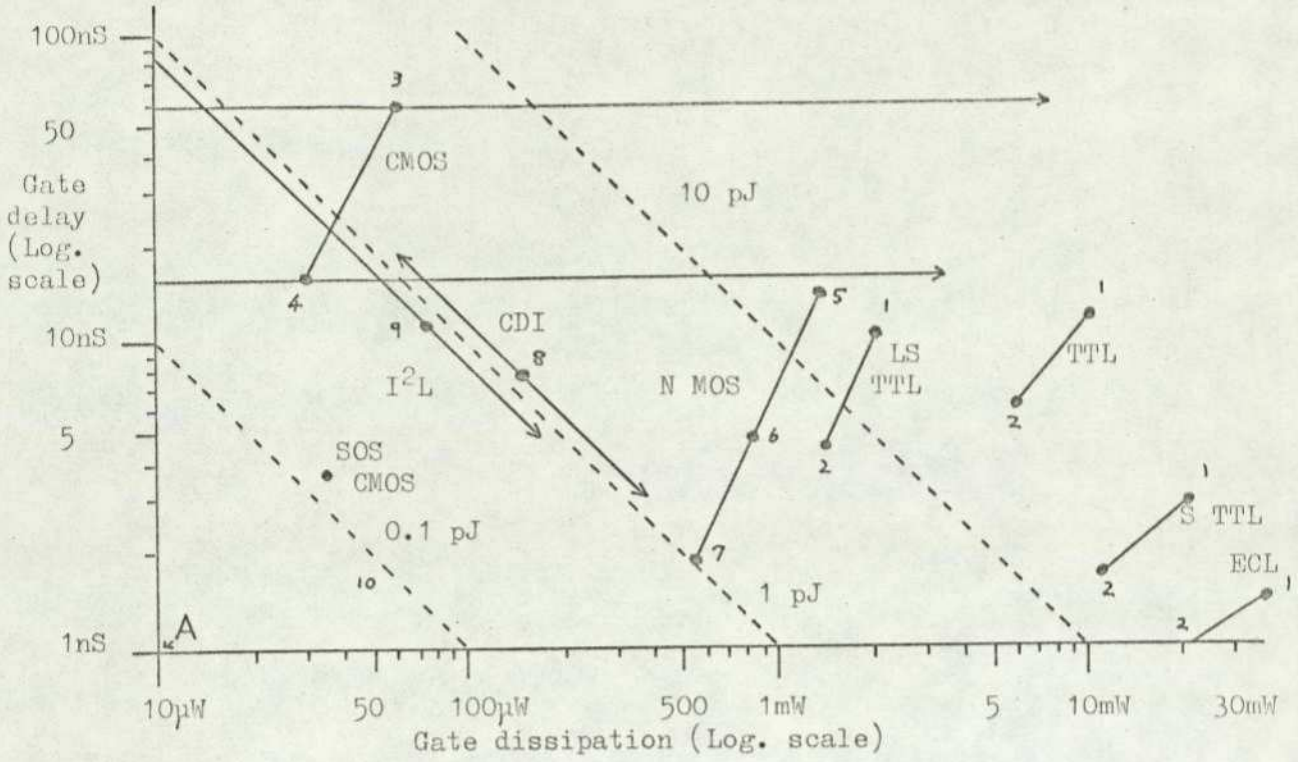


Fig. 41. The Relationship between Gate Delay and Gate Dissipation for Various Logic Families.

Notes :

1. Performance at single-gate level i.e. SSI
2. Performance in large logic arrays i.e. MSI
3&4. CMOS calculated for single gate transition. Lines represent actual power consumption of gate depending on frequency of input transitions.
5. NMOS enhancement-mode device - LSI 1972
6. NMOS depletion-mode device - LSI 1976
7. NMOS scaled-down 'HMOS' process - LSI 1978
8. F 100 CDI gates - variable speed, fixed speed-power product.
9. I^2L gates - ditto
10. Dotted lines represent constant speed-power product lines i.e. product of gate delay and gate dissipation.

4.3 Processing Performance Limitations

In order to set performance limits for an RSP, one must postulate a problem which the RSP should be capable of solving. Since the area of applicability is to be as large as possible, the choice of this 'test' problem is an important step if valid results are to be obtained. From a wide range of computer aided measurement problems to which larger real-time computers have been applied, it appears that real-time signal processing presents the most stringent demands in terms of processing speed and accuracy. Also, since a figure for the maximum bandwidth can be obtained for each system, the relationship between internal processing speed and response speed measured from the outside of the system, can clearly be seen. Therefore, this section will use a real-time signal processing operation as a 'test task' to evaluate the sort of performance an RSP must have in order to be useful.

4.3.1 Processor Word Length

The word length of a processor is the number of bits which form the fundamental unit of data within the machine. This has two independent effects on performance: the length of the instruction word affects processing speed by allowing a wider range of operations to be defined by a single word, while the length of the data word affects the accuracy with which the data can be resolved. There is no necessity for these two types of word to be of the same length, but nearly all processors adopt the convention of using the same word-length for instructions and data so that a homogeneous store can be used with no hardwired division between word types. Since instruction wordlength effects instruction speed, it will be considered in 4.3.2 while this section confines itself to consideration of data accuracy.

4.3.1.1 Fixed Point Systems

In many signal processing operations two sorts of accuracy are involved: coefficient accuracy (i.e. concerned with the coefficient in the processing equations) and data accuracy.

For example, in digital filters the coefficients specify where, in the z plane, the poles and zeroes are located. In high quality filters it is likely that poles will be located very close to the limits of stability (the unit circle in z plane - the $j\omega$ axis in the s plane). Any inaccuracy in their location could effectively cause them to cross the line into the unstable zone (ref. 53). Thus coefficient accuracy limits the maximum quality factor of a digital filter.

FIXED POINT REPRESENTATION

Number of Bits	Accuracy at 'Full Scale' 1 Part In	Approx. Number of Decimal Digits
8	256	2
10	10^3	3
12	4×10^3	3
16	6×10^4	4
24	10^7	7
32	4×10^9	9
40	10^{12}	12

FLOATING POINT REPRESENTATION

Total Number of Bits	Mantissa Bits	Exponent Bits	Full Scale Accuracy 1 Part In	Range
12	8	4	256	4×10^{-3} to 128
16	10	6	10^3	10^{-10} to 10^9
20	14	6	10^4	ditto
24	18	6	2×10^5	ditto
30	22	8	4×10^6	2×10^{-10} to 2×10^{10}
32	24	8	10^7	ditto

Table 41

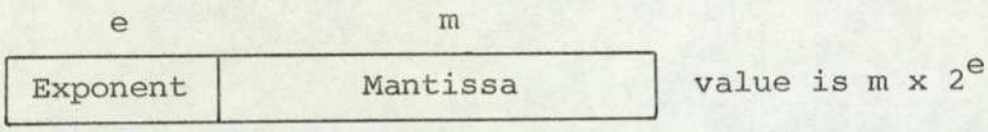
Range and Accuracy of Fixed and Floating Point Representations

Inaccuracies in the data representation have the effect of adding noise to the signal in the same way as quantisation (by the A-D converter at the input), which, although strictly a non-linearity, may be considered as added noise. Thus the accuracy of data representation sets the noise performance of a digital filter.

So far only the accuracy of the representation has been considered. The other important criterion is the range of data magnitudes that can be represented. Clearly, for a fixed point representation, the range is the reciprocal of accuracy. However, the accuracy (as quoted in table 41) is the accuracy at 'full scale' only, so that for a 16-bit word, for example, if a range of 10^3 was required, then the accuracy that could be maintained across any part of that range would be one part in 64. It can be seen that a fixed point representation cannot provide sufficient range and accuracy unless the number of bits used is large (>36).

4.3.1.2 Floating Point System

Floating point representation improves this by maintaining a fixed precision across a large range.



The table shows ranges and accuracies for various numbers of bits. Since one part in 10^4 ('4 decimal places') may represent a minimum acceptable accuracy for most signal processing applications, a mantissa of at least fourteen bits is required.

For this to be usable, it must have a range of about 10^3 so the exponent has to be five or six bits. This gives the minimum number of bits in which a useful floating point representation can be achieved as about twenty. In practice twenty-four is taken as the usable minimum. This occurs in 24-bit machines (e.g. FM1600B) which have a single-length

floating point facility, or as triple-length operations in 8-bit machines. 16-bit processors generally use two words with a mantissa of twenty-four bits and an exponent of eight bits. This division of bits may not be ideal, but it is generally easier to have the divide on half-word boundaries rather than at some arbitrary number of bits.

4.3.1.3 Multi-length Working

Achieving effective lengthening of words by multi-length working would be an attractive proposition but for the speed degradation caused. Adding two 16-bit words on an 8-bit processor must take at least three operations (add least significant words, add in carry to a most significant word, add most significant words) compared with the one operation for a 16-bit processor. Also twice as many data fetch cycles will be needed to fetch the data from store. Thus the total number of memory cycles required is:

(A = B + C A, B and C are 16-bit numbers)

8-bit Processor	16-bit Processor
3 instruction fetch	1 instruction fetch
4 data fetch	2 data fetch
2 data restore	1 data store
9 Total Store References	4 Total Store References.

4.3.2 Instruction Repertoire and Speed

The instruction set of a processor to be used in an RSP must be capable of performing the processing operation within a given time. Whether this is achieved by a few relatively slow, powerful instructions or by a large number of relatively simple operations, is not of primary importance. That is to say the two instruction set properties - speed and repertoire - trade off against one another.

4.3.2.1 Repertoire

Obviously all normal logical and arithmetic operations are required, as with any processing demand. However, some additional operations are desirable for efficient use in a real-time system. Since flags are used widely to synchronise operations or indicate conditions in real-time software, it is desirable to have set/clear and test instructions which operate on single bits in store. The alternative is to simulate these instructions by logical ANDing or ORing (wasteful of program and execution time) or use whole words as flags (wasteful of data storage).

As will be shown later, the 'test and set' instruction (test the value of a bit and set it or clear it as one indivisible instruction) is essential for the realisation of dual or multi-processor systems. Its lack can lead to extra hardware being built to simulate it.

The demands of signal processing require more arithmetic than the simple single precision add and subtract, and a multiply/divide facility, preferably with multi-length arithmetic, is essential. Failing this, a wide range of shift instructions can lead to the efficient implementation of these functions in software.

It has already been stated (4.3.1.2) that floating point working is important for signal processing applications, so this facility would prove extremely useful.

4.3.2.2 Instruction Speed

In a digital processing system the maximum frequency which can be resolved in a waveform is half the sampling frequency (sampling theorem). The sampling frequency is determined by the speed at which the set of calculations for one sample can be completed and the next sample taken. A basic block of digital filter is a bi-quadratic (i.e. the filter polynomial is one quadratic expression divided by another) which requires about five multiplications or divisions and about twenty-five

other simple arithmetic operations. A software loop to perform multiplication or division has a dynamic instruction count of about six instructions per bit - that is, about 100 instructions for a 16-bit multiply/divide. The time taken for the bi-quadratic iteration is therefore:

$(5 \times 100) + 25\ t_i$ t_i is the average instruction time of the processor

t_i	Sampling Frequency	Maximum Bandwidth
1.5 µS	1,250 Hz	630 Hz
3 µS	630 Hz	320 Hz
5 µS	380 Hz	190 Hz
8 µS	240 Hz	120 Hz
10 µS	190 Hz	95 Hz

The above table (apart from underlining the need for a fast hardware multiply/divide unit) shows the heavy dependence of bandwidth on instruction time. Clearly more complex filtering operations would have to be performed even more slowly. It is important to realise that in real-time signal processing, unlike many commercial 'real-time' tasks, slowing down response speed means that a process becomes impossible to realise, not just more inconvenient or expensive.

4.3.3 Input-Output System

The question of how peripheral devices are to be controlled is dealt with in Chapter 8. However, two aspects of I-O system design warrant consideration here. These are the interrupt and DMA structures and the bus bandwidth.

4.3.3.1 Interrupt and DMA Structures

Interrupts are used to initiate execution of a routine on a command external to the processor. They are the only method by which a high speed response can be made to an external condition without the processor being occupied almost entirely in waiting for the condition to arise. The processor must be capable of accepting a number of interrupts and steering the requests to different locations in the program store. The hardware must also be able to arbitrate between two simultaneous interrupt requests on an assigned priority basis. It is necessary that interrupts, preferably individual interrupts, can be prevented (locked-out) by the software so that critical parts of a program are not interrupted. Clearly as much information about machine status as possible must be stored and restored so that programs can correctly be restarted after an interrupt.

Direct Memory Access (DMA) involves a peripheral device controlling the bus to make direct accesses to the store, without processor intervention. The device controller must be able to provide an address for the data and preferably count the number of transfers as well, so that a complete sequence of DMA operations can be performed by a single control word. The speed of the DMA channel should be close to the maximum of which the memory is capable, so that the high potential speed of this system can be fully exploited.

4.3.3.2 Bus Bandwidth

Bus bandwidth is the maximum number of data transfers per second possible in the bus. Since most micro-processors use only one bus for all communications, this figure must be the sum of all instructions and data references, program controlled I-O transfers and DMA transfers. The latter can be the limiting factor, since instruction execution can occupy the bus for such a large proportion of the total time that the DMA transfer rate is extremely restricted. Synchronous buses suffer less from this because a certain amount of cycle-stealing (taking unused time slots for DMA transfers) can

lead to high DMA and instruction rates simultaneously (5.2.1). If the DMA-instruction conflict becomes serious, it may be necessary to divide the bus into isolated lengths on which independent transfers can occur and only join them into one bus when needed (5.2.2).

4.4 Summary

By setting a 'test task', in this case simple digital signal processing, one can compile a list of performance characteristics which an RSP must have in order to execute the task. Consideration of physical properties necessary for true portability leads to a similar set of criteria. Clearly these limits are no more objective than the choice of problem for the test task, but it is necessary to have some quantitative guidelines for an RSP design before considering the different possibilities. Such an approach also avoids the intellectual dishonesty of tailoring the problem to the solution, instead of vice versa.

The main constraints for the initial development are summarised below.

Physical

Size	0.2 x 0.2 x 0.3 m
Weight	5 - 7 Kgs
Power	25 - 40 W
Memory Hold Time	10 - 12 hours

Processing Performance

Wordlength	At least 16 (double length working for floating point)
Instruction Set	All normal arithmetic and logical operations Single bit tests and set/clears Multi-length instructions Hardware multiply/divide, Floating point arithmetic } desirable
Instruction Speed	Simple operation time 3μS
Interrupt System	Vectored and prioritised Multi-level selective lock-out facilities

DMA	Automatic address provision and transfer count High channel speed (e.g. 1MHz for a 600nS memory)
Bus	Plenty of unused time which DMA transfers can use.

CHAPTER 5

EVOLUTION OF THE RSP DESIGN

5.1 Introduction

One of the prototype RSP's was constructed from a Ferranti F100 micro-processor, and this was used to investigate various approaches to the solution of some of the performance limits outlined in Chapter 4.

In particular the problems of achieving a high I-O data rate, and at the same time continuing full speed program execution, are considered. The first suggested solution, the two-port store, was designed before it was known that an I-O system, in the form of interface sets, was to be supplied. When it was realised that interface sets implemented a split bus system, this was adopted in preference to the earlier proposal.

The inability of the F100 to perform some arithmetic operations with sufficient speed leads to investigation of the use of the special processor concept to compensate for these failings.

5.2 Maintaining High I-O Data Rates

5.2.1 The Two-Port Store

A two-port store is a block of storage which can be accessed simultaneously from two different buses. In practice the accesses are closely interleaved in time, rather than being truly simultaneous. Making data available to more than one bus at the same time would require that individual memory circuits be constructed with two address inputs and two data input/outputs. While such circuits do exist (e.g. ref. 54), their storage capacity is small (<64 words) owing to the difficulty of designing an efficient addressing mechanism (the usual 'cross-over' system of specifying a row and column address and placing data from the selected cell onto a bus will only cope with one address and one item of data at a time).

If DMA accesses and instruction fetch and execute cycles are to be interleaved without increasing program running times, then unused 'time-slots' on the bus must be detected and used for DMA. In a synchronous or semi-synchronous bus system such time slots are clearly defined; and since they are predictable and of fixed length, can easily be used for DMA purposes. Fig. 51 shows the bus contents of a simple synchronous processor during execution of one instruction.

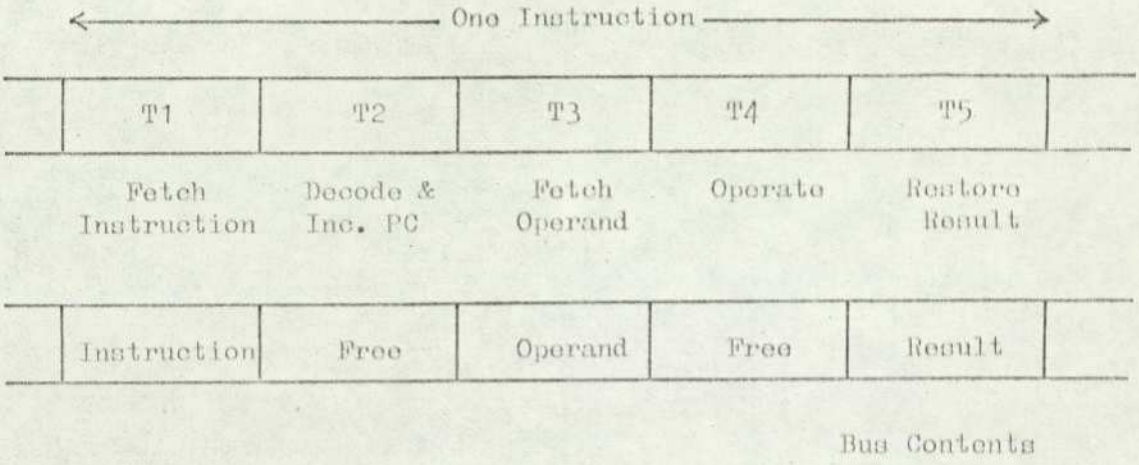
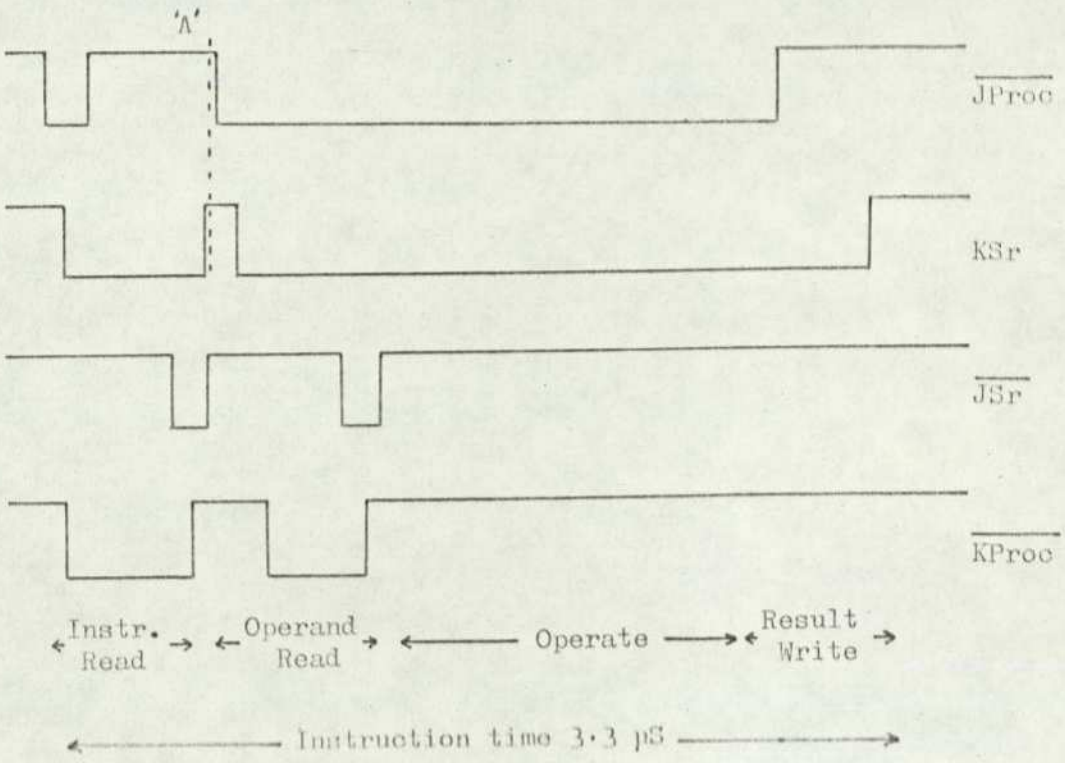


Fig. 51. A Simple Synchronous Processor - Bus Contents during Execution of an Instruction.

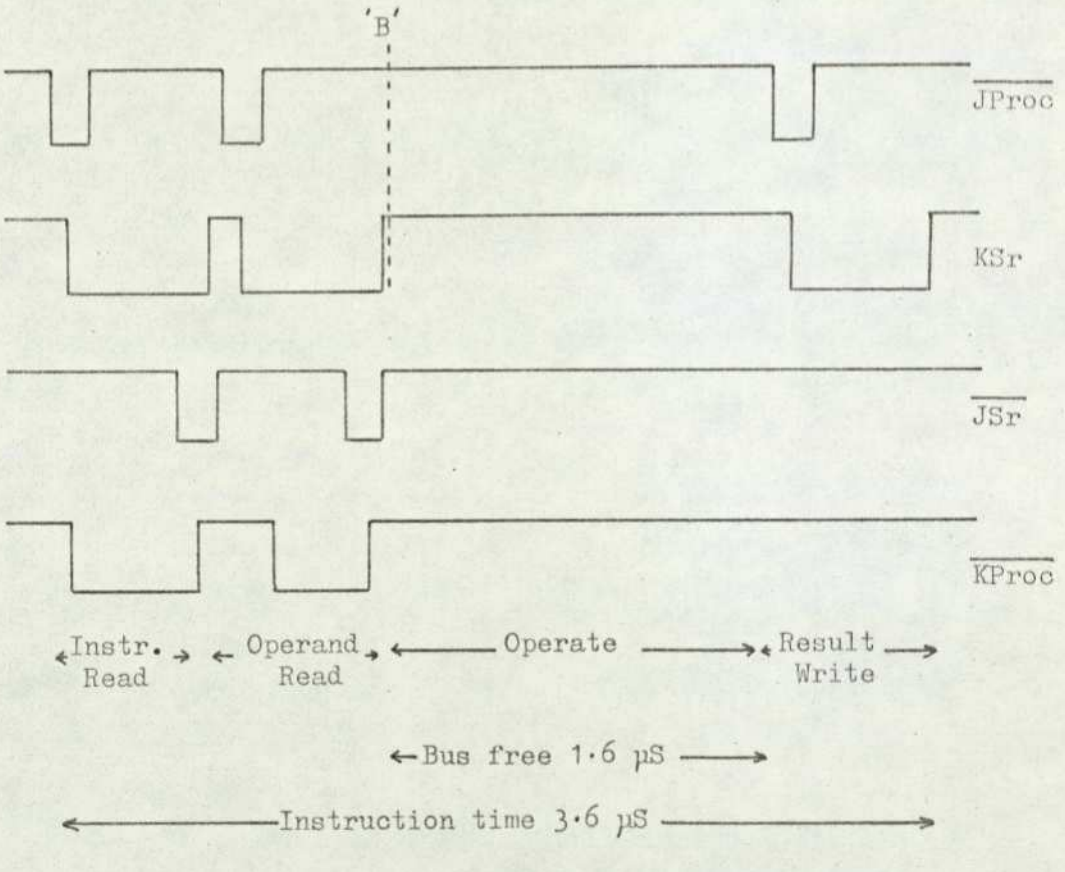
Since the bus is free during T_2 and T_4, these times can be used for DMA transfers. This can lead to the so-called 'transparent DMA', i.e. a situation where DMA transfers have no effect at all on execution time.

In asynchronous processors (e.g. F100L) the detection of such slots is more difficult since in such processors there are, by definition, no timing rules to determine bus contents. Instead the DMA circuitry must monitor the bus control signals to determine suitable times for DMA transfers. Appendix VI gives details of the F100 bus control signals from which it can be seen that the bus is free when all four timing signals are inactive (high).

Fig. 52 (a) shows the bus control signals during a typical instruction. At point 'A', after the instruction has been fetched but before it has been decoded, all the control signals are high. It is at this point that DMA requests are usually accepted by the processor. The granting of such a request inevitably means that instruction execution is slowed



(a) Normal Operation.



(b) With Read/Modify/Write cycles converted to Read then Write.

Fig. 52. Bus control waveforms for a typical instruction (F 100L).

since the processor would normally be ready to proceed with the operand fetching within 100nS. Instead it has to wait until the DMA cycle is completed. The instruction shown in fig. 52 is of the 'St = St + A' type where St is store location and A is the accumulator. Although instructions which leave the result in the accumulator (e.g. A = St + A) do not have a read/modify/write cycle, and so do not hold up the bus during the operation, they are unable to accept DMA request during the operation time, or at any time other than point 'A'. Also instructions of the 'St = A' type which only need a write cycle actually use a read/modify/write cycle and so can be treated as the example in fig. 52. The reasons for these two apparently inconsistent methods of operation are obscure but probably relate to internal hardware savings within the micro-processor. So it can be seen that all F100 instructions have one point at which DMA can be accepted, and this acceptance will always increase the instruction execution time.

In a real-time sampled data system, 'transparent DMA' is desirable for a number of reasons. Firstly it maintains maximum program speed while DMA transfers are occurring, but, more importantly, it means that a constant program speed can be maintained while the DMA load varies over a wide range. This latter point is important in signal processing since speed/bandwidth calculations will be invalid if processing speed is subject to a 20 - 30% fluctuation depending on data I-O load. The original proposal to obtain transparent DMA was to create the time slots needed, by converting all read/modify/write cycles into read then write ones, thereby freeing the memory during the operation. This can be achieved by intercepting the $\overline{\text{JProc}}$ line from the processor and altering its timing as shown in fig. 52 (b). This increases the instruction execution time by a fixed amount (read time + write time - read/modify/write time + 1 logic beat) but means that the memory is free during all the operation time (16 - 17 logic beats). In this system a DMA request would be accepted at point 'B' and usually two could be dealt with, one after the other, without affecting instruction execution.

In order to assess the efficacy of this approach, some calculations were made of the instruction rate v. DMA rate for the normal and modified systems. These were based on a crude estimate of instruction occurrence frequency. The assumptions were:

1. 30% of instructions are of the form St = St + A, i.e. load a store location or return a result to it.

2. 60% of instructions are of the form A = St + A, i.e. load accumulator or return result to accumulator.

3. 10% of instructions are miscellaneous bit tests or shifts which are broadly similar to the first category but which have no indirect addressing modes.

4. 20% of instructions capable of using indirect addressing do so, and half of these are auto increment or decrement.

5. The cycle times are Read cycle (R) = 500nS; Write cycles (W) = 500nS; Read/Modify/Write cycles (M) = 700nS; Logic beat (L) = 100nS.

Total number of each cycle type in 1,000 instructions:

		Occurrences	R	W	M	L
A = St + A	TOTAL	600				
	Direct Add	480	2	-	-	18
	Indirect Add	60	2	-	1	19
	Ind. + Auto inc./dec.	60	2	-	1	34
St = St + A	TOTAL	300				
	Direct Add	240	1	-	1	18
	Indirect Add	30	1	-	2	19
	Ind. + Auto inc./dec.	30	1	-	2	34
Miscellaneous		100	2	-	1	10

Adding up the various numbers of cycles gives an estimate of the execution time for 1,000 instructions. This comes to 3.129mS which includes 580 read/modify/write cycles. If these are converted to read and write cycles, the execution time becomes 3.303mS - a 5% increase.

Fig. 53 shows a graph of instruction rate versus DMA transfer rate. This assumes that a DMA transfer takes one read or write cycle and two logic beats - a total of 700nS. It is also assumed that the number of DMA slots is 58% of the number of instructions, as was shown above.

The graph shows that for DMA rates below about 100K transfers/sec. the normal system maintains its small (about 5% with no DMA) advantage in processing speed. Between 200K transfers/sec. and 800K transfers/sec. the modified version has a considerable advantage after which the two performance curves converge. At 1.4M transfers/sec. the bus is occupied continuously with DMA and no instructions are possible. It is significant that the modified version suffers no speed restrictions for DMA rates up to 350K/sec. and is only 10% slower than in non-DMA mode with a DMA rate of 550K words/sec. It may be thought that DMA rates higher than 100K transfers/sec. are not of general interest in signal processing. This is true of continuous operations, such as digital filtering, where faster rates would allow no time for processing in between samples. However, it is often required to take a block of samples and store them by DMA so that a piece of waveform can be analysed later. Under these 'burst' conditions, sampling may be required at close to the maximum DMA channel speed (limited by memory cycle time). This would typically be 1M transfers/sec. or greater.

The foregoing has assumed a single two-port store where bus occupancy and store use are the same thing. If, however, the system is extended to a number of two-port stores, the availability of each store block is increased as the processor spends less of its time accessing the given block. The original proposal for the RSP I-O system envisaged an input-output controller (IOC) working into one port of a two-port

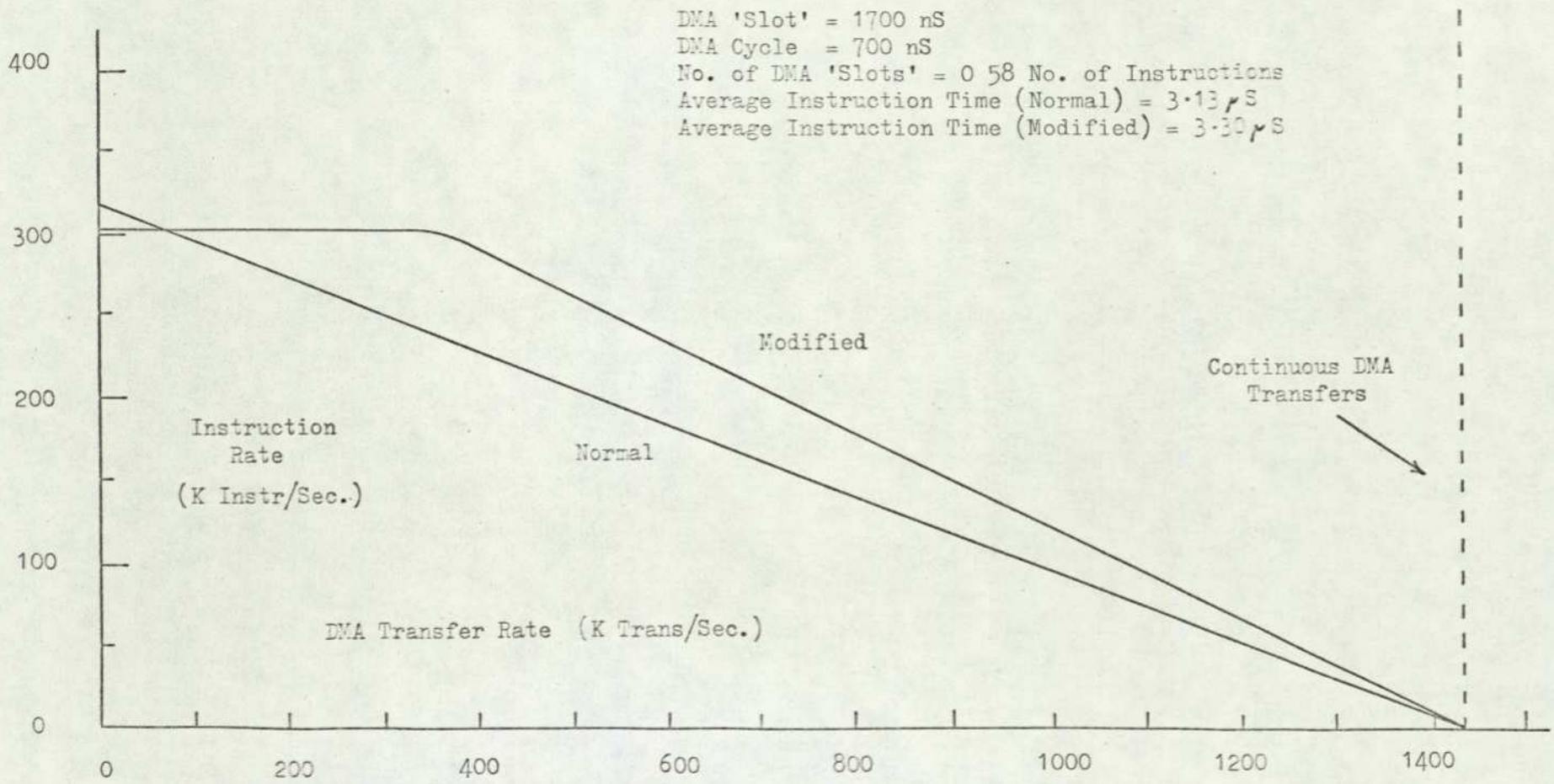


Fig.53 Instruction Rate versus DMA Transfer Rate.

store while the processor accessed storage through the other port. The IOC could be either a hardwired unit or a second processor system. The main processor was also to have a small fast store available to it, for the storage of much used operands. Apart from increasing execution speed, this facility ensured that memory accesses were distributed between store blocks. The consequent increase in availability for DMA transfers in the two-port store ensured that higher DMA rates could be maintained without detriment to processing speed.

The development of the system was to take place in three phases as shown in fig. 54. The two-port store consisted of a 4K non-volatile store developed by a colleague (ref. with two identical interfaces attached. Initially one of these interfaces was developed with the store as part of the original single processor RSP.

However, towards the end of phase 1, the author became aware of the existence of interface sets with a mode of operation (bus extension) which allows coupling of two processor buses. The development was, therefore, curtailed at that point. The store interface which has a number of features besides the conversion of RMW cycles (e.g. a 'write-protect' facility using a seventeenth bit), is described in appendix VI.

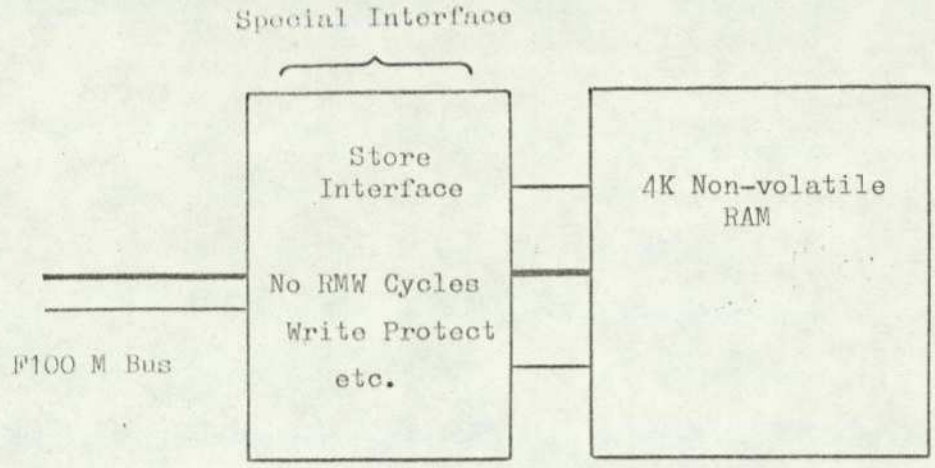


Fig. 54 (a). Development of Interface with 4K RAM

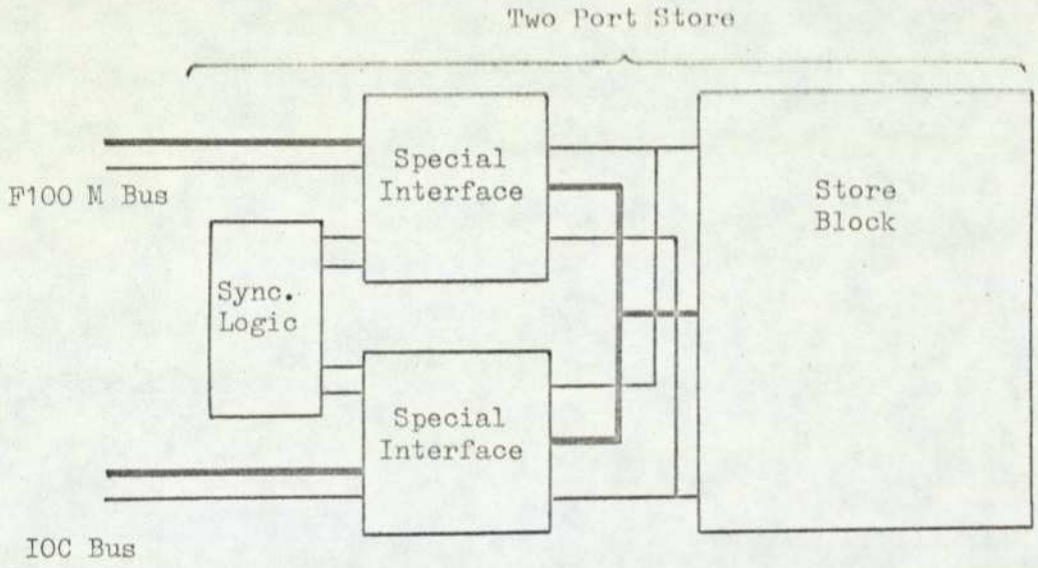


Fig. 54 (b) Two Port Store with F100 M System

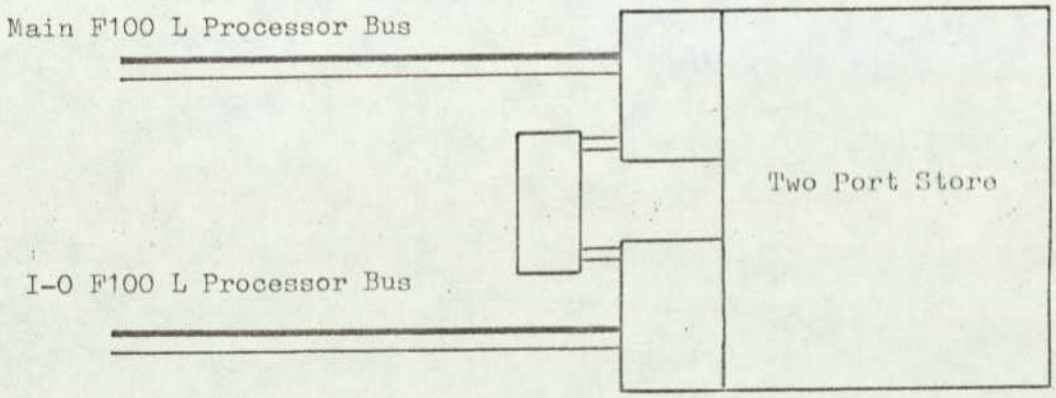


Fig. 54 (c) Two Port Store Linking Dual F100 L Processors.

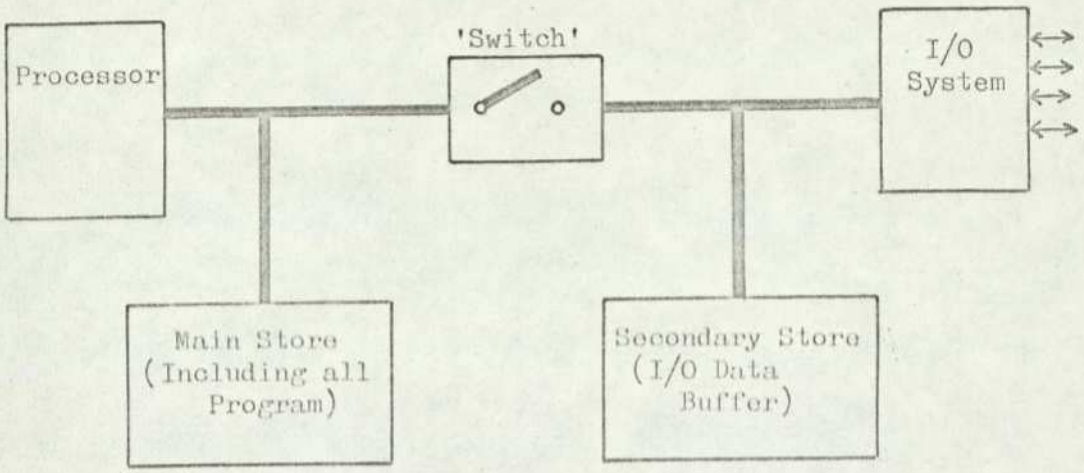
5.2.2 Dividing the Bus

An alternative way of achieving high concurrent DMA and instruction rates is to divide the bus into two or more parts, each with some associated storage on which simultaneous transfers may take place. The parts may then be joined for transfers between the store blocks. Fig. 55 shows a simplified diagram of such an arrangement used to implement an I-O system.

The F100 interface sets have a mode of operation in which two sets (connected back-to-back) act as a link between two, otherwise independent, buses (ref. 21). In such a system it is necessary to distinguish between a primary and a secondary bus since, if the two buses are treated identically, it is possible for the system to 'lock-up' with each bus trying to access the other simultaneously. This occurs because the bus extension interface set, as it is known, is not switched on and off by software but accesses or isolates the secondary bus according to which addresses are being used.

From the primary bus, the extension set appears as a block of store locations. Only if a store cycle on the primary bus involves an address to which the extension set has been wired to respond will the extension set pass on the request to the secondary bus. This occurs as a DMA request so that from the secondary bus the extension set appears as a peripheral which makes DMA requests. In this way a block of store can be made accessible to both buses, while the buses are isolated at all times when not accessing the common store.

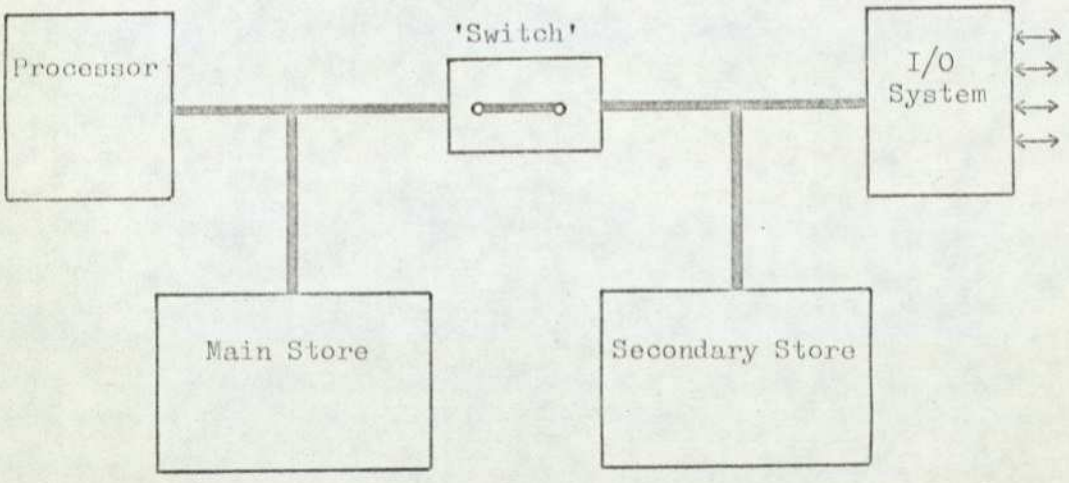
A dual processor RSP based on bus extension interface sets is described in 'Micro-computer Architecture for Software Defined Instruments' presented at International Microcomputers, Minicomputers and Microprocessors '77, Geneva, May 1977, and published in Proceedings.



(a) With the two halves of the bus isolated, simultaneous instruction execution and data I/O transfers can occur.

Note :

The I/O system could be a second processor with its instructions in the secondary store.



(b) With the bus halves joined, the processor can access the data in the secondary store.

Fig. 55. A split-bus system. The 'switch' can be under program control or it can be operated by the processor using an address in the secondary store.

The buffering techniques and signal flow of the processor is described in 'Dual Processor Hardware for Signal Processing' presented at International Microcomputers, Minicomputers and Microprocessors '78, Geneva, June 1978, and to be published in Proceedings.

'Micro-computer Architecture for Software Defined Instruments' presented at International Microcomputers, Minicomputers and Microprocessors '77, Geneva, May, 1977, and published in Proceedings.

This paper (pp. 102-109) has been removed for copyright reasons

'Dual Processor Hardware for Signal Processing' presented at International Microcomputers, Minicomputers and Microprocessors '78, Geneva, June 1978, and to be published in Proceedings.

This paper (pp. 110-115) has been removed for copyright reasons

5.3 Compensating for the Arithmetic Shortcomings of the Processor

As has already been stated (4.3.2.1), multiply and divide operations are important in signal processing, and many other, applications. The F100L has neither of these as standard instructions although a single integrated circuit to perform multiply/divide functions is under development. The F100L does, however, have a facility whereby its instruction set may be extended by special purpose hardware. This hardware is known as one or more special processing units (SPU's) which can be triggered by unused machine codes.

5.3.1 The F100L Special Processing Unit Facility

The F100L has a range of instructions which cause no internal processing to be performed, but instead start a 'handshake' sequence between the processor and an SPU. This informs the SPU that a reserved instruction is on the bus, so that it can determine if the instruction refers to itself or another SPU. Since the SPU instruction has ten unused bits, patterns can be chosen to trigger various functions, or operand addresses can be passed over in the instruction word. The F100 interface set has a mode of operation for interfacing an SPU in which it responds to a block of SPU instructions and fetches and restores operands by DMA. The main processor halts operation until the SPU function is completed and the handshake line released (short functions), or the processor can run on and be informed of completion by a program interrupt (longer operations). Since, in general, the ten bits in the SPU machine code word are insufficient to specify the operation completely, additional words (e.g. operand addresses) can be placed after the SPU word. In this case the SPU must reload the processor program counter so that the additional words are not executed.

5.3.2 Design of Special Processing Units

A simple signed multiplication unit was built and interfaced to an F100 as a block of store locations. This was done because the non-availability of interface sets made operation in the SPU mode difficult. The multiplier performs two's complement multiplication of two 16-bit numbers to a 32-bit result using a serial-parallel implementation of Booth's algorithm (ref. 55 and 56).

From consideration of this unit it became apparent that operation time was limited more by store accesses than by actual processing operations. A 16-bit by 16-bit operation with a 32-bit result needs four store accesses. In SPU mode these would be by DMA so the total time would be about 4 x (700nS) = 2.8μS. If the operands are to be transferred under program control, then this overhead is increased to about 4 x 3.5μS = 14μS. Since the prototype multiplier (despite its partially serial operation) can multiply in less than 1.5μS, it can be seen that the operand fetch/store times are the dominant feature of SPU performance.

This becomes more serious still for floating point units which require two words to store each variable, thereby implying six store cycles per operation. It therefore becomes necessary to have a block of fast registers (possibly larger than sixteen bits each) on which the operations can be performed.

These can be loaded, all processing operations (including storage of intermediate results) can be performed, and the block can then be transferred to store. The operations could be controlled by a single word being written to the SPU. Fig. 56 shows a proposed design for a floating point SPU. This could conveniently be based on 'bit-slice' processors which have an internal 'scratchpad' of sixteen registers and a micro-instruction time of 75 - 100nS (e.g. ref. 57).

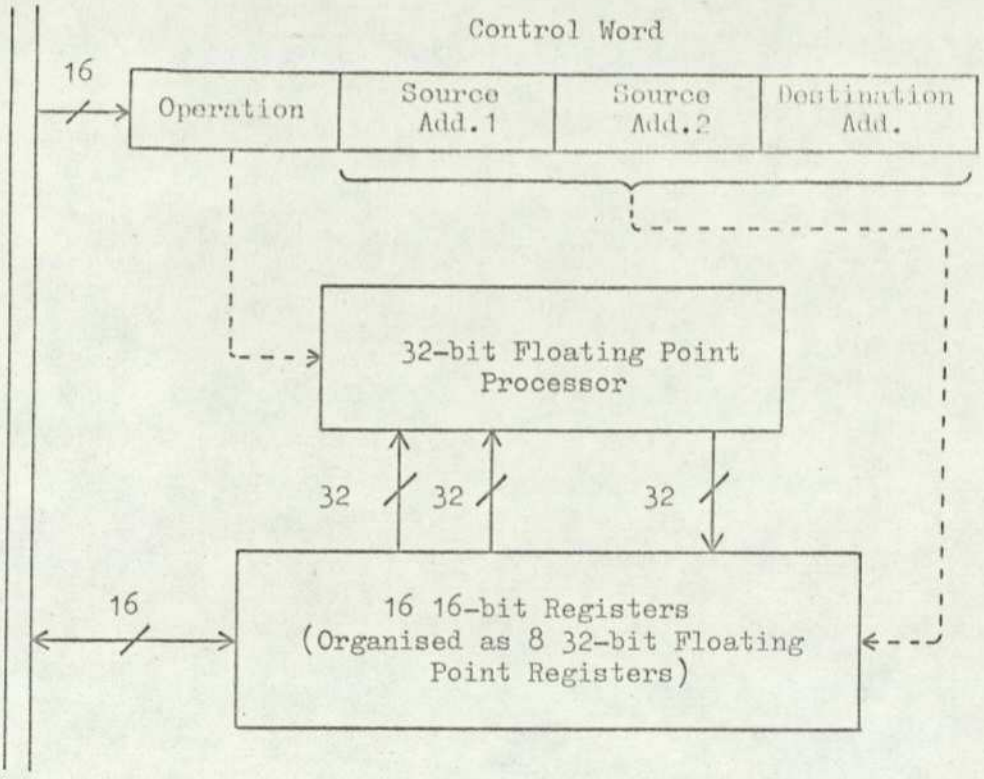


Fig.56 A Floating Point SPU.

5.3.3 Validity of the SPU Approach

Since the RSP is a flexible processing unit in which instrumentational processes are realised as software, it seems inconsistent to start replacing software routines by blocks of special purpose hardware. The advantage of the RSP approach is that the generality of the hardware combined with the flexibility and ease of design of software leads to an easily and cheaply designed instrumentation system. Such an arrangement is not, however, invalidated by replacing standard software processes by hardware units, providing the process is sufficiently clearly defined that no variations are allowed. For example, it is pointless to build a hardware digital filter since it is reasonable to assume that most applications will require a different design which would mean a hardware modification - exactly what the RSP seeks to avoid. However, one may say that all filtering/convolution operations require that 'sums of products' be accumulated, and one may therefore incorporate this function in hardware without jeopardising the generality of the system. If this technique is widely

applied, the main processor becomes simply a means of organising the various hardware units within the system. As hardware becomes smaller and more powerful, it seems that this is the path the RSP, and probably all processing systems, will take.

The other limiting factor on SPU is the one of power consumption as mentioned in 4.2.2.1. The micro-programmable bit slices mentioned earlier, for instance, are Schottky TTL and thus have a power consumption which puts them out of the question as far as a portable RSP is concerned.

5.4 Summary

Ways of overcoming the two major performance limiting factors for an RSP, input-output data rates and the lack of high speed arithmetic, have been outlined.

The two-port system offers the best I-O performance particularly if more than one store block is used. However, it requires the use of non-standard interfaces to the memory, and modifications to the timing relationships of the bus signals. With the advent of standard interfacing components (interface sets) the two-port system becomes less attractive as opposed to the split bus arrangement which is specifically catered for in the interface set design.

Special processing units provide the means to compensate for the shortcomings of the processor's instruction repertoire, but at a cost in additional hardware and power consumption. The main difficulty is seen to be one of isolating processes large enough to be worth replacing by hardware but still completely general in nature.

CHAPTER 6

TWO PROTOTYPE RSP's - DESIGN & OPERATION

6.1 Introduction

During the research project, two prototype RSP's were constructed to allow techniques to be evaluated and designs to be developed. Early in the project an RSP based on the F100M, an MSI equivalent of the F100L, was constructed. Some eighteen months later a dual processor based on F100L chips and LSI interface sets (that had subsequently become available) was constructed. Both these prototypes were demonstrated at the International Microcomputer, Minicomputer and Microprocessor Exhibitions in Geneva (while the papers found in Chapter 5 were being presented at the associated conference), the MSI version in May 1977 and the dual F100L system in June 1978. These demonstrations involved real-time measurements already undertaken by specialised equipment in the author's laboratory and were, therefore, representative of the applications envisaged for the RSP.

6.2 The MSI Prototype

The MSI equivalent (the F100M) differs from the single chip version (F100L) in some respects. The central processor is functionally identical although operating at a slightly slower clock rate. The size difference, five printed circuit cards compared to a single 40-pin package, meant that true portability was not possible. A 19" rack configuration was used to give maximum accessibility while castors provided a certain amount of 'in-laboratory' mobility. The MSI implementation gave rise to high power supply requirements (10 - 15 A @ 5v) and consequently fans were needed to remove the dissipated heat.

The only 'architectural' differences between the F100L and the F100M are in the I-O system. Because the economics of LSI circuits and their discrete logic equivalents are very different, it is impracticable to reproduce the LSI interface components (interface sets) in discrete logic. Instead a four-channel multiplexed input-output controller (IOC) is

used. This does not offer all the facilities of the interface sets. In particular, there is no proper priority system for interrupt and DMA requests.

6.2.1 Prototype Structure

Fig. 61 shows a block diagram of the MSI prototype RSP. The hardware provided by the manufacturer consisted of the processor (five cards), 4K word store and interface (two cards), IOC (two cards) and a one-card interface to a front panel. Although the presence of a front panel is contrary to the principle of the RSP, it was felt necessary to include it for hardware testing at the prototype stage.

The hardware development initially consisted of the building of the four peripheral units needed for signal processing: the main computer link, an analogue-digital, digital-analogue converter, a real-time clock and a hardware multiplier unit. The converter development was the responsibility of the author.

6.2.2 The A-D, D-A Converter

Appendix I gives a detailed description of the design of the A-D, D-A converter.

One of the main problems to be investigated was that of timing of the signal samples. The usual method is to have the real-time clock interrupt the processor and the interrupt program, then control the A-D converter to take a sample. Since interrupts are only acknowledged at the end of instructions, it follows that the delay between the real-time clock pulse and the sample being taken, may vary by an amount equal to the length of the longest instruction. That is:

$$t_f < \text{Delay} < t_f + t_i$$

where t_f is a fixed delay concerned with saving the program counter and working registers, and t_i is the maximum instruction time.

