COMPUTER HARDWARE TO SUPPORT CAPABILITY BASED ADDRESSING IN A LARGE VIRTUAL MEMORY

Thesis

submitted for the Degree of

Doctor of Philosophy

by

DAVID ANDREW ABRAMSON

B.Sc. (Hons)

Department of Computer Science

Monash University

August, 1982

TABLE OF CONTENTS

SUMMARY

The research described in this thesis was undertaken with the aim of providing a suitable computer architecture for supporting the development and execution of large software systems decomposed into modules according to the information hiding principle. In the course of this work, the author developed two models relevant to the achievement of this aim.

The first model is framed in terms of a memory management and addressing scheme which bases protection on capabilities and overcomes the major memory management and address translation problems found in other capability-based architectures.

The second model arose from the author's practical work in modifying an existing computer (a Hewlett Packard HP2100A) to support this architecture. It proposes a general technique for upgrading relatively primitive computers to support more advanced features, in terms of addressing modes, additional registers, new instructions and virtual memory.

Chapter 1 provides background information which led the author to undertake this research, and explains the structure of the thesis.

Chapter 2 surveys the conventional memory management systems, and describes a number of the more common problems associated with them.

Chapter 3 describes the hardware used by most memory management systems.

Chapter 4 surveys current capability based addressing schemes and highlights their problems.

Chapter 5 describes the new architectural model and shows how it solves the problems raised in earlier chapters.

Chapter 6 addresses the problem of how to implement the new model both cheaply and quickly. In doing so, it develops a general technique which can be used to implement new computer architectures.

Chapter 7 describes a practical implementation of the addressing scheme described in chapter 5 using the technique defined in Chapter 6.

The concluding chapter examines the extent to which the two models proposed in this thesis have been successful and practical.

The two major contributions of this research work are the new addressing model proposed in Chapter 5, and the architectural enhancement model proposed in Chapter 6.

The new addressing model avoids the two major problems of current capability based computers, namely memory management problems associated with small and large segments, and also address translation problems which arise in systems which make abundant use of segments. The model is shown to be more efficient than the addressing schemes used in other capability systems. Unlike other capability based and conventional computers, it is flexible enough to efficiently implement many different capability addressing structures. Consequently, the software ideas can change and evolve, without affecting the hardware.

The new enhancement technique allows many different architectural enhancements to be implemented and tested as an extension of an existing computer system, and thus allows a full scale evaluation of the ideas to be made. Because the technique allows complex structures to be constructed quickly, accurately and cheaply, it avoids the problems found in many theses which propose new architectures without coming to terms with their practical implications.

In addition to these contributions, during the course of the implementation work, a new address translation unit was devised which, whilst not significantly different in concept, is significantly different in implementation from many other units.

DECLARATION

This thesis contains no material which has been accepted for the award of any other degree or diploma in any other university, and to the best of my knowledge contains no material previously published or written by another person, except where reference has been made in the text of the thesis.

Signed:

David Abramson

Department of Computer Science,

Monash University,

August, 1982.

ACKNOWLEDGEMENTS

I am most grateful to Dr Les Keedy, who has unofficially supervised my post-graduate work. He has always been ready to listen to my ideas and guide my research. Without his enthusiasm this thesis would never have been completed.

I am also grateful to my supervisor, Professor Chris Wallace. He has contributed greatly, particularly while I was designing the MONADS II hardware. Many complex problems and solutions became easier to understand because of his assistance.

Dr John Rosenberg has had a large influence on the direction of my research. I am not only thankful for his professional advice, but also his friendship. I also appreciate his time spent proof reading this manuscript.

The design of the MONADS II processor has been influenced greatly by the members of the MONADS project, particularly Mr. Peter Dawson, Dr. Ed Gehringer, Mr. Mark Halpern, Dr K. Ramamohanarao, Dr. Ian Richards, Mr. David Rowe, and Mr. John Wells.

I am indebted to the department technical officers, particularly Mr David Duke and Mr Steve Garrison, who constructed much of the MONADS II hardware.

Many of the bugs in the MONADS II system would not have been found without the help of Brian Wallis, who developed the hardware kernel for the system.

I would like to thank Ms. Lyn Winberg for typing part of the manuscript. Thanks are also due to Dr Ken McDonell, who helped with the text formatting.

During the first year of my candidature I was funded by a Commonwealth Post Graduate Award which I appreciate greatly.

Without the friendship and encouragement of my very close friends over the last four years this thesis would never have been completed. In particular, Heather has offered much support during the long process of

writing. I am grateful for the love and patience of my family, who have put up with so much.

Finally, I appreciate the love and sacrifice of my parents. They created an environment in which it was pleasant to learn and study, often at their own expense. I only wish that my father had lived to see this work completed.

DEDICATION

This thesis is dedicated to my loving father, Dr Phillip Abramson, who died shortly before it was completed. I miss his encouragement, love and friendship so much.

1. Introduction

The research described in this thesis was conducted as part of the MONADS project in the Department of Computer Science at Monash University. The thesis topic is concerned with the development of computer hardware, in the form of the MONADS II system, but in addition to descriptions of the actual MONADS II development the thesis describes two general models which have more general applicability. The first of these is a model for building capability-based computer systems in a more flexible way than has hitherto been attempted. The second model describes a general approach which can be adopted to enhance existing relatively simple computers to support a wide variety of extensions, such as the addition of new addressing modes, new registers and support for a virtual memory.

1.1. The MONADS Project

1.1.1. Aims of the MONADS Project

The MONADS project (Keedy, 1978, 1981; Keedy, Abramson, Rosenberg and Rowe, 1982) began in 1976 with the intention of investigating methods for developing large complex software systems. The techniques used in the project are based on the information hiding principle, as advocated by Parnas (1971, 1972) and others (Wirth, 1977; Liskov and Zilles, 1974; Keedy and Rosenberg, 1981; Keedy and Rosenberg, 1982b). Using this principle software systems are decomposed into small information hiding modules, each of which performs a specific task. The data structures and algorithms used by these modules are totally hidden from other modules which make use of their services, and communication between modules is by a procedural interface. Unlike many language based solutions (Liskov, Snyder, Atkinson, and Schaffert, 1977; Dahl, Myhrhaug and Nygaard, 1968; Wulf, London and Shaw, 1976), the MONADS project provides support for the information hiding module at an architectural level (Keedy, 1982b). Moreover, modules are used uniformly to represent all addressable objects, even those which conventionally have their own mechanisms for addressing, protection and sharing, such as files (Keedy and Richards, 1982).

From an implementation viewpoint, modules are constructed from segments of memory, each of which must be addressed from within the

module and protected from access by other modules. For this reason, both modules and segments are addressed by capabilities (Fabry, 1974). Algorithms are implemented by code segments, and data structures can be held in data segments. Because the interface is strictly procedural the segments can only be directly addressed from within a module.

1.1.2. History of the Project

The first phase of the project involved building an operating system (Keedy, 1978). The idea was to demonstrate that an operating system, and in fact any large software system, could be broken into small units, each of which is implemented by an information hiding module. During the system design it became clear that a conventional computer architecture was ill-suited to the new software methodology. The major area of concern was the way that information and modules are shared and protected. Consequently, the MONADS I processor was developed in about 1978 (Hagan and Wallace, 1979; Wallace, 1978; Hagan 1977) from a modified Hewlett Packard HP2100A. This processor provided a small virtual memory (4 x 32k word address spaces) and a number of previously unsupported addressing modes, such as process stack addressing and base and index register addressing. This new hardware was still unable to implement all of the required support functions, and the idea of a hardware kernel was developed in an attempt to bridge the hardware-software gulf (Rosenberg and Keedy, 1978; Rosenberg, 1979). Much was learned from this preliminary implementation. Apart from the concept of a hardware kernel, a process structuring model was designed (Ramamohanarao and Keedy, 1978; Ramamohanarao, 1980; Keedy and Ramamohanarao, 1979), and a model was developed to describe the way that the information hiding modules should be addressed and protected (Richards and Keedy, 1978; Richards, 1982).

The MONADS I hardware presented a number of major implementation problems. First, the hardware was not totally secure. While individual user programs could be protected from each other, it was difficult to protect a program from corrupting sensitive information held on the process stack (e.g. linkage or parameter information), allowing security violations to occur. Second, user programs (and the associated data and stack space) could not be larger that 32k words, the size of one address space. This severely restricted the use of the system. Third, the

hardware provided a paged addressing scheme, which presented sharing and protection problems. Fourth, the hardware could only support three concurrent user programs. Finally, because many of the important functions were implemented in the kernel (due to insufficient room in the microprogram control store) the system was quite inefficient.

Because of these drawbacks, the author developed the MONADS II processor in 1980, again from a Hewlett Packard HP2100A minicomputer. However, this computer differed from MONADS I in two important respects. First, a different construction technique was used (Abramson, 1982a). Second, the MONADS II hardware was not specifically designed for the MONADS software methodology, but was a general capability based addressing processor able to implement many different software structures (Abramson, 1982b). At the same time as the new hardware was being designed, the MONADS software group was defining the addressing structure of the information hiding modules, which was then mapped onto the hardware. Following this, a new hardware kernel (Wallis, 1980) and a new operating system were designed. MONADS II removed some of the restrictions of MONADS I by providing a large virtual memory (Abramson, 1981), a uniform addressing mechanism and many new addressing modes (Abramson, 1980).

Whilst MONADS II demonstrated all of the principles of the new software structures, it was developed as a pilot system capable of supporting only a limited number of concurrent user programs. Thus, work began on a new processor, MONADS III (Keedy and Rosenberg, 1982), which built on the experience gained from the MONADS II system, but which would be powerful enough to provide a fast computer utility to a number of users. While MONADS II and MONADS III have many features in common (Keedy, Rosenberg, Abramson and Rowe, 1982), and may be coupled to form a multi-computer system, they differ significantly at the implementation level.

1.2. Objectives of the Thesis

This thesis contributes to the implementation of the MONADS software ideas by developing two general models, which are used in the MONADS II processor design. The first provides a hardware addressing unit, which allows information to be shared and protected in a uniform,

flexible and efficient manner. This unit should be able to be used with a number of different software structures. The model draws a clear distinction between functions which should be supported in hardware (for efficiency reasons) and functions which should be implemented in software or firmware (for flexibility reasons). Because of this flexibility the hardware may be used in other capability projects as well as the MONADS project.

The second model defines a technique for implementing complex and different computer architectures quickly and cheaply. This not only enables a full scale evaluation of the new addressing model to be performed (in terms of supporting a real program development environment) but also serves as a general technique for evaluating many new architectural ideas. This scheme has many advantages over existing techniques, particularly in terms of efficiency and simplicity.

1.3. Layout of the Thesis

Chapter 2 describes the memory management models used by conventional computer systems, such as linear memories, paged memories, segmented memories and paged and segmented memories. By examining these systems in terms of their ability to protect and freely share information, we show that they do not provide an adequate addressing scheme. The chapter concludes that the segmentation scheme offers the best logical advantages, but acknowledges that it also has many implementation problems.

Chapter 3 examines the addressing hardware currently used in computer systems, and describes the common building blocks. These blocks are then used later in the thesis to implement a new addressing model.

Chapter 4 examines an addressing model based on the segmentation scheme, called capability based addressing, and shows how it solves some of the shortcomings of conventional architectures. The problems associated with the implementation are also discussed in detail.

Chapter 5 describes a new addressing model which is based on the capability addressing scheme, but which solves the outstanding problems and provides a flexible and uniform hardware addressing mechanism. A major design consideration is that the addressing hardware should not only be able to implement the MONADS software structures, but should be

flexible enough to implement those of other capability based processors.

Chapter 6 examines the ways in which the new addressing model could be implemented, and shows that many conventional techniques are not suitable. Building a totally new computer is discarded because of lack of time and money, and other interpretive techniques (such as software and firmware implementations) are often too inefficient. The chapter then describes a general model for enhancing primitive computer architectures, and demonstrates some of the enhancements which are possible.

Chapter 7 describes the MONADS II computer system, and demonstrates the practically of the two new models proposed in this thesis.

Chapter 8 concludes the thesis, commenting on the relevance of this research and suggesting additional work which might be undertaken.

2. Conventional Memory Organizations

This chapter examines the conventional memory management models used in many computer systems. First, we consider the types of requests which a memory system must be able to satisfy. These requests come from a number of different types of software. Second, we describe the conventional memory management models used in many current computer systems. These models can then be examined in terms of the different software requests, exposing the advantages and disadvantages of each scheme.

2.1. Software Environment

From the viewpoint of the information storage system of a processor, memory demands come from three classes of software: user programs, the operating system and compilers.

2.1.1. User Programs

User programs are designed to perform some task on behalf of a user. They require the information store to possess a number of attributes, namely:

- The store must be able to save and retrieve both high speed computational data (e.g. program variables) and permanent data (e.g. file data).

- User programs which share the central processor must be protected from corrupting data belonging to other users.

- User programs should be protected from corrupting their own code and read-only data.

- User programs should be able to share certain data with other users. This allows processes to cooperate with each other in performing one task.

- The information storage, sharing and protection system should be simple to use.

- The information storage system should be efficient in space and time.

- The store should be large enough to hold all computational and permanent data.

- The store should allow dynamic memory allocation for dynamic data structures.

A user program demands a number of functions from the memory system, and is not concerned with the way that the requests are implemented. As we shall see when we examine the various conventional memory management models in use, some of these basic requirements are not well supported.

2.1.2. The Operating System

The operating system has the task of controlling all the user programs which execute on the processor. With respect to the information storage system, the operating system must:

- allocate memory when a user program is loaded, or requires more memory;

- deallocate memory when a user program terminates, or releases memory;

- control the protection system;

- control the sharing system;

- control any address modification hardware;

- be able to share code modules between users, in order to save space (this form of sharing, unlike data sharing, is not visible to the user program).

Unlike user programs, which only use the memory, the operating system must also manage and control the memory system. These requirements demand that the memory system is easy to manage and control. Again, not all of these are well supported in conventional

memory management schemes.

2.1.3. Compilers

Compilers are affected by the memory system in two respects. First, most compilers execute as a normal program (although they may have some extra privileges) on behalf of a user, and thus have the same needs as a user program. Second, they are used to translate user programs from high level languages into the machine code of the processor, and are expected to understand how to address memory. The main requirement that the compilers make on the memory system is that the addressing, sharing and protection systems are simple to use. This means that the code is easy to generate, and the compiler code generator is simple in design. The memory system should be able to represent and address the logical structures of a program, e.g. arrays and subroutines, rather than requiring the compiler to translate them into units which are meaningful to the memory.

2.2. Conventional Memory Management Systems

This section describes a number of conventional memory management models used for addressing the main, or computational, memory of a processor. Most conventional computer systems also use a secondary memory (such as a disk or drum) to hold permanent data. Whilst some also use the secondary store as part of their main memory storage system (as in virtual memory systems), the mechanisms for addressing secondary memory are not considered, because they only affect the operating system software, rather than the addressing hardware.

2.2.1. Linear Memories

The earliest memory management model used was the 'linear memory' model. In this scheme the main memory of a processor is viewed as a set of linearly arranged addressable storage locations, and each word (or location) of memory is accessed by supplying an address. These absolute addresses may be manipulated by a user program in order to access various data items. The way that linear memories are used varies depending on how many concurrent users the system must support.

2.2.1.1. Single User Systems

In the earliest computer systems, the central processor was allocated to a particular user when a job commenced, and was only relinquished when that job had terminated. In such systems, the program and associated computational data is loaded into the linear memory, and execution begins at the start of a program. The sequencing, loading and unloading of jobs is controlled by a supervisor program, which is loaded into the bottom of the memory. (In very early computer systems the supervisor was only capable of loading binary program tapes into memory.) User programs are then loaded into the memory above the supervisor. The 'linear memory' scheme, shown in Figure 2.1, has the advantage that it is simple. It also has a number of disadvantages. First, no program can be larger than the size of the main memory (although some systems use a manual overlaying scheme). Second, if a program starts a slow autonomous operation, such as an input-output transfer, the processor must remain idle until the transfer is completed. Third, there is no way of protecting the supervisor program from being corrupted by the user program. This problem was resolved by introducing a 'fence register' into the processor, which is loaded with the highest address of the monitor program. By preventing the user program from writing into memory below that address the supervisor can be protected. Fourth, because the supervisor program resides at the bottom of the memory, the user program no longer resides at address zero. Consequently, the start address, and the addresses of all variables and labels, must be adjusted so that the program begins at the top of the supervisor. This operation is called static relocation and can be quite expensive. The linear memory scheme has been used in many old single user systems, such as the HP2100A (Hewlett Packard, 1972).

2.2.1.2. Multi-user Systems

The linear addressing model can also support a multi-user environment. In this case the central processor is shared amongst many different user programs. Each program receives a slice of processor time, and is suspended either when the time slice expires or an input output transfer is started. Multi-user systems are able to use the processor more efficiently, because it is used to execute another job while other users are suspended, rather than being left idle.

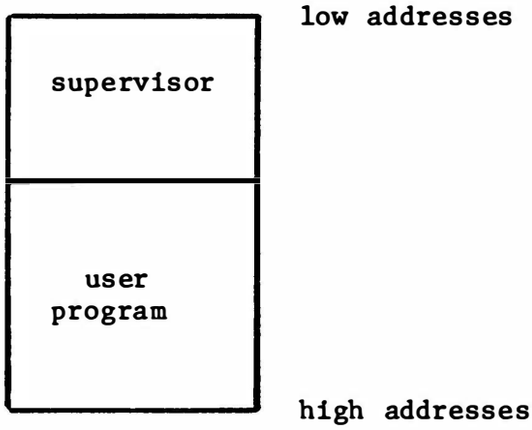


Figure 2.1 - a single user linear memory

The problem with constantly swapping the executing program is how to allocate the main memory. Two main solutions have been used. First, the entire memory is allocated to a program when it enters its time slice. All of the program code and data is loaded into memory (from a secondary storage device). When the time slice is over, the memory image is copied back into secondary memory, and the next program is loaded. This method has a number of disadvantages. While it may utilize the central processor better than in the single user system, the memory may still be under-utilized. A small program still uses the entire user area of main memory, wasting the rest of the space. Also, the time spent swapping code and data in and out of main memory is excessive, during which time the processor cannot be used. To avoid the load and unload operations every time slice, a second memory allocation scheme can be used, as shown in Figure 2.2. In this method all, or many, of the current programs are loaded into memory, each packed into the available space. When a program enters its time slice, the old register values are reloaded and the program is restarted. At the end of the time slice, the register values are saved and another program is restarted. This scheme avoids copying the code and data for a program into memory before it can be executed, and is thus much more efficient than the first solution. It also has a number of problems. First, because programs contain absolute addresses the code must be statically relocated before it can be loaded. In a single user system once the code has been relocated, it can be used repeatedly. However, in a multi-user system, the code must always be

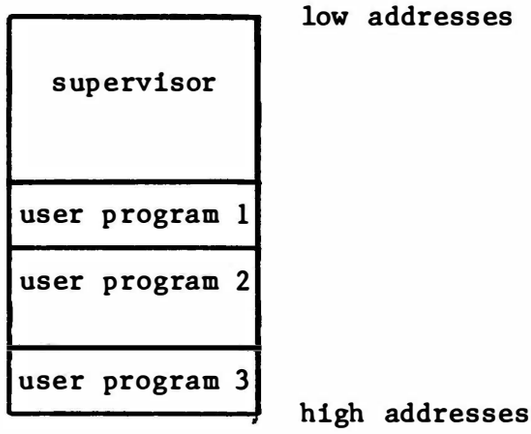


Figure 2.2 - a linear memory multi user system

relocated before it is loaded, as its position may change each time the program is executed. This operation is expensive. Second, the simple fence register scheme cannot protect programs from corruption. A more elaborate protection scheme places a base register at the lowest address of a program, and a limit register at the top of the program. Addresses can then be restricted to a particular addressing region. Third, as programs are loaded and unloaded, the memory may become fragmented. This 'external fragmentation' complicates loading new programs, as insufficient free space may be available to hold an entire program. To simplify the problems of static relocation and memory management, a dynamic relocation scheme was devised.

In the dynamic relocation scheme each program assumes that it is loaded at address zero, and the processor augments each address by the contents of a base register before it reaches the memory. This new address can also be validated against a limit register, thereby protecting other programs from corruption. These dynamic relocation registers also assist with memory management. A statically relocated program cannot be moved in memory once it has been loaded, because it may have absolute addresses in data variables. Thus, if insufficient contiguous space is available, it may be impossible to load a program into memory. Dynamic relocation registers, however, allow a program to be moved; thus the memory may be periodically reorganized. Unfortunately the cost of moving programs in memory is quite high, and it may be more

efficient to waste some of the main memory.

Another solution aimed at simplifying the external fragmentation problem is the partitioned scheme, used in the ICL System 4 (ICL, 1971) and some of the IBM 360 range (Belady, Parmelee and Scalzi, 1981). In this scheme the main memory is divided by the operating system into a number of fixed size partitions. When a program is loaded into memory it is placed in a free partition. Because memory is allocated in fixed size units, the task of memory management becomes much simpler, and external fragmentation is eliminated. Unfortunately internal fragmentation occurs, as programs which are smaller than a partition will waste space. Thus, while memory management may be simpler, the fixed partition scheme may waste even more space than a variable space allocation scheme. Also, large programs must be manually overlayed so that they fit in a partition. The effect of internal fragmentation may be diminished by using small partitions; however, this limits the maximum size of a program (unless it is broken down into overlays).

Another problem with the linear memory model is that from the compilers' viewpoint the memory image is totally unstructured. It must allocate space within the address space for all of the data structures and code of a program without any assistance form the memory management system.

2.2.1.2.1. Sharing Memory

It is often desirable to share access to areas of memory between user programs. In the swapping scheme sharing is awkward. The supervisor program must allocate an area of shared memory at a reserved address, and as programs are loaded and unloaded this area must remain unaltered. In this way many programs can share access to a statically defined set of data items. In the scheme which loads all programs into memory at the same time, data items may be shared by using the absolute address of the item. This procedure does not work if base and limit registers are used to protect a program, or if dynamic relocation registers are used. Sharing of code is extremely difficult in a linear memory scheme. Because of the problems, very few linear memory computers allow sharing at all.

2.2.1.2.2. Protection between User Programs

We have already shown that user programs may be protected from corruption by base and limit registers. This scheme has been used in many linear memory computers, such as the ICL1900 series (ICL, 1976). The partition scheme can also provide inter-program protection, as shown in Figure 2.3. In this scheme the hardware recognizes a number of fixed size areas (in the IBM 360 range this was 2k words in size), and associated with each area is a protection lock which holds the identity of the program. (If the partition size chosen by the operating system is larger than the area size recognized by the hardware then a number of protection locks may be assigned to the same partition.) The central processor has a current-protection-key register, which holds the identity of the currently executing program. When a program enters its time slice the current-protection-key register is loaded with the identity of the program. Each time memory is addressed, this value is compared to the protection lock for the area being addressed. If they differ, then a protection violation has occurred, and the program is aborted. Consequently, a program can only address areas which it owns. Whilst it is conceptually possible to dynamically change the values held in the protection locks to facilitate controlled sharing, this has not been implemented. Thus, it is very difficult to share memory between

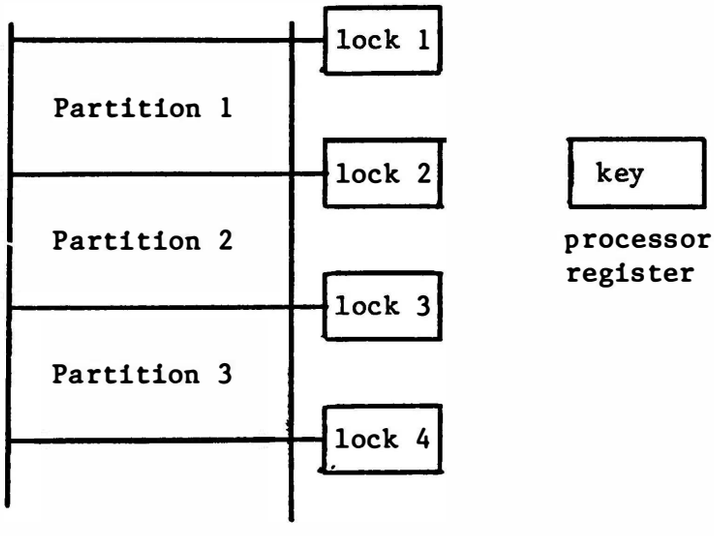


Figure 2.3 - the protection key system

programs. None of the protection systems discussed protect a program from corrupting its own code. Most of these schemes make the sharing of memory extremely difficult.

2.2.2. Paged Virtual Memories

The concept of a paged virtual memory was first proposed and implemented by the Atlas design team in the late 1950s (Fotheringham, 1961; Kilburn, Edwards, Lanigan and Sumner, 1962). The scheme consists of detaching the logical view of memory as seen by a program from the physical organization of the main memory. The logical memory, or virtual memory, is mapped onto the main memory by a mapping function. The implementation of this addressing structure requires that both the virtual and physical memories are divided into a number of fixed size pages. Consequently, an address (either virtual or physical) consists of two portions, a page number field and a within page displacement field, as shown in Figure 2.4. A mapping unit can then map pages of virtual memory onto pages of physical memory as shown in Figure 2.5. The within page displacement from the virtual address is used as a within page displacement for the physical page.

The paged scheme has a number of general advantages. First, the type of reference applied to a page of virtual memory may be controlled. Associated with each page of virtual memory may be some access rights, which determine whether the page may be read from, written into, or executed as code. With this information, the processor can monitor every memory reference and report addressing errors. Unlike a linear memory, the paged memory organization can protect a user program from modifying its own code, and pages of constants can be protected from being modified. Second, a contiguous area of virtual memory need not be allocated contiguously in main memory. The mapping function can map

Figure 2.4 - a paged virtual address

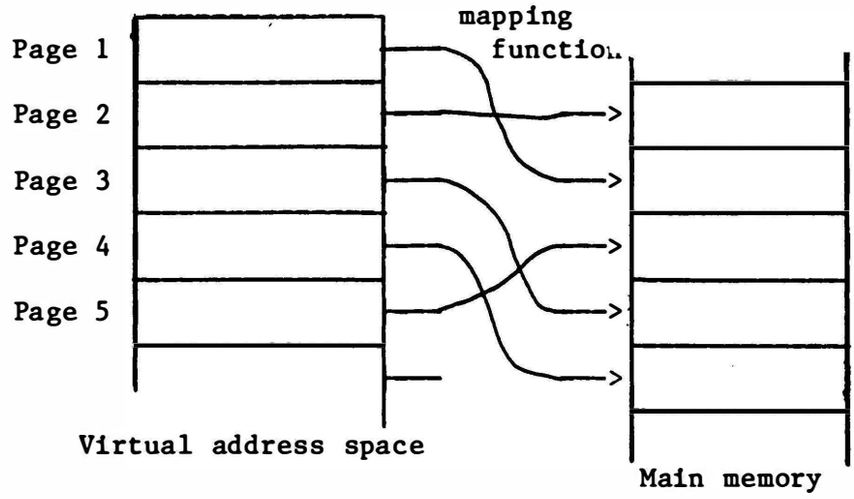


Figure 2.5 - a paged virtual memory

pages of virtual memory arbitrarily onto main memory, allowing large contiguous areas of virtual memory to be created. This makes memory management easier and eliminates the external fragmentation experienced in the linear memory scheme. Third, the virtual space may be larger than the physical space. Any pages of virtual memory which are not mapped onto a page of physical memory can be tagged absent. A reference to these pages causes an addressing error, called a page fault, which is similar to an access rights violation. Because the virtual address space may be very large, programs larger than the main memory may be loaded. If an absent page of virtual memory is addressed, a supervisor program may fetch a copy of the page required from secondary memory, find a free page in main memory, place it in main memory, and update the mapping function. When the reference is attempted again, the processor will address the correct page of physical memory. This operation is called demand paging, as pages are only fetched into main memory on demand. Some systems have experimented with prepaging, i.e. attempting to fetch pages before they are required. Unfortunately, it is very difficult to determine the access patterns of a program, and because the overheads of moving unneeded pages into main memory are high, prepaging is not commonly used. Fourth, as we will see shortly, user programs may be protected from interference from each other.

The paged scheme also has some disadvantages. Because memory is allocated in fixed size units, some space will be wasted in the last page of the virtual address space. This internal fragmentation is the same problem experienced in the partitioned linear memory. We shall examine some more serious disadvantages later, in terms of the logical properties of pages.

2.2.2.1. Single User Systems

In a single user system, the program executing on the processor is loaded into the virtual address space. To the user program, the memory appears to be linearly arranged. However, unlike the linear memory model, the pages may be allocated randomly from main memory. In addition, pages may be protected from inadvertent corruption. When the program terminates, the virtual address space can be loaded with a new user program.

2.2.2.2. Multi-user Systems

Most paged computers support a multi-user facility by creating a number of virtual address spaces. Each user program is loaded into a different address space and the main memory is composed of pages of many different programs. Each of the virtual spaces is then mapped onto the main memory by its own mapping function, as shown in Figure 2.6. When a program enters its time slice, the appropriate mapping function is selected so the program can only address its own pages. When the time slice is over, a new mapping function is selected. This arrangement is similar to the swapping technique used in the linear memory scheme. However, only the mapping function is altered after each time slice (which can be performed very quickly), rather than copying each address space to and from secondary memory. The demand paging system allows a program to load its pages into main memory as they are required.

2.2.2.3. Address Translation

So far we have assumed a mapping function between virtual addresses and physical addresses, without considering how this function can be implemented. This translation operation is performed each time the processor accesses memory. The virtual address may either be translated into a main memory address, in which case the reference can proceed, or

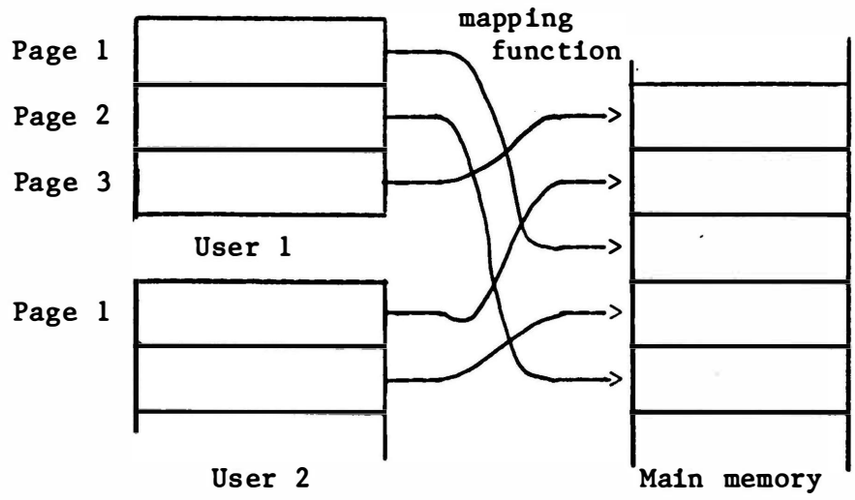


Figure 2.6 - a paged multi-user system

a secondary memory address, in which case the page is fetched into main memory. The implementation of the mapping function depends on the size of the virtual memory and the size of the main memory, and four categories can be identified: small virtual address spaces, small main memories, large virtual address spaces with large main memories and very large virtual address spaces. These categories will be examined again in the next chapter, which deals with the hardware necessary to translate addresses. Here we briefly consider the implementation of these different categories from the view of the operating system software.

2.2.2.3.1. Small Virtual Address Spaces

When the virtual address space is comparatively small the virtual page numbers can be translated into physical page numbers by a directly indexed table held in main memory, as shown in Figure 2.7. The virtual page number is used as an index to a page table entry. The page table entry contains the physical page number, the access rights for the page, and an absent/present flag, which is set if the page is present in main memory. If the page is not present, then the physical page number field can be used to hold the secondary memory address of the page. When this table is small enough, it may be held in a special address translation memory, rather than in main memory, and this is discussed in more detail

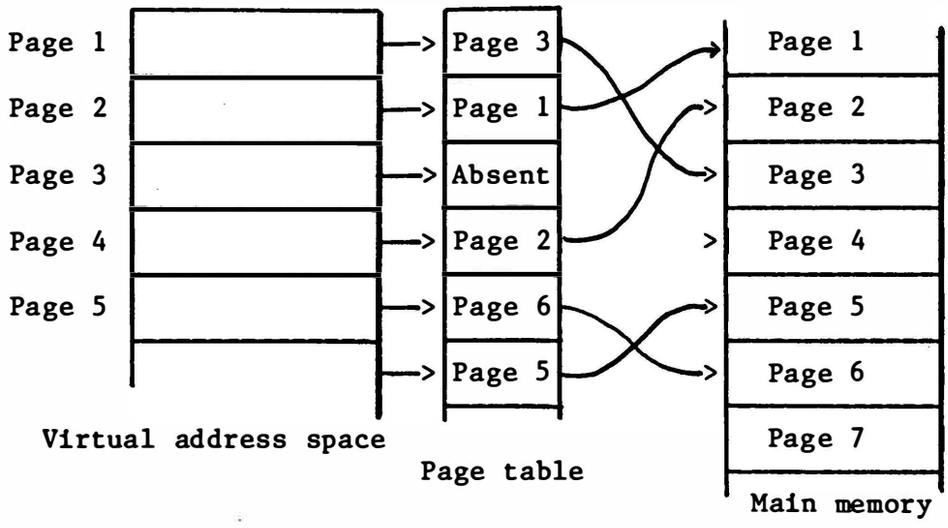


Figure 2.7 - a small virtual address space

in the next chapter. The size of the main memory has little effect on the size of the mapping table, and only influences the width of each page table entry. The size of the virtual memory affects the length of the table, and the scheme is thus only viable for small virtual address spaces.

In a single user system only one page table is required. In a multi-user system, a page table is required for each virtual address space. A register is often used to hold the address of the current page table. When a user program enters its time slice, this register can be modified to point to the new page table, thus changing the mapping function. Similar address translation schemes have been successfully used in processors such as the HP21MX (Hewlett Packard, 1974), Data General Eclipse (Data General, 1974).

2.2.2.3.2. Small Physical Memories

If the main memory has a limited number of pages, a different translation technique can be used. Rather than indexing the page table on virtual page number, this method uses the main memory page number as an index value, as shown in Figure 2.8. Each page table entry contains a virtual page number, some access rights and an invalid flag. When a virtual address is translated into a main memory address, the page table

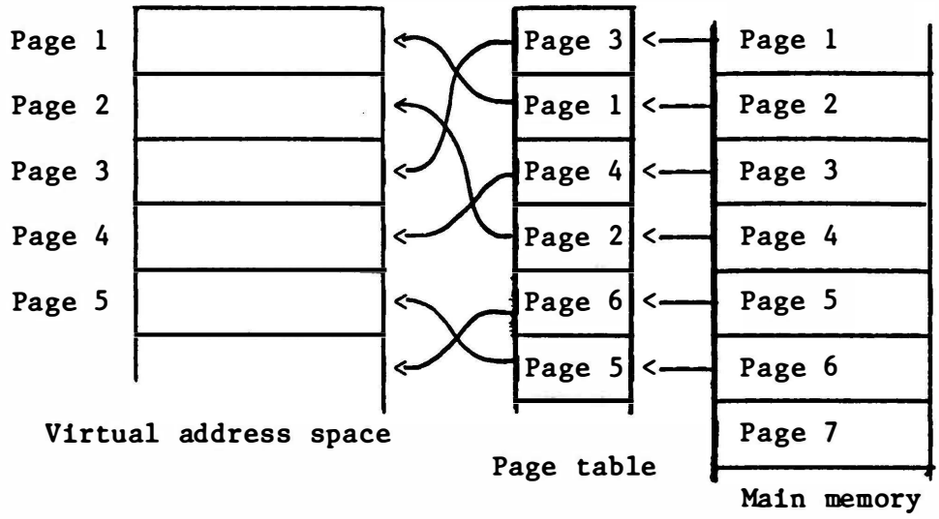


Figure 2.8 - a small main memory

is searched associatively for the virtual page number. If found, the index value of the page table entry is used as the main memory page number. If not found, then the virtual page is not present in main memory, and is fetched from secondary memory. If a page of main memory is not mapped to a page of virtual memory, then the invalid flag must be set. Special hardware is required to search the page table, and this is described in Chapter 3. This technique is sensitive to the size of the main memory, as this affects the length of the page table, and is only used when the main memory is small. In a single user system there is only one page table. In a multi user system, each user address space uses a different page table. Those pages of main memory which do not pertain to a user program, must have their page table entries flagged invalid.

2.2.2.3.3. Large Virtual Address Spaces

When both the virtual address space and the main memory become large the techniques discussed above become infeasible. A large virtual address space makes the directly indexed page tables too large to be placed permanently in main memory, or in a special hardware table. Similarly, because of poor hardware an associative search becomes difficult. (In the next chapter we will examine some associative schemes which are effective.) One solution places directly indexed tables in

main memory (which can be addressed by a special processor register), and swaps them between memory and disk as they are required. In some systems the page tables may be placed in virtual memory themselves, as in the VAX 11/780 computer (Digital Equipment Corp., 1979), and the paging mechanism can be used to address the page tables. Consequently, the page tables are mapped onto main memory by page tables, which may then be small enough to place permanently in main memory. When an address is translated, the virtual page number is used to form a virtual address of a page table entry. This virtual address is then translated into a main memory address by the page table for the page tables. At any stage, either of these translations may cause a page fault, i.e. the page table entry is not present in main memory at the time. Each address translation may cause a number of page faults to be generated. Further page faults may be generated when a page is removed from main memory (possibly to find room for a page which has already caused a fault), as the present/absent bit must be updated in the page table entry. To prevent an endless loop of page faults, this scheme must always have a pool of free pages. In this way a page may be brought into memory without having to remove another page, and thus cause further page faults. The scheme requires special hardware to assist in address translation, and this is discussed in the next chapter.

2.2.2.3.4. Very large virtual address spaces

This class of virtual memory is typically used to hold file data as well as computational data. The only processor which has attempted this operation (namely MULTICS (Organick, 1972)) did not use a very large virtual address, and could use normal page tables. In this class of address space the page tables would certainly be placed in virtual memory, and would be very large indeed. Consequently it requires special hardware to translate addresses. Such hardware is discussed in Chapters 3, 4 and 7.

2.2.2.4. Protection

A paged virtual memory offers a multi-user system a number of levels of protection. First, programs are protected from corruption by each other. Because each program is loaded into its own virtual address space, and has its own mapping function, it is impossible for a program

to inadvertently address the pages of another. Second, a program may be prevented from corrupting pages of code, or non-modifiable data. In this way a program is partially protected from itself. Third, the operating system (or supervisor program) is treated as another user program, and is loaded into its own address space. Thus, fence register schemes are not needed to protect the operating system from corruption.

2.2.2.5. Sharing

Whilst enforcing an effective protection mechanism, a paged memory scheme also allows controlled sharing of data and code. The page table structure allows a page of main memory to appear in more than one virtual address space, as shown in Figure 2.9. Moreover, each address space can have different access rights for the page. Thus, one program may be allowed to read from a shared page, whilst another may be allowed to read from and write to the page.

Because the operation of removing a page from main memory may become complicated by the need to locate and update all page table entries which refer to it, a variation of this sharing scheme may be used. To simplify this problem, some systems maintain an additional table, located between the page table entries and the main memory, as shown in Figure 2.10. Each page table entry which addresses a shared

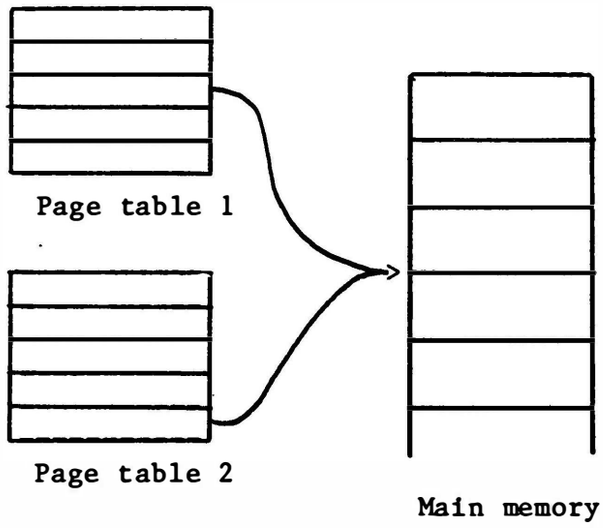


Figure 2.9 - shared pages

page contains an index into the additional table. This table entry then contains the main memory page number. If the page is banished from memory, only one entry needs to be adjusted.

2.2.2.6. Memory Allocation

Memory allocation in the paged scheme is much simpler than in the linear model. Because memory is allocated in fixed size units, there will always be an area of the correct size available when space is required, even if some other pages must be removed from main memory.

2.2.2.6.1. Page Replacement

When a page is brought into main memory from secondary memory, it is often first necessary to remove another page. The choice of which page to remove may have a significant effect on the efficiency of the computer system, i.e. if the wrong page is discarded then it may need to be fetched again soon afterwards, leading to an inefficient situation known as 'thrashing'. Many different discard algorithms have been devised (Denning, 1970), the most common being Random, First In First Out (FIFO), Least Recently Used, Atlas Loop Detection (Kilburn, Edwards, Lanigan and Sumner, 1962) and Working Sets (Denning, 1968, 1980); these often require hardware assistance (Morris, 1972) and will not be

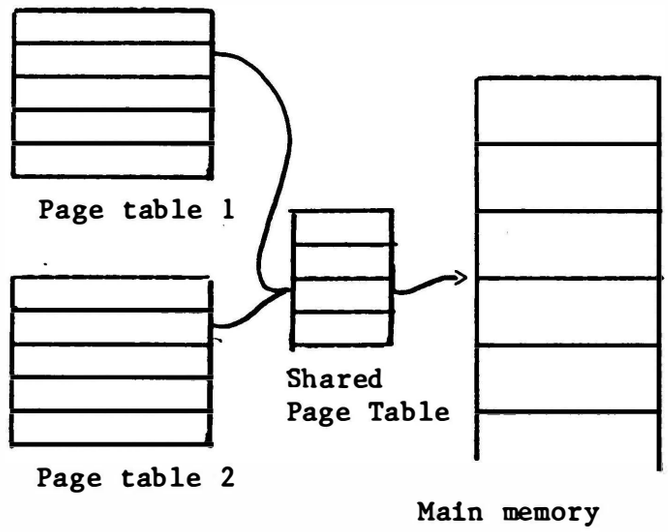


Figure 2.10 - shared pages

discussed further.

2.2.2.6.2. Internal Fragmentation

Because memory is allocated in fixed size units, namely pages, each address space will totally occupy an integral number of pages, and only partly occupy the last page of the address space. Consequently, each address space will waste, on average, half a page of memory. This internal fragmentation is also experienced in the partitioned linear memory, and is a major disadvantage of the paged memory scheme. It has been shown that a significant proportion of memory may be wasted from internal fragmentation (Randell, 1969).

2.2.3. Segmented Memory Schemes

The segmented memory scheme is similar to the paged memory model. Both map a logical view of memory onto the main memory of the processor, and both allow information to be protected and shared amongst users. However, pages are an inappropriate unit of protection and sharing as many unrelated structures may be placed in the same page. Also, the compiler must place the logical structures of a program into the pages of a linear address space, which both hides the logical structure of programs from the architecture and requires more address calculation by the compiler. To avoid these problems the segmented scheme divides the virtual memory into a number of variable length segments, instead of fixed length pages. Each logical component of a program, such as a code procedure, data array, and scalar variable, is loaded into a segment of memory, rather than being arbitrarily decomposed into fixed length pages. Each segment is protected by a set of access rights, in the same way as pages may be protected.

Each process has access to a set of segments. Typically, a segmented address consists of a segment number (which is usually numbered relative to the segments of a program or process) and an offset within the segment. This address, as shown in Figure 2.11, is then translated into a main memory address before the reference can proceed.