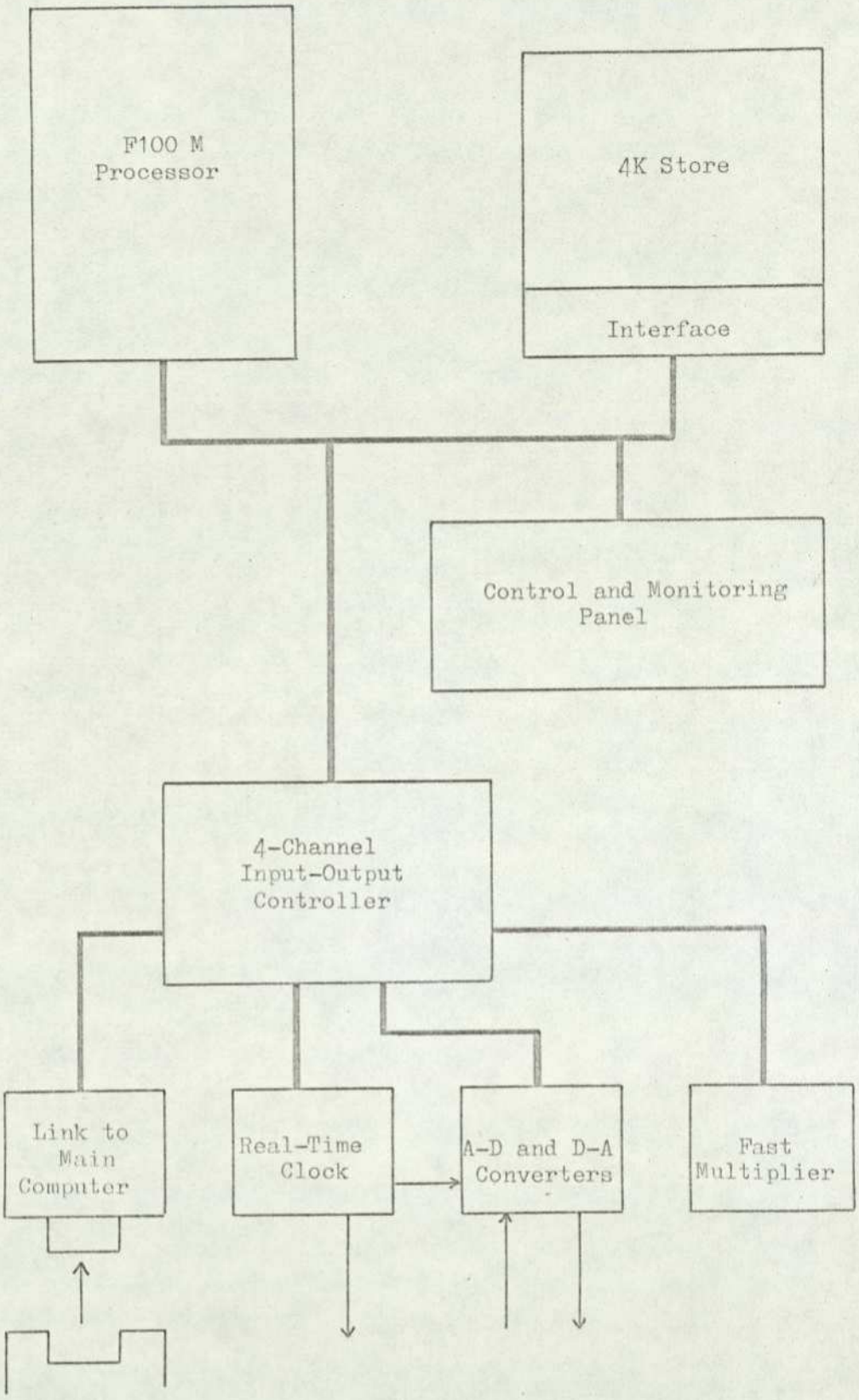


Fig. 61. The F100 M - based Prototype RSP.

For the F100L the longest instruction takes approximately 9.8μS. If one assumes that, say, a 1% fluctuation in sampling rate is the maximum acceptable, then the maximum sampling rate is approximately $\frac{1}{0.980}$ KHz.

This shows that if an interrupt controlled A-D converter is used in an F100L system, then sampling rates greater than approximately 1KHz lead to unacceptable (>1%) fluctuations in sampling frequency.

The processor is capable of executing simple signal processing tasks with sampling rates of 10 - 15KHz and can acquire blocks under DMA at well over 100KHz, so this limitation is not acceptable.

The method chosen in the RSP is to make the real-time clock pulse request the interrupt and simultaneously cause the signal sample to be taken. By the time the interrupt is acknowledged and the registers have been saved, the sample has already been converted. This reduces the delay from clock pulse to sample and, more importantly, eliminates variations in delay.

6.2.3 The Real-Time Clock

The real-time clock was designed by a project student (ref. 58) to give pulses at programmable time intervals which could be used to control signal sampling. It was designed to allow complete hardware control of the timing functions in the processor without the software intervention which would necessarily involve the type of errors discussed in 6.2.2.

A 32-bit down counter is loaded under program control, and decremented by a 1MHz clock pulse derived from a 10MHz crystal. When the counter underflows (reaches '-1'), a pulse is produced which can cause any of three events:

a) An interrupt can be requested;

b) Additionally a marker bit can be set which is readable by the program;

c) Additionally a signal sample can be taken by the A-D converter.

The original counter value can be reloaded to initiate another count down sequence (continuous pulsing) or not (single-shot). The precision of the RTC is a constant 1μS throughout the ranges 1 to 2^{32} -1μS - approximately seventy-two minutes.

6.2.3.1 Some Comments on Real-Time Clock Design

Most real-time clocks consist of a counter which can be loaded and decremented at a fixed rate, and which causes an interrupt when it reaches 0 or -1. The range of delays provided is small since it is argued that long delays can be accomplished by performing further stages of counting in the software of the interrupt program. It is argued, for example, that if a simple counting interrupt program can execute in 100μS, then there is no need for the RTC to provide delays of > 10mS, since at this point the processor would only be spending 1% of its time in handling interrupts. However, this mode of operation does not allow direct pulsing of the A-D sampling circuit and therefore introduces timing errors. Moreover, since at each interrupt the counter must be reloaded and restarted by software, such timing errors are cumulative. It is thus important that the whole range of sampling speeds can be directly selected from the RTC without software intervention, and the RTC requirements for range and precision are accordingly most stringent.

The design of the prototype RTC involved a fixed precision across an (unnecessarily) large time range. The number of control bits could have been minimised by the use of a programmable pre-scaler followed by a shorter counter. This would have allowed a wide range with a variable precision. Such a system, fig. 62, is proposed for use in the later (dual processor) prototype. Fig. 63 shows the precision of each type as a function of the delay interval.

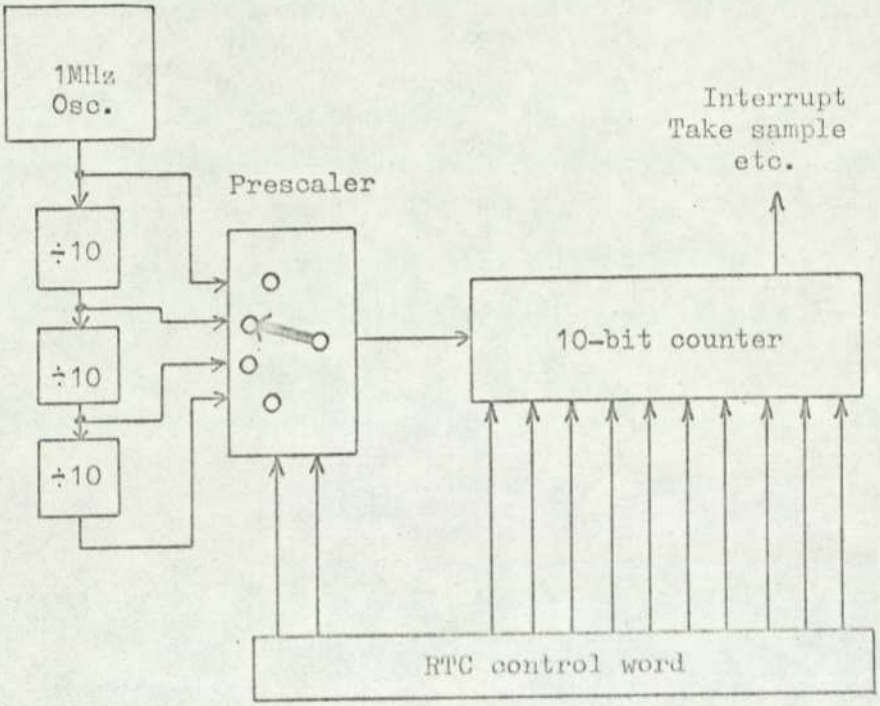


Fig. 62. A real time clock with programmable prescaler.

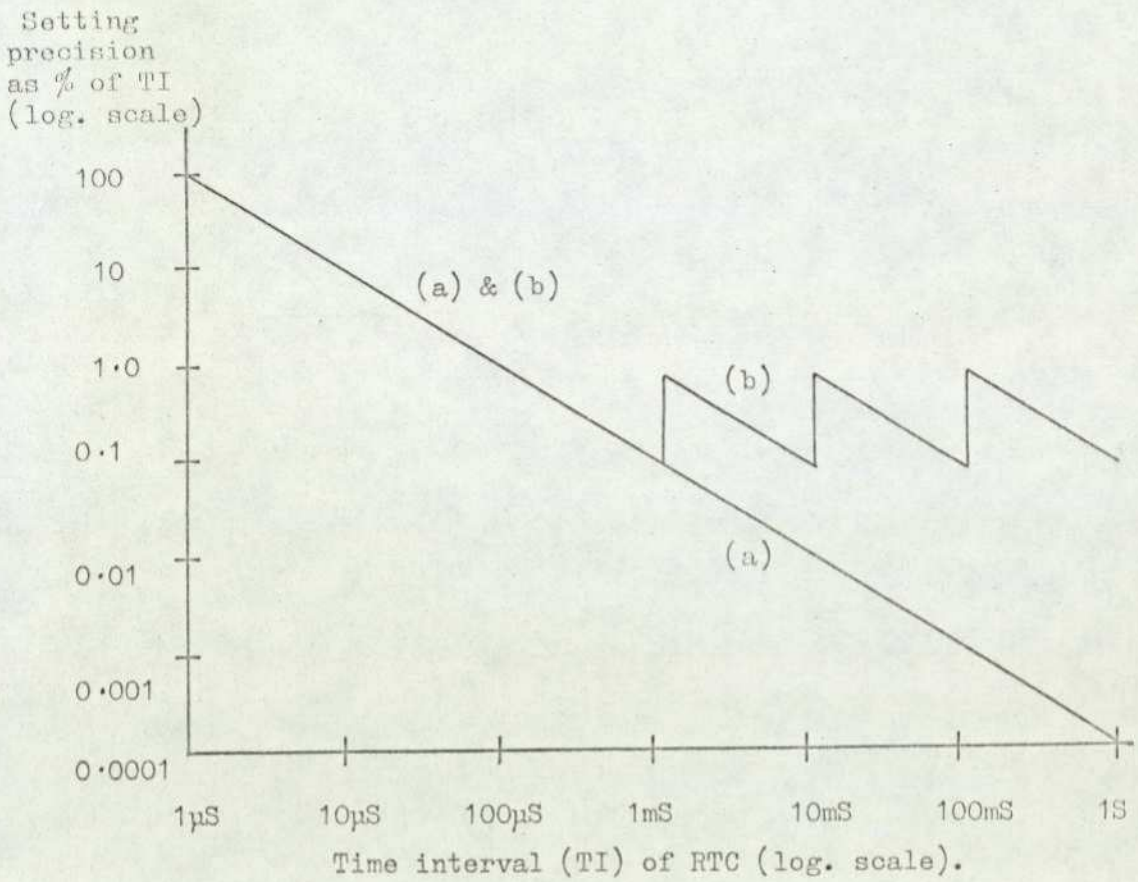


Fig. 63. Setting precision versus time interval for
(a) real time clock used with F 100M
and (b) prescaled RTC shown in Fig.62.

6.2.4 A Demonstration of the MSI Prototype

The signal processing performance of the prototype, together with its A-D and D-A converters and real-time clock, was assessed by programming a simple real-time data acquisition task. This task, the capture and display of randomly-occurring phenomena within an electrical signal, was chosen for two reasons.

Firstly the capture and display of random phenomena had previously been undertaken within the research group in connection with natural charge transport in liquid dielectrics, and a large body of experience had been gained with the use of powerful hardware transient recorders used on-line to the laboratory computer (ref. 59). Thus comparisons could be made between hardware and software realisation of the same process.

Secondly this demonstration shows, in a straightforward way, the relation between processing speed and bandwidth that underlies much of the development work on the RSP.

The transient recorder demonstration takes an electrical signal (from a microphone), constantly samples it and stores the sample in a cyclic buffer in the processor store. When a sample exceeds a certain value (the 'trigger' value), a preset number of samples are taken before the sampling process is halted. In this way the cyclic buffer has, at the end of the process, samples of the waveform both before and after the triggering value. If the number of samples to be taken after the triggering one is set to half the number in the cyclic buffer, the piece of waveform stored has the triggering event half-way through it. Thus information about the signal before the trigger is preserved. Fig. 64 shows the process.

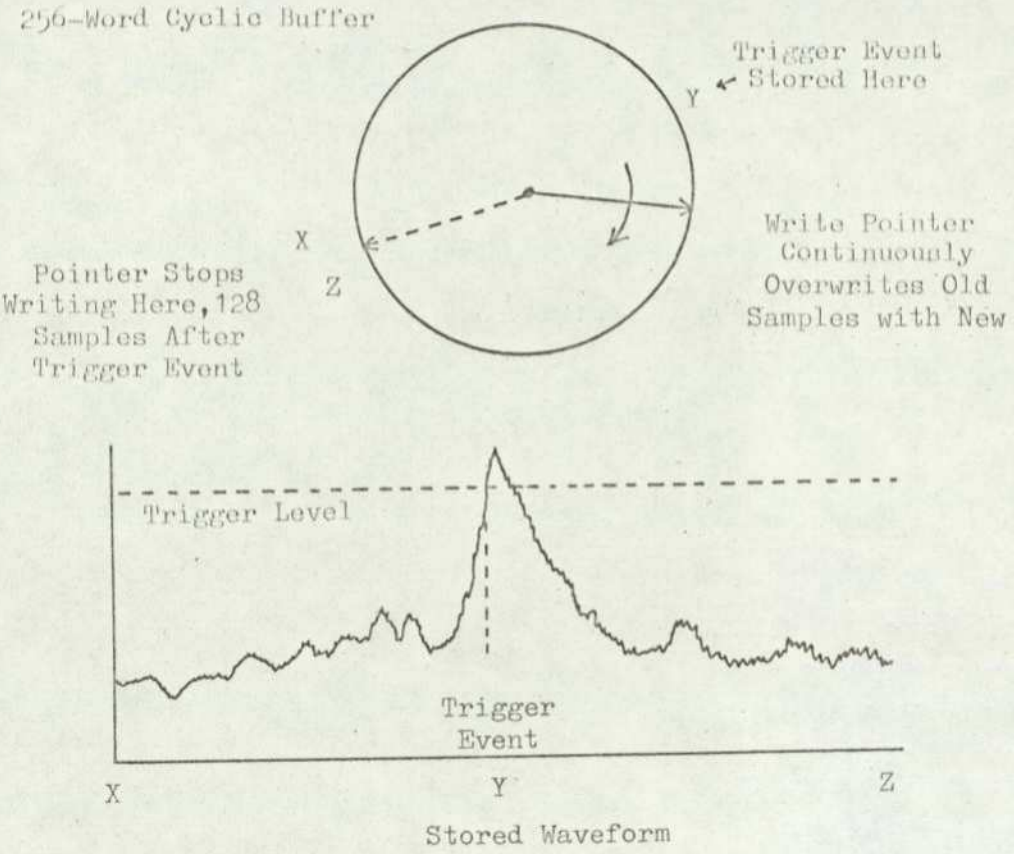


Fig. 64. Principle of 'Transient Recorder' Routine.
(The waveform X to Y is 'pre-trigger' information.)

This portion of waveform is then displayed on an oscilloscope.

In terms of the RSP concept, the demonstration consists of the concatenation of two standard processes: capture a buffer-full of waveform and display that buffer on an oscilloscope screen. Fig. 65 shows a flow diagram of the 'transient recorder' module based on a 256-word buffer. Samples are stored cyclically in the buffer, overwriting themselves until the trigger value is exceeded. Once this occurs, FLAG is set and COUNT is decremented for each sample. When COUNT is exhausted, the process is terminated. In this example COUNT is 129, so 128 samples are stored each side of the triggering value.

The program sets its own sample rate by programming the real-time clock to give repetitive sampling pulses to the ADC. The last sample is marked by setting the top bit so that a subsequent display routine can synchronise properly to the waveform. (Since only eight bits are used for A-D and D-A conversions, the top bit can be used as a flag

without interfering with the data. Later models will use high precision conversion, but this is unlikely to exceed twelve bits, so the same technique can be used.)

Fig. 66 shows the display routine. The buffer is repeatedly written to the DAC whose output is connected to the oscilloscope input. The top bit of each sample is removed before conversion to a voltage. When the 'end of buffer' marker is found (top bit set), a special control code is written to the RTC. This code does not effect the sampling rate but causes the RTC to emit a pulse on a synchronising output. This output is connected to the 'external sync.' input of the oscilloscope. In this way the waveform is displayed in its correct position on the screen.

In an RSP system, the two modules could be called by a 'high level definer' (ref. 60) by specifying the module name and relevant parameter. The demonstration example, for instance, could be called by:

'TRANSIENT RECORDER'

SAMPLING INTERNAL (μS)	100
BUFFER NAME	DEMO
NUMBER OF SAMPLES AFTER TRIG.	128
TRIGGER VALUE	200

'DISPLAY BUFFER'

NAME	DEMO

The definer, from a directory of buffer and data areas, would know that DEMO was a 256-word buffer.

NOTE: This demonstration was exhibited at IMMM '77 Exhibition in Geneva in June 1977. It was interesting to note that it was the only exhibit concerned with real-time processing of audio-frequency signals.

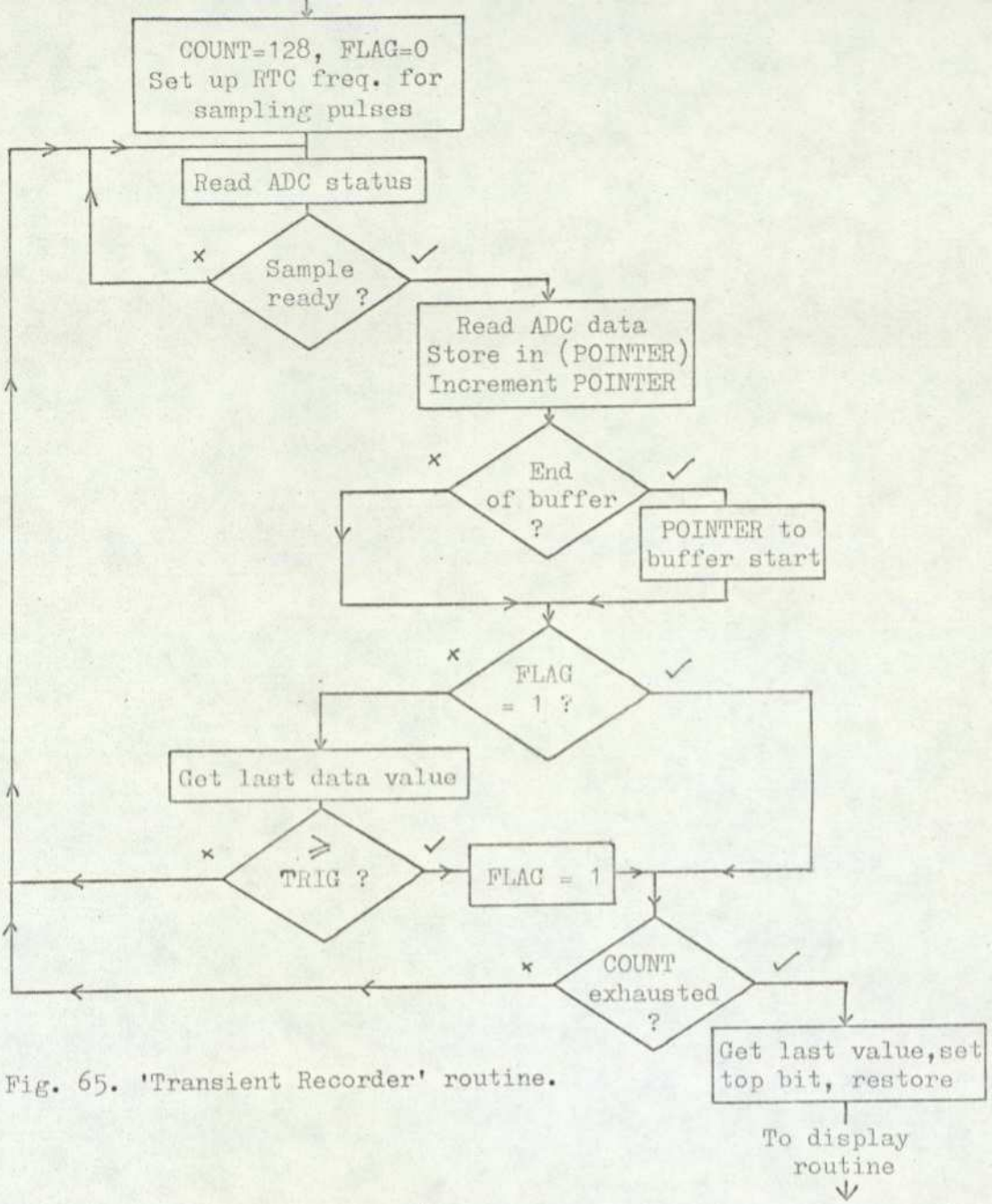


Fig. 65. 'Transient Recorder' routine.

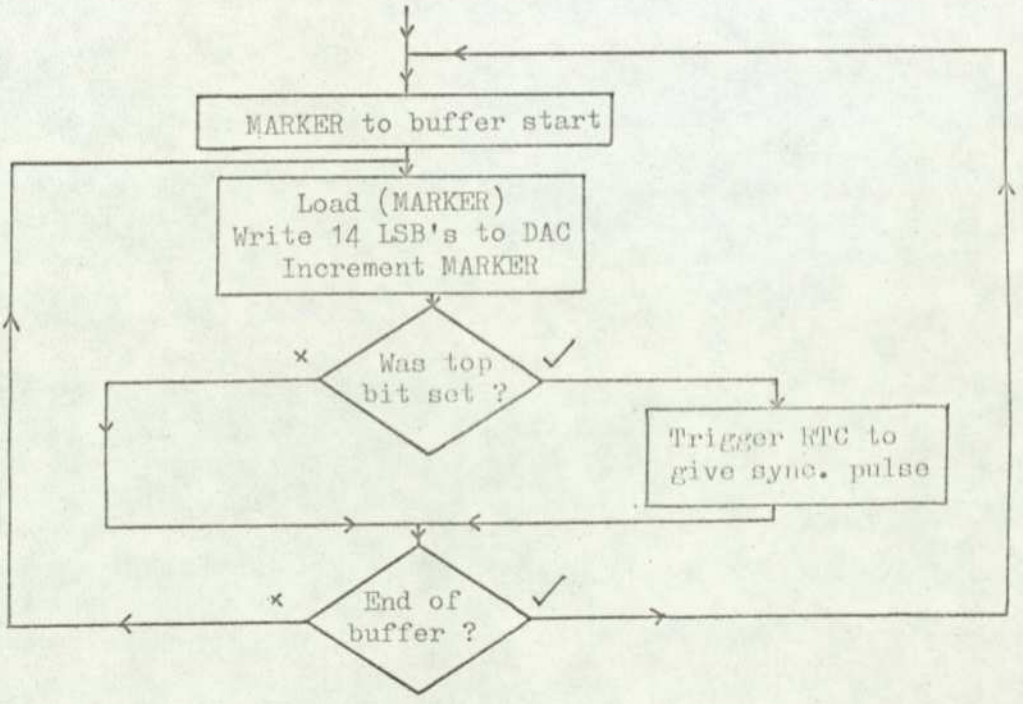


Fig. 66. 'Display Buffer'routine.

6.3 The Dual Processor Prototype

A dual-F100L system was constructed based on the design shown in 'Dual-Processor Hardware for Signal Processing' (fig. 4, Chapter 5) except that no SPU was included on the main bus.

The two store blocks were standard 4K word dynamic RAM's and extensive use was made of interface sets for electrical buffering, and for the bus extension unit which joins the two buses. As with the earlier prototype, no attempt was made at miniaturisation and the same 19" rack system was employed.

6.3.1 The Bus Extension Unit and Choice of Addressing Ranges

As previously described (Chapter 5), the bus extension unit consists of two complete interface sets joined back-to-back so that memory references to certain areas of store are passed from the primary bus to the secondary bus as DMA requests.

Care is needed in the selection of address ranges since each processor must have addresses in the range 0 - 2K available to it. This is because only these locations are directly accessible and only locations 1 - 255 can be used as index registers.

While it is possible to overcome these problems by software limitations which prevent any conflicts of register use, the fact that location 0 is dedicated to being the stack pointer, poses a problem. In practice, it is easier to perform a simple relocation between buses so that the secondary store is 'seen' to be at different addresses by the two processors.

In the prototype the relocation was performed by a circuit represented in fig. 67.

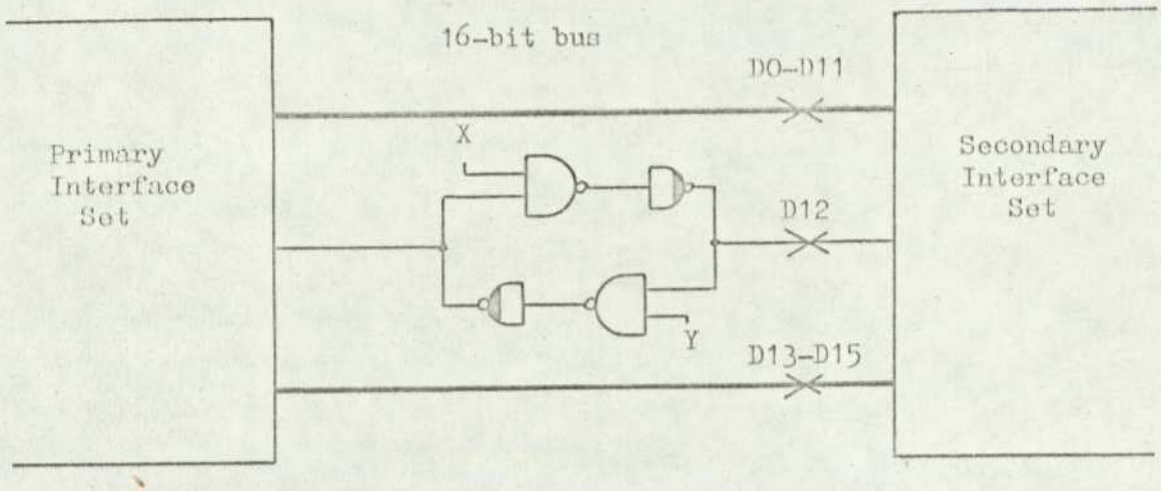


Fig.67. Address Relocation between the Two Buses of the Dual Processor.

One of the sixteen data lines (D 12) between the two interface sets is broken by the arrangement shown in fig. 67. Control signals X and Y are derived from the bus control signals present on the 'processor side' of the interface sets. 'Y' is high whenever a transfer is taking place from the secondary to the primary bus. 'X' is high for transfers in the opposite direction except when an address is being transferred. In this case it is low so that the address received at the secondary bus always has bit twelve as zero. The address range of the primary bus interface set (which is in 'store' mode) is 4 - 8K. This ensures that addresses between 4 and 8K on the primary bus will be passed to the secondary bus but will appear there as 0 - 4K. Thus each processor has 0 - 4K available to it but the main processor can also access the I-O processor's store as 4 - 8K. In the prototype the convention was adopted of using the higher 2K of the I-O store as common store while the lower 2K is private to the I-O processor. That is, the main processor never references addresses between 4K and 6K. Fig. 68 shows a diagrammatic representation of the addressing, including the main link (primary bus) and peripherals (secondary bus).

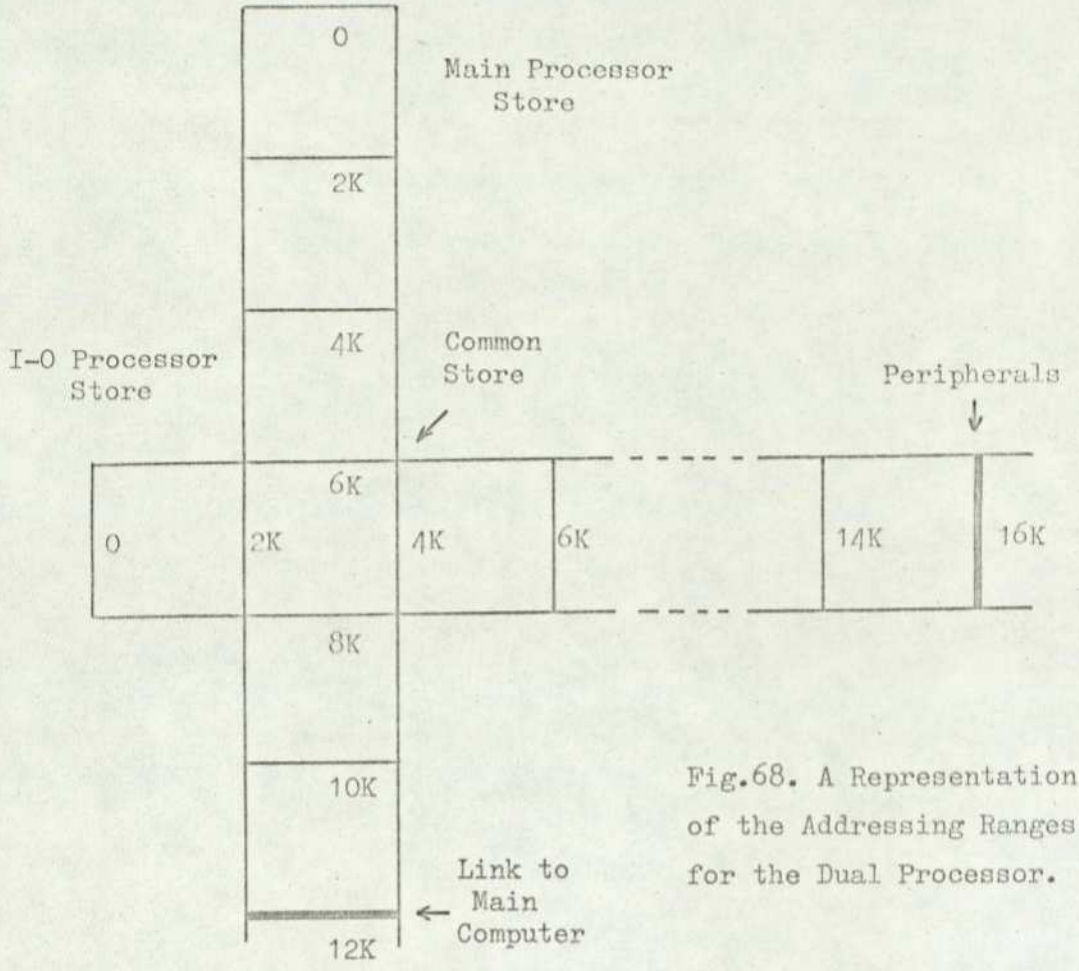


Fig.68. A Representation of the Addressing Ranges for the Dual Processor.

6.3.2 A Demonstration of the Dual-F100L Prototype

The performance and mode of operation of the dual processor were investigated by programming a real-time data acquisition and display task. As with the previous prototype, the task was chosen by selecting an operation already performed in the laboratory by specialised hardware. For some years a pulse height analyser (PHA, ref. 61) has been used to gather statistics about current pulses in stressed liquid dielectrics, and it was this function that was performed by the demonstration.

Pulse height analysis involves the logging of the numbers of pulses of various amplitudes and the display of this information on a screen in a graph of the form of fig. 69 (a).

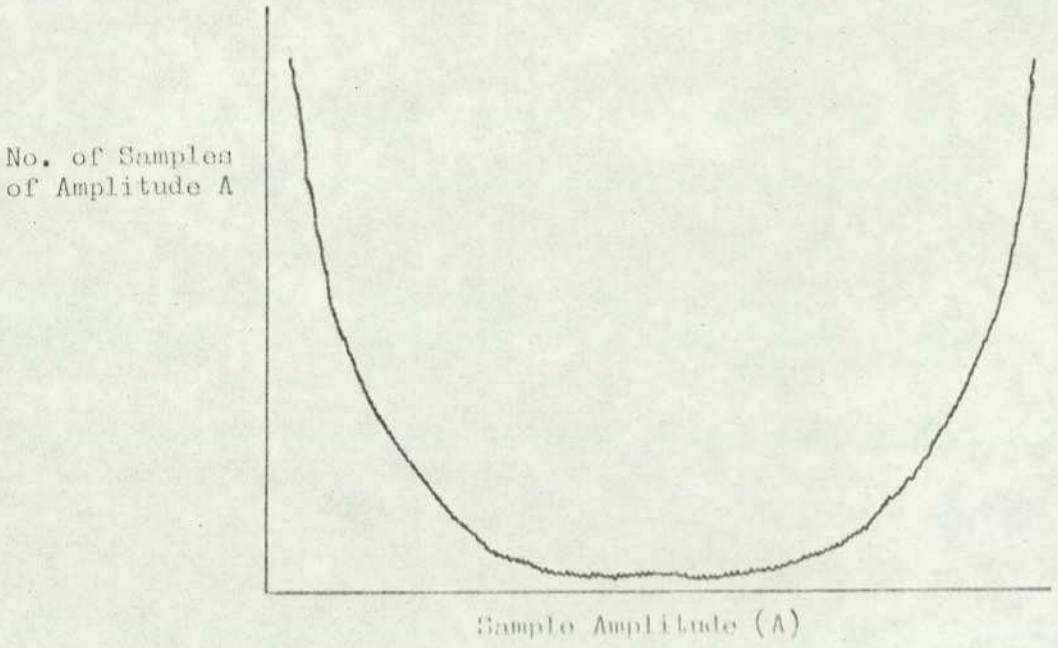


Fig. 69 (a). A Pulse Height Analyser Display. (Sine wave input)

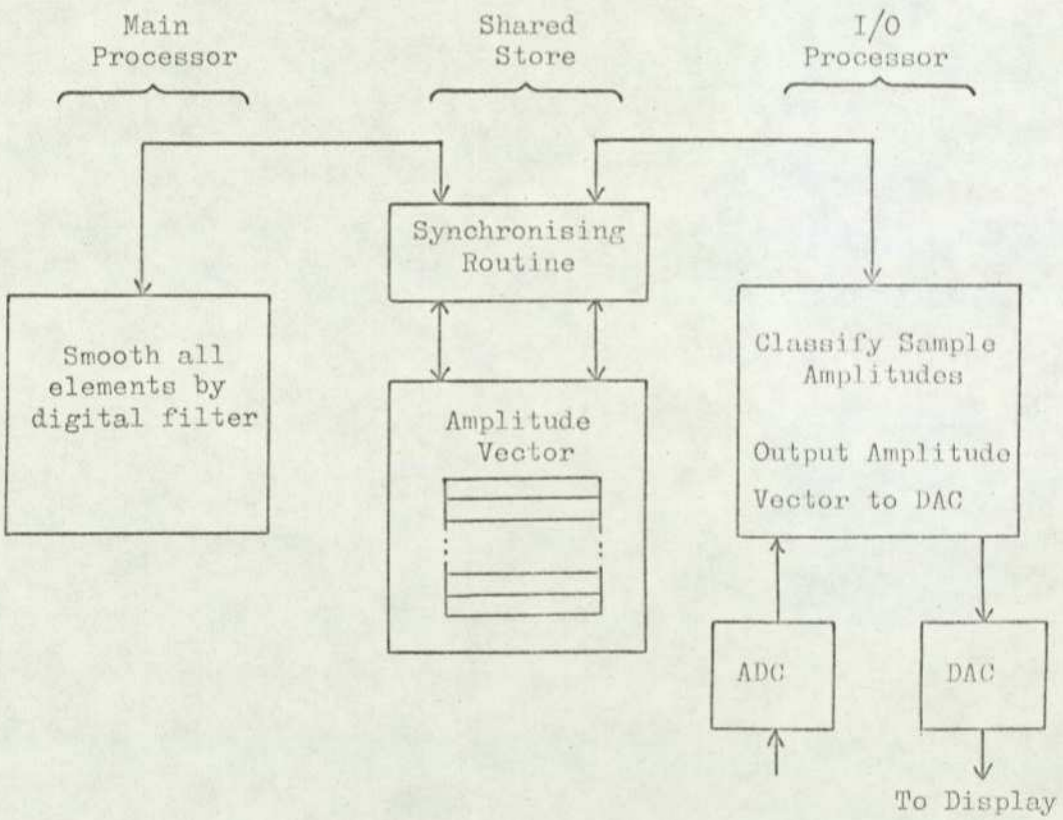


Fig. 69 (b). 'Division of Labour' in dual processor PHA.

Since the signal (or stream of pulses) is being continuously measured, the graph will increase until all points reach the highest possible value. To prevent this, it is necessary continuously to reduce the height of each point by a fixed fraction so that the average height of the curve remains constant. In this way, a continuous picture of signal amplitude distribution is presented. The need to acquire the signal, classify the samples according to amplitude, and display the distribution while simultaneously applying the reducing function to each graph point, leads to a natural parallelism in the application. In the demonstration, the I-O processor controlled the sampling, categorisation and display functions while the main processor applied the reducing function to the graph points. The division is shown in fig. 69 (b).

The I-O processor also drove a small visual display unit which displayed the number of samples that had been 'out of range'.

The demonstration formed a useful 'test-bed' for synchronisation and buffering techniques for dual processor configurations.

CHAPTER 7

INFLUENCE OF CPU ARCHITECTURE

7.1 Introduction

The choice of which type of micro-processor to use in the RSP is important, not just because it is the major component in the system, but because its choice dictates the whole system design philosophy. Since the range of commercially available micro-processors is expanding apace, it would be fruitless to review specific types since such a comparison would be out-of-date as soon as it was written. Instead, this chapter will discuss the various facilities and performance parameters of micro-processors, with particular reference to real-time systems. The discussion falls into two broad categories: instruction set properties and hardware capabilities (i.e. bus structure, interrupt system, etc.).

7.2 Instruction Set

In the broadest sense, an instruction set provides three facilities: a range of operations that may be performed, different types of operands which may take part in the operations, and ways of changing the order in which the operations are performed.

7.2.1 Operations

7.2.1.1 Bit and Shift Operations

It has already been stated (Chapter 4) that certain operations are of particular importance to real-time systems. The single-bit operations of set, clear, invert and test are important for flags, and the indivisible 'test and set' operation is essential for realising 'locks' to synchronise processes in multi-processor systems.

A wide range of shift instructions is needed if efficient use is to be made of storage space. In data acquisition systems it is often necessary to form tables of data in which several fields are packed into a single word. If

efficient packing and unpacking is to be performed, then variable length shifts (i.e. shift any number of places in a single instruction) must be used.

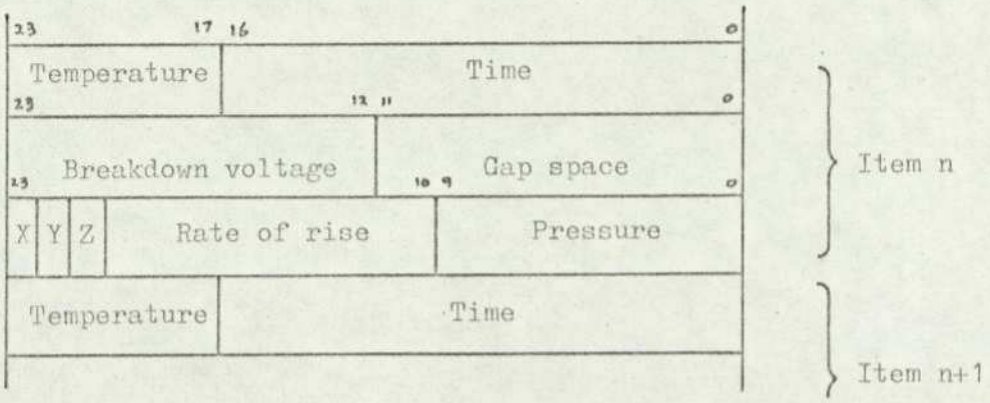


Fig. 71. A packed table in a 24-bit computer. (Example taken from dielectric breakdown research in the Author's laboratory)

Fig. 71 shows an item of data in a table stored in a 24-bit machine. Each item of data occupies three words which contain six different size fields and three single-bit flags (X, Y and Z) to indicate yes/no conditions about the data being acquired. A simple updating operation on this table might be "set flag Y and increase the voltage field by an amount V". In a processor with the instructions described above, this could be performed in four operations:

```
Load V
Shift left 12 places
Add into word 2
Set bit 22 of word 3.
```

In a processor with only a single place shift and no bit operations this would become:

```
Load a counter with 12
Load V
Shift V left one place
Decrement counter
Repeat if counter not = 0
Add into word 2
Load value 01000 . . . 000
OR into word 3.
```

The other important properties of shift instructions are their ability to deal with two's complement numbers without changing their signs. The so-called 'arithmetic shift' in which the sign bit is maintained for right shifts, and an overflow indicated if the sign bit changes during a left shift, is useful. Some processors use a simpler system in which the programmer has the option of filling the most significant end of the word with 0's or 1's during right shifts. This necessitates the original value of the sign bit being established by a test instruction before the shift.

Cyclic shifts (rotate) do not seem to be widely used except to swap bytes when two bytes are stored in a 16-bit word. This is often included as a separate instruction. Double-length operations are invaluable especially for double precision multiplications and divisions. Where a shift facility exists in only one register (e.g. the accumulator), it is extremely inefficient to implement double-length shifts on a machine which does not have an instruction to cover this facility.

In the foregoing discussion it has been assumed that the number of places to shift or the position of the bit to be tested or operated upon, is known when the program is written. If, however, this is to be a variable calculated by the program, then an instruction which takes the shift length (or bit position) from a register, is needed. Simulating this instruction with simpler operations is difficult and usually involves putting a single place shift in a loop and loading the loop counter with the variable. A similar arrangement can be used to bring a bit to a fixed position. The alternative is to modify the instruction while the program is running. Apart from the obvious dangers of this approach, it makes the program unsuitable for storage in ROM.

7.2.1.2 Arithmetic Operations

Double precision operations are facilitated by an instruction which increments a register if the 'carry' condition flag is set - the 'add in carry' instruction. This is used to transfer the carry from the least significant pair of words to the most significant pair. The F100 has a 'multi-length flag' which, when set, causes the carry flag to be added in when an operation is performed. This is inefficient since the programmer must ensure that the flag is clear at all other times.

No mention has been made of normal logical and arithmetic operations, since it is reasonable to assume that all processors have a good range of these. The one exception concerns the subtract operation which is usually only provided in the form A = A - B whereas A = B - A might also be useful. The F100L only has the facility to subtract the accumulator from another operand; not the inverse. This is inefficient and inconsistent with the idea of using the accumulator as the main working register.

Few micro-processors presently available have the facility to do two's complement multiplication and division. The multiply/divide facility that exists in some processors is an unsigned one that needs 'embedding' within software to restore the correct sign to results. Thus the actual multiply time is many times greater than that quoted in the manufacturers' literature.

7.2.2 Operand Types

This section considers such facilities as: the number of operands referenced in a single instruction, the addressing modes that may be used, the range of locations that may be referred to and whether constants can be included in operands.

7.2.2.1 Number of Operands

The number of operands that can be referenced in a single instruction is often referred to as the number of addresses, e.g. a two-address machine. The most general form of operation is of the form A = B + C thereby implying three operands: two for the source data and one for the result. Three-address machines are those which allow this type of operation, i.e. three distinct addresses are defined for each instruction. Two-address machines confine the user to operations of the type A = A + B that is, the result is returned to one of the locations from which the source data came. One-address machines have a single working register (e.g. the accumulator) and all operations are performed between this register and the single specified operand. The result is usually returned to the accumulator. That is, operations are restricted to the form X = X + A where only the location of A may be specified.

The foregoing applies to all simple arithmetic and logical operations but for conditioned jumps and shifts, some of the operand fields will be used to specify jump lengths, shift lengths, etc. It is therefore reasonable to consider operations of the type 'A becomes B shifted left C places' as being 'three addresses', although A and B are the only addresses specified. C is used to hold the number of shift places. Similarly one address field may be sacrificed to hold a jump displacement for a conditional relative jump.

It is clear that a three-address machine will require fewer instructions to perform a given task, but it will also use those instructions inefficiently as not all operations are inherently 'three-address' anyway. To obtain a crude estimate of the efficacy of different operand referencing systems, some programs were re-written for a variety of hypothetical processors. The programs were of a character-handling type and were originally written for a three-address machine, the FM1600B. They were re-written for a two-address machine with the same operation repertoire, and similarly for a one-address processor where all results were returned

to the accumulator. The programs were also written for a one-address machine such as the F100L where results may be returned to the accumulator or to the referenced location. For example, if it is desired to add R1 to R2 and put the result in R3, the processors would handle it thus.

Three address	R3 = R1 + R2
Two address	R2 = R2 + R1 R3 = R2
One address ordinary or F100-type	Acc = R1 Acc = Acc + R2 R3 = Acc (Acc = Accumulator)

Whereas R4 becomes R4 plus R5 could be performed as:

Three address	R4 = R4 + R5 (one field wasted)
Two address	R4 = R4 + R5
One address	Acc = R4 Acc = Acc + R5 R4 = Acc
F100-type	Acc = R5 R4 = Acc + R4

The results of the programs written are shown below.

Program Number	Number of Instructions				Number of Instructions (Normalised)			
	3 add	2 add	1 add	F100	3 add	2 add	1 add	F100
1	35	43	69	71	1.00	1.22	1.97	2.02
2	39	44	75	66	1.00	1.12	1.92	1.69
3	37	46	72	70	1.00	1.24	1.94	1.89
4	29	36	57	53	1.00	1.24	1.96	1.82
5	51	62	98	94	1.00	1.21	1.93	1.85
TOTAL	191	231	371	354	AV.	1.20	1.94	1.85

These results are not to be taken as accurate since they are too small for statistical validity and are not based on real-time programs. However, they show reasonably consistent results indicating that a two-address program takes about 20% more instructions than the three-address, while the one-address machine takes about 95% more program. The F100-type one-address machine is slightly better at about 85% more instructions.

This would seem to indicate that while the one-address machine is at a considerable disadvantage with respect to the three-address one, the two-address machine suffers only slight lengthening of the program.

These figures must be taken in conjunction with the number of registers which can be accessed by each address. Two- and three-address machines usually have a bank of fast registers, not in the main store, which can be accessed directly. Other locations must be accessed by index registers. One-address machines often allow a large number of main store locations (e.g. 2,048 in F100L) to be used. If the number of fast registers in a two- or three-address can be large enough to allow storage of all working registers and intermediate results, then execution speed will be increased considerably. This is because it is not economically feasible to have the fastest available circuits in main store, and the bus protocols will slow down the data transfers anyway. The internal registers can be fast, high power devices with no complicated accessing procedures.

The limiting factor on the number of registers is usually the number of bits needed to address one individual register. Thus the number of addresses in the instruction, the number of registers in the processor, and the wordlength, are inter-related.

A brief analysis of some assembly language programs written for the FM1600B was undertaken to determine how the operand field are used. With the exception of unconditional jumps and subroutine calls, all FM1600B instructions contain three 5-bit operand fields and a 9-bit opcode. The operand fields may be used to address the internal registers (of which twenty-three are general purpose), as a signed displacement for relative jumps, as a numerical constant, as a number of places to shift or a bit significance. The percentage use of such fields is shown below.

Two operand fields, one unused

2 registers	19%
1 register, 1 numerical constant	25%
1 register, 1 jump displacement	3%
1 register, 1 shift/bit significance	14%
Total 2 fields	61%

Three operand fields

3 registers	1%
2 registers, 1 numerical constant	4%
1 register, 1 jump displacement, 1 shift/bit	9%
1 register, 1 jump displacement, 1 constant	9%
Total 3 fields	23%

Miscellaneous (e.g. jumps, subroutine calls, etc.) 16%

These figures show:

1. Only in 20% of instructions do the operand fields refer only to registers.

2. 21% of instructions use a field as a jump displacement.

3. 29% of instructions use a field as a numerical constant.

4. 23% of instructions use a field as a shift/bit significance.

This means that the ability of the fields to hold jump displacements, bit significances and constants is of as much importance as the ability to refer to one of a number of registers. Four hypothetical three-address processors are now considered in the light of these requirements.

	Number of Registers	Bit Significance	Jumps or Constants		% Jumps
Machine 1 (16-bit) 3-bit operand fields 7-bit opcode	8	0 - 7 (Insufficient)	+4	+3	50%
Machine 2 (20-bit) 4-bit operand fields 8-bit opcode	16	0 - 15 (Insufficient)	+8	+7	68%
Machine 3 (24-bit) 5-bit operand fields 9-bit opcode	32	0 - 31 (Sufficient)	+16	+15	75%
Machine 4 (30-bit) 6-bit operand fields 12-bit opcode	64	0 - 63 (Sufficient)	+32	+31	94%

The percentage of jumps accommodated by a given size displacement field is based on a method of estimation to be described later (7.2.3.1).

From these figures it can be seen that 3- and 4-bit operand fields are insufficient to allow all bits in their respective processor words to be defined, and do not allow a high proportion of jump displacements to be accommodated. However, five and six bits are adequate in both these respects. Thus the three-address structure is applicable only to word-lengths of about twenty-four bits or over. Since the present limitations of fabrication and packaging technology preclude such long wordlengths in integrated circuit manufacture, it is reasonable to conclude that the three-address structure is not relevant to a discussion of single-chip micro-processor design.

Experience with a 16-bit micro-processor (CP1600) which has eight internal registers (of which six are general purpose) suggests that this number is too small, while the FM1600B with twenty-three general purpose registers has more than enough internal storage for most current variables. Thus it may be reasonable to conclude that a two-address machine with twelve to sixteen general purpose registers is an optimum structure for a multi-address micro-processor of small enough wordlength for integration in a single chip.

7.2.2.2 Addressing Modes

The addressing modes of a processor are the ways in which it interprets a field in an instruction as an address in main store. The following conventions are adopted in describing addressing modes:

	x	means contents of location x
	(x)	means the contents of the location whose address is in location x.
If	R	is the value of the addressing field
	R	is the direct addressing of locations
	(R)	is the indirect (pointer) addressing.

These are the two basic forms from which all others can be formed providing the program can manipulate the register R.

Extensions are:

(R), R + 1	Indirect with auto-increment
(R), R - 1	Indirect with auto-decrement.

This pair can be used for moving pointers up and down tables. If the incrementing occurs before the register is used as an address and the decrementing occurs after, or vice versa, then the pair of modes can be used to manipulate a stack (last in, first out) structure.

The other basic mode is (R + D) where D is a displacement for the location held in R, stored as another field in the instruction.

These represent the basic addressing modes with most processors providing all of these, although only the first two are necessary. The F100 does not have the last facility which is a serious omission.

The provision of an adequate range of addressing modes is important to efficient programming. Some processors (ref. 62) have attempted to optimise these modes to correspond to the data structures found in high level languages. A common practice among micro-processor designers is to restrict certain addressing modes to certain registers instead of having all modes available on all registers. This saves instruction length but requires great care and forethought in programming.

7.2.2.3 Constants

Constants are fixed numbers which have to be introduced into the instruction stream. This is usually done as a two's complement number occupying a field in the instruction which would otherwise be used for an operand address. Alternatively the number is put as the second word of a two-word instruction. The length of a constant field could be considered in the same way as the size of relative jump displacements is estimated in (7.2.3.1).

However, such constants are of four distinct types:

1. Small integers used as loop counters or in general arithmetic operations.

2. Eight-bit bytes used in character handling routines.

3. Masks of 0's and 1's for packing and unpacking numbers.

4. Addresses used to load index registers.

As a broad generalisation, the first type could largely be accommodated in four bits while the second type requires eight. Masks will generally be the same length as the processor word while the size of addresses will depend on the amount of store in the system. Generally the last type will be as long as the processor word.

This means that it is desirable to have at least a small (four-bit) field in the instruction which can be used for constants, and one as large as eight bits would be useful. Masks and addresses must generally be included as a second word.

The F100 suffers considerably from not having any constant facility in one-word instructions. All numbers must be entered as the second word of a two-word instruction. Thus even simple operations such as x = x + 1 must be realised as:

```
Acc = 1 (two-word instruction)
x   = x + Acc.
```

7.2.3 Means of Altering Instruction Flow

Instruction flow is controlled by conditional and unconditional jumps. These instructions are crucially important to efficient programming (e.g. in a large program for a three-address machine, nearly 35% of instructions were jumps, most of them conditional). The two factors affecting the efficiency of methods of transferring control are the way the jump address is specified, and the range of tests upon the results of which the jump may be made conditional.

7.2.3.1 Relative or Absolute Jumps

Relative jumps have a signed quantity representing the displacement of the jump destination from the instruction position. This means that, for a finite number of bits in the displacement field, jumps are limited to locations

within a fixed distance of the jump instruction. Many processors adopt a field size of eight bits meaning that jumps of up to 128 places forwards or up to 127 places backwards are possible. The FM1600B, however, adopts a smaller field of only five bits. In order to estimate the required size for the displacement field of a relative jump, statistics concerning jump length (i.e. the distance between the jump destination and the jump instruction) were obtained. This was done by modifying an F100 simulator which ran on a PDP11 (ref. 63). The occurrence of jump types and their length were recorded by running a large suite of floating point programs through the simulator.

Fig. 72 shows a graph of the number of bits in the displacement field versus the fraction of jump instructions that could be accommodated. Two's complement representation of displacement is assumed. The graph shows that 54% of instructions could be held in four bits, while 95% could be held within seven. Although these figures apply only to the F100 processor, it is possible to project what they would have been for other machines. It is reasonable to assume that a jump length represents a certain amount of processing which has to be repeated (backwards jump) or avoided (forward jump). Since the amount of processing is dependent on the program, not the processor, the jump length can be calculated for another processor from the knowledge of how many instructions it takes to complete a given task, compared with the machine on which the original statistics were taken.

Figures have already been presented (7.2.2.1) which suggest that the comparative program length for a three-address, two-address and the F100 processor are 1, 1.2 and 1.85 respectively. It would therefore seem reasonable that the graph of fig. 72 would be displaced towards the vertical axis by an amount $\log_2 (\frac{P\ F100}{P\ PROC})$ bits where $\frac{P\ F100}{P\ PROC}$ is the comparative program length for the processor, - 1.85 for the three-address machine and $\frac{1.85}{1.2} \approx 1.54$ for the two-address machine.

These graphs are also shown in fig. 72. Approximate results are tabulated below.

Processor Type	Length of Displacement Field (Bits)				
	4	5	6	7	8
F100	54	70	75	95	97
Two-address	65	73	88	96.5	97.5
Three-address	69	75	94	97	98

Percentage of jumps accommodated

Thus it can be seen that, while eight bits covers most jumps; for a multi-address machine fewer bits could reasonably be used.

Absolute jumps simply specify the whole destination address and can, therefore, always reach the required location in one instruction (unless a paging system restricts addressing). This is wasteful if most jumps are only a few places forwards or backwards, but it is a simple and quick type of jump since no arithmetic operation is needed to form the new address. Absolute addressing is particularly wasteful for small wordlength processors where there are not enough bits to address the whole store in one word. However, in larger wordlength machines, it is possible to address the whole store and still have bits left over in one word (e.g. FM1600B has a 24-bit word but is rarely fitted with more than 64K of store so it has at least eight bits unused - when addressing any store location).

Note

The F100 is an appropriate machine on which to compile statistics of the above type. This is because it has a simple jump structure where all jumps have a 15-bit address specified, i.e. an absolute jump. This means that the true jump length implied by the program can be measured. In machines where there are several jump types (e.g. with relative jumping an absolute jump is still provided for when the

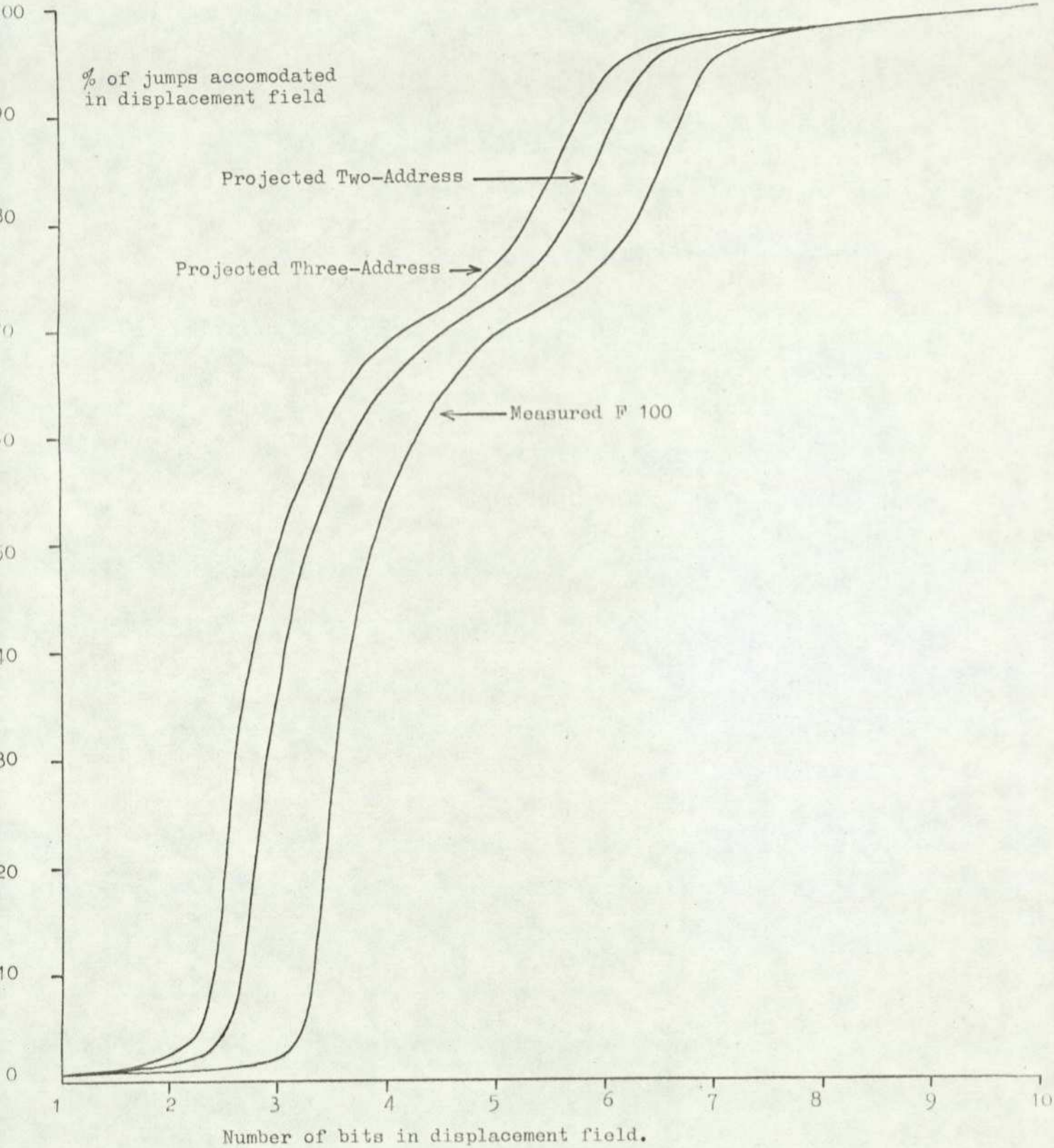


Fig. 72. Distribution of jump lengths for F 100 (measured) and two- and three-address machines (projected).

jump distance is out of range) the pattern of jump length becomes distorted by the constraints of the machine. An example is the FM1600B where conditional jumps which are out of range of the displacement field are realised as conditional skips around absolute unconditional jumps. This means that a number of short relative jumps are compiled which are not implied by the processing operations but only by the shortcomings of the processor. Such machine nuances subtly distort the instruction statistics.

Addendum

Much time was spent investigating ways of collecting instruction occurrence statistics on both the micro-processor and the main (FM1600B) computer. There were seen to be three ways of collecting dynamic information on instruction occurrence:

1. Writing a simulator for a processor and building in a program to compile the figures required. This method has been mentioned and was the only one to give results.

2. Building a piece of equipment exclusively to monitor instruction types which would connect to the computer being measured via the main data/address bus.

3. Making the computer interrupt itself after every instruction so that the type can be logged.

(1) was convenient because the F100L simulator had already been written and was easily modifiable. The method had certain disadvantages: input-output programs cannot be monitored since the I-O system of the simulator is not compatible with that of the real machine, nor could it easily be made so.

(2) would be a very attractive idea but for the hardware complexity required. Programs can be monitored in real-time so that I-O routines can always be examined. However, the high speed that real-time monitoring implies means that all the information must be logged into a fast memory for examination later. Thus a large, high-speed memory is required. The obvious idea of making the monitoring device another processor is generally not feasible since an instruction rate five to ten times faster than that of the monitored processor would be necessary.

(3) was the method tried on the FM1600B. This computer has a 'program protection unit' (ref. 64) which interrupts the processor if program limits are violated or after a certain number of instructions have been executed. This last facility was used to make the unit interrupt the processor after every instruction. The interrupt program could then look back into the link stack to find out the main program address at the time of the interrupt. This allows the instruction type to be identified and logged. After much programming effort devoted to persuading the unit to function, it was discovered that program execution could not always be resumed properly after an interrupt. It was finally established that the unit sometimes inhibited the calculation of the next instruction address so that, on these occasions (notably conditional jumps), correct program execution could not be resumed after the interrupt.

It was found easier to cause interrupts after each instruction on the F100 since a signal, IR_d , is available which pulses once during the 'instruction fetch' phase of an instruction. This can easily be connected to the interrupt line to cause interrupts after each instruction. The lack of suitable I-O devices and suitably large stores prevented this being used to any useful extent.

A novel method of logging instruction types which may be worth considering is the use of a pulse height analyser (PHA) (ref. 61). Such a device (which is available on-line in the author's laboratory) compiles histograms of the occurrence of the amplitudes of pulse presented to it. These pulses can be derived, via suitable gating, from the data/address bus of the processor and are converted to amplitude pulses by a simple D-A converter. The PHA available has a maximum pulse rate of about 80K pulses/second, i.e. one every 12μS. It was thus about four times too slow for the F100 processor. Slowing down the processor by reducing its clock rate proved impossible because the dynamic memory refresh was derived from the clock, and data was not retained properly at lower clock rates. Otherwise this seems to be a convenient method of instruction type logging.

7.2.3.2 Range of Conditions Tested

The simplest conditional jump system used by many microprocessors (including F100) is to have a status word with bits set to indicate carry, zero result, overflow, etc. All conditional jumps can then be made as bit tests on the status word. The condition flags are set during the last operation before the test. An instruction is usually included which sets up the flags, but stores no result, so that operands can be tested without performing any operations. This system means that two instructions are always needed to test a value, but it is reasonable to assume that the first operation will often be needed anyway.

A more efficient type of conditional jump system is one in which an operation can be combined with a test. The FM1600B includes a wide range of such instructions with the option of storing or not storing the operation results. This leads to efficient programming as the number of conditional jumps is a large part of the total program.

7.3 Hardware Capabilities

The hardware aspects of CPU architecture include bus structure, input-output system (including DMA and program interrupt facilities) and the provision or otherwise of monitoring facilities. Since DMA and interrupt systems have been discussed already (4.3.3.1), and their effects on the interfacing problem are discussed in 8.2, this section will only summarise the points brought out elsewhere.

7.3.1 Bus Structure

The way in which data and addresses are transferred between the processor and the memory and I-O devices differs from processor to processor. The two main divisions of types are combined or separate data and addresses, and asynchronous or synchronous transmission of information.

7.3.1.1 Combined or Separate Data and Address Buses

The advantage of transferring addresses and data on the same bus is that the total width of the bus is reduced. This is important from the point of view of the CPU chip where the number of pins is likely to be critical. It is also important if the bus is likely to be a long one, with peripherals and memory blocks distributed along its length. In this case the cost of cabling (probably twisted pairs) and bus drivers and receivers is reduced in proportion to the reduction in bus width. The disadvantage of such a system is that the two signal types must be separated before they can be used by devices. This is either done by circuits close to the CPU and the separate data and address buses run to all devices, or all devices have their own interface converters (e.g. interface sets) to perform the decoding.

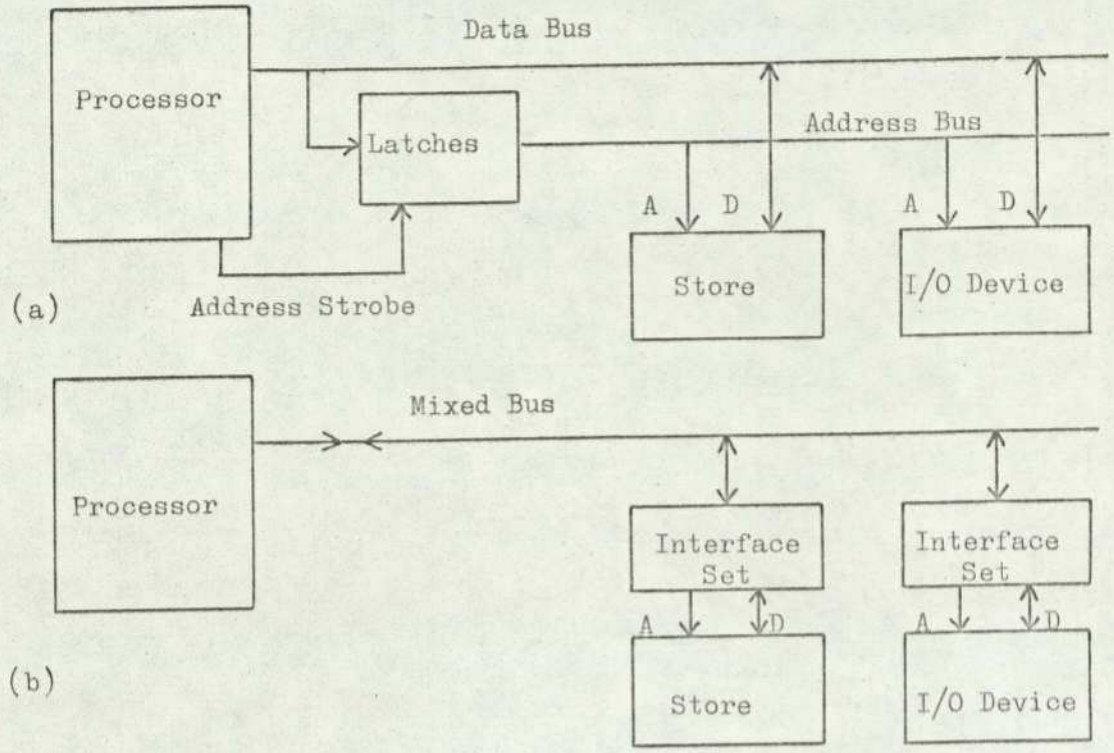


Fig. 73. Different approaches to mixed-bus systems.

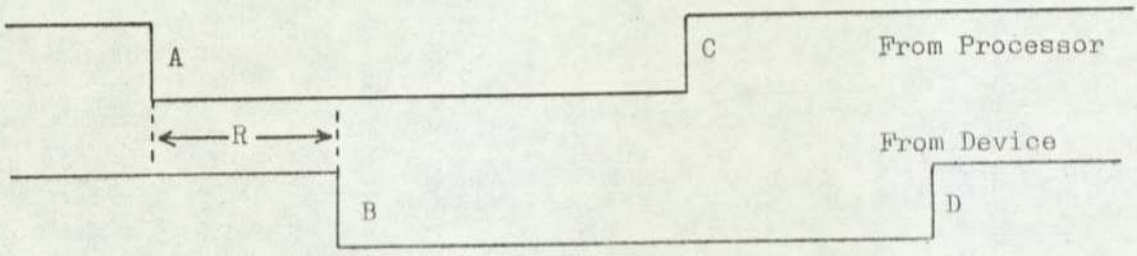
Fig. 73 shows the two combined bus systems. (a) uses less logic since the bus decoding is only done once, whereas (b) does it at every device interface. Broadly (a) is most suitable for minimum logic systems where the whole processor is contained within one physical unit. Otherwise it represents the worst of both alternatives, combining the slow speed of a mixed bus with the high cabling costs and unreliability of separate buses. System (b) uses more logic (although the device interface could be realised as one LSI circuit) but maintains a very simple arrangement with one bus. Thus this type of bus is more suitable to a distributed system with long cabling runs.

7.3.1.2 Asynchronous or Synchronous Buses

A synchronous bus is one in which all data transfers are identified by the time at which they occur. Usually the processor clock signal is divided down to produce several time slots with one type of transfer (address, operand, instruction, etc.) allowed at each time slot.

As already mentioned (5.2.1) this is a simple system which requires little logic to sort out transfer times. It is also potentially very fast, since logic delays can be mitigated by anticipating the next time slot. The disadvantage of this type of transfer is that the speed of the transfers is limited by the timing delays on the bus. Thus, for long buses, slow signalling rates must be used. Also there is no facility for the processor to check that one transfer is complete before beginning the next. This means that it is inefficient to mix memories of widely different speeds in the same bus, since the timing would have to be adjusted to cope with the slowest one. A variation (semi-synchronous buses) interposes 'wait' states to cope with slow devices.

Asynchronous buses are based on 'handshake' sequences of signals and have no timing rules. The basic handshake sequence of two signals is shown in fig. 74.



Edge A Processor puts transfer address on bus.

Edge B Device recognises its own address and responds.

Edge C Processor puts data on the bus.

Edge D Device acknowledges receipt of data.

(R is the response time of the device. If this is excessive the processor will assume a device failure)

Fig. 74. A 'handshake' sequence for controlling data transfers.

It can be seen that all transfers are identified by edges on the control lines. Clearly the device can take as long as it likes to acknowledge either address or data transfers without the system breaking down. Alternatively the processor

can time the acknowledgement delays and signal an error if these become excessive. Thus the system is capable of detecting faults in devices or their interfaces. The interface logic can be complex, as the edges, and the order in which they occur, must be detected. The absence of timing rules also means that memories of different speeds can be mixed on the bus with each operated at its maximum speed.

The choice of synchronous or asynchronous bus resolves itself as the choice of mixed or separate buses did in (7.3.1.1). That is, the synchronous bus is a simple system, with no error-detecting capabilities, which can be fast if bus lengths are short. The asynchronous system is slower, is not susceptible to timing delays, and has the capability of detecting errors within the system. Thus the latter is more suitable for distributed high reliability systems whereas the former will be used to greatest advantage in small high speed systems.

7.3.2 Interrupt and DMA Structures

The ideal capabilities of an interrupt system have already been stated (4.3.3.1) as:

1. The ability to accept a number of different interrupts, arbitrate between simultaneous requests, and steer each interrupt to its own address.

2. The ability to save the machine status upon accepting an interrupt, and restore the status so that program execution can be resumed correctly after the interrupt.

3. The ability selectively to lock out individual interrupts.

Most processors have the first two facilities, although most use a vectored interrupt system. This means that interrupts are not steered directly to an interrupt program but rather to a block of locations where jumps to the program can be stored. This increases interrupt response time and complications can arise if the block is in a fixed and inconvenient position in store. Most microprocessors have a simple interrupt lock-out facility, but some have a register which can be loaded with a value representing the lowest priority interrupt which will be allowed. Requests from lower order interrupts will be rejected. This is a most useful facility since it allows very important interrupts (e.g. power fail) to be enabled all the time.

A minor point worth noting is whether a processor latches interrupts or not. This is important since, if an interrupt request occurs while interrupts are locked out, it will be lost altogether in an unlatched system, while the latched arrangement will give the interrupt as soon as interrupts cease to be locked out. Under different circumstances both these methods have advantages.

DMA facilities include: '

1. The ability of a device to gain control of the bus and supply data and address information for store transfers; and,

2. The ability of the DMA device to count transfers so a fixed number can automatically be effected.

The first facility is universally available, but transfer counting is not so frequently found. Simultaneous DMA requests must be arbitrated upon as with interrupts.

7.3.3 Monitoring Facilities

The speed with which a system can be installed and tested depends mainly on the range and power of the test and monitoring facilities available. It is usual for the 'debugging' phase of a program to be many times longer than the time taken to write it. In non real-time operations, testing can be done largely by software through the use of simulators, tracing programs, break-point routines, etc. However, in real-time programs, particularly where hardware and software are being developed together, these techniques are of little use. In these cases, one must fall back on the monitoring facilities built into the processor or available as separate hardware.

The 'front panel' is the commonest monitoring device, consisting of a row of lights and switches by means of which registers can be loaded and examined, and the program may be stepped through one instruction at a time. It is common practice in micro-processor design for the front panel to be an ordinary peripheral driven by software resident in the system. This is a very elegant idea but it has difficulties when hardware units are being tested in conjunction with their driving software. In earlier (discrete) processors, when, for example, the operator wished to examine the contents of register 4, he would select this on the front panel. This would connect the indicator lights to a set of wires from the hardware which represented register 4. Thus the monitoring was purely passive. The micro-processor technique would be to interrupt the processor to a program which reads the front panel switches. It then decides what action is necessary and writes the required information to a peripheral device - in this case, the indicator lamps. Thus the monitoring operation involves the processor in a change of state. This can destroy the test sequence as far as the hardware is concerned, since its operation may be synchronised with individual machine states. It is thus very difficult, with a program-driven front panel, to test hardware which is intimately connected to the processor.

7.4 Software Aspects

The foregoing has assumed that programs for a real-time system will be written in a low-level language by a programmer who will take care to exploit the processor features to get maximum performance from the system.

While this will be true in a large number of cases (the great majority of real-time programs for micro-processors are written in assembly language, and this will continue to be the practice for time-sensitive parts of programs for some years to come), it is evident that the use of high level languages and operating systems will effect the importance of the CPU characteristics discussed. This section attempts to find whether the CPU aspects mentioned earlier in the chapter are made more or less relevant when considered in conjunction with the sort of software which is likely to run on the system.

7.4.1 Software Overheads imposed by the Register Structure

It has already been stated that provision of a large number of internal registers can considerably enhance processor performance by speeding up instruction execution (7.2.2.1). This is true once a problem is 'set up' within the processor, that is, the various variables are in the working registers. However, if problems have to be changed quickly (e.g. a subroutine call or handling an interrupt), a large number of internal registers can be a handicap.

In a single-address machine (e.g. F100L) all operations are performed between the accumulator and a store location. Since a large number (2,047) of locations can be accessed directly, it is possible to allocate a set of locations to each subroutine in a program. This means that only the accumulator has to be saved and restored when a subroutine is called or an interrupt accepted. Thus the single-address system provides very fast 'context switching' although operations after the switch will be slower as already outlined.

A multi-register machine must store all the working register in main memory, and unstore them when a subroutine or interrupt is encountered. Experience with a multi-register machine suggests that the number of working registers that must be saved when a subroutine is called is usually four to six and very rarely greater than eight, including index registers. Many multi-register machines have duplicate banks of registers, so that swapping over one set of registers for the other creates a context switch without the need to save working values. This is useful for ensuring a fast interrupt response, but is of no use for subroutines, since it can only occur once and thus precludes nesting of routines. The same constraint applies to interrupts.

A variation on this idea is provided by one processor (TMS 9900, ref. 65) which has a two-address 16-register architecture. However, the registers do not exist within the processor but are located in the main store. A base address register identifies the start of the 16-register block. Each subroutine (or interrupt) loads the base register with a different value and automatically obtains sixteen 'new' registers for its use. This is very efficient since it allows nested subroutines and interrupts. However, the main advantage of an internal register bank (avoidance of lengthy main store access cycles) is lost altogether since at least three store cycles (one instruction and two operand) are required for each instruction.

Thus, although the F100 and TMS 9900 storage allocation systems seem to differ considerably, they are, in fact, identical. The only difference is that F100 allocates storage at assembly time while the TMS 9900 does it while the program is running.

7.4.2 Effect of High Level Languages

While it is true that compilers for high level languages can be written which use multiple internal registers efficiently (ref. 66), it is generally the case that very few registers will be used and these generally for fixed

purposes. Thus, although the multiple internal register type of processor is potentially more efficient than a single accumulator type, it is doubtful whether most compilers will exploit the potential. In the light of this, the simple one-address machine of F100 type looks a better proposition if used with a high level language such as CORAL 66. Its instruction set is well suited to this and many of the operations mentioned (e.g. table packing and unpacking) which are specifically included in CORAL, can be efficiently implemented by F100.

7.5 Summary

This chapter has attempted to consider the influence of the processors' characteristics on the performance of a real-time system such as the RSP.

The main conclusions are summarised below under their various headings:

7.5.1 Instruction Set

Single-bit operations are important for flag manipulation, and an indivisible test and set operation is vital for synchronising two processors. A wide range of shift instructions is needed for efficient packing and unpacking of tables. Multiply and division are important operations but are invariably needed in two's complement form, which, if not available, must be realised in software.

The number of operands which can be accessed in a single instruction has an important effect on program length. Some sample programs suggest that a one-address machine may use 95% more program than a three-address, while a two-address uses 20% more. This suggests that the improvement from one-address to two-address is significant while that from two-address to three-address is small. The advantage of having a number of fast registers is underlined, and it is suggested that eight is insufficient.

The importance of addressing modes is mentioned as is the ability to include at least small constants within a single-word instruction.

Relative jumps are more efficient than absolute jumps, but the ideal length of the displacement field is difficult to determine. Results suggest that an eight-bit field is capable of holding over 97% of all jumps. Three-address machines are seen to need only a six-bit field to express a high proportion of jump displacements. The suitability of some machines for compiling jump length statistics is discussed.

7.5.2 Hardware Capabilities

Combined data and address buses provide an efficient way of reducing the number of pins on a CPU device and of cutting bus cabling costs. This in turn improves reliability. Such systems need LSI interface components to regenerate separate buses for each device else the advantages of the system are lost. Separate buses are faster for small systems where the large number of bus lines is not a disadvantage.

Synchronous buses are faster and simpler than asychronous ones but do not have a checking facility to enable fault detection. The speed of devices on the bus must be closely matched. Asynchronous buses allow devices of different speeds to operate on a bus, and give a fault detection facility. These are more appropriate to reliable systems.

Interrupts and DMA transfers are almost standard facilities on most processors. The ability to count DMA transfers and the facility selectively to lock out interrupts are useful.

7.5.3 Software Aspects

Single-address machines are easier to write efficient compilers for, whereas multi-register machines are unlikely to be used efficiently. Thus, if high level languages are to be used, the single-address machine is more attractive than earlier comparisons would suggest.

CHAPTER 8

THE PERIPHERAL CONTROL PROBLEM

8.1 Introduction

Much of the effort involved in the design of real-time signal processing systems based on computers or micro-processors is expended on the interfacing of devices, transducers or instruments, to the processing system. In this context 'interface' means the circuitry which conditions signals (by altering their voltage levels, timing relationships, formats, etc.) from one device so that they are acceptable and meaningful to another.

8.2 Requirements of the Interface

An interface must facilitate the fast and accurate transfer of information between an external device and the processor or its store. This information is conveyed not only by data but also by control and status signals, which enable the processor to control the device and the device to make known to the processor the occurrence of external conditions. By looking at the place of the interface in the overall processing system, one can make these general requirements more specific. Most computer systems (and all the more powerful micro-processor systems) are organised as a collection of units which communicate via a universal data bus. Input-output (I-O) devices are treated as storage locations within the 'address-space' of the processor. This 'memory-mapped' I-O system means that I-O devices must be connected to the main processor bus, and thus need to control, or at least interpret, the bus signals.

Processors are invariably provided with an interrupt line, by which means current program execution can be suspended while a more urgent program is run at the request of the interrupting device. Also, a means is usually provided for a device with a high data rate to read from, or write to, the processor store without any processor intervention. In this direct memory access (DMA) mode the accessing device must control the bus in place of the processor.

The requirements of the device interface circuitry can now be stated in more detail:

1. To effect program controlled transfers of information (data, status and control) between the device and the processor bus.

2. To allow the device to interrupt the processor.

3. To take control of the device so that data may be transferred by DMA.

While not all devices will require all these facilities, it is reasonable to expect that high speed signal peripherals will need most of them.

8.3 The Interface Design Problem

The problem which confronts the system designer (especially in a research and development environment where a large variety of peripheral devices and processors must be interfaced, often for short periods only) is that each device requires a different piece of circuitry to be developed in order to connect it to a processor. If a device is to be shared between different processors, then it may need more than one interface. The interface development can thus take up a disproportionate amount of time. Several methods of avoiding this problem have been devised, and these will be discussed before proposing a further method.

8.4 The Present Range of Solutions

While it is true that an engineer experienced in the use of a particular processor system can devise interfaces for devices with considerable speed, manufacturers have tried various methods in an attempt to reduce interface development time.

8.4.1 Use of Standard Interface Components

In this method the manufacturer provides a set of standard components (originally printed circuit cards, now LSI circuits) which do all the basic control and interpretation of the bus signals. They may also contain interrupt addresses, DMA address counters, data latches and a variety of other useful components. The designer of a peripheral control unit (PCU) then only has to design that part of the circuitry unique to the particular device. The supreme example of this is the F100L where the interface set (ref. 21), three LSI circuits, contains all the bus control logic for address recognition, program controlled transfers, interrupts, direct memory accesses and a certain amount of system error detection. Thus the designer's load is considerably eased while operational speed is kept high since all interfacing is handled by hardware.

8.4.2 Software Handling of the Device

In this method the device is connected to the bus by the simplest of data paths and latches, often incorporated in a standard component. The only facilities provided are the means to write to, or read from, the device or its status/control word. All handling and control of the device are done by software, and interrupts and DMA are not allowed. Instead the processor must 'poll' all the I-O devices to see if one requires attention. The advantage of this system is the extreme simplicity and degree of standardisation of the hardware, and the implicit ease with which interface protocols can be designed in software. However, operation

is inevitably slow, and a large amount of processor time is occupied in continuously looking for requests from peripheral devices. This degrading of the processor performance is not acceptable in most signal processing applications, and it is only in low demand areas (particularly 8-bit microprocessor systems) that this method dominates. The situation can be improved slightly by the use of a separate processor for I-O operations but this brings with it the problems of dual or multi-processor operations.

8.4.3 Defining a Standard Interface for Instruments

Some manufacturers and international organisations have defined standard interfaces for instruments. This really solves no problems - it merely transfers the design effort from the system designer to the instrument designer.

Two of the most popular standards are CAMAC, much used in computerised nuclear instrumentation systems, and Hewlett Packard Instrument Bus (now IEEE Standard 448 - 1975) used with calculator controlled systems.

8.4.3.1 CAMAC

The CAMAC standard (ref. 82) defines a mechanical unit ('crate') as a rack capable of accepting one controller and up to twenty-four controlled modules. These modules share two (one read, one write) 24-bit data buses and a number of other bused control signals. Each module has two personal signal lines to the controller, one for demanding attention and the other for receiving orders from the controller. Data transfer rates up to one word per μS are possible with CAMAC.

8.4.3.2 HP-IB

The Hewlett-Packard Instrument Bus (ref. 83) uses a 16-line cable which connects all instruments and controllers in parallel. Eight of the lines form a parallel data bus, three more are status indicators and the remaining five are control.

Up to thirty-one 'device addresses' can be recognised (electrical considerations limit the number of devices in any one system to about sixteen) with each address being designed a listener, a talker or a controller, or any combination. Numerical information, usually transmitted as BCD digits, and characters can be transferred at rates up to 1MHz. This system is very easy to assemble and it is not necessary to have any controlling device at all - one talker and one or more listeners represent a minimal system. The controlling device is usually a programmable calculator although computer interfaces do exist for HP-IB.

8.4.4 Use of Autonomous Peripheral Processors

Some micro-processor manufacturers provide self-contained micro-computers designed to operate with the main processor, to act as programmable interface components. An example of this is the General Instruments PIC 1640 (ref. 22), a peripheral processor designed to operate with the CP1600. This processor has a 12-bit instruction with a powerful range of operations, including single-bit test and set operations and subroutine calling. Thirty-two 8-bit registers are included, some of which are accessible to the main processor and thus act as a communication area between them. Two bi-directional 8-bit ports communicate with the peripheral device, while the PIC is also connected to the main processor bus.

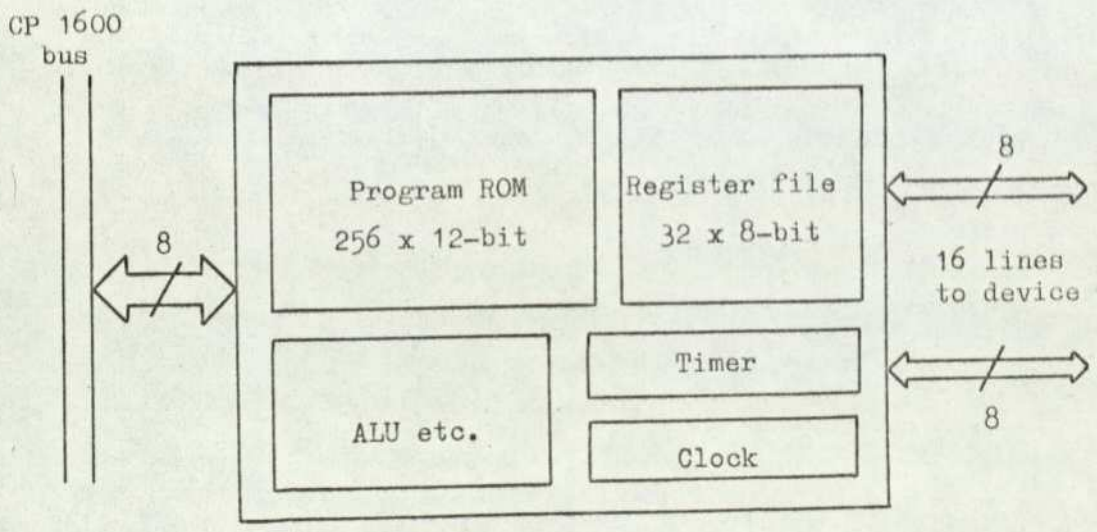


Fig. 81. The PIC 1640.

With such a system the interface is realised entirely in the software of the PIC, thus giving the advantage of standard hardware and ease of design without loading the main processor.

8.5 Disadvantages of Present Methods

In the context of a flexible instrumentation system, such as the Roving Slave Processor, it is important that processing performance should not be compromised any more than is necessary to give a system of integrity and flexibility. When interfacing a device for a particular function, one may be able to estimate maximum data rates, maximum response delays and acceptable timing inaccuracies, and design the circuitry to be no better than these. However, in a flexible system which will serve many purposes, one must make no such assumptions and one must, in general, assume that all the processing power of the system will be needed, at least sometimes.

Consideration of the instruction and store timings of a powerful micro-processor such as the F100L sets the order of magnitude of timing delays which are acceptable at the device interface. The F100L executes an instruction in 3 to 4μS and can accept DMA requests at a rate approaching 1MHz. Since typical peripherals might be D-A and A-D converters, which are readily available, with conversion times of 1μS and 5μS respectively, it seems reasonable to specify that interface delays must not exceed 'a few micro-seconds'.

Autonomous peripheral processors such as the PIC 1640 have an instruction time of 4μS, so the simplest program loop takes 30 to 40μS. Also, it only transfers 8-bit values, so that two cycles are required to transfer the ten or twelve bits typically used in A-D and D-A operations.

Software handling by the main processor improves the time response due to the increased processing power of the main device but has compensating disadvantages in that the main processor is halted while it is dealing with device handling. In particular, continuous polling of I-O devices by the processor is extremely time-wasting and considerably reduces the signal processing bandwidth of the system. Thus it can be seen that these methods, though attractive from the minimal hardware point of view, are an order of magnitude slower than required.

Clearly the bus control signals must be handled at 'hardware speed' by the device interface, and the use of standard hardware units is an attractive solution. Such units (e.g. F100 Interface Sets) will provide all the logic for program controlled transfers, DMA transfers and program interrupts. The problem then becomes one of controlling the device at sufficient speed while maintaining flexibility through the use of programmable logic.

8.6 A Programmable Interface Converter

In an attempt to combine the speed of hardwired logic with the advantages of programmability, the author designed and built a programmable interface converter to work in conjunction with the F100 interface sets. The design of the converter is described in "The Case for a Simple, Fast Programmable Interface Converter" presented at IERE conference on Programmable Instruments, Teddington, November 1977, and published in the proceedings. The prototype converter is described in detail in appendix II (a) while (b) investigates the speed limitations of such processors.

"The Case for a Simple, Fast Programmable Interface Converter" presented at IERE
conference on Programmable Instruments, Teddington, November
1977, and published in the proceedings.

8.7 Recent Developments

One of the alternatives mentioned in the foregoing paper, the use of a programmable logic array (PLA), has recently been developed into a workable device. A PLA is an array of AND and OR gates which, if provided with suitable feedback paths via delay elements, can be used as a programmable state machine.

This technique suffers from the array size being limited by the number of pins on the PLA device. However, about eighteen months after the work described in appendix II a manufacturer introduced a device with the feedback paths and latches incorporated with the PLA in a single device. This Field Programmable Logic Sequencer (FPLS, ref. 27) has sixteen inputs, eight outputs and six feedback lines, as shown in fig. 82.

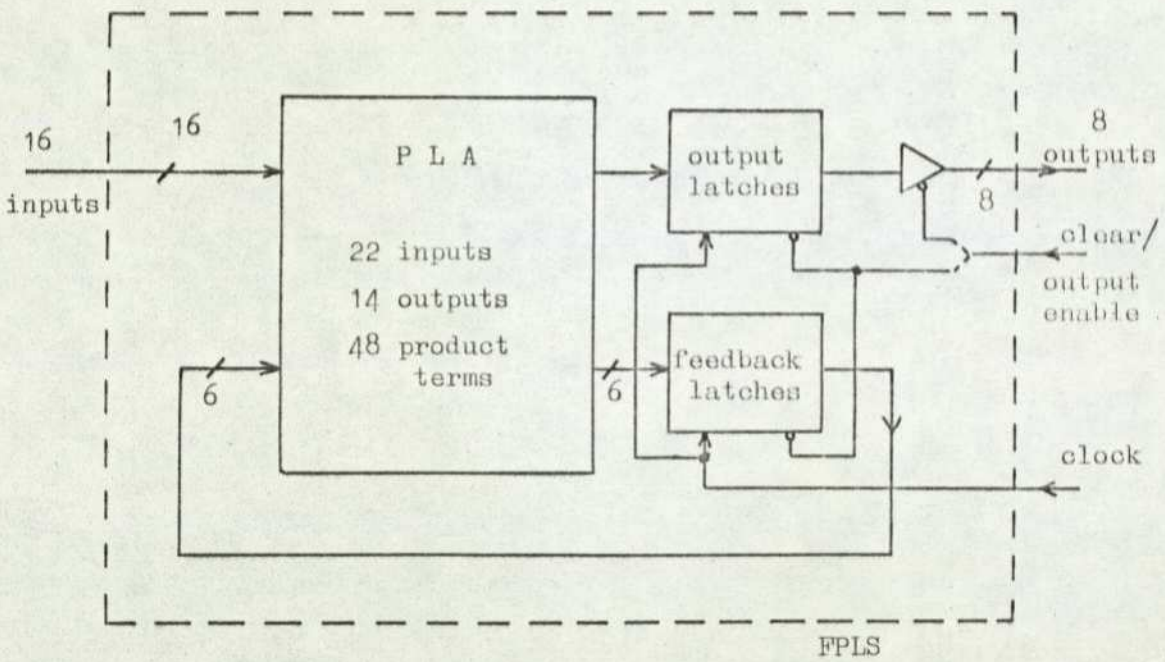


Fig. 82. The Field Programmable Logic Sequencer (FPLS).

Up to forty-eight input combinations can be decoded. The device can be operated at high speed as the cycle time is limited only by the sum of the delay through the AND/OR gates and the latch settling time.

This device effectively renders the converter described in appendix II , obsolete. However, the difficulties of programming the FPLS are considerably greater, and there is little room for expansion of the device if the 40-pin package is to be retained.

CHAPTER 9

THE RSP SOFTWARE SYSTEM

9.1 Introduction

It has already been stated that the RSP depends for its programming and testing phase on a host computer. This chapter describes the RSP support system installed on the computer which services the prototype RSP's.

9.2 The Range of Software Support Needed

In order to program and test an RSP, the following software items are needed:

1. An assembler to translate a low level assembly language into machine code.

2. An 'operating system' of some sort with at least the facility to load a program, load and monitor locations, and run and stop the processor. This may be by hardware, in the form of a front panel (7.3.3), or software in conjunction with a teletype.

The above is a basic list that would enable programs to be written, assembled, loaded, run and modified on a microprocessor. In addition to these, the following would be desirable if software systems of any size or complexity are going to be built:

3. A text editor, so that source listings can be stored and edited, preferably on a disc backing store.

4. A link editor, so that independently-compiled segments of program (e.g. subroutines) can be joined to form a complete program.

5. A library system which allows compiled segments to be stored in a convenient medium for retrieval by the link editor.

6. A de-assembler which will work in conjunction with the 'examine' facility of the operating system, and allows areas of compiled code to be displayed as source language.

7. A breakpoint facility which allows examination of locations during program execution.

8. A simulator which allows micro-processor programs to be run on the host machine for initial testing.

9. A compiler for a high level language, e.g. CORAL 66.

Items 1 to 9 form a comprehensive list of programming and de-bugging aids which would allow programs to be built up and tested on a micro-processor system.

9.3 Cross- or Resident-Software

Cross-software is that written to support one machine but which runs on another. In a micro-processor context, it usually means software which runs on a minicomputer or bureau mainframe computer and produces programs for a micro-processor. Resident-assemblers are assemblers which run on the actual micro-processor itself. In the past all software has been resident-software since it was not reasonable to expect a computer purchaser to have access to another machine just for software preparation. However, certain differences between main computers and minicomputers, and micro-processors make cross-software more attractive.

9.3.1 Differences between Minicomputers and Micro-Processor Systems

The development of micro-processor software aids has followed closely that of software for minicomputers. There are, however, inherent differences between these systems and their smaller counterparts which make this close analogy inappropriate. Some of the characteristics of the two types of systems are summarised below.

Minicomputer	Micro-Processor System
General purpose systems	Fixed purpose systems
Large store > 16K	Small store, often < 4K
Efficient backing store, e.g. hard disc	Floppy disc (relatively expensive and poor performance)
General peripherals, e.g. teletype, VDU, reader punch, etc.	Special peripherals, e.g. A-D and D-A converters

It can be seen that a micro-processor system is likely to have little store, possibly a floppy disc backing store, no reader or punch, and at best only a teletype for control. While it is possible to add peripherals or store to a micro-processor to make it loosely comparable (it can never be truly comparable - the speed differential between a floppy disc and a hard disc is too great), the resulting system

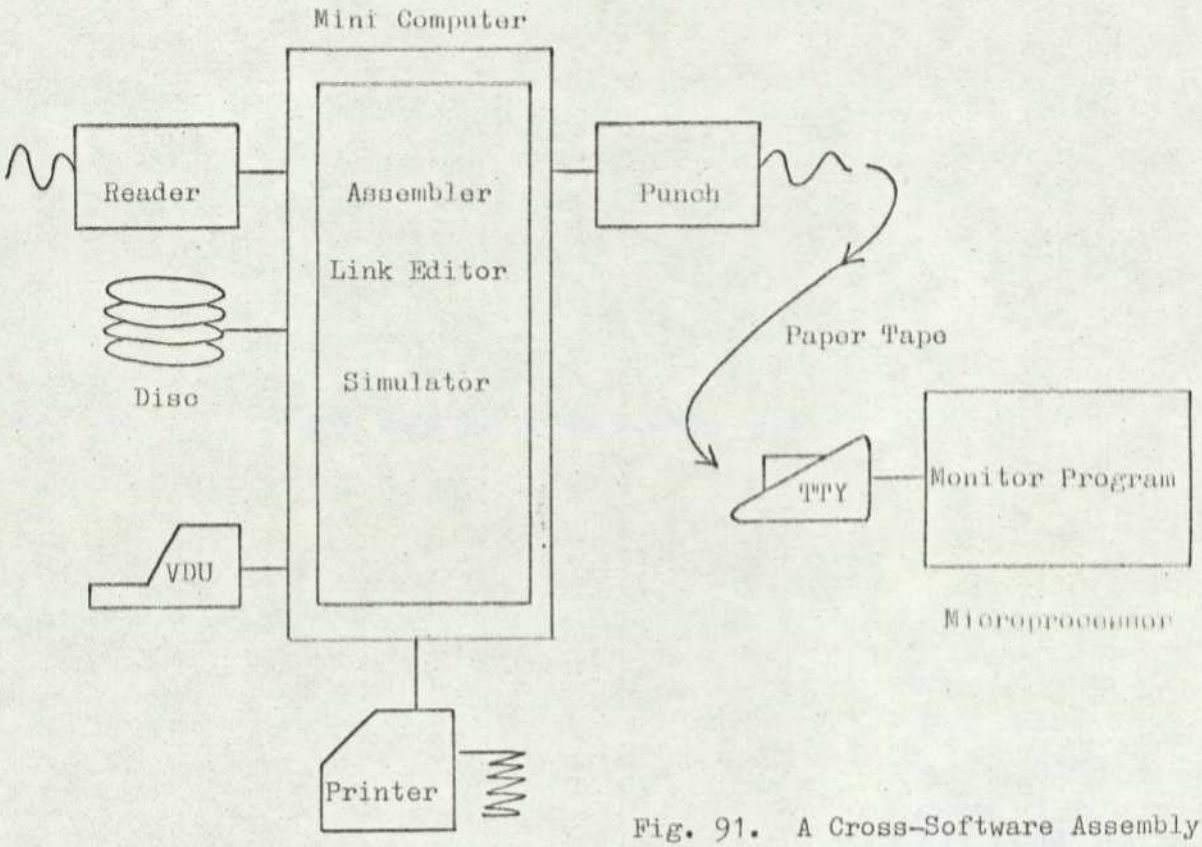


Fig. 91. A Cross-Software Assembly System.

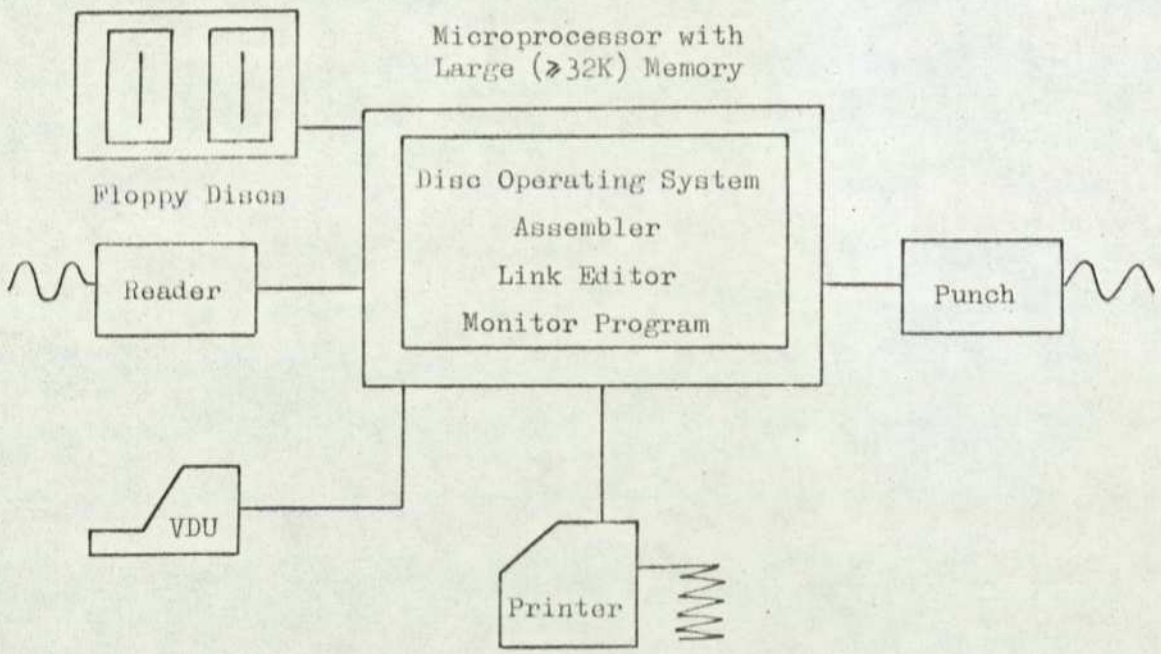


Fig. 92. A Resident Software Assembly System.

will only be suitable for software preparation and will be of such great cost that the original advantage of a micro-processor has been lost. For example, a micro-processor system for compiling a high level language consists of a central processor, 64K of store, dual floppy disc system, VDU for control and serial printer for program listings. The total cost is comparable to that of a minicomputer system.

9.3.2 Cross-Software Approach

Fig. 91 shows a cross-software system based on a minicomputer. It could equally well be based on a large mainframe computer owned by a bureau, in which case the output tape would be received at the terminal. In either system, this output tape is loaded into the teletype reader of the micro-processor system for loading.

The advantage of such a system is that the main computer can have a large store, and can therefore accommodate a powerful operating system which gives easy access to all system resources. Most importantly a large and fast backing store can hold libraries, compilers, etc. A text editor is likely to be installed on the system anyway.

9.3.3 Resident-Software Approach

Fig. 92 shows a micro-processor system capable of generating its own software. Many of the de-bugging operations and a loader routine are in ROM in the system. The large memory is capable of holding a complete assembler but high level compilers will be overlaid many times from floppy disc. This is a slow process (one master segment and two overlays take about half a minute to load). The system is cheaper than buying a minicomputer system, but its performance is much lower. All the software will tend to be less flexible and powerful, as program space has been saved at the expense of ease of use.

9.4 The Shortcomings of the Present Systems

Resident-assemblers are slow, and generally less powerful and flexible than cross-assemblers. They need considerable extra peripherals whose costs completely dwarf that of the original system. Cross-software provides a much better approach providing a suitable host machine exists or can be accessed. However, the cross-software system is of no help during the system de-bugging phase. This is when program errors which are only apparent when the program is installed on the operating hardware, show up. Even the cross-software system relies on an inbuilt de-bugging program and controlling device (e.g. teletype). Since a teletype costs in the region of £1,000 while workable micro-processor systems can be purchased for £200, the same imbalance of expenditure exists as with the resident-software.

9.4.1 The Difficulties of Simulating Real-Time Systems

The only way in which a conventional cross-software system helps the user to correct program execution faults is by providing a simulator. This is a program which runs on the host computer and accepts as input compiled machine code. It mimics the behaviour of the micro-processor, and allows registers to be examined and loaded and the program to be stopped and re-started. Better examples of simulators also return accurate timing information by evaluating the number of store and logic cycles in each instruction, and calculating program times accordingly. This is very useful in time-critical segments of code.

Such a simulator is invaluable for de-bugging programs with a lot of internal processing, such as arithmetic or sorting routines. However, since much real-time work is concerned with I-O operations, the simulator's usefulness is limited. It is possible to simulate input and output operations, but this is not useful unless the hardware to which the processor is connected can also be simulated. Carrying this process to its logical conclusion, one must simulate the whole of the environment in which the processor is going

to work. Even if this were possible in terms of required complexity of program, it is a dangerous practice. The 'real world' inevitably produces sequences of events timed in ways which were never anticipated when the real-time program was written. A programmer trying to simulate the 'real world' with a program would find it impossible not to build into such a simulation the same (possibly erroneous) assumptions that were the basis of the program to be tested.

9.4.2 In-Circuit Emulation

The manufacturers' answer to the problem of testing real-time systems with hardware and software interaction is the in-circuit emulator (ICE) (ref. 52). This is a device which contains a micro-processor of the type being tested, and has a 'flying lead' which plugs into the hardware system being tested, in place of the original micro-processor. The ICE has an inbuilt operating system which allows programs to be run, modified, etc., with a range of monitoring facilities, while appearing to the hardware as an ordinary micro-processor. Thus it is claimed that testing can be done in-circuit and at normal running speed. This last point is not entirely true since cabling delays and the need for drivers means that ICE's run slower than the processors they replace. At least one micro-processor manufacturer uses a dual processor ICE (ref. 67) with all the monitoring and operating programs run on one processor, while the program being tested runs on the other. In this way execution errors cannot cause the operating system to be disturbed. ICE's provide a good testing facility but need large system resources to back them up. Thus they are easily applicable to resident-software facilities where the necessary store and peripherals exist anyway, but are expensive if purchased as separate items as for a cross-software arrangement.

9.5 The RSP Programming and Testing Facility

For the reasons mentioned in 9.3.2, a cross-software approach was adopted for the programming of the RSP. This took two forms: a manufacturer-provided system consisting of assembler, link editor, library management system and simulator which runs on the University's multi-access system available in the laboratory through a terminal, and a purpose-built programming system installed in the on-line computer in the laboratory.

9.5.1 The Standard F100L Programming System

The programming system provided by the manufacturers for the F100 consists of a literal assembler, link editor and library system, and a simulator (ref 68).

The assembler is based on free-format literal language with named quantities being variable or preset and local or global. A macro facility (including nested macros) is provided. Store maps and summaries are output during assembly.

The link editor accepts 'intermediate code' output from the assembler and joins blocks to form a complete program. The user can make up a library of assembled segments which the link editor can access to build programs.

The simulator accepts the machine code output of the link editor and completely simulates the F100 processor. Facilities are provided for halting program execution when certain addresses or instructions are executed, or after a certain number of instructions have been executed. Monitoring and modification of code and data is provided. Comprehensive timing information is given by specifying the length of read, write, read-modify-write and logic cycles. The user is enabled to write additional simulation programs to simulate the behaviour of other hardware items in a system, and link these into the basic simulator. This is a difficult facility to use but is very powerful providing it is not abused (9.4.1).

The three programs are written in FORTRAN, and have been installed on both the multi-access system (ICL 1900 - MAXIMOP) and the FM1600B.

9.5.2 The RSP Host/Slave Programming System

Fig. 93 shows the RSP programming system based on the FM1600B computer. This will be seen to resemble closely the conventional cross-software system of fig. 91. The important difference is that, instead of producing a paper tape output which must be loaded into the RSP by a teletype, the completed program is transferred directly to the RSP store by DMA transfers, through a special interface into which the RSP plugs. This removes the need for any standard peripherals on the RSP but, more importantly, provides the possibility of fast, two-way, communication between the host and slave computers. This is the key to the operating system, since it allows the main processor to act as an 'intelligent front panel' for the RSP, monitoring and controlling it during execution.

The sequence of operations is as follows:

1. The program is written and input to the FM1600B on paper tape or through the VDU keyboard.

2. The assembler (fetched from disc) assembles the program in an 'image RAM' space in the main computer store. Errors in the source text may be corrected by a resident text editor.

3. Basic execution of the program can be checked by the simulator and timing information can be obtained.

At this point the program has been verified and modified as much as is possible without installing it in the RSP and allowing it to run.