2.2.3.1. Address Translation

For each segment address, the main memory base address of the segment, and the size of the segment must be determined. Also, to allow

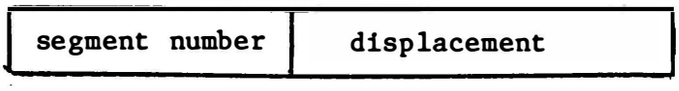


Figure 2.11 - a segmented address

segments to be removed from main memory and fetched on demand (in the same way as demand paging) it must be possible to flag a segment as absent from main memory, and to supply its secondary memory address instead. There are two common methods of address translation, segment lists and tagged absolute descriptors.

2.2.3.1.1. Segment Lists

In this scheme, each process has access to a list of segments, as shown in Figure 2.12. Memory reference instructions can address segments by supplying a segment number relative to the process (i.e. starting at zero) and a within segment offset. The segment number, which forms an index into the process segment list, must then be translated into a main memory base address. Contained in each entry of the segment list is the main memory base address of the segment, the size of the segment (or the main memory limit address), a set of access rights and a present/absent flag. The word in memory address is calculated by adding the base address to the offset.

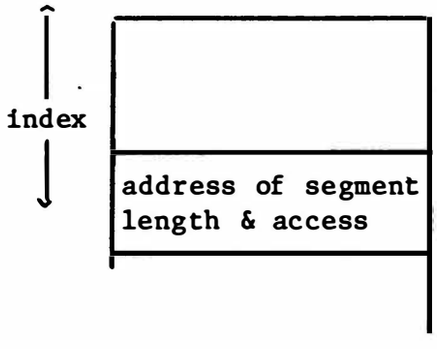


Figure 2.12 - a segment list

The current segment list is usually held in main memory, and may be located by a special processor register, in much the same way as a page table. The segment list is protected from corruption by the user program in the same way as other segments may be protected, as we will see later.

2.2.3.1.2. Tagged Descriptors

An alternative address translation system is the tagged absolute descriptor mechanism, as used in the B6700 family of computers (Organick, 1973). In this scheme, each segment descriptor, consisting of a main memory base address, segment size, access rights and present/absent flag, is placed in the data area of a process, e.g. the process stack, and is possibly intermixed with program variables. These descriptors are then used as pointers to segments. A segment cannot be addressed without the correct descriptor. So that descriptors cannot be modified, and used to address other segments of memory, it is possible to protect them by using tag bits (Feustal, 1972; Myers, 1978a, 1978b). Each word of main memory has a tag field attached to it. A word which is tagged as a descriptor cannot be modified by a normal user program. (Although the B6700 hardware allows a program to modify descriptors, the compilers prevent high level language programs from changing them.) The tags are also used for detecting the difference between integer variables, character variables etc. Absolute descriptors have the disadvantage that they are not always easy to find when they must be updated (e.g. if a segment is removed from memory, the segment address and present flag must be updated). An illustration can be found in the B6700 computer, which provides a special instruction for finding all descriptors for a particular segment. This process is not only time consuming, but also means that the stack segments can never be removed from main memory, as they may hold active descriptors. Segment list entries, unlike descriptors, are always held in a well known place and can be easily found.

2.2.3.2. Memory Allocation

In the paged memory scheme, memory is allocated in fixed size units. Consequently, memory allocation is relatively easy. When space is required, a page frame must be found in main memory which, in the

worst case, may mean that another page may need to be removed from memory. In the segmentation scheme, each segment is of a different size, and must occupy a contiguous area of main memory. Moreover, it must either be totally loaded into memory, or totally absent. This may mean that a large segment cannot ever be loaded into memory because there is not enough space. (The B6700 uses a modified segmentation scheme and allows very large segments to be divided into a number of 'pages'. This will be discussed later). A number of allocation policies have been used to try and allocate space in the most efficient manner such as Best Fit, First Fit and the Buddy system (Knuth, 1978). Often these policies are augmented by a compaction scheme, in an attempt to waste as little main memory as possible. Compaction can also be used when no particular allocation policy is used. Space is simply used up sequentially until there is none left, and then the memory space is compacted.

2.2.3.2.1. Compaction

Often the main memory may undergo some data compaction to try to remove areas which are too small to be of any use. During this time, all the processes executing on the processor are stopped, and a special operating system routine packs all the segments together. This scheme has two important drawbacks. First, the compaction operation is expensive in time. It must be performed by the central processor, during which time no other process can run. Second, all the segment table entries must be updated to reflect the new segment addresses. While this operation is possible, it also is extremely expensive. Moreover, if the B6700 type of absolute descriptors are used, these must also be updated. Unfortunately, such descriptors are very difficult to locate, as they may be mixed with data variables.

2.2.3.2.2. External Fragmentation

Regardless of the allocation policy, unless memory compaction is used frequently (which would be far too expensive) a large amount of memory space will be wasted, because very few areas will be exactly the same size as the segments. This is called external fragmentation, and is also experienced in the linear memory model in which memory is allocated in variable size units. It has been shown that this space loss can add up to a significant proportion of the available main memory space

(Randell, 1969). External fragmentation is, however, often less serious than the internal fragmentation found in paged memories.

2.2.3.2.3. Segment Replacement

If sufficient space in main memory cannot be found for a segment, an already loaded segment may need to be removed. This operation is similar to the removing a page from a paged main memory. It is possible to apply the same kind of algorithms used in the paged model, such as Random, Least Recently Used, etc. However, in the segmented scheme the segment which is removed from memory must leave sufficient space to hold the new segment. Consequently, algorithms such as Least Recently Used can only be applied to those segments which are large enough (or to groups of contiguous segments).

2.2.3.2.4. Dynamic Segments

One form of segment which complicates the task of memory management is the dynamic segment. These segments are initially allocated a fixed amount of space, like all other segments. However, during the lifetime of the segment it may grow in size. Examples of such segments include stacks, queues, lists and heaps. Space for dynamic segments may be allocated in two ways. First, extra space may be found contiguously in main memory, which may mean that a segment must be removed. Unfortunately, this may not always be possible. Second, the segment may need to be copied from its current place in memory, to a new contiguous area large enough to hold the entire segment. This is an expensive operation, and is avoided if possible. If the data structure has embedded link pointers, such as in a heap, and if the pointers are absolute memory addresses, then contiguous space need not be allocated. However, this organization is impractical because it complicates memory management significantly, as these pointers must be updated when the memory is rearranged.

2.2.3.3. Protection

Because it is impossible for user programs to modify segments which are not addressed by the process segment list, users may be protected from corruption by other users. To ensure that the segment list entries only point to the correct segments, the list itself (and any descriptor)

is protected from being modified by a user program. Also, a program can be protected from inadvertently modifying its own code or constant data by setting the access rights for each segment to prevent modification. In the paging model information is randomly packed into pages of memory. Because there is no logical link between the access rights of objects in a page and the access rights of a page (unless this has been specifically arranged by the compiler), pages form the wrong unit of protection. Segments, however, are used to represent logical objects, and thus form the correct unit of protection. They also form the best unit of sharing.

2.2.3.4. Sharing

An important feature of the segmentation scheme is that many users may share access to a single segment. Again, unlike pages, segments are used to hold individual objects. Two users may wish to share access to an object, such as an array, but may not wish to share all of the objects held in a page of memory, unless the page holds only one object. A number of different schemes have been devised which allow programs to share segments. The simplest implementation is achieved by loading the same memory base and limit addresses into more than one segment list. Thus, any process with the same segment list entry will automatically address the correct segment, regardless of the segment number chosen. Furthermore, each segment list entry may use different access rights to address the segment. This simple implementation causes two problems when a code procedure is shared amongst a number of user processes, and is illustrated by the example shown in Figure 2.13 (Fabry, 1974), where the code addresses a shared subroutine and a process-own data segment.

First, it is not clear how the shared program should be coded to allow each process, with a different segment list structure, to refer to the same object (e.g. the subroutine). Process 1 should use a 'call segment 2' whereas process 2 should use a 'call segment 1'. However, since the code is shared it must contain the same call operand in each process. Second, it is not clear how the main program should be coded so that the segment numbers which it has assigned to objects do not conflict with those used for different objects within the separately compiled subroutine. Thus, the segment number used for the data segment must not be the same as the segment number used for the subroutine. A

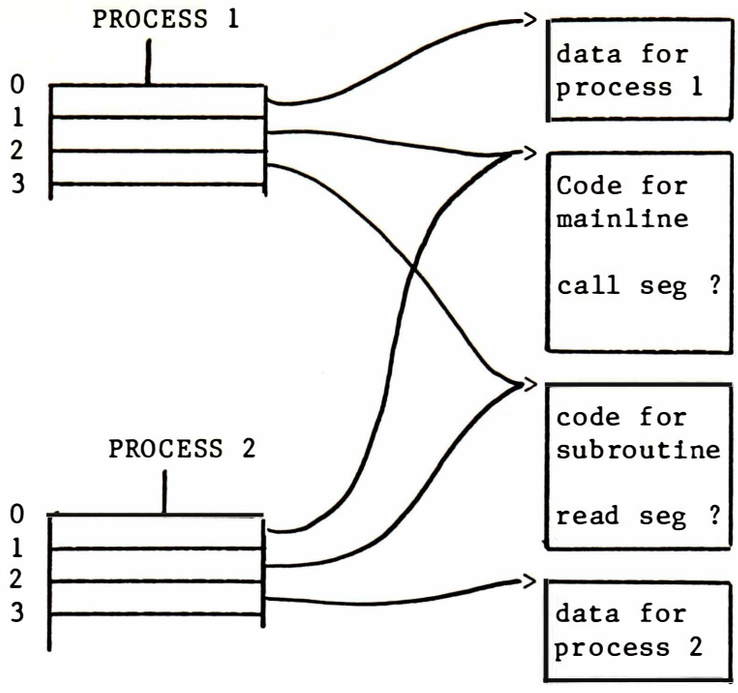


Figure 2.13 - shared segments

number of different solutions have been used, which we now describe.

2.2.3.4.1. Uniform Addressing

The most obvious solution is the uniform addressing scheme, which has been successfully used in the B6700 family of computers. The scheme demands that all code, mainline and subroutines, are compiled at the same time. Because of this all images of shared segment references (e.g. instructions) will have the same segment number, regardless of the process in which the reference occurs. Thus a segment which is shared will be known by the same segment number. Also, since the segment numbers for the mainline and the subroutine are assigned at the same time, there will be no conflict. The scheme can be implemented by maintaining a list of segment numbers at compile time. When a new segment reference is discovered, a new segment number is assigned.

The scheme has two main disadvantages. First, it is not always convenient to compile all the subroutines together with the mainline, especially if subroutine libraries are used. Second, when a segment is removed from memory, all of the segment lists which address the segment

must be altered. This operation may be extremely expensive. Another proposal, the indirect evaluation scheme, attempts to remove these two problems.

2.2.3.4.2. Indirect Evaluation

The indirect evaluation scheme solves the two sharing problems by means of linkage segments, which are used to dynamically translate the segment numbers used within a program (either the mainline or a subroutine) into the segment numbers held in the process segment list. An example is shown in Figure 2.14.

Each process retains the segment list used in the uniform scheme. However, additional linkage segments are associated with each code segment. These linkage tables then translate the segment numbers in the code into those required by the process segment list. Each linkage segment is located by an entry in the process segment list, and a processor register is used to address the current linkage segment. While the process is executing within the mainline, the linkage segment number associated with the mainline code is loaded into the processor register. Any reference to a segment is first translated into a process segment number, via the linkage segment. When the process enters the subroutine, the processor register is altered to point to the new linkage segment. Any segment references are then translated by a different linkage segment. The only exception to these translation rules is when parameters are addressed from a subroutine. In this case the process segment numbers are used.

The indirect evaluation scheme solves the sharing problems in two ways. First, each subroutine (and the mainline) is associated with its own linkage segment. Thus, the segment numbers used in the mainline may be the same as those used in the subroutine, and still address different segments. Second, the process segment lists may be ordered in any way, provided that the linkage segments for a shared code segment map the code segment numbers onto those of the process list correctly. Thus, different processes may share the same code segment even though the structure of their process segment lists is different. The linkage segments for each of the processes can be ordered to correct the segment numbers. This solution is effective because it maps a program onto a

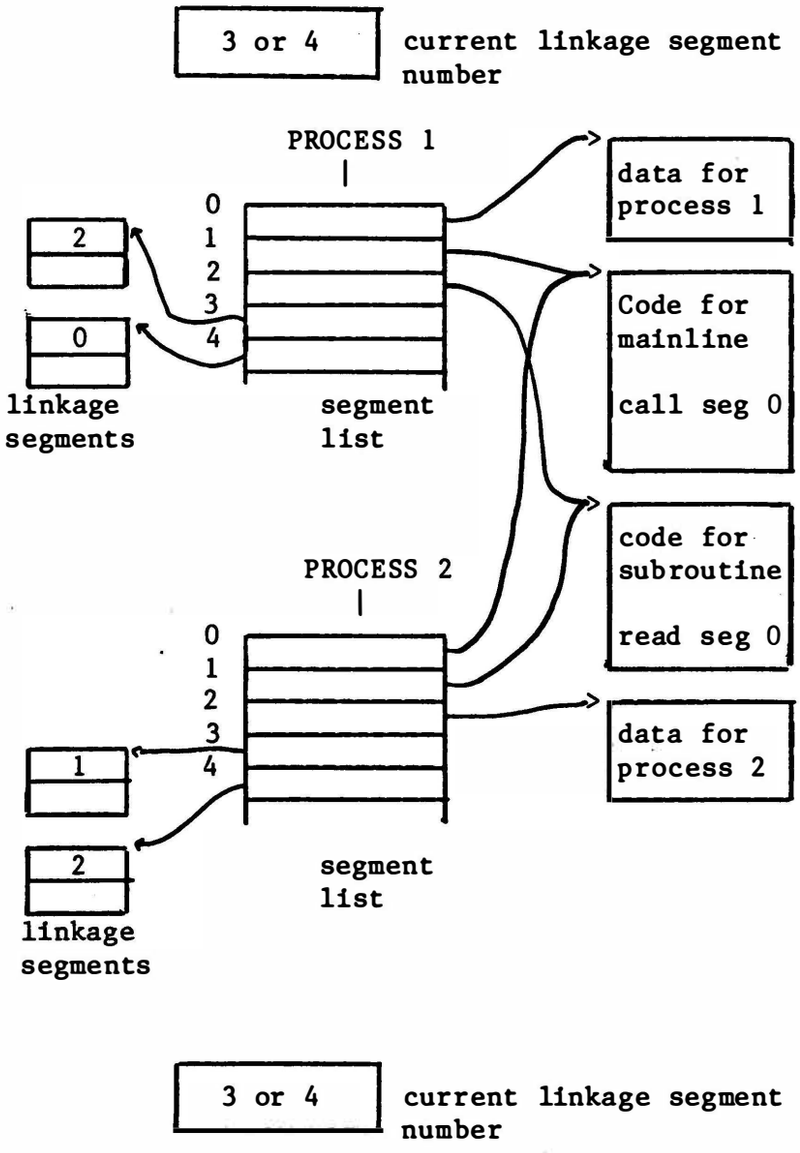


Figure 2.14 - the indirect evaluation scheme

process.

Indirect evaluation has some problems. First, both the linkage segment and the process segment list must be consulted on every memory reference. Second, the scheme does not solve the problem of searching all process segment lists for references to a segment when it is removed from memory. Third, free standing data structures do not have a linkage segment. These segments may, however, contain embedded pointers to other segments and like programs may be shared between processes. Thus, these segments suffer all of the naming problems experienced with program

segments. The only sensible solution to this problem is to not allow free standing data structures to address other segments without a code body, as proposed by Parnas (1972) and others (Wirth, 1977; Liskov and Zilles, 1974). In spite of its problems, the indirect evaluation scheme is used by MULTICS (Organick, 1972). An optimization of the indirect evaluation scheme is the multiple segment list scheme.

2.2.3.4.3. Multiple Segment Lists

The multiple segment scheme, shown in Figure 2.15, associates a segment list with each mainline and subroutine. A processor register is used to point to the current segment list, depending on whether the mainline is executing or one of the subroutines. Any reference made within a routine is translated into a main memory address via the segment list associated with that routine. When a subroutine is entered the processor register is modified. The scheme differs from the indirect evaluation scheme by removing the process segment list. Segment addresses are translated directly by the segment list associated with the code routine, rather than via a central process list. Thus, it is more efficient than the indirect scheme.

Whilst the removal of the process segment list may improve the speed of the system, it also destroys the mechanism for passing parameters. One solution to this problem is that each subroutine call creates entries for the parameters in the subroutine's segment list (Evans and LeClerc, 1967). Recursive calls are only allowed if multiple copies of the new segment list can be created, an expensive operation. An alternative solution addresses parameters via descriptors, rather than via the segment list, which may be held on the process stack (as in the B6700).

2.2.4. Segmented and Paged Memories

The segmented and paged memory scheme combines the segmented memory model and the paged memory organization with the aim of gaining both the logical advantages of segmentation, and the memory management advantages of paging. In this scheme, the user program addresses a set of segments. However, in distinction from the segmentation scheme, each segment is composed of a number of pages. Thus, while the user program perceives a number of variable length segments, the operating system can allocate

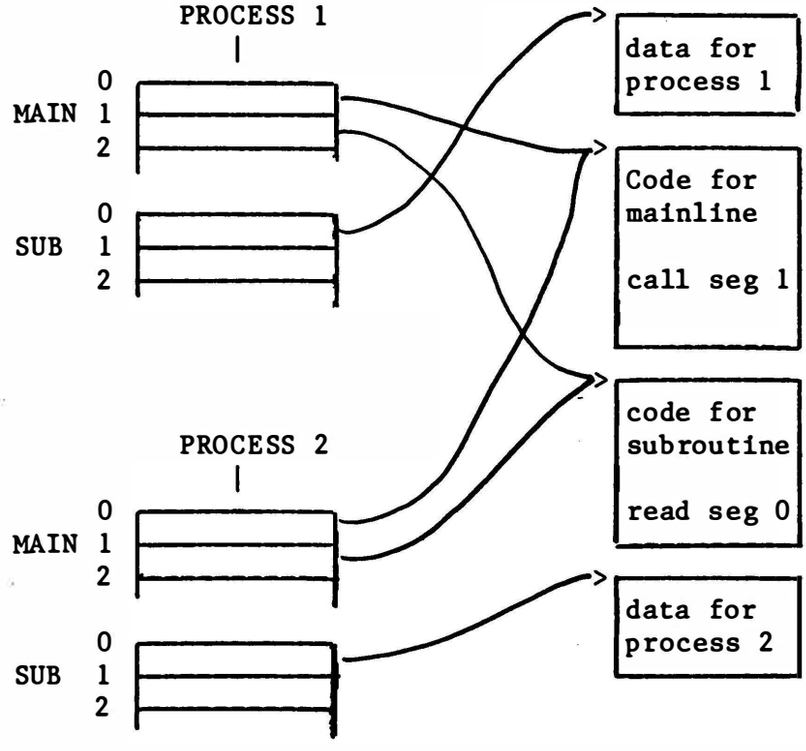


Figure 2.15 - multiple segment lists

main memory in units of fixed size pages. It also has the advantage that large segments can be addressed without the need to load the entire segment into memory. Only those pages which are being referenced need be loaded. If any other pages are accessed, a page fault is generated and the pages can be loaded from secondary memory.

The processor addresses are now composed of three fields: a segment number (within the process), a page number within the segment, and an offset within the page. This address, as shown in Figure 2.16, is then translated into a main memory address before the reference can proceed.

2.2.4.1. Address Translation

In the most widely used segmented and paged scheme, address translation is performed by a combination of segment lists and page tables, as shown in Figure 2.17. Unlike the purely segmented scheme, the segment list entries hold the main memory address of a page table

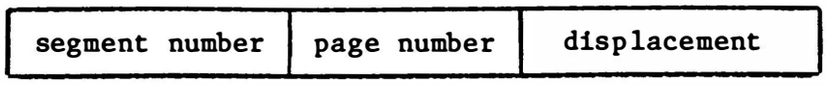


Figure 2.16 - a paged and segmented address

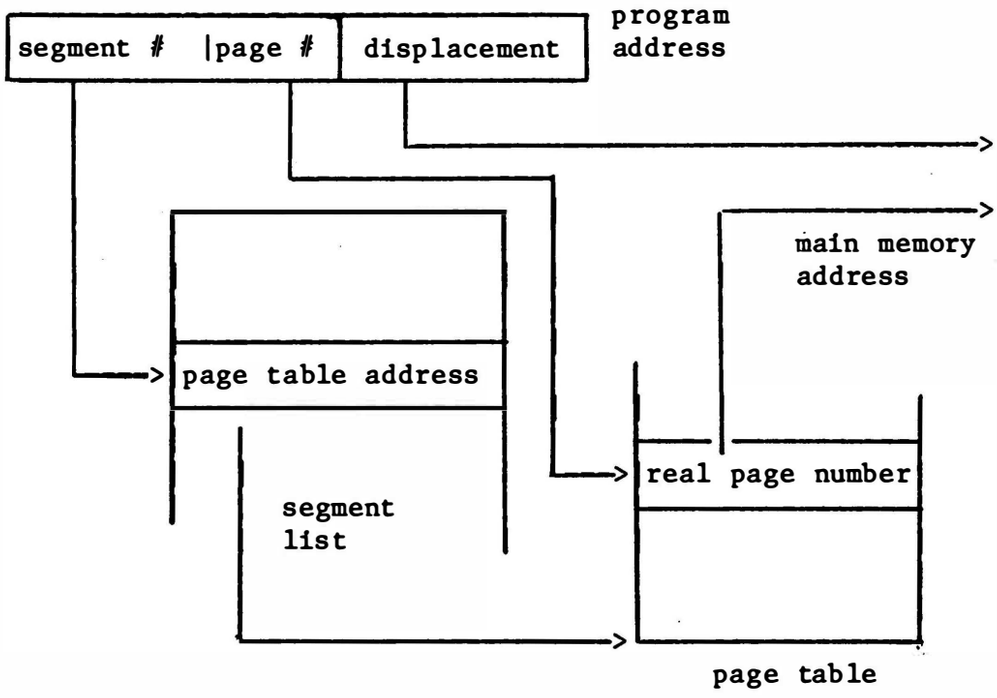


Figure 2.17 - paged and segmented address translation

for the segment. The page table contains entries which hold the main memory page frame numbers of the pages within the segment. Along with the address of the page table, each segment list entry also holds the page table size (i.e. the segment size), an absent flag (which is set if the page table for the segment is not in memory) and the access rights of the segment. Thus, segmentation is still used as the unit of protection. The page table entries hold the main memory page number and an absent/present flag. Segments are paged in the same way as address spaces in a purely paged system. Because this technique uses two tables before main memory can be addressed, special hardware is often used to speed up the translation (such as a cache memory, as described in the next chapter).

Whilst the B6700 is classified as a purely segmented machine, it does allow very large segments to be divided into a number of fixed length pages. Address translation for large segments is accomplished in a manner similar to that described for the paged and segmented memory model, except that the processor views a large segment as a collection of smaller segments, each of which is referenced by a different segment descriptor. It is then the responsibility of the compiler to generate code to address segments via lists of descriptors. However, unlike the paged and segmented model described above, the use of paging in the B6700 is not uniform and therefore offers the operating system no assistance with the management of the virtual memory.

2.2.4.2. Protection

Since the paged and segmented addressing scheme provides a process with a segment list in the same way as the segmented model, it inherits the same protection properties as the segmented scheme. Segments are used as the logical unit of protection. Other processes cannot address segments for which they do not have segment list entries. Moreover, segments still have access rights associated with them as in the purely segmented scheme. The introduction of paging only has an effect on memory management.

2.2.4.3. Sharing

Segments may be shared between processes in this scheme by placing the same segment list entry in more than one segment list. In this way, more than one process has access to the same page table. This arrangement is superior to the purely segmented scheme, because when various pages of a segment are removed from memory, only the one page table entry needs to be updated. In the segmentation scheme every segment list which addresses the segment must be updated if the segment is removed from main memory. However, this simple implementation has some problems. First, if a segment is deleted, or totally removed from memory, all of the segment list entries must still be updated. Second, if the page table for the segment is moved in memory, all of the segment list entries for that segment must be updated. Some systems (such as the ICL2900 series (Keedy, 1977)) have solved these problems by introducing an extra level of indirection between the segment list and the page

tables of shared segments. In this scheme, each segment list entry for a shared segment points to an entry in a global segment list, which in turn points to the correct page table. Thus, if the segment is deleted, or the page table is relocated, only the global segment list is updated. The disadvantages of this solution are that it increases the number of translation operations required to map segment addresses onto main memory addresses, and space must be allocated for the global segment list.

The paged and segmented memory scheme inherits the same problems for addressing segments from shared code segments as the segmentation model. Accordingly, the same solutions may be applied.

2.2.4.4. Memory Allocation

Whilst the paged and segmented model inherits the advantages and disadvantages of segmentation from the user programs viewpoint, it also inherits the memory management advantages and disadvantages of the paged model. As in the paged scheme, memory is allocated in fixed size units, thus large segments do not require contiguous areas of main memory. The problem of external fragmentation, experienced in purely segmented memories, is also removed. However, the internal fragmentation of the paged memory scheme becomes far more serious. Rather than wasting half a page of memory per address space, as in the paged scheme, the paged and segmented model wastes half a page per segment. If the segments are small in size, as experiments have shown to be a common occurrence (Batson and Brundage, 1977), then this can waste a substantial amount of memory. This effect can be diminished by using a very small page size. Unfortunately, this increases the size of the page tables significantly, posing memory management problems for these tables (such as finding space) and possibly creating more page faults.

A few solutions to this problem have been proposed. The MULTICS designers originally suggested that the processor could support two page sizes, one of 64 words and one of 1024 words. Because of the problems of maintaining two different types of page tables, this scheme was never implemented. Randell (1969) proposes a scheme in which segments may still be divided into pages, but memory is allocated in smaller fixed size units (of powers of two) called quanta. The scheme uses

conventional page and segment tables except that the page table entries contain a main memory quantum number rather than a page frame number. This quantum number is then added to the within page displacement to produce a main memory address. Large segments are composed of pages (and a number of quanta for the last page) and small segments are composed purely of a number of quanta. Moreover, since small segments only require one page table entry, this relocation information is held in the segment list rather than in a page table. This organization, called partitioned segmentation, causes much less fragmentation than the segmented or paged and segmented schemes. Also, since a number of small segments are packed into one page, the cost of transferring small segments between main and secondary memory is reduced. However, because memory is allocated in variable size units (even though they are units of quanta, which are powers of two), memory management becomes increasingly difficult, and may even become as awkward as in the purely segmented scheme. Consequently, neither of these solutions satisfactorily solves the small segment problems experienced in a segmented and paged memory.

2.3. Conclusion

The aim of this chapter was to determine the extent to which the conventional memory management schemes fulfil the needs of computer programs. We demonstrated that programs can be divided into three classes, each of which places different kinds of requests on the information system.

From this examination we have determined that the segmentation scheme has by far the most advantages, because of its logical properties. These properties are lacking in the linear and paged memory organizations. User programs can be divided into segments of memory, allowing logical structures to be protected and shared between users. The major disadvantage of the segmentation scheme is that it complicates the task of the operating system, because memory is allocated in variable size units. The paged and segmented scheme attempts to solve these problems by simplifying memory management, at the cost of internal fragmentation. The few proposed solutions to this problem appear to be ineffective. Later in the thesis we shall reconsider this problem, and make use of another solution.

Now that we have discussed the logical structure of the conventional memory management models, we can describe the hardware which is commonly used to implement these schemes.

3. Computer Memory Hardware

3.1. Introduction

In Chapter 2 we examined the conventional memory management models, but did not consider the hardware necessary to implement these structures. This chapter provides a summary of the current memory technology, and shows how these conventional memory organizations can be implemented. Later in this thesis, we develop a new memory organization, and show how it can be implemented with current technology.

3.2. Memory Building Blocks

A number of different memory building blocks are available, each suited to a particular environment. The following devices are discussed in detail in this chapter:

1 Registers
2 Fast addressable memories
3 Large storage devices
4 Associative memories
5 Cache memories

3.2.1. Registers

The most elementary form of computer storage device is the register, which is usually capable of retaining one word of information. The active components of a register are flip-flop devices, each capable of remembering the state of a binary digit presented at their inputs. Flip-flops may be concatenated to form a register of any length, as shown in Figure 3.1.

The input values are usually saved when the register is addressed, via a control line (or clock input). The output is always available, and only changes state when the control line is pulsed.

Registers were first implemented using thermionic valve devices, but most modern types are made of semiconductors. It is also possible to produce very fast registers, implemented with high speed logic.

Many processors use registers for holding temporary results, instruction operands, addresses and short term data items. Registers are not used for holding large amounts of data because of the high number of inputs and outputs required, and also because of their physical size.

3.2.2. Fast Addressable Memories

Large fast memory devices have been built using many different media and techniques. These memories, unlike registers, are capable of storing a large amount of data. Two classes of memory have been used, serial memories and random access memories. Both classes of memory have been used to hold the data and the instruction stream of a program.

3.2.2.1. Serial Devices

Serial memories are only capable of storing a sequential bit stream. To change the state of a bit, the unit must wait for the correct digit to appear at the output of the memory, change the bit, and restore the sequence.

A number of serial devices have been built, namely various delay line units. The units are not in common use now, not only because a processor cannot randomly extract data efficiently (because on average half of the memory must first be read), but also because they are volatile in nature.

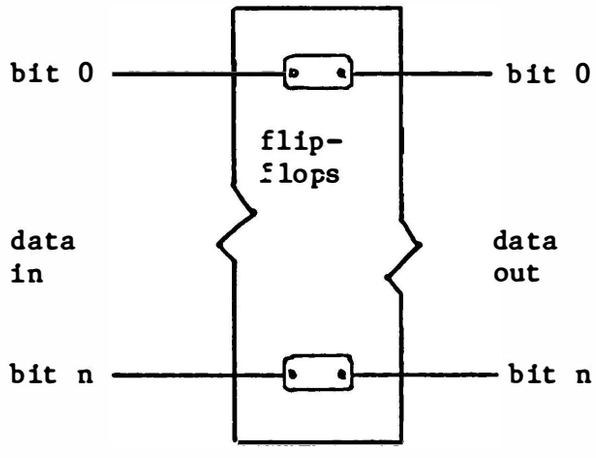


Figure 3.1 - a register device

The serial devices were superseded when the core memory was designed, in the early 1950s.

3.2.2.2. Random Access Devices

Random access devices differ from serial memories by allowing any bit in the memory to be addressed randomly, rather than by waiting for the bit to appear in the serial bit stream. The main effect of such devices is to improve the average speed at which the bits can be retrieved. The first random access memory to achieve wide-spread use was the core memory.

3.2.2.2.1. Core Memories

Core memories were introduced in the early 1950s, and are still in use in many modern computers. The scheme relies on the magnetic hysteresis properties of small ferrite cores. Each bit is saved in one core.

Large matrices of cores may be constructed, forming a core plane of many thousand bits. Core memories have two main advantages over serial devices. First, the core plane may be addressed by a row and column number. Thus, information may be retrieved in any order from store. Second, the cores are non-volatile, and withhold their magnetic polarization indefinitely.

Core memories are now being replaced by modern semiconductor memories, for two main reasons. First, each core plane requires a large amount of extra electronics to address and retrieve data. Second, the construction of each core plane is a complex and time consuming procedure. The modern semiconductor memories can be made with far less labour.

3.2.2.2.2. Modern Memory Devices

The introduction of micro electronic chips has enabled the construction of very highly populated memory devices, capable of saving many thousands of bits per chip. Two main classes of device are available, static and dynamic.

Static memories are constructed from many thousand flip-flops, and can thus hold many thousands of bits of information. The flip-flops are

individually referenced via an address word and, like core memories, may be randomly accessed. Current static memories can provide a retrieval time as low as 1 nanosecond and up to 2 microseconds.

Dynamic memories are also capable of storing many bits of information. However, unlike static memories, each bit is saved in a capacitive device, rather than a flip-flop. Each capacitor can only retain the data for a fixed length of time, and thus is dynamically refreshed. Because their internal construction is more compact, dynamic memories can achieve a higher bit density than static memories. Like static memories, dynamic memories may be randomly addressed.

Both static and dynamic memories suffer the drawback that they are volatile, and without power they lose their information.

3.2.3. Large Storage Devices

The addressable memories described are invariably too small and far too expensive to hold large amounts of information for any length of time. In addition, with the exception of core memories, they are volatile, and are thus unsuitable for long term storage of data.

Large data bases are thus held in large permanent memories, such as magnetic tape, disk and drum. The retrieval times, and storage capacities, vary across these different media. However, they are usually too slow to use as the computational memory of a processor.

This chapter is concerned with the hardware implementations of main computational memories, thus disk, drum and tape memories will not be considered further.

3.2.4. Associative Memories

The most common form of computational memory is accessed via an address word, which acts as a direct key identifying a data cell, shown in Figure 3.2. Most addressable memories are usually constructed from random access devices, though serial memories could be used. Another, less commonly used form of memory, is the 'associative' or 'content addressable' memory. This memory is capable of retrieving data via the content of the cell, rather than by the address of the cell.

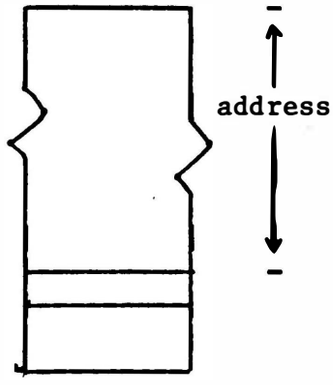


Figure 3.2 - an addressable memory

Typically, each cell in the memory is divided into a key field and a data field, as shown in Figure 3.3. When the memory is addressed, all key fields are compared to the search key. Any cells which have the same key field as the search key (or bear some relationship with the key, such as less than or greater than) are read, and the data retrieved, or updated. If more than one match occurs, the memory must include some multiple resolution logic to extract each response individually.

The use of such a memory may not be immediately apparent, and is best illustrated by example. Consider a table of surnames and residential addresses, each surname corresponding to an address. The surname can be loaded into the key area of an associative memory; the

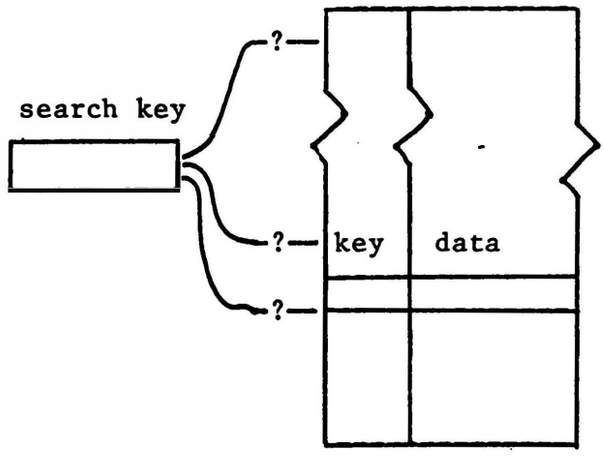


Figure 3.3 - an associative memory

residential address into the data area. It is then obvious that the memory can find all occurrences of a given surname, and provide the corresponding residential addresses.

Associative memories are extremely useful in improving the performance of central processors, and this will be discussed in more detail later. We will now describe some of the more common implementation techniques used to construct associative memories.

3.2.4.1. True Content Addressable Memories

The implementation chosen for a content addressable memory (CAM) varies depending on how the memory is to be used. In many cases, the CAM must perform very high speed association and retrieval; in such cases a true parallel CAM is used.

For implementation reasons, a distinction is made between the key field and the data field of a cell, as shown in Figure 3.4. Since the data field is not addressed by association, it may be held in a separate word addressable memory. When a key field match is found, the appropriate data field may be read and/or updated.

In a true parallel CAM, each bit of the key word is simultaneously

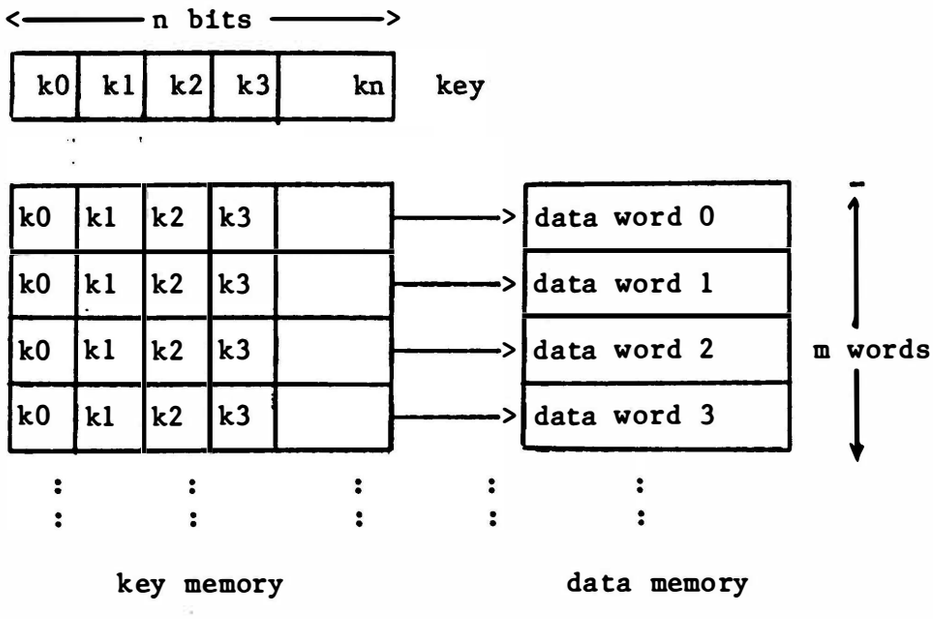


Figure 3.4 - a content addressable memory

compared with each bit of the key fields of all the words of the CAM. Any key field which indicates a match (i.e. the keys are the same, or have a relationship to each other) is flagged, and the appropriate data cell retrieved. The worst case retrieval time is clearly the sum of the time to compare two keys and the time to read the data memory. This time remains the same regardless of the number of cells in the memory.

The actual implementation of a true parallel CAM is, unfortunately, complex. Each bit of a cell must not only hold the bits of the key, but also contain a comparator. Thus, in a memory of m words, and a key size of n bits, a total of m * n bits of storage must be provided, and m * n comparator devices.

The high cost of each cell places severe restrictions on the ultimate size of the associative memory. Consequently, most true parallel associative memories are quite small in size. Other techniques are available when larger, but slower, associative memories are required.

3.2.4.2. Linear Scan - Word serial - Bit parallel

If the speed at which the data is retrieved from the CAM is unimportant, the memory may be scanned sequentially, rather than all cells testing their keys on parallel, as shown in Figure 3.5. In this scheme, the key memory is replaced by a fast addressable memory, of m rows and n columns. When a key is addressed, each row of the memory is read sequentially, and the n bits of the cell are compared to the n bits of the search key. If any cell indicates a match condition, the data memory may be addressed in the same manner as the true parallel CAM.

The hardware required for this scheme is simpler than the true parallel CAM. Rather then m * n comparators, only n are required. Because the key memory is word addressable, and only a small number of comparators are required, quite large word serial memories may be constructed. The worst case retrieval time of these CAMs is the time taken to read and compare all of the keys in the key memory, plus the read time of the data memory.

This device reads each word serially, and compares all bits of the key in parallel. Another CAM structure reads the words in parallel and compares the key bits serially.

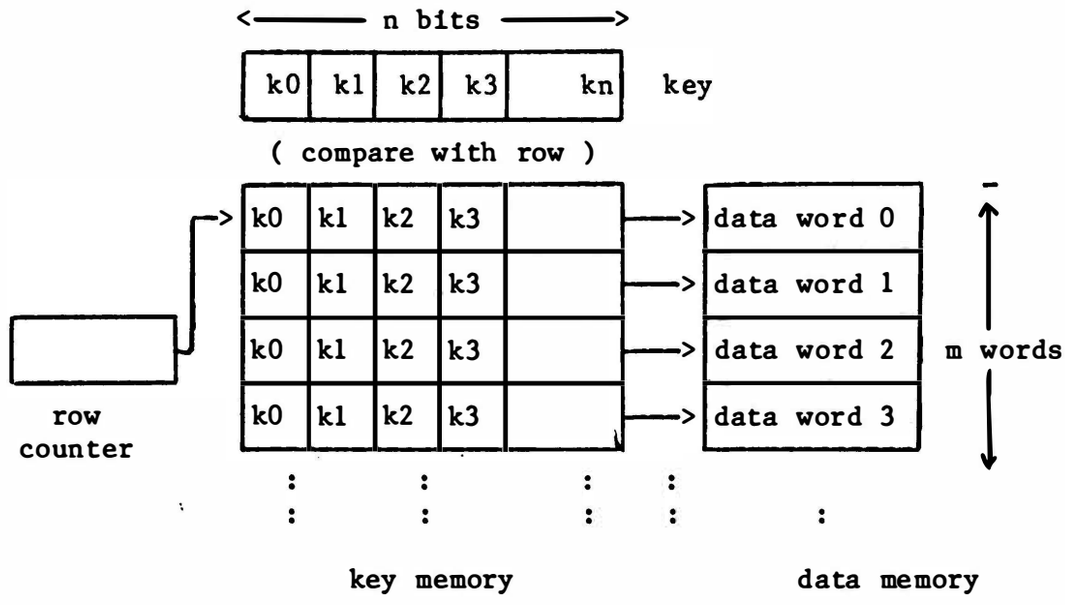


Figure 3.5 - a word serial - bit parallel CAM

3.2.4.3. Linear Scan - Word parallel - Bit serial

An alternative organisation may be devised which is faster than the word serial CAM, but slower and less expensive than a true parallel CAM. The word parallel system, shown in Figure 3.6 is the logical inversion of the serial scheme. In this method, the same bits of all key words are compared in parallel. Each key field is written serially into the memory, rather than in parallel, and is saved down a column rather than across a row.

When a key is addressed, each row is read sequentially. After each read, all m bits are compared to the corresponding bit in the search key. At each stage, a match for a column is saved. If, after the entire n rows have been read, a column matched for every row, then the data word can be retrieved from the data memory.

This scheme uses m comparators rather than n and, given that n is usually less than m, is usually more expensive to implement than the word serial memory. However, only n reads are required to match all keys, rather than m. Providing that there are more keys to be compared than bits in a key, this approach is faster than the serial scheme.

The word parallel scheme poses an important problem. Whilst it is capable of executing faster retrievals than the word serial memory, a key field can only be updated serially, requiring n write operations. It is also likely that all other m columns will be forced to execute an update cycle (as all will share a write signal), even though they are not being modified.

A modification of this scheme allows bit parallel access for write cycles and word parallel access for associative retrievals.

3.2.4.4. Skew Addressing

In this scheme, shown in Figure 3.7, a key word is held diagonally in the store, rather than being confined to a particular row or column. The address supplied to each column of the memory is skewed, or offset, by one relative to the next column. The memory still possesses the property that each row holds the same bit of each key field, and thus by sequentially scanning the rows, the same associative search used in the bit serial scheme may be used. However, when a key is updated, each bit is held in a different column, and thus all bits of the key may be written in parallel.

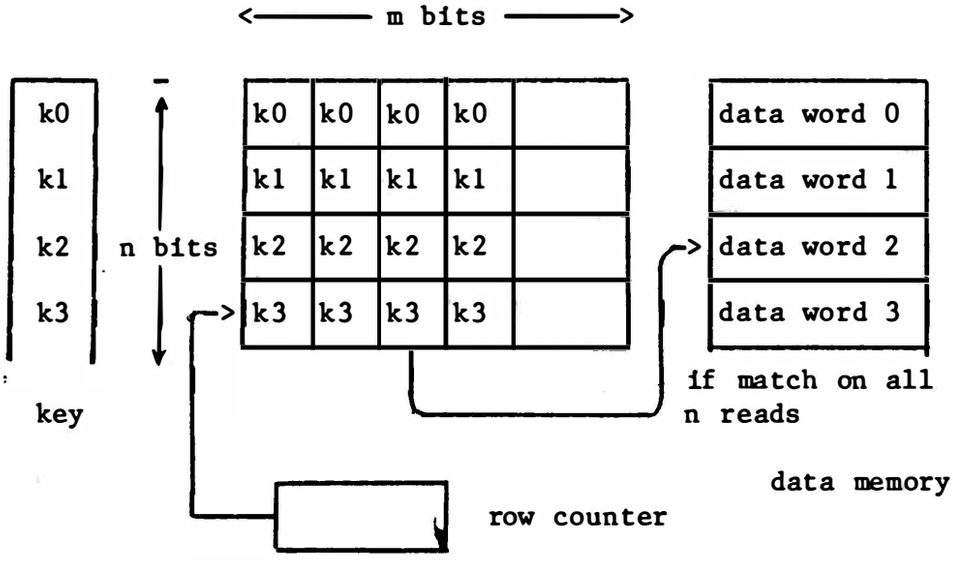


Figure 3.6 - a word parallel - bit serial CAM

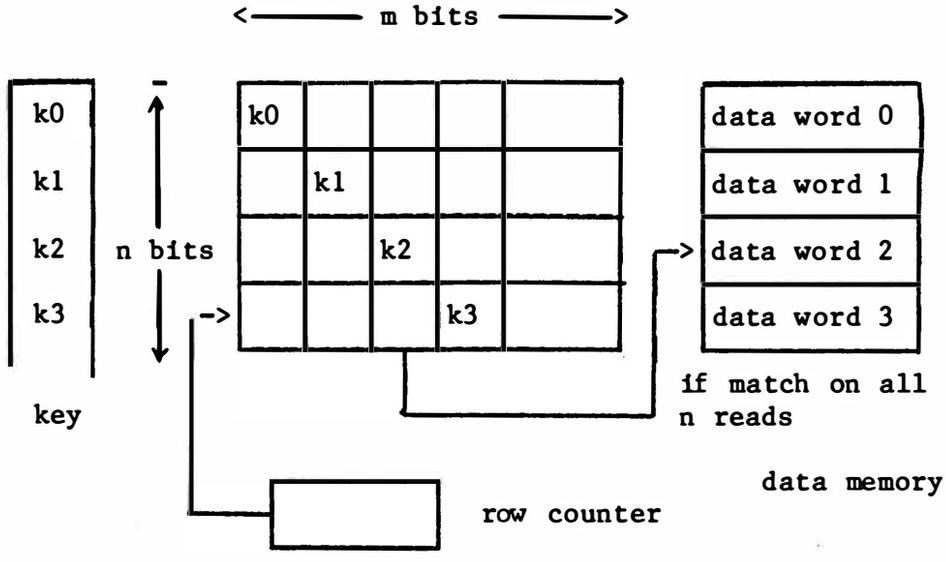


Figure 3.7 - skew addressing

3.2.4.5. Other Searching Algorithms

It is theoretically possible to implement other searching strategies in an associative memory, such as a binary searches, tree searches and hashing techniques. However, very few of these algorithms have actually been implemented in hardware, and will not be described here. A hashing algorithm is used to implement a large associative memory later in this thesis. A full description of this unit may be found in Chapter 7.

3.2.5. Cache Memories

Most processors are connected to their memory units via an address bus and a data bus, as shown in Figure 3.8. When a memory read or write request is initiated, the CPU must wait until the memory has completed a memory cycle which, depending on the main memory speed, may be in the order of micro seconds.

Substantial speed savings may be experienced by placing a small, very fast, associative memory between the processor and the memory. Thus a CAM is used to retain copies of the most frequently used memory locations. The scheme relies on some address locality; once a location has been referenced it is likely to be used again. Thus, once a location

is used, a copy of the data is placed into the cache. When a reference to the same location is made in the future, the cache copy may be used rather than the slower main store.

The key used in these associative memories is the main memory address. The data field of the CAM is used to hold a copy of the data. Cache memories can offer excellent speed improvements, as described in (Strecker, 1978).

3.2.5.1. Memory Write Operations

When the central processor requests a write operation, two different write algorithms can be used. First, the data can be updated in both the cache location (if it is present in the cache) and the main store. This protocol, called 'write through', has many advantages, as discussed in (Kohonen, 1978; 1980). The second alternative is to only update the location in the cache. The main store location is only modified when the variable leaves the cache memory, in which case the correct value is written to memory.

Whilst this solution avoids unnecessary memory write operations, it suffers from two problems. First, special hardware must detect when a location leaves the cache, and write the data back to main store. Second, two different copies of the same location exist, which complicates the sharing of variables in a multi-processor environment.

In addition, the 'write through' approach may be implemented so that it is no slower than the cache only write solution, by overlapping the main memory write with the next processor operation. Consequently, the former solution is usually implemented.

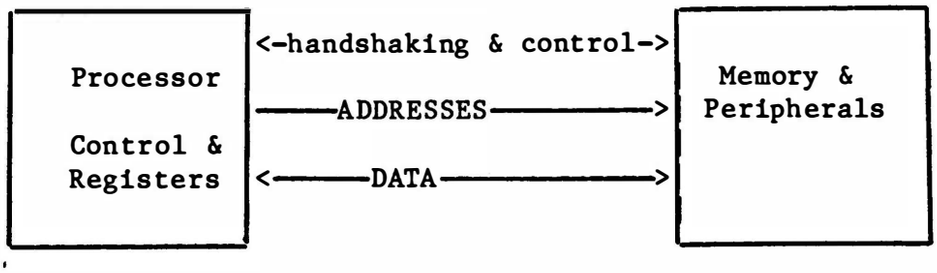


Figure 3.8 - a typical processor configuration

3.2.5.2. Inserting and Deleting Items

An important consideration affecting the efficiency of a cache memory is which words to insert into the cache and, when space is required, which words to remove.

Most systems use a simple, but effective, demand insertion protocol. When a word is fetched from main store, it is automatically copied into the cache. The likelihood that the word will be addressed again is quite high, making it an ideal choice to insert in the cache.

Two different cases may arise when a word is to be inserted. First, if there is sufficient free space, then any free location may be used. If, however, there is no free space, then a word must be removed. Various deletion algorithms are possible, the most common being Random or Least Recently Used.

Random selects a location at random from the cache. This algorithm is easy to implement, especially in hardware, and chooses a word quickly. Unfortunately, it often removes the wrong word. Least Recently Used selects the word which has remained unused for the longest time. Whilst harder to implement than Random, and even though it is slower in choosing a cell, this algorithm tends to choose a better location to remove. In spite of these advantages, Random is usually used because of the ease of implementation, and the speed of operation.

3.2.5.3. Data Caches and Address Translation Caches

The data cache memories discussed so far are capable of retaining copies of the most used words of memory in high speed memory. It is often desirable to retain entries from the address translation tables, to improve the speed at which addresses can be translated. In many cases, a separate address translation cache is provided, and this will be discussed in the latter part of this chapter. An important distinction between these two is that an address translation cache is not usually modified. Thus, the data from these caches need not usually be written back to main memory.

3.2.5.4. Implementing Cache Memories

An important attribute of a cache memory is high speed. A slow cache may offer no speed improvement over the main memory. Many

different types of cache memory have been implemented, the most popular being the freely loadable cache, the direct mapping cache and the set associative cache.

3.2.5.4.1. The Freely Loadable Cache

The most obvious implementation technique for building a cache memory is by using a true parallel CAM. The cache may be searched very quickly (as all comparisons are performed at the same time), and address-data pairs may be loaded into any position within the CAM.

Unfortunately, true parallel CAMs are often too small to hold enough main memory data locations. Consequently, other techniques have been developed especially for use in cache memories.

3.2.5.4.2. Direct Mapping

A direct mapping cache is constructed from very high speed addressable memory. Each cell holds both the key field and the data field, as shown in Figure 3.9. The key field is used to hold the main memory address and the data field holds the memory data at that address.

The index position of a cell in the cache is calculated from the key value, and is often extracted from the least significant bits of the key (although a randomising function may be applied). When the processor requests a memory cycle, the cell contents at the calculated index value are retrieved. The key field is then compared to the main memory address and, if equal, the data field is returned to the

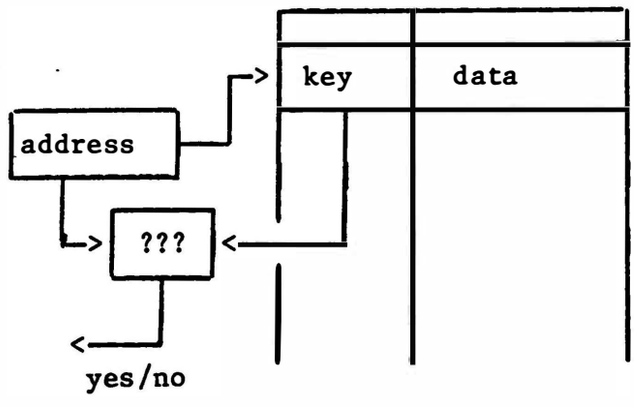


Figure 3.9 - direct mapping cache

processor. If a write operation was requested the field in the cache is updated.

This scheme is only limited in speed by the cache retrieval time and the delay time of the comparator. Both of these may be very fast, producing a CAM many times faster than main memory. Unlike the freely loadable cache, this organization requires only a one word comparator, and is thus inexpensive to produce.

The most important criticism of the scheme is that it in not truly associative; two addresses which have the same low order bits will 'home' to the same cache cell, and cannot be held in the cache at the same time.

This restriction is not serious for a number of reasons. First, the choice of low order bits for a randomising function guarantee that successive main memory locations can be held in the cache. Sequential addressing is particularly common when instructions are fetched, thus the 'clashing' is not a serious drawback. Second, the cache only needs to hold a high percentage of the words being constantly addressed, not all of them. If a word is not held in the cache, either because there was no room, or it clashed with another address, then the processor may still continue by using the main memory. Provided that only a small percentage of commonly used locations are absent from the memory, the cache will still give a significant speed improvement. Third, the effect of addresses homing to the same cell may be diminished by increasing the size of the cache itself. Thus more bits from the memory address are used to calculate the index value, decreasing the likelihood of a clash.

Unlike the true parallel CAM, an address can only be inserted into one cell of a direct mapping cache. Thus, if an address is to be inserted into the cache, it must be placed in the correct index position, possibly removing an address. The same choice of insertion algorithms is not available for the direct mapping cache; the address is either inserted in the correct cell, or not at all.

An important optimization of this style of cache which reduces the size of the memory, is to only save the bits of the key not used to calculate the index value. This can save many bits of high speed memory.

Another modification of the direct mapping cache is the set associative cache.

3.2.5.4.3. The Set Associate Cache

If more than one address homes to the same cell of a direct mapping cache, then only one address entry can be saved. All others must reside in main memory. In a set associative cache this limit is extended to 2 or 4 such entries. A collection of addresses which home to the same cell is called a 'set'

The scheme is implemented by providing more than one direct mapping cache, each placed side by side, as shown in Figure 3.10. Thus, two units allow two different addresses to be held at the same index position. When a third address is to be saved at an index position, a choice is made of which address entry to discard. Either Random or Least Recently Used may be applied, however, because of the ease of implementation random is usually chosen.

Two way and four way set associative memories offer extremely good performance and are often used for both data and address translation caches (Strecker, 1978).

3.3. Implementing Memory Organizations

This section examines the memory constructs which have been used to implement the structures described in Chapter 2, namely linear memories,

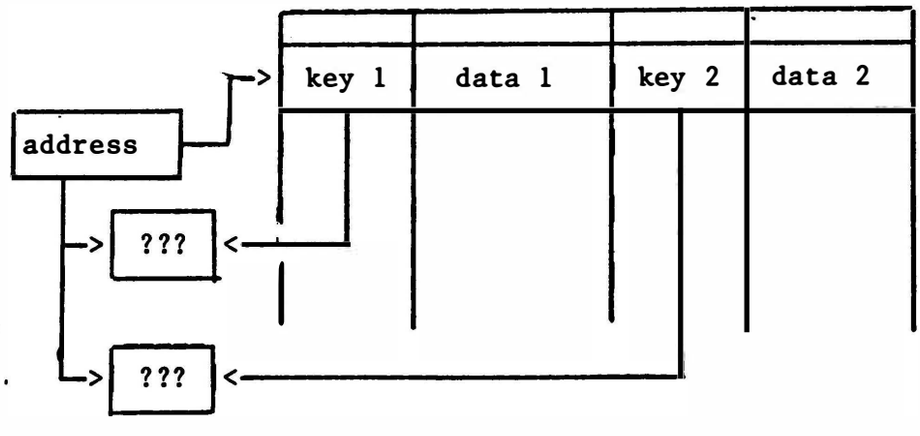


Figure 3.10 - a set associative memory

paged memories, segmented memories and paged-segmented memories.

3.3.1. Linear Memory Schemes

Linear memory schemes have been implemented with varying degrees of hardware support. In all of these, the logical view of memory is the same as the physical view, and no address distortion is introduced (excluding simple linear offsets). In the simplest case very little hardware is required to allow the processor to address memory.

3.3.1.1. Basic Scheme

In the most basic linear memory scheme, shown in Figure 3.8, each address generated by the processor is transferred directly to the memory unit. The memory itself is constructed from the addressable memory described in section 3.2.2.

The inclusion of a fence register (see Chapter 2) has no effect on the actual addresses. The processor addresses are simply compared to the value of the fence register and, if an address if detected below the fence value, an interrupt is caused. The relocation scheme (see Chapter 2.2.1.2) requires slightly more complex address modification hardware.

3.3.1.2. Relocation Registers

When a system is fitted with base and limit registers, the addresses produced by the processor are different from those accepted by the memory unit, as shown in Figure 3.11.

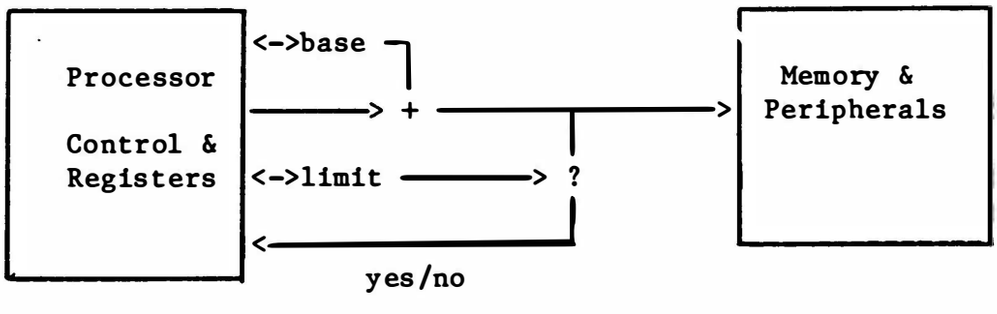


Figure 3.11 - relocation hardware

Whilst the linearity of the address space is preserved, a program may be relocated in the main memory. Each address from the processor is augmented by the contents of a base register, and the result may be validated against the contents of a limit register. The address is modified by a fast adder, which has little effect upon the total memory access time.

When the processor addresses are distorted, such as in a paged memory, much more hardware must be provided.

3.3.2. The Paged Memory Scheme

The paged memory scheme was first implemented on the Atlas computer (Fotheringham, 1961; Kilburn, Edwards, Lanigan and Sumner, 1962) in 1958. Since that time many different hardware implementations have been designed and built. As stated in Chapter 2, these implementations fall into four classes:

(1) Processors with small virtual address spaces
(2) Processors with small main memories
(3) Processors with large main and virtual memories
(4) Processors with very large virtual memories

Each class has different properties, which influence the techniques used to implement them.

3.3.2.1. Small Virtual Address Spaces

This class includes processors in which the addressing range of an individual process is quite small, even though the combined space of all processes may be large.

Each time the processor generates a virtual address, the address is mapped onto the physical memory. The mapping operation is usually performed by a page table, often held in main memory itself. If the address space is small enough, it is possible to place the contents of this page table in a special fast mapping memory, placed between the processor and the memory, as shown in Figure 3.12.

Each time a virtual address is generated, the page number is extracted from the rest of the address (leaving a within page displacement), and used as an index into the mapping memory. The entry

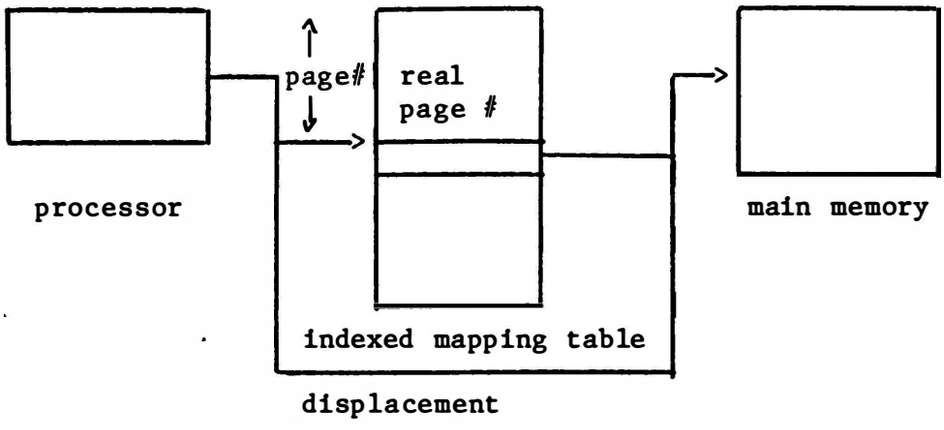


Figure 3.12 - a small paged memory

addressed in the mapping table is used to hold the main store page number, a set of access rights, and a valid flag. The main store page number is concatenated with the page displacement to form a main memory address. The other fields in the table are used to validate the type of access, and to detect page faults.

The time required to translate an address in this method is the time taken to read the mapping memory, which can be made a small fraction of the main memory cycle time. Using this scheme the overhead incurred by address modification is extremely low.

Depending upon the number of processes concurrently executing on the processor, it may be possible to dedicate a separate address translation memory for each process, as found in (Hagan, 1977). However, if the processor executes many processes, the contents of the mapping memory may be loaded from the page tables in main store when the process is scheduled for execution.

This method of address translation can only be used when the number of virtual pages in an address space is small, because one entry is required for every page of virtual space. The amount of main memory has little effect on the size of the table, only on the width of the individual mapping entries.

The mapping memory must usually be implemented from fast addressable memory. Consequently the cost of a mapping memory for a large address space becomes prohibitive. In addition, the cost of

swapping the contents of the address translator becomes too high. Thus, different techniques are used when the virtual address size becomes large.

3.3.2.2. Small Physical Memories

When the size of the main store is small (as it was on the Atlas computer) even though the virtual space may be large, a different address translation mechanism may be used. In these cases, the page tables contain many empty entries. By inverting the structure of the page tables and, rather than indexing the tables by virtual page number, using the physical page number as a key, the size of the tables may be reduced dramatically, as shown in Figure 3.13. Address translation may be accomplished by associatively matching the virtual page number with the cells of the mapping memory. Because the address mapping must be fast, the mapping memory must be constructed from a true parallel CAM.

Thus, an associative memory large enough to hold the page table entries for the entire main store can be used to translate all of the processor virtual addresses. Any virtual address which cannot be found in the memory, is not present in main store, and should cause a page fault.

In most systems the virtual addresses are not unique between processes. Thus, the associative memory must either be cleared and

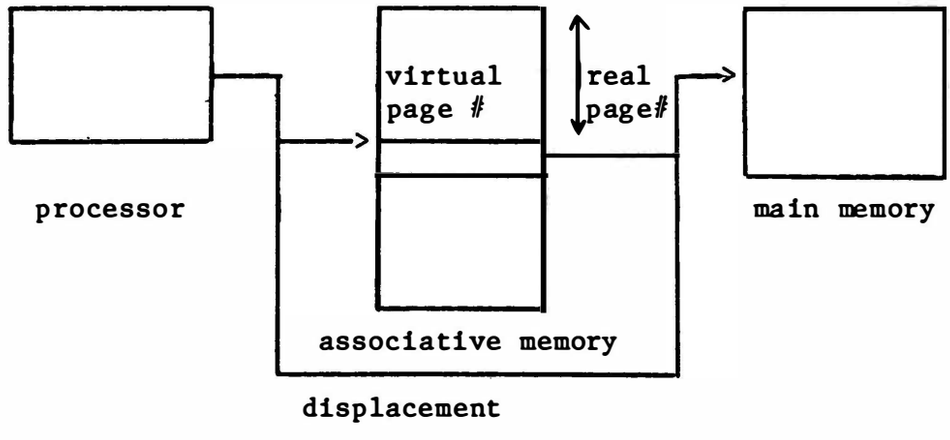


Figure 3.13 - a small main memory

reloaded on each context change, or the virtual address must contain the process number.

Unfortunately, most true parallel CAMs are quite small, and this technique may only be used when the number of main memory pages is small. The scheme is comparatively insensitive to the size of the virtual address (unlike the previous method), as this only affects the width of the memory entries and the comparators. The next technique is used when both the virtual address size and the main memory address size are large.

3.3.2.3. Large Virtual and Physical Memories

Traditionally, very little hardware support has been available for the translation of large virtual addresses. Clearly, a table indexed on virtual page number cannot be provided, because of the size of the virtual address space. Likewise, a truly associative parallel CAM cannot be provided because of the size of the main memory. Thus, the address translation in large memories is nearly always accomplished via tree structured page tables held in main memory. In some circumstances, these tables are so large that they are held in virtual memory, and are paged in and out of main memory like all other pages of virtual memory (Digital Equipment Corp., 1979), which causes many complications, as discussed in Chapter 2.

Because the cost of consulting these tables on every memory reference would be prohibitive, it is common to augment this mechanism with an associative memory, or address translation cache, capable of holding the most commonly used page table entries as shown in Figure 3.14.

When a virtual address is translated, the cache memory is first consulted. If the translation table entry is not found, then the page tables are searched. This entry may then be placed in the cache in order to assist future references to this page.

Because virtual addresses are not usually unique, the cache must be cleared on each process switch, and be allowed to reload itself when a process is started. Because the entries of the cache are not modified, there is no need to write the entries back to the page tables when the process is changed. Various forms of cache are utilized. Since the

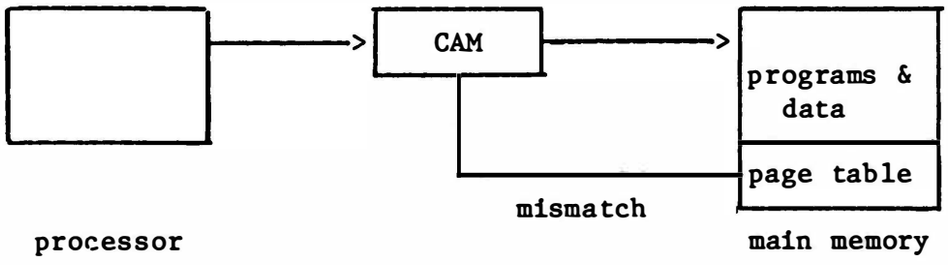


Figure 3.14

cache need not be fully associative (i.e. the page tables can always be read in the case of a cache miss), quite large set associative memories are often used for assisting this address translation.

Whilst this technique is effective for quite large virtual address spaces, the space required by the page tables is still considerable. Consequently, this scheme has not been used on a virtual address size above 32 bits (as in the VAX 11/780).

3.3.2.4. Very Large Virtual Spaces

When the virtual address becomes very large (for example 48 or 64 bits) conventional page tables are no longer an effective method of address translation, for a number of reasons. First, an enormous amount of space is required to hold the page tables. These page tables would certainly be held in virtual memory themselves. Second, because of the the size of the page tables, and the complexity of the retrieval algorithms, translating an address using the tables is a slow process. Even if the address translation cache provides a very high 'hit' rate, the small percentage of memory references which use the page tables will be so slow that the average memory reference time will fall dramatically. Third, as described in the last chapter, a page fault operation may generate a number of further page faults in an attempt to translate an address.

To avoid the problems associated with holding the page tables in virtual memory another technique may be used. In this method, the page tables are never used to translate main memory addresses, and are only used to find the location of a page in secondary memory. This approach

was used by the Atlas computer. All addresses for pages in main memory were translated by an associative memory, which held page table entries for every page of main store. Unfortunately, true parallel associative memories large enough to manage the large main memories now commonly in use, are not available.

One computer, MU6-G (Edwards, Knowles and Woods, 1980) has used a word serial associative technique to emulate a large CAM. However, because this method is so slow, the processor also uses an additional pseudo associative cache to achieve a respectable translation time. Another computer, the IBM System/38 (IBM, 1978, 1980; Houdek, Soltis and Hoffman, 1981) also uses an associative translation technique, which is described in Chapter 4.

3.3.3. Segmented Memories

Like large paged virtual memories, very little hardware support has been developed for segmented memories. The two conventional translation mechanisms are descriptors (as used in the B6700 family) and segment lists. Whilst it may be possible to provide special mapping memory for the segment lists or the descriptors, they are usually held in main store. These translation mechanisms are augmented by cache memories to improve their translation times.

Thus, when a process generates a segment address, the cache is searched for the segment relocation information. If it is not found, then the segment list (or descriptor) is used to translate the address, and the information is placed in the cache for future reference. Since segment numbers, like page numbers, are not usually unique between processes, it may be necessary to clear the cache on a process change. If additional indirection tables are used, the cache may also need to retain entries from these tables.

3.3.4. Segmented-Paged Memories

Address translation in segmented-paged virtual memories is similar to that of purely paged memories, except that both page and segment tables are usually used to translate virtual addresses into main store addresses. These tables can be augmented by a cache memory which holds the most frequently used page and segment table entries. The key used

for the associative search is the combined segment and page numbers. In all other respects, the address translation is the same as for the paged memories.

3.4. Conclusions

We have described the digital technology used in the construction of memory systems, and have shown how to implement the conventional addressing schemes. The next chapter describes a different method of addressing memory, called capability based addressing, and shows how schemes based in this technique are built.

4. Capability Based Addressing

In Chapters 2 and 3 we examined various conventional memory organizations, and compared the advantages and disadvantages of each.

The segmentation scheme appeared to offer many logical advantages. However, this addressing scheme usually applies only to segments of memory. The other objects which are addressed by a program, such as files, I-O devices and other programs, are addressed, shared, protected and synchronised by different mechanisms.

For example, file data is not retrieved in the same way as data from an array. Sharing a bounded buffer between processes is not implemented with the primitives used for sharing access to a file. Record access within files is often synchronised by 'record locks', whereas shared code and data may be synchronised by semaphores. Files are protected by directories, whereas code and data are protected by isolated address spaces, or protected domains.

A capability based addressing scheme (Dennis and Van Horn, 1966) attempts to extend the logical view of segmentation to addressable objects other than memory, and provides a uniform scheme for sharing and protecting these objects. This uniform approach has the advantage that it simplifies, and even removes, many of the mechanisms which are duplicated in most conventional computer systems. This chapter comprises four sections. The first discusses the properties of capabilities. The second concentrates on the objects that they address. The third section categorises the various implementations, and develops some general models. The fourth section examines some particular objects which can be addressed and protected by capabilities and the effects that they have on the models.

4.1. The Properties of Capabilities

A capability is a protected pointer which gives a program the ability to address an object. A capability is logically composed of two fields, <object name> and <access rights>. The name field holds the name of the object which the capability addresses. The access rights field describes the way in which the object may be addressed by that capability. Capabilities are normally regarded as possessing the following intrinsic properties:

- The object name should uniquely define the object. No two objects should ever have the same name. The name, which is typically encoded as a long integer, is assigned when the object is created, and is never reused.

- Possession of a capability allows a program to address the object. If a program does not possess a capability for an object, then no other mechanism allows the program to reference the object. This property provides a means of implementing the principle of least privilege and of enforcing a 'need to know' security policy.

- A capability describes how the object may be manipulated. The access rights may be used to restrict certain operations, whilst allowing others. A capability for a memory segment, for example, may contain access rights defining whether a segment may be read by a user and whether it may be modified.

- A capability should not be forgeable. The two fields within the capability contain sensitive information. If possession of a capability is the only means of accessing an object, then a user must be prevented from changing the name to point to another object, and from changing the access rights which define the way in which the object may be accessed.

- Object names should never be reused, even after the object has been destroyed. This property is important if capabilities for deleted objects are not reclaimed. If the names were reused, the capability for the old object could be used to address a new object assigned the same name. Some systems relax this rule by collecting all old capabilities.

- Capabilities facilitate easy sharing. An object may be shared amongst all users who possess capabilities for the object. The capability mechanism allows as few and as many users as necessary to share access to an object.

- Capabilities facilitate different views of an object. Because

each capability contains an access rights field of its own, different users may be given different views of the same object. Thus, one user may only be allowed to read a segment, whilst another may be allowed to write to the same segment.

Having enumerated the properties of capabilities, we shall now briefly consider the objects that they address.

4.2. Capabilities and Objects

Two different classes of objects are important, memory segments and other types of objects.

A memory segment is a logical collection of information held either in main memory or in secondary memory. Capability systems often differ from the conventional architectures discussed in Chapter 2 in that they use a common segmentation mechanism for storing both computational data and file data. In both cases each segment is addressed by a segment capability, which contains a unique name permanently associated with the segment.

The access rights field of a segment capability is typically used to indicate modes such as read access, write access or execute access. Some systems distinguish between two sorts of segment; one which holds data, and one which holds capabilities. These two properties are often distinguished in the list of allowable access rights. A segment with capability access is protected from arbitrary modification by users, thus guaranteeing the integrity of the capabilities which it contains.

Many other objects are traditionally addressed by special addressing mechanisms. For example, most systems provide separate input-output sub-systems to access objects such as card readers and printers. But these, and other high level abstractions such as files, stacks, queues, ports and user defined abstractions may also be addressed by a uniform capability mechanism. In this case instances of such objects are assigned unique names, which are then used within the capabilities which address them. The access rights field may be used to restrict certain operations on the objects in the same way as for memory segments. For example the holder of a capability for a port might be allowed to send messages via the port but not receive them via the same port.

This chapter concentrates on the mechanism used to address memory segments within a capability based addressing scheme. (The addressing of other high level objects by capabilities will be considered later.) The addressing of memory segments is particularly important because of the way in which segments are used. Each time an instruction is fetched from a code segment, the capability system must be used. Each time an instruction addresses a data item, the capability mechanism must again be used. Provided that an efficient and flexible implementation is found for addressing memory segments via capabilities, an efficient implementation for other objects should also be possible.

4.3. Implementing a Capability Addressing Scheme

This section considers two important areas related to the implemention a capability addressing scheme. The first is how to protect and use the non-forgeable capabilities. The second area is far more complex, and involves the techniques for implementing the capability model on real processors. Several existing systems are examined, and two general models are drawn.

4.3.1. Protecting and Using Capabilities

The storage of capabilities poses two implementation problems. First, as stated earlier, capabilities must be unforgeable. Thus, the mechanism used for storing and using capabilities must also protect them from corruption. Second, capabilities are quite long. It may be infeasible to use them directly as operands for instructions, and embed them within the instruction stream. Two solutions to these problems have been proposed, partitioning and tagging.

4.3.1.1. Partitioned Segments

In a partitioned machine two different sorts of segment are recognised, data segments and capability segments. A capability segment may only be used for holding capabilities, and may never be directly manipulated by data instructions. Special instructions are provided for manipulating capability segments which allow capabilities to be moved, created, deleted and copied to other segments.

The capability segment associated with an executing program is often called a C-list, shown in Figure 4.1. Instructions may address

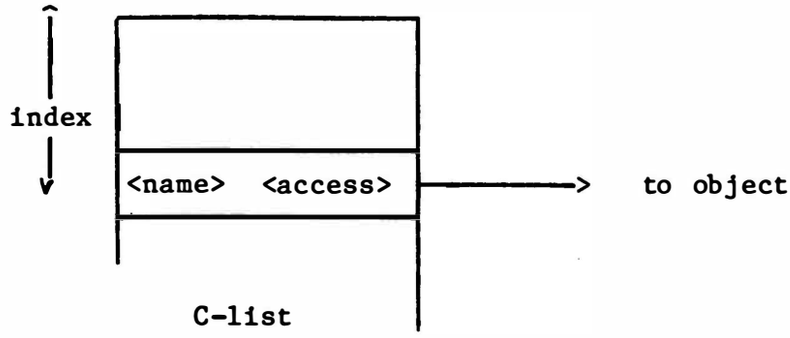


Figure 4.1 - a partitioned addressing scheme

objects by specifying the index value of the capability in the C-list. C-lists solve both of the problems described in 4.3.1. Because capabilities are held in a special protected segment, there is no way of modifying pointers, or creating illegal pointers. This protection system can be enforced by creating capabilities which point to the C-list with an access type of 'capability', rather than type 'data'. Thus, the capability mechanism can be used to protect C-lists themselves.

Moreover, the size of the index value is many times smaller than the name of the objects, and may be efficiently coded as an instruction operand.

Whilst the C-list may appear to be the same as the segment list discussed in Chapter 2, the capability addressing scheme does not inherit the linkage problems with shared code segments, as experienced in the segmentation scheme (Fabry, 1974). The main reason for this is that most capability systems associate a C-list with a code body. Thus, even if many processes use the same code, they all use the same C-list.

Partitioning has the overhead of an extra segment per addressing environment, but is still widely used. This scheme is used by Plessey 250 (England, 1972), Intel iAPX432 (Intel, 1981a, 1981b), CAL (Lampson and Sturgis, 1976), CAP (Needham, 1977; Wilkes and Needham, 1979) and HYDRA (Wulf, et. al. 1974; Wulf, Levin and Harbison, 1981). A slightly modified version of partitioning is used in StarOS (Gehringer and Chansler, 1981) (and conceptually Hydra). In this system each segment may be partitioned into a data portion and capability portion. Apart from reducing the need for an extra segment, the two schemes are

conceptually identical.

Wilkes (1980) proposes placing some of a program's capabilities in the same segment as the code, thus reducing the number of segments required. Because the code itself is protected from corruption, the capabilities cannot be illegally modified. In this scheme, care must be taken that the capabilities cannot be executed, which can be achieved by some form of fence register scheme. Apart from requiring extra protection hardware, this scheme does not remove all of the extra C-lists. Some capabilities must be modified and copied around the system, and others must be addressed as parameters to a program. Neither of these two types of capability can be held in the code segment, mainly because they are not necessarily owned by the code body. Thus, extra addressing mechanisms must still exist to cater for those capabilities not held in a code segment.

4.3.1.2. Tagging

An alternative solution to the problems raised in 4.3.1 is the tagged approach. In this scheme each word of store, or each structure in store, is assigned a tag field. This field defines which operations are allowed on the data, and is checked before the data item is used. Thus, integers may be tagged as type 'integer', and may not be used as operands for a floating-point instruction (Myers, 1978a, 1978b).

Similarly, capabilities may be tagged as type 'capability', which prevents them from being used as data items, or from being modified. Tagging capabilities solves the protection problem, but it does not reduce the operand size. The length of the effective address is governed by the mechanism which references the capability. Tagged capabilities, however, may be placed on the process stack, and addressed relative to the stack registers, reducing the operand size. The IBM System/38 (IBM, 1980) uses an additional addressing table to shorten the operand size.

Tagging has a number of advantages and disadvantages when used as a general data protection mechanism. These are discussed by Gehringer and Keedy (1982). It also possesses a number of disadvantages when specifically used to protect capabilities, some of which are relevant to this discussion.

First, it is sometimes necessary to garbage collect old capabilities. This task is more complex in a tagged architecture than in a partitioned organization. In a tagged system, capabilities will be distributed over various segments of store, and are thus harder to find than if they are all grouped together in a special segment (Wilkes and Needham, 1979). Second, in a tagged scheme, the type 'capability' is associated with the capability, rather than with the view of the capability. It is often necessary for system functions to manipulate capabilities as though they were data. A system which allows different views of a capability, such as the partitioned scheme, makes this objective easier to achieve. Third, the tagging approach only solves the problem of protecting capabilities. Another mechanism, which often duplicates many features of a capability scheme, must be used to shorten the operand size (such as in the IBM System/38).

Tagging has only been used in the IBM System/38 to protect capabilities, but has been proposed for use in many other systems (Myers, 1978a, 1978b; Myers and Buckingham, 1980; Bishop, 1977; Gehringer, 1979).

4.3.2. Names and Mapping Information

We have shown how capabilities may be used to address objects, and how they may be protected. This section examines how the unique name of an object can actually be used to address the object. Section 4.3.2.1 demonstrates the need for some mapping information in capability systems. Sections 4.3.2.2 and 4.3.2.3 describe the current implementations of the mapping mechanism, and section 4.3.2.4 describes the hardware required for an efficient implementation. Section 4.2.3.5 shows how the mapping information for memory segment objects may be extended and used to address abstract, or extended, types of objects.

4.3.2.1. The Need for Mapping

Like many conventional computer systems, most capability systems find it necessary to realize a number of discrete levels of addressing. This occurs because the logical name space, which holds all objects that have ever been created, will always be many times larger than the amount of real store which a system can provide (either main memory or secondary memory). Three levels of address are often visible, defining

three spaces: the name space, the virtual space and the real space.

The name space must be large enough to hold all objects that have ever existed, and all objects which will ever exist in the lifetime of the system.

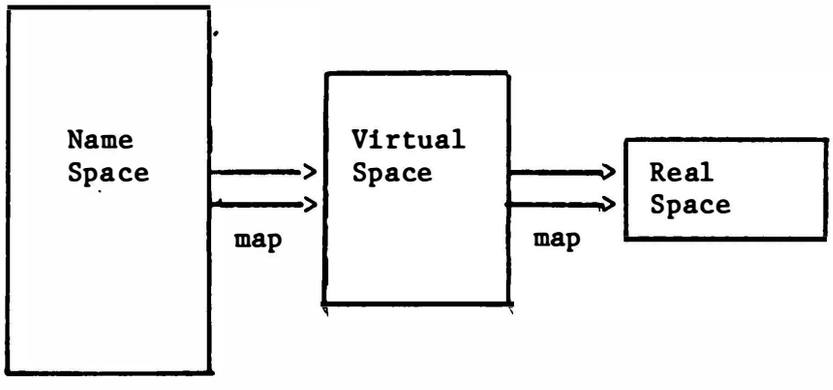
The virtual space is sometimes as large as the name space, but may instead define a smaller area, which is still larger than the amount of real memory.

The real space must be large enough to hold all current objects, and may consist of a combination of secondary memory (such as a disk) and fast main memory.

Unlike name space addresses, virtual and real addresses may be reused after objects have been destroyed, reducing their size considerably. The relationship between the three levels is shown in Figure 4.2. Because the sizes of the spaces are different, it is necessary to use a mapping function to translate addresses from one space to another.

4.3.2.1.1. Direct Mapping

In the direct mapping model, shown in Figure 4.3, a capability is used directly to address the object, without performing any name



size(name space) > = size(virtual space) >>> size(real space)

Figure 4.2 - the three addressing levels

conversion or translation. Such a model could be realized by one of two different methods.

First, the real store could be associatively addressed. In this scheme, the 'name' in a capability is recognised by the associative store, and thus allows the correct data to be addressed. Unfortunately, very large associative stores are not available, as mentioned in Chapter 3, and this technique is not possible.

Second, the capability could contain an extra field, holding the real address of an object. This technique is also unsuitable, because the task of memory management becomes extremely difficult. When objects are moved in store, all capabilities for the object must be updated; this is a time consuming operation. This problem is evident in the B6700 family of computers (Organick, 1973).

Thus, because of practical difficulties, the direct mapping model is never actually used in capability based systems.

4.3.2.1.2. One Level Translation

Because of the problems encountered in a direct mapping system, most capability based computers place a mapping function, or structure, between the capability and the object in real store, as shown in Figure 4.4.

In this scheme, the unique name found in the capability is translated, via a mapping structure, into a real memory address. Real memory addresses may be safely reused by modifying the mapping information in such a way that no old names ever map to the reused real store addresses.

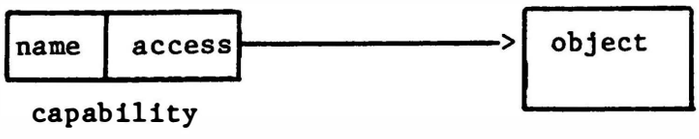


Figure 4.3 - the direct mapping model

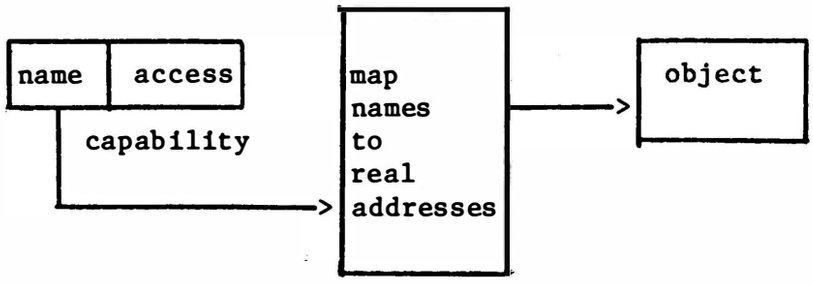


Figure 4.4 - one level translation

In such systems, the name space is the same size as the virtual space, and is usually very large. Systems which use this model are HYDRA, the Intel iAPX432, CAL, the Plessey 250 and CAP. All of these systems use the mapping structure to translate segment names into main store addresses, and none allow virtual addresses to be reused. Bishop (1977) proposes a scheme which belongs to this class, but uses a smaller name space than is necessary. He therefore allows names to be reused proposing a complex method of collecting old addresses to deleted objects. Unlike the other systems in this class, Bishop uses the mapping structure to translate virtual page addresses into main store page addresses. Segments are loaded into virtual pages without any further mapping.

4.3.2.1.3. Two Level Translation

If the name space is larger than the virtual space, then two levels of mapping are required between a capability and an object, as shown in Figure 4.5.

In the two level scheme names are first translated into virtual addresses, and then virtual addresses are translated into real addresses. Whilst names are very large, both virtual addresses and real addresses may be safely reused when virtual and real objects are destroyed. This model is employed in the IBM System/38 and in a proposal by Gligor (1978). Both of these systems use the second mapping structure to translate virtual page addresses into real page addresses.

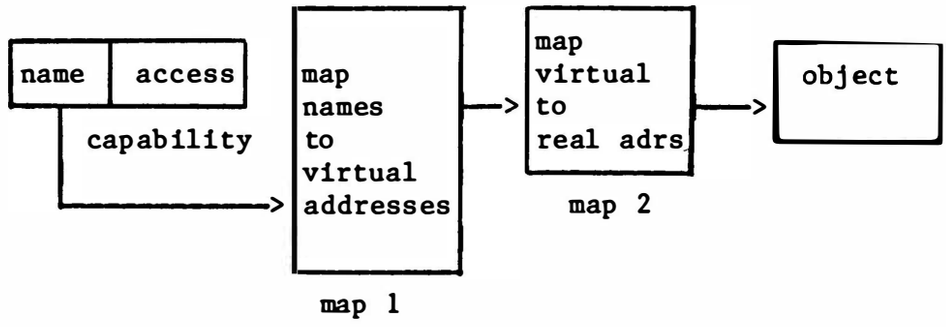


Figure 4.5 - two level translation

The two level scheme, whilst logically equivalent to the one level system, has some effect upon the implementation efficiency, and on the organization of the store. The next section examines the implementations, both used and proposed, for mapping names onto virtual addresses

4.3.2.2. Translating Names into Virtual Addresses

The task of translating the very large object names into smaller virtual addresses has been attempted on two systems, the IBM System/38, a real production computer, and in a system proposed by Gligor (1978).

Gligor's solution consists of two sections. The first involves an additional field in the capability, used to hold an index value. The second involves the use of a large mapping table which translates names into virtual addresses. This scheme is shown in Figure 4.6.