4. The RSP operating system transfers the image RAM to the real RSP store through the special interface.

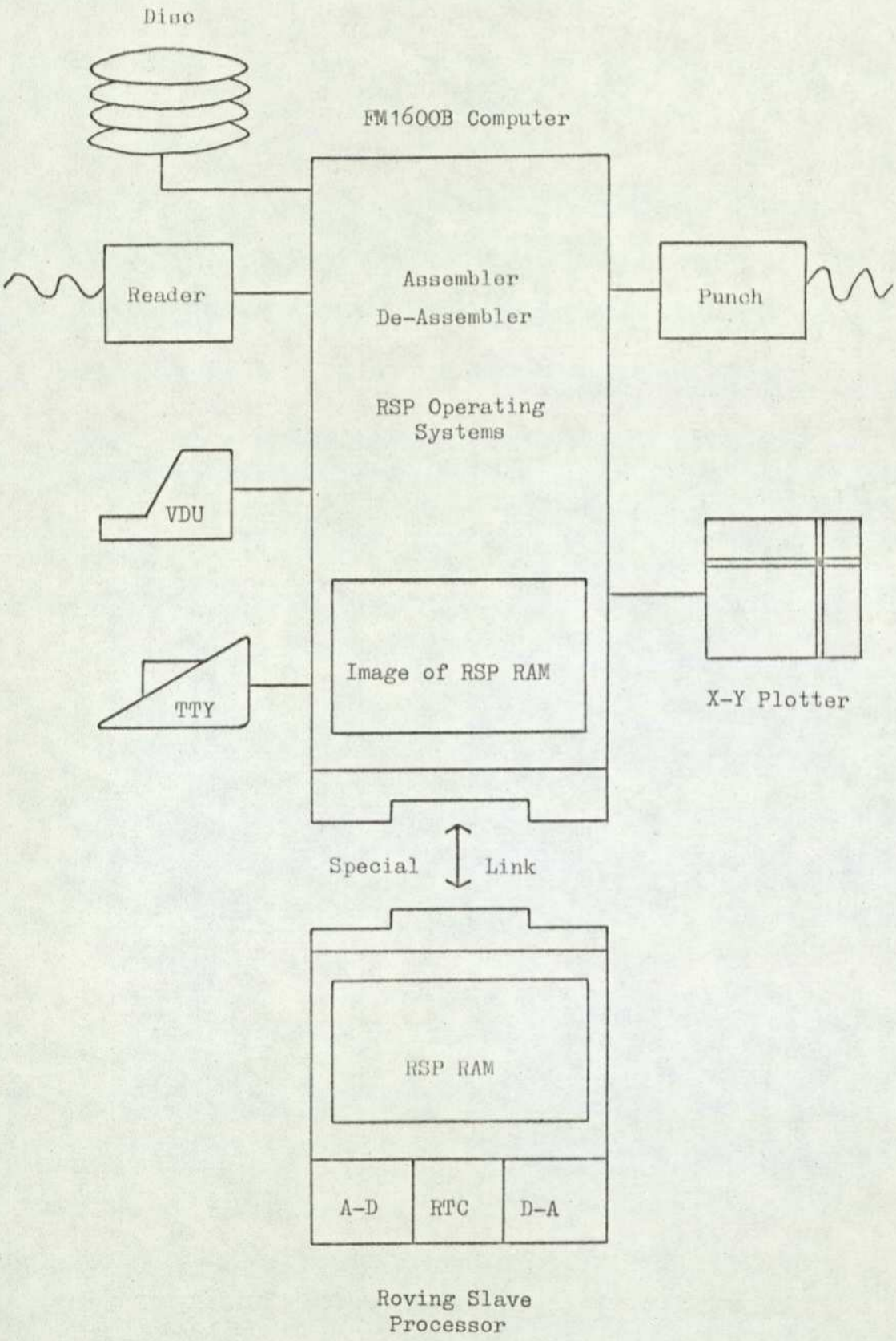


Fig. 93. The RSP Programming System.

5. With the RSP connected to its experimental test rig, the program can be run and halted under the control of the main program.

6. The entire RSP store contents can be copied back into the image RAM in the main computer.

7. The RSP operating system allows locations of the image RAM to be examined and modified. Results can be monitored, and the machine code language can be displayed as source language by use of the de-assembler incorporated in the operating system. Corrections can be made to the machine code.

8. 4, 5, 6 and 7 are repeated until the operator is convinced that the program is working correctly.

9. The RSP is detached from the data link and removed to the experimental site for the duration of the experiment.

10. When the experiment is complete, the RSP is re-connected to the main computer and the store contents transferred to the image RAM.

11. Results in the image RAM are displayed, stored or plotted by the main computer.

Thus the full range of peripherals and facilities of the main computer can be brought to bear on the problem of RSP programming and testing, without interfacing any devices directly to the RSP.

The assembler mentioned is a non-standard one written by the author specifically for the RSP project. It is based on an algebraic assembly language, which has been found to be easily assimilated by engineers. The assembler (appendix IV) was written in assembly language for the FM1600B computer.

A large part of any system development time is spent in identifying and removing errors in the program. In microprocessors this involves much manipulation of compiled machine code in the store or, in the case of the RSP system, in the image of the store in the main computer. The programmer is continually translating the binary patterns in store to the source instructions on his program listing in order to understand what his program is doing. That is to say, he is 'de-assembling' machine code back to source statements. The RSP editor includes a de-assembler, which allows blocks of stores to be monitored and displayed as source statements. Appendix V describes the de-assembler.

The RSP operating system (ref. 69) makes use of the fact that the host computer has a longer wordlength than the RSP (twenty-four compared with sixteen bits) to mark words which have been modified in the image store area. This means that, when the image RAM is loaded into the RSP, all 4K words are transferred initially. Subsequently, modification of a word in image RAM will cause a marker bit to be set. On the next transfer operation only words with their marker bits set will be transferred. This speeds up the process since the I-O channel used on the FM1600B during the development was a slow general purpose one. Eventually fast serial links will be used to connect RSP's to their hosts, which will allow them to be distributed about the laboratory unrestricted by bulky and expensive multi-core cables.

It became evident during the writing of the assembler that the level of diagnostic software on the FM1600B fell far short of that required for the development of a large system, such as the RSP assembler and operating system. Consequently the author first wrote a suite of routines, the breakpoint facility, to enable program execution errors to be traced and corrected. This facility (appendix III) allows markers to be inserted in compiled machine code so that when they are encountered, program execution is suspended while the user monitors and corrects the contents of the registers.

Because the algebraic language does not use named identifiers, it lends itself to de-assembly, that is, the conversion of machine code back into source statements.

The language is described in 'An Unorthodox Approach to Microprocessor Language' presented at IERE Conference 'Computer Systems and Technology', Brighton, 1977, and published in the proceedings.

'An Unorthodox Approach to Microprocessor Language' presented at IERE Conference 'Computer Systems and Technology', Brighton, 1977, and published in the proceedings

This paper (pp. 203-212) has been removed for copyright reasons

9.5.3 The CP1600 Self-Assembly System

The foregoing describes the true RSP system being developed with the F100-based prototype RSP. Another prototype being evaluated is based on the GI. CP1600 16-bit processor. This has been used for general assessments of processing performance and investigation of storage media, but is not a true RSP in that it does not depend on a host computer.

The CP1600 has a self-assembler supplied as a paper tape for loading into its 8K memory. A built-in (ROM stored) operating system drives the comprehensive front panel and implements a machine code editor.

The facilities are listed below:

Paper Tape Programs

Assembler (basic assembly language)
Super Assembler (semi high level language)
Text Editor
Test Routines
Mathematic Routines (fixed and floating point operations)

ROM Operating Systems

Memory Protection (upper and lower 'write protect' limits)
Load and Examine Any Locations
Run/Stop/Single Step Program
Insert Breakpoints
Simple Utility Routines (teletype driver, input-output routines)

The CP1600 has a high speed tape reader, digital cassette system and teletype interfaced to it, as well as A-D D-A converters and a real-time clock.

9.6 Applicability of High Level Languages

Most of the foregoing discussion has assumed that programming will be based on low level assembly languages. While it is true that high level languages are beginning to be used, it is the author's belief that their impact on real-time programming of micro-processors will remain small for a number of years.

A study of a number of large computer installations (ref. 70) produced figures comparing the efficiency of some high level languages with assembly language. In particular comparisons were made of PL 1 versus IBM assembler on an IBM 360 machine, CORAL 66 versus PLAN on an ICL 1900, and PL 360 versus ASSEMBLER on an IBM 360. PL 1 is a general purpose high level language, CORAL 66 is a real-time language which is available on both F100, FM1600B and the ICL 1905 at the University, and PL 360 is a high level assembler for the IBM 360 series of computers. The figures were:

Language	Production Ratio	MC Size Ratio	CPU Run-Time Ratio
PL 1 (Optimised)	2.0	4.0	2.6
CORAL 66	3.0	1.4	1.1 to 1.2
PL 360	1.3	1.0	1.0
Assembly Language	1.0	1.0	1.0

Production Ratio (x) =

$$\frac{\text{Development Time for Assembly Language Program}}{\text{Development Time for Language x}}$$

This shows that the general purpose high level language PL 1 offers a useful increase in programmer production but at a large cost in compiled code size and execution time. CORAL 66 provides a large decrease in program development time at a modest cost in program size and very little slowing of execution speed. However, the high level assembler, PL 360, gives a useful increase in production at no cost in program size or execution speed.

It thus seems that, in time-critical programs of the sort encountered in real-time instrumentation, the use of a high level language such as PL 1 (or its micro-processor derivative, PL/M) is quite unacceptable. CORAL provides the quickest way to a working program but uses 40% more store than is necessary. A high level assembler provides the best solution by preserving the speed and compactness of assembly language programs.

Thus if the algebraic language in use on the FM1600B/F100 system were to be enhanced to provide named operands, operand typing and IF-THEN-ELSE, DO-WHILE, etc. constructs, the best solution would be reached. Such a language would also avoid the compilation problems associated with a language such as CORAL which requires many passes, large intermediate files and a special processor for incorporating assembly language segments into the CORAL source. This facility would, of course, be unnecessary in a high level assembler.

The GI. CP 1600 'super assembler' is a step in this direction but it falls far short of what is required due to its incomplete implementation of block structuring, which means that blocks may not be nested within other blocks.

Other real-time languages (e.g. RTL-2 which appears to bear a more direct relationship to low-level operations) should be investigated but the lack of suitable compilers, and the difficulty of writing them oneself, prevents this.

A programming technique being developed within the Research Group (ref. 60) removes the need for general purpose high level languages. In this system each instrumentational process is represented by a block of software with all options available. The 'instrument' is designed by selecting appropriate blocks and linking them by means of the 'high level definer'. This control program selects options for all the blocks and ensures correct signal flow from block to block. These blocks can be written in a low level language and carefully optimised since the user has only to select them from the library.

9.7 Summary

Micro-processor software support, particularly for de-bugging operations, is crucially important.

Resident-assembly systems are compact and cheap but are generally slow and difficult to use. Cross-assemblers offer many of the advantages associated with a large range of peripherals and a good operating system, but are generally not able to help during the de-bugging phase except by simulation. This is shown to be of limited usefulness in real-time systems, and the practice of simulating the 'outside world' as well as the processor is dangerous.

The in-circuit emulator provides good run-time testing facilities but is more applicable to self-assembler systems. The RSP host/slave system uses the host computer, not only for programming, but for run-time monitoring as well. The fast bi-directional link enables intimate communication between host computer and micro-processor. This is not possible with the conventional paper tape system.

High level languages are generally inefficient, and the incorporation of high level constructs into assembly languages (particularly of the algebraic kind) is a better approach. Lack of suitable compilers means that some, potentially useful, languages cannot be investigated.

The 'high level definer' system enables previously-written segments to be linked to form a complete processing system. This provides a high level description of an instrumentational program without the use of general purpose high level languages.

CHAPTER 10

DISCUSSION & SUGGESTIONS FOR FURTHER WORK

10.5 Suggestions for Further Work

10.5.1 Processor Measurement

10.5.1.1 Processor and Instruction Set

10.5.1.2 RSP System

10.5.2 RSP System Development

10.5.2.1 Processor Development

10.5.2.2 Software Support

10.5.3 RSP Project Status

10.6 General Remarks

10.1 Introduction

This research project is concerned with developing the Roving Slave Processor from an idea to a practical reality. In so doing, the author has concentrated on the basic philosophy of the system in order to emphasize the fundamental problems and constraints that apply. The result of such an exercise takes two forms: the development of demonstrable equipment, and the formulation of design choices and the criteria on which these must be based. In this respect the author has attempted to produce a true 'engineering thesis' rather than adopting the quasi-scientific form in which results are necessarily numerical or graphical.

10.2 Special Factors influencing Design

While many of the factors upon which the RSP design depend are common to most branches of computer engineering, there are some which are of specific importance in Computer Aided Measurement.

10.2.1 Time

This is important in sampled data systems in two ways. In the first place, the minimisation of processing time is important because processing speed and the external characteristics of the process are closely related via constraints such as the sampling theorem. This means that all processing operations and hardware facilities must be viewed critically in the light of the time delays they introduce. Thus the current trend to categorise systems in very broad terms (e.g. assessing a computer system/operating system/language processor combination in terms of jobs processed per hour) is not useful, since the processing systems tend to be simpler; and, as a designer, one must have the ability to trace performance characteristics directly to design choices.

Neither is the commercial approach to performance, the faster the better, acceptable since in sampled data systems, one must complete the processing of one sample quickly enough to deal with the next, when it arrives. Thus the important factor is the ability to guarantee that samples can be dealt with at a certain rate. Because of this, techniques which offer statistical increases in performance (e.g. instruction look-ahead stacks, cache stores, etc.) are of dubious value. If the output is never to be corrupted (i.e. no samples are to be lost), then only worst case timings are of interest.

The second problem associated with timing is the need to maintain a representation of 'real-time' which is accessible to the program. To this end real-time clocks need to be used to control sampling operations. The danger of 'slippage' between time and the processor's representation of it resulting from software delays, has been emphasized (Chapter 6).

In order to avoid this problem, the author's design for the ADC/DAC incorporates a mechanism whereby a real-time clock (or external signal source) can control sampling directly while a flag bit, set on completion of the sample conversion, can be examined by the processor to confirm that the sample has been read. This method does not seem to be reported in the literature: most designers seem content to have the RTC cause interrupts and perform the sampling in the interrupt program. The author's approach also requires a flexible RTC design capable of providing the control pulses directly, over a wide range of sampling speeds and with reasonable precision. Again this does not accord with the literature which generally describes RTC's with very coarsely adjustable frequencies, with further dividing performed in software.

The prototype RSP with RTC and ADC/DAC demonstrated (Geneva Exhibition 1977) that it could perform a realistic CAM data acquisition task (one for which special equipment is used in the author's laboratory) at reasonable speed

(≈10 KHz sample rate) without measurable 'jitter' of the sampling frequency. Using the normal system of real-time clock interrupts, the jitter would have been approximately 10%.

It is relevant to consider this 'jitter' problem in the light of some of the newer processors which have built-in multiply/divide operations. These operations take considerably longer than the normal instruction time and are generally executed as a single, non-interruptable action. To take, as an example, a 16-bit processor with a multiply time of 40 μS would give 40% jitter at 10 KHz sampling rate. In this respect the extension of the F100L by the provision of a signed hardware multiplier/divider operation as a special processing unit (ref. 21), is to be welcomed as interrupt response times will not be affected.

10.2.2 Economic Factors

As engineering is an economic activity, factors such as cost of equipment and cost of manpower cannot properly be ignored. Indeed the raison d'etre of the RSP is an economic one based on the poor level of utilisation of specialised laboratory equipment. The author has, therefore, resisted the temptation to design non-standard processing system components which have no possibility of economic manufacture or reduction in size and power consumption through increased integration.

It was this principle that led the author to abandon the two-port store interface in favour of the split bus system based on standard interface sets. It also influenced the design of the interface converter (8.6) so that the device was designed to fit in a standard 40-pin package and could be fabricated using conventional ULA circuits.

It must be emphasized that the continuing development in hardware requires that decisions such as those above, be continually re-evaluated in the light of changing circumstances. For example, the appearance (8.7) of a device

similar to the programmable interface converter means that the two-port store interface may be replaced by a single chip programmable controller and two 8-bit latches. Thus it would no longer represent a significant increase in hardware over the conventional interface.

The author has attempted to clarify the nature and basis of the design choices that must be made to assist in the specification of future components. The comparison of existing devices has only been included in so far as it serves to underline fundamental advantages and disadvantages of different approaches.

10.3 Summary of Results

10.3.1 Introduction

The application of computers to difficult measurement problems can lead to a number of advantages, such as greater objectivity, repeatability over a large number of experimental runs, and the possibility of varying the stimuli according to the nature of the response. Also the mapping of the physical world's continuously variable quantities onto a finite number system allows the use of important mathematical techniques which have no analogue in the physical world (e.g. deconvolution, perfect filtering, etc.). However, much of the work in the field is ad hoc in nature, and a coherent philosophy is needed to establish CAM as a distinct branch of measurement science.

CAM can be extended by the use of one or more Roving Slave Processors served by a larger host computer. Such a system not only assures high utilisation by confining specialisation to software but also allows processing power to be brought to remote experimental sites.

The physical limitations within which such an RSP must be built can be established by consideration of the word "portable". As portable servicing equipment (e.g. oscilloscopes) are now used which are approximately 200 mm by 200 mm by 300 mm, with a weight of five to seven kilogrammes, this forms a basis for the first stage development of an RSP. Once size and weight have been fixed, the physical problems resolve themselves to those of reducing power consumption sufficiently to allow reasonable operating temperature within the enclosure and to allow a small and light enough power supply. These considerations lead to a power dissipation of about 35 to 40 watts within the enclosure, of which about 20% is lost due to the inefficiency of the power supply. Thus designs which are likely to call for a power dissipation of greater than 30 watts are not going to lend themselves to incorporation in a truly portable device.

The establishment of limits on processor power are more difficult. However high the power of a system, one can always postulate a problem which would require more. The method employed is to consider signal processing operations. This was chosen because there exists an external characteristic, the sampling speed, which can be used as a performance measure, and also because signal processing is one of the more demanding operations found in CAM.

10.3.2 Instruction Set and Architecture

Assessments of processing power are difficult to make in any general sense. A number of specific processors can best be compared by consideration of the storage space and execution time of a number of 'benchmark' programs. If a suite of such problems, each chosen to test specific processor facilities, is used (e.g. ref. 71), then useful results illuminating the strengths and weaknesses of the devices, can be obtained. This method, however, has some considerable difficulties concerned with the choice of benchmark. A recent authoritative survey of 16-bit micro-processors (ref. 72) used a prime number calculation algorithm as a test program, which hardly seems a typical application of micro-processors. In real-time systems, I-O is important, but it is difficult to assess due to the diversity of systems used, as well as the effect of external hardware on I-O efficiency. This highlights the most fundamental problem in processor performance assessment: how is it possible to formulate a problem without presupposing the form of the solution? The designer of a certain processor which is shown to be poor at a particular program might rightly point out that the range of facilities which the processor offers, would enable the problem to be tackled in a completely different way. Re-writing the benchmark would be unfair to other processors. This means that test problems of this sort must be defined in sufficiently general a way that individual processors can employ the method that suits them best. This implies a high level language definition

of the problem where the machine nuances can be coped with by the compiler. However, we will no longer measure processor performance, but instead the performance of the processor compiler combination.

The situation of larger machines (e.g. minicomputers) is not similar. In the first place, for a large amount of work high level languages will be used, so the machine/compiler performance is more relevant anyway. Secondly, the hardware facilities, I-O system, store cycle time, etc., are likely to be fixed, so meaningful benchmark times can be quoted. In microcomputer systems this is not so. Thus one must conclude that the diversity of external hardware possibilities for a given processor and the need to program in a low level language (where speed is critical) mean that benchmark programs serve as only crudest measures of processing power where micro-processors are concerned. The author has, therefore, attempted to abstract from processor attributes those points of direct relevance to the design of a real-time system such as the RSP. Although the important points appear as a list, it is important to realise the inter-dependence of many of them.

10.3.2.1 Number and Range of Addresses

The author's laboratory employs a three-address minicomputer (FM1600B) and a one-address microcomputer (F100L) as the master and slave processors in the RSP project. This allows contrasts of the number of addresses to be made. By analysing some assembly language programs, the author has demonstrated that the three-address facility of the FM1600B is almost never used except when one of the address fields is used for another purpose such as holding a jump displacement, a numerical constant, or a number of places to shift.

It is also possible to demonstrate (7.2.2.1) that the three-address structure is only usable for word lengths of about 24 bits or above. It is not workable in any reasonable sense for word lengths less than 20 bits. Thus one must

conclude, at a time when integrated circuit and packaging technology limit word length to 16 bits, a three-address machine is not practical for an integrated circuit realisation. This leads to the conclusion that a two-address structure with a bank of fast registers is the best design for a micro-processor if a multi-address structure is required.

A one-address machine performs all operations between an accumulator and a main store address. The number of main store locations that can be accessed is an important design factor. As a generalisation, this number is the sum of the local variables (i.e. those specific to a particular subroutine) and the global variables which are available to all programs. Thus efficient store addressing with a small operand field can be accomplished by making store accesses refer to a fixed store area (globals) or one defined by a pointer (locals). Each subroutine would set the pointer to its particular store area. One of the 16-bit processors not used in this work (TMS 9900, ref. 65) exploits a variation of this technique within a two-address structure. The F100L needs a large (11-bit) field to address all 2,048 locations which are used for local and global variables. This is wasteful of bits in the word.

The efficiency of accumulator processors depends on the amount of processing they do on a variable before returning to store. Thus they tend to be efficient for long chains of calculations but not for performing minor operations on a series of variables. Conversely, the effectiveness of the two-address structure is based on its ability to keep all those variables of immediate interest in its bank of fast registers. This implies two important points: firstly, that the bank is large enough to hold at least most of such variables, and, secondly, that the programmer is aware of when variables are going to be used or have ceased to be of interest.

Any attempt to establish the number of 'live' variables at any one time founders on the lack of a suitable definition of what is 'live' and what is not. Subjective results from the users of machines with eight internal registers (of which six are general purpose) suggest that this number is not sufficient. The FM1600B with 23 general purpose registers has more than enough; so it seems reasonable to suggest that 12 to 16 general purpose registers are sufficient. Whilst the programmer can, on an intuitive basis, allocate variables to fast registers, high level language compilers tend to use multiple registers in inflexible ways. Routines for fast register allocation (ref. 66) are generally inefficient for small routines. However, many compilers make no attempt to optimise fast register use but instead use each register for a fixed purpose, e.g. accumulation, loop counting, array subscript calculation, etc. (ref. 10).

Therefore, a one-address structure is more likely to be used effectively by a high level language compiler while a good assembly language programmer can achieve greater efficiency with a two-address system.

10.3.2.2 Operations

The importance of shift and bit operations has been underlined (7.2.1.1), as has the combined set and test operation for synchronisation of multiple processors. The basic range of logic and arithmetic functions (e.g. add, subtract, and, exclusive or, etc.) are shared by most processors, although the F100L implements subtract in a way quite inconsistent with the use of the accumulator as the main working register. This is a serious oversight.

The operations of multiply and divide have an important effect on processing performance, and present microprocessors offer an incomplete (e.g. unsigned) facility in this respect. Floating point operations are necessary in many signal processing applications but are, at present, beyond the capability of single chip micro-processors.

10.3.3 Hardware Capabilities

Of at least as much importance as the processing power of the device is the mechanism by which it communicates with the rest of the system and thence to the outside world. Since nearly all micro-processors use memory mapped I-O systems with peripheral devices occupying storage locations, the design of the main bus is crucial to the efficiency of I-O operations.

10.3.3.1 Bus Structure

The two basic choices in bus design are mixed or separate data and address buses, and synchronous or asynchronous transmission of data. Separating the data and address buses leads to a fast system with less hardware for the user (e.g. the address need not be latched), but can double the width of the bus. This is often unacceptable where the 'pin-out' limit on integrated circuits is concerned, and also for diffuse systems where bus length may be large, and the cost of driving and receiving the extra lines may not be warranted. The extra lines are also a potential source of unreliability. Mixed buses (data and address multiplexed onto a single bus) are slower and require removal and latching of the address by all peripherals. However, the reduction in bus width often outweighs this disadvantage. Synchronous transfer is simple and potentially fast, but cannot easily be made secure. Asynchronous systems operated in a handshake mode have inherent security in that receiving devices must acknowledge receipt of data, but the method imposes certain time overheads which may or may not be acceptable.

To take two extremes, a synchronous, separate bus system could be made extremely fast and might be suitable for communicating along the backplane of a single piece of equipment where the number of connections is not a severe constraint. Conversely an asynchronous combined system would be suitable for a physically larger system, especially where security is important.

Some of the points are illustrated by the two RSP prototypes: the F100 prototype used by the author and the CP1600 device used by a colleague. The F100 combined asynchronous bus requires 21 connections (16 data/address and five control) for all program controlled transfers and four more (DMA Request/Accept, Interrupt Request/Accept) to control peripheral initiated actions. The comparatively complex timing relationships dictate the use of interface sets for bus handling. Besides simplifying the interface, these provide good fault detection facilities for all types of transfer. The CP1600 chip has a 16-bit mixed synchronous system which is split into 16-bit address and 16-bit data by address latches on the processor board. A 3-bit output is decoded to give eight control signals which define the bus contents at any one time. Thus for basic control a bus of 40 connections is used. Prioritising of interrupt and DMA functions is not provided. Interfacing to the bus is simple with no address latching but fault handling must be provided by extra hardware.

The apparent inconvenience of the F100 bus is considerably mitigated by the existence of the interface sets and the overall system is compact and secure. The CP1600 bus lends itself to minimum hardware interfaces but this advantage is lost if proper safeguards are introduced.

10.3.3.2 Interrupt and DMA Facilities

Interrupts and DMA operations are crucially important to efficient high demand, input-output, systems. Both facilities need a priority system to arbitrate between simultaneous requests. The daisy chain system used by the author has the advantage of being suitable for a proper bus organised system which can easily be expanded. Centralised priority circuits which avoid the delays introduced by the daisy chain require 'star' connections to devices, thus violating the bus concept.

10.3.4 Dual Processor Operation

In order to have reliable dual processor operation, one must deal with communication and synchronisation in a rigorous way. This needs an indivisible test-and-set instruction unless special purpose hardware is to be built. Since efficient operation demands that the two processors be closely coupled, the overheads involved for the communication mechanisms must not be so great that the potential advantage of the dual processor is lost.

The dual processor configuration can be realised by splitting the bus or by the provision of a two-port store. The latter system offers some advantages, but both need to detect 'free' slots in instructions when the processor is not using the bus or memory block. The detection of such slots is more difficult in asynchronous bus systems, and a processor with a number of well-defined 'free' slots would have considerable advantages in a dual processor system. The two-port store system requires special purpose interfaces unlike the split bus which can be implemented by standard F100 interface sets.

The natural distinction in real-time signal processing between input-output operations and the manipulation of the signal samples leads to this division being adopted for the dual processor. It is still important, however, to maintain efficient I-O structures, and it has been demonstrated (ref. 76) that cyclic buffers in a common area of store can be manipulated entirely by DMA operations and program interrupts without other processor intervention. This underlines the importance of a good DMA facility.

10.3.5 Software

The design choices involved in the selection of a software system for the RSP fall into two categories: the choice of a language type and the selection of a range of testing aids.

10.3.5.1 Language Type

While much thought is given to the improvement of high level languages, assembly languages are not generally considered to be worth the expenditure of any effort. This has led to their dismissal by many people. In true real-time programs, where speed is critical, assembly language programming is still predominant. The author thus felt justified in expending some effort (and a considerable amount of development time) in the design and implementation of an algebraic assembly language for the F100L. The particular form of language was dictated by a requirement for similarity with the language already in use in the laboratory. The assembler produced by the author was extremely compact in contrast to commercial assemblers written in FORTRAN for portability.

The algebraic language lends itself to de-assembly, i.e. regeneration of source statements from compiled machine code. This operation is essential for fault diagnosis during program development, but seems to be largely ignored by manufacturers. The author wrote a de-assembly program for the F100 which is compact, and which was later incorporated into the RSP operating system, resident in the host computer.

The algebraic language and its de-assembler have aroused considerable interest outside the academic world.

Such a language could be extended to include IF-THEN-ELSE and other high level structures without loss of efficiency, or the obscuring of any machine facilities.

High level languages such as CORAL 66 offer portability* and considerable reductions in programming time. Self compiler systems require large stores and floppy disc systems, and even then many overlays are necessary for a typical compilation. Thus high level languages are better suited to cross-compilation systems. They do not offer the sort of efficiency of assembly language programming, and thus need to have the facility to include low level language inserts for time-critical parts of the program. This can complicate the compilation process.

10.3.5.2 Program Development and Testing Aids

Correcting and testing programs account for the largest part of program development time, so aids designed to shorten these phases are crucially important.

Simulators are useful in allowing basic program testing within a host machine, and are also able to time critical sections of program. Simulators are of limited use where I-O operations predominate, particularly in cases where hardware/software interaction (e.g. by the use of DMA and interrupt facilities) is intimate. The technique of extending simulators to incorporate elements of the 'outside world' as well as the computer system being tested, is dangerous particularly if such extensions are done by the programmer whose code is to be tested. The likelihood is that the same misconceptions will be built into both the program and the simulation, thus enabling a faulty program to appear sound.

* CORAL 66 has been implemented on two 16-bit micro-processors: the F100L and the TMS 9900. This has been done by the same software designers using a standard front end compiler and modifying the back end (code generation). This promises true compatability between 16-bit micro-processor CORAL 66 programs.

In-Circuit Emulation removes these difficulties by allowing programs to execute in the final system connected to its operational environment whilst maintaining a monitoring and controlling function for the purpose of dealing with errors. However, such systems are more appropriate to self-assembly installations.

The prototype RSP has been tested with its programming system consisting of the algebraic assembler/de-assembler and operating system in the master computer connected to the RSP via the link. Initially this link took the form of a slow parallel one working through a general purpose I-O channel in the main computer. Because of this the transfer rate was only a few hundred words/sec. but the general principle of maintaining an image of the RSP store within the master computer has been proved.

10.4 Technology Trends

The research project has so far lasted for four years (1974/78) during which time many developments have occurred which will affect the development of the RSP. It is the purpose of this section to discuss the more important of these changes.

10.4.1 High Level Languages

Throughout this thesis emphasis has been placed on assembly language programming in order to maintain efficiency and high speed operation. Recent developments, both hardware and software, have made the use of high level languages more appropriate.

10.4.1.1 Hardware Developments

As has already been outlined (7.4.2) little can be done to optimise instruction types for high level languages, as their compilers tend to use processors in very simple-minded ways. However, operand types and addressing structures can be optimised for high level language use. A recent processor, Intel 8086 (ref. 73), combines 16-bit operations (including signed divide and multiply) with addressing structures suitable for high level language operation. In particular it has a simple virtual memory facility so that addresses can be relocated by adding to one of four relocation registers, which divide the physical address space into four areas: code, data, the stack and one spare partition. The processor also has instructions to implement the 'lock' function for synchronisation in multi-processor systems The 8086 is thus extremely well suited to an RSP type application.

There are two points of dubious merit with this processor. Firstly, it has a six-deep instruction look ahead stack which means that execution times are reduced on average without affecting worst case times. This has dangerous possibilities

in real-time sampled data systems (10.2.1). Secondly, because of the powerful instruction code, there are a large number of functions and subsidiary operations in the machine code. The retention by the manufacturer of a literal representation for this has produced an inscrutable assembly language. This underlines the advantages of an algebraic assembly language (9.5.2) particularly for machines with rich instruction sets. (The author suspects the manufacturers have no interest in a good assembly language as they wish to encourage high level languages on this processor.)

10.4.1.2 Software Developments

While the problems associated with speed and efficiency may be mitigated by the developments described above, some problems remain especially those concerned with intimate hardware/software interaction. An important such area is the synchronisation of dual or multi-processor configurations. However, recently the high level language PASCAL has been extended to Concurrent PASCAL, a language designed specifically for multi-processors. This language incorporates within each program a segment which provides rigorous communication and synchronisation facilities between processors. PASCAL is also attractive because it is available with a compiler system which affords easy implementation. It has already been stated (9.6) that the difficulty of writing specific compilers has hindered much high level language work on micro-processors.

The PASCAL compiler consists of a standard 'front end' which performs syntax checking on the source statements and produces as an output a 'P' language which must be interpreted by the micro-processor. Thus implementation on a specific processor involves only the writing of an interpreter for the 'P' language. This is considerably easier than writing a conventional code generator. The advent (ref. 74) of processors which actually execute a 'P' language further increases the attractiveness of this approach.

Thus PASCAL, particularly Concurrent PASCAL, seems to offer an appropriate range of facilities combined with ease of implementation.

10.4.2 Semi-Conductor Technology

The rapid increase in component packing density and the consequent decrease in power consumption are clearly important with regard to a truly portable RSP. The tendency to reduce power supply requirements not only in terms of current drain but also to a single 5 volt supply, makes the provision of small high efficiency power supplies easier.

The increasing power of single chip micro-processors means that many desirable features, including eventually floating point operations, can be integrated with a central processor in a single chip.

The importance of input-output operations has already been stated (Chapter 5). The impact of new technology on this seems to be minimal.

10.4.2.1 The Input-Output Configurations

In a true real-time system the processor's role is to be responsive to the needs of the outside world. This means that actions arise, in the first place, as requests for processor activity signalled by a peripheral. Thus the responsiveness and performance of real-time systems depend on the degree of peripheral autonomy that is allowed by DMA and program interrupt structures. Newer processors have improved performance in these respects (e.g. the 8086 has 'transparent' DMA and extensive interrupt capabilities) but few peripheral controllers have been produced to exploit these facilities. Most micro-processor manufacturers have concentrated on simple program-driven controllers for minimum hardware configurations. There is a need for a multi-channel

peripheral controller which would allow each channel to have vectored and prioritised interrupts with selective lock outs, DMA block transfers with automatic address provision and transfer counting, and allow automatic re-setting of address pointers on completion of a block transfer. In this respect the F100L interface sets come nearer to the ideal than any later devices.

The disadvantage of such a circuit would be the number of pins required, but with the advent of technologies such as I^2L, gate speeds can be adjusted over a wide range on the same chip. This would allow small fast serial input-output sub-systems to be incorporated within the larger devices.

Industry's slowness to appreciate the power of such systems (with the exception of the military area), is unfortunate.

At the beginning of this section it was stated that little could be done to optimise processors for high level language use. This is true where the translation to machine code is done by a conventional compiler. A recent trend in compiler construction (ref. 48) means that the processor is left with an intermediate language to interpret (10.4.1.2).

10.5 Suggestions for Further Work

Because this thesis represents the first work in the field of processor development for measurement systems, within the Department, the section dealing with future work is necessarily large and detailed. The tasks which remain to be done before the RSP concept is brought to a useful reality, fall into two categories: those which are concerned with performance measurement and the assessment of the processing system, and those which involve the development and testing of the software and hardware aspects of the system. It must be emphasized that such a project as this is a team effort, and many of the points to be considered are already being dealt with.

10.5.1 Performance Measurement

If one is to have more than a subjective impression of how effective a given structure or technique is, then one needs detailed performance figures. This applies to individual processors as well as the overall RSP system.

10.5.1.1 Processor and Instruction Set

As semi-conductor technology advances, so the variety of devices that may be used as the basis for an RSP, grows larger. An objective selection of a particular device requires knowledge of the sort of function that the device will perform when used in the RSP. Therefore it is relevant to carry out an analysis of instruction types used in various programs which must be representative of the sort of task for which the RSP will be used. This is difficult because such an analysis on any given machine will not measure the occurrence of the sort of functions the programmer wants, but rather a combination of their requirements and the particular facilities provided by the processor. This problem can be approached from two different (indeed opposite) directions; each solution involving the simulation of an imaginary processor with an idealised instruction set.

This set can be either extremely simple with each instruction doing one elementary operation, or it can be very complex so that almost any conceivable operation (including subsidiary functions) can be performed in a single instruction. If synthetic programs (representative of typical CAM applications) are compiled into the imaginary machine code, then the occurrence of instruction and operand types will reflect the demands of the program as there are no machine limitations to distort them. Analysis of the code would reveal such factors as: the relative advantages of a three-address machine as opposed to a two- or one-address processor, the distribution of sizes of literal fields and jump displacements, and the usefulness of single bit operations. Careful consideration of these results will lead to a greater degree of objectivity in device selection.

10.5.1.2 The RSP System

The dual processor configuration with cyclic buffering and a signal processing program, poses a number of questions to which the system designer needs quantitative answers. In particular the proportion of time spent dealing with cyclic buffer pointers and that involved in the synchronisation procedures (both of which may be regarded as 'overheads'), should be measured. An alternative approach to this would be to measure the maximum sampling frequencies for a given filter as a function of a changing DMA load. This last measurement could also be used to compare various (e.g. split bus, split store block, dual-port store) dual processor arrangements.

The obtaining of such figures is difficult and cannot be done by simulation unless the process involves simulation of individual control lines and bus contents at a very detailed level. Work is already in hand within the Research Group to collect figures concerning the synchronisation overhead of the dual processor. This is done by a logic analyser connected to the processor bus.

Once the figures have been obtained, it will be possible to estimate the efficacy of various dual processor configurations, something which other workers in the field seem reluctant to do.

10.5.2 RSP System Development

Much development work has been proceeding in parallel to that described in this thesis, both on the processor itself and on its support system.

10.5.2.1 Processor Development

While evaluation of the prototype dual processor RSP is continuing, work is underway to build a truly portable RSP dual processor, based on the F100L. The size of this is 200 mm x 200 mm x 300 mm, and it is powered by a light switched-mode power supply. Some of the interface set functions have been performed by discrete logic, since this saves board space and power where only a few of the interface sets capabilities are being exploited. It is intended that both these RSP's will have the Ferranti multiply/divide chip added shortly. This will provide signed operations with execution times of less than 10 μS when used as a special processor unit with the F100L.

Much developmental effort has been expended on the design of the host/slave link. The original used a general purpose I-O channel and was slow. The replacement will probably be a fast serial link, possibly based on a Ferranti design (ref. 75) which allows data transfers, controlled from either end, at rates of 5-6M bits/sec.. The disadvantage of this would be its lack of standardisation so alternatives, possibly including IEEE-448, are being considered. It must be stated that eventually the RSP concept will be implemented as a portable software package communicating through a standard hardware interface.

10.5.2.2 Software Support

Programming languages have been investigated by the Group, including CORAL 66 (available on the University's multi-access system as well as FM1600B and F100L) and, more recently, PASCAL. Work is now beginning to implement PASCAL by writing an interpreter for its intermediate languages. This will provide firm comparisons of size and speed of code. Structured algebraic languages should also be investigated although the effort involved is considerable to a team with a predominantly 'hardware' background.

Suites of filter routines have been produced, and some attempts have been made automatically to generate program modules by specifying process parameters. Much work is needed here: the ultimate aim is to be able to specify filter, signal averaging, FFT, etc. modules and for these to be automatically linked into a measurement system.

10.5.3 RSP Project Status

Fig. 101 was drawn at the RSP project outset in 1974 to indicate the expected paths the research and development activities would take. A similar diagram drawn at the end of 1978 (Fig. 102) shows what actually happened. While the form of the two diagrams is different, it will be seen that most of the items of Fig. 101 have given rise to work reported in Fig. 102. It can also be seen that many of the suggestions for further work are already in hand within the Research Group.

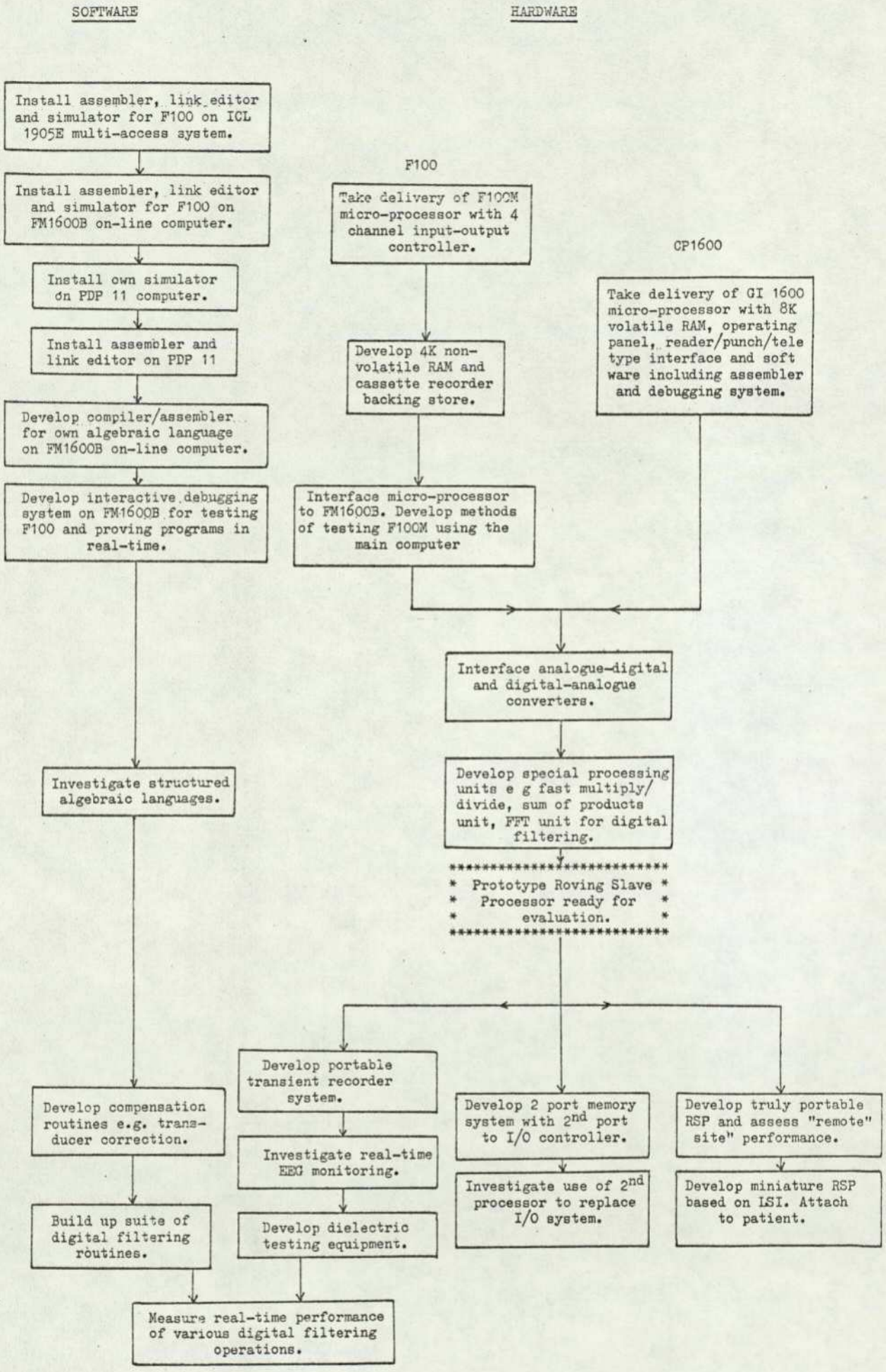


Fig. 101. Expected RSP development, 1974.

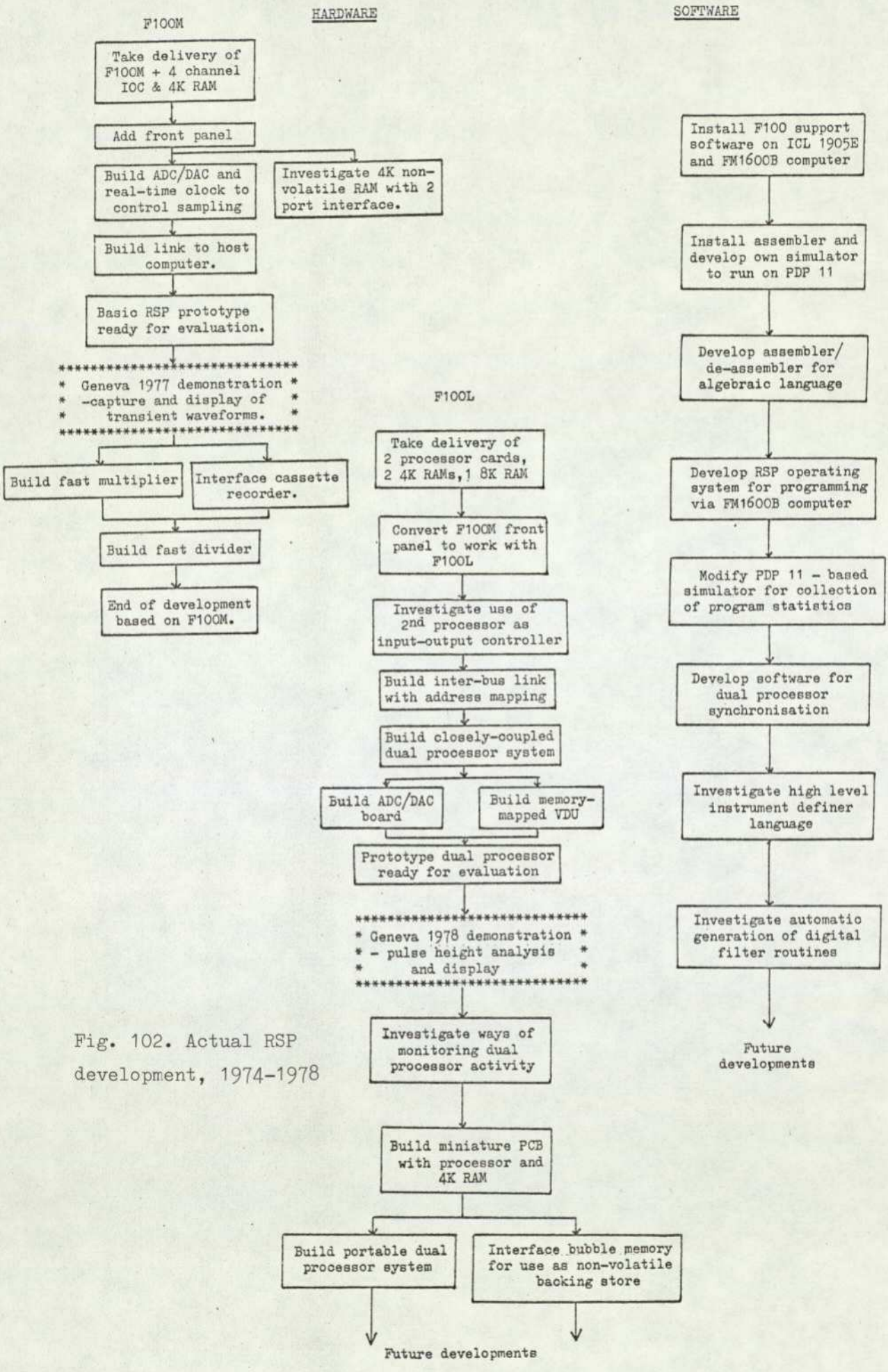


Fig. 102. Actual RSP development, 1974-1978

10.6 General Remarks

The equipment and software produced by the author form part of a team project in which other work has been carried out, both in parallel and subsequently, and some of the principles laid down have already been put into practice.

Much of the work reported in the field of hardware and software system design concentrates on modularity, consistency, simplicity and elegance of idea. These goals are often achieved by increasing levels of abstraction so that the hardware is continually being isolated from the user. Very little mention is made of the price paid for such abstractions in terms of program size or execution speed, indeed it is often implied that such factors ought not properly to be considered. In true real-time systems these penalties can be severe and the system designer cannot ignore them. It is thus imperative that where speed is sacrificed, the designer is aware of how much processing power is being lost, and that such loss is minimised. The author thus makes no apology for his consideration of these factors, wherever design choices arise.

The intention of this thesis has been not only to describe hardware and software contributions to the realisation of the RSP principle, but also to lay down a philosophy which will enable subsequent work to be carried out in a systematic and rigorous manner. This is by far the most important aspect of the work, and if the author has succeeded, it will not, like the more practical contributions, be rapidly overtaken by technological advance. Nevertheless, it can be recorded as an achievement that an idea, which could be important in the field of CAM, has been brought to fruition during the period of the project.

CHAPTER 11

CONCLUSIONS

The formalisation of CAM, and the application to it of the RSP, have been shown to give considerable advantages to the designer of a measurement system, both from the economic and the technological viewpoints.

The design of an RSP has been shown to be conditioned by two major constraints: the need to maintain portability and the need to avoid restriction of operating techniques due to the use of special purpose hardware. Dual processor systems, while posing problems of efficient synchronisation and communication, contribute to these goals by exploiting the natural division in real-time computing between input-output operations and arithmetic functions.

The support system described uses novel programming and testing methods to ensure that the economic advantages gained by the use of inexpensive micro-processor systems for the RSP are not lost.

Detailed consideration of aspects of processor and system performance have allowed the formulation of general design guidelines which will not be invalidated by future device developments. Such an exercise requires detailed knowledge of processor operation and this is not always available.

Before the work described here, CAM was a diffuse, barely identifiable subject physically confined to the vicinity of a computer installation. Yet the nature of the problems brought to the author and his colleagues underlined that it was incapable of coping with the most relevant and urgent measurement tasks posed by modern science and technology. The development of an RSP, from an idea to a practical reality as described in this thesis, has made a marked contribution to the mitigation of this inadequacy.

REFERENCES

References

Note: References for the review paper on Computer Aided Measurement (Chapter 2) are not included here but are to be found on pages 42-45

1. Brignell, J.E. and Wright, R.E. 'The Economics of the Microprocessor as a Component in Industrial and Laboratory Systems' Conf. 'Application of Microprocessors in Instrumentation and Control Systems' Organised by SIRA and Warren Spring, The City University, September 1976.

2. Comley, R.A. PhD Thesis, The City University, 1978.

3. Ferranti Ltd. FM1600D/FM1600E Computer System Principles, p. 5. Ferranti DSD, Bracknell.

4. Chapin, G.C. 'What is Different about Tactical Military Operational Programs' AFIPS Conf. Pub. 42, NCC 1974, p. 787.

5. Knuth, D.E. 'An Empirical Study of FORTRAN Programs' Software - Practice & Experience, Vol. 1, No. 2, 1971, p. 105.

6. Barak, A.B. and Aharani, M. 'A Study of Machine-Level Software Profile' Software - Practice & Experience, Vol. 8, 1978, p. 131.

7. Wirth, N. 'The Design of a PASCAL Compiler' Software - Practice & Experience, Vol. 1, No. 3, 1971, p. 309.

8. Sumner, F.H. 'Measurement Techniques in Computer Hardware Design' Infotech State of Art Report, 'Computer System Measurement', p. 367.

9. Tanenbaum, A.S. 'Implications of Structured Programming for Machine Architecture' Comm. ACM, Vol. 21, No. 3, 1978, p. 237.

10. Wichmann, B.A. 'Algol 60 Compilation and Assessment' Academic Press, 1973, London and New York.

11. Fuller, S.H. and Barr, W.E. 'Measurement & Evaluation of Alternative Computer Architectures' Computer, October 1977, p. 24.

12. Barbacci, M.R. and Siewiorek, D.P. 'Evaluation of CFA Test Programs via Formal Computer Descriptions' Computer, October 1977, p. 36.

13. Bell, C.G. and Newell, A. 'The PMS and ISP Descriptive Systems for Computer Structures' AFIPS, SJCC, 1970, p. 351.

14. Bell, C.G. and Newell, A. 'Computer Structures: Readings and Examples' McGraw Hill, New York, 1971.

15. Teorey, T.J. 'General Equations for Idealised CPU-I/O Overlap Configurations' Comm. ACM, June 1978, Vol. 21, No. 6, p. 500.

16. Lorin, H. 'Parallelism in Hardware and Software: Real and Apparent Concurrency' Prentice Hall, New Jersey, 1972.

17. Thurber, K.J., Jensen, E.D., Jack, L.A., Kenney, L.J., Patton, P.C., Anderson, L.C. 'A Systematic Approach to the Design of Digital Bussing Structures' AFIPS FJCC, 1972. Conf. Pub. 41, Part II, p. 719.

18. Ferranti Ltd. FM1600E 'Input-Output Controller Design Spec.' Doc. DS1112, Issue 4/S15, Ferranti DSD, Bracknell.

19. Washburn, J. 'Making Minicomputer I/O Upwards Compatible' Electronics, March 17 1977, p. 100.

20. Beaston, J. 'Second-generation Microcontrollers take on Dedicated-function Tasks' Electronics, November 23 1978, p. 127.

21. Ferranti Ltd. 'HSM-150 F100L Hardware and Systems Manual' Ferranti DSD, Bracknell.

22. Rush, P. 'Peripheral Microcomputer and Advanced General Purpose Interfaces for Microcomputer Control Systems' Proc. of IMMM '77, p. 291.

23. Fletcher, W.I. and Despain, A.M. 'Simplify Sequential Circuit Design' Electronic Design, 1971, No. 14, July, p. 70.

24. Howard, B.V. 'Partition Methods for Read Only Memory Sequential Machines' Electronic Letters, June 1972, Vol. 8, No. 13, p. 334.

25. Heath, C. 'Designing Programmable Sequential Logic Circuits' Electronic Engineering, February 1977, p. 45.

26. Sweet, A.W. 'A New Approach to Programmable Logic using a 1-bit Processor' IEE Computer Systems and Technology Conf. Proc. 1974, p. 29.

27. Danbury, A.J. 'Filling the Complexity Gap' Electron, 12th December, 1977, p. 51.

28. Ricci, D.W. and Nelson, G.E. 'Standard Instrument Interface Simplifies System Design' Electronics, November 14 1974, p. 95.

29. Clout, P.N. 'Interfacing' in 'On-line Computing in the Laboratory' Editors: Rosner, R.A., Penney, B.K. and Clout, P.N. Advance Publications, London, 1975.

30. Kirsten, F.A. 'CAMAC Specification' IEEE Trans. Nuc. Sci. NS20, 1973, p. 562.

31. Grimsdale, R.L. 'The Architecture of a Reconfigurable Multi-Micro-Computer System - Polyproc' IERE Conf., Computer Systems and Technology, Conf. Pub. No. 36, p. 113, 1977.

32. Searle, B.C. and Freberg, D.E. 'Micro-processor Applications in Multiple Processor Systems' Computer, October 1975, p. 22.

33. Ferranti Ltd. 'Ferranti Mk II Store Interface Specification' C27 S9, Issue 4/S15, Ferranti DSD, Bracknell.

34. Kinnie, C. and Maerz, M. 'Dual Port RAM hikes Throughput in Input-Output Controller Board' Electronics, August 17 1978, p. 107.

35. Intel '8080 Assembly Language Programming Manual' Intel Corp., Santa Clara, California, 1973.

36. Nicoud, J.D. 'Standard Mnemonics and Software Support for Microprocessors' Proc. IEEE Conf. ISCAS, 1975.

37. Brown, N. 'Modular Programming in FL/M' Computer, March 1978, p. 40.

38. Sandness, R.G. 'An 8080 Resident Fortran Compiler' IMMM '77 Conf., p. 214.

39. Depledge, N. 'CORAL 66 - A Practical High-Level Language for Minicomputer and Application Program Development' Infotech State of the Art Report, 'Real Time Software', p. 673.

40. Moore, C.H. and Rather, E.D. 'The Use of FORTH in Process Control' IMMM '77 Conf., p. 284.

41. Barnes, J.G.P. 'RTL/2 Design and Philosophy' Heyden, London, 1976.

42. Wirth, N. 'PL 360 - A Programming Language for the 360 Computer' Journal ACM, Vol. 15, No. 1, January 1968, p. 37.

43. Bell, D.A. and Wichmann, B.A. 'An ALGOL-like Assembler for a Small Machine' Software - Practice & Experience, Vol. 1, 1971, p. 61.

44. GEC Computer Ltd. 'GEC 4080 Computer Technical Description' Ch. 5, 'Babbage', p. 37, GEC Computers Ltd.

45. Moon, J.B. 'BSAL-80 A Block Structured Language for Microprocessors' IMMM '77 Proc., p. 217.

46. Wright, D.M. 'A Medium Level Programming Language' Proc. Intenepcon, Brighton, 1978, p. 124.

47. Hansen, P.B. 'The Architecture of Concurrent Programs' Prentice Hall, New York, 1977.

48. Berry, E.R. 'Experience with the PASCAL P-Compiler' Software - Practice & Experience, Vol. 8, 1978, p. 617.

49. DEC 'ODT - On-line Debugger' PDP 11 Software Handbook, p. 2-29, Digital Equipment Corporation, 1975.

50. Fryer, R.E. 'The Memory Bus Monitor - A New Device for Developing Real-time Systems' AFIPS, NCC 1973, p. 75.

51. Smith, J.H. 'A Logic State Analyzer for Microprocessor Systems' Hewlett-Packard Journal, January 1977, p. 2.

52. Yen, M.Y. 'Fast Emulator debugs 8085-based Micro-computers in Real-time' Electronics, July 1977, p. 108.

53. Knowles, J.B. and Edwards, R. 'Effect of a Finite Wordlength Computer in a Sampled Data Feedback System' Proc. IEE, Vol. 112, p. 6, June 1965.

54. Texas Instruments 'SN74172 Product Specification - 16-bit Multiple Port Register' Texas Instruments Inc., Texas, 1976.

55. Mick, J.R., Springer, J. and Ghest, C. 'A High-speed Parallel Multiplier - The Am 25LS14' Application Notes, Advanced Micro Devices, California, 1977.

56. Mick, J.R. 'Understanding Booth's Algorithm in 2's Complement Digital Multiplication' Application Notes, Advanced Micro Devices, California, 1977.

57. A.M.D. 'Am 2901, Am 2909 Technical Details' Advanced Micro Devices, California.

58. Tracey, R.N. 'A User Guide to the F100 Real-time Clock' Electronic Engineering Department Report, The City University, 1976.

59. Buffam, C.J. and Brignell, J.E. 'On-line Capture and Analysis of Random Phenomena' 'On-line Computing' Editors: Rosner, Penney and Clout, Advance Publications, London, 1975.

60. Comley, R.A. and Hewish, T.R. 'The Software Realisation of Instruments' International Microcomputers, Minicomputers and Microprocessors, Geneva, 1977, p. 41.

61. Rhodes, G.M. Ph.D. Thesis, The City University, 1973.

62. Parsons, B.J. and Dobson, D.P. 'Reliability Considerations and Design Aspects of the HSD Space Computer' IERE Conf. Pub. 36, Supplement, Computer Systems and Technology, 1977.

63. Bridgstock, E.C. 'Extensions to the Ferranti F100 Simulator' Computer Science Department Project Report, The City University, 1977.

64. Swann, D.W. 'Adapting a Computer Operating System for use in a Laboratory Environment' Electronic Engineering Project Report, The City University, 1974, p. 22.

65. Texas Instruments 'TMS 9900 Microprocessor Data Manual' Texas Instruments, 1975.

66. Horwitz, L.P., Karp. R.M., Miller, R.E. and Winograd, S. 'Index Register Allocation' Journal ACM, Vol. 13, No. 1, p. 43, January 1966.

67. Signetics 'TWIN Microprocessor Prototyping and Developing System' News Release, Geneva, May 1977.

68. Ferranti 'F100 Software Manual - PM150' Ferranti Ltd., DSD, Bracknell, 1976.

69. Brignell, J.E. and Buffam, C.J. 'The Totally Programmable Instrument' IERE Conf. 'Programmable Instruments', NPL 1977, Conf. Pub. No. 38, p. 1.

70. CCA 'Evaluation of Programming and Systems Techniques - Choosing the Programming Language' CCA Guide No. 7, HMSO, 1975.

71. Penney, B.K. 'The Implications of Microprocessor Architecture in Speed, Programming and Memory Size' IERE Conf. Pub. 36, p. 63, Computer Systems and Technology, Brighton, 1977.

72. Barron, I.M. (Editor) 'Microcomputer Analysis' Vol. 1, Nos. 6 and 7, Mackintosh Publications Ltd., 1977.

73. Katz, B.J., Morse, S.P., Pohlman, W.B. and Revenel, B.W. '8086 Microcomputer Bridges Gap between 8 and 16-bit Designs' Electronics, February 1978, p. 99.

74. Posa, J.G. 'Microcomputer made for PASCAL' Electronics, October 12 1978, p. 155.

75. Ferranti 'Ferranti 'B' Serial Interface Specification and Operational Description' Report 962, Issue 4/515, Ferranti DSD, Bracknell.

76. Mainwaring-Samwell, P. and Brignell, J.E. 'Dual Processor Software for Signal Processing' IMMM '78 Proc., p. 63.

ACKNOWLEDGEMENTS

I would like to express my appreciation for the constant advice and encouragement of my supervisor, John Brignell. My colleague and co-researcher in this field, Dick Comley, must be thanked for his advice and many practical contributions to the realisation of the RSP. All my colleagues in the Research Group are to be thanked for their many helpful comments.

Thanks are due to the Electrical and Electronic Engineering Department of The City University for the provision of research facilities, and to the U.K. Science Research Council for the provision of financial maintenance.

Messrs. Ferranti Ltd. are to be thanked for their co-operation during the research work. In particular, the contributions of [redacted] are much appreciated.

Finally, [redacted] must be thanked for her typing, and for her remarkable patience and support, especially during preparation of this thesis.

APPENDICES

There follow six appendices which describe in detail various practical aspects of the work outlined in the body of the thesis.

Appendix I	The ADC/DAC Board for the F100M
Appendix II	The Programmable Interface Converter (PIC)
Appendix III	The FM1600B Breakpoint Facility
Appendix IV	The DIXPAC Assembler
Appendix V	The DIXPAC De-Assembler
Appendix VI	The Store Interface Unit

Some appendices describe equipment or software packages (I, III, IV, V) which are in a form for general use whereas others (II, VI) describe experimental equipment built to test specific principles. The level of documentation reflects this distinction by concentrating on constructional details or underlying concepts, as appropriate.

APPENDIX I

The ADC/DAC Board for the F100M

Appendix I describes the ADC/DAC board for use with the F100M micro-processor system.

Introduction

The board provides a single 8-bit DAC and a single 8-bit ADC with status bits to indicate when a sample is ready and when the sampling rate is too fast. Sampling is controlled by an external input from, for example, a real-time clock. No analogue signal conditioning is provided on the board.

Construction

The board is constructed from a standard 42 DIL-position printed circuit card using predominantly 74-series TTL logic packages. Fig. 1 shows the layout of packages on the board. The two DAC's are Ferranti ZN425E devices which provide 8-bit resolution and a 2.55V precision reference voltage. They also have internal counters but these are not used. The Successive Approximation Register (SAR) is an Advanced Micro Devices Am2502 device. This contains the logic for implementing a successive approximation ADC when used with a DAC and a comparator. The comparator used is a 710 device.

Outline of Operation

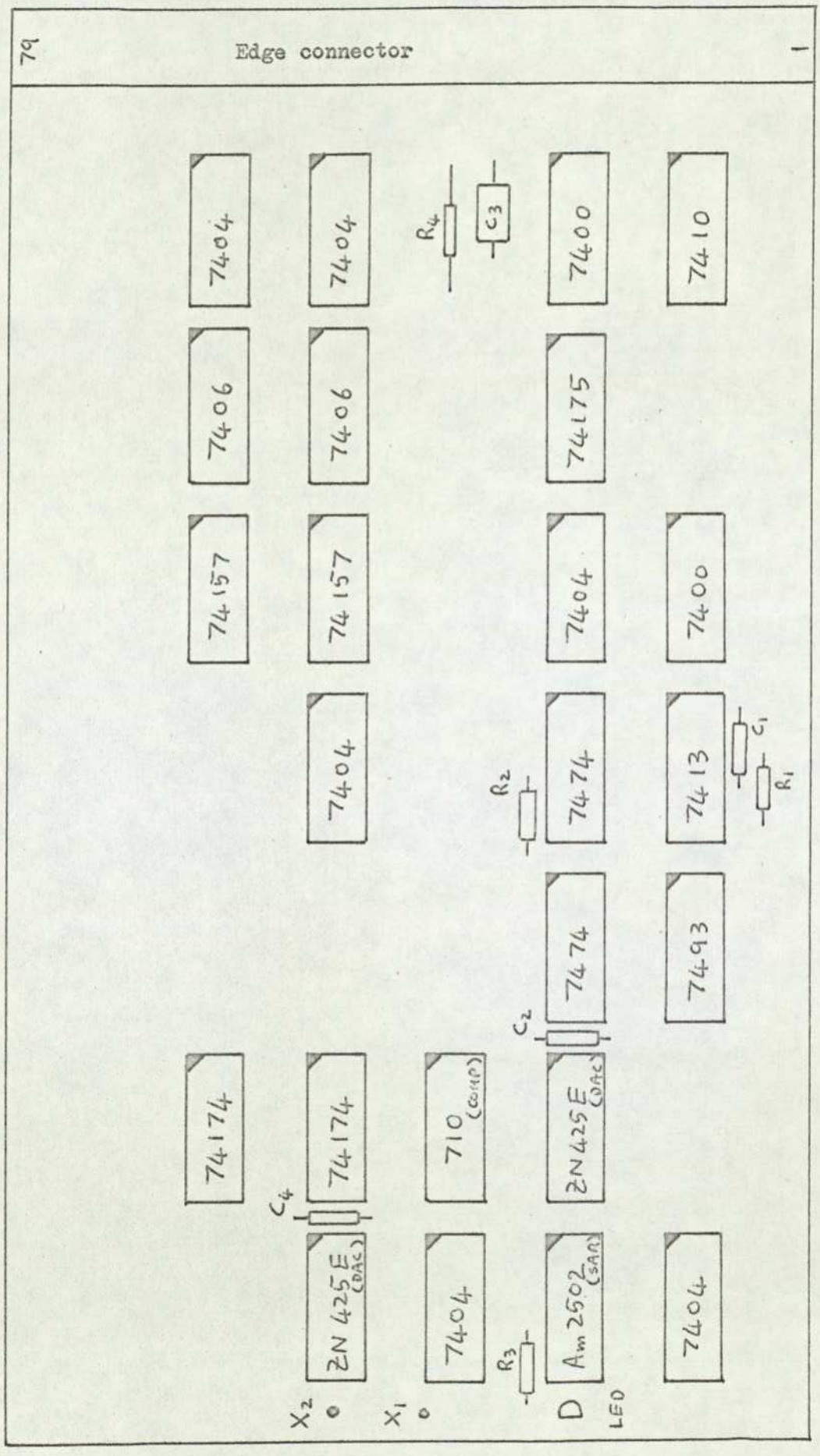
Fig. 2 shows the design of the board. The Schmitt gates (A4) are connected as an oscillator to provide a square wave at approximately 8MHz. This divides down by A5 to provide the 500KHz clock used by the SAR. This allows 2μS for successive DAC values to settle before the next value is applied to the comparator.

The SAR is started by $\overline{\text{strt}}$ and completion is indicated by CyCo. When the sampling pulse sets B5, the conversion is started: CyCo clears B5 and sets B4(a) to indicate that a sample is ready. Subsequent sampling pulses will cause

B4(b) to be set and the LED will be lit indicating two successive sample pulses have occurred. B4(a) is cleared whenever a data read is caused, so that providing sample pulses and reads occur alternately, B4(b) is never set. B4(b) is cleared only by the manual reset signal $\overline{\text{ManRs}}$.

B2 handles the interface control signals and, with SrWt and StAdl, generates the data strobes and multiplexer select and enable signals. D6 and E6 latch values and present them to the DAC D7. D3 and E3 select ADC data or status (bottom two bits only) and drive the output gates D2 and E2. Inputs are buffered through D1 and E1.

Fig. 1. ADC/DAC board - package layout.



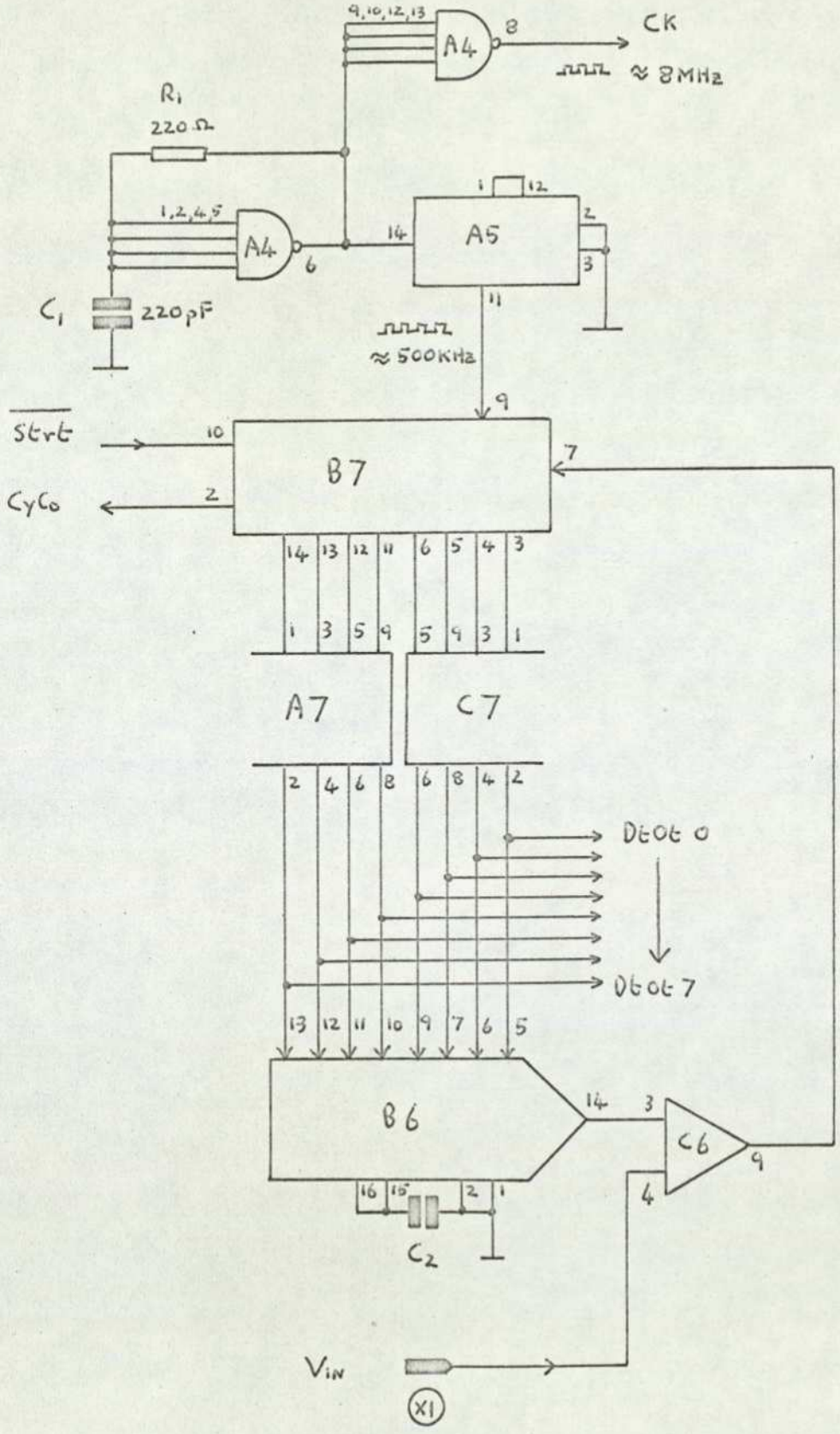


Fig. 2a. ADC and clock generator.

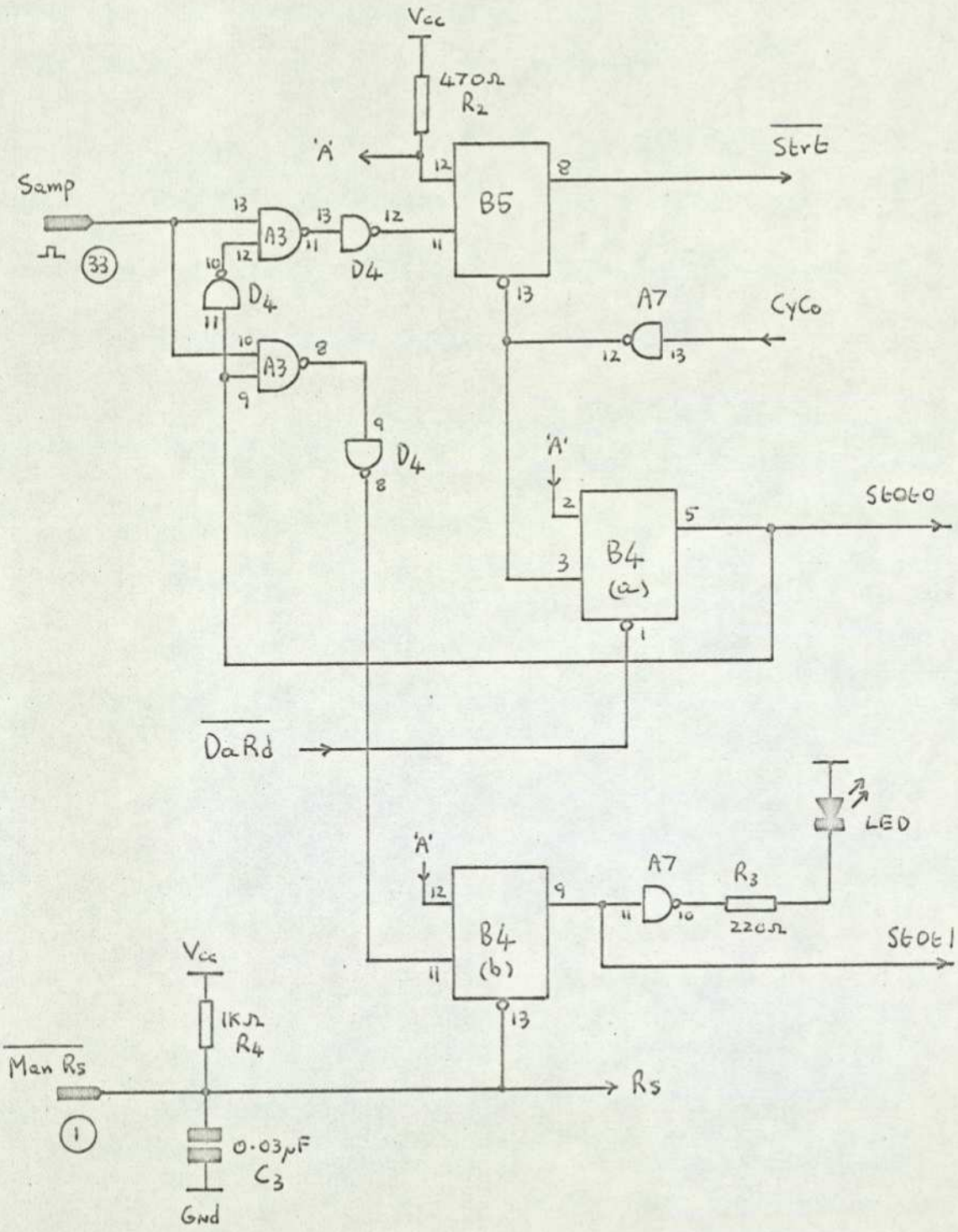


Fig. 2b. ADC sampling logic.

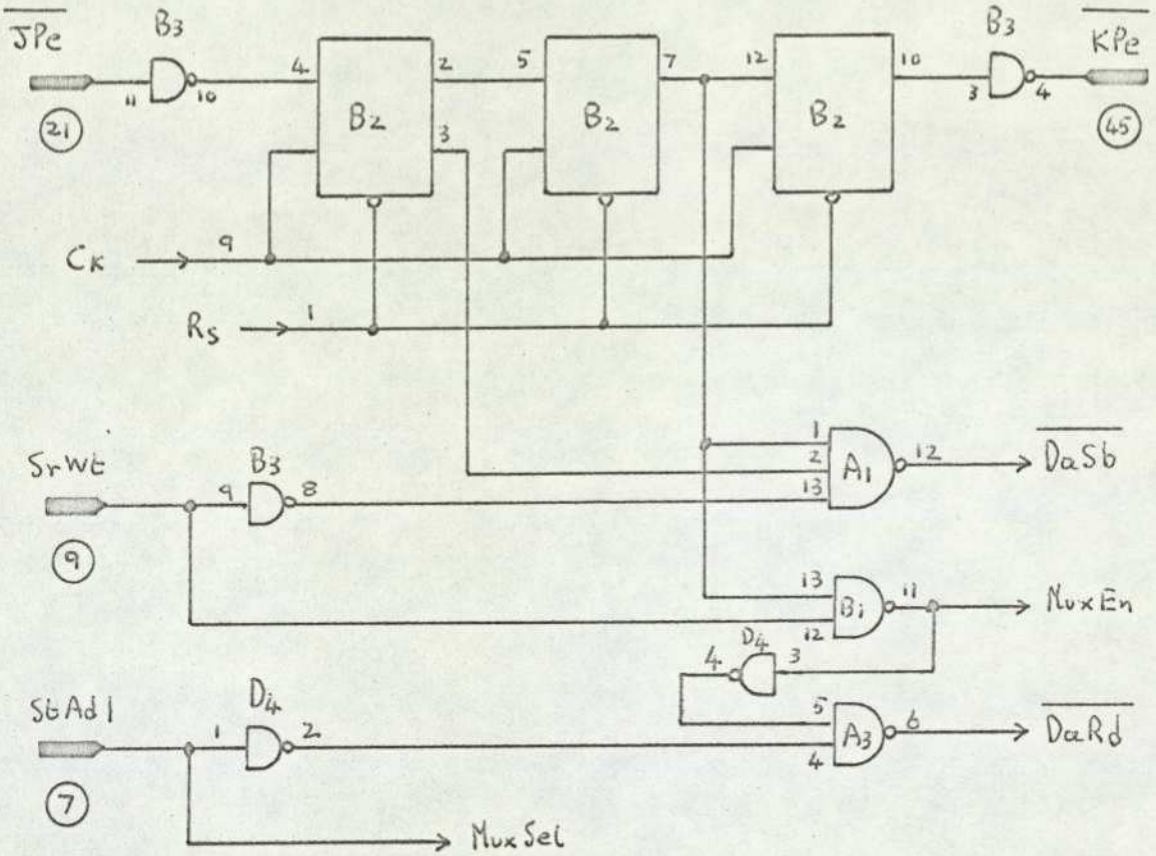


Fig. 2c Interface logic.

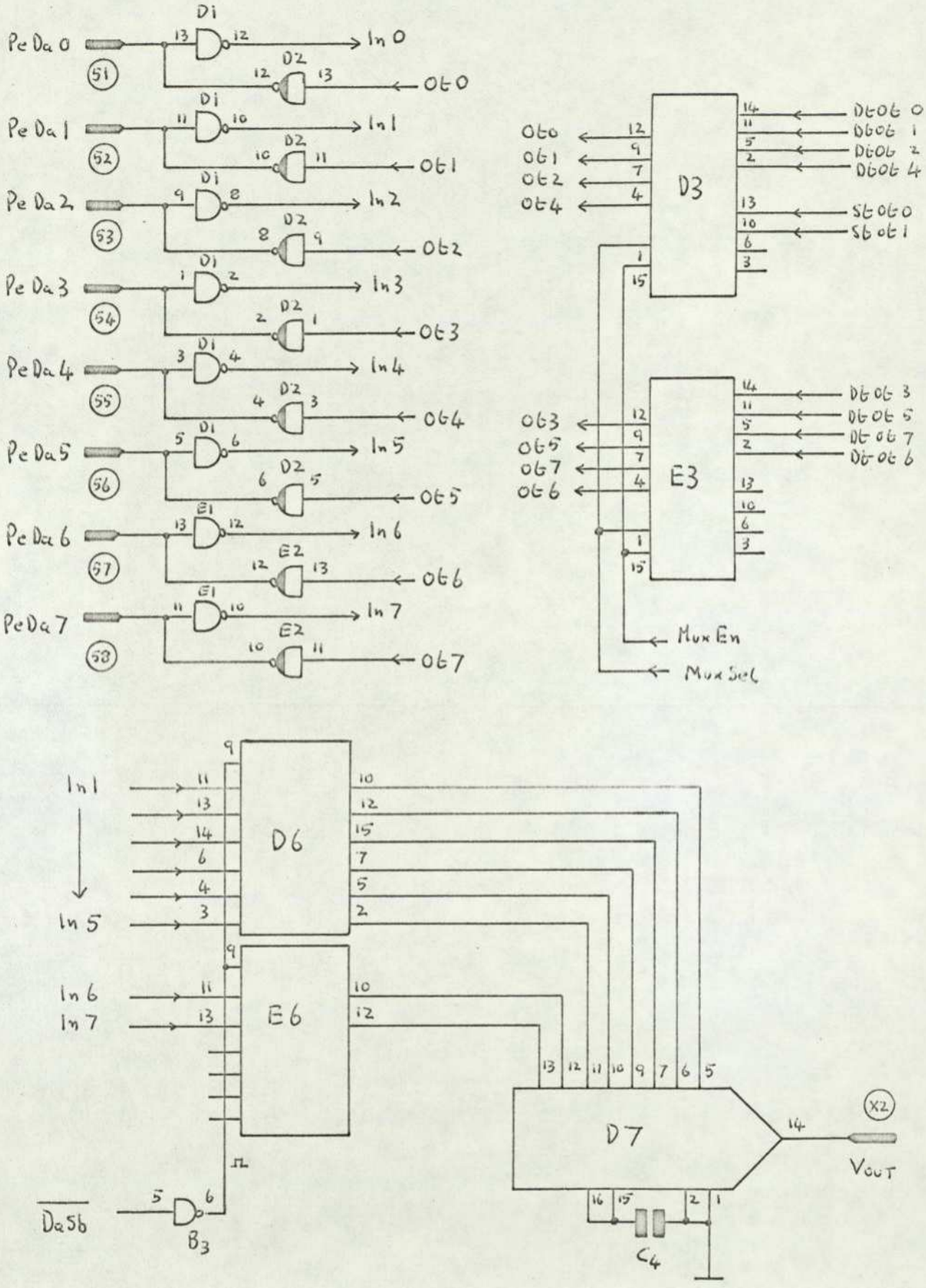


Fig. 2d. DAC latches and bus drivers.

Operating Notes for the ADC/DAC Board

1. Programming

The ADC/DAC board operates on channel 1 of the F100M IOC and therefore occupies addresses 32764 - 32767.

Writing a value to loc. 32764 (data word) causes a voltage proportional to the value of the least significant half of the data to appear on the DAC output.

Reading loc. 32764 causes the ADC value to be loaded into the least significant half of the accumulator. This must be done only when a valid sample is present.

Reading loc. 32766 (status word) causes the two status bits to be loaded into the least significant bits of the accumulator. Bit 0 is set when a sample is ready to be read. Bit 1 is set when the sampling rate is too high, i.e. two samples have been taken without a read instruction between. Bit 0 is cleared by a data word read; bit 1 can be cleared only by a manual reset.

The normal mode of operation for reading ADC samples is shown below.

2. Hardware

The ADC/DAC board fits into slot 13 of the prototype F100M 19" rack.

It takes +5V and +12V supplies from the normal backplane distribution pins. It also requires -6V @ approximately 20mA.

The voltage ranges for both input and output are 0 to +2.55V giving a voltage step of 10mV per binary count. For maximum accuracy the source of the ADC voltage should have an impedance of approximately 10KΩ. This ensures that the comparator inputs are balanced. The load on the DAC output should be greater than approximately 50KΩ.

The sampling pulse should be a positive-going pulse shorter than the sample conversion time, i.e. less than approximately 16μS.

	Man Rs	1	2	
		3	4	
*	St Ad 0	5	6	
	St Ad 1	7	8	
	Sr Wt	9	10	
	Ov	11	12	
	+5v	13	14	
*	Pe Accpt	15	16	
		17	18	
		19	20	
	J Pe	21	22	
	Ov	23	24	
*	Dir	25	26	
*	In Sr Cc	27	28	
*	Q	29	30	
		31	32	
	Samp	33	34	
	Ov	35	36	
		37	38	
*	Pe Rq	39	40	
		41	42	
		43	44	
	K Pe	45	46	
	Ov	47	48	
	+5v	49	50	
	Pe Da 0	51	52	Pe Da 1
	Pe Da 2	53	54	Pe Da 3
	Pe Da 4	55	56	Pe Da 5
	Pe Da 6	57	58	Pe Da 7
	Ov	59	60	
	+12v	61	62	
*	Pe Da 8	63	64	Pe Da 9 *
*	Pe Da 10	65	66	Pe Da 11 *
*	Pe Da 12	67	68	Pe Da 13 *
*	Pe Da 14	69	70	Pe Da 15 *
	Ov	71	72	
	+12v	73	74	
		75	76	
		77	78	
	-6v	79	80	

Fig 3. Edge connections for ADC/DAC. F 100M slot 13
* not used.

APPENDIX II

The Programmable Interface Converter (PIC)

a) Operation and Construction

b) Timing Considerations

Appendix II

The Programmable Interface Converter (PIC)

a) The Operation and Construction of the Prototype

Introduction

The prototype PIC was constructed on two printed circuit cards using standard 74-series TTL packages. Because the prototype is to be used for experimental investigations, it contains monitoring and control functions which would not normally be included in such a device.

Overall Operation

The PIC consists of an 8-bit up/down program counter, a program ROM, instruction register and associated logic. The instruction set is shown below.

Clear output bit P	0	0	0	P	P	P	P	P
Set output bit P	0	0	1	P	P	P	P	P
Wait if input bit S is clear	0	1	0	S	S	S	S	S
Wait if input bit S is set	0	1	1	S	S	S	S	S
Skip if input bit S is clear	1	0	0	S	S	S	S	S
Skip if input bit S is set	1	0	1	S	S	S	S	S
Jump to address a	1	1	a	a	a	a	a	a

The program counter is always incremented when an instruction is fetched. For a skip it is incremented again, while a wait causes it to be decremented so that the instruction is repeated. Jumps cause the jump address to be loaded into the program counter.

Clock Generator and Reset Controls

Figure 1 (a) shows the two phase clock generator, 'Run/Stop' and 'Single-Step' controls and reset circuits. The symmetrical two phase clock is generated by dividing an externally applied square wave (CkIn) by two, and gating the resulting anti-phase signals with the original waveform.

The two waveforms ø1 and ø2 must be capable of being stopped and re-started in such a way that the break always occurs after a complete ø2 pulse and before the next ø1 pulse or correct operation of the PIC will not be maintained. The output of D-type U3b controls the two phases of the clock, and is itself clocked by a pseudo-ø2 signal generated by U6a. The 'Single-Step' switch is 'debounced' by the set-reset circuit U7c and d, and used to set the D-type U4a which has the D input held high so that the circuit acts as an edge-triggered set-reset flip-flop. It is cleared by ø1. The output of the flip-flop is gated with the 'Run' signal (debounced from the run switch by U7a and b), and presented to the D input of the lockout flip-flop U3b.

Figure 2 shows the various waveforms in the circuit during a Run/Stop sequence and during a single-step operation. Note that by means of mechanical 'ganging', the Run/Stop switch is always in the 'Stop' position when single-step is operated, and also single-step is spring loaded to return to its off position when it is not operated.

U1 generates four pulses which control the resetting of various parts of the system. U1 is a power open-collector driver pulled up by R2-R5 to give sufficient drive to control the large number of gates which load it. When power is first applied, C1 is discharged, so the output of U2a is high. As C1 charges through R1, so U2a turns on and the output goes low. Thus a pulse is generated. This also occurs if the 'Manual Reset' button is pushed as C1 is discharged.

The pulse thus generated causes Clx to go low, then high, which clears the instruction register and resets the program counter to zero. The input of U2a is pulled up to five volts and presented as an edge connection, RsSel. If this is left floating (high), then Cla and Clb will pulse each time Clx does, so all the output latches will be cleared to zero. If RsSel is earthed, then PrSe will pulse each time Clx does, so that all the output latches will be preset to zero.

Thus the converter can be used in systems where the natural (quiescent) state of control lines is zero or one by selecting the reset condition using RsSel.

Input Multiplexer and Output Latches

The prototype was built with 16 inputs and outputs although the machine code allows expansion up to 32 of each. Figure 1 (b) shows the input demultiplexer, output decoder and output latches.

The 16 inputs go to U8, a 16 line to one line multiplexer. The four control lines of the multiplexer go to the four least significant bits of the instruction register output. Also connected to these are the control lines of a 16 wa decoder, U9. The 16 outputs of U9 go to the Ck inputs of the 16 output latches so that, by enabling U9 by means OtSt, one output latch can be selectively clocked. All the preset controls of the output latches are joined to form a PrSe signal, and the clear controls are joined in two groups of eight to form Cla and Clb. Latch outputs are buffered by inverters to avoid reflection problems which can arise when latch outputs are connected to long lines. The output latches have their D inputs joined in two groups of eight which connect (one directly, one via buffering) to IR5.

Logic Unit

Figure 1 (c) shows the gating within the logic unit. U18c produces the signal Eq when the value of InSel, the selected input, is identical to the value of Ir5. U19a decodes the 'wait' instruction and, via U2f, decrements the program counter when Eq and ø2 are true. U19b performs a similar function decoding the 'skip' instruction, but its output is combined with ø1 by U18d so that the program counter is incremented at ø2 for a satisfied skip and always at ø1. U20a decodes the 'jump' instruction and leads the program counter from the instruction register at ø2, while U20b decodes the bit 'set' and 'clear' instructions and produces the output strobe pulse OtSb.

Program Counter and Instruction Register

Figure 1 (d) shows the program counter, ROM, instruction register and monitoring logic. U23 and U24 form an eight-bit up-down synchronous counter. LdPgCt, IncPgCt and DecPgCt control the loading, incrementing and decrementing respectively. The six least significant inputs are connected to the output of U21 and U22, the eight-bit instruction register. Thus LdPgCt causes the six least significant bits in the instruction register to be loaded into the counter. The instruction register is clocked by ø1 and cleared by Clx. U25 and U26 form an eight-bit two-way multiplexer and drive eight LED's. The switch 'display select' enables the user to display either instruction register contents (instruction) or the program counter (address) on the eight LED's. The eight inputs of the instruction register are the data outputs of the program ROM. The address inputs for the ROM are the six outputs of the program counter.

Note that when IncPgCt occurs at ø1, IncPgCt is gated with ø1 derived through two inverters from ø1 at U5a. The instruction register is clocked with ø1 from the first of these inverters. This ensures that the 'data hold' requirement of U21/U22 are met, i.e. the ROM output is stable when they are clocked.

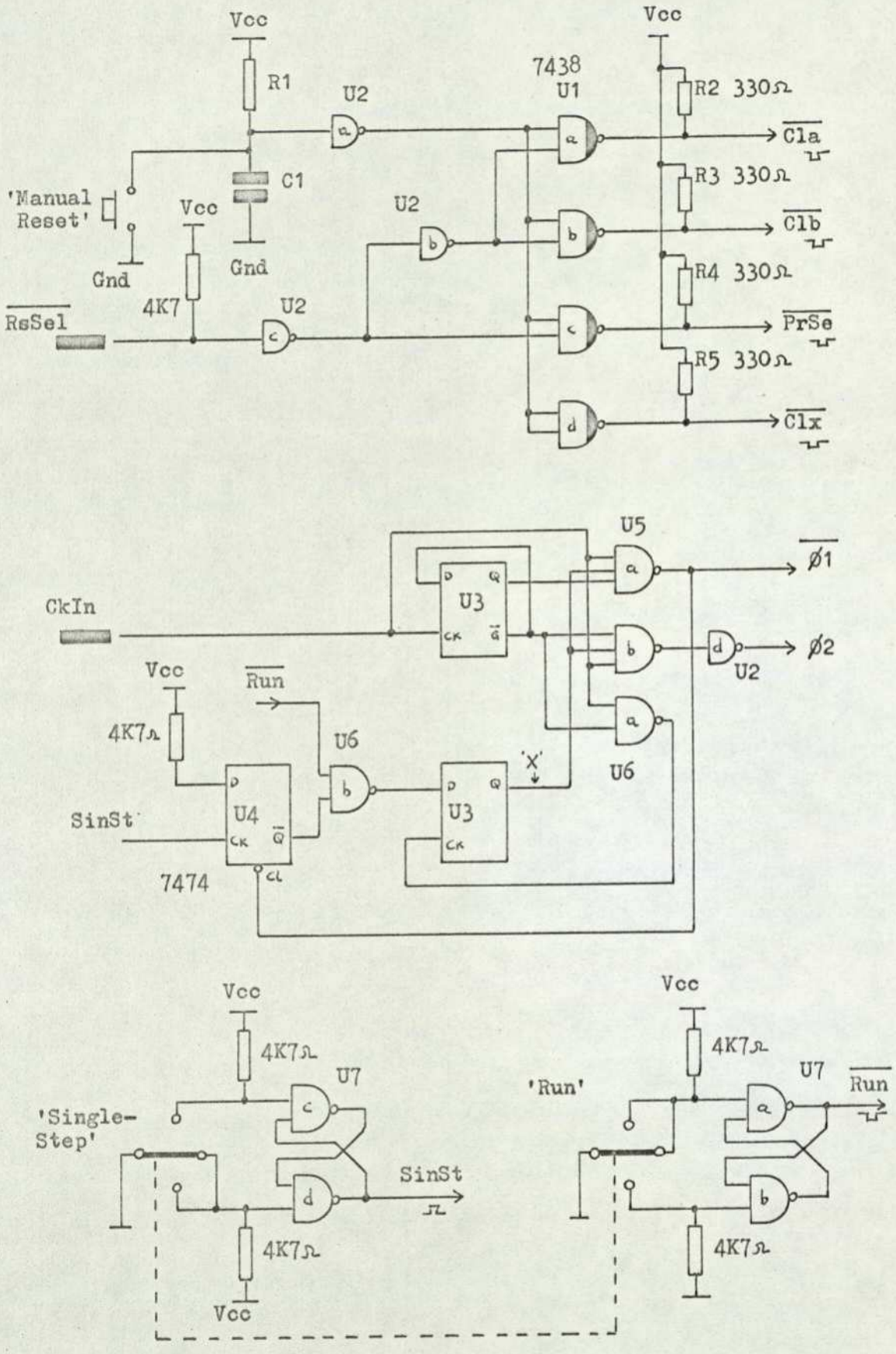


Fig. 1(a). PIC - Clock generator and reset controls.

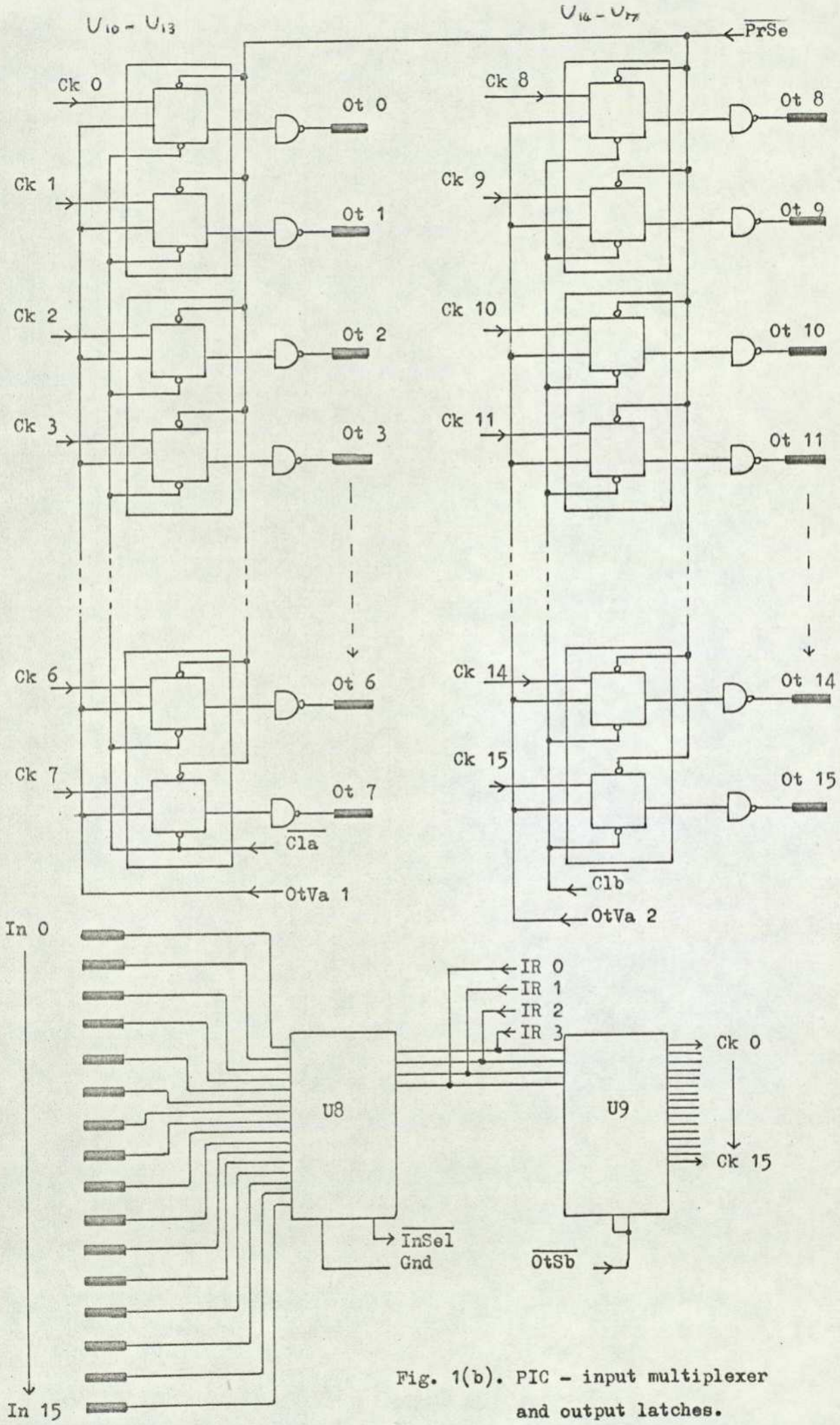


Fig. 1(b). PIC - input multiplexer and output latches.

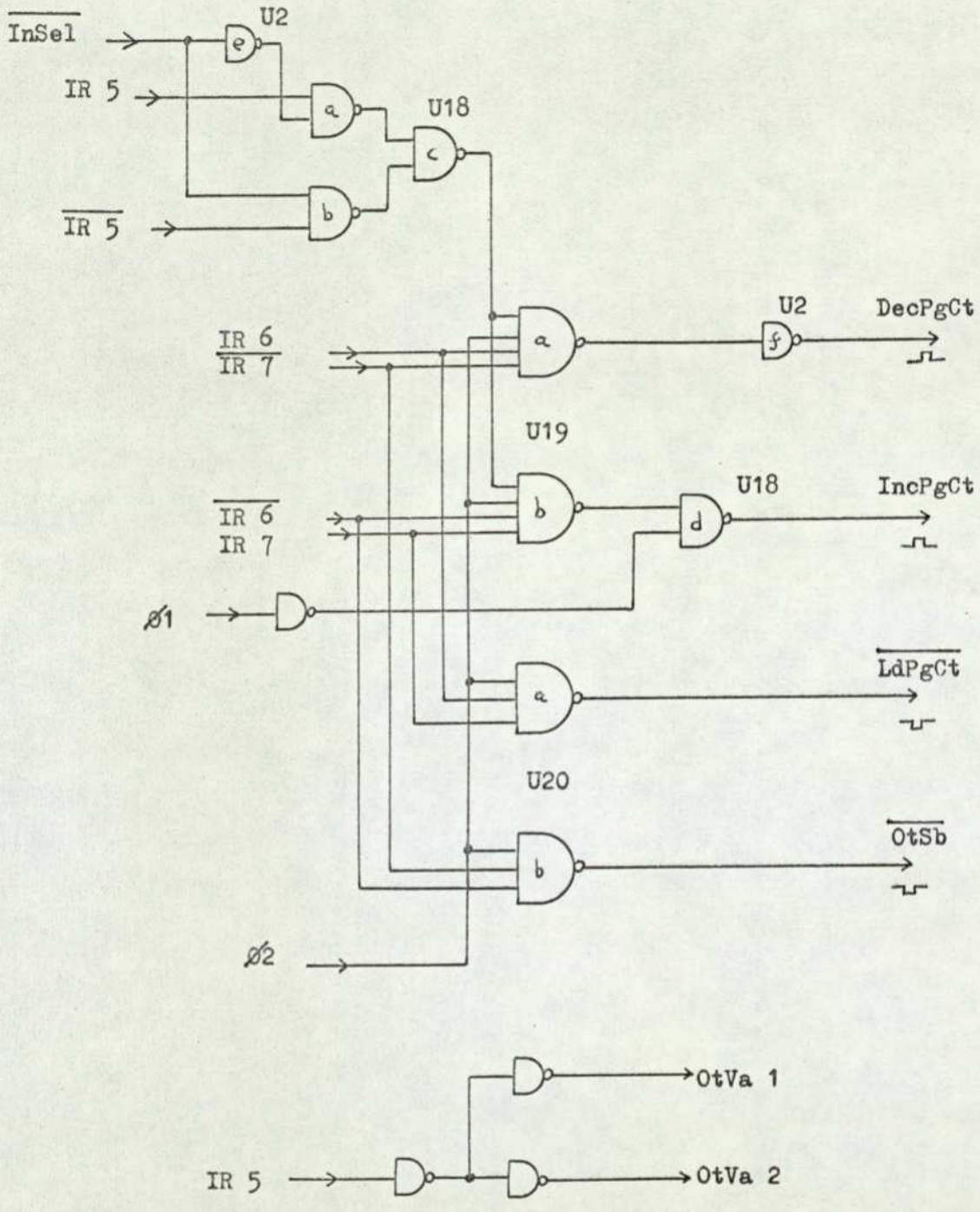


Fig. 1(c). PIC - logic unit.

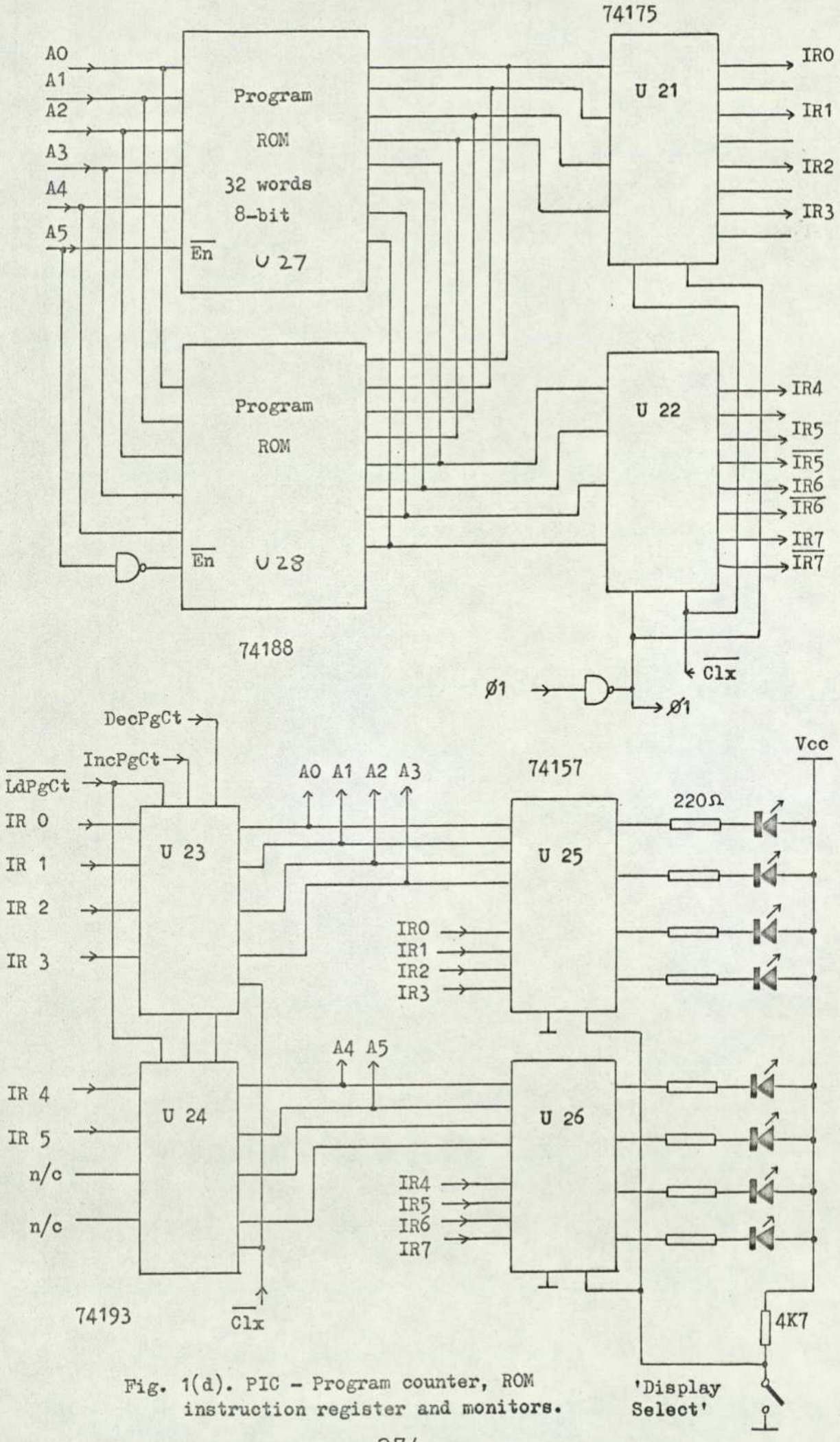


Fig. 1(d). PIC - Program counter, ROM instruction register and monitors.

Logic devices.

U1	7438	drivers
U2	7404	inverters
U3	7474	D type flip-flops
U4	7474	ditto
U5	7410	gates
U6	7400	↓
U7	7400	gates
U8	74150	input selector
U9	74154	output decoder
U10 to U17	7474	D types (output latches)
U18	7400	gates
U19	7420	↓
U20	7410	gates
U21	74175	latches (instruction reg.)
U22	74175	ditto
U23	74193	up-down counter (program counter)
U24	74193	ditto
U25	74157	multiplexer (display driver)
U26	74157	ditto
U27	74188	program ROM
U28	74188	ditto

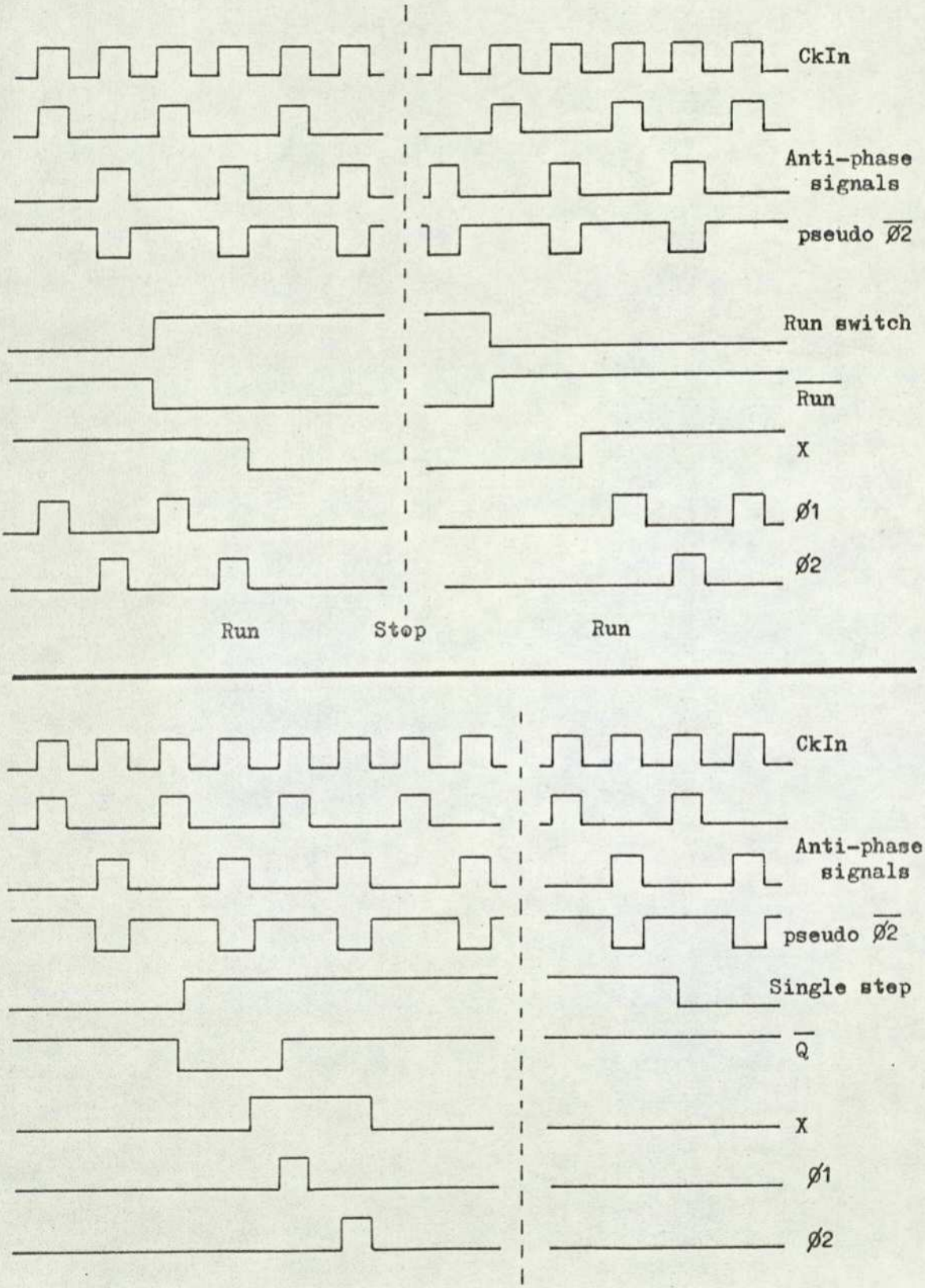


Fig. 2. Clock generator waveforms during run/stop and single step operations.