Each capability holds the name of the object, and also a shorter index value into the object mapping table. When a capability is used, the object mapping table is read, and the virtual address of the object determined. Entries in the object map are only reused when all old capabilities have been found and destroyed. This garbage collection operation only occurs when the object map overflows. Gligor proposes placing the map in virtual memory itself in order to prolong the time between garbage collections, because an object map in virtual space can afford to be longer than one held in main memory.

The IBM System/38 provides two different addressing mechanisms; one which uses large unique object names, and one which only uses virtual

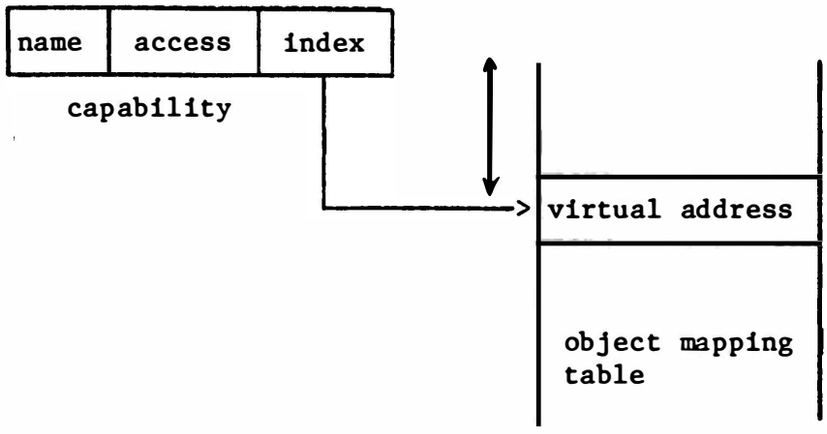


Figure 4.6 - Gligor's name translation

addresses. When the unique names are used, the names are mapped onto virtual addresses; however, when objects are referenced by their virtual address, this mapping process is not used. Thus, the IBM processor in many ways may not be considered a true capability processor.

Object addresses in the System/38 are 64 bits in length, whereas virtual addresses are only 48 bits. The virtual space is composed of paged segments. By allowing segment addresses to be reused, the 64 bit names are mapped onto the 48 bit virtual addresses of the System/38 hardware.

Segments are grouped into segment groups, each of 256 segments. As shown in Figure 4.7, up to 2^24 different segment groups may be formed from a 48 bit address.

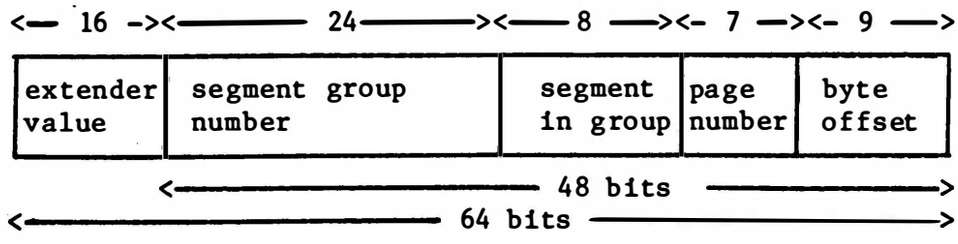


Figure 4.7 - the IBM System/38 address format

If no two segment groups of the same group number (but different extender value) are allowed to exist at the same time, then the extender value can be dropped, and the remaining 48 bits used to address the virtual store. A segment group can only be reused when all segments in the group have been deleted. When a segment group is reused, a new extender value is assigned, giving the object a unique 64 bit name. This mapping scheme, whilst simple, allows 64 bit names to be efficiently mapped onto 48 bit addresses. Unlike Gligor's scheme, no actual mapping table is required.

An obvious danger with reusing segment groups in this way is that capabilities for deleted objects may be kept, and later reused to address a new object with the same segment and group numbers as the old object. To prevent this problem, the extender value for a particular segment group is stored in the header of the group. When a segment within the group is addressed, the 16 bit extender from the capability is compared with the extender in the appropriate group header. If they are not the same, then the capability is for an old object, and the reference is aborted. This mapping technique has a number of attributes:

(i) No mapping table is required. The mapping information is distributed over the segments which are addressed.

(ii) The extender must be held in the group header. Whilst only two bytes long, the entire header page must be present in store when the group is addressed.

(iii) Only 2^24 different segment groups may exist at any time. This is such a large number that it is unlikely to be a restriction.

(iv) The group header must be read for every reference made to a segment. This overhead in incurred only when 64 bit names are translated into 48 bit addresses, which may be avoided much of the time.

Because the capability mechanism on the System/38 includes some inherent inefficiencies, another method of addressing objects is provided. In this method, 64 bit names are never used, and instructions can directly use 48 bit virtual addresses.

Segments may instead be addressed via the Operand Description Table (ODT) associated with a particular program. This table describes the type and size of each operand used in the program. The type information is used for validating that the data type is compatible with the instruction type. Many instructions use this field for performing automatic type conversions.

Operands are mapped onto the segmented memory via another table of the same size as the ODT, the Operand Mapping Table (OMT)(which conceptually can be regarded as an extension of the ODT). Each ODT entry corresponds to an OMT cell which holds the 48 bit virtual address of the object. Because capabilities are held in segments of store, they are also addressed via the OMT. This addressing mechanism is demonstrated by an example in Figure 4.8.

Because two different addressing mechanisms are present, one using unique 64 bit names, and the other using 48 bit reusable virtual addresses, care must be taken when unique names are generated. When an OMT address is used, the object is referenced without validating an extender value; thus the same segment group numbers must never be

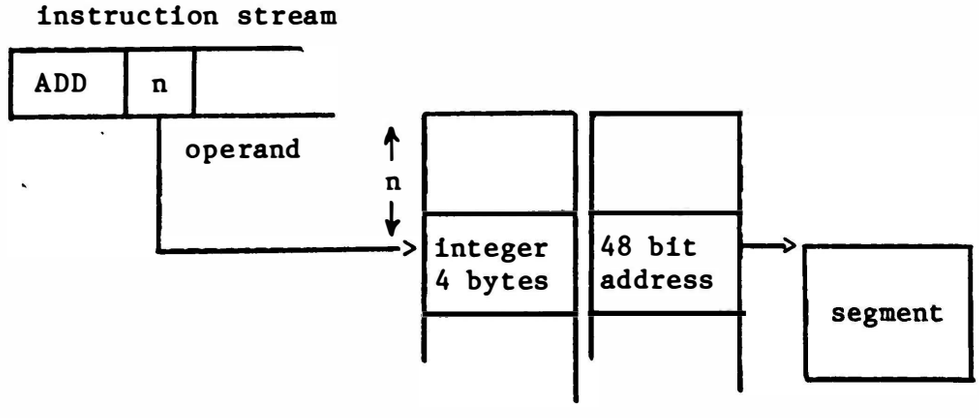


Figure 4.8 - the IBM addressing mechanism

allocated to a capability. Moreover, OMT entries must never be saved between program invocations.

This section has demonstrated two different methods of mapping names to virtual addresses. The next section examines how virtual addresses are mapped onto real addresses.

4.3.2.3. Translating Virtual Addresses into Real Addresses

Both the one level and the two level mapping models require translation of either names or virtual addresses into real secondary memory addresses or main store addresses. As we will discover later, secondary memory and main memory addresses are usually produced by different mechanisms; however, we will examine the main store addresses primarily.

Unlike name space translation, all of the virtual address translation systems use tables to map virtual addresses onto real addresses. Moreover, because these tables implement the virtual memory system, they cannot be easily placed in the virtual memory. Instead, they are placed in real memory, often at a 'well known' place.

Virtual address translation tables may be categorized into four classes, each with a different organization: linear lists, conventional page tables, reusable index tables and hash tables. This section will examine each organization, and explain how various capability based computers use the tables.

4.3.2.3.1. Linear Lists

A common method of mapping addresses in conventional computers is to provide a mapping table, indexed by part of the virtual address. Each cell can contain the real memory address corresponding to each virtual address. Such tables are used in small paged processors, such as MONADS I (Hagan, 1977) and in segmented machines.

The technique becomes impractical in capability systems because the virtual addresses, or names, become far too large. In spite of early documentation for the Intel iAPX 432, which suggests that this technique can be used, no capability systems appear to have used a linear mapping table.

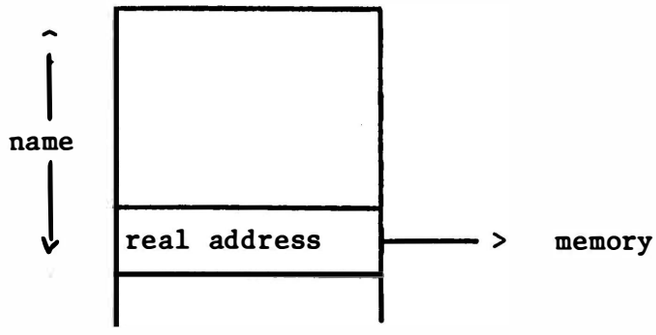


Figure 4.9 - a linear mapping table

4.3.2.3.2. Conventional Page Tables

Bishop (1977) proposes the use of conventional page tables to translate the virtual addresses of a capability system into real page addresses. The virtual address size suggested is in the range of 40 to 50 bits, and the virtual space is composed of a number of variable size paged areas.

Whilst a processor was not built, Bishop's paper design relies on normal page tables to translate virtual addresses. Unfortunately, the page table space for a 50 bit virtual address would be in the order of 2^40 page table entries. Because of the space required, these would need to be placed in virtual space. A conventional system which supports its paged store in this way is the VAX 11/780 (Digital Equipment Corp. 1979). The VAX, however, only uses a 32 bit address, which is 2^18 times smaller than that of Bishop's processor.

Bishop also suggests that an associative memory (similar in nature to that of MULTICS (Organick, 1972)), with a hit rate of only 50 %, would significantly speed up the address translation. (This figure is calculated in the thesis (Bishop, 1977)) However, if as many as half of the addresses requiring translation used the page tables, which are many orders of magnitude slower than an associative memory, then the effective memory cycle time would be excessively slow.

Thus, in spite of Bishop's proposal, it would appear that conventional page tables are not a suitable method for translating very large virtual addresses. It is notable that Bishop's processor design

was not built.

4.3.2.3.3. Reusable Index Tables

A number of capability processors have used a small indexed table, and a modified capability format, to translate unique names into real addresses. The capability is altered to include an index value field, as shown in Figure 4.10.

The indexed table contains entries which hold the base address, the size, and the possible the resident status (whether the object is in memory or not) of the memory segment. The Plessey 250, Chicago Magic Number Computer (Shepherd, 1968; Yngve, 1968), and CAP-3 (Wilkes and Needham, 1979) use such a technique. In these processors, the base address from the central mapping table is added to an offset within the segment to form a main store address. The offset is validated against the segment size, and the mode of access is compared to the access rights. Violation of the size or access constraints causes an exception condition.

In these three systems, the size and organization of the central mapping table may have an effect on the efficiency of the system. If this table is used to hold the mapping information for all of the objects, then it will become too large to hold in main store. It must, therefore, only hold the most active capabilities.

In addition, unless a garbage collection scheme is built which collects and invalidates capabilities for objects which no longer exist,

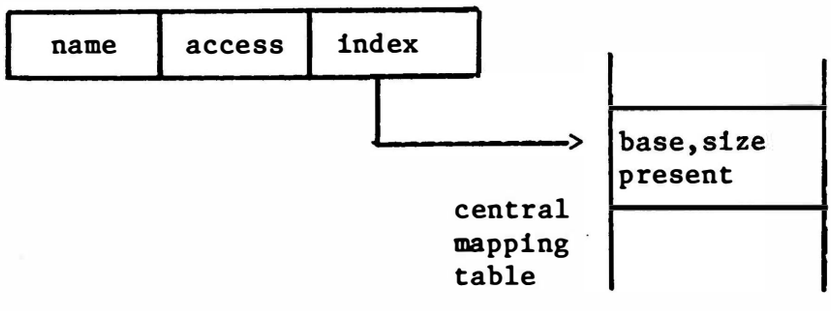


Figure 4.10 - a directly indexed table

or appear in the main store, the mapping table will continue to grow in size. Entries can only be safely reused when all capabilities which address them are destroyed or invalidated. To avoid this garbage collection, the CAL system places the name of the object in the central mapping table as well as in the capability. In this way entries may be safely reused. When a central mapping table entry is used, the name field from the capability is compared to the name field in the entry. If they are not equal, then the object being referenced either no longer exists or does not use that entry any more, and the slot in the table has been reused. Thus, unlike the tables in the Plessey system, entries can be reclaimed without having to collect all old capabilities first.

The reusable index tables are effective providing that there is a method of reusing cells. The next section describes a hashed table organization.

4.3.2.3.4. Hash Tables

Hash tables of various forms have also been used to translate names and virtual addresses into real addresses. Fabry suggests that a hash table, indexed by a hashed form of an object name could be used to hold the mapping information about the object (Fabry, 1974). Hydra uses a number of hash tables to translate names into real addresses, and the IBM System/38 uses a hash table to map virtual pages onto real pages.

The System/38 uses a hash table to translate 48 bit virtual addresses into main store addresses, shown in Figure 4.11. As far as the virtual address translator is concerned, the address is composed of a 39 bit page number, and a 9 bit offset. The 39 bit page number is then mapped onto a main store page number. Because of the size of this address, conventional page and segment tables are inappropriate.

The mechanism used is only responsible for translating addresses in which the page is actually in main memory. If a page is not in store, then other tables are consulted. This approach was chosen in the Atlas (Fotheringham, 1961) address translator, although the Atlas translation mechanism was a true parallel content addressable memory (CAM).

The System/38 maintains a hash table in main memory, which is indexed by a hashed version of the virtual page number. The address translator microcode follows overflow chains until either the address,

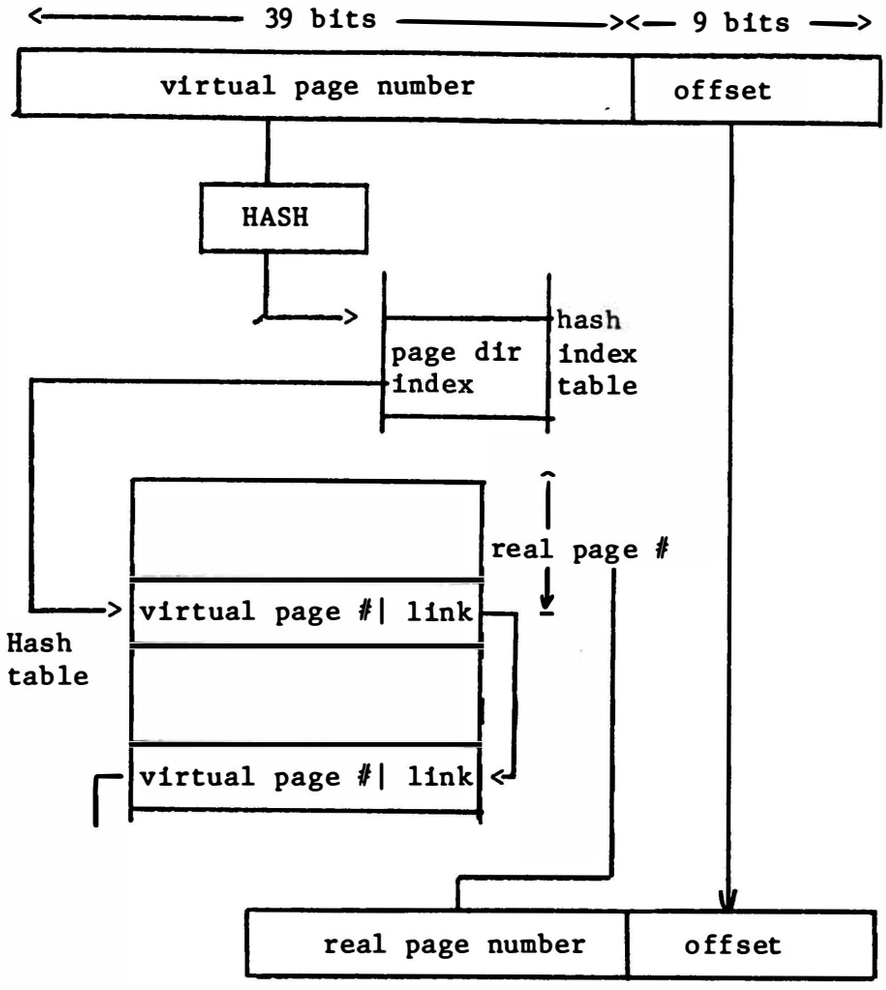


Figure 4.11 - the IBM address translator

or an end of chain, is found. The latter causes a page fault. If the page is found, then a translated page number is formed and placed in a lookaside buffer. IBM expect that an average of 2.25 main store accesses (IBM, 1978), plus the time spent in microcode, on top of every memory reference which uses the address translator. (Some references use real addresses via a special register, and thus avoid this overhead).

Myers proposes the use of a hash table to translate object names into memory addresses (Myers and Buckingham, 1980). However, rather than incorporating an overflow strategy, Myers uses an allocation scheme which does not allow any two name to hash to the same cell. Any name which would clash with an existing one is not used. In practice this scheme would waste an enormous number of potential names, which is

particularly serious in SWARD as they are only 32 bits in length.

Hydra uses a hash table, the Global Symbol Table (GST), to translate object names into memory page frame numbers. This table maintains entries describing the location and the nature of the objects.

Hydra distinguishes between two classes of segments, active segments and passive segments. Active segments are resident in main memory, whereas passive segments are resident in secondary memory. Consequently, Hydra provides two different GST's, an active GST and a passive GST. The active GST, which is resident itself in main memory, translates names for all active segments, and a small number of passive segments. The passive GST is resident in secondary memory, and translates names for all passive segments. This scheme is shown in Figure 4.12.

4.3.2.3.5. Active and Passive Segments

The distinction between active (main store resident) and passive (disk resident) segments is made not only in the Hydra system, but in all of the capability systems under discussion. The active segment table is always much smaller than the passive table, and must be addressed for all active segment references. Consequently, the active tables are

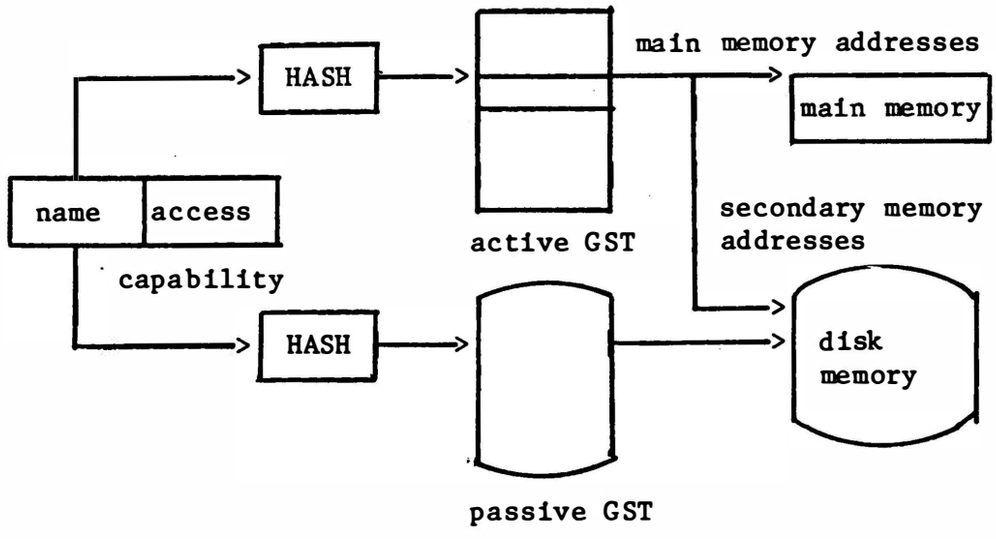


Figure 4.12 - the Hydra address translator

always loaded into main memory.

The passive table, which is much larger than the active table and is addressed much less frequently, can be placed in secondary memory. Most of the literature fails to document the passive translation table, however the table is present in all systems, including the Plessey 250 , CAL, CAP, Intel iAPX432 and the IBM System/38.

4.3.2.4. Efficient Address Translation

In systems which use a central mapping table to locate segments, the table must be consulted each time a capability is used to address a segment of memory. Such references not only include addressing data, but also fetching instructions from the code segments of a program. It was demonstrated by the CAL system that without adequate hardware support a capability based addressing scheme cannot be efficiently implemented. A graphic example can also be drawn from the IBM System/38. An average of 2.25 main store references per memory access would slow the memory access down to some 30% of full speed. Accordingly, the IBM processor, and all other systems apart from CAL, have provided specific hardware support. Such hardware can be divided into two classes, visible addressing registers and automatic caches.

4.3.2.4.1. Visible Addressing Registers

The hardware provided by the Plessey 250 and Chicago processors was in the form of a number of high speed, directly addressed, capability registers, as shown in Figure 4.13.

When a program wishes to address a segment, it must first load a capability register with the main memory base address of the segment, the main memory limit address and access rights information. This information is taken from both the capability and the central mapping

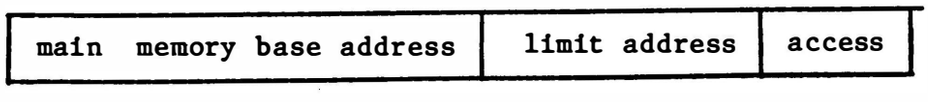


Figure 4.13 - a capability register

table. Capabilities which have not been loaded into registers are termed passive, and when a capability has been loaded into a register the capability is termed active (using Hydra terminology). Since instructions refer to data by the capability register number, only active capabilities can actually address store. Capability registers are usually used with a modifier (or index) register, which augments the base address.

The addressing register scheme has the advantage that it is extremely efficient. This is because object names are only translated into memory addresses when the register is loaded. Unfortunately, there are also a number of disadvantages. The only time that logical names are used to address segments is when the capability registers is loaded. Once the base and limit values have been loaded into a register, it is impossible to move the segment around in main memory, without checking all capability registers in all domains (and in all processors in the Plessey 250). It is also difficult to determine when a segment is no longer being addressed, and when it can be safely moved without an old register still being valid.

Hydra also uses some relocation registers to address store. Hydra was implemented on a PDP11 computer, which provides a number of small (64 k byte) paged address spaces, each consisting of 8 pages. Each page is addressed relative to a relocation register. When capabilities are activated in Hydra, the object is made visible in the 64 k byte address space of the user program, as shown in Figure 4.14. This is done by copying the relocation information from the Global Symbol Table (GST) into the appropriate relocation register. Because the PDP11 is a paged processor, segments in Hydra are all 8k bytes in size. Larger objects are composed of basic 'page' objects. Also, since the PDP11 cannot support demand paging (because some instructions are not repeatable), the relocation registers must be specifically loaded under program control, like the addressing registers of the Plessey 250. Since only a small number of pages are allowed in an address space, and because large objects must be composed of many pages, capabilities are frequently made active and passive in Hydra, an expensive operation.

Because of the problems associated with relocation register schemes, many processors provide automatic address translation caches.

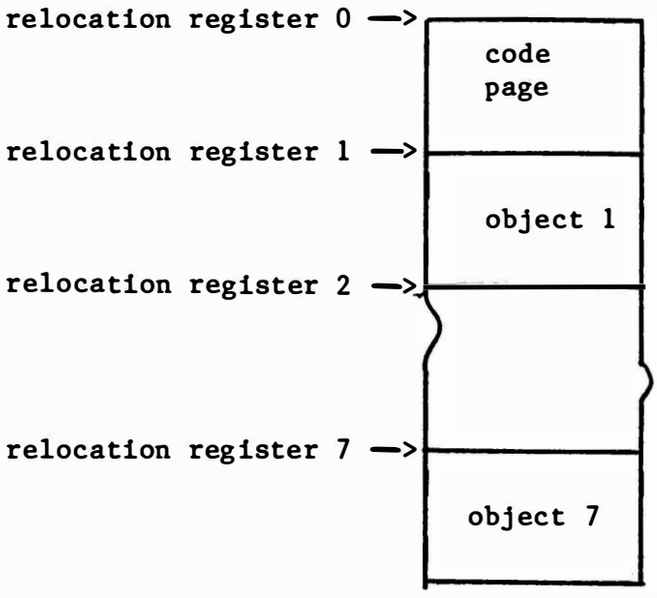


figure 4.14 - the Hydra address space

4.3.2.4.2. Address Translation Caches

Address translation caches are used to augment the active tables used to translate names, or virtual addresses, into main memory addresses. Such caches, which were described in Chapter 3, are used in CAP, Intel iAPX432, and the IBM System/38. The IBM system also provides a Resolved Address Register (RAR), for use by the microcode, which bypasses both the cache and the hash table (Houdek, Soltis and Hoffman, 1981).

In general, these caches reduce the number of main store accesses per memory reference significantly. It would appear that the cache chosen by Bishop (1977) does not reduce the number of accesses by a significant amount (as Bishop assumes a hit rate of only 50%). To be effective, such caches should aim for hit rates in excess of 90 %, because the active table search times are much greater than the access time of the caches (Strecker, 1978).

4.3.2.5. Logical Properties of Objects

To date we have only considered the essential administrative information which must be associated with objects, such as where the object actually resides, and how large it is. Some logical information

can also be associated with the object, possibly controlling the way in which the object may be addressed.

For segments, it is often desirable to have, in addition to the access rights, an extra property associated with the capability, namely a length attribute. Such a field allows some capabilities to only address part of a segment, whilst others can address the entire segment. CAP is the only processor of those described which actually allows this refinement (although the capability format defined for SWARD would allow size refinement). Each capability in CAP includes base and limit values relative to the original segment. Like the access rights field, these can be validated when the capability is used. All other systems associate the length of a segment with the segment itself.

If logical information pertains to an object itself, rather than a view, it can be placed in the central mapping table. For example, the segment size in CAP and similar systems is held in the object map. Moreover, if we associate a logical type field with an object, then other abstract, or extended, types of object can be addressed by the same mechanism which addresses memory segments. In this case, segments become a particular type of object which may be addressed from memory reference instructions. Other types of object may be addressed by special instructions (as in the IBM System/38) or in general by special code bodies (called type managers (Wulf, Levin and Harbison, 1981).

Processors such as CAP-3, Intel iAPX432 and Hydra allow extended type objects by placing a type field in the object map entry. Such systems as CAL, Gligor's scheme and Bishop's proposal associate the type field with the view, and place this information in the capability. Whilst there appears to be no reason for such a decision, it does allow different users to treat an object as different basic types.

Regardless of where logical information is actually placed, such type fields allow the processor to support abstract objects in a uniform manner. For this reason, we have placed little emphasis upon extended types of objects, and have concentrated on memory segments.

4.4. Memory Segmentation

In this section we consider some of the problems encountered in addressing segments in the capability schemes just described. An

important feature of a capability based addressing scheme is that all data is stored in segments of memory, regardless of how large or small. Many segments will, by nature, be either very large or quite small in size.

Large segments are often necessary to represent the data from files. Particularly large files must be composed of many segments; the larger each segment is, the fewer segments required. Thus, the capability mechanism should ideally be able to provide efficient support for large segments.

Small segments are generated from small procedures, data structures, stack frames etc. Studies have indicated that it is not uncommon for many very small segments to be generated, even in a conventional computer architecture (Batson and Brundage, 1977). The problem of managing small segments has been realized by many (e.g. Randel, 1969; Fabry, 1974; Wilkes and Needham, 1979; Wilkes, 1980; Gligor, 1978; Lanciaux, Schiller and Wulf, 1976 and Keedy, 1980).

Most of the capability systems we have examined do not provide an efficient environment for using both small and large segments. Two primary areas of contention appear to be the mapping tables and memory management, which we will now discuss.

4.4.1. Mapping Tables

The mapping tables become inefficient to operate when a large number of active segments must be supported. A large number of segments often increases the size of the table significantly, and may increase the access time as well. Moreover, if the active table becomes too large, it cannot be held permanently in main store, seriously affecting system efficiency.

Wilkes's proposal (1980) simply tries to remove some of the extra small segments present, and does not attempt to solve the basic management problem (see section 4.3.1.1). However, a few proposals have been made to ease the management of the central mapping tables.

The scheme proposed by Gligor (section 4.3.2.2) places the name space translation table in virtual store rather than in real store. Gligor suggests that because the object map resides in virtual memory,

it can grow to a much larger size, allowing the memory to support many small segments. He claims that the object map may be allowed to grow in virtual space, and slots need not be reused for a long time. Consequently, old capabilities need not be found and deleted as often. Unfortunately, any locality of reference which is exhibited within pages of store will not necessarily be reflected by locality within the map pages. This is because the order of the map entries bears no relationship to the location of segments within pages. Thus, after a short time it may be necessary to keep all of the pages of the map resident, even though only a few words of each page may be required. Hence Gligor's scheme does not adequately solve the basic problem of many segments.

Bishop solves the problem of map management by eliminating the object name map altogether. Instead, his system maps pages of memory, rather than segments. The number of virtual pages will remain constant regardless of how many segments each page contains. It is unfortunate that the conventional page tables proposed by Bishop are unsuitable for translating 50 bit addresses.

Lanciaux, Schiller and Wulf (1976) suggest that many small segments could be placed together in a large segment. This scheme reduces the number of map entries required, as each large object contains map entries for the objects which it contains. It does, however, create the problem of large segments, which together with small segments complicate memory management.

4.4.2. Memory Management

The task of memory management becomes more complex when the system must support both very small and very large objects. In segmented schemes, such as the Plessey 250, Chicago Magic Number computer, CAP and Intel iAPX432, small segments tend to fragment the main store excessively. In addition, small segments are expensive to transfer between primary and secondary memory. Large segments, on the other hand, must either be totally resident or absent from store. If insufficient space is available then the segment cannot be loaded, and the task which causes the memory reference must be suspended until space is made available. Such problems are also experienced in the conventional

segmented processors, discussed in Chapters 2 and 3.

In a paged machine, like Hydra, all segments must be exactly one page in size. Consequently, large segments cannot exist and must be composed of much smaller ones. Small segments waste much of the page that they occupy.

A paged and segmented memory, such as that of the IBM System/38, manages large segments very well by only loading those pages which are required, but this wastes a large amount of space for small segments, which must occupy at least one page. Again, such problems are experienced in the conventional paged and segmented processors, such as Multics.

A few of the processors which we have examined in this chapter have attempted to alleviate the complex memory management of both large and small segments. The solutions involve placing many small segments into each page of virtual memory. Both Bishop and Gligor place segments consecutively in virtual memory. Thus, many small segments can be placed in one page, and large segments may occupy many pages.

In Gligor's scheme all segments are organized randomly in store. Consequently, after a small amount of time, one would expect the virtual store to become fragmented, as segments are deleted and created. Moreover, there is no guarantee that segments which are addressed together will reside in the same page. Thus, this scheme may behave as badly as a paged and segmented scheme, in which the entire page is required in order to address only a small segment.

Bishop attempts to place all segments which are addressed together in an 'area', which consists of a variable number of pages. Thus, segments which are addressed together will be swapped between primary and secondary memory at the same time. Whilst the page locality of Bishop's scheme is superior to Gligor's, one would still expect the virtual space to become as fragmented as the real store of other capability processors. Because areas are all of different sizes, when an area is deleted, or moved, a hole is left in the virtual space. Even though the capabilities for the hole are deleted, it may not be possible to reuse the area without considerable store reorganization. Accordingly, Bishop provides a very elaborate garbage collection scheme.

Lanciaux's solution suggests that many small segments may be placed in one large segment. Consequently, all segments will be swapped between primary and secondary memory at the same time. Unfortunately, whilst solving the small segment problem, this scheme may create a large segment problem instead.

4.5. Conclusion

This chapter has described all the significant current capability based computers, and has developed some general models into which these systems can be placed. Most importantly, by this analysis, it has shown the difficulties that these systems experience in certain common situations.

The next chapter will reexamine these difficulties, and propose an addressing model which can avoid many of the problems.

5. A New Capability Based Addressing Model

This chapter develops a new addressing scheme which is capable of addressing, protecting and sharing the logical structures of a program (or information hiding module) in a uniform manner. This scheme is based on segmentation, which, as we saw in Chapter 2, has suitable properties for structuring logical objects. The scheme also makes use of capabilities as the mechanism for addressing, sharing and protecting such segments.

5.1. Aims of the Model

In Chapter 4 we examined some real capability based processors and some theoretical models. Whilst the philosophy of many of these systems was admirable, their implementations often exhibited considerable problems. Some of these problems were intrinsically associated with the approach, such as using a segmented store. Others, however, were present because of inadequate hardware. Examples of the latter include Hydra, in which the available hardware affected the maximum segment size dramatically, and CAL, which used a conventional, and quite unsuitable, computer. Because of inadequate hardware the CAL system was effectively useless and was abandoned.

The model presented in this chapter defines a hardware interface which can successfully implement most, if not all, of the systems discussed in Chapter 4. Processors for these systems based on the proposed model may even be simpler and more efficient than the original hardware used to implement them.

The requirements of the model may be summarized in terms of five basic aims: to solve the memory management problems associated with most capability based processors, to solve the address translation problems associated with other capability based systems, to produce a uniform addressing mechanism, to produce an efficient capability addressing mechanism, and to produce a flexible hardware unit. Some of these aims are not shared by the existing capability systems. We shall now consider these basic aims in turn.

5.1.1. Memory Management

Most of the capability systems discussed in this thesis apply a segmented main memory scheme in order to achieve segmented addressing. Unfortunately, this scheme does not cater well for either very large segments or for very small segments. Large segments are awkward because they must be held in contiguous memory. Small segments are inefficient to swap between main and secondary memory because the time taken to initiate the transfer may exceed the time taken to actually transfer the data.

Some systems have attempted to use paging as a basis for memory management. Hydra used a paging system by forcing all segments to be one fixed size. This scheme simplifies the memory management task, but does not solve the small and large segment problem. Small segments waste much of the page that they occupy, and large segments can not exist. Thus, this scheme creates even more segments than are logically required, as large segments are constructed from many smaller segments.

Some solutions (Gligor and Bishop) have used paging as the memory management model, and have superimposed a segmentation scheme on top of the virtual memory. Whilst these proposals have solved some of the small and large memory management problems, they still have inherent inefficiencies, as discussed in Chapter 4. The model proposed in this chapter attempts to solve the outstanding memory management problems of all these capability based computers.

5.1.2. Address Translation Problems

Many of the capability based processors experience significant problems in translating virtual addresses into memory addresses, especially when the system is burdened with many small segments. A source of contention is the central object table which contains an entry for each segment in the system and is usually split into an active table and a passive table. When the system contains many small segments the size of the central object table becomes excessive, and translation times may be increased.

In those systems which have removed the central object table, such as Bishop's (Bishop, 1977), the task of address translation is significantly simplified. The model proposed in this chapter seeks to

remove the overhead of many small segments, by removing the central object table altogether.

5.1.3. Uniformity and Simplicity

In a true capability based addressing scheme all local and permanent data should be addressed by the same mechanism. Only one way of addressing data should be provided, unlike systems such as the IBM System/38 which provide two different addressing mechanisms.

With one common addressing mechanism the system design becomes much simpler. A simpler design in not only easier to understand, but often yields a more orthogonal and less expensive implementation. Moreover, only one sharing and protection mechanism is required. The model proposed in this chapter avoids unnecessary duplication by providing only one way of addressing memory.

5.1.4. Efficiency

The CAL system demonstrated that a capability based addressing scheme requires hardware support for an efficient implementation. Even in those systems which have provided hardware support for addressing memory, the use of capabilities still creates inefficiencies, as described in the last chapter. The model proposed in this chapter defines a hardware addressing structure which can be efficiently implemented with current technology. Moreover, the model is capable of implementing many different software structures without significant overheads.

5.1.5. Flexibility

Most processors, both capability based and of conventional design, are built with a specific addressing structure in mind. For example, the instruction operands in the Intel iAPX432 processor expect a particular C-list structure. The operands of the CAP system expect a different C-list structure. Because these organizations are so well understood by the processor hardware (and firmware) it is unlikely that one processor could efficiently or easily implement the C-list structure of another processor.

The lack of flexibility in some of the existing systems is not a problem, only because the system design does not change significantly at any stage. However, in a research environment a flexible processor is extremely desirable, as it allows the hardware to survive a number of major redesigns of the software ideas. The model proposed in this chapter should be capable not only of efficiently implementing a particular addressing structure, but also of implementing any of the other capability addressing structures described in Chapter 4, such as the different C-lists of CAP, Intel iAPX432 etc. The model can achieve this flexibility by providing a general hardware unit which provides a capability based addressing style, and a small section of software (or firmware if the host machine is microcoded) which understands the addressing structure. If the software ideas change at any stage, then the hardware may remain the same and the software or firmware may be changed.

5.2. Object Addressing

The capability based addressing schemes described in Chapter 4 all have the property that all addressable objects are treated alike in terms of addressing and protection. All are addressed via the capability mechanism which the processor uses. Such references can be categorized into two classes, memory segments and high-level objects. High-level objects include I/O devices, data abstractions, program modules (Keedy, 1982a) and type managers (Wulf, Levin and Harbison, 1981).

When a memory segment is addressed (via memory reference instructions) the capability mechanism is used to find a segment of memory and make it available to the program. Thus, in a purely segmented system the central object table may contain the main memory address of the segment, and the size of the segment. The access rights field of the capability can then be used to restrict certain operations on the segment. To produce efficient memory references this mechanism is nearly always augmented by some special hardware.

High level objects are also addressed via the capability mechanism. However, the central object table contains information which declares that the object is not a memory segment and requires further software or

firmware assistance. (Alternatively, this information may be held in the capability (Lampson and Sturgis, 1976).) These high level objects are not usually addressed by the normal memory reference instructions. Type checking information may then validate the type of instruction against the type of object. For example, a memory segment may be addressed by an add instruction, but a program module is addressed via a call instruction.

From this viewpoint, capability support can be built into a processor in two separate areas: first, a section of hardware which allows efficient manipulation of memory segments; second, a body of software, or firmware, which interprets operations on high level objects. Thus, the knowledge of high level objects need not be built into the processor. The information which usually resides in the central object table about high level objects (e.g. the type of the object) can now either reside in the capability for the object (as in the CAL system) or can be found in segments associated with the object itself (e.g. with the code which manipulates the object (as in the MONADS system). The implementation of operations on high level objects is left entirely up to the software or firmware concerned. This general approach in used in the addressing model described in this chapter. This allows us to design hardware which is very efficient at addressing segments of memory and yet, when combined with suitable firmware, provides a flexible addressing structure. We will now consider the form of the memory segmentation hardware.

5.3. Segment Addressing

5.3.1. The Basic Form of a Capability

The virtual memory of the proposed capability based addressing scheme is addressed via a number of capability registers, each of which holds a segment capability. These capability registers are the only addressing mechanism available to the processor. Each register, shown in Figure 5.1, contains three fields: an address, a length and some access rights. Before we discuss the precise nature of these fields, it will be useful to consider the advantages of a scheme based on registers:

(1) The problem of operand size for addressing memory via capabilities, discussed in Chapter 4, disappears in a register based system because once a register has been loaded with a capability subsequent references need only specify a register number, which is likely to be of the order of four bits.

(2) Registers hide the nature and structure of the logical addressing mechanism from the processor instruction set. The model is invariant to the method of saving capabilities (i.e. C-lists of various structures or tagged protected memory) and the actual structure of a C-list or tagged memory need not be determined at the hardware level (for example, whether the C-list allows tree structures or lattice structures). Thus, the scheme is flexible, because the software structures may be modified without affecting the hardware.

(3) Because registers can uniformly address all kinds of segment, no special registers are required, for example to implement a stack pointer, display registers, etc. Indeed, a combination of a capability register and an index register can be used not only to address data but also to control program sequencing.

(4) Because registers are normally built from high speed logic, they have the same advantages as capability caches (cf. IBM System/38 and Intel iAPX432), but they are generally less expensive and in some cases easier to implement. Because the scheme only translates logical addresses (of the form C-list number and slot number) into capabilities when the register is loaded, it avoids many unnecessary memory accesses by removing repeated references to the C-list.

(5) Given the use of registers, protection can be efficiently implemented by allowing only particular microcoded or kernel instructions (or only instructions executing in a special machine state) to modify their contents. This makes it impossible to modify a capability illegally once it has been placed in a register. The protection of capabilities outside of registers depends on the C-list structure, or tagging mechanism, which the processor provides.

A register based addressing scheme does have some basic disadvantages. First, it requires the compiler or assembler programmer to allocate and deallocate the registers. This problem is not considered

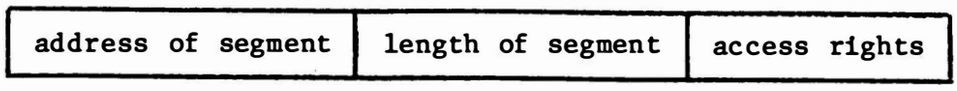


Figure 5.1 - a capability register

serious enough to overpower the advantages of the scheme, for two reasons. Assembler programmers are far better at judging the working set of a program than a cache, and can choose the correct registers to allocate. Furthermore, compilers often have to allocate data registers, and have successfully done so for a long time. Addressing registers are no more difficult to allocate than data registers. Also, the compiler can form conventions which dedicate the use of certain registers. For example, one register may be used for addressing scalars at lexical level zero, whilst another register may be dedicated for addressing data at the current lexical level. Such conventions can help register allocation significantly. Capability registers are also easier to allocate than many of the addressing registers used in conventional processors, because they are the only addressing mechanism. Thus, the compiler is only concerned with one addressing scheme, rather than many.

Second, the registers may need to be saved and reloaded when a module is entered by a call instruction, or when a process switch is executed. When a new module is entered the compiler may either invalidate the active capability registers, or the call instruction may save their contents. If the registers are invalidated, then the program must restore them after the call. If they are saved, then the return instruction must restore the original contents. If a capability cache is used instead of registers, then it must be invalidated when the call instruction is executed. The cache will then reload itself after the call as the capabilities are used. Thus, the cache scheme is equivalent to the register scheme which invalidates the contents of the registers. If the registers contents are saved prior to a call, then on return the old capabilities are simply copied from an image in memory (e.g. as part of the linkage on the stack). However, when the cache is reloaded, the C-list entries must be retrieved, which could take longer than a simple

copy operation.

When a process change is made, the registers must be saved for the executing process, and the registers for the new process must be loaded. If a capability cache is used it must be cleared of capabilities from the old process, and will automatically reload when operands are used. Providing that some hardware support is provided to support efficient process changes (such as described in chapter 7) there is no reason why the register based scheme should be less efficient than the cache scheme.

5.3.2. The Load-capability-register Instruction

Because in the proposed scheme the logical structure of the addressing mechanism is hidden from the hardware, special software (or firmware) must be written which understands this structure. One such instruction is the load-capability-register instruction. This instruction (or kernel routine if the machine does not possess a microcoded control unit) accepts a capability register number and a program address, and loads the capability found at that address into the register. If the processor uses a C-list for holding capabilities, then the program address may define a C-list number and a slot number, as described in Chapter 4. If the system must at some later stage understand a different C-list structure, then only the load-capability-register instruction need be altered. All other data manipulation instructions address their operands via a capability register. This combination of microcode and hardware gives the model a large degree of flexibility but still allows very efficient addressing.

5.3.3. Representation of a Capability

A memory capability, shown in Figure 5.1, is composed of three sections: an address, a length and a set of access rights. The key difference between these registers and those of the Plessey 250 (England, 1972) is that our capability uses a virtual address, rather than a main memory address. As shown in Chapter 4, the use of main memory addresses both causes difficulties in re-organizing store and also means that the main memory must be segmented. Apart from the difficulties of organizing a segmented memory, a central object table is required in the Plessey 250 to map segment addresses onto main memory

addresses, which causes further problems related to the size of the object table, as discussed in chapter 4. The use of a virtual address in the capability registers avoids these problems. First, the store can be physically reorganized without affecting the addresses held in registers. Second, from the viewpoint of the memory management system the memory does not appear to be segmented This removes the problems of a segmented memory, and also means that the system does not need a central object table.