(b) Timing Considerations

In a high speed device such as the PIC, timing is of the greatest importance. It was originally thought the access time of the program ROM would be the limiting factor, but it was subsequently found that the settling time of the program counter was a more serious factor.

The PIC was built with standard 74-series TTL devices, as it was felt that the power consumption of an ECL or 74S-series implementation would be excessive.

Timing is controlled by a two-phase clock $\phi 1$ and $\phi 2$. In the final design, as described in appendix II (a), only the leading edges of the clock waveforms are used, so a single-phase clock could be used with events synchronised to either the leading or trailing edges. It was felt during the design that the greater flexibility afforded by a multi-phase clock could be used to advantage in speeding up the PIC design.

Fig. 3 shows the elements that make up the delays between $\phi 1$ and $\phi 2$ and between $\phi 2$ and $\phi 1$. Table 1 shows the delays of the devices used in the PIC. From Fig. 3 it will be seen that:

Delay $\phi 1$ to $\phi 2$ = IR settling time + input value selection time + instruction decade time
= 20 (30) = 23 (35) = 3 times 10 (19)
= 73 (122) nS

Note average gate delay = $\frac{1}{2}$ (t_{Ghl} + t_{Glh}) = 10 (19)

Delay $\phi 2$ to $\phi 1$ = PC settling time + ROM access + IR latch set up time
= 47 (71) + 34 (50) + 20
= 100 (141) nS

Assuming a symmetrical clock the minimum cycle time of the PIC is 200 nS (average) or 280 nS (worst case). In practice it would be slower than this since not all gate delays have been dealt with.

The prototype achieved a speed of 250 nS/cycle which accords well with the average speed calculations.

It is not clear what is responsible for the variation between the manufacturers' average and worst case delays for logic devices, e.g. are the spreads for one device over a range of temperatures or the production variations between devices or a combination of the two? Without this information it is difficult to decide how close to design to the 'average' figures.

It is felt that even with an extensive re-design of the PIC (e.g. use of a non-symmetrical clock, use of a combinational incrementer instead of a program counter, etc.), it will be difficult to get the cycle time down to about 100 nS - one of the original design goals.

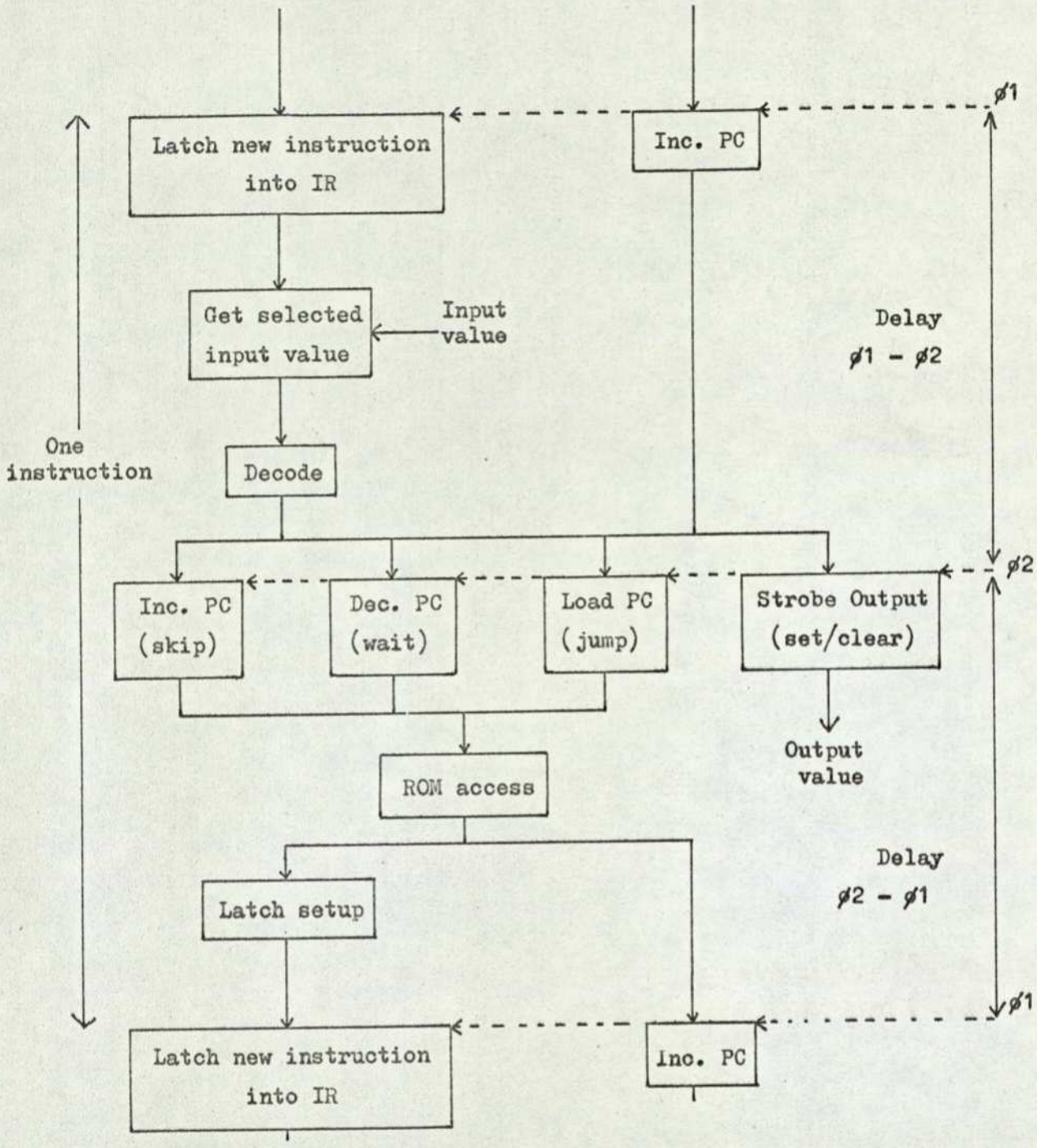


Fig. 3. Timing diagram for the PIC.

Delay times of PIC devices.

Instruction register latch - 74175		
t_{Lst}	data stable time before clock edge	20 nS min.
t_{Lhd}	data stable time after clock edge	5 nS min.
t_{Lq}	delay from clock edge to Q output	20 (30) nS[1]
t_{Lwd}	clock pulse width	20 nS min.
t_{Lf}	time between clock edges[2]	40 nS min.
Program counter - 74193		
t_{Cwd}	clock pulse width	20 nS min.
t_{Cc}	delay from clock to carry or borrow	16 (24) nS
t_{Cq}	delay from clock edge to Q output	31 (47) nS
t_{Cst}	data stable time before load pulse	20 nS min.
t_{Chd}	data stable time after load pulse	0 nS min.
t_{Cf}	time between clock edges[2]	40 nS min.
Decoding gates - 7400/7404/7410/7420		
t_{Ghl}	delay from input to output, high-to-low	8 (15) nS
t_{Glh}	delay from input to output, low-to-high	12 (22) nS
Input selector - 74150		
t_{Ss}	delay from select control to output	23 (35) nS
t_{Sd}	delay from data inputs to output	13 (20) nS
Program ROM - 74188		
t_{Rad}	delay from address to data output	31 (50) nS
t_{Rsl}	delay from select to data output	34 (50) nS

Notes :

1. Average time (maximum time).
2. At maximum clock frequency.
3. The settling time for the 8-bit program counter is $t_{Cc} + t_{Cq}$ i.e. 47 (71) nS.

Table 1.

APPENDIX III

The FM1600B Breakpoint Facility

Appendix III - The FM1600B Breakpoint Facility

INTRODUCTION

During the testing of the DIXPAC assembler and de-assembler it was realised that the main computer (FM1600B) lacked facilities for tracing program execution errors. The only facility available was the 'query' routine in which certain statements are marked such that when they are encountered, an identifier and the value of the expression represented by the statement are output. Apart from the limited amount of information provided, the query has to be marked at assembly time so that extra instructions can be inserted. When a program is deemed to be operating correctly, it must be re-assembled without queries to produce a normal compiled form.

The breakpoint facility, comprising two subroutines and three data areas, was written by the author to enable monitoring points ('breakpoints') to be inserted in assembled code and removed again, without recourse to the assembler.

MODE OF OPERATION

A breakpoint is a point in a compiled program where execution is suspended and control is transferred to a monitoring program, the breakpoint monitor. This program allows the user to ascertain the state of registers, index register and condition stats. at the point where execution was suspended. The program can be resumed from that point until the next breakpoint is encountered. The breakpoint system, therefore, comprises two distinct parts: a routine to modify the compiled code so that the breakpoint is marked, and the breakpoint monitor to control actions when the breakpoint is encountered.

THE BREAKPOINT INSERTION AND DELETING

The breakpoint insert/delete routine is called from an operating system - usually the machine code editor. It has three functions:

1. To replace machine code at a specified address with a subroutine call to the breakpoint monitor.

2. To store the code that was overwritten in (1) in a form that can be executed from the end of the breakpoint monitor (this is to be explained).

3. To record the address, type and number (an arbitrary identifier to distinguish various breakpoints) of the breakpoint so that the monitor can identify which one has been encountered.

The first point is straightforward and simply involves the overwriting of the specified address by the subroutine call. The second point is more difficult. When the subroutine call is encountered, the breakpoint monitor is entered and the user examines or modifies registers. When this is finished, then he resumes program execution. The next code to be executed must be that which was originally where the subroutine call is now. The easiest way of achieving this is to re-write the original code into the space where the subroutine call is, and jump to it. In this way program execution is resumed correctly. However, in so doing, the breakpoint has been deleted which may or may not be desirable.

The system used in the breakpoint routine is to relocate the replaced code so that it can execute from a different place in store, namely the end of the breakpoint monitor. The code is followed by a jump to the location after the breakpoint. This code is then executed on leaving the breakpoint monitor. This ensures that the replaced code is executed, the breakpoint remains intact and program execution is resumed correctly.

```
1102    V6  = V6 + 1
1103    V12 = V6  R , L
1104    N1  = V6
1105    V12 = VN1 + V3
```

Piece of compiled code (FIXPAC equivalent of m/c) with addresses in store.

```
1102    V6  = V6 + 1
1103    →S10053
1104    N1  = V6
1105    V12 = VN1 + V3
```

Code with a breakpoint (call to breakpoint monitor) at location 1103.

```
     Breakpoint
      Monitor                    End of breakpoint monitor
last word of monitor

V12 = V6  R , L                  (replaced word)

→1104                            (jump to next location)
```

This is a simple example since the replaced code is a single word instruction with no jumping addresses. The existence of two word instructions and jumps (both absolute and relative) considerably complicate matters.

Two word instructions mean that both words are transferred to the end of the monitor for execution. The insert/delete routine has to ascertain from the first word if a two word instruction is involved. The placing of breakpoints on the second word of a two word instruction is not allowed, nor is it a sensible thing to do.

Absolute jumps can be relocated without change while relative jumps are changed so that, if satisfied, they jump to absolute jumps which transfer control to the original jump destinations.

Single Word Instruction

```
1122    V16 = V16 + VN1          Breakpoint here
1123    V3  = V3 + 1

        V16 = V16 + VN1          Modified code
        → 1123
```

Two Word Instruction

```
1122    V16 = V16 + VNO          Breakpoint here
1123    + 120
1124    V3  = V3 + 1

        V16 = V16 + VNO
        + 120                    Modified code
        → 1124
```

Absolute Jump

```
1122      → 1380                    Breakpoint here
1123      V3  = V3 + 1

          →1380                     Modified code
```

Relative Conditional Jump (One Word)

```
1122      V3  = V3 + 1
1123      → - 2, V3 = 0             Breakpoint here
1124      V6  = V6 + VN1
```

(A relative address of - 2 means execute the previous instruction.)

```
          → + 1, V3 = 0
          → 1124                    Modified code
          → 1122
```

If the condition is not met, then the next instruction encountered will be jump to 1124 so control will continue downwards. If the condition is met, then the jump to 1122 will be encountered and the previous instruction will be repeated.

Relative Conditional Jump (Two Word)

```
1122      V3  = V3 + 1
1123      → - 2, V3 + VN0 = 0
1124      + 86
1125      V6  = V6 + VN1

          → + 2, V3 + VN0 = 0
          + 86                      Modified code
          →1125
          →1122
```

It will be seen that up to four words of modified code can be produced for each breakpoint. Up to thirty-two breakpoints can exist at any one time and a four word block is reserved for each in the data area S10052.

Thus the second action of the insert/delete routine is to produce a block of up to four instructions, as outlined, and store this in the appropriate place in S10052.

The third action involves making an entry in a table S10051 which contains the following information:

1. The address (absolute location) of the breakpoint.

2. The number (an identifier 0 - 31) of the breakpoint.

3. The breakpoint type.

Three types of breakpoint, causing different actions when encountered, are allowed. They are known as the 'E' type or 'examine', the 'P' type or 'print' and the 'S' type or 'skip'.

'E' Type Breakpoints

Upon entering the monitor the breakpoint number is printed as is the total number of times the breakpoint has been encountered. The user is then invited to examine the contents of the registers.

'P' Type Breakpoints

The occurrence and identifer numbers are printed as above but program execution is immediately resumed.

'S' Type Breakpoints

The occurrence number is incremented and recorded but no printed output is produced. Execution is immediately resumed.

The insert/delete routine also records the original 'untouched' code which was replaced, in S10054. This is used when the breakpoint is deleted so that the exact code can be replaced.

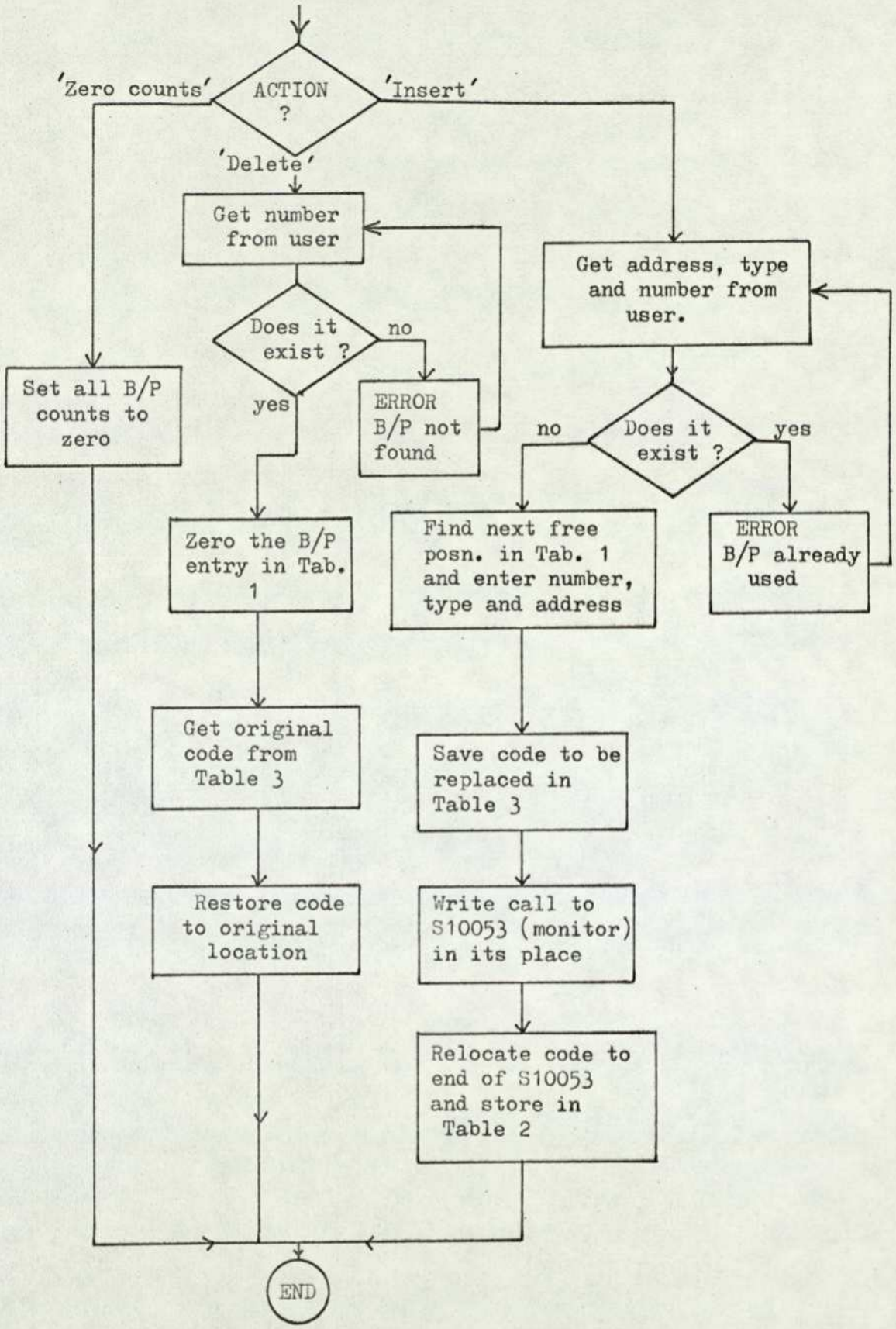


Fig 1 Overall View of S 10050, the Breakpoint Insert/Delete/Zero Routine.

S10050 Insert Breakpoint

Function

Inserts, deletes and zeros counters for breakpoints.

Entries

+0	Insert breakpoint
+1	Delete breakpoint
+2	Zero breakpoint counters

Entry Conditions

+0	V23 = breakpoint number 0 to 31 V22 = breakpoint address
+1	V23 = breakpoint number 0 to 31
+2	None

Result

+0	Jump to breakpoint monitor is inserted in code Displaced code is saved in S10054 Modified code saved in S10052 Breakpoint details stored in S10051 Breakpoint number marked as 'in use' Message 'INSERTED' is printed
+1	Breakpoint deleted and replaced with code from S10054 Breakpoint number marked as 'free' Breakpoint details scrubbed from S10051 Message 'DELETED' is printed
+2	All breakpoint counts in S10051 are zeroed Messaged 'ZEROED' is printed

User Interaction

+0	On the prompt 'B/P TYPE?' the user enters E, S or P to define breakpoint type
+1	None
+2	None

Error Exits

+0 If breakpoint type is not E, S or P, then 'TYPE NOT RECOGNISED: RE-ENTER NEW TYPE' is printed
If breakpoint number is already in use, then 'B/P NUMBER ALREADY USED: TRY A DIFFERENT ONE' is printed before exiting subroutine

+1 If breakpoint does not exist, then 'B/P NOT FOUND: TRY AGAIN' is printed on exit

+2 None

Registers Destroyed

None

Internal Storage

AL 60 is a two-word block with 32 flags to indicate which breakpoints are 'free'.

AL 80 points to the next free space in S10051.

Link Nest Depth

5 (maximum)

Other Routines used

S10051, S10052, S10054, S555, TEXT

Storage

346 words

Input-Output

Input - stream 5
Output - stream 4

Stops

None

```
N
S10050 INSERT B/P

U

      ←0                              'INSERT'
      ←24                             'DELETE'
      ←32                             'ZERO COUNTERS'
[0]   VN3=N1,N3-1
      VN3=N2,N3-1
      VN3=V21,N3-1
      VN3=V20,N3-1
      V20=V23
      N1=VN0
      AL60
      ←40,V23-24>0
      ←41,VN1[V23]=0
      ←42

[40]  V23=V23-24
      V0=V0,N1+1
      ←41,VN1[V23]=0
      ←42

[41]  V23=V20
      N2=VN0                          ' TABLE 1 ENTRIES COUNT '
      AL80
      N1=VN0
      AS10051                         ' TABLE 1 '
      ←50,N1=N1+VN2<0     ?20         ' N1 TO NEXT FREE POSN. '
      VN1=V22&VN0
      OCT00177777                     ' MASK ADDRESS '
      V23=V23&31                      ' MASK COUNT '
      V23=V23[[16]],L
      VN1=VN1+V23                     ' SET UP WORD 1 '
      V0=V0,N1+1
      VN1=0
      V0=V0,N1+1
      VN1=VN0
      -1
      V0=V0,N1-1
      V19=4
      TEXT
B/P TYPE ?
[50]  V19=5
      ←S555,1                         ' READ TYPE '
      V21[7]=0        ?21
      ←51,V21≠VN0=0
      +69                             ' "E" ? '
      ←52,V21≠VN0=0
      +83                             ' "S" ? '
      ←53,V21≠VN0=0
      +80                             ' "P" ? '
      V19=4       ?22
      TEXT 1P
TYPE NOT RECOGNISED : RE-ENTER NEW TYPE
      ←50
[51]  V21=0       ?23
      ←55
[52]  V21=1       ?24
      ←55
```

```
[53] V21=2     ?25
[55] V21=V21[[20]],L
     VN1=VN1+V21                    ' ENTER TYPE NO. '
     N2=VN0
     AL80
     VN2=VN2+2                      ' UPDATE ENTRIES COUNT '

     V23=V23[[16]],R                ' B/P NO. '
     V20=V23
     N1=VN0
     AL60
     ←35,V23-24>0
     ←36
[35] V23=V23-24
     V0=V0,N1+1
[36] VN1[V23]#0
     V23=V20
     N1=VN0
     AS10054                        ' UNTOUCHED FIDS CODE '
     ←55,N1=N1+V23<0
     N2=V22
     VN1=VN2                        ' COPY UNTOUCHED CODE '
     N1=VN0
     AS10052                        ' MODIFIED CODE '
     V23=V23[[2]],L                 ' 4* B/P NO. '
     ←1,N1=N1+V23<0
     V23=0

     V22=VN2&VN0                    ' MASK OUT A FIELD '
     OCT00076000
      ←1,V22#VN0#0                  ' JMP IF NOT 28 '
     OCT00070000
     V23[0]#0       ?1             ' FLAG 0 '
[1]  V22=VN2&VN0                    ' MASK OUT B FIELD '
     OCT00001740
     ←2,V22#VN0#0                   ' JMP IF NOT 28 '
     OCT00001600
     V23[1]#0       ?2             ' FLAG 1 '
[2]  V22=VN2&VN0                    ' MASK OUT C FIELD '
     OCT00000037
     ←3,V22#28#0                    ' JMP IF NOT 28 '
     ←3,VN2[18]#0                   ' JMP IF D = 1 '
     V23[2]#0       ?3             ' FLAG 2 '
[3]  V22=VN2&VN0                    ' MASK OUT I FIELD '
     OCT06000000
     ←4,V22#0                       ' JMP IF I NOT 0 '
     V23[3]#0       ?4             ' FLAG 3 '
[4]  V22=VN2&VN0                    ' MASK OUT F FIELD '
     OCT70000000
     ←5,V22#VN0#0                   ' JMP IF F NOT 7 '
     OCT70000000
     V23[4]#0       ?5             ' FLAG 4 '
[5]  ←6,V22#VN0#0                   ' JMP IF F NOT 6 '
     OCT60000000
     V23[5]#0       ?6             ' FLAG 5 '
[6]  ←7,VN2[16]#0                   ' JMP IF S1 NOT 0 '
     V23[6]#0                       ' FLAG 6 '
[7]  V0=V0
```

```
     V19=4
     TEXT 1P
INSERTED
     V22=VN2&VN0                    ' MASK OUT A FIELD '
     OCT00076000
     V22=V22[[9]],L
     V22=V22[19],R    ?8            ' V22= SIGNED A VALUE '
     V22=N2+V22
     V22=V22&VN0
     OCT00177777
     V22=V22+VN0                    ' V22= FIDS  ←N2+A '
     OCT70400000
     V20=VN2&VN0                    ' BLANK OUT A '
     OCT77701777
     V21=V20+VN0                    ' V21= FIDS COPY BUT A=+1 '
     OCT00002000
     V20=V20+VN0
     OCT 00004000                   ' V29= FIDS COPY BUT A=+2 '
     V19=N2+1         ?9
     V19=V19&VN0
     OCT00177777
     V19=V19+VN0                    ' V19= FIDS  ←N2+1 '
     OCT70400000



     ←8,V23[5]≠0
     ←9,V23[6]≠0
     ←10,V23[3]≠0
[8]  ←12,V23[2]≠0      ?10

     ←12,V23[1]≠0
     ←12,V23[0]≠0
     ←13


[9]  ←13,V23[3]≠0      ?11
     ←8

[32] VN3=N1,N3-1                    ' ZERO COUNTERS'
     VN3=N2,N3-1
     VN3=V21,N3-1
     VN3=V20,N3-1
     N1=VN0
     AS10051
     V20=-32,N1+1                   ' SKIP FIRST ENTRY '
 B3] VN1=VN1&VN0                    ' ZERO COUNTERS '
     OCT77600000
     ←33,N1=N1+2<0
     ←33,V20=V20+1<0
     V19=4
     TEXT 1P
ZEROED
     ←30
```

```
[42] V19=4
     TEXT 1P
B/P NUMBER ALREADY USED : TRY A DIFFERENT ONE
     ←30


[60] +0
     +0




[80]  +0


[10] ←11,V23[4]#0          ?12
     ←14,V23[2]#0
     ←14,V23[1]#0
     ←15


[11] ←12,V23[2]#0          ?13
     ←12,V23[1]#0
     ←12,V23[0]#0
     ←13


[24]  VN3=N1,N3-1
     VN3=N2,N3-1
     VN3=V21,N3-1
     VN3=V20,N3-1
     V20=V23
     N1=VN0
     AL60
     ←37,V23-24>0
     ←38
[37] V23=V23-24
     V0=V0,N1+1
[38] VN1[V23]=0
     V23=V20
     V20=-32
     V22=V23&31                    ' MASK B/P NO. '
     V22=V22[[16]],L
     N1=VN0
     AS10051

[21] V21=VN1&VN0
     OCT07600000
     ←22,V21#V22=0
     ←21,N1=N1+2<0
     ←21,V20=V20+1<0
     V19=4
     TEXT 1P
     B/P NOT FOUND - TRY AGAIN
     ←30
```

```
[22] V20=N1
     V21=VN1&VN0
     OCT00177777
     N1=V21
     V22=V22[[16]],R
     N2=VN0
     AS10054
     ←22,N2=N2+V22<0
     VN1=VN2                    ' RE-WRITE FIDS CODE '

     N1=V20
     VN1=0
     V0=V0,N1+1                 ' ZERO BLOCK 1 '
     VN1=0
     V20=N1-1
     N2=V20
[23] VN2=V[N2+2]
     ←25,VN2#VN0#0
     -1
     N1=VN0
     AL80
     VN1=VN1-2
     V19=4
     TEXT 1P
DELETED
     ←30

[25] V0=V0,N2+1
     ←23

[12] V23=VN0
     AS10053
     VN1=VN2
     VN2=V23+VN0,N1+1
     OCT70000000
     V0=V0,N2+1
     VN1=VN2
     V0=V0,N1+1
     VN1=V19+1
     ←30

[13] V23=VN0
     AS10053
     VN1=VN2          ?15
     VN2=V23+VN0,N1+1
     OCT70000000
     VN1=V19
     ←30
```

```
[14] V23=VN0
     AS10053
     VN1=V20
     VN2=V23+VN0,N1+1
     OCT70000000     ?16
     V0=V0,N2+1
     VN1=VN2
     V0=V0,N1+1
     VN1=V19+1
     V0=V0,N1+1
     VN1=V22+1
     ←30


[15] V23=VN0
     AS10053
    VN1=V21     ?17
     VN2=V23+VN0
     OCT70000000
     V0=V0,N1+1
     V0=V0,N2+1
     VN1=V19
     V0=V0,N1+1
     VN1=V22+1
     ←30


[30] V20=VN3,N3+1
     V21=VN3,N3+1
     N2=VN3
     V0=V0,N3+1
     N1=VN3
     V0=V0,N3+1
     ←L
END
S
10050 0
S10050 INSERT B/P;
```

S10051 Breakpoint Details Table

Storage

64 words

Setup by

S10050
Can be modified by S10053

Used by

S10050, S10053

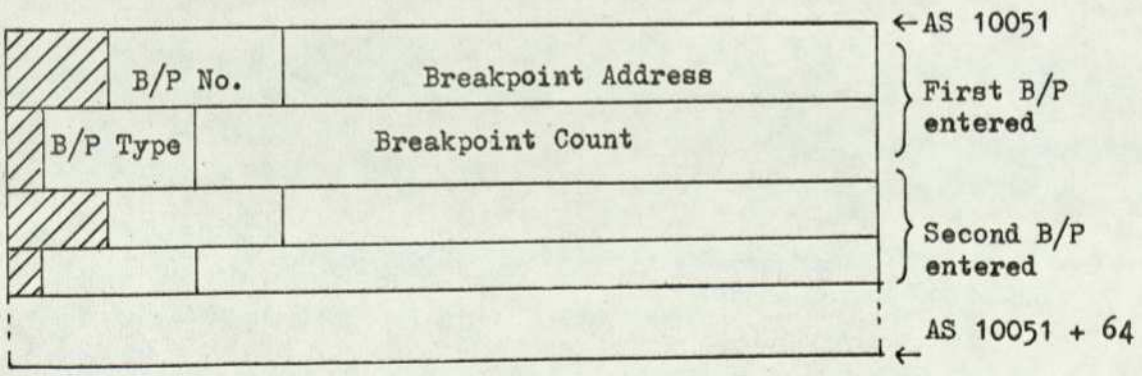
Markers

S10050 AL 80 points to next free position in S10051

Format

Two words for each breakpoint entered in the order in which they were inserted by the user. When a breakpoint is deleted, the following entries are moved up two words to fill the space and the 'next free position' marker is decremented twice.

B/P Type is 0 - E
1 - S
2 - P



S10052 Modified Code

Storage

128 words

Setup by

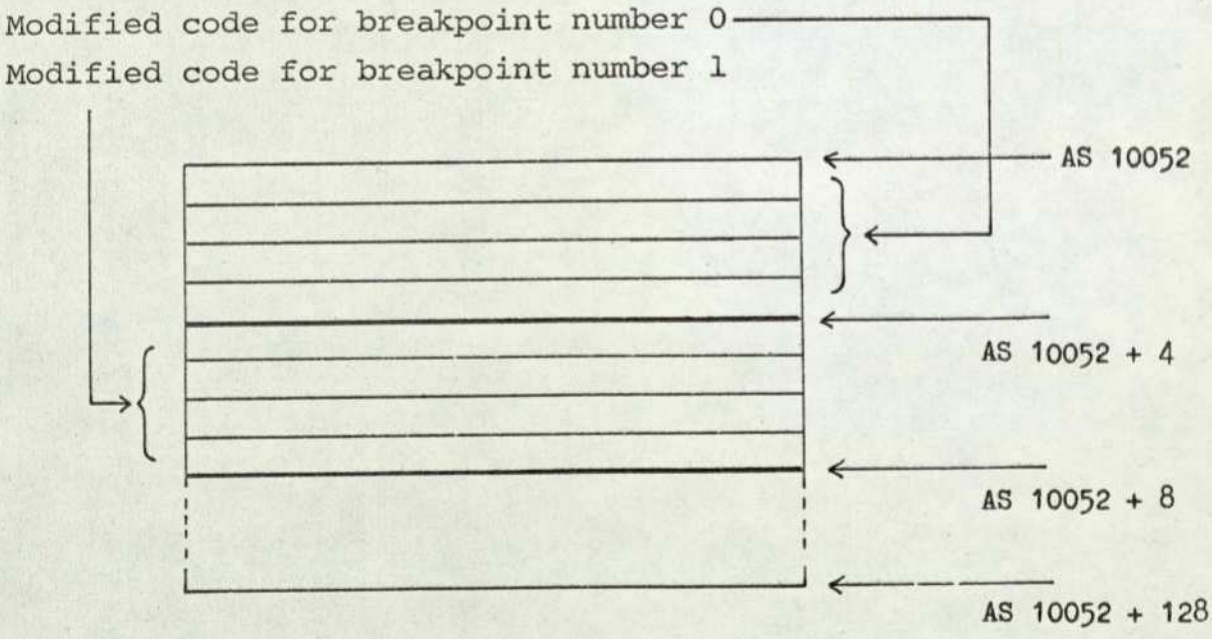
S10050

Used by

S10053

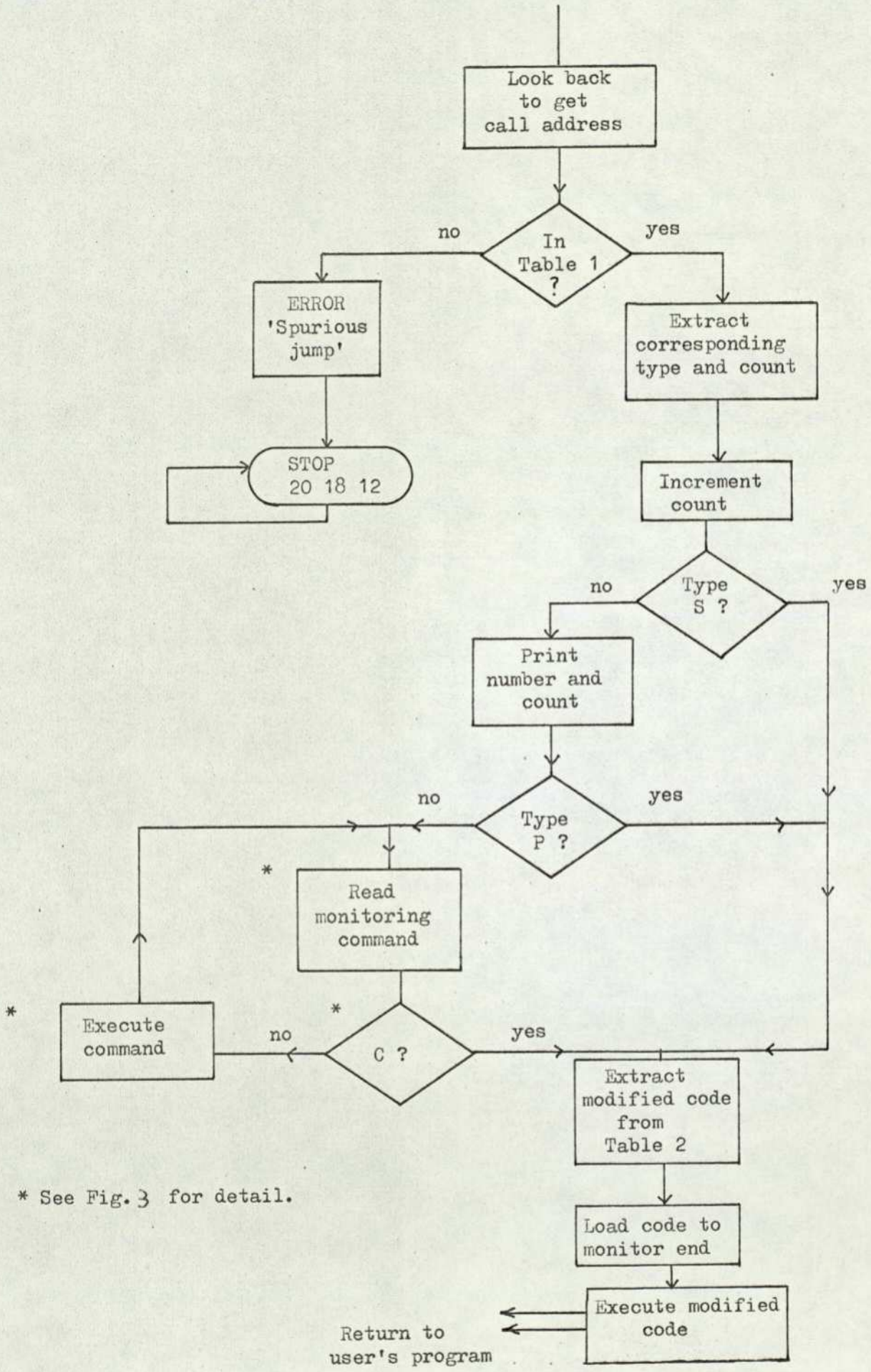
Format

Up to four words for each breakpoint being the modified code which the monitor executes to return to the user program. Each block is entered in a position corresponding to its breakpoint number.



THE BREAKPOINT MONITOR

The breakpoint monitor is called when the breakpoint is encountered. The address from which the call came is found by looking back into the link nest and this is compared with the address in table 1 in order to identify which breakpoint it is. If no breakpoint is found, then the error message 'ERROR - SPURIOUS JUMP AT' followed by the call address, is printed. The appropriate breakpoint count is incremented and the type is ascertained. If this is 'P' or 'E', then the breakpoint number is output together with the count. If the type is 'E', then the user is invited to input an examine or modify command. After this the command 'c' (continue) causes the modified code for the breakpoint in table Z to be copied to an area at the end of the monitor. Waiting registers are restored and the monitor is left via the modified code. 'S' types jump directly to the restoring of the code while 'P' types do so after printing the number and count values.



* See Fig. 3 for detail.

Fig. 2 Overall View of S 10053, the Breakpoint Monitor.

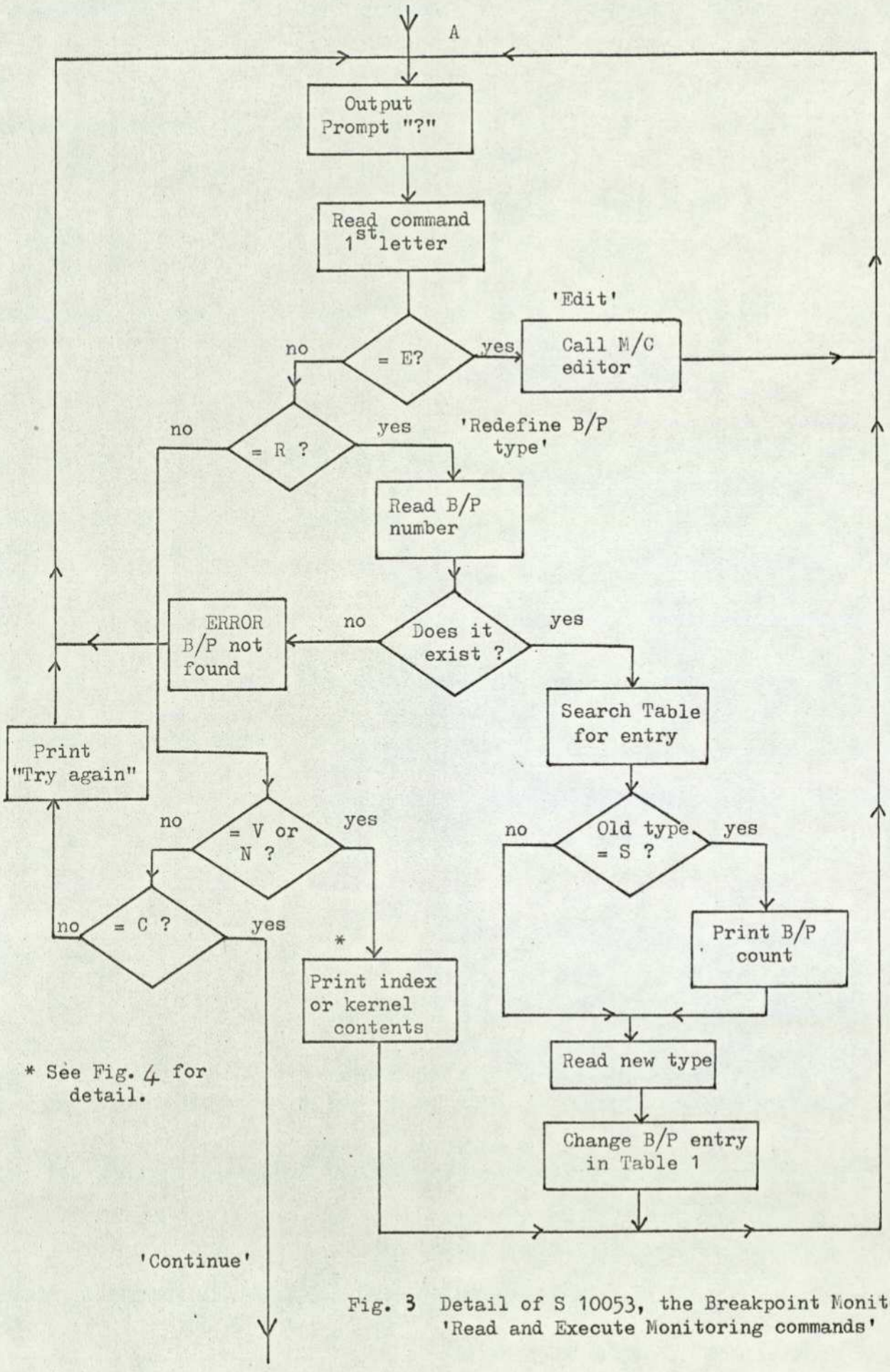


Fig. 3 Detail of S 10053, the Breakpoint Monitor.
'Read and Execute Monitoring commands'

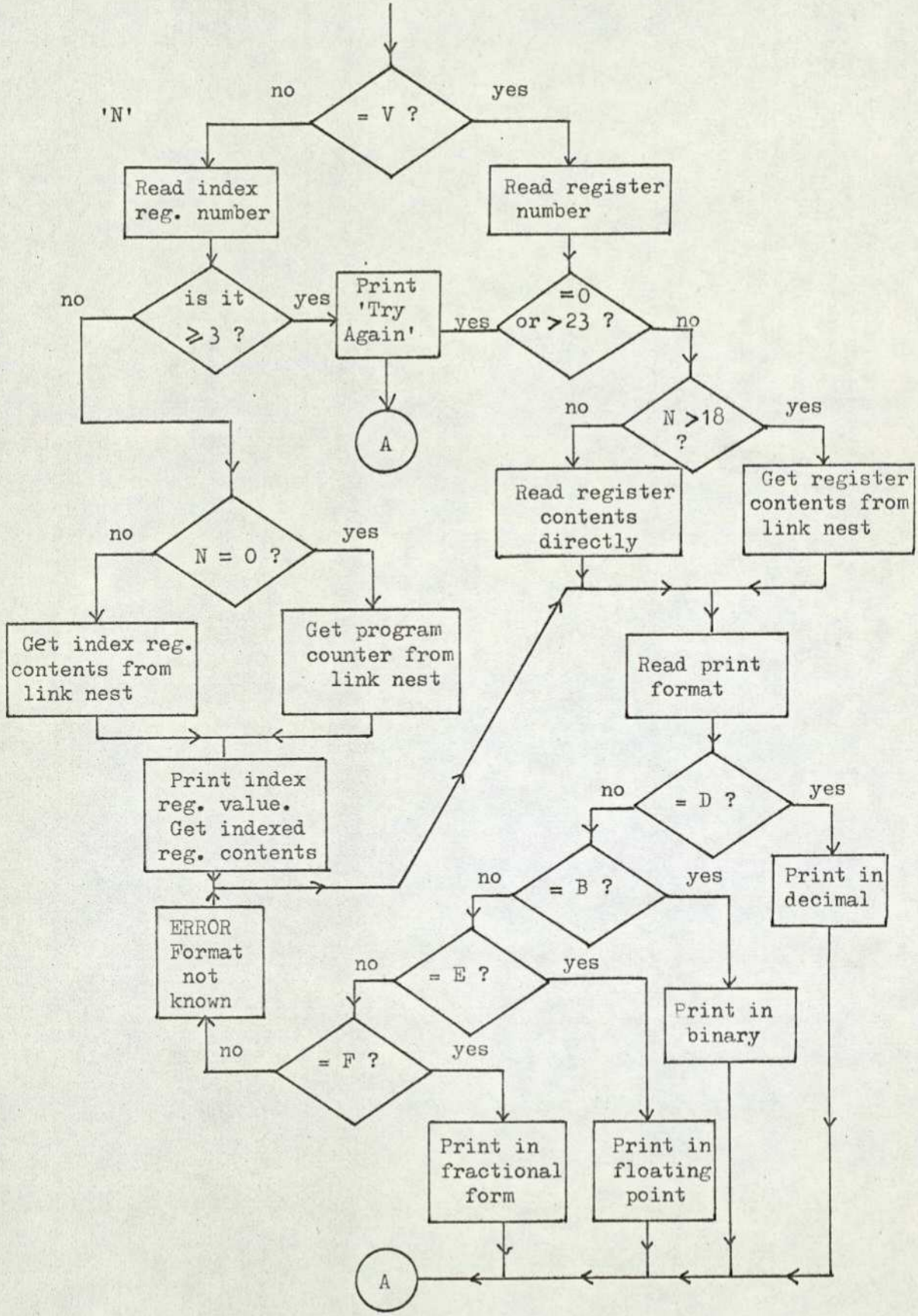


Fig 4 Detail of S 10053, the Breakpoint Monitor.
'Print index and kernel contents'

S10053 Breakpoint Monitor

Function

Allows registers to be examined when called from a breakpoint

Entry

+0 from inserted breakpoint

Entry Conditions

S10051 contains an entry for a breakpoint at the address from which S10053 was called

S10052 contains modified code

Result

The user is invited to examine register contents and/or call the MC editor. The routine fetches modified code from S10052 and exits via this back to the user program.

User Interaction

S-type breakpoint - None

P-type breakpoint - Monitor prints the number and number of occurrences of breakpoint

E-type breakpoint - Monitor prints out as for P-type, then invites user to examine registers. User may type:

Vx	x is 1 to 23 for registers 1 to 23
Ny	y is 0, 1 or 2 for PC or index register 1 and 2
C	for continue, i.e. return to user program
Rw	w is 0 to 31 for re-define breakpoint type
E	for edit, i.e. enter the MC editor.

For Ny the value of the index register is printed. After this (or immediately after typing Vx) the user may type:

D	for decimal format
B	for binary + signed decimal format
E	for exponential, i.e. floating point format
F	for fractional format.

This causes the contents of the register or the location pointed to by the index register, to be printed in the required format.

For Rw, if the breakpoint was an S type, 'B/P COUNT IS' followed by the breakpoint count will be printed. Then 'NEW B/P TYPE?' will be printed, after which the user may enter E, S or P. The message 'RE-DEFINED' is then printed.

Note that printing of '?' means that the monitor is waiting for a breakpoint command.

Error Exits

For Rw 'TYPE NOT RECOGNISED: RE-ENTER NEW TYPE' will be printed if E, S or P is not entered. 'B/P NOT FOUND: ENTER NEW B/P COMMAND' will be printed if the breakpoint to be re-defined does not exist.

If x = 0 or 23 or if y is 2, then 'TRY AGAIN' is printed.

Registers Destroyed

None

Internal Storage

AL 66 holds the modified code used for returns to user program

Link Nest Depth

7

Other Routines Used

S10051, S10052, S5814, S100, S1, TEXT, S555, S32, S15

Storage

380 words

Input-Output

Input - stream 5
Output - stream 4

Stops

Stop 20 18 12 and message 'ERROR - SPURIOUS JUMP AT' followed by jump address is printed if breakpoint monitor is entered from a location where a breakpoint has not to be inserted. This is a disastrous condition requiring a system re-start.

```
N
S10053 BREAKPOINT MONITOR

U

      VN3=N1,N3-1                    ' STORE INDEX REGS. '
      VN3=N2,N3-1
      STK VN3,23,5                   ' STORE KERNELS '
      V20=-32                        ' BLOCK 1 COUNT '
      V22=V[N3+7]
      V22=V22-1                      ' PC ( B/P ADDRESS) '
      V22=V22&VN0
      OCT00177777
      N1=VN0
      As10051


[3]   V21=VN1&VN0
      OCT00177777
      ←1,V22#V21=0                   ' SEARCH '
      ←1,N1=N1+2<0
      ←3,V20=V20+1<0
      V19=4
      TEXT 1P
ERROR-SPURIOUS JUMP AT
      V23=VN0
      +193
      V20=V22
      ←S1
      ←S555,2
[65]  V19=0
      V22=VN0
      +21068                         ' STOP 20 18 12 '
      ←S555,6
      ←65

[1]   V22=VN1&VN0
      OCT07600000
      V22=V22[[16]],R                ' V22= B/P NO. '
      V0=V0,N1+1                     ' NEXT WORD '
      V20=VN1&VN0
      OCT00177777
      VN1=VN1&VN0
      OCT77600000
      V20=V20+1
      V20=V20&VN0
      OCT 00177777
      VN1=VN1+V20
      ←30,VN1[20]=0                  ' TEST MODE '
      V21=31
      ←15
```

```
[20] ←S5814                          ' EDITOR '
     ←4


[31] V19=4
     TEXT
   B/P NO. ?
     V19=5
     V23=0
     ←S100                           ' READ B/P NO. '
     V21=V21&31
     V20=-32
     N1=VN0
     AS10051                         ' TABLE 1 '
[32] V22=VN1[[16]],R
     V22=V22&31
     ←33,V21#V22=0                   ' SEACH FOR B/P '
     ←32,N1=N1+2<0
     ←32,V20=V20+1<0
     V19=4
     TEXT 1P 1E
B/P NOT FOUND : ENTER NEW B/P COMMAND
     ←4


[33] V0=V0,N1+1                      ' SECOND WORD OF BLOCK '
     V20=VN1[[20]],R                 ' MASK OUT MODE BITS '
     V20=V20&3
     ←34,V20#1#0
     V19=4
     TEXT 1P
B/P COUNT IS
     V20=VN1&VN0                     ' MASK OUT COUNT '
     OCT00177777
     V23=VN0
     +193                            ' WRITE COUNTER VALUE '
     ←S1
 3[4] V19=4
     TEXT 1P
NEW B/P TYPE ?
[38] V19=5                           ' READ NEW TYPE '
     ←S555,1
     V21[7]=0
     ←40,V21#VN0=0                   ' "E" '
     +69
     ←41,V21#VN0=0                   ' "S" '
     +83
     ←37,V21#VN0=0                   ' "P" '
     +80
     V19=4
     TEXT 1P
TYPE NOT RECOGNISED : RE-ENTER NEW TYPE
     ←38
```

```
[30  ←16,VN1[21]=0
     V21=31
[16] V19=4
     TEXT
   B/P
     V23=VN0
     +193
     V20=V22                          ' WRITE B/P NO. '
     ←S1
     TEXT
  (
     V23=VN0
     +225
     V20=VN1&VN0
     OCT00177777
     ←S1
     ←S555,2
     V20=VN0
     +169
     ←S555,0
     ←S555,2
    V20=10
     ←S555,0
     ←S555,2


[15] V22=V22[[2]],L                   ' 4* B/P NO. '
     N1=VN0
     AS10052
     ←15,N1=N1+V22<0
     N2=VN0
     AL66
     VN2=VN1,N1+1
     V0=V0,N2+1
     VN2=VN1,N1+1
     V0=V0,N2+1
     VN2=VN1,N1+1
     V0=V0,N2+1
     VN2=VN1
     ←12,V21#31=0


[4]  V19=4
     TEXT
?
     V19=5                            ' READ REG. TYPE '
     ←S555,1
     V21[7]=0
     ←5,V21#VN0=0                     ' JUMP IF "V" '
     +86
     ←7,V21#VN0=0                     ' JUMP IF "N" '
     +78
     ←12,V21#VN0=0                    ' JUMP IF "C" '
     +67
     ←31,V21#VN0=0                    ' JUMP IF "R" '
     +82
     ←20,V21#VN0=0                    ' JUMP IF "E" '
     +69
     ←8
```

```
[7]  V19=5
     V23=0
     ←S100                    ' READ INDEX NO. '
     ←8,V21-3>0               ' N > 3 ? '
     V19=7
     V22=V19-V21              ' LOOK UP IN L-N '
     V20=N3                   ' SAVE LNP '
     ←1,N3=N3+V22<0           ' NON-JUMP '
     V22=VN3                  ' RESTORE LNP '
     N3=V20
     ←10,V21=0
     V20=V22
     V19=4
     V23=VN0
     +193                     ' WRITE INDEX REG. VALUE '
     ←S1
     ←S555,2
     TEXT
  VN
     V23=VN0
     +225
     V20=V21
     ←S1
     V20=VN0
     +160
     ←S555,0
     ←S555,0
     ←S555,2
     N1=V22
     V22=VN1
     ←9


[8]  V19=4
     TEXT  1E
TRY AGAIN
     ←4                       ' ERROR MESSAGE AND RETURN '


[10] V20=V22&VN0              ' MASK OUT PC '
     OCT00177777
     V23=VN0
     +193
     V19=4                    ' WRITE PC VALUE '
     ←S1
     TEXT
    Q4/Q2/Q1 = OCT
     V20=V22[[21]],R
     V23=VN0
     +193
     ←S1
     ←S555,2
     ←11
```

```
[40] V21=0
     ←39
[41] V21=1
     ←39
[37]V21=2
[39] V21=V21[[20]],L
     VN1=VN1&VN0                    ' DELETE OLD TYPE BITS '
     OCT00177777                    ' ENTER NEW TYPE '
     VN1=VN1+V21
     V19=4
     TEXT 1P 1E
RE-DEFINED
     ←4


[5]  V19=5                          ' READ REG. NO. '
     V23=0
     ←S100
     ←8,V21=0                       ' V0 ERROR '
     ←8,V21-24>0                    ' V > 23 '
     ←6,V21-19>0                    ' V > 18 ? '
     N1=V21
     V22=VN1                        ' V22 = RESULT '
     ←9


[6]  V20=N3                         ' SAVE LNP '
     V22=V21-19                     ' LOOK UP IN LINK NEST '
     ←1,N3=N3+V22<0                 ' NON-JUMP '
     V22=VN3
     N3=V20                         ' RESTORE LNP '
     ←9
```

```
[9]   V19=5
      ←S555,1
      V21[7]=0
      V19=4
      V20=V22
      ←14,V21≠VN0=0
      +68
      ←35,V21≠VN0=0
      +70
      ←36,V21≠VN0=0
      +69
      ←17,V21≠VN0=0
      +66
      TEXT 1P
FORMAT NOT RECOGNISED : RE-ENTER NEW FORMAT
      ←9


[14] V23=VN0
      +65
      ←S1
      ←11
[35] V23=VN0
      +87
      ←S1



      ←11
[36] ←S32,1
      ←11
[17] ←S15,1
[11] ←S555,2
      V20=10
      ←S555,0
      ←S555,2
      ←4



[12] LDK VN3,23,5
      N2=VN3
      V0=V0,N3+1
      N1=VN3
      V0=V0,N3+1
      V0=V0,N3+1
[66]  +0
      +0
     +0
      +0
END

S
10053 0
S 10053 B/P MONITOR;
```

S10054 Untouched Machine Code

Storage

32 words

Setup by

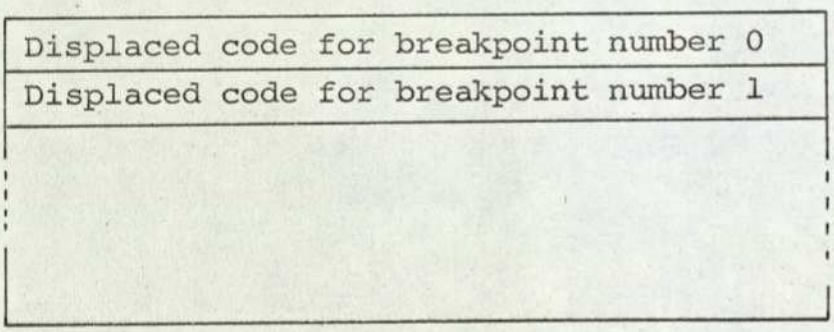
S10050

Used by

S10053

Format

Each entry is the code displaced by the breakpoint.
Entries are positioned according to breakpoint number.