The length field of the capability holds the size of the segment, and must be large enough to allow large segments. Ideally, this field is the same size as the virtual address. However, it may be considerably less without being restrictive. By contrast, the length field used in Bishop's capability is too small (9 bits) to allow large objects to be created if the length is treated as a byte or word count. Alternatively if the length field is considered as a larger unit (e.g. a page) then the unit of protection and store allocation is not sufficiently granular in Bishop's scheme.

The access rights field must allow operations to be performed or restricted, such as read only, write only, read-write, execute etc. These can be encoded in a bit pattern.

Thus the registers which we propose differ significantly from those of the Plessey 250. The format of the capability is similar to that of Bishop, except that the length field of the model will be large enough to address large segments. The model differs significantly from those systems which implement segmentation at the memory level, and use a central object table, such as CAP, Hydra, Plessey 250, Gligor, Intel etc. We shall now briefly consider the refinement properties of the capabilities.

5.3.4. Refinement of Capabilities

In will be recalled that in Chapter 4 we introduced the concept of capability refinement. All of the systems discussed in this chapter allowed the access rights of a capability to be reduced, and a diminished copy of the capability given to another user. These capabilities then have access to the same object as the master, but with fewer access privileges. A capability may also be refined in range as

well as type of access. This type of refinement is useful when a procedure wishes to grant another user access to only part of a data structure (e.g. when passing a parameter by reference). In the model proposed in this chapter, access to a segment may be refined by modifying the base virtual address and the segment length fields of the capability. The new capability can then only address part of the original structure, as shown in Figure 5.2. Surprisingly, very few systems have allowed a capability to be reduced in range, although the Plessey 250, Bishop and CAP could allow such refinement.

Whilst the format of the Plessey 250 registers would in principle allow a segment addressed by a register to be refined in size, the capability format does not contain a limit field. All capabilities point to a central object table which contains the the main memory limit of the object. This limit is later copied into the register when it is loaded. Also, because the Plessey 250 registers hold main memory addresses, when moving a segment in main store it would be difficult to determine whether a capability pointed to part of the segment in question without checking if the refined base and limit were contained in the segment. Thus, the Plessey 250 addressing scheme does not allow segments to be refined in size. This is shown in Figure 5.3.

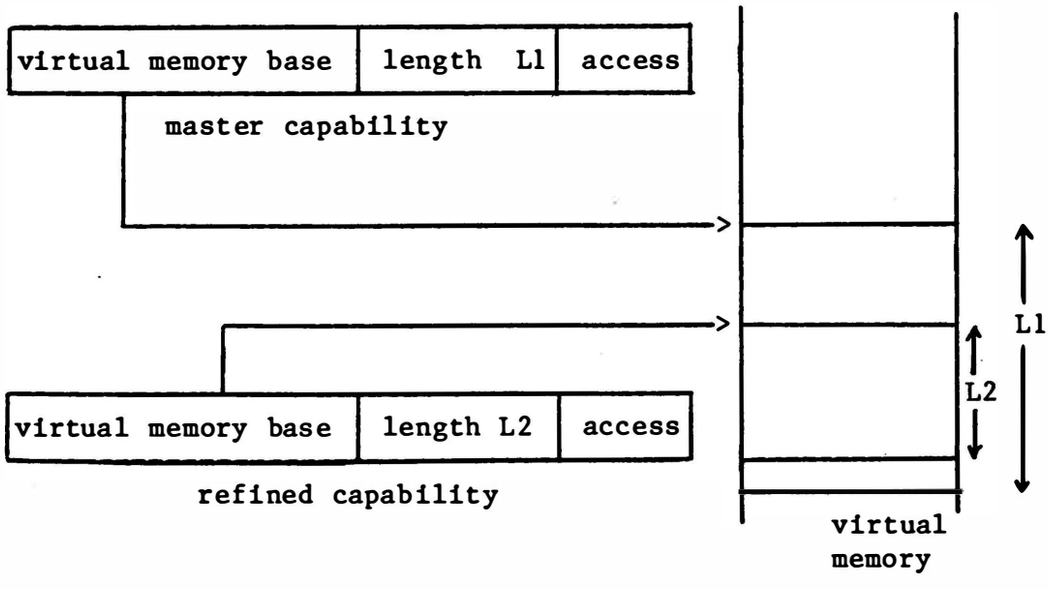


Figure 5.2 - a refined capability

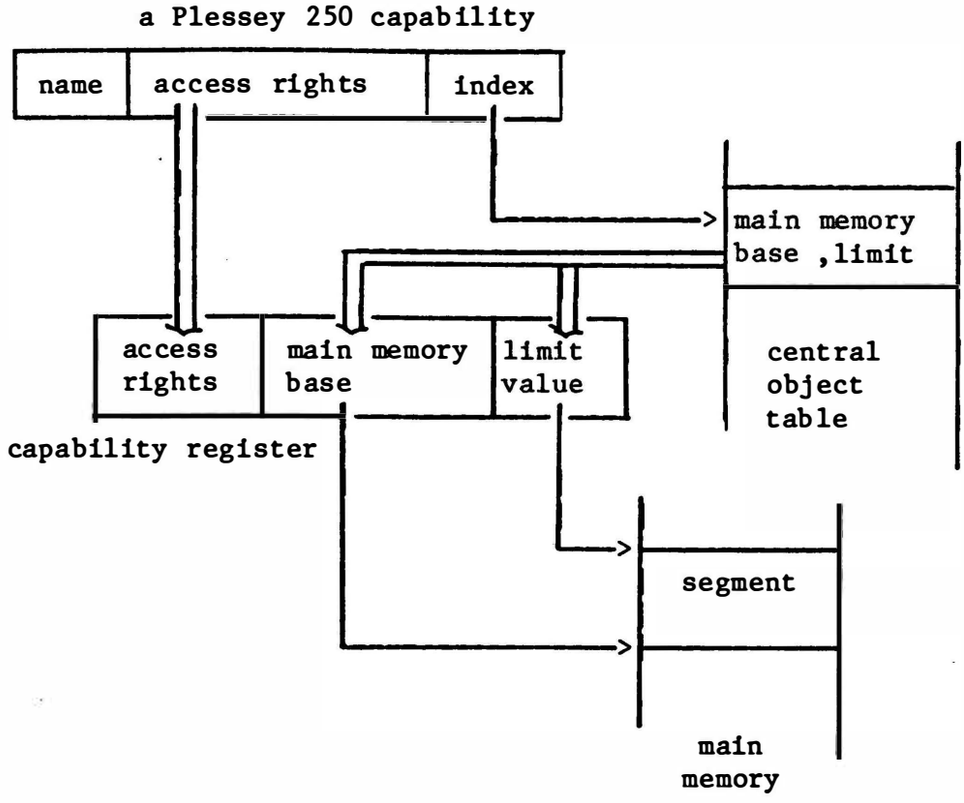


Figure 5.3 -The Plessey 250 - no size refinement

Bishop's capability contains a virtual base address and also the size of the segment. Thus a capability may be created which only addresses part of the original segment. Unfortunately, Bishop does not use enough bits to allow large objects to be addressed (or alternatively support sufficient granularity).

In CAP both the capability and the central object table contain a size field and a base field as shown in Figure 5.4. The fields in the central object table are used as absolute main memory bounds of the segment, whereas the values in the capability are interpreted relative to the original bounds. Consequently, a refined capability, which only allows access to part of the segment, may be created.

The refinement system of the model capability registers closely matches that of Bishop's capabilities. However, the size field of our capability is large enough to allow large segments to be addressed. Because both of these systems use virtual addresses in capabilities, there is no danger in allowing many capabilities to reference part of an

object. In CAP, however, one set of base and limit values must be associated with the main memory properties of a segment, and another set of values must be associated with the capability. Both of these values must be validated before the reference can proceed to memory. This overhead is not present in the model capability system. Thus, the refinement qualities of the model appear to match, and in most cases improve on, those of other systems. It is noteworthy that very few systems allow this useful operation at all.

5.3.5. Summary

So far, the capability addressing model fulfils some of its primary aims. The use of capability registers allows a flexible hardware unit to be constructed. If the addressing structure is modified at any stage, then the load-capability register instruction can be modified. Because they are the only addressing mechanisms available to the processor, they

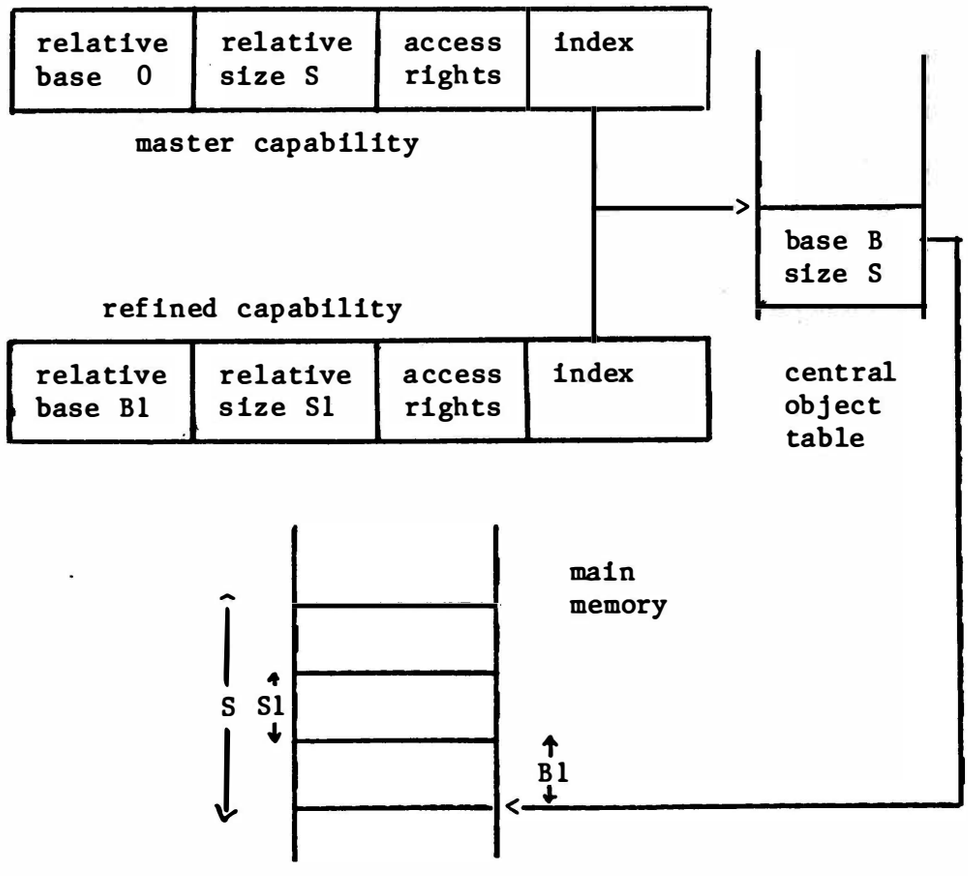


Figure 5.4 - CAP refinement of capabilities

also provide a uniform, simple and efficient method of addressing store. In order to fulfil all of the aims we must describe a virtual memory which can hold small and large segments.

5.4. Virtual Memory

5.4.1. Requirements of the Virtual Memory

In Chapters 2, 3 and 4 we examined many different virtual memory organizations. In this section we will examine the requirements of the virtual memory which is used by the model. They are as follows:

(1) Virtual addresses should be large and unique. When a segment is created it consumes a range of virtual addresses, which eventually reside in C-lists and capability registers. When a segment is deleted, the address may either be found and destroyed, or never reused. A large addressing range means that it is not necessary to reuse addresses, saving on the number of addresses which need to be found and deleted.

(2) The virtual memory must be the only memory mechanism. This uniform treatment of store means that all data, files and code, are present in the same virtual memory without support from a separate file store. This technique was pioneered in MULTICS and has been used in other capability systems with many advantages (Rosenberg and Keedy, 1981a).

(3) The tables, or mechanism, used to translate virtual addresses to main store addresses should not affect the way in which the virtual memory management software organizes the secondary store. This condition is not met in many existing systems, such as MULTICS. The page table structure which is used by the hardware, or firmware, to translate virtual addresses into main memory addresses is also used by the software to locate pages in secondary memory. If the software wishes to change the table format then the hardware may also need to be modified. Greater flexibility is desirable because better secondary storage methods may be devised after the hardware has been built. Thus secondary memory address translation and main memory address translation should be independent.

(4) Virtual store management should be simple. If virtual addresses are ever reused, the virtual space may become fragmented due to objects

being created and destroyed. Both Gligor (1978) and Bishop (1977) propose the use of large paged virtual memories for holding segments. Gligor packs segments into virtual space in a random manner, whereas Bishop places common segments in areas, or groups. The first scheme, whilst conceptually simple, means that the virtual space may become very fragmented in time. Bishop's scheme does not totally avoid this problem, as areas themselves are variable in size. The virtual store should be organized so that if addresses are ever reused, the store can be reorganized without massive data manipulation.

(5) The virtual store should efficiently support both large and small segments. This problem is vastly simplified by implementing the segmentation at the register level. It then only becomes necessary for the virtual space to hold both large and small areas. All of the models previously discussed fail to provide an acceptable mechanism.

(6) Real store management should be simple. Unlike the segmented schemes of some capability systems, the model can choose another main store organization without losing the logical advantages of segmentation. Thus a simpler main store scheme can be used instead of the complex and inefficient segmented scheme.

Unfortunately all of the virtual memory systems discussed in the earlier part of this thesis fail to provide a suitable virtual memory which supports all these requirements. Another scheme, not previously discussed, allows a conventional processor to efficiently support small and large segments. The next section will discuss this model.

5.4.2. A Small Segment Model

Keedy (1980) proposes a memory management model which allows a conventional processor to support both large and small segments without the inefficiencies described in Chapters 2 and 4. The scheme uses capabilities which hold a virtual address, segment length and access rights. The virtual address is further composed of an address space number and an offset within the address space. Each offset is composed of a page number and a within page displacement.

Address translation is performed via a number of tables, shown in Figure 5.5. The address space list is consulted to find the location in main memory of the page table for the space. Each page table entry

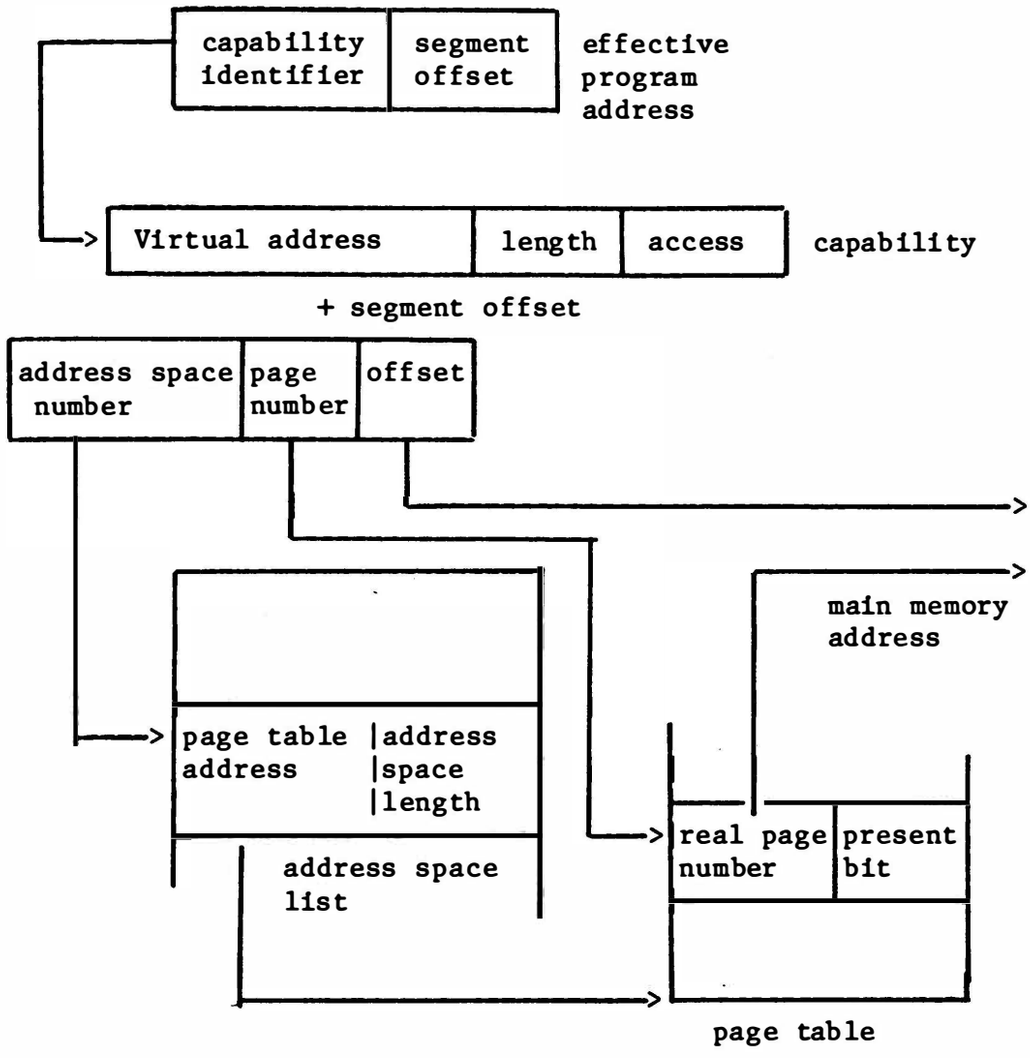


Figure 5.5 - the Keedy addressing scheme

reveals either the main memory address of the page or the secondary memory address. This model is similar to the paged and segmented scheme discussed in Chapter 2, and thus could be supported by a processor similar in nature to MULTICS. Unlike MULTICS, however, a segment offset is added to the virtual address before it is translated into a main memory address. Thus, because many segments may be placed in one page of main memory, this model can support items 4, 5 and 6 of the model aims, namely simple real and virtual store management and support for small and large segments. All the advantages of the scheme are discussed in Keedy (1980). However, the following are particularly relevant.

5.4.2.1. Simple Real Memory Management

The main memory is far easier to manage in this model than the segmented solutions because store is allocated in fixed size pages. Provided that some reference locality is achieved, several independently addressed and protected segments can be packed into a single address space, and the amount of space lost to internal fragmentation is on average only half a page per address space rather than half a page per segment (or more for small segments). Thus while internal fragmentation is not entirely eliminated, the amount of space wasted in this way can be greatly reduced.

5.4.2.2. Simple Virtual Memory Management

The virtual memory is easier to manage than that of Gligor or Bishop because the virtual space is allocated in fixed size units, namely address spaces. Typically, because of reference locality, all the segments of a module are placed together in a single address space. If the module is deleted, and all old addresses within the space are collected and destroyed, then the address of the address space may be reused. Because the address spaces are all of the same size, the hole left in the virtual space is not of a variable size, unlike those of Bishop and Gligor. Consequently, the virtual space will not become as fragmented as those of Bishop or Gligor.

Even though the address spaces are all of a fixed size, spaces smaller than the maximum size do not actually require this fixed amount of disk space to be allocated. Thus, the scheme does not require any more disk space or page table space than other schemes.

5.4.2.3. Support for Small and Large Segments

The scheme does not use a large central object table, but rather a smaller address space list, and can therefore support many small segments efficiently. As more segments are added to an address space, the address space list will remain the same size, and not grow like the central object tables in many of the capability systems. Moreover, provided that a reasonable amount of locality of reference is exhibited, many small related segments may be placed in one page, reducing the amount of wasted space and making segment swapping more efficient. Large

segments may be composed of many pages. Because only those pages actually being addressed are held in main store the scheme does not have the large segment problems experienced in segmented schemes.

Thus, the scheme solves both the memory management problems and the address translation problems associated with many small and large segments. However, the model in this form does not support requirements 1, 2 and 3 of the model aims, namely large unique virtual addresses, a uniform store and separate main and secondary memory address translation systems. The next section shows how the model can be modified and used to provide a virtual memory with all the required attributes.

5.4.3. Applying the Memory Management Model

Requirements 1, 2 and 3 of the model demanded a large uniform virtual memory and a separate main and secondary memory address translation system. A large uniform uniquely addressed memory which holds all data and files implies an address size of the order of 64 bits, as used in some other capability systems. The model described in section 5.3 implies an address size comparable to processors such as the ICL2900, MULTICS etc, and of the order of 32 bits because it uses page tables in main memory for address translation. Unfortunately, a simple scaling up of the tables is not possible because the large address is 2^32 times that of the conventional address. The problems with conventional page tables were discussed in Chapter 4. Moreover, the table structure would be used for both main memory and secondary store address translation, contrary to the requirements set out in section 5.4.1. Thus, in order to use the memory model, the address size must be expanded to about 64 bits in size and another address translation mechanism must be found. Also, to allow large segments to be created, the size of an individual address space must be larger than 2^16 words, as implied in the model (if half of the address is used for the address space number). Thus, the 64 bit address must be composed of an address space number field of about 32 bits, and a within address space displacement of 32 bits. This would allow the largest address space to be 2^32 words in size. In order to find a suitable address translation mechanism we can consider a number of the techniques described in this thesis.

Gligor's addressing scheme assumes the presence of a robust virtual memory without indicating how to provide such a mechanism. Bishop attempts to use conventional page tables to translate addresses. We showed in Chapter 4 that this technique is unsuitable because of the size of the directly indexed page table. For the same reasons, the page and segment tables proposed by Keedy, and used by the ICL2900 series, MULTICS, Prime 750 (Prime, 1979) etc, are unsuitable because of the space required for the tables, and the time taken to translate an address.

The best form of address translation for an address of this size is the associative technique used by Atlas, IBM System/38 and MU6-G. These methods only attempt to translate addresses for those pages resident in main memory, and leave the software free to organize the secondary memory translation tables in any suitable way.

Thus, by increasing the address size to 64 bits and by using an associative address translation scheme the Keedy model can provide an acceptable virtual memory for our capability model.

5.4.4. Summary

The specification for the capability addressing model of this chapter is now complete, and is summarized in Figure 5.6. The model provides a flexible, uniform, simple and efficient method for addressing store. The virtual memory required can be realistically provided by a modification of the Keedy model.

5.5. Application of the Model

The first three sections of this chapter described a hardware model which can be used to support a capability style of addressing. A major consideration in the design was that the model be flexible enough to cater for a number of different software ideas. This section will demonstrate that the model is flexible by applying it to three different different software models: the Intel iAPX432, CAP-3 and MONADS.

As described earlier, the implementation of the model consists of two separate sections, the hardware registers and address translation mechanism, and a body of software or firmware. In each of the examples to be considered, a different load-capability-register instruction must

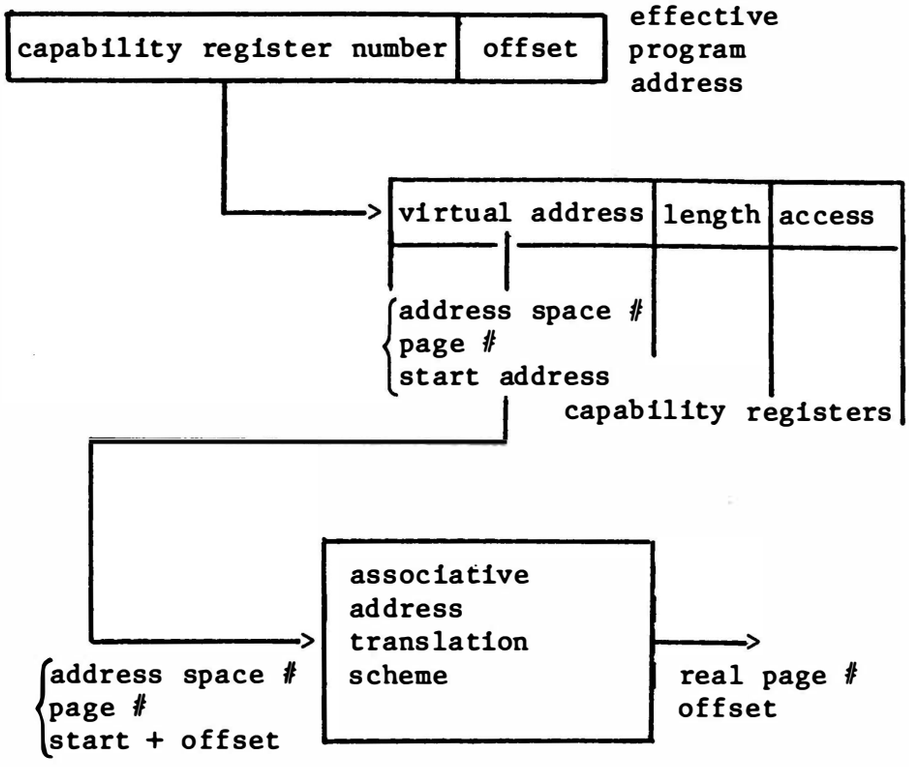


Figure 5.6 - the new addressing model

be implemented to address segments. The hardware can then be used to support the addressing structure. Access to high level objects will typically be via a call instruction, which can be supported in microcode or software and does not affect the hardware.

5.5.1. The INTEL iAPX432

The Intel iAPX432 supports information hiding modules, each of which consist of a number of memory segments. We will now describe the C-list structure used by the Intel processor.

5.5.1.1. The Intel Addressing Structure

The iAPX432 uses two different types of segment, data segments and access segments. Data segments are used to hold data and code. The addressing environment of a module is defined by an access segment, which contains all of the capabilities for the addressable segments. A capability consists of a unique segment number and a set of access rights.

The segment number is translated into a main memory address by a central segment table. Each segment table entry contains a start address, segment size, presence bit and additional segment information. Whilst the literature suggests that this central table is indexed directly on segment number, such a scheme is quite inappropriate and we assumed in Chapter 4 that in practice some other scheme is used.

While other tables are used to bind access and code segments to a module (e.g. context and domain objects), a program may address memory by supplying an index into the access segment, called the segment selector, and an offset within the segment, as shown in Figure 5.7. An access segment may in turn address another access segment, and a tree structured addressing environment may be created, as shown in Figure 5.8. Thus, rather than identifying a number of specific classes of segment the iAPX432 only recognizes two main classes, and allows the environment to be structured as a tree of capabilities.

As discussed in Chapter 4, the Intel processor is particularly poor at supporting very large and very small segments. Large segments complicate store management and small segments are expensive to swap in and out of store, and also increase the size of the segment list. All of these problems are removed when our hardware model is used to support the iAPX432 addressing structure.

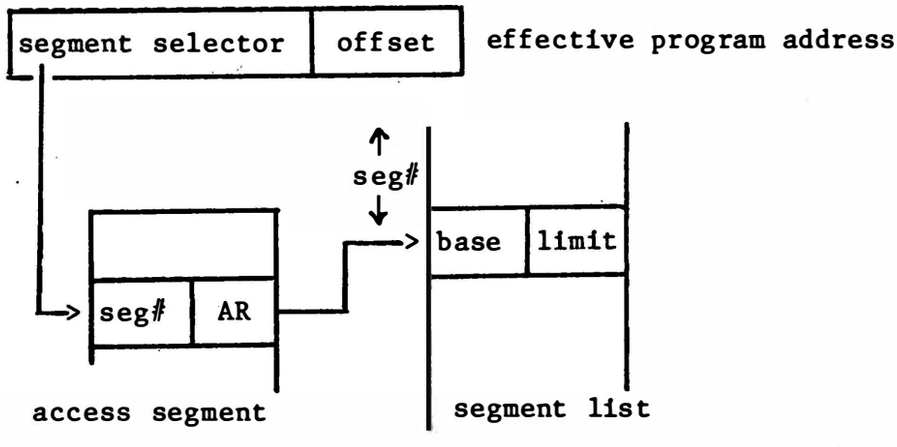


Figure 5.7 - the Intel iAPX432 addressing structure

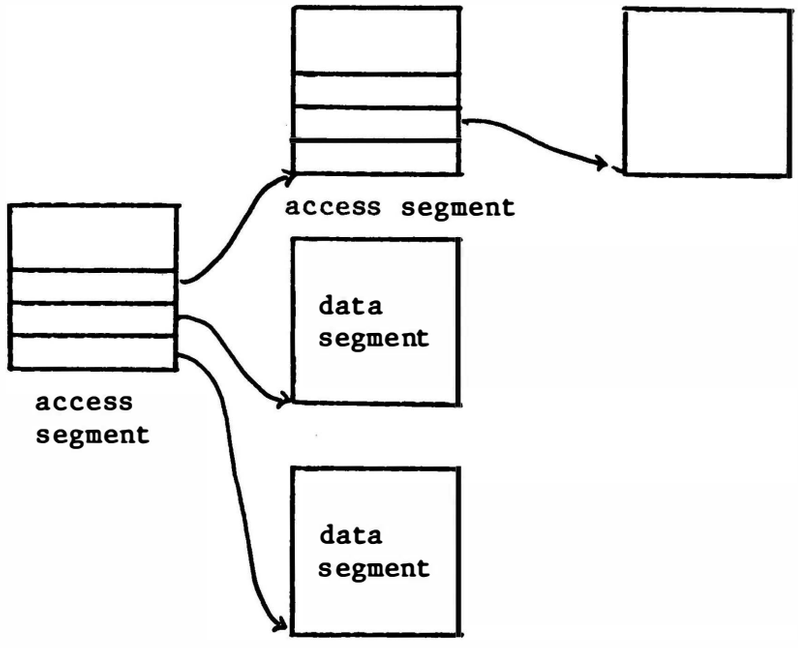


Figure 5.8 - a tree of access segments

5.5.1.2. Mapping the Intel iAPX432 onto the Model

In order to map the Intel software configuration onto the model, the processor must adopt the capability format of the model. This change does not affect the use of access segments, but does allow the model to be used to address store. In addition, a load-capability-register instruction must be provided to translate addresses of the form <segment selector> into a capability. This instruction must also detect capabilities for high level objects and stop them from being loaded into registers. High level object support can then be provided by special software of firmware. With these changes, which do not affect the aims of the system, the iAPX432 inherits the simplicity and efficiency of the model, in three areas.

First, there is no longer any need for the central segment list. Without this list, the system can efficiently support many small segments. Second, the real store is no longer segmented, avoiding store management problems with large segments. Third, because the real store is paged, many small segments may be swapped in one operation. The new addressing scheme is shown in Figure 5.9.

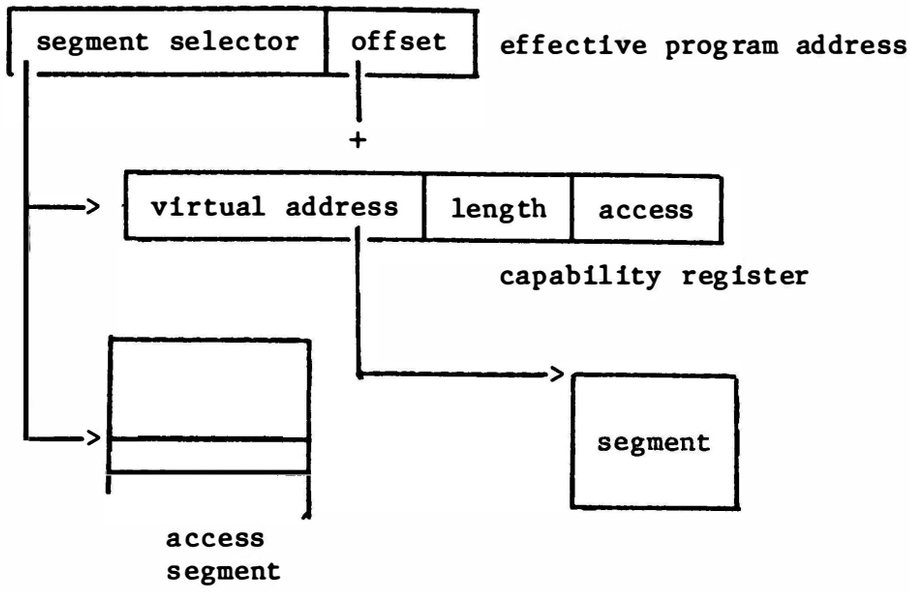


Figure 5.9 - the new Intel addressing scheme

Access segments can be protected from corruption by only ever granting capabilities to them with read-only access to a user program. Thus, a program may be able to use an access segment capability as the target of a data manipulation instruction, but cannot modify the contents of the access segment itself.

This implementation shows that the capability register scheme is not only flexible enough to implement the C-list structure of the Intel iAPX432, but also improves the efficiency of the final addressing mechanism. The secondary memory translation is no longer dependent on the segment list, and may be freely modified.

5.5.2. CAP-3

CAP-3 is a capability based computer which addresses a segmented memory via a C-list structure.

5.5.2.1. CAP-3 Addressing Structure

Segments in CAP are addressed via one of the different C-lists attached to a protection domain. Each C-list is addressed by a domain descriptor, and there are 16 domain descriptors attached to any domain. Each C-list entry holds a capability which defines the bounds of the

reference and the access rights. Capabilities also contain a pointer into a central object table, which holds the central mapping information for the segment. A program can address memory by forming a 32 bit virtual address, which is constructed from a 4 bit C-list selector (one of the 16 domain descriptors), an 8 bit capability number (relative to the C-list chosen), 16 bits of offset within a segment and 4 unused bits. This address is mapped onto a real memory address via the central object table, which contains an entry for every active segment. A capability cache helps speed up the address translation for frequently used segments. The addressing structure is summarized in Figure 5.10.

CAP-3 differs from the Intel processor by using a different C-list organization. In CAP, a domain can only address one of the C-lists for which it has a domain descriptor. Unlike the Intel processor, CAP-3 cannot construct a tree of capability segments. Since CAP-3 uses the same main store organization as the iAPX432, it possesses the same inefficiencies when small and large segments are addressed. We will now show how the capability register addressing scheme may be applied to the CAP-3 architecture.

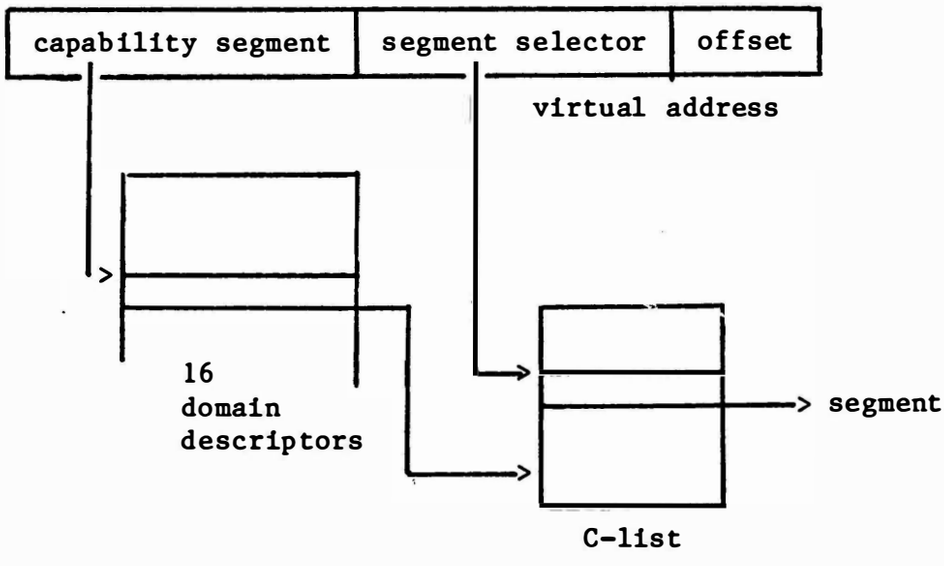


Figure 5.10 - the CAP addressing structure

5.5.2.2. Mapping CAP-3 onto the Model

As in the Intel processor a load-capability-register instruction must be written which understands the C-list structure of CAP-3. This instruction accepts a 4 bit C-list selector value, an 8 bit capability selector and loads a register with the capability.

Instructions can then address the segments of store via the capability registers. The scheme inherits the advantages of the model. Small and large segments can be efficiently addressed and transferred in and out of store. The problem of managing the central object table disappears as the table is eliminated. The only instruction which understands the use of domain descriptors and C-lists is the load-capability-register instruction.

5.5.3. MONADS

The MONADS system requires software systems to be constructed from a number of information hiding modules, each composed of a number of memory segments, namely local data segments, file data segments, retained data segments, code-related data segments, parameter segments and code segments (Keedy, 1982a). We will now describe the addressing structure and show how the model hardware may be used to implement this structure.

5.5.3.1. The MONADS Addressing Structure

An information hiding module in MONADS is active and may address its data segments when a process is executing within the code segments of the module. Under such circumstances a process stack will be present. This stack forms the centre of the MONADS addressing structure. Each class of segment is addressed via a separate segment list. Thus, for example, each segment of local data is addressed via an offset relative to the local segment list, and each segment of file data is addressed via the file segment list. Each segment list contains a list of capabilities for the segments of the class. A capability, shown in Figure 5.11, consists of a virtual address field, a length field and a set of access rights. Thus, any segment may be addressed by a segment list number (called the base number b) and a segment number (s), as shown in Figure 5.12. Moreover, each segment list is addressed via the

Figure 5.11 - a MONADS capability

BASE table, as shown in Figure 5.13.

The process stack forms a convenient place to hold the BASE table and some of the C-lists, because some of these pertain to the module and process intersection. Other C-lists, such as the file C-list, are held off the stack. Thus, the BASE table conceptually holds capabilities for the C-lists, and each C-list holds capabilities for each segment in the

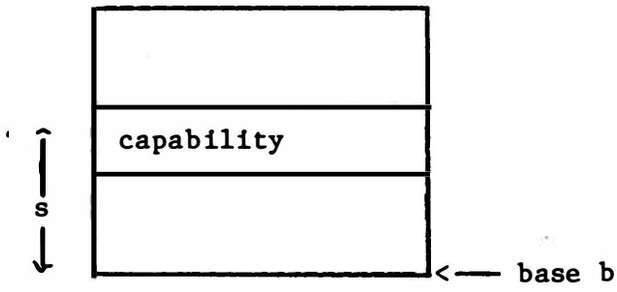


Figure 5.12 - a segment list

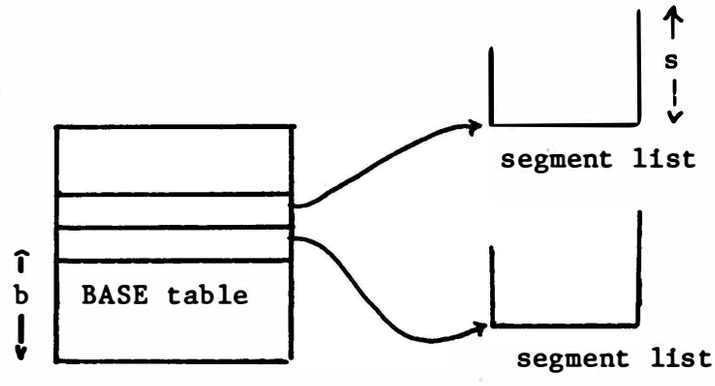


Figure 5.13 - the BASE table

class. Because the C-lists are not general purpose, like in the Intel iAPX432, but dedicated, capabilities within a C-list cannot point to another C-list.

An instruction forms a logical address of the form <base number, segment number, offset>. The addressing structure, summarized in Figure 5.14, allows this address to be translated into a capability and offset. The next section shows how the model proposed in this chapter may be used to support this structure.

5.5.3.2. Mapping the MONADS Software Structure onto the Model

In mapping the addressing structure described in 5.4.1.1 onto the model hardware care must be taken to protect the contents of the BASE table and the various C-lists. As with the other implementations, a load-capability-register instruction must be developed which translates a logical address, of the form <base number, segment number>, into a capability, and saves the capability in a register. The register number may then be used as the operand for future memory reference

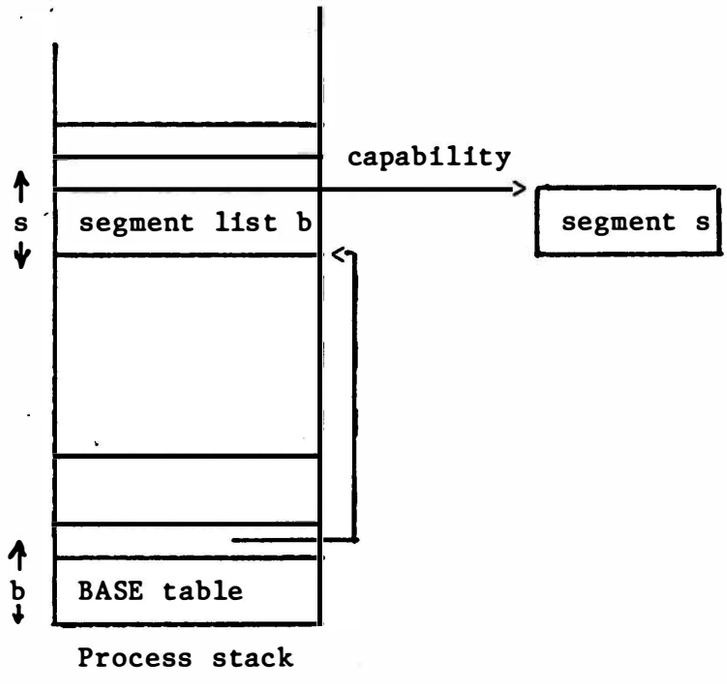


Figure 5.14 - the MONADS addressing structure

instructions. This translation is clearly efficient. The logical address is only translated into a capability when a register is loaded. Subsequent accesses to the segment bypass this translation.

In order for the load-capability-register instruction to translate the logical addresses, it must have access to the BASE table and also the various C-lists. Because the stack itself resides in virtual memory, these tables are addressable via the capability registers. The areas of the stack which contain sensitive information may be protected by never issuing to programs capabilities to the information. Moreover, because the tables are simply segments, there is no need for them necessarily to reside on the stack at all. In fact, those lists which do not belong to a process reside in other segments of virtual memory. The implementation is summarized in Figure 5.15.

5.5.4. Summary

In this section we have implemented three different capability addressing structures using the addressing hardware proposed in this

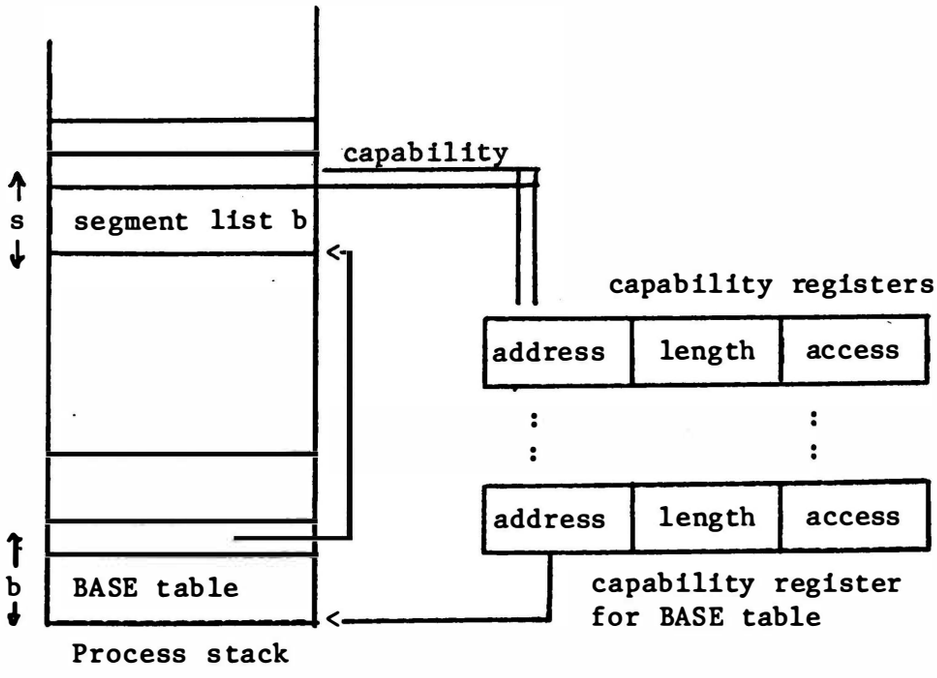


Figure 5.15 - the new MONADS addressing scheme

chapter. In each case, the hardware remained unmodified, and a new load-capability-register instruction was written for the new structure. From the detailed examination of the capability systems in Chapter 4 it would appear that the model could be applied to all of the different C-list structures in the same way as those described in this chapter. In addition, the hardware is not concerned whether the capabilities are taken from a C-list or from tagged memory. Thus, the proposed model could also implement those systems which use tagged memory rather than a segment list to hold capabilities.

Because the capabilities are interpreted by software before they are used to address memory, the scheme does not interfere with the way that the systems manage high level objects. The only requirement that the hardware places on the load-capability-register instruction is that only segment capabilities can be placed in a register. Thus, the model seems to have fulfilled its aim of flexibility.

5.6. Evaluation of the Hardware Model

This section addresses two questions. First, has the model proposed in this chapter fulfilled its primary aims? Second, how does this solution differ from the other capability based computers discussed in Chapter 4?

5.6.1. Model Aims

We can now reconsider the aims of the model, cited in section 5.1.

5.6.1.1. Memory Management

We showed in Chapter 4 that many systems have difficulty in managing a memory addressed by capabilities. The model avoids many of these problems by using the memory management model described in section 5.4. This scheme allows both large and small segments to exist in a paged virtual memory. Large segments may occupy as many pages of real memory as required. Because many small segments may be packed into a page, many related segments may be swapped between main and secondary memory at the same time. Thus, the model has solved many of the memory management problems.

5.6.1.2. Address Translation Problems

In systems which use a central object table for segment address translation, the size of the table may become excessive if the processor addresses many small segments. The model proposed in this chapter avoids this problem by eliminating the object table altogether. Because segments are no longer a unit of main memory, no mapping information needs to be maintained about the segment which cannot be placed safely in the capability for the segment. Moreover, we suggest that the virtual addresses are translated into main memory addresses by an associative address translation system, which can not only efficiently translate large addresses, but also detaches main memory address translation from secondary memory address translation. Thus, the model has solved many of the problems associated with address translation.

5.6.1.3. Uniformity and Simplicity

The scheme is uniform in two respects. First, the capability registers are the only way of addressing store. No extra mechanisms exist which bypass the capability structure. Section 5.5 showed that these registers alone are sufficient to implement a real addressing structure above the hardware. Second, all data, regardless of how large or small it is, or how long it exists, is stored in the virtual memory. No other secondary memory is visible to the programmer.

Because of this uniformity, the overall addressing mechanism is comparatively simple. Only one addressing system is available, and only one protection system is needed. Thus, the model satisfies the aim of uniformity.

5.6.1.4. Efficiency

The efficiency of the solution may be judged by considering two separate functions: (i) the translation of a program address (e.g. of the form C-list index and offset) into a capability, and (ii) the translation of a capability into a main memory address. The first of these is only performed when a capability register is loaded, or when a domain switch is performed, and is dependent upon the access time of the C-lists. Provided that (a) sufficient capability registers are available to contain the working set of the process (Denning, 1968,

1980), (b) capability registers are allocated sensibly, and (c) some hardware support is provided for domain changes, then the cost of loading the registers may be ignored.

The second function depends on two factors. The first is the access time of the registers. This time is usually so small that it may be ignored. The second is the virtual address translation time. This time depends on the translation technique chosen. In Chapter 7 we propose a mechanism which compares favorably with schemes such as MU6-G and IBM System/38. Provided that an associative scheme is chosen, there is no reason in principle why this address translation should be inefficient.

5.6.1.5. Flexibility

The hardware proposed in this chapter was designed to be flexible enough to survive a number of changes in software ideas. In section 5.5 we applied the model to a number of different addressing structures. In each of these the model was capable of implementing a different addressing structure with only a different load-capability-register instruction, and with instructions for the different high level objects. Thus, by example, it appears that the model is sufficiently flexible. The model gains its flexibility from the clear distinction between those functions which must be supported in hardware (purely for efficiency reasons), and those functions which can be implemented by microcode (and are thus easy to change).

5.6.2. Comparison to Other Systems

Whilst consideration of the primary aims has highlighted many differences between the model and other capability based systems, the model can best be compared with other work in this area by considering a number of its features.

5.6.2.1. The Use of Registers

The model is similar to the Plessey 250 and the Chicago magic number computer in the use of capability registers. The model avoids many of the problems associated with the Plessey 250 by using a virtual address in the capability register rather than a real address. No other systems known to the author make use of such registers, or are as flexible as the proposed system.

5.6.2.2. The Capability Format

The format of the model capability is similar to that of Bishop. Both include a virtual address, a length field and access rights. However, the length field of Bishop's capability appears to be too small to allow large objects to be created, or for sufficient addressing granularity. The CAP capabilities differ in address specification but do allow a length field. The model capability is also similar to that of the IBM System/38. However, the System/38 allows store to be addressed without capabilities and provides two different addressing mechanisms. Most of the other systems discussed use an object number as an address rather than a virtual address.

5.6.2.3. Refinement

Whilst all of the capability systems allow the access rights of a capability to be refined, CAP and Bishop are the only systems which allow the bounds to be refined. As discussed in 5.3.4 Bishop's length field is far too small to be of any use. The CAP length field is duplicated in both the capability and the central object table because of the segmented store. This duplication is avoided in our model.

5.6.2.4. Real Store Management

The use of paging vastly simplifies the management of real store. Systems which also use paging are Hydra, IBM System/38, Bishop and Gligor. In Hydra, paging has the effect of restricting the maximum segment size to one page. In the IBM System/38 segments must be at least one page in length, which wastes a great deal of space for small segments. Because Gligor allows segments to be placed arbitrarily in virtual space, there is no guarantee that segments with a common owner are placed within the same page. Bishop's scheme, like our model, guarantees this by using areas, or address spaces. This allows common segments to be swapped in and out of store in one operation. The model differs vastly from schemes such as Plessey 250, Chicago Magic Number Computer, Intel iAPX432 and CAP which use a segmented store.

5.6.2.5. Virtual Store Management

The only schemes which use virtual addresses in a similar way to the model are Gligor and Bishop. We showed in Chapter 4 that both of

these virtual spaces can become fragmented after use. The model avoids this problem by allocating virtual space in fixed size units.

5.6.2.6. Small and Large Segments

We showed in some detail the difficulties experienced by other capability systems in supporting both small and large segments. The model uses a memory organization which allows most of these problems to be avoided or minimized, allowing it to be far more efficient than the systems discussed in Chapter 4.

5.6.2.7. Address Translation

We showed that the model requires an associative address translation scheme to operate efficiently. Two processors which provide such a scheme are MU6-G and the IBM System/38. In Chapter 7 we will propose another associative mapping technique which compares favorably with these two.

5.7. Conclusion

This chapter has defined a hardware model for implementing a capability based addressing scheme. Many theses conclude at this stage without demonstrating the effectiveness of their solution. We were concerned that the model should be implemented to show that the solution is practical. Unfortunately, in achieving such an implementation, we faced two problems.

First, funds had to be found to produce this implementation. Second, time had to be found to produce a working processor. The next chapter proposes a model which allowed a working implementation of the hardware to be built both cheaply and quickly. Chapter 7 then applies this model to the capability addressing scheme to produce a capability based computer system.

6. An Architectural Enhancement Technique

In Chapter 5 we proposed a hardware model which can be used to support many different software structures, particularly those of the MONADS project. It was particularly important that these ideas be implemented as the internal structure of a new processor, and not remain untested. It was also important that the MONADS software group could have a capability based computer system to use for the development of their software structures and ideas.

Many new ideas are often only designed and documented at a conceptual level and are never actually implemented as the basic structure of a new processor, e.g. the Chicago Magic Number Computer (Shepherd, 1968; Yngve, 1968), Gligor (1978), and Bishop (1977). Unfortunately, many major design flaws are not discovered until an attempt is made to implement the design. Moreover, some designs cannot be implemented at all. Thus, a real implementation determines both that the ideas are basically sound and that they can be efficiently built with the available techniques. The problem faced by the author was how to demonstrate the effectiveness of the capability registers, both cheaply and quickly, and still produce a usable computer system.

This chapter comprises three main sections. The first examines some of the standard implementation techniques. The second proposes a hardware enhancement model, and the third demonstrates the model by citing examples of some architectural enhancements.

In the next chapter we describe how the technique was actually used to build the MONADS SERIES II computer system, and to implement the capability registers described in Chapter 5.

6.1. Realizing a New Architecture

A system designer is presented with two alternatives when attempting to implement a new architecture. First, the architecture can be incorporated into a totally new computer system. This approach, whilst logically the more desirable, often involves many more hours than may superficially appear necessary.

Apart from implementing the instructions which pertain to the new architecture, basic arithmetic and logic instructions must also be

implemented. These instructions, while conceptually simple, may occupy a large part of the machine microcode (and/or hardware). For example, to realize a general shift instruction, not only microcode must be written but also some special hardware may need to be built (such as a parallel shifter).

Extra devices (such as interfaces and controllers) must be constructed purely to operate the new processor. Some of these devices may require a large amount of design effort; effort which is not directly connected to the original architectural aims. Many software packages must then be developed, such as assemblers, compilers, loaders and bootstraps.

Consequently, the project often grows in size and large group management problems are encountered. Much of this extra effort appears to be directed to the devices which must communicate with the processor, rather than to the processor itself. Thus, because of the extra effort involved, the full scale production of a new computer simply to test out some architectural enhancements is often not viable in a research environment.

The second alternative consists of modifying or using an existing computer system (called the 'source' architecture) in order to test out a new architectural design (called the 'target' architecture). This approach has the advantage that the design time and effort may be dramatically reduced. Many of the features of the source processor may be inherited, for example the input-output system and the basic instruction set. However, great care must be exercised to prevent the source architecture from restricting the scope and effectiveness of the target.

6.2. Using an Existing Computer System

Three different techniques may be used when the target architecture is constructed on top of a simpler source machine. First, an environment may be constructed in software. Second, if the source processor uses a microcoded control unit, the target may be implemented in firmware. Third, the actual hardware of the source processor may be modified to implement the target architecture.

6.2.1. A Software Emulation

This solution may take a number of forms. The most general is to produce a program (called the interpreter) which interprets instructions for the target machine. The interpreter emulates the fetch-execute cycle of the target processor, and executes target instructions by using small sections of source instructions. Interpreting the new architecture offers many advantages. Because the interpreter is a program, often written in a high level language, it may be easily modified. Complex debugging and monitoring aids may be incorporated in the design, allowing the designers to measure and judge the effectiveness of the new processor. At the same time as emulating the target architecture, the source machine may be executing many other programs.

This approach also has some major disadvantages. The ultimate execution speed of the target processor is often far too slow to support realistic tests . Moreover, it is not always obvious whether efficient hardware can later be constructed, somewhat diminishing the effectiveness of the implementation.

A slightly more efficient software emulation involves another different body of code (called the Kernel) which attempts to provide a normal source machine program with attributes from the target processor. Programs for the target machine are compiled into source machine instructions. When a target machine operation is required which cannot be directly translated into a short sequence of source instructions, a call to the kernel is executed, which performs the task and returns control to the source program.

Whilst far more efficient than an interpreter, the kernel solution tends to highlight the architectural features of both the source processor and the target, often with disastrous effects. (Such an example is found in CAL (Lampson and Sturgis, 1976)). Moreover, this technique may not be able to manage a target machine which is dramatically different in design from the source. Thus, a target program may degenerate mainly to kernel calls and appear the same as an interpretive solution.

Because many source instructions may be required to emulate a target instruction, the speed of the kernel is often far too slow to support a realistic test environment. Many different types of kernel have been written. A good review is found in Rosenberg (1979).

A common disadvantage is that both the kernel and the interpreter often occupy large amounts of memory and may reduce the space available for user programs significantly.

The advantages of these solutions are mostly logical. An interpretive solution can usually emulate the target architecture successfully. The disadvantages are mostly practical. Poor execution speed often makes the model useless.

6.2.2. A Firmware Implementation

Another technique used is to emulate the target architecture in firmware (or microcode). This solution is clearly only applicable if the source machine uses a microcoded control unit and possesses a writable control store.

The internal microcycle of most processors is several times faster than their fetch-execute cycle. Consequently, target machine instructions can be much more efficiently emulated with microcode than with software. Because new instructions can be placed in writable control store, the processor can continue to execute normal source machine programs at the same time as target programs.

Unfortunately, most processors provide only a small writable control store and, more importantly, a limited number of uncommitted operation codes. Thus, it is usually difficult to microcode all of the operations required by the target machine.

Even when sufficient store and entry points are available, this technique often encounters another important problem. Many target instructions may implicitly require storage space, which must be provided by the source machine mainstore. (An obvious example is the implementation of a virtual memory system, which requires page tables in order to translate addresses). In many cases the fact that target operations are implemented in microcode may not be sufficient to make them efficient. The operations may be limited in speed by the time

taken to scan or search various data structures which, if built into hardware, would have used much faster store and searching strategies. (Examples of such an address translation system are found in Belgard (1976), Cohen (1973), Tanenbaum (1979), D'Hautcourt-Carrette (1977) and Sitton and Wear (1974).

In addition, the structure of the micro instruction is usually designed for the source instruction set, not the target. Consequently, it is often quite difficult to write the target microcode on the source machine.

Thus, a firmware emulation, whilst much more efficient than a kernel or interpretive solution, is often still too slow to provide a usable system. Moreover, the implementation often leaves too much of the source processor architecture visible, affecting the attributes and view of the target architecture.

In the situation where speed is important, the only solution may be to provide special hardware.

6.2.3. Modifying the Source Hardware

The third possibility is to modify the hardware of an existing machine. Clearly, this technique can offer the best performance. Traditionally, however, this method is only used when the target architecture does not differ greatly from the source architecture.

Small changes such as small modifications to the instruction set, adding virtual memory hardware (an example is found in Hagan (1977)) and detecting extra error modes (such as those in HYDRA), have been done successfully. Each of these changes, however, has not introduced major architectural enhancements to the source processor.

In fact, it is clear that the major changes possible with an emulation environment are not always possible when modifying the hardware of an existing machine.

The technique is often rejected because it may alter the environment for normal source machine programs as well as target machine machine programs, dedicating the use of the source machine.

In spite of the disadvantages and practical difficulties a number of architectural changes have been achieved by hardware changes. The

next section examines some of the more common hardware modification schemes used.

6.3. Hardware Modifications

Many specific changes are possible when the processor design is modified. These depend upon the internal implementation of the processor itself, and will not be considered further. This section describes one of the most general modification techniques used. This requires an examination of the general structure of many computers.

6.3.1. Processor Configurations

Most computer systems can be divided into two main parts, the CPU and the memory, connected usually by a 'clean' set of interface signals, shown in Figure 6.1.

The signals involved in the interface can typically be divided into three sections; addresses, data and control/handshaking information. The CPU communicates with the memory mostly by read and write commands. When the CPU executes a read operation control information is generated together with an address pattern. The CPU may then wait for data, which is passed back over the data pathway. When a write is executed data is sent with the address to the memory unit

The connections between the CPU and memory section may be generalized to form a system bus which connects to devices other than the memory.

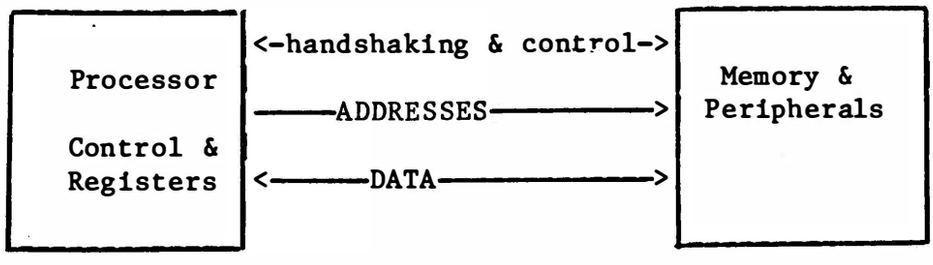


Figure 6.1 - a typical processor configuration

It is the 'clean' nature of the interface between CPU and memory which is often employed when architectural enhancements are introduced.

6.3.2. Breaking the Address Bus

One technique used to enhance the architecture of the source processor is to introduce extra logic into the address pathway between the CPU and the memory, shown in Figure 6.2.

If the architectural enhancement is the addition of a virtual memory system, then the extra logic may be used to modify, or translate, the processor addresses before they reach the memory. Such a system is described in Hagan (1977).

If, however, the target architecture is to include more registers, these may be assigned addresses and placed in the extra logic. Read and write commands directed to these addresses are 'stolen' by the extra logic and may never reach the memory.

The extra logic in some systems appears to the source processor as a block of memory, but the data in the locations is calculated by the logic rather than being the previously saved values. Such a system is described in Wallace (1978) to implement a stack mechanism and addressing registers.

Many systems have been constructed which place special significance upon certain addresses within the address space. Many rely on special addresses for performing I/O operations (such as the PDP11 and VAX computers (Digital Equipment Corp., 1979)). All, however, only 'steal' a limited number of addresses for such operations, and perform very specific operations. None of these systems make dramatic architectural

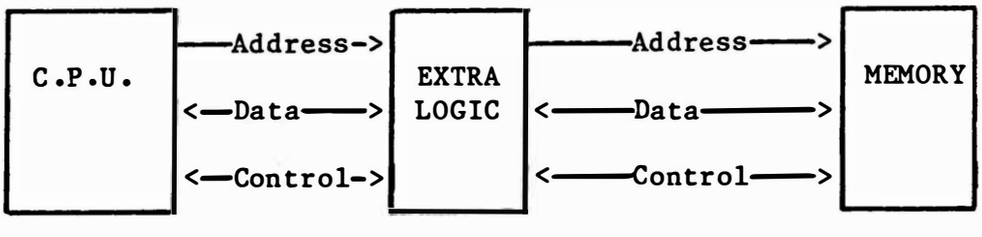


Figure 6.2 - breaking the address bus

changes. Such systems do, however, suggest that treating the addresses from a source processor in a special way may be used as a general mechanism for enhancing an existing machine architecture. The next section proposes such a model.

6.4. An Enhancement Model

The systems discussed in the last section used the processor addresses in various ways. If rather than using a dedicated piece of extra logic, another fast processor is placed in the address path, a general mechanism for dramatic architectural enhancements is created. In such a scheme, the processor addresses are treated as instructions by another, small fast processor, the intermediate processor, as shown in Figure 6.3. These new instructions may be tailored to the target architecture.

The intermediate processor reinterprets all of the CPU addresses, and executes them as though they were instructions. Some may be sent to the memory unit, whilst others may be used internally.

The intermediate processor appears as a piece of memory to the source processor. When a memory reference occurs, the source processor is suspended and the intermediate processor is started. The intermediate processor then executes the function associated with the memory address and return control to the source processor

The model possesses some particularly notable attributes.

i) Many new operation codes are available, thus many new target

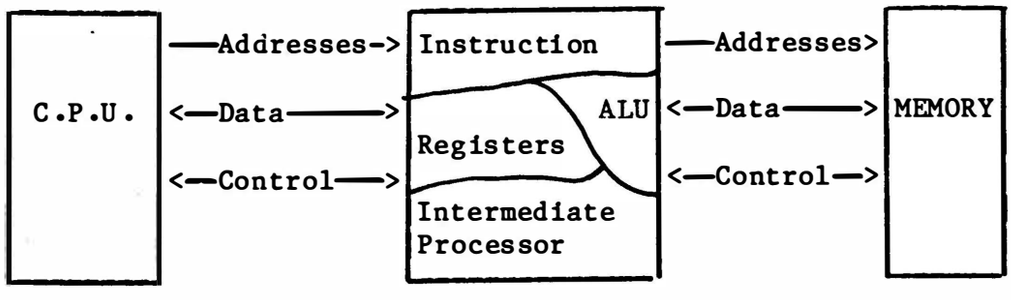


Figure 6.3 - the enhancement model

operations may be supported. The potential number of codes available is the size of the address space.

ii) Because the intermediate processor is a general processor many different target operations may be attempted, from very simple memory references to complex data manipulation.

iii) Extra target architecture registers may be located in the structure of the intermediate processor, and can be manipulated by read and write commands from the source processor.

iv) Normal memory references can be made to proceed from the source processor to the memory with very little delay.

v) Complex target operations may be added to the source without major modifications to the source processor hardware. Thus, the source processor may be a mainframe, a minicomputer or possibly even a microprocessor.

vi) The new architecture is partly transportable among source processors. Most of the target architecture is housed within the intermediate processor itself.

vii) The intermediate processor may be removed, or made logically transparent; thus it is not difficult to allow the source processor to execute normal source programs instead of target machine programs.

viii) The target architecture inherits all of the input/output devices, controllers, communication system, frame and power supplies from the source processor. It also inherits the basic instruction set from the source processor. This vastly reduces the amount of effort required to implement a working target architecture.

ix) Depending upon the address interpretations it may be possible to execute source programs on the new target machine. At the very least, these programs can execute on another source processor of

the same type. Thus the assemblers, compilers and loaders already available for the source processor may be modified to produce code for the target architecture. Consequently, some software development may be avoided.

x) Because the intermediate processor only consists of a central processor unit it may be easily constructed, possibly from bit slice components. This processor is often simpler in design than the source machine as it only implements those features of the target which are new.

The next section will consider the application of this model and give examples of the architectural modifications which are possible.

6.5. Application of the Enhancement Model

6.5.1. Dividing the Address Space into Areas

In order to apply the enhancement model, the address space of the source processor must be divided into areas, and the new target functions must be assigned addresses from an area. Whilst an arbitrary division is allowed, two key logical areas can be identified; the code area and a special area, as shown in Figure 6.4.

Because the fetch phase of the source processor remains unmodified, an area in the address space must be reserved for addresses which are to be interpreted as a region of code. When the source processor issues a fetch in this region, the intermediate processor returns an instruction. The second area can be further divided into the many new functions which the target processor must provide. We will now consider the types of functions that can be provided.

6.5.2. Some Architectural Enhancements

The model is capable of a wide range of enhancements, many of which cannot be achieved by the emulation techniques discussed earlier in this chapter. We shall now consider some examples:

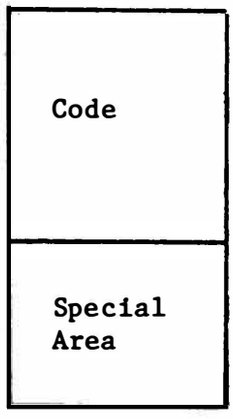


Figure 6.4 - dividing the address space

6.5.2.1. Adding New Registers.

Three classes of registers are often required in a processor: data registers, address registers and status registers. Each new register is held within the intermediate processor, and is assigned an address from the source processor address space. The register may then be loaded from or stored into from the source processor, as though it were actually a word of store. The intermediate processor may treat the register purely as an internal register.

Data registers are the simplest form of register, and usually only require load and store operations. Addressing registers can also be loaded from and stored into by the source processor, but or may be used indirectly to address store. Status registers are typically read from the source processor and loaded from hardware control lines, such as interrupt masks, error flags, timers, etc. Figure 6.5 shows how the registers may be integrated into the address space of the source processor, where Ar is the address of the new register in the address space. Data may move between the source processor and the register via the memory location allocated for the register, and may also move between the intermediate processor via internal data pathways.

6.5.2.2. Adding New Instructions.

New instructions, which are activated when a particular source address is referenced, may be implemented within the intermediate

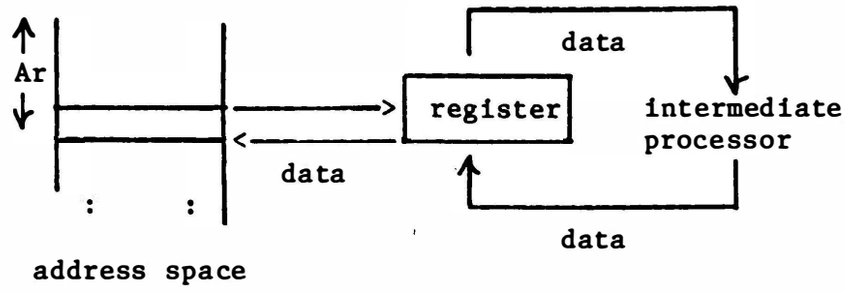


Figure 6.5 - adding new registers

processor. Thus, the source address appears to act as an instruction word for the intermediate processor. A range of addresses may be allocated, all of which activate the same intermediate instruction, but which use some bits from the source address to specify an operand. This scheme is shown in Figure 6.6. In this diagram Ai is the address of the new instruction, and the constant n can define a frame of addresses relative to the instruction in the address space. The instruction address may then be converted into a microcode entry point in the intermediate processor, which defines code to interpret the new instruction. Examples of instructions are:

- An instruction which manipulates some of the intermediate processor registers, e.g. add 1 to register n. This type of instruction is totally executed within the intermediate processor.

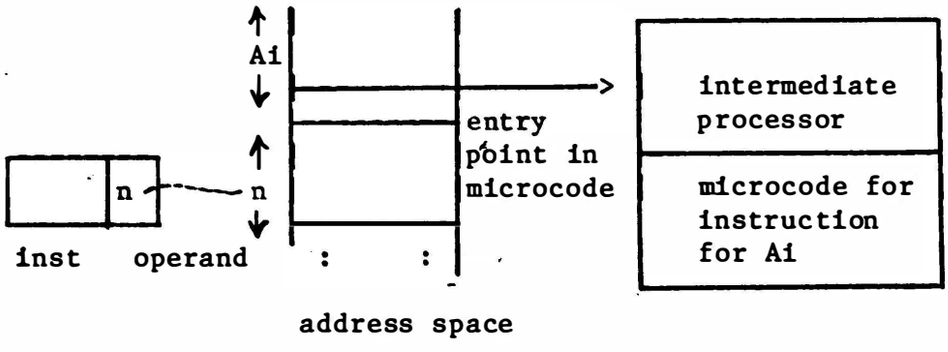


Figure 6.6 - adding a new instruction

- A long move instruction, which uses two addressing registers, and a counter register, and moves data around the store. This instruction iteratively addresses store until the block is moved.

- A context switch instruction, which changes processes within the intermediate processor.

6.5.2.3. Adding New Addressing Modes.

The basic addressing modes of the source processor may be augmented by new modes, provided within the intermediate processor. Some examples are:

Indexing.

Certain addresses within the address space may cause a mainstore address to be calculated from a combination of addressing registers. When the location is referenced, the intermediate processor may calculate a store address, retrieve the data, and return it to the source processor. This dynamic address calculation may be used to provide an index mode, which may not be present in the source architecture. An example of index mode addressing is shown in Figure 6.7. In this diagram Aa defines the location of the new addressing mode in the address space. If this location is referenced then the intermediate processor forms a main memory address by adding the contents of an addressing register and a modifier register together. This effective address is then used to reference store.

Stacks.

Certain modes may use an addressing register to reference store, and then modify the contents of the register. This operation could provide push and pop instructions. Stack frames may also be defined relative to a stack register, by reserving a number of locations within the source address space. An example is shown in Figure 6.8. Separate addresses are assigned for push and pop operations. If either of these addresses is referenced the value of the stack pointer register is modified, and a main memory address is generated. If a frame relative to a frame pointer is required, the constant n is added to the value in

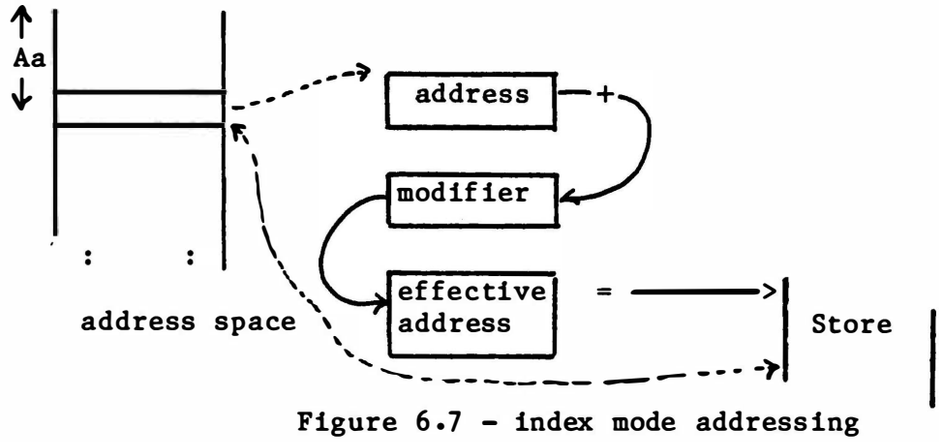


Figure 6.7 - index mode addressing

this register in order to form a main memory address.

Constants.

If part of the source processor address is returned as data, by the intermediate processor, then a number of constants may be referenced without reading the mainstore. This mode of addressing is often called immediate mode. An example is shown in Figure 6.9. The intermediate processor returns the value i when the location Ai is read.

6.5.2.4. Adding a Virtual Memory.

Because the source processor addresses are isolated from the mainstore, the intermediate processor can develop virtual rather than

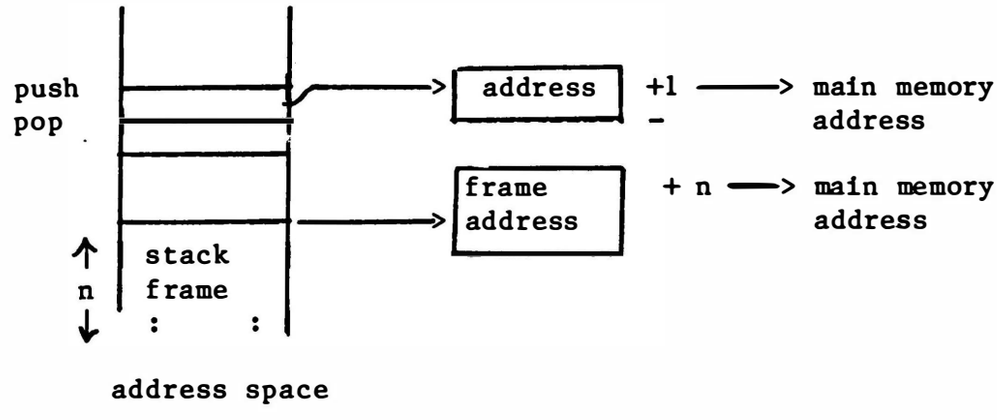


Figure 6.8 - stack addressing

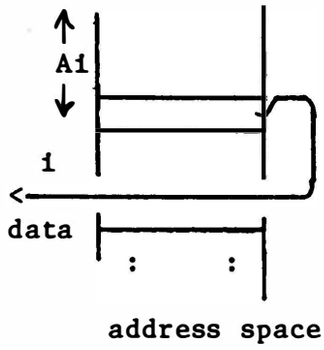


Figure 6.9 - constant addressing

real addresses. Special translation hardware may then be placed between the intermediate processor and the memory subsystem.

6.5.2.5. Expanding the Address Size.

The size of the addressing registers within the intermediate processor may be many times the size of the source processor address. Thus, the effective address space size of the target architecture may be much larger than that of the source processor.

6.5.2.6. Detecting Errors.

The intermediate processor may detect many error conditions, e.g. removing too many items from a stack, addressing beyond the top of a stack, addressing memory which is protected, etc. These may then be reported to the source processor. Control registers may be used to describe the nature of the error.

6.6. Conclusions

In this chapter we have developed a general mechanism for expanding the power of an existing computer. The solution is both cheap and efficient. By considering some examples we have shown that the model is, at least theoretically, realistic.

The next chapter will use this technique to expand the architecture of a very simple mini-computer and, at the same time, implement the

addressing structure proposed in Chapter 5.

7. The MONADS SERIES II System - An Implementation

Chapter 6 described a technique for enhancing the architecture of a primitive source processor. In this chapter we show how the enhancement model has been applied to the implementation of a capability based computer system according to the design proposed in Chapter 5, using a primitive source minicomputer, an HP2100A (Hewlett Packard, 1972).

Section 1 defines the aims of this system, which is known as the MONADS Series II computer. Section 2 describes the HP2100A source processor hardware. Section 3 describes the intermediate processor developed to expand the HP2100A. Section 4 defines the MONADS II address translation hardware, and compares it to other similar schemes. Section 5 comments on the modifications made to the HP2100A processor. Section 6 describes the software packages developed during the construction of the MONADS II system. The chapter concludes by demonstrating that the MONADS II computer system fulfils its primary aims.

7.1. The MONADS SERIES II System - Primary Aims

The MONADS II hardware has a number of major aims:

(1) To demonstrate that the capability register addressing scheme, proposed in Chapter 5, is realistic and can be efficiently implemented. This aim is tested by using the architectural model as the centre of the MONADS II addressing structure.

(2) To provide a pilot system for future software development work on the MONADS project. Because of time and fiscal constraints it was not possible to produce a computer utility with the speed and power of large mainframe computer systems. However, the MONADS II system is suitably scaled down so that it can still support a number of users, each developing software modules. The processor is considered a testbed for both the hardware and the software ideas.

(3) To demonstrate that the enhancement technique proposed in Chapter 6 is both practical and powerful in scope.

(4) To provide supporting hardware for the addressing structure described in Chapter 5, in the form of a hardware address translation scheme.

The MONADS Series II system is composed of a number of key components, as shown in Figure 7.1. The system is constructed around a HP2100A minicomputer which provides all of the basic computing facilities, such as a standard instruction set, and an input-output system. All of the complex addressing modes which are required by the MONADS architecture are provided by the intermediate processor. This unit develops a 31 bit virtual address, which is translated into a main memory address by the virtual memory manager. The current configuration is connected to 400k bytes of semi-conductor memory. The rest of this chapter will examine these components in detail, and show how the entire system fulfils these aims.

7.2. The HP2100A Processor

The HP2100A is typical of many 16 bit minicomputers of the same era, and incorporates a microcoded control unit, two general purpose accumulators and 32k words of memory. Processor addresses are constructed from 16 bit words, 15 bits of which form the memory address. The top (most significant) bit determines whether addresses are direct memory addresses, or are part of a chain of indirect addresses.

7.2.1. The View of Memory

With its 15 bit addresses, the HP2100A can address up to 32k 16 bit words of core memory. This address space is divided into 32 1k word 'leaves' [1]. Thus, the memory address is logically composed of a 5 bit leaf number, and a 10 bit within leaf displacement. The leaf with a number of zero is called the base leaf, and the leaf number in which an instruction resides is called the current leaf.

---

[1] The HP2100A literature refers to leaves as pages. However, we have adopted the present terminology, preferring to use the term 'page' as a unit of virtual memory transfers.

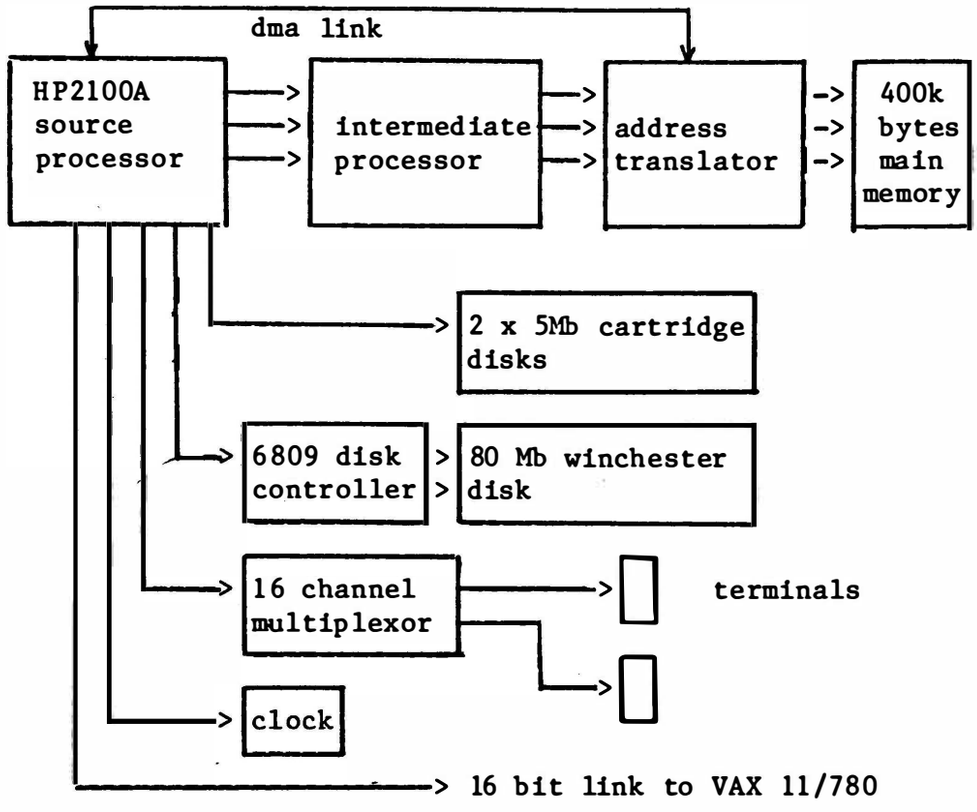


Figure 7.1 - the MONADS Series II computer system

If a memory address is used directly, the contents of the location are treated as data. However, if the address is used indirectly, then the contents of the location are treated as an address. In general, arbitrarily long indirect address chains may be created in memory.

7.2.2. The Instruction Format

Instructions are divided into two main classes, the memory reference group and the register and I/O group. Most instructions are held in 16 bits. The format of the memory reference instructions are as follows:

bit 15	indirect bit
bits 14 - 11	function code
bit 10	base or current leaf bit
bits 9 - 0	within leaf displacement

The function code specifies which operation the instruction is to perform. (Valid functions are load, store, or, and, add, compare, increment, jump, jump subroutine and exclusive or). Some functions may be applied to either of the processor accumulators. The instruction operand is specified by the address field. Because this field is only 10 bits in length, the instruction can only address one leaf of store. The base leaf or current leaf bit determines whether the 10 bit address is used within the base leaf or the leaf in which the instruction resides.

To allow an instruction to reference all of the store, an address may be placed in a word of memory (called a link) and used indirectly, by setting the indirect bit of the instruction. The processor will follow an indirect chain of addresses until a word is found with a zero top bit. The last address in the chain defines the effective address of the operand. The use of 10 bit address fields allows a program to reference most of its data with 16 bit instructions and to use links only when the data is not in the base or current leaf.

7.2.3. The Input-Output (I/O) System

The HP2100A processor supports a primitive input-output system which allows a program to communicate with any of 64 devices (although some of these have special meaning). The bottom 6 bits of an I/O instruction specify which device the instruction is addressing. Transfers of 16 bit data words may be directed to or from a device under program control.

7.2.4. The Direct Memory Access System (DMA)

Certain devices, such as disks, require interword service times faster than a programmed I/O loop can operate. To communicate with these devices, the processor provides two autonomous direct memory access channels. Each channel, once set up, can transfer a block of data between memory and a device without processor intervention. The DMA system operates by stealing cycles from the processor when it requires

attention.

7.2.5. The Control System

The HP2100A is controlled by a micro-programmed state machine. The processor can be equipped with 256 x 24 bit words of basic instruction set, 256 words of extended (floating point) instructions, and up to 512 words of writable control store. The writable control store appears as part of the I/O system, and can be read from or written to under program control.

7.2.6. Interrupts

The HP2100A supports a vectored interrupt system. When an interrupt occurs, control is transferred to one of 64 base leaf memory locations, any of which can then transfer control to an interrupt service routine. Interrupts are normally processed at the end of an HP2100A instruction.

7.3. The Intermediate Processor

This section describes the main features of the intermediate processor designed and implemented by the author to extend the functionality of the basic HP2100A.

7.3.1. Functionality

7.3.1.1. Privilege Modes

In order to protect sensitive information within the intermediate processor, the hardware may operate in one of two modes: kernel mode or user mode.

In kernel mode two conditions are created. First, all code is fetched from a special code address space, which holds the kernel program. Second, the HP2100A may modify any of the intermediate processor registers. Kernel mode may be entered in one of two well defined ways. First, every interrupt causes the processor to enter kernel mode. This is necessary because the kernel software contains the interrupt service routines. Second, a special HP2100 micro instruction can set the processor into and out of kernel mode. Thus HP2100A instructions may enter kernel mode to address privileged registers.

In user mode code is fetched from the current software subsystem, and only certain intermediate processor registers may be addressed.

7.3.1.2. Addressing Structure

The intermediate processor enhances the HP2100A architecture in the following ways. First, the single 32k address space in extended into a virtual space of 2^31 words. This consists of 2^16 separate address spaces, in the sense described in Chapter 5, each of 32k words. While a full scale capability system would ideally require more and larger address spaces (e.g. 2^32 by 2^32), the MONADS II addressing range is sufficient to demonstrate the principles involved and to support a pilot system.

Second, the intermediate processor supports 16 sets of new registers, and so can efficiently support process-switching between 16 processes. Each register set includes 16 standard capability registers, six special capability registers intended to address code, constants, base leaf links, and scalars on the stack (there are three registers for this task), eight modifier registers for addressing relative to capability registers, and eight associated counter registers which can be used for efficient loop control. In addition, various control registers are provided to support timers, interrupt handling, etc.

7.3.1.2.1. The Capability Registers

The intermediate processor provides each of the 16 processes executing on the HP2100A with 16 capability registers for addressing segments of memory. Each register is composed of 4 16 bit words, as follows:

word 1: Address space number - 16 bits
word 2: Displacement within address - 15 bits
word 3: Length of segment - 16 bits
word 4: Access bits - read, write, kernel, invalid - 4 bits

The address space number defines one of the 64k byte addressing regions in virtual memory. The displacement is used to mark the start of the segment in the address space. The length field marks the end of the segment in the address space. The read and write bits determine whether

the segment may be read from or written into. The kernel bit specifies that the segment may only be addressed if the processor is in kernel mode. The invalid bit prevents the register from being used, and is set when a register is uninitialized. A capability register can only be loaded when the processor is in kernel mode (e.g. executing a load capability register instruction) and thus its contents are protected from corruption. Because the HP2100A only has 16 bit data pathways, four write cycles are required to set up each register. The HP2100A microcode provides a load capability register instruction of the type discussed in Chapter 5.

A capability register may be used as an operand of any of the HP2100A memory reference instructions. When used, the 31 bit address is treated as a paged virtual address. The displacement field is checked against the length field, and an interrupt is sent to the HP2100A if a violation occurs. If the mode of access is contrary to the read or write bits, or the kernel bit is set and the processor is not in kernel mode, or a register is invalid, an interrupt is sent to the HP2100A.

The displacement held in the register may be modified by two different methods. In the first, a small constant offset in the range 0 - 7 may be dynamically added to the value in the register. Alternatively a value held in a modifier register can be used to index into a segment defined by a capability register.

7.3.1.2.2. The Modifier Registers

The intermediate processor provides each process with eight modifier registers. A modifier may be combined with a capability register to dynamically address data relative to the capability. The modifier may be treated as containing either a word offset or a byte offset. If a word offset is specified, the contents of the displacement field of the capability is added to the modifier and, subject to checks on the length field and the access field, a word of data is referenced. If a byte offset is specified, the modifier is converted to a word offset and the designated byte is transferred to or from store.

Modifier registers are particularly useful for searching and scanning through segments of store. A set of associated counter registers assists in counting loop iterations.

7.3.1.2.3. The Counter Registers

Associated with each of the 8 modifier registers is a counter register. These registers may be set to an initial value and used as loop control registers. Special instructions are provided by the intermediate processor for manipulating a modifier and associated counter as follows:

(1) set the counter register to a value
set the modifier to zero.

This instruction is useful for initializing a loop counter.

(2) add 1 to a counter register
add 1 to a modifier register
return the contents of the counter.

This instruction may be used for keeping track of the number of loop iterations performed whilst stepping through a segment.

7.3.1.2.4. Extra Capability Registers

Special capability registers are provided for addressing code, a frame of scalers on the stack, a set of constants and the HP2100A address links. These items are addressed by extra capability registers, rather than the 16 general capability registers, because of peculiarities of the HP2100 processor and for efficiency reasons, and consequently differ slightly in format to the general capability registers.