CUPIDS User Guide

The breakpoint routines and a machine code editor were combined into a single debugging utility known as CUPIDS. The following document is a user guide to the program.

CUPIDS

(City University Programmers Interactive Debugging System)

This software package has been developed to facilitate the interactive diagnosis and correction of program errors. It comprises an on-line machine code editor and a break-point routine. The machine code editor allows instructions or blocks of instructions to be located, examined and corrected. It also allows the insertion of break-points into compiled machines code to enable the monitoring of registers at any point during the execution of a program.

A break-point transfers control to a monitoring program (the break-point monitor) when it is encountered, otherwise it does not effect program execution. Kernel registers, index registers and the store locations to which they point may be examined directly at a break-point and the machine code editor may be called for diagnosis/correction of code.

The break-point routine and machine code editor are now described.

Break-point Routine

Three types of break-point causing different actions when encountered are available:

E (Examine)

Prints out break-point number.

Prints out number of times (including the present) the break-point has been encountered.

Waits for user to interrogate registers.

P (Print)

Prints number and counts as above.

Continues with execution of program.

S (Skip)

No print-out.

The count is incremented for later reference.

Up to 32 break-points, identified by their break-point numbers, can exist at any time. P and S type break-points are used to trace paths through programs (the S type is useful in loops where print-out would be prohibitively slow) while the E type can, in addition, be used to monitor registers and perform editing operations.

The break-point monitor provides a number of facilities when at an E type break-point; these are outlined below.

VX<space>F	Print contents of kernel X using format F
NY<space>F	Print contents of index register Y as an unsigned integer followed by VNY using format F*
E (Edit)	Enter the machine code editor
R (Re-define break-point type)	User specifies break-point number and new type. If old type was S the counter value is output
C (Continue)	Normal program execution is resumed.

The following registers are available for interrogation:

V1 to V23 inclusive, NO, N1, N2, VN1, VN2 and the stats. Q1, Q2 and Q4.

When NO is interrogated the value of the program counter is output followed by the state of the three Q stats. as an octal number. The break-point monitor prints a question mark when it is waiting for the user to enter a command.

* In this case the format control character, F, is typed after the value of the index register has been output.

The following formats are available for printing register contents;

D (Decimal)	Contents printed as signed integer of up to 7 digits.
B (Binary)	Contents printed as 24 0's and 1's followed by decimal equivalent.
F (Fraction)	Contents printed as signed fraction with 7 figures after the decimal point.
E (Exponent or floating point)	Contents are printed as mantissa and exponent (SLF form).

Examples

In the following examples the characters underlined are those typed by the user, others are text output from the routines.

```
B/P 3  (4)
B/P 6  (4)
B/P 2  (6)
?   V16  D  +6
?   V12  D  +492
?   V12  B  000000000000000111101100  +  492
?   V12  F  +0.0000586
?   V12  E  +0.509317  E  -  10
?   E
EDIT CMD.
MB
START ADDRESS 512 LENGTH 2
512  0110  19  0  4
512  0110  20 12  0
EDIT CMD.
END
?   R  B/P NO. ? 7
NEW B/P TYPE ? E
RE-DEFINED
?   C
```

In this example break-points 3 and 6 are P type break-points each of which has been encountered 3 times previously, while 2 is an E type with a count of 6. The user obtains the decimal value of kernel 16 followed by the value of kernel 12 interpreted as decimal, binary, fraction and floating point.

He then calls the machine code editor and monitors a block of core. Break-point 7 is then re-defined as an E type.

```
B/P 8 (12)
? NO  658  Q4/Q2/Q1  =  OCT  2
? N1  1987 VN1  D  +68
? N2  2030 VN2  B  000001010001010100001010  +  333066
? N3  TRY AGAIN
? C
```

In this example the program counter at break-point 8 is 568 and Q2 is set while Q1 and Q4 are clear. N1, VN1, N2 and VN2 are examined but N3 cannot be obtained so the user is invited to enter another command.

Notes:

1. The following instructions must not be break-pointed:
   link restore (return)
   modify index register involving a modification constant of 28 in the C field.
2. The break-point occurs before execution of the instruction to which it is attached.

MACHINE CODE ('FIDS') EDITOR

The following commands are recognised by the editor.

Command	Meaning
M	Monitor a single location
MB	Monitor a block
C	Correct a single location
CB	Correct a block
L	Locate an instruction
MSK	Set locate instruction mask
LMK	Locate with instruction mask
MR	Monitor repeat
MBR	Monitor block repeat
CR	Correct repeat
CBR	Correct block repeat
DTF	Convert decimal to FIDS A B C
FTD	Convert FIDS A B C to decimal
IBP	Insert breakpoint
DBP	Delete breakpoint
ZBPC	Zero breakpoint count
IBPR	Insert breakpoint repeat

These commands and their use is explained in further detail below. All commands are terminated by a newline. Entry to the editor dictionary is denoted by the text <Edit CMD> Output on stream 4. Note that the output from monitor or monitor block is output on stream 6. In the following examples the user input has been underlined.

M Monitor a single location

```
EDIT CMD.
M
 ADDRESS  512  0110  19 0 4
EDIT CMD.
```

MB Monitor a block of code

```
EDIT CMD
MB
  START ADDRESS  512  LENGTH  4
512  0110 19 0 4
513  0110 20 12 0
514  0110 23 0 0
515  7000 10 0 0   10240      (Decimal of jump address)
EDIT CMD.
```

C Correct a single location

```
EDIT CMD.
C
  ADDRESS  512  TO  0110 19 0 6
EDIT CMD.
```

CB Correct a block of code

```
EDIT CMD.
CB
  START ADDRESS  516  LENGTH  2
516  0203 0 0 28
517  0000 5 8 23
EDIT CMD.
```

L Locate a machine code instruction

```
EDIT CMD.
L
INSTRUCTION  0116 21 12 2 BETWEEN WORD 512 AND WORD 2000

 LOCATED  AT  560
     AND  AT  700
     AND  AT  1861
EDIT CMD.
```

MSK Set up octal mask pattern for locate with mask

```
EDIT CMD.
MSK
 OCT01000037
EDIT CMD.
```

LMK Locate a machine code instruction with mask

This command masks the input instruction and each location of core scanned before making the comparison test.

```
EDIT CMD.
LMK
INSTRUCTION  0010 2 0 28 BETWEEN WORD 769 AND WORD 800

LOCATED  AT  782
    AND  AT  790
EDIT CMD.
```

NOTE: The 2 in the A field will be ignored in the comparison test.

Certain of the commands recognised by the FIDS editor do not require an address parameter (such commands end in 'R') but use the last address parameter entered.

MR Monitor repeat

```
EDIT CMD.
MR
 0203 0 0 28
EDIT CMD.
```

In this example the last address entered was 516 in the earlier CB example.

CR Correct repeat

```
EDIT CMD.
CR
  TO  0103 0 0 28
EDIT CMD.
```

MBR Monitor block repeat

```
EDIT CMD.
MBR
  LENGTH  3
516  0103  0  0 28
517  0000  5  8 23   5399
518  7212 21 21  8
EDIT CMD.
```

CBR Correct block repeat

This functions in a similar way to CB but with the repeat address used as the start address of the block to be corrected. Only the length of the block has to be entered.

DTF Convert decimal to FIDS A B C

```
EDIT CMD.
DTF
514   0000 0 16 2
EDIT CMD.
```

FTD Converts FIDS A B C to decimal

Converse of DTF.

IBP Insert breakpoint. (See also section on breakpoints)

This allows a breakpoint to be inserted and designated.

```
EDIT CMD.
IBP
 ADDRESS  657  B/P NO. = 8  B/P TYPE ? E
INSERTED
EDIT CMD.
```

DBP Delete breakpoint

Deletes a previously inserted breakpoint.

```
EDIT CMD.
DBP
  B/P NO. = 8
DELETED
EDIT CMD.
```

ZBPC Zero break point count

See details in section on breakpoint counts.

IBPR Insert breakpoint with repeat address

This is used to insert a breakpoint just after a location has been monitored.

```
EDIT CMD.
M
  ADDRESS  514  0110 23 0 0
EDIT CMD.
IBPR
  B/P NO. = 2  B/P TYPE ? E
INSERTED
EDIT CMD.
```

It is possible to monitor kernels 0 to 23 location 24 when monitored gives the state of the SCM software handswitches. Locations 27 & 28 can not be monitored. Locations 25, 26, 29, 30 and 31 may all be monitored.

NOTE. No locations between 0 and 31 inclusive may be monitored by the editor when it is called from a break-point.

Storage and Subroutine Requirements

CUPIDS consists of 9 subroutines whose lengths are as shown

S1020	44
S1021	94
S5814	586
S5815	50
S10050	346
S10051	64
S10052	128
S10053	380
S10054	32
Total	1724 words

It also calls the following standard subroutines

S1
S12
S15
S32
S100
S555

CUPIDS uses the SC Interface for all input/output operations.

CUPIDS is entered via a call to S 5814.

Richard Young
4th May 1976

APPENDIX IV

The DIXPAC Assembler

Appendix IV - DIXPAC

INTRODUCTION

The DIXPAC assembler was written by the author to enable the DIXPAC algebraic assembly language to be prepared on the host (FM1600B) computer for transfer to the RSP. The DIXPAC language was defined in such a way as closely to resemble the algebraic language FIXPAC, used on the FM1600B. This was to avoid confusion for people working with the host and slave processors by the adoption of similar conventions in the description of operations. The assembler was written in FIXPAC. Because both languages closely reflect the machine structures on which they were designed to operate, and because these structures are quite different, the languages differ widely. However, in algebraic languages, the meaning is conveyed by the symbols used to represent the operand types and sub-operations, which go to make the machine code instructions. So, by keeping these descriptions the same, two different languages can each be made intelligible to users of the other.

THE DIXPAC LANGUAGE

Because of the need to maintain a symbology established in FIXPAC, the author was left with few choices when designing the DIXPAC language. In particular, the following conventions were inherited from the FM1600B language:

Labels - Numerical labels (no alphabetic characters) at the start of a line in square brackets, e.g. [16].

Direct Addressed Operands - e.g. V34 means the contents of location 34.

Indirect Addressed Operands - e.g. VN2 is the contents of the location whose address is in location 2.

Shift Length and Bit References - in square brackets after the operand reference, e.g. V6[2] is bit 2 of location 6.

Jumps and Subroutine Calls - e.g. →6 means jump to label 6. If conditional, then condition separated by a comma, e.g. →6, V2[4] = 0 means jump to label 6 if bit 4 of location 2 = 0.

Constants - are either entered directly, as a signed number, or placed on the line following and indicated by 'VNO', e.g.

```
A = VNO
  + 318
```

means load accumulator with the value on the next line (+ 318).

Comments - are any character enclosed by primes.

The only major addition for DIXPAC is the use of 'A' to indicate the accumulator. This gives rise to the language as defined in the appendix to the paper 'An Unorthodox Approach to Microprocessor Language'. The paper includes an example.

Other examples are:

V12[3]≠0	Set bit 3 of location 12
A = VN23, N + 1	Increment location 23 and use the result as an address. Load the accumulator with the contents of this address.
A[1], L	Shift the accumulator one place left.
→6, V13 = V + 1≠0	Increment location 13 and if the result is not zero, jump to label 6.

THE ASSEMBLER

Introduction

The assembler is simple and is based on the sequential search of look-up tables. Extensive error reporting facilities are incorporated. The assembler can be run in 'batch mode', i.e. a source tape input and an object + error messages output, or in interactive mode. In this mode, the user sits at a teletype and enters a line at a time to the assembler. If an error is found, then a message is output and the user is invited to re-type the line. This continues until a correct line is input. This technique is particularly useful for teaching students the language since they can learn by a 'trial and error' method.

Assembler Structure

Fig. | shows an overall view of both passes of the assembler.

The characters are read from a normal input stream by S10006 until a newline is detected. Nulls, deletes, primes and anything enclosed by primes (comments) are removed. Each character is checked against a legality mask (S10011) which defines whether each of 128 characters is legal in DIXPAC or not. The resulting line is packed one character to a word in S10001 with a newline and a delete terminating it. S10005 searches the line for labels (i.e. starting with "[") and extracts the label number using S10007. The label number and machine code address (in V13) are entered into the label table, S10012. The label is then edited from the line in S10001.

S10004 searches the line for machine code instructions, i.e. those beginning "MC". If found, it reads the following hexadecimal characters into the machine code word. A following non-hex character will cause an error.

If no "MC" was found, then S10008 takes the line and removes all numeric characters from it, packing the remaining alphabetic characters three to a word in the data area S10003. S10015 is a preset look-up table consisting of 109 entries, each of four words length. Each four word block represents up to twelve characters, packed in the same way as S10003. Each of the 109 entries represents one of the possible instruction types in DIXPAC. S10002 compares S10003 with each of the entries in turn. When a word is found, a corresponding one word value is extracted from S10006, the machine code results table. The value so obtained consists of two parts: the bottom sixteen bits are a skeleton of the machine code word onto which must be added all the numerical fields, while the top eight bits are divided into a four bit flag field (indicating certain conditions) and a four bit classify field which tells the assembler which numerical fields are required to be added to the skeleton machine code word. S10020, S10021, S10022 and S10023 are called, if their appropriate classify bits are set, to extract respectively "V" and "VN" references, integer fields, bit and shift references, and jump or subroutine calls. Each of these use S10007, via its driver S10013, to extract numerical fields from S10001.

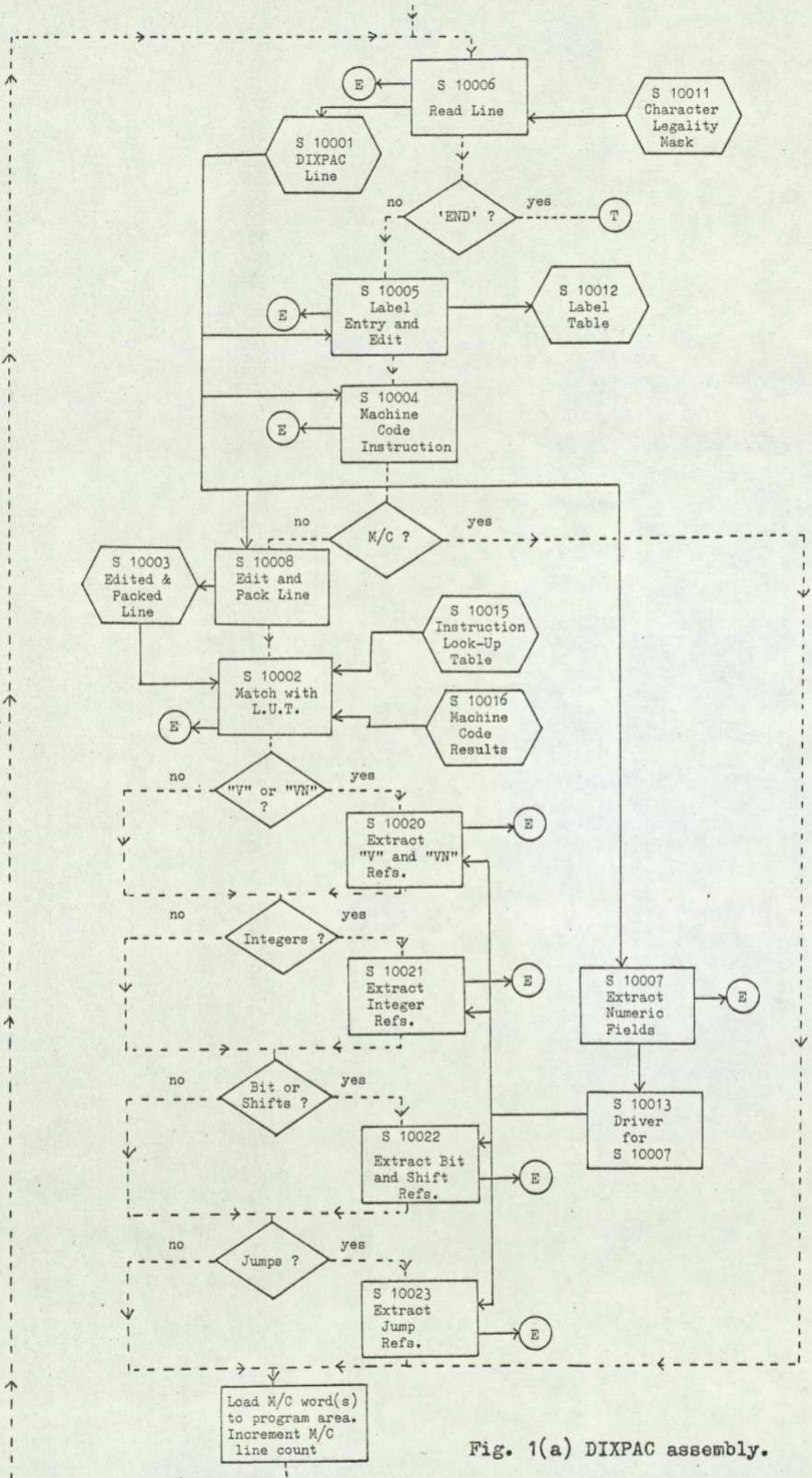


Fig. 1(a) DIXPAC assembly.

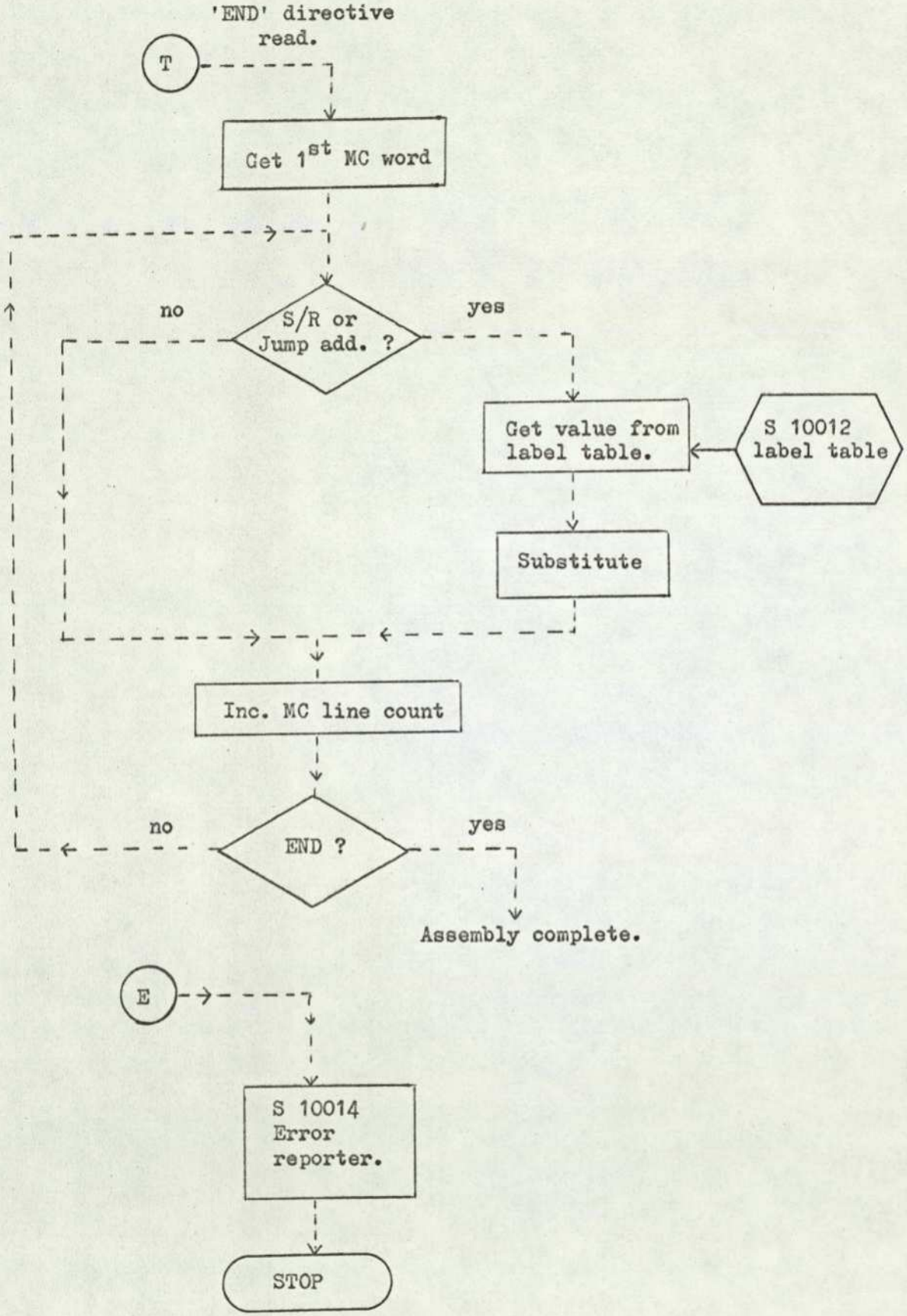


Fig. 1(b) DIXPAC assembly.

When jumps or subroutines are added in by S10023, it marks the word containing the subroutine or jump reference by setting bit 22 or 21 of that word. The value it puts in it is the value of the label. Thus, if a jump to label 6 is required, the value 6 will be entered and bit 21 will be set. During the second pass, the assembler can identify such references and substitute label addresses for label values.

At this point the machine code value of the DIXPAC line is complete as a one, two or three word instruction in V17, V16 and V15.

The machine code words are loaded to the program area and the machine code line count is incremented by an appropriate amount. The program then returns to read another line. This continues until an 'END' directive is read. Control then transfers to the second pass - S10009. This routine substitutes label addresses for label values using the label table (S10012). Errors are reported for labels not found. The marker bits 22 and 21 are cleared, but if either was set, then 23 is set. This serves to show that the substitution has taken place but subroutine calls and jumps are still marked. This allows the whole program to be relocated at a later time.

Some of the features included in DIXPAC are not implemented in the present assembly system. In particular, it is not possible to link previously compiled subroutines into a program. The 'directive read' subroutine is not properly implemented and only a 'Z' for END is recognised. The system is normally used in an interactive mode with the user entering each line and getting an error message immediately.

FUTURE DEVELOPMENTS

The following facilities are needed to make the assembler into a properly workable system:

1. A better set of directives to enable individual subroutines to be compiled, and to force a new start address for a piece of code.

2. A loader with a relocating facility so that the assembled code could be shifted anywhere in store.

3. A library system to enable compiled segments to be stored and rettieved.

SUBROUTINE DETAILS

There follows detailed descriptions of the DIXPAC subroutines. For program listings refer to Electrical and Electronic Engineering Dept. Report 'The DIXPAC Assembler' 1976.

DIXPAC Subroutines

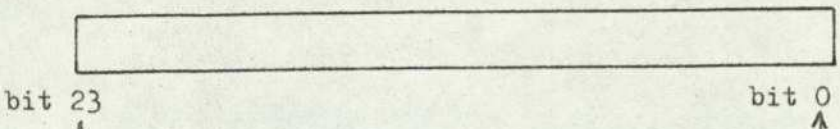
	Number	Title	Type
	10001	DIXPAC line	D
	10002	Match with look-up tab.	E
	10003	Edited and packed line	D
	10004	Machine code instructions	E
	10005	Edit and record labels	E
	10006	Read DIXPAC line	E
	10007	Extract numeric fields	E
	10008	Edit and pack line	E
	10009		
	10010		
	10011	Character legality mask	P
	10012	Label table	D
	10013	Driver for S 10007	E
*	10014	Error reporter	E
	10015	Instruction look-up table	P
	10016	Machine code results table	P
	10017		
	10018		
	10019		
	10020	Extract "V" and "VN" references	E
	10021	Extract integers and stop no.	E
	10022	Extract bit and shift refs.	E
	10023	Extract jumps and subroutines	E

Notes:

1. Subroutines are:
   - E - executable code
   - D - data area (read/write)
   - P - preset data area

2. * S 10014 is not 'called' but jumped to. After putting out the error message, current line is abandoned.

DIXPAC Flag Word V 18



0	Illegal character	(10006)
1		
2	> 5 digits read	(10007)
3	No field found	(10007)
4	M/C instruction *	(10004)
5	Non-hex character	(10004)
6	No look-up match	(10002)
7	Label ≥ 512	(10005)
8	"VNO" found *	(10020)
9	"VN" ref. ≥ 256	(10020)
10	"V" ref > 2K but < 32K *	(10020)
11	"VO" ref.	(10020)
12	"V" ref. ≥ 32K	(10020)
13	Integer out of range	(10021)
14	Stop or ext. func ≥ 512	(10021)
15	Bit ref. too large	(10022)
16		
17	Jump or sub. ≥ 512	(10023)
18	"VNO" ref. not found	(10018)
19		
20	"VNO" ref, found *	(10021,10023)
21		
22		
23	"VNO" ref. expected *	(10018)

Notes:

1. The numbers in parenthesis represent the subroutine which sets the flag.
2. The condition is met when the bit is set.
3. * indicates a non-error condition.

S 10001 DIXPAC LINE

DATA AREA

SET UP BY: S 10006

USED BY: S 10004, S 10005, S 10007, S 10008
S 10020, S 10021, S 10022, S 10023

FORMAT

Line stored one character to a word, without parity, in the 7 LSB's .

1st word	Newline	
2nd word	1st char.	DIXPAC Line
3rd word	2nd char.	
⋮	⋮	
Nth word	Last char.	
N+1th word	Newline	
N+2th word	Delete (127)	

S 10002 MATCH WITH LOOK-UP TABLE.

Current mark is 4

FUNCTION:

Compares edited line in S 10003 with a look-up table in S10015. If match is found then used to extract MC result from S 10016. If not - flagged.

ENTRY:

S 10003 is edited and packed line

S 10015 is look-up table.

S 10016 is MC results table

EXIT

V 17 contains MC word if successful.

FLAGS

V 18 bit 6 set to indicate no match i.e. bottom of table reached.

REGISTERS DESTROYED

V 17

LINK NEST DEPTH

4

No.OF WORDS

40

OTHER S/R CALLED

S 10003, S 10015, S 10016

NOTES:

1. Size of table is 4* no. of entries.
2. Mark 4 is 436 long (109) entries.

S 10003 EDITED AND PACKED LINE

DATA AREA

SET UP BY: S 10008

USED BY: S 10002

FUNCTION

The area consists of four words each with three non-parity characters packed in. Unused char. positions are zero.

S 10004 DECODE MACHINE CODE INSTRUCTIONS

Current mark is 1

FUNCTION

Searches DIXPAC line in S 10001 to see if it contains "MC" i.e. is a machine code instruction. The resulting word is placed in V 15

ENTRY

S 10001 contains DIXPAC line.

EXIT

V 15 contains machine code word

FLAGS

V 18 bit 4 set if MC found

V 18 bit 5 set if a non-hex. character found after MC

REGISTERS DESTROYED

V 15

LINK NEST DEPTH

3

No. OF WORDS

41

OTHER S/R CALLED

S 10001

NOTES

1. No check on the number of characters read
2. MC instruction should contain four hex. chars.

S 10005 LABEL TABLE AND LABEL EDIT

Current mark is 2

FUNCTION

Searches DIXPAC line in S 10001 for a label. If found it edits it out and enters the label number and MC line count in the label table (S 10012). If too large a flag set.

ENTRY

S 10001 is DIXPAC line.
S 10012 is label table
V 13 is machine code line count.

EXIT

S 10001 is line with no labels
S 10012 is label table + new entry

FLAGS

V 18 bit 2 set if label 5 digits.
V 18 bit 3 set if no label found.
V 18 bit 7 set if label 512.

REGISTERS DESTROYED

None

LINK NEST DEPTH

8

No. OF WORDS

58

OTHER S/R CALLED

S 10001, S 10007, S 10012.

NOTES

1. A note of the next free psition in the label table is kept internally.

S 10006 LOAD AND CHECK DIXPAC LINE

Current mark is 6

FUNCTION:

Reads DIXPAC line from stream 5, removes parity and stores characters one to a word in S 10001. Various chars. are edited out and the rest are checked for legality against a mask in S 10011

ENTRY:

N1 points to where line is to be stored
e.g. S 10001

EXIT:

Line stored as indicated.

FLAGS:

V18 bit 0 set for illegal char. read.

REGISTERS DESTROYED:

N1

LINK NEST DEPTH:

7

No. OF WORDS:

50

OTHER S/R USED:

S 555,1 , S 10011

NOTES:

1. The line is stored with a newline at the start and reading is terminated by newline which is stored at the end with a delete after it.
2. The edited chars. are ; nulls, deletes, primes and anything enclosed by primes.
3. If an illegal char. is read the delete terminator is placed after the last legal char.
4. The data area is not cleared.

S 10007 EXTRACT INTEGER FIELDS

Current mark is 3

FUNCTION:

Searches a line pointed to by N1 for an integer field preceeded by one or two characters. If found it outputs the value of the field, the number of digits read and the terminating char.
If not found or>5 digits read, then flags set.

ENTRY:

V22 contains 1st match char. or space (32) if single match required.
V21 contains second (only) match char.
N1 points to start of line e.g. S 10001

EXIT:

V22 is number of digits read
V21 is terminating char.
V20 is integer value of field

FLAGS:

V18 bit 2 set if >5 digits read
V18 bit 3 set if no match found

REGISTERS DESTROYED:

V20, V21, V22, N1

LINK NEST DEPTH:

1

No. OF WORDS:

28

OTHER S/R CALLED:

none

S 10008 EDIT AND PACK DIXPAC LINE

Current mark is 7

FUNCTION:

Accepts DIXPAC line and edits out all digits. The remaining characters are packed three to a word in a data area pointed to by N2. Up to four words are used - unused ones are zeroed.

ENTRY:

N1 points to line e.g. S 10001

N2 points to data area for result e.g.S 10003

EXIT:

Data written to N2, N2+1 etc.

FLAGS:

None

REGISTERS DESTROYED:

N1 , N2

LINK NEST DEPTH:

3

No. OF WORDS

26

OTHER S/R CALLED:

None

NOTES:

1. All integer values of data area words are +ve since bit 23 is never set due to use of non-parity characters.

S 10011 CHARACTER LEGALITY MASK.

PRESET DATA AREA

USED BY: S 10006

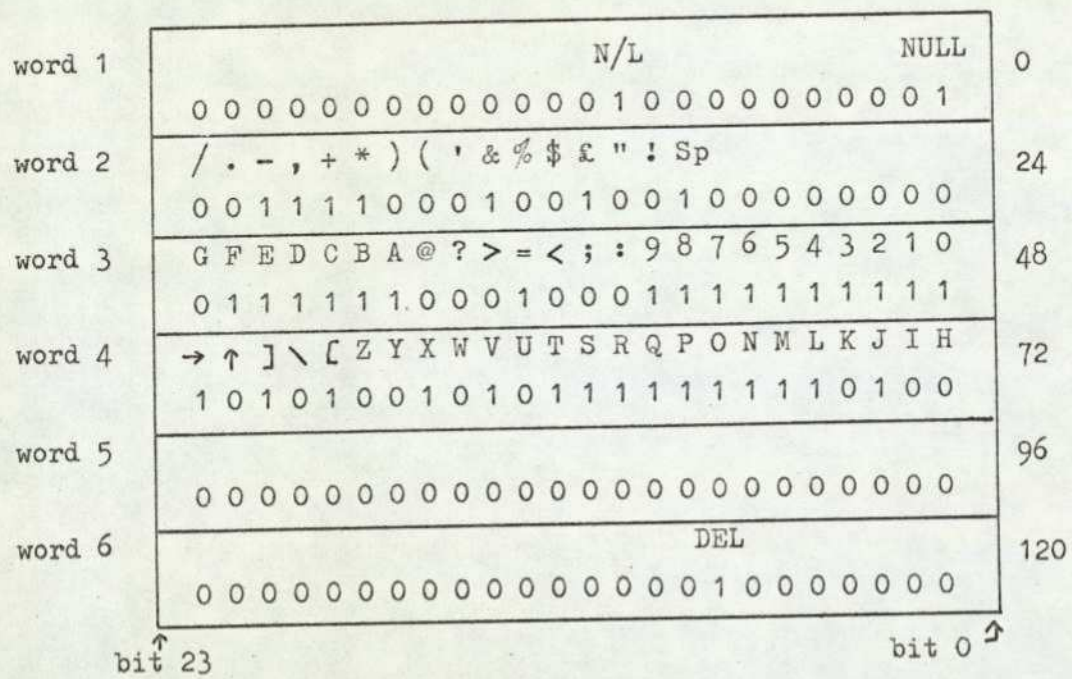
FUNCTION

Provides a mask of 0's and 1's to determine if characters are illegal or legal in DIXPAC.

No.OF WORDS 7

FORMAT

Six 24-bit words hold 144 markers of which 128, addressed by the non-parity value of the characters, are significant.



S 10012 LABEL TABLE

DATA AREA

SET UP BY : S 10005

USED BY : S 10009

FORMAT

The 9 msb's are the label number and the 15 lsb's are the machine code line count. One word is used for each label entry.

SIZE 512 words

NOTES:

1. Labels are entered in the order in which they are encountered in the program.
2. The next free position in S 10012 is held in S 10005.

S 10013 DRIVER FOR S 10007 SEARCH

Current mark is 4

FUNCTION

Loads V 21 and V 22 from a single word in V 20
Sets N 1 to point to S 10001 (DIXPAC LINE)
Calls S 10007 (search for numeric field).
Checks flags and calls error reporter.

ENTRY

V 20 contains two non-parity characters to be loaded into V 21 and V 22.

V 21	V 22
2nd char.	1st char. (space)

V 22 is space (32) if single match required

EXIT

No errors;
As for S 10007
Errors:
S 10014 entered if V 18 bit 3 set

FLAGS

As for S 10007

REGISTERS DESTROYED

N1, V20, V21, V22.

LINK NEST DEPTH

1

No. OF WORDS

12

OTHER S/R CALLED

S 10001, S 10007, S 10014

NOTES

S 10014 ERROR REPORTER

Current mark 7

FUNCTION:

Examines flag word in V18 and outputs to stream 4 an error message depending on flags. Also puts out MC line number.

ENTRY:

V18 contains flag word

EXIT:

No exit - stops.

FLAGS:

Clears all flags except V18 bit 8

REGISTERS DESTROYED:

V19, V20, V22

LINK NEST DEPTH:

0 - no link address stored.

No. OF WORDS:

114

OTHER S/R CALLED:

S 1, S 555,6 TEXT

STOPS:

Exit is via STOP 00 00 00

S 10015 LOOK - UP TABLE

PRESET DATA AREA

USED BY: S 10002

FUNCTION

Provides blocks of four words, each representing three characters, for matching against the edited and packed DIXPAC line.

NO OF WORDS 436 (109 entries)

NOTES:

1. The text in primes is the un-edited instruction from which the entry is produced. All numbers are meaningless as only alphabetic characters and punctuation are matched.

S 10016 MACHINE CODE RESULTS TABLE.

PRESET DATA AREA

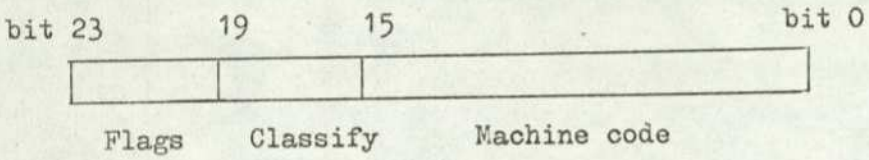
USED BY: S 10002

FUNCTION

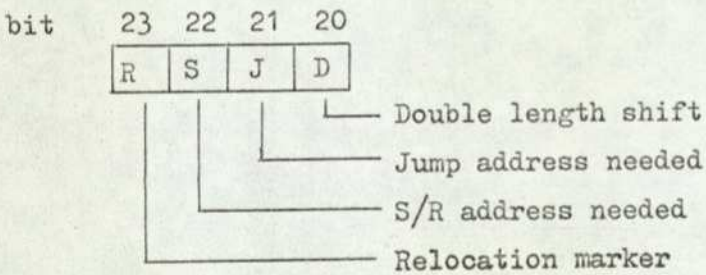
Provides skeleton machine code words with all numeric fields set to zero. Also provides classify bits to indicate which numeric fields are needed and some flags.

No. OF WORDS 109

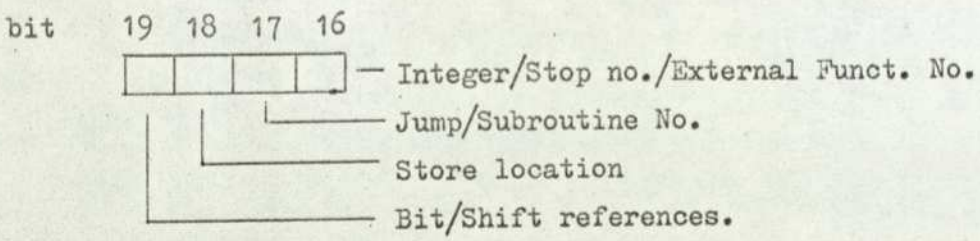
FORMAT



Flags.



Classify Bits.



S 10020 CHECK "V" AND "VN" REFS.

Current mark is 4

FUNCTION

Checks size of no. fields following V and VN refs. If legal adds into MC word or forms second word of two word instruction. Flags illegal refs (0 & ⩾32K) VN0 refs. and V refs⩾2K.

ENTRY:

V17 contains MC word.
S 10001 is DIXPAC line.

EXIT:

V17 is updated MC word.
V16 is next word if two word instruction.

FLAGS:

V18 bit 8 set for VN0 ref
V18 bit 9 set for VN ref. ⩾ 256
V18 bit 10 set for V ref.⩾2K (but legal).
V18 bit 11 set for V0 ref.
V18 bit 12 set for V ref.⩾32K.

REGISTERS DESTROYED

N1, V16, V17.

LINK NEST DEPTH

7

No. OF WORDS:

58

OTHER S/R CALLED

S 10007, S 10001, S 10013.

S 10021 CHECK INTEGER FIELDS.

Current mark is 3

FUNCTION:

Searches DIXPAC line for stops, external functions and +ve and -ve integers. Adds value into V16 or V17 if correct else sets flags

ENTRY:

S 10001 is DIXPAC line.
V17 contains MC word and flags.

EXIT:

Legal stops and ext. function no. s are added into V17. Legal integers are placed in V16.

FLAGS:

V18 bit 14 set if ext.fn. no. or stops 512
V18 bit 13 set for integer ≥ 32768 or < -32768

REGISTERS DESTROYED:

N1

LINK NEST DEPTH:

5

No. OF WORDS:

74

OTHER S/R CALLED:

S 10001, S 10007, S10014

NOTES:

If >5 digits read, V18 bit 2 set.

S 10022 CHECK BIT AND SHIFT REFS.

Current mark is 1

FUNCTION:

Searches DIXPAC line for bit and shift refs and checks size. Flags error if too large or not found. If legal adds into MC result in V17.

ENTRY:

S 10001 is DIXPAC line.
V17 contains MC word.

EXIT:

V17 has legal bit and shift refs. added in.
S 10014 error reporter called for errors.

FLAGS:

V18 bit 2 set for no match.
V18 bit 3 set if>5 digits read.
V18 bit 15 set for ref. too large.

REGISTERS DESTROYED:

V17

LINK NEST DEPTH:

5

No. OF WORDS:

32

OTHER S/R CALLED:

S 10013, S 10014

NOTES:

1. The max. bit and shift refs allowed are:
16 for S/L ops. on Acc. or store location
7 for S/L ops. on Q register.
32 for D/L ops. on Acc. or store location.
23 for D/L ops. on Q register.

S 10023 CHECK JUMP AND S/R REFS.

Current mark is 3

FUNCTION:

Searches DIXPAC line in S 10001 for jump and subroutine refs. including "AL" and "AS" refs. Flags out of range refs. also falgs AL and AS. Places value in V17 or V16 and sets flags to indicate if jump or S/R ref.

ENTRY:

S 10001 is DIXPAC line.
V17 is MC word.

EXIT:

If legal:
V16 contains label or S/R no. for a jump.
V17 contains label or S/R no. for AL/AS refs.

FLAGS:

For a jump (AL ref.) V16(V17) bit 21 set.
For an AL/AS ref V18 bit 20 is set.
For a S/R call/AS ref. V16/V17 bit 22 is set.
For any illegal ref. ($\geqslant 512$) V18 bit 17 is set.
Error reporter called for illegal refs.

REGISTERS DESTROYED:

N1, V16, V17

LINK NEST DEPTH:

5

No. OF WORDS:

62

OTHER S/R CALLED:

S 10014, S 10001, S 10013

APPENDIX V

The DIXPAC De-Assembler

Appendix V - The DIXPAC De-Assembler

INTRODUCTION

The DIXPAC de-assembler was written by the author to enable blocks of F100 machine code to be displayed as DIXPAC source language. This is particularly useful during debugging operations.

The de-assembler consists of six subroutines which can be incorporated within an existing machine code editor (e.g. the RSP operating system), or run with a driver program as a complete system. The input format is system dependent: the one shown takes integer values from paper tape.

DE-ASSEMBLER OPERATION

S10045 reads the machine code into V17, V16 and V15. There are two entry points: the 'initial read' fills all three locations from the input stream while the 'update read' puts a single new value in V15, V17 takes the old V16 value and V16 takes the old V15 value. Thus the de-assembler always has the current machine word and the two following words available to it. S10043 decodes V17 and decides whether it represents a one, two or three word instruction. The number of words is placed in V22.

Instructions are de-assembled by S10042 (non F = 0) and S10044 (F = 0). These routines use S10040 to print operands and S10041 to print indexing terms for indirect operands. If S10040 finds a 'VNO' reference, it flags it and the driver program prints the VNO value.

Some F = 0 machine codes can be represented by two alternative DIXPAC statements, depending on the state of an internal latch. Since S10044 cannot predict the intent of the programmer, both alternative forms are output - the least likely form in parenthesis.

OPERATING DIFFICULTIES

There are three possible difficulties in the use of a de-assembler:

1. Unless the label table (and symbol table if named variables are used) is preserved from assembly, then absolute addresses must be displayed.

2. The de-assembler will try and decode data areas embedded within code segments.

3. If the de-assembler is started on the second or third word of a multi word instruction, then all the subsequent decoding may be nonsense.

The second and third points are unavoidable in de-assembly systems but are less trouble in practice than in anticipation, since one generally knows the whereabouts of data areas. The first point is dealt with in two ways: since DIXPAC does not have named variables, this difficulty does not arise; while the problem of labels is avoided by printing their absolute addresses and outputting the addresses of all instructions before printing the de-assembled form.

SUBROUTINE DETAILS AND AN OUTPUT EXAMPLE

There follows program listings of the de-assembler, and an example of the de-assembler output from an input tape of a manufacturer-supplied program. It will be seen that with some interpretation the original text is recoverable.

```
START ADDRESS = 1

32768
1536
49216
24640
38
20512
24584
32811
18496
32832
49180
97
24617
24618
24619
24627
8224
135
17
104
18717
8224
22557
8224
28745
20
61515
255
72
100
200
8228
49180
20482
14336
34858
135
36
34857
12288
32764
32765
32766
32769
496
18475
384
47
34858
14336
32767
32797
18483
32798
18475
32799
18474
61498
0
0
0
0
0
END
```

Input to de-assembler - decimal equiv. of a loader program.

Untouched de-assembler output. (Some data areas have been interpreted as program).

```
     1    A=VN0
+1536
     3    A=V64&A
     4    V64=V64-A
     5    A[[6]],R              (A,A[[6]],RD)
     6    V32=V32+A
     7    V8=V8-A
     8    A=V43
     9    VN64=A
    10    A=V64
    11    A=V28&A
    12    A[[1]],L              (A,A[[1]],LD)
    13    V41=V41-A
    14    V42=V42-A
    15    V43=V43-A
    16    V51=V51-A
    17    ←32,L
    18    ←17,A[7]=0
    20    A[[8]],L              (A,A[[8]],LD)
    21    VN29=A,N+1
    22    ←32,L
    23    VN29=VN29+A
    24    ←32,L
    25    ←20,V73=V+1#0
    27    ←75
    28    A[15]=0
    29    A[8],L              (A,A[8],LD)
    30    A[[4]],L              (A,A[[4]],LD)
    31    A[8]#0
    32    ←36,L
    33    A=V28&A
    34    V2=V2+A
    35    ←LNQ
    36    A=VN42
    37    ←36,A[7]=0
    39    A=VN41
    40    ←L
    41    ←32765,VN252=V+1#0,N-1
    43    ←32769,VN254=V+1#0,N-1
    45    Q[0]=0
    46    VN43=A
    47    ←47,Q[0]=0
    49    A=VN42
    50    ←LNQ
    51    ←32797,VN255=V+1#0,N-1
    53    VN51=A
    54    A=V30
    55    VN43=A
    56    A=V31
    57    VN42=A
    58    ←58
    59    A[0],R              (A,A[0],RD)
    60    A[0],R              (A,A[0],RD)
    61    A[0],R              (A,A[0],RD)
    62    A[0],R              (A,A[0],RD)
    63    A[0],R              (A,A[0],RD)
```

```
 1    A=VN0
      +1536
 3    A=V64&A
 4    V64=V64-A
 5    A[[6]],R
 6    V32=V32+A
 7    V8=V8-A
 8    A=V43
 9    VN64=A
10    A=V64
11    A=V28&A
12    A[[1]],L
13    V41=V41-A
14    V42=V42-A
15    V43=V43-A
16    V51=V51-A
17    -32,L
18    -17,A[7]=0
20    A[[8]],L
21    VN29=A,N+1
22    -32,L
23    VN29=VN29+A
24    -32,L
25    -20,V73=V+1#0
27    -75

28    +255
29    +72
30    +100
31    +200

32    -36,L
33    A=V28&A
34    V2=V2+A
35    -LNQ

36    A=VN42
37    -36,A[7]=0
39    A=VN41
40    -L

41    +32764
42    +32765
43    +32766

44    A=V1
45    Q[0]=0
46    VN43=A
47    -47,Q[0]=0
49    A=VN42
50    -LNQ

51    +32767

52    A=V29
53    VN51=A
54    A=V30
55    VN43=A
56    A=V31
57    VN42=A
58    -58

59    +0
60    +0
61    +0
62    +0
63    +0
```

De-assembler output with minimal manual interpretation.

S 10040 PRINT V AND VN REFS.

FUNCTION

Decodes machine code word in V 17 and prints appropriate operand i.e. Vx, Vy, VNz or VNO. If latter then flag set. x is value of 'N' field, y is value of next word, z is value of 'P' field.

ENTRY

V 17 is machine code word
V 16 is second word (if any).

EXIT

Operand printed on stream 4

FLAGS

V 18 bit 0 set if VNO ref. found.

REGISTERS DESTROYED

None

LINK NEST DEPTH

5

No. OF WORDS

43

OTHER S/R CALLED

S 555, S 1

```
N
S10040 PRINT V AND VN REFS;

U

      STK VN3,23,5
      V19=4

      V22=V17&VN0                    ' MASK OUT N '
      OCT00003777
      _2,V17[11]£0                   ' IF N NOT = 0 THEN '
      _4,V22=0
      V20=VN0                        ' WRITE "V" '
      +86
[1]   _S555,0
      V20=V22
      V23=VN0                        ' WRITE N '
      +224
      _S1                            ' AND END '
      _5

[2]   _3,V22=0                       ' IF P NOT = 0 THEN '
      V22=V22&VN0
      OCT00000377
      V20=VN0
      +86                            ' WRITE "VN" '
      _S555,0
      V20=VN0
      +78
      _1


[3]   V20=VN0
      +86                            ' WRITE "V" '
      _S555,0
      V22=V16&VN0
      OCT00003777
      _1


[4]   TEXT
VN0
      V18[0]£0                       ' SET VN0 FLAG '
      _5

[5]   LDK VN3,23,5
      _L
END

S
10040 0
S10040 PRINT V AND VN REFS;
```

S 10041 PRINT "VN" INDEXING

FUNCTION

Decodes machine code word in V 17 and if operand is "VN" type, then the indexing term (if any) is printed preceeded by a comma. If not then no effect.

ENTRY

V 17 contains machine code word

EXIT

Indexing printed on stream 4

FLAGS

None

REGISTERS DESTROYED

None

LINK NEST DEPTH

5

No. OF WORDS

28

OTHER S/R CALLED

Text output (S 12)

```
N
S10041 PRINT INDEXING;

U

      STK VN3,23,5
      V19=4

      _3,V17[11]=0                    ' IF I = 1 '
      V22=V17&VN0
      OCT00000377
      _3,V22=0                        ' AND IF P NOT =0 THEN '
      V22=V17&VN0
      OCT00001400
      V22=V22[[8]],R                  ' V22 = R FIELD '
      _1,V22£1=0                      ' R = 1? '
      _2,V22£3=0                      ' R = 3? '
      _3                              ' END '


[1]   TEXT
,N+1
      _3


[2]   TEXT
,N-1
      _3

[3]   LDK VN3,23,5
      _L
END

S
10041 0
S10041 PRINT INDEXING;
```

S 10042 DE-CODE NON- FO INSTRUCTIONS

FUNCTION

Decodes machine code in V 17 (also V16,V15) and prints de-assembled form if not F = 0. If f ≠ 0 then routine has no effect.

ENTRY

V 17 contains machine code word
V 16/ V 15 contain 2nd and 3rd words if present.

EXIT

Instruction printed on stream 4.

FLAGS

"Invalid Instruction" output if F = 14.

REGISTERS DESTROYED

None

LINK NEST DEPTH

5 (10) due to internal subroutine

No. OF WORDS

153

OTHER S/R CALLED

S 10040, S 10041, S 555, S 1, S 12,

```
N
S10042 DE-CODE NON-F0 INSTRUCTIONS;

U

      STK VN3,23,5
      V19=4

      V22=V17&VN0
      OCT00170000
      V22=V22[[12]],R                    ' V22 = F FIELD '
      _11,V22=0                          ' END IF F=0 '
      _20,V22-8<0                        ' JUMP IF 1 - 7 INC '
      _12,V22£11=0
      _7,V22£15=0
      TEXT
A=
[12]  _S10040
      V22=V22-8
      SJ V22
      AL1
      AL2
      AL3
      AL3
      AL4
      AL5
      AL6

[1]   _10

[2]   TEXT
+A
      _10

[3]   TEXT
-A
      _10

[4]   TEXT
&A
      _10

[5]   TEXT
£A
      _10

[6]   TEXT 1P
INVALID INSTRUCTION
      _11

[7]   TEXT
-
      _30,L
      _11

[10]  _S10041
      _11
```

```
[20] V22=V22-1
     SJ V22
     AL14
     AL15
     AL16
     AL17
     AL18
     AL19
     AL21

[14] TEXT
SJR A
     _11

[15] TEXT
-
     _30,L
     V18[2]£0
     TEXT
,L
     _11

[16] TEXT
_L
     _9,V17[11]=0
TEXT
NQ
[9]  _11

[17] _S10040
     TEXT
=A
     _10

[18] _S10040
     TEXT
=
     _S10040
     TEXT
+A
     _10

[19] _S10040
     TEXT
=
     _S10040
     TEXT
-A
     _10

[21] TEXT
-
     _22,V17[11]=0
     V22=V17&VN0
     OCT00000377
     _23,V22=0
     _24
```

```
'      ********* INTERNAL S/R TO WRITE JUMP ADDRESSES *********

[30] STK VN3,23,5
     _34,V17[11]=0
     V22=V17&VN0
     OCT00000377
     _32,V22=0
     V19=4
     V20=VN0
     +86
     _S555
     V20=V22
     V23=VN0
     +224
     _S1
     _31,V17[0]=0
     _S10041
[31] _35

[32] V20=V16
[33] V23=VN0
     +224
     V19=4
     _S1
     _35

[34] V22=V17&VN0
     OCT00003777
     V20=V22
     _33

[35] LDK VN3,23,5
     _L

       '     ********************************     '

[23] V20=V15
[25] V23=VN0
     +224
     _S1
     TEXT
'
     _S10040
     TEXT
=V+1£0
     _10

[22] V22=V17&VN0
     OCT00003777
    _23,V22=0
[24] V20=V16
     _25

[11] LDK VN3,23,5
     _L
END

S
10042 0
S10042 DECODE NON-F=0 INSTRUCTIONS;
```

S 10043 CALCULATE NO. OF WORDS IN M/C INSTRUCTION

FUNCTION

Decode m/c in V17 and calculate if instruction is one, two or three words long.

ENTRY

V17 contains m/c word.