The register used for addressing code is formed by the concatenation of the HP2100A program counter with a code address space register. The code register differs from the capability registers by not checking the bounds of the reference and also by not checking the access rights. When the processor was first built, the software group associated with the MONADS project decided that all the code of a subsystem should reside in a large single unit within the code address space. Consequently, bounds checking was not required, and could be safely removed. Subsequent work has revealed that this is not satisfactory, and that code should be constructed from protected segments. In the new scheme, the code address space register would be

formed from one of the general capability registers. Similarly, because no mechanism allows a program to write to its own code address space, there is no need to validate the access rights. Thus, code is not addressed by the capability registers more for historical reasons than any logical reasons.

Three capability registers are used for addressing the process stack; one defines the top of the computational area, and the other two define local variable stack frames. Whilst logically the same format as the general capability registers, these three registers differ slightly in physical format. Because all require the same address space number, namely the current stack address space number, they may share the one register.

Another capability register allows a program to directly address up to 512 constants without the need for a modifier register. Because only a small constant offset may be specified from a capability register (i.e. 0-7), modifier registers must often be used to address scalers in large segments. The offset of the scaler can be held in the constants segment, which can then be addressed via the special constant capability register. Because of the special nature of these constants, a full capability register is not required. In a processor which allowed a larger constant offset relative to the start of a segment this register would not be required, and is only needed because of the small addressing range of the HP2100A source processor.

The last special capability register allows a program to address up to 512 address links. These links are required by the HP2100A to address all of its 32k word address space, and are usually only needed when an instruction wishes to transfer control out of the leaf in which it resides. Because of their special significance, and the way that they are addressed, a general capability register is not required. In a processor which did not require address links this register would not be needed.

7.3.1.2.5. Summary

The intermediate processor allows a program to address the virtual space by either the 16 capability registers or the six extra capability registers. No other addressing mechanisms exist. The six extra

addressing registers differ only in physical format from the 16 capability registers, mostly because of the addressing restrictions of the HP2100. Accordingly, the intermediate processor may be considered a real implementation of the model cited in Chapter 5. The processor also possesses a number of other features which assist the MONADS software, which we will now describe.

7.3.1.3. Process Changes

All of the registers described so far are held in an internal register file of the intermediate processor. Most of these registers pertain to a particular process.

In an operating system which applies the in-process technique even to job management (Ramamohanarao, 1980), such as the MONADS system, the number of processes present at any time is quite small, as no other system processes exist. Consequently, the MONADS II hardware currently provides 16 sets of registers (although this number is easily expanded). When a process switch occurs, an intermediate processor instruction switches all of the registers, allowing very efficient process changes to be executed.

7.3.1.4. The Kernel

Embedded in the intermediate processor is support for the MONADS hardware kernel (Rosenberg, 1979; Wallis, 1980). This body of code is responsible for providing high level support functions for the software, for managing I/O functions, for memory management and for responding to interrupts. The kernel code is activated when the processor enters kernel mode. The intermediate processor assumes that the kernel code is held in address space number 1, and a software convention dedicates process register set zero to the kernel.

7.3.1.5. Control Registers

The intermediate processor includes a number of other registers which are required by the operating system. A mask registers returns the cause of the last violation interrupt. A number of time registers provide the time since bootstrap and process time limits. A register which counts the number of instructions executed assists in monitoring process behaviour.

7.3.1.6. Additional Features

Whilst we have now described the most important features of the intermediate processor, a number of other support features are also provided. These are described in detail in Appendix A. In addition, a large amount of HP2100A microcode provides instructions, including load capability register, and these are discussed in Wallis (1980).

7.3.2. Address Mapping

It will be recalled from Chapter 6 that the technique proposed in the processor enhancement model for addressing extensions to the source hardware (e.g. capability registers) involves setting aside certain addresses in the source processor's address space which are reinterpreted by the intermediate processor.

The address mapping calculations are performed on a 16 bit HP2100A address, constructed from the 15 bits word address and an indirect bit as the 16th bit. The division chosen is as follows:

0 - 777b	frame relative to constant capability register
1000b - 1377b	frame relative to stack capability register 1
1400b - 2000b	frame relative to stack capability register 2
2000b - 76777b	code space, relative to code capability register
76000b - 77777b	special control leaf. This leaf contains access pathways to all of the MONADS II registers and addressing modes
100000b - 100777b	frame relative to links capability register
101000b - 177777b	indirect forms of addresses 1000b - 77777b

Note: The character 'b' denotes the use of the octal number system.

Using this allocation, the HP2100A memory reference instructions can easily address the constants, links and stack frames by setting the base leaf bit to zero. To simplify addressing the special control leaf, the HP2100A has been modified so that any instruction with the current

leaf bit set, except the jump instruction, produces an address in the special control leaf rather than the current leaf. Thus, the base/current leaf bit of the instruction is reinterpreted as a base or special leaf bit. Because it is not sensible for data instructions to address the code space, the current leaf mode is only required for jump instructions. In addition, the special control leaf, like any leaf, may be addressed via a link word.

The interpretations of the addresses within the special control leaf vary greatly, and are found in Appendix B. Access to all the capability registers, modifiers, counters and control registers is gained through this leaf. In addition, access to the memory via the capability registers and stack registers may be gained through this leaf. The next section examines the implementation of the intermediate processor.

7.3.3. Implementation Details

7.3.3.1. The Intermediate Processor Bus Structure

The intermediate processor is based around a 16 bit bi-directional data bus, as shown in Figure 7.2. Attached to the bus are a number of units, namely:

(1) Various individual dedicated registers (shaded lines in Figure 7.2)

(2) A high speed arithmetic unit and accumulator (shaded dots in Figure 7.2)

(3) A register file (unshaded in Figure 7.2)

Units may claim the bus for the duration of one microcycle, and either read a 16 bit pattern from the bus or place a 16 bit pattern on the bus. The 32 bit registers are implemented as two, individually addressable, 16 bit registers. The next section examines the role of each of the dedicated registers.

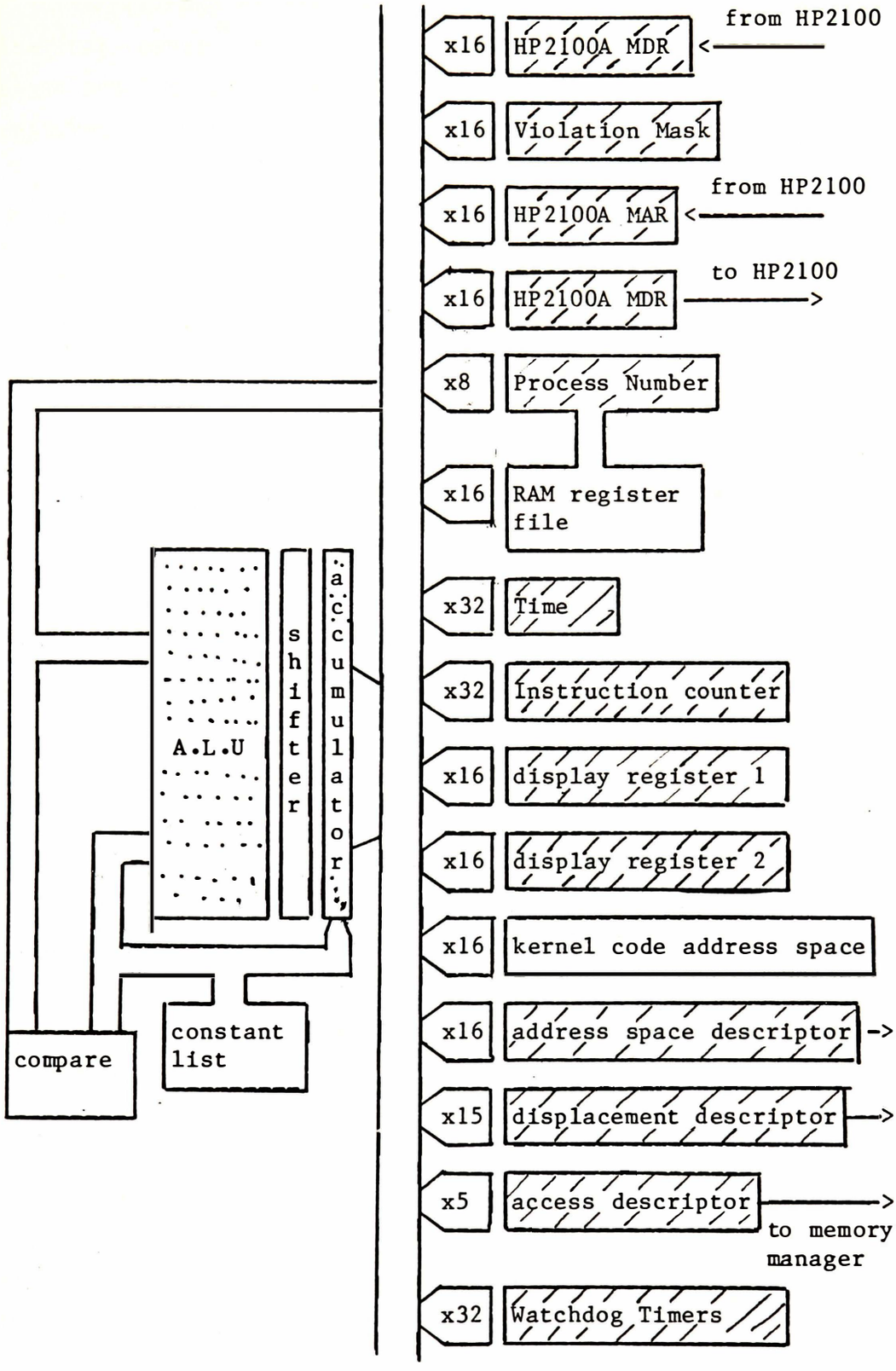


Figure 7.2 - the intermediate processor structure

7.3.3.2. The Dedicated Registers

Unlike many of the registers of the MONADS II system, certain internal registers require dedicated hardware support. Consequently, these are not held in fast register memory, but are allocated individual registers. Fifteen such registers exist, namely:

(1) An address space descriptor

(2) A displacement descriptor

(3) An access control descriptor

(4) Two watchdog timer registers

(5) Two instruction counter registers

(6) Two display registers

(7) Two time registers

(8) A process number register

(9) The HP2100A memory address register

(10) The HP2100A memory data register

(11) A violation mask register

7.3.3.2.1. The Descriptor Registers

The first three registers, the address space descriptor, the displacement descriptor and the access descriptor, are used for addressing the virtual memory. They can be loaded with the contents of a capability register (held in the register file) and are always available for the memory manager.

When a memory reference is requested, the address space descriptor and displacement descriptor are concatenated to form a 31 bit virtual address. At the same time, the bit pattern held in the access register is validated against the mode of access. The bit set of the access register is identical to that of the access field of a capability register, and thus includes read, write, kernel and invalid access bits.

7.3.3.2.2. The Watchdog Timer Registers

The watchdog timer registers are concatenated to form a 31 bit timer register. If the most significant bit is clear, the timer decrements its value every millisecond, until zero. When a zero value is reached, an interrupt is sent to the HP2100A. If the most significant bit of the timer is set, then the count is inhibited. This register is used to alert the operating system when a process has exhausted its time allocation.

7.3.3.2.3. The Instruction Counters

The two instruction counter registers are concatenated to form a 32 bit instruction count. Each time the HP2100A enters a fetch instruction phase the counter is incremented. This counter is useful for monitoring process behaviour.

7.3.3.2.4. The Display Registers

The display registers allow the intermediate processor to display 16 bit values, in octal, on the front panel of the processor. In addition, one of the display registers may act as an index register into the register file.

7.3.3.2.5. The Time Registers

The time register displays the number of milliseconds since the internal processor was initialized. This 32 bit count allows programs to accurately time events.

7.3.3.2.6. The Process Number Register

The process number register is used to select which bank of the register file is made visible. A context switch consists mainly of changing the value held in this register.

7.3.3.2.7. The HP2100A Memory Address Register

This register is held within the HP2100A and is used to determine which function should be executed by the intermediate processor. In addition, the register contents may be placed on the internal bus.

7.3.3.2.8. The HP2100A Memory Data Register

Like the memory address register, the memory data register is held in the HP2100A, and may be loaded or read by the intermediate processor. It is this register which forms the data communication path between the HP2100A and the intermediate processor.

7.3.3.2.9. The Violation Register

The violation register holds a bit map in which each bit indicates the cause of an interrupt, which may be examined by the operating system kernel.

7.3.3.3. The High Speed Arithmetic Unit and Accumulator

The arithmetic and logic unit (ALU) is attached to the bus, and allows high speed arithmetic and logic operations to be performed between two inputs. One of the ALU inputs is permanently tied to the bus, whilst the other may either be connected to the accumulator, or one of seven predefined constants. The output of the ALU is returned to the accumulator via a shifter. The value of the accumulator may later be placed onto the bus.

A comparator is always actively comparing the two inputs of the ALU. The ALU is capable of ADD, SUBTRACT, AND, OR operations and of LEFT or RIGHT shifts.

7.3.3.4. The Register File

Most of the registers described in section 7.3.1 (for example the capability registers, counter registers, etc) are held in the register file. When a register value is requested, the appropriate entry is read, and the data is placed onto the bus. Those registers which require hardware assistance have their data copied from the register file into the dedicated registers.

Each bank of registers holds 128 x 16 bit values. When the process number register is altered, a different bank in the file is made visible. When a context change is executed, those register values held in the dedicated 15 registers are copied back to the file. Because the 128 process own registers are only logically switched, rather than physically moved, very fast process switches are possible.

7.3.3.5. The Control Unit

The intermediate processor is controlled by a micro-programmed state machine. The control store consists of 1024 x 24 bit micro-instruction words, 512 of which are devoted to implementing the MONADS II instruction set and 512 for debugging and diagnostic instructions.

When the HP2100A issues a memory request, the HP2100A memory address register is mapped into a microcode entry point value, and a stream of microcode is executed. When an end-of-instruction micro-instruction is executed control is returned to the HP2100A.

Each micro-instruction is composed of 7 fields for controlling bus activity, ALU function and interrupt generation. The format is described in Appendix C, along with the MONADS II microcode listings.

7.3.3.6. Summary

Section 7.3 has described the functionality and structure of the intermediate processor. Further details may be found in the appendices.

7.4. The Memory Manager

The intermediate processor develops a 31 bit virtual address, which must be translated into a main memory address. This task is performed by the MONADS address translation hardware, which we now describe.

7.4.1. Functionality

7.4.1.1. Nature of the Problem

In Chapter 3 we described the address translation mechanisms commonly used for mapping paged (or paged and segmented) virtual addresses onto paged main memories. The schemes were divided into four categories:-

1 - Systems with small virtual memories
2 - Systems with small main memories
3 - Systems with large virtual memories
4 - Systems with very large virtual memories

The MONADS II system, like other capability based architectures, belongs to the fourth category. In Chapter 4 we explained why the

conventional solutions to category 4 systems cannot be used in capability based addressing schemes. Those solutions which are effective, however, are associative address translation mechanisms, such as those of MU6-G (although this machine is not capability based) and the IBM System/38. Consequently, the MONADS II system also uses an associative translation mechanism, but employs a different implementation technique to the MU6 system and the IBM System/38.

7.4.1.2. Aims of the MONADS II Address Translation

The MONADS II address translator has been designed to fulfil a number of aims, as follows:-

- The mechanism used must employ an associative technique, for reasons explained in Chapters 3 and 5.

- The unit must only hold entries for those pages of virtual memory actually present in main store. This criterion reduces the number of entries required and makes the table size proportional to the size of the main memory.

- The unit should indicate a page fault for any page not present in memory.

- The unit must be fast.

- The unit should be self contained and not require software support for translating addresses. This is necessary so that the limited power of the HP2100 source engine is not wasted on address translation.

- Operating and loading the unit should be well defined and easy to execute in software, again to avoid wasting the power of the HP2100.

A large associative memory is capable of achieving all of these aims. However, such memory is not currently available. The MONADS II scheme attempts to emulate the functions of an associative memory and thus fulfil these primary aims.

7.4.1.3. The MONADS II Address Translation Hardware

The MONADS II virtual address can be interpreted as being composed of three fields: an address space number, a page number and a within page displacement. The address space and page numbers are concatenated to form a virtual page number, which must be translated into a main memory page number. The displacement is removed from the virtual address and forms part of the main memory address. Any address is either translated into a physical memory address or causes a page fault interrupt to occur.

Figure 7.3 shows the logical organization of the address translator. A high speed sparsely occupied hash table with embedded overflow chains is used to emulate a large associative memory. Each hash table entry consists of a valid field, a virtual page number field, a main memory page number field, and a link field.

During the translation of an address, the address space and page numbers are hashed to produce a uniformly distributed hash table cell address. The virtual page number field of the hash table entry is compared with the virtual page number being translated. If they are equal and if the cell is valid, the main memory page number is used to address main store. The link field is used to form a list of all virtual addresses which hash to the same cell. The link field is followed until the virtual address being searched for is found, or an

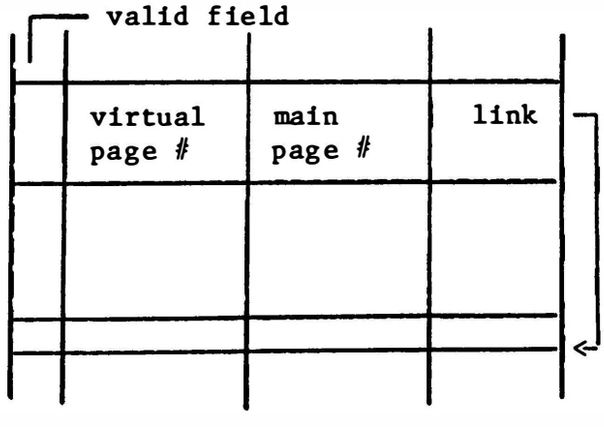


Figure 7.3 - the address translator

end of chain is found. We will show later that providing the hash table is sparsely occupied, i.e. has more cells than there are pages of physical memory, this hash table structure emulates a large associative memory very efficiently.

The retrieval algorithm is the simplest to implement, and in the MONADS II system it is implemented entirely in hardware. Consequently the address translator can map addresses very efficiently. The insertion and deletion algorithms are more complex and are implemented in kernel software. We will now describe the retrieval, insertion, and deletion algorithms in detail.

7.4.1.4. Retrieval

Retrieval of a mapping cell is performed by the address translation hardware, and conforms to the algorithm provided in Figure 7.4.

A cell is used providing that it is valid and the virtual page number in the cell corresponds to the page number being translated. An overflow chain is only followed if (a) the cell is valid, (b) an overflow chain is present, and (c) the list pertains to the cell itself (i.e. it is not simply part of another list). Condition (c) is detected by hashing the virtual page number of the cell and validating it against the cell address. If a virtual address is not found in the hash table, then a page fault interrupt is sent to the processor.

7.4.1.5. Insertion

Insertion of a page mapping entry into the hash table is performed when a page is brought into main store, and is handled by the kernel software. The algorithm, shown in Figure 7.5, is slightly more complex than that of retrieval, and is divided into three cases. First, a mapping entry is being inserted into an empty cell. In this case the cell is loaded with the mapping information, made valid and the overflow chain terminated. Second, a mapping entry is being inserted into a cell which has an overflow chain. In this case, the entry is placed in a free cell of the table, and is chained into the second position of the current overflow chain. Third, a mapping entry is being inserted into a cell which is part of another overflow chain. In this case, the cell is used in the same way as the first case, but the old contents of the cell

```
{form of the hash table}

 type tableentry = record of
          begin
          vpn:    virtual-page-number;
          valid:  boolean;
          rpn:    real-page-number;
          ov:     link-field
          end;

 var:

          vpn:   virtual-page-number;
          rpn:   real-page-number;
          table: array[1..size] of tableentry;

begin

     i:= hash(vpn);
     j:= hash(table[i].vpn);

     if     j <> i or
        not table[i].valid  then  error('page fault');

     while    table[i].vpn <> vpn  and
              table[i].ov <> nil
        do    i:= table[i].ov;

     if vpn = table[i].vpn
                    then rpn:= table[i].rpn
                    else error('page fault);


end.
```

Figure 7.4 - the retrieval algorithm

are placed in another free cell of the table. The overflow chain of the foreign cell is then updated to point to the new free cell.

7.4.1.6. Deletion Algorithm

A map entry is deleted from the hash table when a page is removed from main store, and is again handled by the kernel software. The algorithm is divided into three cases, shown in Figure 7.6. First, the entry being deleted is in the correct cell of the table and has no overflow chain. In this case the cell is made invalid. Second, the entry being deleted is in the correct cell but has an overflow list. In

```
begin

   i:= hash(vpn);
   j:= hash(table[i].vpn)

   if i <> j and table[i].valid
       then  begin    {a foreigner is in home cell}
                      {insert new page and banish foreigner}
                      {category 3}
        while i <> j   begin lastj := j; j := table[j].ov end;
        {banish j}
        new := freecell;
        table[new].vpn := table[i].vpn;
        table[new].rpn := table[i].rpn;
        table[new].ov  := table[i].ov;
        table[new].valid := true;
        table[lastj].ov := new;
        {load in new page}
        table[i].vpn := vpn;
        table[i].rpn := rpn;
        table[i].ov  := nil
        end
        else
   if table[i].valid
        then begin {chain in after this cell}
                   {category 2}
             new := freecell;
             table[new].vpn := vpn;
             table[new].rpn := rpn;
             table[new].ov  := table[i].ov
             table[new].valid := true;
             table[i].ov := new
             end
        else
             begin {cell is invalid}
                   {category 1}
             table[i].vpn := vpn;
             table[i].rpn := rpn;
             table[i].ov  := nil;
             table[i].valid := true
             end;
end;
```

Figure 7.5 - the insertion algorithm

```
begin
   home := hash(vpn);
   i    := home;
   j    := hash(table[i].vpn);

   if i <> j or not table[i].valid then error ('not present');

   {try to find the page in the table}

   while  table[i].vpn <> vpn and
          table[i].ov  <> nil   do
                begin lasti := i; i:= table[i].ov end;

   if table[i].vpn <> vpn then error ('not present')
         else begin {page is found}
              if i <> home then   {page is part of list}
                                  {category 3}
                              begin
                              table[lasti].ov := table[i].ov;
                              table[i].valid  := false
                              end
                         else begin  {page is in home cell}
                              k := table[i].ov;
                              if k <> nil then  {there is a list}
                                                {category 2}
                                     begin
                                     table[home].vpn := table[k].vpn;
                                     table[home].rpn := table[k].rpn;
                                     table[home].ov := table[k].ov;
                                     table[k].valid := false
                                     end
                                     else  {category 1}
                                      table[home].valid := false
                               end
                end

end.
```

Figure 7.6 - the deletion algorithm

this case, the next entry of the list is copied into the head of the list, and the old cell made invalid. Third, the entry is found within an overflow chain. In this case, the cell is made invalid and is removed from the chain. The cell which previously pointed to the deleted cell is changed to point around the cell.

7.4.2. Implementation Details

The address translator is built as a stand alone piece of hardware. The interface consists of an incoming virtual address, an outgoing main memory address and a page fault signal, shown in Figure 7.7.

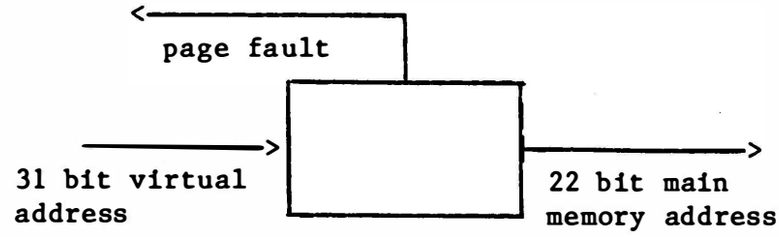


Figure 7.7 - the virtual address translator

7.4.2.1. Internal Structure

The hardware consists of three distinct components, a hashing unit, the hash table and a comparator, shown in Figure 7.8. These three areas are controlled by a fast finite state machine, which performs the retrieval algorithm.

7.4.2.2. Hashing Unit

The hashing unit accepts a 22 bit virtual page number and generates a 10 bit uniformly distributed index into the hash table. The current

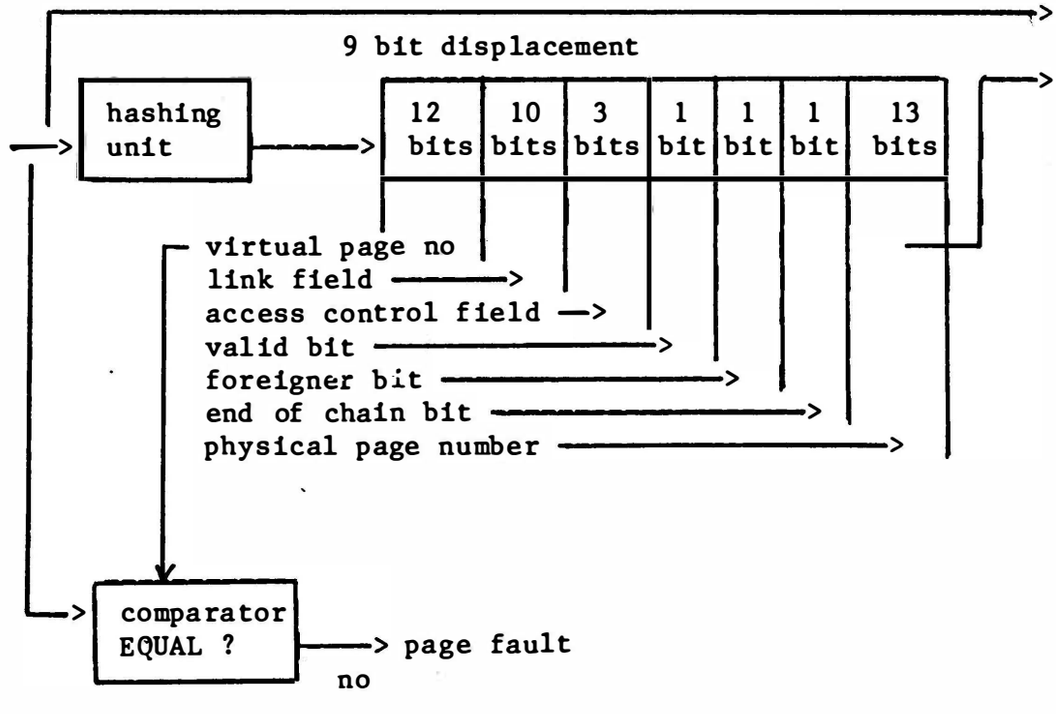


Figure 7.8 - the hash table format

hashing unit assumes a simplistic form and merely extracts the low order bits of both the address space number and the page number. More complex hashing algorithms can produce a better randomising effect for little extra cost, and may be included in later versions of the hashing unit if necessary.

7.4.2.3. The Hash Table

The hash table differs slightly in format from the table described in 7.4.1. The unit is held in high speed bipolar memory, with a cycle time of 50 nano-seconds. Each cell of the hash table is 41 bits in size and the current version of the hardware uses 1024 cells. (This size hash table can easily support the 400k bytes of main memory attached to the system, as we will see later). The seven fields of each cell are as follows:-

1 -	virtual address identifier	- 12 bits
2 -	physical page number	- 13 bits
3 -	Access field	- 3 bits
4 -	valid field (V)	- 1 bit
5 -	link field	- 10 bits
6 -	foreigner field (F)	- 1 bit
7 -	end of chain field (E)	- 1 bit

7.4.2.3.1. The Virtual Address Identifier

This field is used to identify the virtual address which uses the cell. The field is only 12 bits in length as the 10 bits used in the hashing function need not be saved. The identity of the virtual page may be recovered by combining the 10 bit cell address and the 12 bit virtual address identifier field. All the information held in the cell pertains to this virtual address.

7.4.2.3.2. The Physical Page Number

This field is used to hold the page number in main memory of the virtual page. This number is combined with the displacement to form a full 22 bit main memory address.

7.4.2.3.3. Access Control Field

The access control field governs the type of access which the page, as opposed to the segments within the page, may receive, such as read,

write and kernel. This field is not normally required, because such access checks are made by the capability registers. However, because some of the extra capability registers lack an access control field (for implementation reasons), one is placed in the hash table. Consequently, if a segment is addressed via one of the general capability registers, the access rights field is validated for both the capability register and the page of memory. If either one of these checks fails a violation interrupt is generated. Thus, if several segments are loaded into one page, the page access must be the union of the access rights of all of the segments. For example, if a capability to a segment has an access of read only, and another capability has an access of write only to a different segment in the same page, then the page must have both read and write access.

7.4.2.3.4. Valid Field

This boolean field declares a cell to be valid. When the hash table is initialized, all cells are invalid. However, when page map entries are loaded into the table, cells become valid. The hardware ignores the contents of an invalid cell.

7.4.2.3.5. The Link Field

This field contains the address of the next cell in a list. The last cell of the list is linked to the head, so that the software can find the home cell. A special field (rather than a nil value) signifies the end of a list.

7.4.2.3.6. Foreigner Field

Because the virtual address identifier field does not contain all the bits of a virtual page number, it is not possible to determine whether a cell is part of another list or belongs at its current address. The foreigner bit is set if the virtual page number would not hash to the cell address at which the mapping information is held. Thus, all cells which are part of a list, except the head, are tagged as foreign cells.

7.4.2.3.7. End of Chain Field

This boolean field declares a cell to be at the end of a list. It is required because no special null value is reserved for the link field. This also allows the last cell of a chain to be linked to the top of the list and yet still be recognized by the hardware as the last cell.

7.4.2.3.8. Summary

The cell format in the actual address translator differs from that described in section 7.4.1 by the addition of the access field, the foreigner field and the end of chain field. The foreigner field is required because the virtual page field is smaller than the size of the virtual page number. This optimization represents a saving of 9 bits. The end of chain field is required because no special null address is reserved. The access field is not logically required, but is present because some of the extra capability registers are not of the same format as the 16 standard registers provided.

7.4.2.4. The Comparator

This unit tests the virtual address identifier field, and those bits of the virtual page number not used by the hashing function, for equality. If equal, the physical page number field value is used as the translated page number.

7.4.2.5. The Finite State Control Machine

The hash table, hashing unit and comparator are controlled by a small finite state machine. This machine is designed to follow overflow chains and detect various page fault conditions. The flowchart shown in Figure 7.9 describes the cycle of the machine.

When a virtual address requires translation the machine is started. The cycle either retrieves a main memory page number from the hash table, or exits with a page fault condition.

7.4.2.6. The Software Algorithms

The algorithms used for inserting and deleting items from the hash table are basically the same as those described in 7.4.1. However,

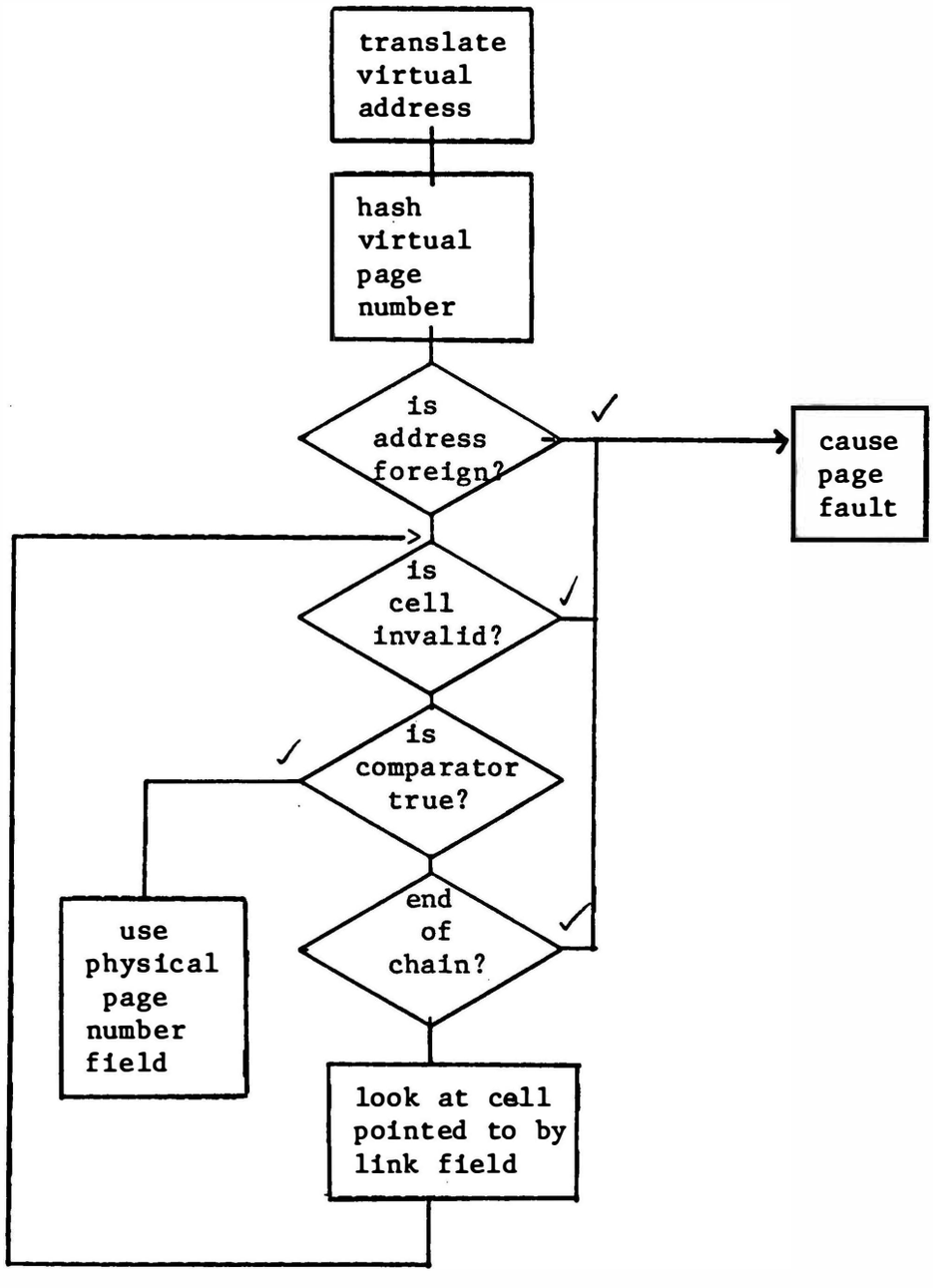


Figure 7.9 - the hardware retrieval algorithm

because the actual table format differs slightly, the algorithms also differ slightly. The foreign bit simplifies the steps which determine whether a cell belongs at a particular address. The modified algorithm only needs to test the value of this field, rather than hashing the virtual page number held in the cell. The end of chain bit must be tested rather than examining the link field in order to determine if a

chain has ended. The last item in a list must be chained to the head of list, so that the algorithms can find the head of any list. This was previously determined by hashing the virtual page number held in a cell. In all other respects, the algorithms remain unaltered.

7.4.2.7. Communicating with the Hash Table

To the software responsible for initialising and maintaining the data in the mapping hardware, the hash table and associated registers appear in a special address space, the memory control address space (number zero). Values may be saved into or read from the various fields of the hash table by executing memory reference instructions on this address space. The contents of the hardware tables are protected by the normal capability addressing mechanism. The format of the memory control address space is defined in Appendix D.

7.4.2.8. Address Spaces 1, 2, 3 and 4

Four of the 2^16 address spaces are not mapped by the hash table address translator, but by directly indexed map tables, held in high speed bipolar memory. Address space 1 holds the code of the kernel, whilst address space 2 is reserved for the kernel data. Address spaces 3 and 4 are reserved for the two DMA channels. Because the kernel itself must handle the page replacement and mapping algorithms and it has its own locked down pages, the kernel is mapped by its own address translator. Whilst not strictly necessary, this decision simplifies the memory management. Moreover, the directly indexed tables can translate an address in unit time, unlike the hash table in which the translation time varies depending on the chain length. This timing consideration is not of consequence for normal programs, but is extremely important for the DMA channels, which must receive immediate and fast attention, when addressing a fast device (such as disk). It is also desirable that the kernel program execute as fast as is actually possible.

Another reason for the inclusion of the special map tables is that they significantly simplified the hardware development. A base level address translator was available, and allowed the processor to execute 'kernel like' test programs before the hash table unit was debugged. The map tables for these address spaces are also referenced via address space zero, the format of which is found in Appendix D.

7.4.2.9. The Peek Operation

For an efficient implementation of the page replacement software it is important to determine whether a page reference would cause a page fault to occur, without actually generating a page fault. Whilst this software could execute a software retrieval algorithm, the hardware provides a fast mechanism which allows an instruction to test for a page fault. This is implemented by means of a special bit in the access field of a capability register, which, if set, causes the page fault interrupt for the reference to be inhibited. The program may then examine the violation register (see section 7.3.3.2.9) to determine whether an interrupt would have resulted. For security reasons, the peek operation only inhibits the actual interrupt if a fault condition exists; the reference is still aborted, whether or not the peek bit is set.

7.4.2.10. Performance of the Address Translator

A potential danger with using a hash table is that the number of collisions (or clashes), to any one cell and the average chain length may become unacceptably high. Acceptable performance can, however, be obtained if the hash table is sparsely occupied (i.e. a low loading factor). Providing that the hashing unit generates a uniform distribution of hash keys, the expected number of probes (E) to retrieve an item in the hash table can be calculated from:

$E = 1 + a/2$ where a is the loading factor (Morris, 1968)

The current version of the MONADS II processor uses a hash table size four times the number of pages of physical memory (i.e. $a = 1/4$), so $E = 1 + 1/8 = 1.125$, which is acceptably low. In a true associative memory $E = 1$.

The hash table performance is also affected by the efficiency of the hashing function, which should guarantee a uniform distribution of hash keys. The current version of hashing unit uses a combination of low order bits from both the address space number and the page number. Should this function yield poor results, experiments may be made with more complex hashing functions, such as the one used in the IBM

System/38 (IBM, 1978).

Figure 7.10 shows the timing delays inherent in the Series II address translation unit. It can be seen that the minimum access time (t min) will be

$$t_{min} = 0 + 50 + 50 + (300 \sim 700) \text{ ns}$$

$$= 400 \sim 800 \text{ ns}$$

On average

$$t_{av} = (400 \sim 800) + (E-1) \times 100$$

$$= 412 \sim 812 \text{ ns}$$

The variation in the main memory time is dependent on the cycle stealing of the refresh hardware for the dynamic memories used.

To maintain acceptable performance, the value E must be kept low. Thus when additional main memory is added the hash table size must be expanded proportionally. This increase in size does not necessarily affect any of the fields within the hash table. If the hash table is divided into blocks, links may be restricted to the 'block' of hash table in which they exist.

The MONADS II virtual address size is only 31 bits in size, whereas other capability processors use a much larger address. If the MONADS II address were expanded to 64 bits, the size of each cell in the hash table would increase from 41 bits to 73 bits. This increase is less

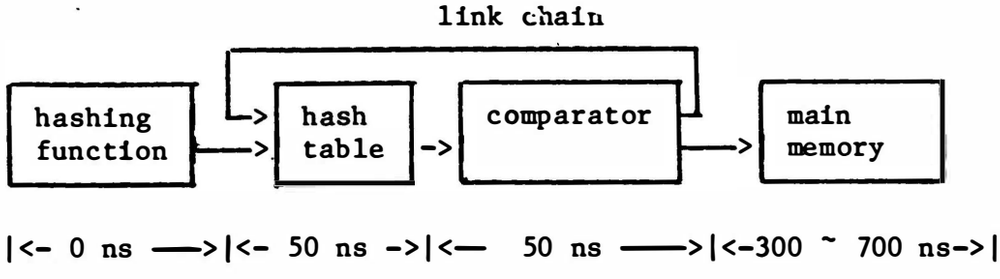


Figure 7.10 - the address translator timing

significant than doubling the size of main memory, i.e. adding a bit to the hash key. Thus, the address translator is relatively unaffected by changes in virtual address size.

Performance of the hash table may be optimized by overlapping the comparison of the virtual page identifier in the current cell to the virtual page number, with the fetch of the cell linked to the current cell. This optimization, however, has little effect if the value E-1 is low.

7.4.3. Alternative Solutions

Only two other computers have attempted to translate long virtual addresses without using the conventional solutions, one is the MU6-G processor described in Chapter 3 (which provides a hardware associative address translator unit) and the IBM System/38, described in Chapter 4 (which uses a pair of main memory hash tables with firmware assistance).

The MU6-G processor uses a serial associative memory, with an average retrieval time of 6 micro-seconds. This unacceptably high time is reduced by a small pseudo-associative cache memory. The IBM System/38 uses a microcoded loop to search a hash table for a page table entry which, since the tables are held in main memory, is bounded in time by the memory access time. Again, this translation process is augmented by a small pseudo-associative cache memory. Thus, both these processors require two address translator units, and microcode (or hardware) for loading and maintaining the cache memories.

The MONADS II processor uses one hardware address translator, and needs no microcode assistance for translating addresses. Moreover, the technology used in the construction of the hash table, and thus the cost, is little more than that of the cache memories used by the other two systems. Thus, the overall complexity of the MONADS II system is less than either the MU6-G or the IBM System/38.

Not only is the technology of the hash table the same as that of the cache memories, but the overflow chains are contained within the table, without the need to address the slower associative tables.

Consequently, one could expect the MONADS II translation unit to offer at least equal, or superior performance to the MU6-G or System/38

translation units.

7.4.4. Conclusions

We have now described both the functionality and implementation of the MONADS II virtual address translator. We have demonstrated that this unit can offer equal or superior performance to the other available units (MU6-G and IBM System/38) and is slightly simpler in design. The technique chosen can easily cater for a larger virtual address, without significant increase in cost and no real increase in complexity. The unit fulfils its aims (section 7.4.1.2)

- it uses an associative address translation technique

- it only maps pages of virtual memory which are present in mainstore

- the unit generates a page fault for any page not present in mainstore

- the unit is fast

- software assistance is only required for performing insertions and deletions. Whilst these algorithms are more complex than the retrieval algorithms, they are only performed when the processor discovers a page fault, an inherently slow operation which can be performed in parallel with the insertion or deletion

- loading the address translator is easy.

The next major section examines the changes made to the HP2100 source processor.

7.5. Modifications to the HP2100A Hardware

One of the advantages of the enhancement technique described in Chapter 6 is that it requires very few changes to the hardware of the source processor. In reality, six small modifications to the HP2100 were required, partly for logical reasons and partly to enhance the efficiency of the processor. These changes were not complex.

7.5.1. The Memory Controller

The basic HP2100A processor is divided into three areas; the central engine, the input-output system and the memory unit. The latter consists of up to 32 k words of core memory, many driver cards and a logic controller card. The logic controller card generates the timing signals for the core stack, and also contains the memory data and address registers.

The controller card was replaced by a plug compatible, but much simpler, card which interfaces the HP2100 to the intermediate processor. The memory data and address registers are made available to the intermediate processor via an interface cable, which connects the processor to the HP2100A.

7.5.2. DMA Logic

Unfortunately, the DMA system used by the HP2100 has full knowledge of the timing and nature of the processor. When the DMA system requires a cycle, it requests the next processor transfer cycle, and without checking the response assumes that the cycle may be used. It also manipulates the major processor buses when transferring data. Thus, when the intermediate processor interface was introduced the DMA logic was also changed to accommodate the new logic. The DMA logic bypasses the intermediate processor, and interfaces directly to the memory manager.

7.5.3. More Writable Control Store

The basic HP2100 only allows 512 x 24 bit words of writable control store. In order to efficiently implement the MONADS operating system, the size of the control store was expanded to 4096 words. This modification was carried out by Dr. J. Rosenberg.

7.5.4. Mapping to Top Leaf

The HP2100 memory reference instructions can easily address the base leaf and the current instruction leaf. Whilst special address interpretations are placed on the base leaf, it was not sensible for data manipulation instructions to address the current instruction leaf, as this only contains code in the new architecture. To simplify the

addressing of the top leaf, which has very many address interpretations, the current leaf data instructions were altered to address the top leaf. Only the control instructions, such as jumps, can address the current code leaf.