EXIT

V22 contains the number of words in instruction

FLAGS

None

REGISTERS DESTROYED

V22

LINK NEST DEPTH

3

NO. OF WORDS

63

OTHER S/R CALLED

None

```
N
S10043 CALCULATE NO. OF WORDS IN M/C INSTR.;

U

     VN3=V23,N3-1
     VN3=V21,N3-1
     VN3=V20,N3-1


     V21=V17&VN0
     OCT00170000
     V21=V21[[12]],R          ' V21 = F FIELD '
     ←4,V21#0
     V21[23]#0                ' FLAG F = 0 '
     V20=VN0
     OCT00006000              ' T FIELD MASK '
     ←1,V17&V20#0
     V21[22]#0                ' FLAG T = 0 '
[1]  V20=V20[[2]],R           ' R FIELD MASK '
     V23=V17&V20
     V23=V23[[8]],R           ' R FIELD '
     ←2,V23#3#0
     V21[21]#0                ' FLAG R = 3 '
[2]  V20=V20[[2]],R           ' S FIELD MASK '
     V23=V17&V20
     V23=V23[[6]],R           ' S FIELD '
     ←3,V23#2#0
     V21[20]#0                ' FLAG S = 2 '
[3]  ←6


[4]  V20=VN0
     OCT00007777              ' N FIELD MASK '
     ←5, V17[11]#0            ' IF I = 0 AND '
     ←6,V17&V20#0             ' N = 0 THEN '
     V21[19]#0                ' FLAG IO NO INSTR. '
     ←6


[5]  V20=V20[[4]],R           ' P FIELD FLAG '
     ←6,V17&V20#0             ' IF I = 1 AND P = 0 THEN '
     V21[18]#0                ' FLAG I1 PO INSTR. '
     ←6


[6]  ←10,V21[23]=0            ' JUMP IF F # 0 '
     ←9,V21[22]=0             ' JUMP IF T # 0 '
     ←7,V21[21]=0             ' JUMP IF R # 3 '
     V22=2
     ←8


[7]  V22=1
[8]  ←14,V21[20]=0            ' JUMP IF S # 2 '
     V22=V22+1
     ←14
```

```
[9]  V22=1
     ←14


[10] ←9,V21#1=0
     ←9,V21#3=0                    ' JUMP IF F = 1 OR 3 '
     ←13,V21#2=0
     ←13,V21#15=0                  ' JUMP IF F = 2 OR 15 '
     ←11,V21[19]#0
     ←11,V21[18]#0                 ' JUMP IF NO IO OR PO I1 '
     V22=1
     ←12


[11] V22=2
[12] ←14,V21#7#0
     V22=V22+1
     ←14


[13] ←9,V21[18]=0
     V22=2
     ←14


[14] V20=VN3,N3+1
     V21=VN3,N3+1
     V23=VN3,N3+1
     ←L
END


S
10043 0
S10043 CALCULATE NO. OF WORDS IN M/C INSTR. ;
```

S 10044 DE-CODE FO INSTRUCTIONS.

FUNCTION

Decodes machine code in V17, V16, V15 and prints de-assembled form if F = 0. If F not = 0 then has no effect.

ENTRY

V17 contains machine code word.
V16, V15 contain other m3c words if present.

EXIT

Instruction printed on stream 4.

FLAGS

None

REGISTERS DESTROYED

None

LINK NEST DEPTH

5 (10) due to internal subroutines

NO. OF WORDS

169

OTHER S/R CALLED

S 1, S 12, S 555,

```
N
S10044 DE-ASSEMBLE F=0 INSTRUCTIONS;

U

      STK VN3,23,5
      V20=V17&VN0
      OCT00170000
      _13,V20£0
      _7,V17[11]=0
      TEXT
EXT
      _15

[7]   _14,V17[10]=0
      TEXT
STOP
[15]  V20=V17&VN0
      OCT00001777
      V23=VN0
      +224
      _S1
      _13

[14]  _8,V17[7]=0                    ' JUMP IF S= 0 OR 1 '
      V19=4
      _5,V17[6]£0                    ' JUMP IF S= 3 '      /
      V20=VN0
      +95
      _S555,0                        ' WRITE "_" '
      _1,V17[8]=0
      _1,V17[9]=0                    ' IF R = 3 THEN '
      V20=V15
      _2                             ' JUMP ADD. IN V15 '
[1]   V20=V16                        ' ELSE IN V16 '
[2]   V23=VN0
      +224
      _S1                            ' WRITE JUMP ADDRESS '
      V20=VN0
      +172
      _S555,0                        ' WRITE "," '
      V18=0
      _50,L                          ' WRITE OPERAND '
      _41,L                          ' WRITE BIT VALUE '
      _3,V17[4]=0                    ' JUMP IF J = 0 OR 2 '
      TEXT
£0
      _4

[3]   TEXT
=0
[4]   _13,V17[5]=0                   ' JUMP IF 0 OR 1 '
      V20=VN0
      +170
      _S555,0                        ' WRITE "*" 7'
      _13
```

```
      _50,L                          ' WRITE OPERAND '
      _41,L                          ' WRITE BIT VALUE '
      _6,V17[4]=0                    ' JUMP IF J = 2 '
      TEXT
=0
      _13
[6]   TEXT
£0
      _13

'  ********* INTERNAL S/R TO WRITE BIT VALUES *********  '

[41] STK VN3,23,5
      V22=15                         ' B FIELD MASK '
      _42,V18[3]=0
      V22[4]£0                       ' 5 BIT MASK FOR DOUBLE LENTGH '
[42] V19=4
      V20=VN0
      +219                           ' WRITE "[" '
      _S555
      _43,V18[2]=0
      _S555
[43] V20=V17&V22                     ' MASK B FIELD '
      V23=VN0
      +224
      _S1                            ' WRITE B VALUE '
      V20=VN0
      +221
      _S555                          ' WRITE "[" '
      _44,V18[2]=0
      _S555                           ' IF FLAG - EXTRA BRACKET '
[44] LDK VN3,23,5
      _L

'          ***********************************          '

'   ********  INTERNAL S/R TO WRITE OPERAND  *********   '

[50] STK VN3,23,5
      V19=4
      _51,V18[3]=0
      TEXT
A,
[51] _53,V17[8]£0                    ' JUMP IF R = 1 OR 3 '
      V20=VN0
      +65                            ' WRITE "A" '
[52] _S555
      _55

[53] _54,V17[9]£0                    ' JUMP IF R = 3 '
      V20=VN0
      +209                           ' WRITE "Q" '
      _52

[54] V20=VN0
      +86
      _S555                          ' WRITE "V" '
      V23=VN0
      +224
      V20=V16
      _S1                            ' WRITE V VALUE '
[55] LDK VN3,23,5
      _L

'          ***********************************          '
```

```
'    ******* INTERNAL S/R TO WRITE SHIFT QUALIFIERS  ******  '

[30] STK VN3,23,5
     V19=4
     V20=VN0
     +172
     _S555                        ' WRITE "," '
     _32,V17[6]=0                 ' JUMP IF S = 0 '
     V20=VN0
     +204                         ' WRITE "L" '
[31] _S555
     _33

[32] V20=VN0
     +210                         ' WRITE "R" '
     _31

[33] _34,V18[3]=0                 ' JUMP IF SINGLE LEN. '
     V20=VN0
     +68
     _S555                        ' WRITE "D" '
     _35

[34] _35,V17[5]=0                 ' END IF J = 0 OR 1 '
     _35,V17[4]=0                 ' END IF J = 2 '
     V20=VN0
     +197
     _S555                        ' WRITE "E" '
[35] LDK VN3,23,5
     _L

'              ********************************                  '

[8]  V18[3]=0                     ' FLAG FOR SINGLE LEN. '
[9]  _50,L                        ' WRITE OPERAND '
     _10,V17[5]=0                 ' JUMP IF J = 0 OR 1 '
     V18[2]£0                     ' FLAG DOUBLE BRACKETS '
     _11

[10] V18[2]=0                     ' FLAG SINGLE BRACKS. '
[11] _41,L                        ' WRITE BIT VALUE '
     _30,L                        ' WRITE SHIFT QUALIFIERS '
     _12,V18[3]£0                 ' END IF ON DOUBLES ! '

     TEXT
        (
     V18[3]£0                     ' FLAG DOUBLE LEN. '
     _9

[12] TEXT
)
[13] LDK VN3,23,5
     _L
END

S
10044 0
S10044 DE-AASSEMBLE F=0 INSTRUCTIONS;
```

S 10045 READ MACHINE-CODE LINE.

FUNCTION

Reads the machine code into V17, V 16 and V 15. Input medium and reading method varies with system. This version reads integer machine code values from stream 5.

ENTRY

Machine code as integer values in stream 5

0+ initial read - next three values read into V 17, V 16 and V 15

1+ update read - next value read to V 15. V 16 becomes old V 15 and V 17 becomes old V 16.

EXIT

V 17, V 16 and V 15 contain machine code as above

FLAGS

None

REGISTERS DESTROYED

None

LINK NEST DEPTH

5

No. OF WORDS

36

OTHER S/R CALLED

S 100 (for integer input format).

```
N
S10045 READ MACHINE-CODE LINE;

U

      _1
      _2
[1]   STK VN3,23,5
      V19=5

      V23=0
      _S100
      V17=V21
      V23=0
      _S100
      V16=V21                    ' INITIAL READ '
      V23=0
      _S100
      V15=V21
      _3

[2]   STK VN3,23,5
      V19=5

      V17=V16
      V16=V15                    ' UPDATE READ '
      V23=0
      _S100
      V15=V21
      _3


[3]   LDK VN3,23,5
      _L
END

S
10045 0
S10045 READ MACHINE-CODE LINE;
```

```
N
DE-ASSEMBLER DRIVER ;

U

      V18=0
      V19=2
      TEXT
START ADD. ?
      V19=3
      V23=0
      _S100
      V12=V21
      _4,L
      _S10045                    ' INITIAL READ M/C '
[1]   _S10042                    ' DECODE LINE '
      _S10044                    ' F = 0 INSRUCTIONS '
      _2,V18[0]=0                ' TEST VN0 FLAG '
      V19=4
      V20=V16&VN0
      OCT00177777
      V20=V20[[8]],L
      V20=V20[8],R               ' WRITE SIGNED VNO VALUE '
      TEXT

      V23=VN0
      +32
      _S1
[2]   _S10043                    ' FIND NO. OF WORDS '
      V12=V12+V22
      V22=-V22
[3]   _S10045,1                  ' UPDATE READ '
      _3,V22=V22+1<0
      V18=0                      ' CLEAR FLAGS '
      V19=4
      V20=10
      _S555                      ' WRITE NEWLINE '
      _4,L
      _1

[4]   V19=4
      V23=VN0
      +229
      V20=V12
      _S1
      TEXT

      _L
END

A
```

APPENDIX VI

The Store Interface Unit

The Store Interface Unit

Introduction

The store interface unit was designed to interface a block of semi-conductor store to the F100M. The initial development was concerned with a 4K non-volatile RAM but, from the outset, the interface unit was designed for operation in a two-port store. Facilities were included for the necessary synchronising process for two-port operation, as well as a selective 'write protect' mechanism.

Design Philosophy

As can be seen from figs. 1-3, the F100 bus is asynchronous, controlled by two pairs of 'handshake' lines: $\overline{\text{JProc}}$ and KSr (transfers to store) and $\overline{\text{JSr}}$ and $\overline{\text{KProc}}$ (transfers from store). The existence of certain items of data on the bus (e.g. addresses, data for writing, etc.) is indicated by edges in the handshake line waveforms. The function of the interface unit is to recognise these edges, pass them on to the store block where appropriate, receive reply edges from the store block and pass these back to the processor. These transfers must be dealt with in a rigorous way if the error detection facilities of the F100 are not to be invalidated.

In order to facilitate two-port operation, the store cycles are modified to increase the amount of free time of the store. This is done by converting all 'read-modify-write' (RMW) cycles to read then write ones. Thus the store block is unaware of the existence of RMW cycles.

Design Details

Table 1 shows the edges that are received at the store interface together with the other signals which define them. Also shown are the generated edges and their definitions.

Edges received and generated at the store interface.

Received Edges.

Edge	Signal change	Definition	Meaning
A	$\overline{\text{JProc}}\downarrow$	-	Address on highway
Q	$\overline{\text{KDev}}\downarrow$	KSr	Accept cycle
W	$\overline{\text{KDev}}\downarrow$	$\overline{\text{KSr}}$	Accept phase
C	$\overline{\text{KProc}}\downarrow$	-	Address removed
D	$\overline{\text{JProc}}\uparrow$	KProc	Null
L	$\overline{\text{JProc}}\uparrow$	$\overline{\text{KProc}}$	Data on highway
R	$\overline{\text{KDev}}\uparrow$	KProc	Data ready
U	$\overline{\text{KDev}}\uparrow$	$\overline{\text{KProc}}$	Data accepted
F	$\overline{\text{KProc}}\uparrow$	-	Data accepted

Generated Edges

Edge	Signal change	Definition	Meaning
P	$\overline{\text{JDev}}\downarrow$	X.Rd.$\overline{\text{DisCyc}}$+X.$\overline{\text{DisCyc}}$.Wt	Start cycle
V	$\overline{\text{JDev}}\downarrow$	Ledge.Rd.WtEn.DisPhs	Start phase
S	$\overline{\text{JDev}}\uparrow$	Fedge	Data accepted
T	$\overline{\text{JDev}}\uparrow$	Ledge.$\overline{\text{Rd}}$+Wedge	Data on bus
E	$\overline{\text{JSr}}\downarrow$	KProc.Redge	Data on highway
G	$\overline{\text{JSr}}\uparrow$	$\overline{\text{KProc}}$.Fedge	Data removed
B	KSr$\downarrow$	JProc.$\overline{\text{KProc}}$.Qedge	Accept cycle
H	KSr$\uparrow$	$\overline{\text{JProc}}$.$\overline{\text{KProc}}$.Fedge	Ready for next tr.
M	KSr$\uparrow$	$\overline{\text{JProc}}$.$\overline{\text{KProc}}$.Uedge	Data accepted

Note:

X = InRnge.JProc.KSr

Wt = WtIl.WtEn

Table 1.

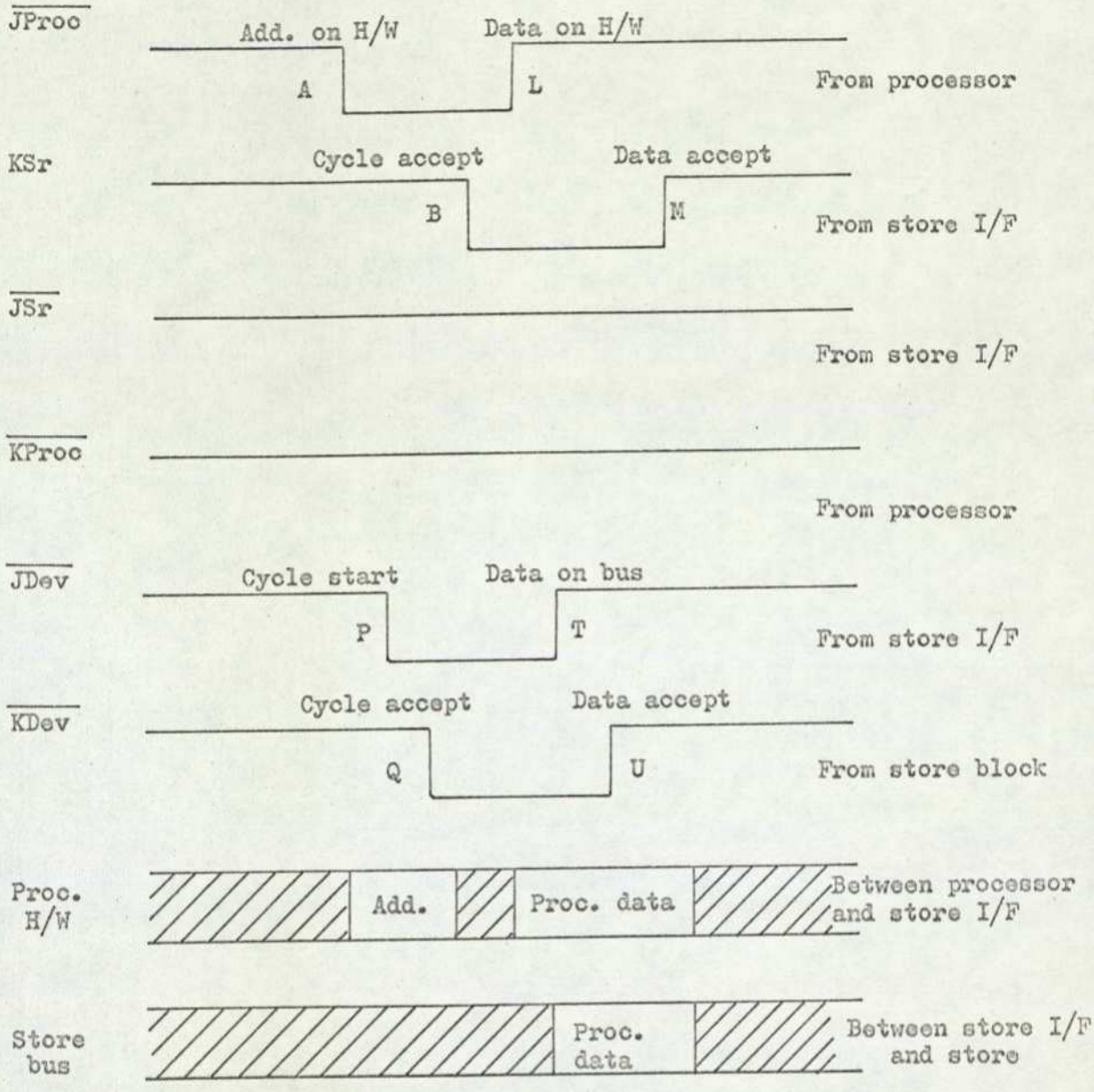


Fig. 1. Bus and store waveforms - write cycle.

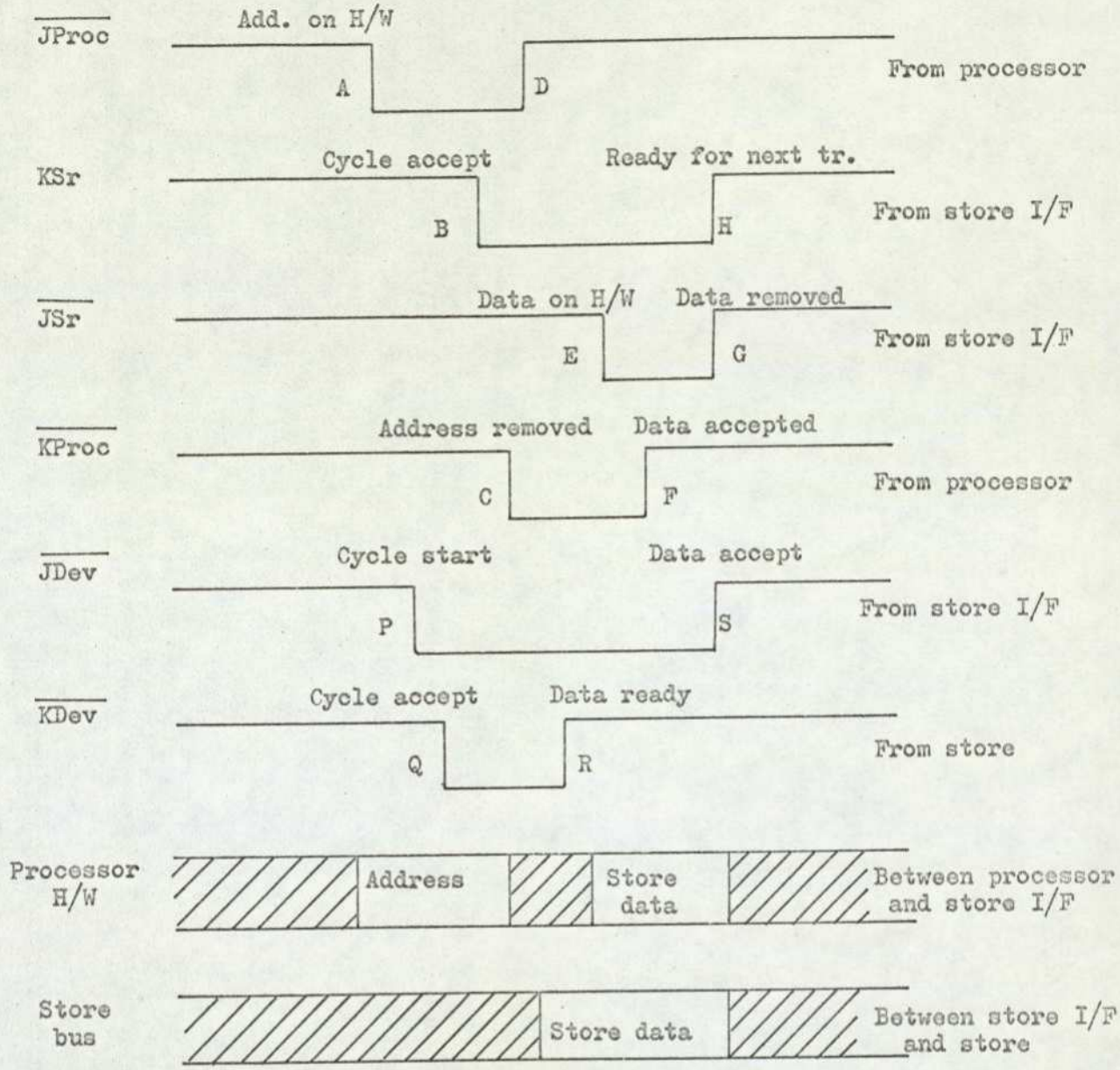


Fig. 2. Bus and store waveforms - read cycle.

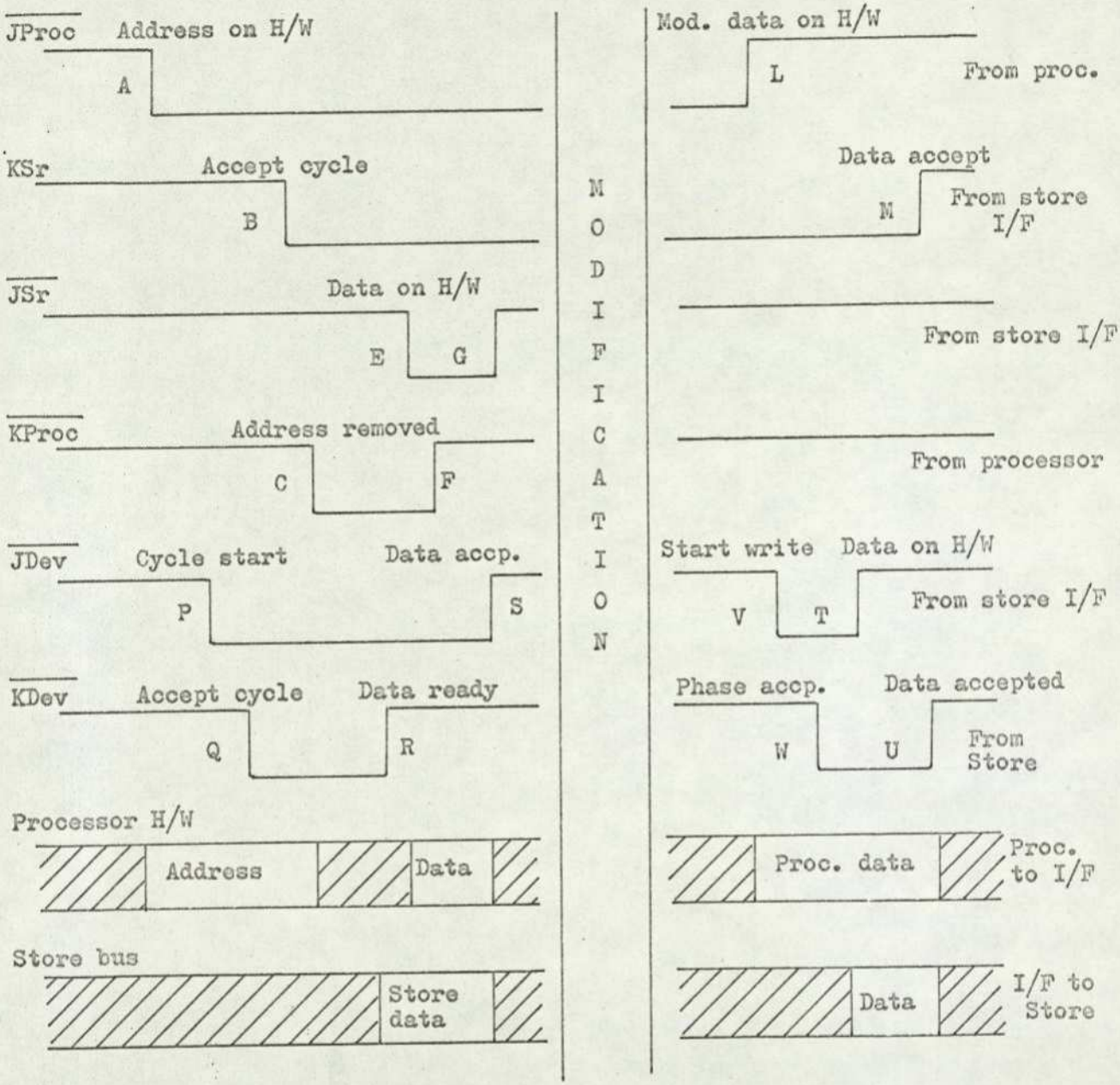


Fig. 3. Bus and store waveforms - read-modify-write cycle.

The interface unit detects edges by means of shift registers with gating to detect the edges as they shift through. In the case of $\overline{\text{JProc}}$ the shift register also generates a delayed form (Del) which is used to allow the address range logic to settle. The shift registers are clocked at approximately 10MHz and are cleared by the $\overline{\text{RsExt}}$ signal. Inverters at the signal input ensure that $\overline{\text{RsExt}}$ puts the registers into a state corresponding to quiescence in the handshake lines.

AdSb causes the bus contents to be latched into the address latches and thereby to be presented to the address range logic. This breaks the 32K address range down into eight 4K blocks each of which is selected by the insertion of a link. When an address in a selected 4K block is present in the latches, InRnge will be high. This signal is gated with Del to allow it to settle.

By a similar process of edge detection, gating and use of set-reset flip-flops, the logic implements the signal definitions of Table 1 and figs. 1-3. The outputs of the store interface are mostly open collector. Those on the processor side (KSr and $\overline{\text{JSr}}$) are pulled up on the backplane and allow multiple store blocks to reply to signals. Those on the store side ($\overline{\text{StAd0}}$-$\overline{\text{StAd14}}$, $\overline{\text{SrRd}}$, $\overline{\text{SrWt}}$, $\overline{\text{JDev}}$, $\overline{\text{PeDa0}}$-$\overline{\text{PeDa15}}$) can be disabled to allow two interfaces to be attached in parallel to one store block.

'Write Protect' Facility

The signal WtEn must be active (low) or all write cycles will be inhibited. This applies to pure write cycles and to the write phase of a RMW cycle. P edge or V edge will not occur if $\overline{\text{WtEn}}$ is inactive. This allows selective areas of store to be protected and defined as 'read only'. There are a number of ways of providing the signal $\overline{\text{WtEn}}$ but the simplest is as shown in fig. 5. This allows the user to

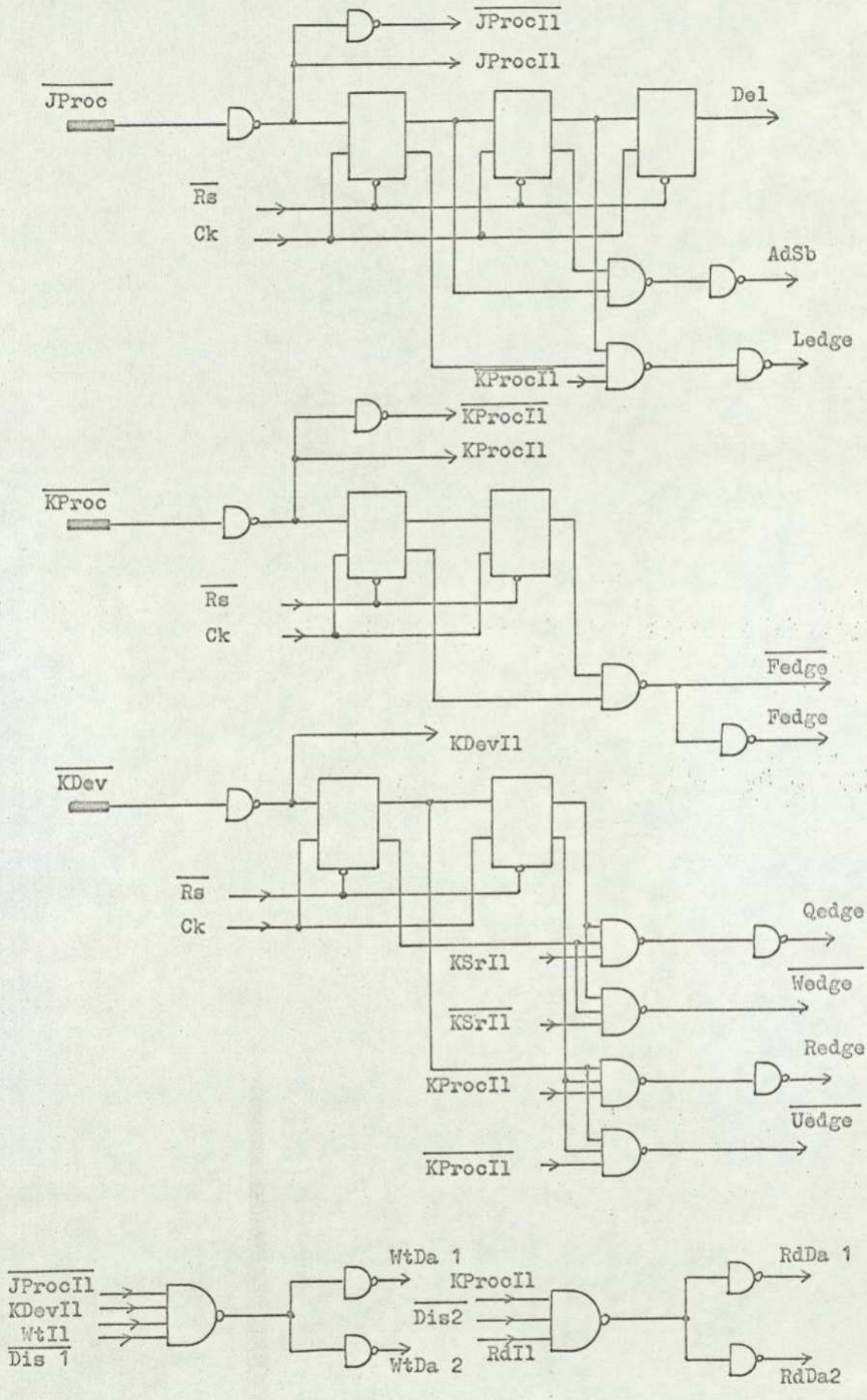


Fig. 4(a). Store interface - edge detectors.

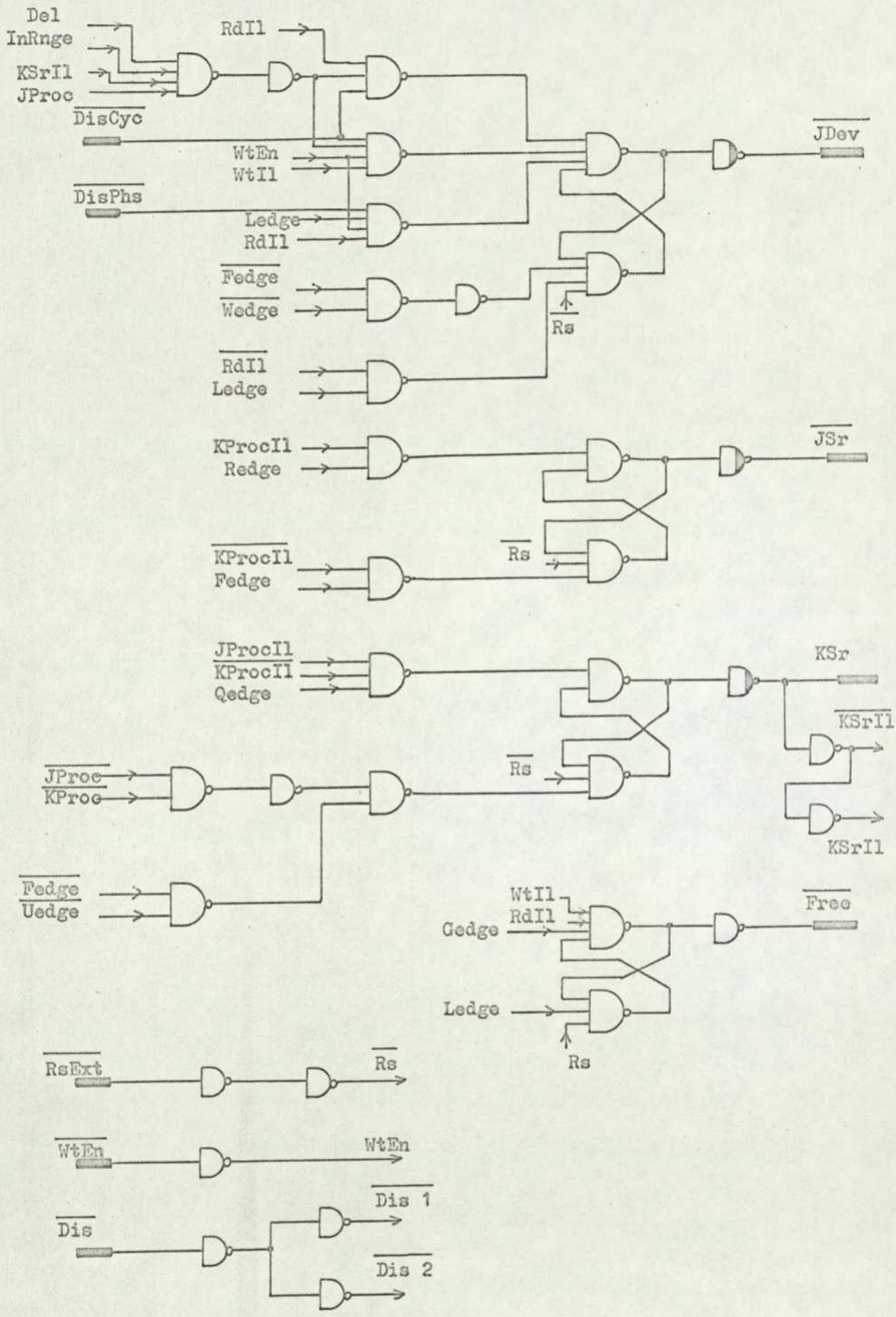
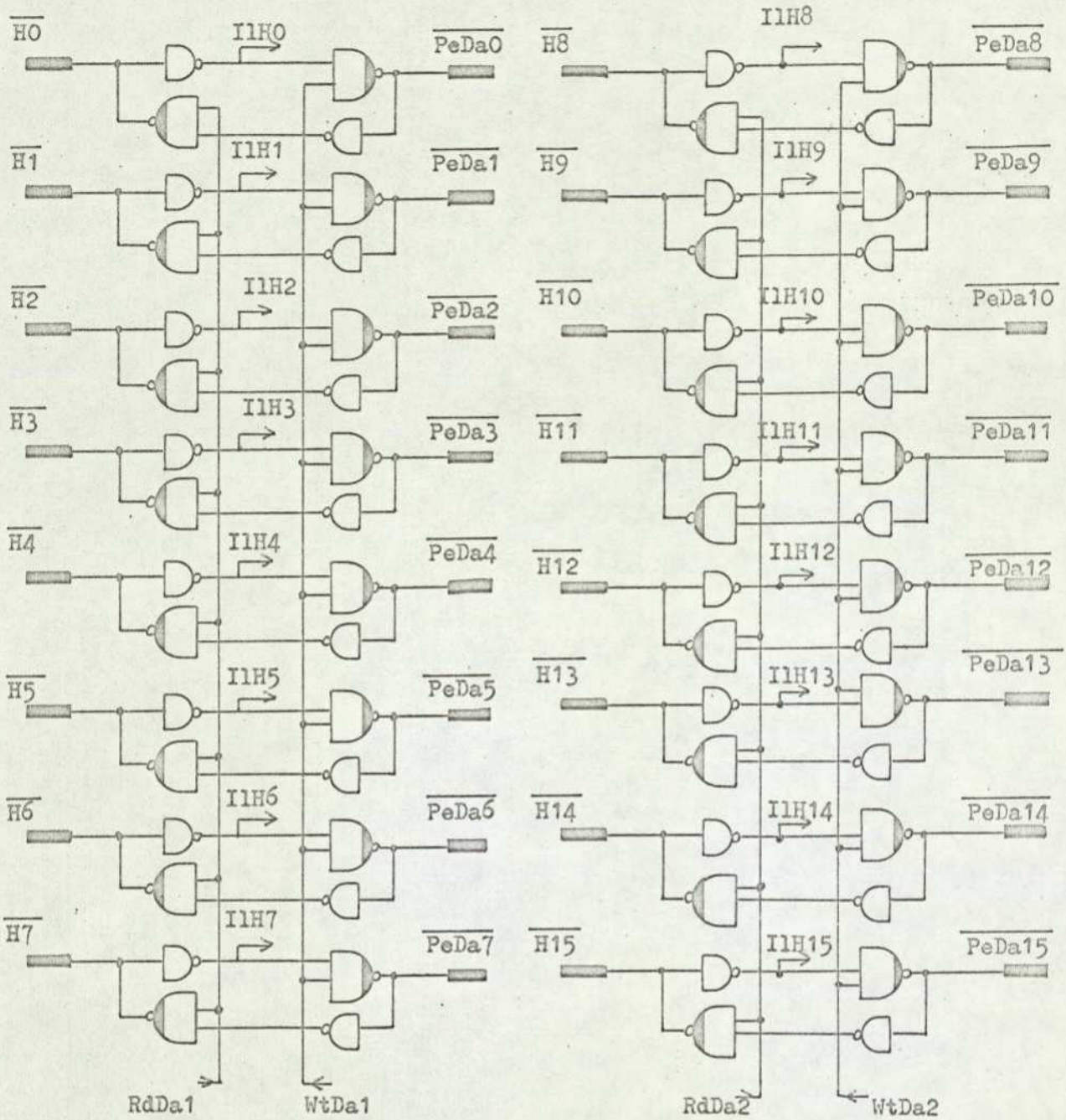


Fig. 4(b). Store interface - edge generators.



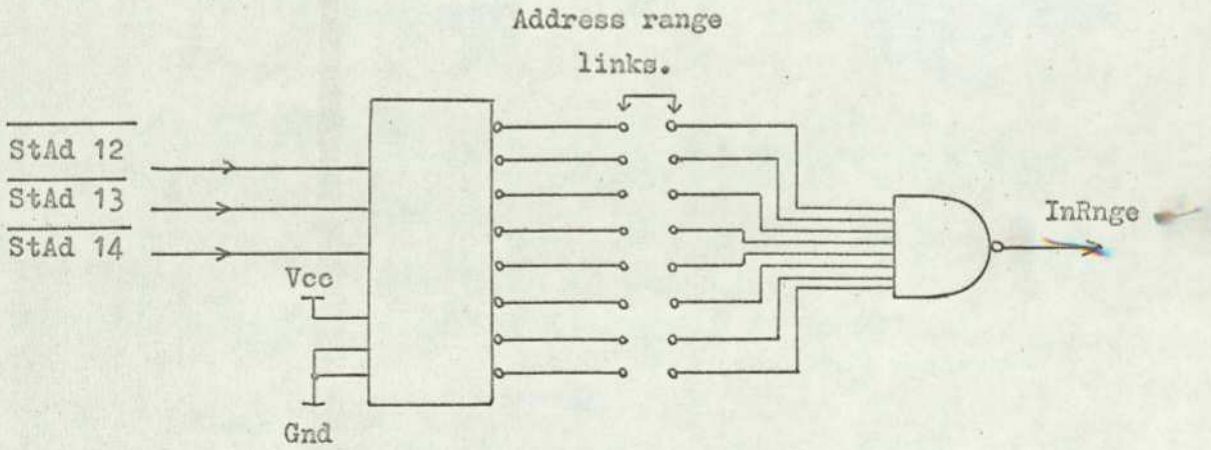


Fig. 4(c). Store interface - highway gating and address range logic.

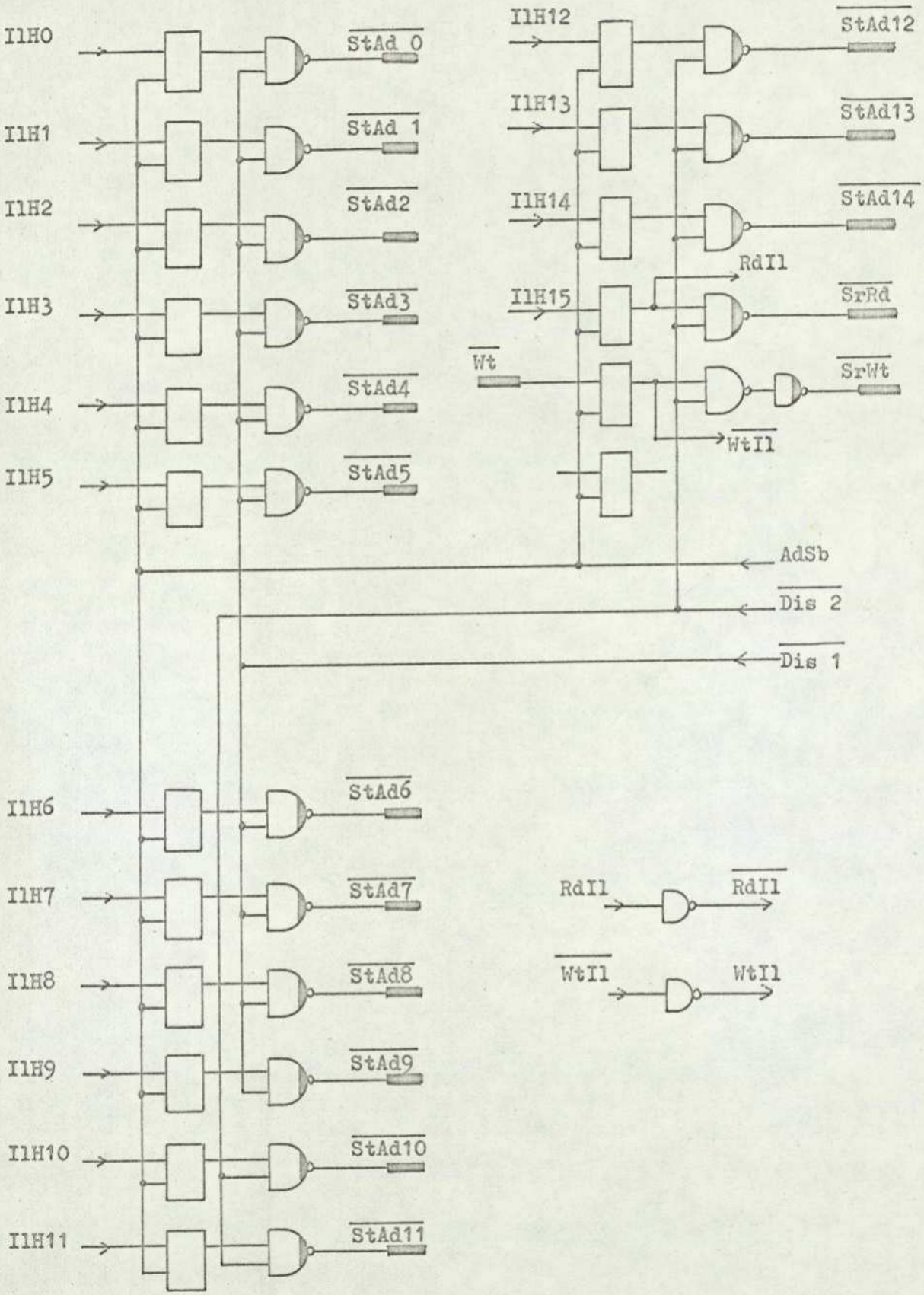


Fig. 4(d). Store interface - address latches.

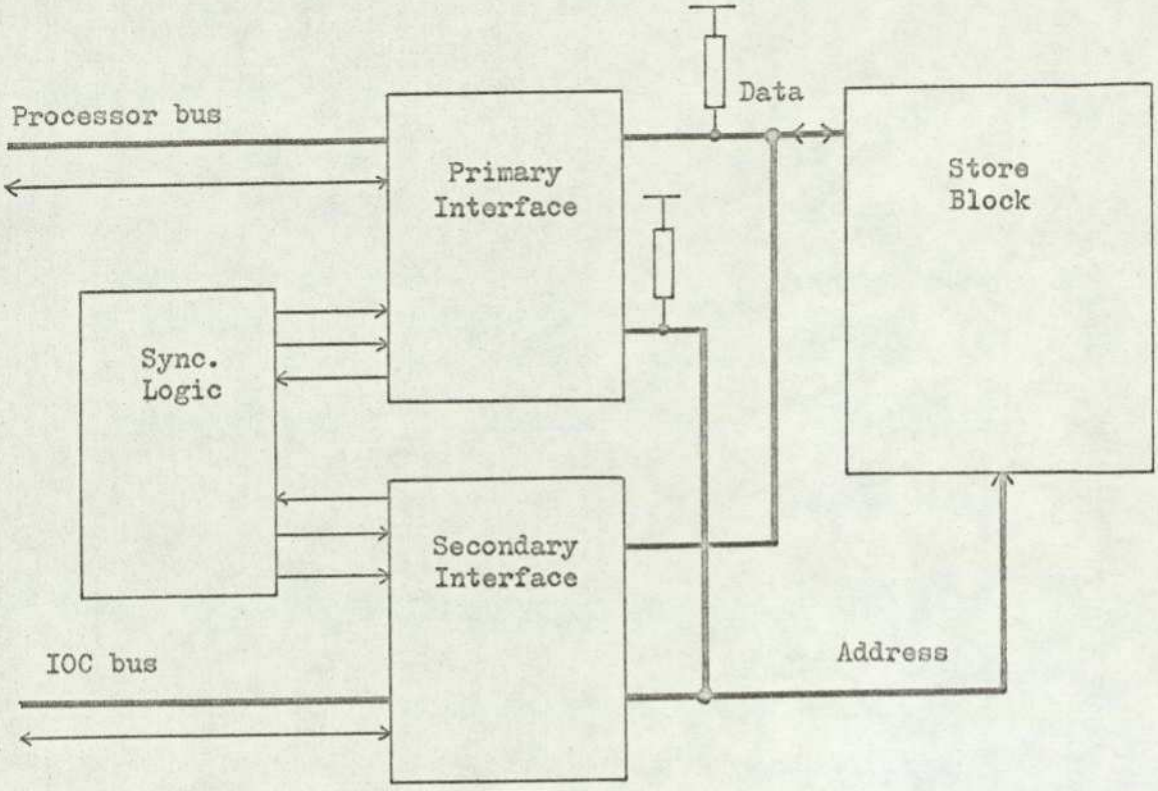
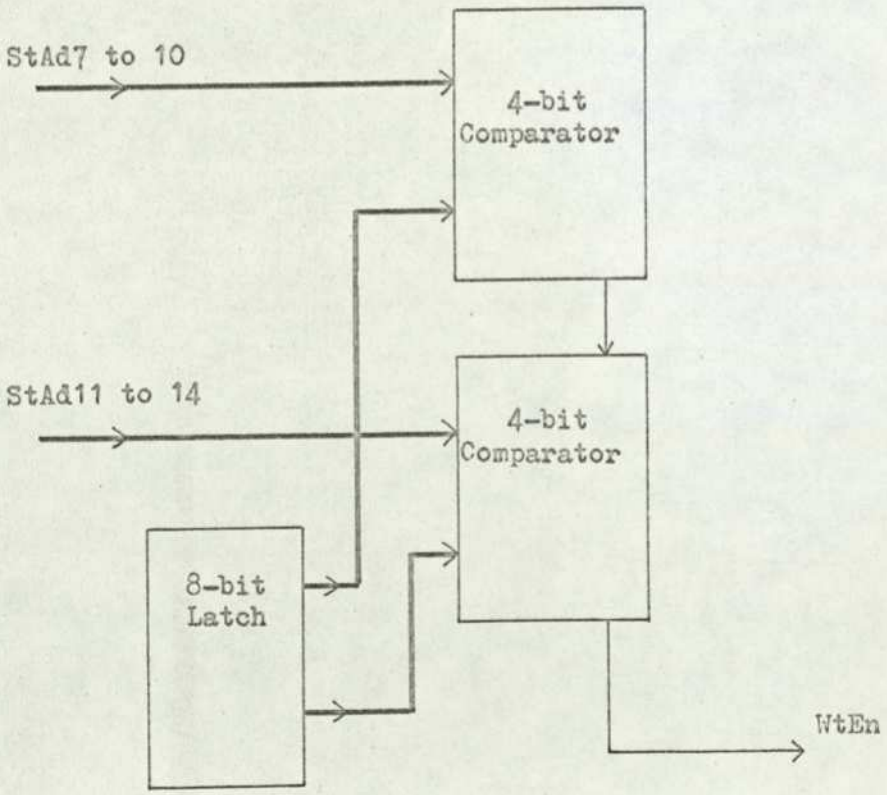


Fig. 6. Two interface units with one store block.



8-bit latch defines start of 'read only' area on 128 word boundary.

Fig. 5. A simple 'write protect' facility.

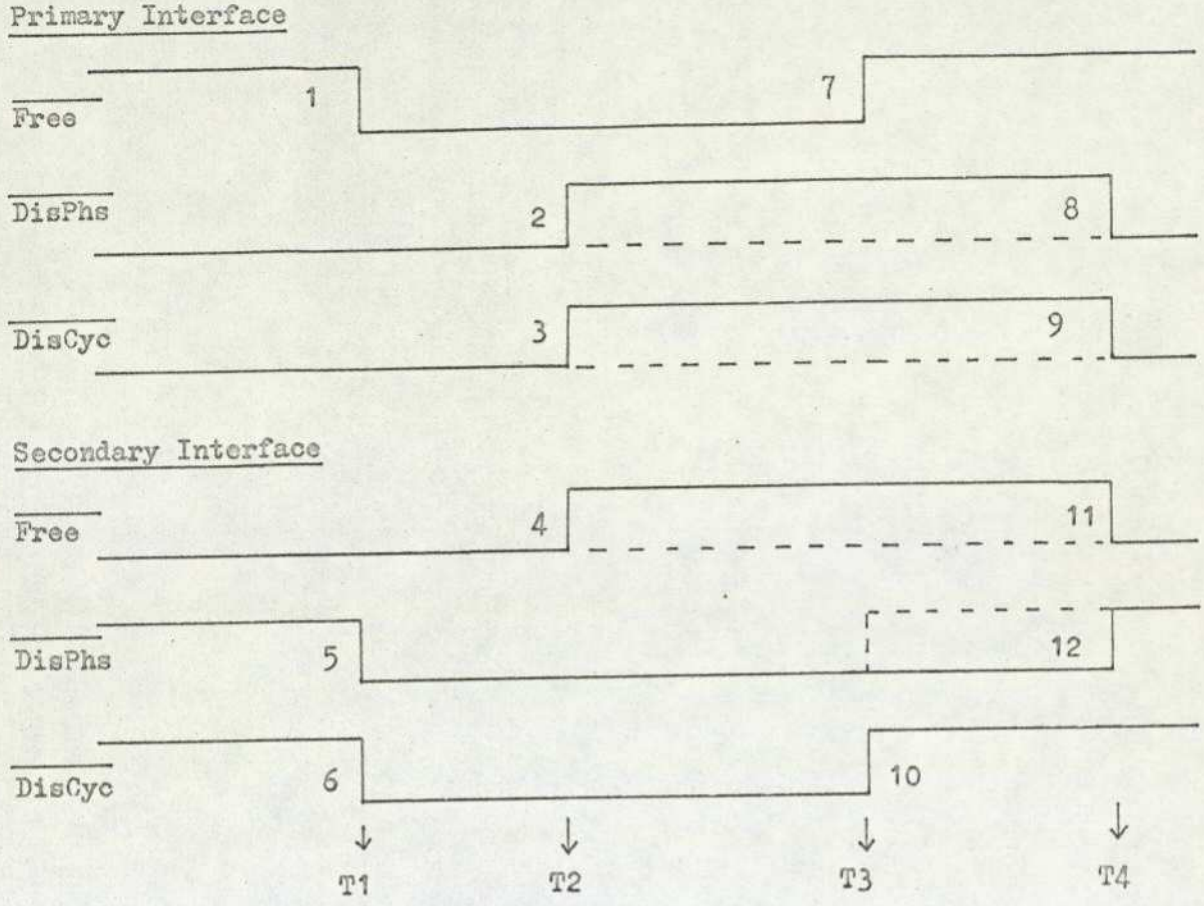
define a store address and protect all locations higher than this boundary.

Other methods using a seventeenth bit could easily be implemented.

Two-Port Operation

Fig. 6 shows two interface units working in parallel with one store block. Such a connection requires a number of facilities within each unit viz. the ability to detect when the store block is free and the ability to abandon (or suspend) transfers if the other block is using the store. Additional logic is required to arbitrate between interfaces in certain cases.

The signal $\overline{\text{Free}}$ is generated between G edge and L edge in a RMW cycle. This corresponds to the time in which the processor is modifying the data, and it is this time which is made available to the other interface. This is a specific implementation of a two-port store and is by no means a general one. It was decided that initially the two-port store would be used exclusively as a means of making available to an intelligent I-O controller, the time in RMW cycles not used by the store. In this case, therefore, the two interfaces are not equal in status: the primary one (connected to the processor) makes time available to the secondary unit (connected to the I-O controller). The secondary unit cannot demand time though it can hold up the primary interface by not releasing the store once it has had time given to it. The signals $\overline{\text{DisCyc}}$ and $\overline{\text{DisPhs}}$ are used to inhibit the primary interface from starting a memory cycle ($\overline{\text{DisCyc}}$) or the write phase of a RMW ($\overline{\text{DisPhs}}$). Fig. 7 shows this process in detail.



Edge meanings.

T1	1	Main interface waiting during 'modify' time
	5&6	Secondary interface enabled
T2	4	Secondary interface accepts a cycle
	2&3	Primary port disabled
T3	7	Primary interface requires store block
	10	No more cycles can be accepted by secondary interface
T4	11	Present cycle finishes at secondary interface
	8&12	Secondary interface completely disabled and primary interface enabled.

Dotted alternative - no cycles requested by secondary interface.

Fig. 7. Synchronising signals for the two port store interface.