7.5.5. Interrupt Logic

The interrupt logic of the HP2100 was modified to allow the intermediate processor to abort an instruction after a fatal error. This allowed page faults to be trapped correctly. In addition, the interrupt vector was moved from the base leaf to the first leaf of the kernel code space.

7.5.6. Asynchronous Interface

The basic HP2100 assumes a standard delay time for the core memory cycle time. Because the intermediate processor takes a variable amount of time to execute different sized instructions, the HP2100 was modified to allow it to wait for a memory acknowledge signal before continuing.

7.5.7. Summary

Most of the changes made to the HP2100 were relatively small and easy to implement. The most complex change was to the DMA logic, mainly because of its poor initial design.

7.6. Software Packages

During the development of the MONADS II hardware a number of software packages were developed by the author. These packages either formed an integral part of the processor, such as the microcode for the intermediate processor, or assisted the construction of the hardware and firmware. This section will briefly describe these modules.

7.6.1. The Intermediate Processor Microcode

The 1024 words of intermediate processor microcode are generated from a microcode source file of about 1400 lines of code. This microcode is responsible for executing the target instruction set, performing diagnostic functions and for configuring the processor map tables prior to bootstrap. A full listing of the code is found in Appendix C.

7.6.2. The Microcode Assembler

The intermediate processor microcode is assembled by a microcode assembler, written for the HP2100A minicomputer running under the DOS-M system. The assembler generates a compiled listing, code files, a symbol table listing and an entry point listing.

7.6.3. The Bootstrap

A number of levels of bootstrap are provided. The lowest level is implemented as a microcoded instruction in the intermediate processor. The next level is held in read only memory of the virtual space, and is written in assembler. The last level is held on disk, and loads the operating system into memory. The middle level bootstrap may also communicate with another HP2100 and act as a fast link monitor.

7.6.4. Utilities

Various utilities were developed, such as a PROM programmer program, and a signal cross reference generator.

7.7. Conclusion

This section concludes the description of the MONADS II system, and demonstrates that the system has fulfilled its primary aims.

(1) The intermediate processor provides a programmer with 16 capability registers. Software has been written which uses these registers and microcode exists which maps the registers onto the MONADS addressing structure. The registers are held in a fast register file, and receive hardware assistance when they are used (for example, the access descriptor register checks that the mode of access is not contravened.) Thus, it is possible to efficiently implement the addressing structure. All the HP2100 memory reference instructions can address the 31 bit virtual space by using only the 4 bit capability register number. Thus, the registers can be efficiently addressed. Moreover, whilst the intermediate processor provides extra registers for addressing code and the stack, some of the 16 capability registers could have easily been dedicated for these purposes. Consequently, the MONADS II system demonstrates the practicality of the capability register addressing scheme proposed in Chapter 5.

(2) Software is at this present time being developed for the MONADS II processor. The repertoire of programs consists of a macro assembler, a Pascal compiler and a Modula compiler, and many test and diagnostic programs. An operating system is being developed which will allow the system to be used as a development testbed. Since the MONADS II system is basically a scaled down implementation of the MONADS architecture, it may be used to develop software using the MONADS concepts.

(3) The enhancement technique developed in Chapter 6 has clearly been demonstrated as practical and powerful. The MONADS II processor is based around a very simple 16 bit minicomputer and yet it provides an advanced architecture to the assembler level programmer. The building of the intermediate processor was demonstrably simpler than developing a totally new processor.

(4) The MONADS address translation hardware is capable of mapping very large virtual addresses onto main memory addresses quickly and simply. The scheme compares very favourably with alternative solutions.

8. Conclusion

This chapter serves three purposes. First, we discuss some of the limitations of the MONADS II computer system. Second, we indicate areas in which future research will be useful. Third, we evaluate the significance of the work described in this thesis.

8.1. Limitations of the MONADS II System

The MONADS II computer system, while representing a real implementation of the two models developed in this thesis, has a number of limitations. We now discuss these, and consider how they might be removed.

8.1.1. The Address Size

As described in Chapter 7, the MONADS II address is 31 bits long, consisting of a 16 bit address space number and a 15 bit displacement. While this may be sufficient for a pilot system, a production environment would require both many more address spaces, and much larger address spaces.

A small address space number means that the number of address spaces will be exhausted quite quickly, and old address space numbers must be reused. As explained in Chapter 5, this requires that all capabilities for these address spaces be found and deleted before the number can be safely reassigned. Whilst this is less serious in the MONADS system than in other capability based computers (because capabilities are not freely distributed) it is still inconvenient and time consuming. A larger address space number allows many more address spaces to be allocated before they must be reused.

The maximum size of an address space should be large enough to hold the data of most large segments (e.g. containing file data) and only very large segments should be composed of many address spaces. If the address space size is too small, then many address spaces are required to contain the data from large segments. This then complicates the way that data is addressed, and uses many more address space numbers than are logically required.

If the MONADS II computer system were being redesigned both of these restrictions could be removed. The address space number held in

the capability registers could be expanded (e.g. to 32 bits) and the within address space displacement (in capability registers and modifier registers) could also be expanded (e.g. to 31 bits). If this were done a restriction would need to be placed on code address spaces, as the Hewlett Packard can only fetch code from a 32k word space. All other address spaces would need to be addressed via capability registers and modifier registers, in the same way as they are currently referenced.

A larger address space would also increase the size of the address translation unit. The unit would need to hold an extra 32 bits of information per cell because of the larger virtual address. This is a small increase in cost (less than twice the size) for a very large increase in the size of the virtual space (from 2^31 to 2^63 words). Later we will mention a new version of the MONADS II computer which implements one of these enhancements namely, more address spaces.

8.1.2. Special Capability Registers

It will recalled that the MONADS II hardware provides 16 general capability registers and 6 special registers. These six special registers (for addressing the process stack, code, inter-leaf links and code related data) were implemented differently partly for historical reasons and partly because of peculiarities of the HP2100A, rather than for theoretical reasons, and could easily have been constructed from general capability registers. In a new implementation, all of these would be general capability registers, as described in Chapter 5. Without these extra registers the addressing mechanism is completely uniform, which simplifies the construction of the hardware, removes the need for access rights at the page level and also simplifies the code generation phase of the compilers.

8.1.3. The Hashing Function

As described in Chapter 7, the hashing function used in the prototype address translation unit of MONADS II is very simple, and it is not expected that this function will yield a particularly uniform distribution of hash keys. Because of this, a more complex hashing function (such as described in (IBM, 1978) and (Ramamohanarao and Sacks-Davis, 1981)) could easily be implemented, without increasing the total address translation time.

8.1.4. Processor Speed

The MONADS II system, while far faster than a software or firmware implementation (as discussed in Chapter 6), is still slower than desired. The most significant cause for this lack of speed is the current implementation of the intermediate processor, which uses 300 nano-second Programmable Read Only Memories to hold its microcode. This limits the micro-cycle time to 300 nano-seconds. This microcode could instead be placed in faster 70 nano-second Read Only Memories, which would allow the processor to achieve a micro-cycle time of about 150 nano-seconds. (A decrease to 70 nano-seconds is not possible because of other timing constraints.) Such a modification would improve the speed of the MONADS II system significantly, because all memory traffic proceeds via the intermediate processor.

8.1.5. The HP2100A Instruction Set

While there were many advantages in inheriting the HP2100A basic instruction set in terms of speed of implementation (see Chapter 6), this instruction set is not ideally suited to the software methodology of the MONADS project. The only solution to this problem is to build a new processor, which was not a viable alternative at the time that MONADS II was built. Since that time funds have become available and a new processor, MONADS III, which we will briefly discuss later, is being designed.

8.1.6. Page Replacement

The MONADS II processor provides no support for page replacement algorithms (e.g. in the form of 'use' bits or 'modify' bits). Thus, the operating system has no knowledge of which pages have been modified, or which pages are being used, and must use a random replacement algorithm. Whilst it is possible to simulate some of this information in software, as in the VAX 11/780 (Digital Equipment Corp., 1979), this is an expensive process, and would consume much of the power of the HP2100A. Consequently, it was decided that special hardware would be built when time was available, and that it was more important to have a slow, but complete, system than an impressive but incomplete one. Taking advantage of the concept of the hardware kernel (Rosenberg, 1979; Rosenberg and Keedy, 1978), this hardware could then be introduced at a

later stage, without any effect on the main operating system software other than improved performance.

8.1.7. Offsets from Capability Registers

Another limitation of the MONADS II hardware is the size of literal offsets which may be used with capability registers. Because this is only three bits, only a small frame of words may be addressed without using a modifier register. Ideally this offset should be as large as the size of an address space, however, there are not enough free bits in the memory reference instructions. The only solution to this problem is to create some new HP2100 memory reference instructions which are two words long. In this case the first word could contain the operation code, and the second word could contain a 16 bit literal offset.

8.2. Future Research

In this section we briefly indicate the direction in which the research described in this thesis may be extended.

8.2.1. MONADS III

Because of the limitations of the MONADS II system a grant was requested (ARGC Grant Number F80/15191) to build a totally new computer system, called MONADS III (Rosenberg, Rowe and Keedy, 1982; Keedy and Rosenberg, 1982a, 1982b). The role of this processor is different from MONADS II. It is designed to support a large number of terminals and provide a fast and powerful computer utility. Eventually the MONADS II processor will form part of the MONADS III system, and provide its communications facilities. Most of the software for MONADS III will initially be prepared on the MONADS II system.

Unfortunately, the design of the MONADS III processor has been suspended due to the resignation of the project's chief investigators and an alternative plan has been adopted. In order that a full system could be realized, work was started on another processor which was basically the same as the MONADS II system, but which removed some of the limitations.

8.2.2. MONADS II/2

MONADS II/2 removes some of the restrictions cited above. The processor increases the address space number field to 32 bits, and uses a larger page size (1024 words instead of 512). In addition, the main memory limit has been extended from a maximum of 512k bytes in MONADS II to 2M bytes. It also provides support for more processes, and a number of additional registers. In this way, MONADS II/2 is less of a pilot system and more of a production machine. Unfortunately, it will never achieve the power of the proposed MONADS III.

8.2.3. Future Work

There are a number of areas in which future research could be directed. The MONADS II computer provides no support for the paging software. This is partly because there was not time to design such hardware, and partly because it is not obvious what paging criteria should be applied to address spaces when a page must be removed from main memory. Whilst the working set model (Denning, 1968, 1980) is appropriate for computational virtual memories in conventional computer environments, it is not clear that this is the case in a system which supports a large uniform virtual memory which contains permanent data (e.g. files) as well as computational data. Further work needs to be done to determine the best policy for removing (and fetching) pages of a module, and then to develop a hardware unit (in the same spirit as the MANIAC II unit (Morris, 1972)) which can monitor and provide this information. (The MONADS II system already allows different page placement policies to be developed and evaluated because the address translation system is not concerned with the organization of the page tables. Thus, alternative secondary storage and page fetching techniques may be attempted without affecting the hardware.)

Another appropriate topic for future work is in the area of backup and recovery. In the event of a hardware (or software) malfunction, the operating system should be able to provide recovery of any data which may have become corrupted. Conventional file systems provide such features by maintaining backup and recovery information when file data is modified. However, in a uniform virtual memory there is no longer a clear distinction between file data and computational data, which

complicates the task of recovery.

The MONADS II operating system provides a symbolic debugger subsystem for use in tracing errors in Pascal or Modula programs (Dawson, 1982). Since the processor provides no hardware support, all the debugger functions are emulated by compiler generated software. Consequently, the debugger is very inefficient. The author is currently designing hardware which detects when breakpoints have been reached (either for data or code addresses) and when variable values have changed. This research will be reported in a separate paper (Abramson and Rosenberg, 1982).

8.3. Achievements and Significance

The most significant achievements of this thesis are threefold. First, we have developed a new hardware model for capability based addressing. Second, we have developed a general technique which can be used to implement complex and novel computer architectures quickly and cheaply. Third, we have implemented a real computer system which demonstrates the viability of these ideas.

8.3.1. The Addressing Model

In Chapter 1 we stated that one of the objectives of this thesis was to provide a hardware unit which allowed information to be shared and protected in a uniform, flexible and efficient manner. We can now determine if these objectives have been met.

8.3.1.1. Sharing of Data and Code

Because of the clear distinction between hardware and system firmware the new model does not enforce any particular sharing policy, but leaves this to the firmware which manipulates capabilities. If a module has a capability for a segment in a register, then it has access to the segment. The firmware must determine whether this capability may be placed in a register. In the MONADS system, sharing of data and code is allowed only through the sharing of the modules themselves, in accordance with the information hiding principle. Thus segment capabilities are never passed freely around the system (Keedy, 1982c), as is the case in many other capability systems, such as Hydra, StarOS, etc. Because the hardware is not concerned with the management of

capabilities it supports either method of sharing.

8.3.1.2. Protection of Information.

Since the model uses a capability based addressing scheme information can be protected from inadvertent corruption either by the owner or another user. A module may only address a segment if the capability has been loaded into a capability register. As this is controlled by system firmware, segments are protected. The access rights field of the capability only allows valid operations to be performed on segments. For example, code segments and read only data segments can be protected from being modified. In addition, since the model has attractive refinement properties, it is possible to restrict access to a particular part of a reference parameter.

8.3.1.3. Flexibility

The model proposed in this thesis is flexible because it confines the use of hardware (which is difficult to change) to mechanisms which (for efficiency reasons) have to be fast, and leaves all important policy decisions to firmware (which is relatively easy to modify). We have demonstrated this flexibility, both on paper and by practical experience. In Chapter 5 we proposed mappings of various software structures onto the hardware, and each time the result was a uniform and efficient addressing structure. In addition, the hardware for MONADS II was built and tested before the MONADS software group had defined the format and nature of the capability structure of a module. This structure was then mapped onto the hardware without difficulty, which was a realistic and practical demonstration of the flexibility of the hardware.

8.3.1.4. Efficiency

The model proposed in Chapter 5 is efficient because it places those operations which must occur quickly in hardware, and uses firmware to support those which do not. Thus, capabilities are held and used in fast hardware registers. The virtual addresses are then mapped onto main memory addresses by a hardware address translator. The structure of the capability lists, and operations on high level objects, are left to the high level firmware as these do not require the same level of efficiency

(e.g. the capability list is only consulted when a capability register is loaded). These may then change and evolve with the software ideas. The implementation of the MONADS operating system, which will support a number of concurrent user programs, demonstrates the overall efficiency of the proposal.

8.3.1.5. Uniformity

The capability registers are the only way of addressing memory, thus providing one uniform method for addressing, protecting and sharing information in the computer utility. This simplifies the hardware construction, the compilers, the hardware kernel, the main operating system and also the task of the computer user.

8.3.2. The Enhancement Model

After we had developed the addressing model, we faced to problem of how to produce an implementation with which we could evaluate its effectiveness. In Chapter 6 we examined a number of alternatives, such as various software implementations, microcode techniques and some limited hardware modifications. Because of time and fiscal constraints we were forced to modify an existing computer. The enhancement model which was developed allowed us to produce a real implementation which is efficient enough to support a number of concurrent user programs.

The value of this model is best illustrated by considering the time taken to design and build MONADS II. The intermediate processor was designed by the author over a period of a few months, built in about 6 weeks, and tested in about 2 weeks. The address translation unit, which was not part of the intermediate processor, took about the same period of time again. Without the new implementation technique, a totally new processor would have been required which would have taken much more man-power than was needed to build MONADS II. For example, MONADS III has already consumed about 4 man-years of effort, and has not yet been completed.

8.3.3. Practical Achievements

The practical achievements of this research work are embodied in the MONADS II system, which is a complete working computer utility. As described in Chapter 7, the MONADS II system consists mainly of the

central processor, main memory, 80 Mbytes of disk and a terminal multiplexor which can be connected to 16 terminals. This configuration can support a number of concurrent user programs.

After completion of the hardware, a hardware kernel was written (Wallis, 1980). This body of code resides partly in firmware and partly in the kernel address space of the processor. It provides a higher level interface to the hardware for the main operating system. At this stage of the project, the primitive components of an operating system have been developed, together with a command line interpreter (Patterson, 1981). Compilers have already been developed for assembly code (Rees, 1981), Modula 2 (Wirth, 1977) and Pascal, and programs written in any of these languages can be executed directly on the MONADS II system. In addition, a C compiler is currently been written (Bird, 1982). At present all compilers and assemblers execute on a VAX 11/780 processor, and code is down-line loaded to MONADS II. It is expected that shortly the compilers will execute directly under the MONADS II operating system.

Once the main operating system has been completed, and the compilers are resident on the MONADS II processor, the system will support a software test and development environment similar in nature to other commercial systems.

8.4. Final Remarks

This thesis has made a contribution in the area of hardware support used in capability based computers. It provides a hardware framework around which a software designer may experiment with different capability structures. The widespread availability of a machine which implements the memory and capability features of this model (e.g. with V.L.S.I. technology) would greatly aid research into operating systems in both universities and industry.

The thesis has also demonstrated a technique which should allow different computer architectures to be evaluated without the need for many man years of effort by expert staff. Once new ideas have been evaluated they may then become the framework for new computer designs.

Appendix A

This appendix defines the new operands which are executed by the intermediate processor. Most of these operands can be used with any of the HP2100A memory reference instructions, and are all located in the top leaf of the address space. Some operands are only effective for load type of instructions (i.e. a read from the address space) whereas others are only effective when a store type instruction is used.

Immediate load instruction

This operand takes the 8 bit value of the address and returns it to the HP2100A. This value is treated as a 7 bit two's complement integer. The bottom bit of the address is used as the sign bit. Negative numbers are sign extended to the full 16 bits. The instruction allows a program to use numbers in the range of -128 to 127 without addressing a segment of store.

n(CRr)

This operand uses one of the general capability registers, CRr, to address a segment in memory. A small constant n (in the range 0-7) may be specified as an offset to the segment start address. The offset is added to the start address and validated against the segment length before the memory reference is executed.

(CRr)[Mm]

This operand also uses one of the general capability registers, CRr, to address memory. However, the segment start address may be modified by the contents of one of the 8 modifier registers, Mm. Thus, the modifier value is added to the start address and validated against the segment length before the segment is referenced.

(CRr)[Mm/2]

This operand is the same as the previous one, except the modifier is treated as a byte count rather than a word count. The modifier value is halved before it is added to the segment start address. When the word of memory is addressed, either the left or the right byte of the word is used depending on the least significant bit of the modifier register.

CRr.subsystem

This operand is used to address the subsystem field of a capability register. This field has not been described in the thesis, and is no longer used by the hardware. When the capability registers were first designed, this field held the identity of the subsystem which owned the contents of a capability register. Validation checks were performed to only allow the subsystem which had originally loaded a register to use the register. This removed the need to invalidate registers when a domain change was executed.

CRr.address-space-number

This operand is used to address the address space field of a capability register.

CRr.displacement

This operand is used to address the contents of the displacement field of a capability register.

CRr.segment-limit

This operand is used to address the segment limit field of the capability register.

CRr.access-rights (decrease only)

This operand allows a program to reduce the set of access rights within a capability register. When a store operation is performed on this address, the data pattern from the HP2100A is 'anded' with the contents of the access rights field, reducing the allowed access.

CRr.access-rights

This operand addresses the access rights field of a capability register.

STACK.address-space-number

This operand is used for addressing the contents of the STACK address space number register. This register is used to form addresses in the process stack, and is combined with various displacement and length register to form a one of the special capability registers.

n(T)

This operand is used to access scaler variables held on the process stack. A virtual address is formed from the displacement in the T register (Top of stack) and the STACK address space number register. The value n may be used to modify the displacement. Unlike the general capability registers, this value may be in the range 0-31, and is subtracted from the displacement.

T.limit

This operand is used to address the limit field of the Top of stack capability register. No stack reference is allowed below this register.

+(T)

This operand performs a push operation on the process stack. The Top of stack displacement is incremented and, providing it is not below the T.limit register, data is saved on the stack.

(T)-

This operation performs a pop operation on the process stack. Data is read from the current top of stack location, and then the Top of stack displacement register is decremented. The Top of stack displacement register is not allowed to fall below the T.limit register.

T

This operand is used for addressing the Top of stack displacement register.

T=T+MDR

This operand is used to modify the contents of the Top of stack displacement register. The value in the HP2100A memory data register is added to the contents of the Top of stack register in one operation.

Mm

This operand is used for addressing the 8 modifier register.

Cc

This operand is used for addressing the 8 counter registers.

ldw +Cc - (load word from +Cc

This operation is only executed when a load type of instruction is used. The contents of counter register Cc is incremented, the contents of modifier register Mc is also incremented, and the new value of the counter register is returned. The instruction is useful for scanning through segments of memory.

stz Cc - (store zero Cc

This operation is only executed when a store type of instruction is executed. Data is saved in the counter register Cc, and the associated modifier register, Mc, is set to zero. This instruction is useful for initializing the modifier and loop counter registers.

L1.displacement

This operand is used to address the contents of the L1 displacement register. This register defines a frame of 256 words in the current STACK address space, and forms one of the special capability registers.

L1.limit

This operand is used to address the L1.limit register. This register forms the limit field of the capability register for addressing scalars on the L1 stack frame.

L1=L1+MDR

This operand is used to modify the contents of the L1 displacement register. The contents of the memory data register is added to the contents of the L1 register.

L2.displacement

This operand is used to address the contents of the L2 displacement register. This register defines a frame of 256 words in the current STACK address space, and forms one of the special capability registers.

L2.limit

This operand is used to address the L2.limit register. This register forms the limit field of the capability register for addressing scalars on the L2 stack frame.

L2=L2+MDR

This operand is used to modify the contents of the L2 displacement register. The contents of the memory data register is added to the contents of the L2 register.

CONSTANT.address-space-number

This operand is used to address the address space number of the constant address space. It froms one of the special capability registers.

CODE.address-space-number

This operand is used to address the address space number of the code address space. It froms one of the special capability registers.

PROCESS-NUMBER

This operand is used to address the process number register. When this register is altered, the current process is exchanged for the new process. All of the process own registers are swapped by the intermediate processor.

CURRENT-SUBSYSTEM

This operand is used to address the current subsystem register. This register is used to hold the identity of the currently executing subsystem. When the capability registers held a subsystem field, this register was used for validation purposes.

VIOLATION-MASK

This operand is used to address the violation register. When the intermediate processor causes an interrupt, this register is loaded with a bit map which describes the cause of the interrupt.

KERNEL-OFF

This operand is used to take the intermediate processor out of kernel mode. The kernel executes this instruction before it returns to a user program.

PROCESS-TIME-MSW

This operand is used to address the most significant word of the process time limit register. This register is decremented every 2^16'th milliseconds and if it reaches zero an interrupt is generated.

PROCESS-TIME-LSW

This operand is used to address the least significant word of the process time limit register. This register is decremented every millisecond and if it reaches zero the most significant word is decremented.

INSTRUCTION-COUNTER-MSW

This operand is used to address the most significant word of the instruction counter register. It is incremented every 2^16'th instruction.

INSTRUCTION-COUNTER-LSW

This operand is used to address the least significant word of the instruction counter register. It is incremented time an instruction is executed.

TIME-MSW

This operand returns the most significant word of the clock.

TIME-LSW

This operand returns the least significant word of the clock.

Appendix B

This appendix defines the mapping details of the HP2100A top leaf addresses. The table shows the BASE address for a particular instruction, followed by the within leaf displacement used for the instruction. These address patterns are decoded by the intermediate processor, and are translated into microcode entry points. The R bit denotes whether the reference is a read from the address space of the HP2100A or a write. Various characters are used in the within leaf displacement. Four bits, 'r', are used to form a capability register number. These are always taken from bits 3-6 of the address. Three bits, 'c', are used to identify a counter register and are taken from bits 0-2 of the address. These same bits are used to address the modifier registers, called 'm', and to form the small three bits capability offset, called 'n'. A five bit stack offset is taken from bits 0-4 of the address. An 'x' in any position signifies a don't care condition.

BASE	M9	M8	M7	M6	M5	M4	M3	M2	M1	M0	R	Instruction
76000	0	0	n	n	n	n	n	n	n	n		Immediate load instruction
76400	0	1	0	r	r	r	r	n	n	n		n(CRr)
76600	0	1	1	r	r	r	r	m	m	m		(CRr)[Mm]
77000	1	0	0	r	r	r	r	m	m	m		(CRr)[Mm/2]
77200	1	0	1	r	r	r	r	0	0	0		CRr.subsystem
77201	1	0	1	r	r	r	r	0	0	1		CRr.address-space-number
77202	1	0	1	r	r	r	r	0	1	0		CRr.displacement
77203	1	0	1	r	r	r	r	0	1	1		CRr.segment-limit
77204	1	0	1	r	r	r	r	1	0	0		CRr.access-rights (decrease only)
77205	1	0	1	r	r	r	r	1	0	1		CRr.access-rights
77400	1	1	0	x	0	n	n	n	n	n		n(T)
77440	1	1	0	x	1	0	0	c	c	c	r	ldw +Cc
77440	1	1	0	x	1	0	0	c	c	c	w	stz Cc
77460	1	1	0	x	1	1	0	m	m	m		Mm
77470	1	1	0	x	1	1	1	c	c	c		Cc
77601	1	1	1	x	0	0	0	0	0	1		+(T)
77602	1	1	1	x	0	0	0	0	1	0		+(T)
77603	1	1	1	x	0	0	0	0	1	1		T
77604	1	1	1	x	0	0	0	1	0	0		L1.displacement
77605	1	1	1	x	0	0	0	1	0	1		L2.displacement
77606	1	1	1	x	0	0	0	1	1	0		CONSTANT.address-space-number
77607	1	1	1	x	0	0	0	1	1	1		CODE.address-space-number
77610	1	1	1	x	0	0	1	0	0	0		STACK.address-space-number
77611	1	1	1	x	0	0	1	0	0	1		PROCESS-NUMBER
77612	1	1	1	x	0	0	1	0	1	0		CURRENT-SUBSYSTEM
77613	1	1	1	x	0	0	1	0	1	1	r	VIOLATION-MASK
77613	1	1	1	x	0	0	1	0	1	1	w	KERNEL-OFF
77614	1	1	1	x	0	0	1	1	0	0		PROCESS-TIME-MSW
77615	1	1	1	x	0	0	1	1	0	1		PROCESS-TIME-LSW
77616	1	1	1	x	0	0	1	1	1	0		INSTRUCTION-COUNTER-MSW
77617	1	1	1	x	0	0	1	1	1	1		INSTRUCTION-COUNTER-LSW
77620	1	1	1	x	0	1	0	0	0	0		T=T+MDR
77621	1	1	1	x	0	1	0	0	0	1		L1=L1+MDR
77622	1	1	1	x	0	1	0	0	1	0		L2=L2+MDR
77624	1	1	1	x	0	1	0	1	0	0		TIME-MSW
77625	1	1	1	x	0	1	0	1	0	1		TIME-LSW
77626	1	1	1	x	0	1	0	1	1	0		T.limit
77627	1	1	1	x	0	1	0	1	1	1		L1.limit
77630	1	1	1	x	0	1	1	0	0	0		L2.limit
77631	1	1	1	x	0	1	1	0	0	1		(T)-
77632	1	1	1	x	0	1	1	0	1	0		(T)-
77633	1	1	1	x	0	1	1	0	1	1		Clear CR.access-rights
77634	1	1	1	x	0	1	1	1	0	0		Double Pop
77635	1	1	1	x	0	1	1	1	0	1		Double Push

Appendix C

This appendix contains the details of the microcode instruction format along with the intermediate processor microcode. Each microinstruction is 24 bits in length, and is composed of 7 fields, as follows:

(1) BUS1 field	7 bits	bit 23
(2) BUS I-O field	1 bit	: :
(3) BUS2 field	4 bits	: :
(4) CONSTANT field	3 bits	: :
(5) FUNCTION field	4 bits	: :
(6) SPECIAL field	3 bits	: :
(7) MEMORY field	2 bits	bit 0

The source line of a microinstruction consists of these 7 fields and also a label field (before the BUS1 field), a jump target field (after the memory field) and a comment field (after the jump target field). The operands which are allowed in the various field are shown in Tables C1 through C7. We will briefly describe the purpose of each field of the microinstruction.

The BUS1 field

The BUS1 field determines which register is connected to the central bus of the intermedite processor. The allowed operands are shown in Table C1. The operand REGSTR takes the contents of DISPLAY register one as the register number. This allows a microprogram to scan through the intermediate processor registers.

The BUS I-O field

This field determines whether the register specified in the BUS1 field is placed onto the bus, or the bus contents are saved into the register. The allowed operands are shown in Table C2. If an INTO directive is issued, the accumulator is placed onto the bus.

The BUS2 field

This field allows the bus contents to be saved in one of the dedicated registers at the same time as another register is specified in the BUS1 field. Allowed operands are shown in Table C3.

Opcode	Code	Description
NOP OR BLANK	0	
MDR	1	Memory Data register
MAR	2	Memory Address register
PROCN	3	Process Number register
STIME1 + 2	4-5	Time
ADRSPC	6	Address Space descriptor
DISP	7	Displacement descriptor
ACCESS	8	Access descriptor
DISPL1 + DISPL2	9-10	Display registers
INST	13	Instruction counter
WCHDG1 + 2	11-12	Watchdog timers
SVR	14	Stack violation register
INST2	15	Instruction counter - lsw
C.SSN	16	CR Subsystem field
C.ASN	32	CR Address Space field
C.DISP	48	CR Displacement field
C.LEN	64	CR Length field
C.ACCS	80	CR Access field
M	96	Modifier registers
C	104	Counter registers
T	112	Top of stack register
L1	113	L1 register
L2	114	L2 register
CASN	115	Constant Address Space #
CCAS	116	Code Address Space #
CSAS	117	Stack Address Space #
CSSN	118	Current Subsystem number
PTIME1	119	Process time - msw
PTIME2	120	Process time - lsw
PINST1	121	Instruction count - msw
PINST2	122	Instruction count - lsw
TR1	123	Temporary register
TB	124	T Base register
L1L	125	L1 Limit register
L2L	126	L2 Limit register
REGSTR	127	Register file indirect

Table C1 - the BUS1 field

Opcode	Code	Description
BLANK	0	
NOP	0	
ONTO	0	Place register Onto bus
INTO	1	Store bus into register

Table C2 - the BUS I-O field

Opcode	Code	Description
BLANK	0	
NOP	0	
MDR	1	Memory Data register
MAR	2	Memory Address register
PROCN	3	Process number register
STIME1 + 2	4-5	Time
ADRSPC	6	Address Space descriptor
DISP	7	Displacement descriptor
ACCESS	8	Access descriptor
DISPL1 + 2	9-10	Display registers
WCHDG1 + 2	11-12	Watchdog timers
INST1	13	Instruction counter - msw
INST2	15	Instruction counter - lsw

Table C3 - the BUS2 field

The CONSTANT field

This field specifies the contents of one of the inputs of the ALU. This may be either one of 7 predefined constants, or the accumulator value. The allowed operands are shown in Table C4.

The FUNCTION field

This field specifies the operation which the ALU is to perform. Apart from the standard arithmetic and logic operations, the field may also be used for setting various processor states (such as kernel mode

Opcode	Code	Description
BLANK	0	
NOP	0	
ACC	0	Accumulator
377	1	Constant 377B
FF	1	Constant 377B
FF00	6	Constant 177400B
177400	6	Constant 177400B
7	2	Constant 7
0	3	Constant 0
1	4	Constant 1
2	5	Constant 2
77777	7	Constant 77777B
7FFF	7	Constant 77777B

Table C4 - the CONSTANT field

or debug state), and for performing a microcode jump. When a jump is specified, the 10 most significant bits of the instruction are used for the jump target address. The allowed operands are shown in Table C5.

The SPECIAL field

The special field is used to detect error conditons, cause interrupts, and to generate conditional microprogram skip operations. If the processor is not in debug mode (which is a special processor state) then any condition which is flagged will cause an interrupt and set a particular bit in the violation register. If the processor is in debug mode, then a flagged condition will cause a microprogrammed skip to occur. The allowable operands are shown in Table C6.

Opcode	Code	Description
NOP	0	
BLANK	0	
ADD	1	Function add
SUB	2	Function subtract
OR	3	Function logical or
AND	4	Function logical and
SWAP	5	Swap bytes of input
LEFT	6	Left shift input
RIGHT	7	Right shift input
JUMP	8	Micro code jump
END	9	Return to HP2100 instruction
KNLOFF	10	Turn the kernel bit off
DBGON	11	Turn Debug bit on
DBGOFF	12	Turn Debug bit off

Table C5 - the FUNCTION field

Opcode	Code	Description
NOP	0	
BLANK	0	
MPL	1	flag BUS < accumulator
MPS	2	flag BUS > accumulator
MPK	3	flag processor not in kernel mode
MPO	4	flag BUS <> accumulator
LSBS	5	Skip if least sig bit set
UNMAP	6	Turn on Unmap bit

Table C6 - the SPECIAL field

The MEMORY control field

This field allows the intermediate processor to start memory references. The processor may issue a read request, a write request, or a conditional request. The latter kind will be either a read or a write depending on the type of operation which the HP2100 requested of the intermediate processor. The allowable operands are shown in Table C7.

Opcode	Code	Description
NOP	0	
BLANK	0	
READ	1	Perform a memory Read
WRITE	2	Perform a memory Write
RW	3	Perform a Read or a Write

Table C7 - the MEMORY control field

The remainder of this appendix contains the microcode used by the intermediate processor. The first section contains the basic MONADS II instruction set, while the second section contains microcode used for initial bootstrapping and diagnostic purposes.

```
Label   Bus 1   I-O   Bus 2    Const    Func   Spec   Mem    Target   Comment

*
*       INTERMEDIATE PROCESSOR MICRO CODE
*       VERSION          9
*       REVISION         2
*       RELEASE          3
*       AUTHOR           DAVID ABRAMSON
*       DATE.            29/2/80
*       CREATED.         29/5/80.
*       MODIFIED         1/9/80.
*       MODIFIED.        29/9/80.
*       MODIFIED.        14/8/80.
*       MODIFIED.        03/12/80.
*       MODIFIED.        11/12/80.   ADDITION OF DEBUG CODE
*       MODIFIED.        16/02/81.   ADDITION OF CLEAR C.ACCS
*       MODIFIED.        03/04/81    DELETION OF SUBSYSTEM CHECK
*       MODIFIED.        07/10/81    ADDITON OF ++(T) AND (T)--
*       MODIFIED.        08/12/81    SPEEDUP CODE FETCHES
*
*       IMMEDIATE LOAD + OR - I
*       LSB IS SIGN
*
TYPE0   MAR     ONTO          177400  OR     LSBS             GET BYTE
                                      JUMP             POS    POSITIVE?
        MDR     INTO                                          NO
        MDR     ONTO                  RIGHT                   GET 7 BITS
        MDR     INTO
        MDR     ONTO          177400  OR                      SIGN EXTEND
        MDR     INTO                  END                     RETURN
                                      NOP
*
*       THIS CODE HANDLES REFERENCES OF THE FORM
*       B(B) + N
*
TYPE1   C.ASN   ONTO  ADRSPC                 UNMAP            GET ASN
        MAR     ONTO          7       AND                     GET N
        C.DISP  ONTO          ACC     ADD                     FORM B+N
        DISP    INTO                                          SAVE IT
        C.LEN   ONTO          ACC     SUB    MPL              CHECK LEN.
        C.ACCS  ONTO  ACCESS                                  CHECK ACC
                                                RW
                                      END                     RETURN
*
*       THIS CODE HANDLES REFERENCES OF THE FORM:
*       B(B) + M(M)
*
TYPE2   C.ASN   ONTO  ADRSPC                 UNMAP            GET ASN
        M       ONTO          77777   AND                     ADD IN MOD.
        C.DISP  ONTO          ACC     ADD                     AND DISPL.
        DISP    INTO                                          SAVE IT
        C.LEN   ONTO          ACC     SUB    MPL              CHECK LEN.
        C.ACCS  ONTO  ACCESS                                  CHECK ACC
                                                RW
                                      END
*
*       READ A BYTE FROM WORD HELD AT BYTE COUNT IN MODIFIER

Label   Bus 1   I-O   Bus 2    Const    Func   Spec   Mem    Target   Comment
```



```
Label   Bus 1   I-O   Bus 2    Const    Func   Spec   Mem    Target   Comment

*
TYPE3   C.ASN   ONTO  ADRSPC                                       GET ASN
        M       ONTO                  RIGHT                       GET WORD NO
        C.DISP  ONTO          ACC     ADD                         FORM B+M
        DISP    INTO                                              SAVE IT
        C.LEN   ONTO          ACC     SUB    MPL                  CHECK LEN.
        C.ACCS  ONTO  ACCESS                                      
                                                READ
        M       ONTO          0       ADD    LSBS                 LEFT OR RIGHT?
                                      JUMP             LHS.R
RHS.R   MDR     ONTO          377     AND                         RIGHT BYTE
        MDR     INTO                  END                         RETURN
LHS.R   MDR     ONTO                  SWAP                        SWAP BYTES
        MDR     INTO                                              SAVE IT
                                      JUMP             RHS.R      RETURN
                                      NOP
                                      NOP
*
*       CODE TO HANDLE A WRITE INTO WORD
*       WHERE THE MODIFIER HOLDS A BYTE COUNT
*
TYPE5   C.ASN   ONTO  ADRSPC
        MDR     ONTO          377     AND                         GET DATA
        TR1     INTO                                              SAVE IT
        M       ONTO                  RIGHT                       WORD NO?
        C.DISP  ONTO          ACC     ADD                         FORM B+M
        DISP    INTO                                              SAVE IT
        C.LEN   ONTO          ACC     SUB    MPL                  CHECK LENGTH
        C.ACCS  ONTO  ACCESS                                      AND ACCESS
                                                READ
        M       ONTO          0       ADD    LSBS                 LEFT OR RIGHT?
                                      JUMP             LHS.W      LEFT
RHS.W   MDR     ONTO          177400  AND                         SAVE LEFT
RHS+1   TR1     ONTO          ACC     OR                          PUT RIGHT IN
        MDR     INTO                                  WRITE       SAVE IT
                                      END                         RETURN
                                      NOP
*
*       READ A B SUBSYSTEM NUMBER
*
TYPE7   C.SSN   ONTO  MDR             END                         GET DATA
                                      NOP
                                      NOP
                                      NOP
*
*       HANDLE POSITIVE CASE OF TYPE 0
*
POS     MAR     ONTO          377     AND                         GET 3 BITS
        MDR     INTO                                              OF DATA
        MDR     ONTO                  RIGHT                       GET 7 BITS
        MDR     INTO                  END                         RETURN
*
*       LOAD A B SUBSYSTEM REGISTER
*
TYPE8   MDR     ONTO          0       ADD                         SAVE DATA

Label   Bus 1   I-O   Bus 2    Const    Func   Spec   Mem    Target   Comment
```

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

        C.SSN   INTO          ACC     END                        RETURN
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       READ A C.ASN REGISTER
*
TYPE9                                         MPK                PROTECTED
        C.ASN   ONTO  MDR               END                        GET DATA
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       LOAD A C.ASN REGISTER
*
TYPE10                                        MPK                PROTECTED
        MDR     ONTO          0       ADD                        GET DATA
        C.ASN   INTO          ACC     END                        RETURN
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       READ A DISPLACEMENT REGISTER
*
TYPE11  C.DISP  ONTO  MDR             END                        GET DATA
*
*         THIS IS THE EXTENSION OF THE BYTE WRITE
*
LHS.W   TR1     ONTO                  SWAP                       SWAP BYTES
        TR1     INTO                                             SAVE IN TR1
        MDR     ONTO          377     AND                        GET BOTTOM BYTE
                                      JUMP                RHS+1  TREAT AS RHS
                                      NOP
                                      NOP
                                      NOP
*
*       LOAD A DISPLACEMENT REGISTER
*
TYPE12  MDR     ONTO          0       ADD    MPK                 SAVE DISPLACEMENT
        C.DISP  INTO          ACC     END                        RETURN
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
```



```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

*       READ THE C.LENGTH REGISTER
*
TYPE13  C.LEN   ONTO  MDR             END                        GET LENGTH
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       LOAD THE C.LENGTH REGISTER
*
TYPE14  MDR     ONTO          0       ADD    MPK                 SAVE MDR IN
        C.LEN   INTO          ACC     END                        LENGTH
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       READ A C.ACCESS REGISTER
*
TYPE15  C.ACCS  ONTO  MDR             END                        GET ACCESS
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       DECREASE THE C REGISTER ACCESS RIGHTS
*
TYPE16  C.ACCS  ONTO          0       ADD                        GET ACCESS
        MDR     ONTO          ACC     OR                         MASK BITS
        C.ACCS  INTO                  END                        SAVE AWAY
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       PRIVILEDGED CODE
*       LOAD A C.ACCESS REGISTER
*
TYPE17                                        MPK                PROTECTED
        MDR     ONTO          0       ADD                        SAVE ACCESS
        C.ACCS  INTO                  END                        RIGHTS
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
```

```
Label    Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

*
*        INCREMENT COUNTER REGISTER
*        AND MODIFIER OF SAME NUMBER
*        RETURN COUNTER
*
TYPE18   M       ONTO          1       ADD                          INCREMENT
         M       INTO                                               MODIFIER REG
         C       ONTO          1       ADD                          INCREMENT
         C       INTO                                               COUNTER REGISTER
         MDR     INTO                  END                          RETURN COUNTER
                                       NOP
                                       NOP
                                       NOP
*
*        STORE IN COUNTER
*        CLEAR MODIFIER
*
TYPE19   MDR     ONTO          0       ADD                          DATA INTO
         C       INTO                                               COUNTER
         M       ONTO          0       AND                          CLEAR
         M       INTO                  END                          MODIFIER
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*        CODE TO HANDLE T - T
*
TYPE20   CSAS    ONTO  ADRSPC                                       STACK SPACE
         MAR     ONTO          377     AND                          GET T
         T       ONTO          ACC     SUB                          T-T
         DISP    INTO                                               SAVE IT
         TB      ONTO          ACC     SUB    MPS                   CHECK LENGTH
                                                     RW
                                       END                          RETURN
                                       NOP
*
*        READ A MODIFIER REGISTER
*
TYPE21   M       ONTO  MDR             END                          GET DATA
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*        LOAD A MODIFIER REGISTER
*
TYPE22   MDR     ONTO          0       ADD                          RETURN DATA
         M       INTO                  END
                                       NOP
                                       NOP
                                       NOP
```


Label Bus 1 I-O Bus 2 Const Func Spec Mem Target Comment

```
                                       NOP
                                       NOP
                                       NOP
*
*        READ A COUNTER REGISTER
*
TYPE23   C       ONTO  MDR             END                          GET DATA
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*        LOAD A COUNTER REGISTER
*
TYPE24   MDR     ONTO          0       ADD                          SAVE DATA
         C       INTO                  END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*        CODE TO HANDLE T PUSH INSTRUCTION
*
TYPE25   CSAS    ONTO  ADRSPC                                       GET STACK
         T       ONTO          1       ADD                          T+1
         DISP    INTO                                               SAVE IT
         TB      ONTO          ACC     SUB    MPS                   CHECK LEN
                                                     RW              DO R/W
         T       ONTO          1       ADD                          REFORM T+1
         T       INTO                                               SAVE IT
                                       END                          RETURN
*
*        CODE TO HANDLE T POP INSTRUCTION
*
TYPE26   CSAS    ONTO  ADRSPC                                       STACK
         T       ONTO  DISP    0       ADD                          GET T
         TB      ONTO          ACC     SUB    MPS                   CHECK LENGTH
                                                     RW              DO R/W
         T       ONTO          1       SUB                          FORM T-1
         T       INTO          ACC                                  SAVE IT
                                       END
                                       NOP
*
*        READ THE TOP OF STACK REGISTER
*
TYPE27   T       ONTO  MDR             END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
```

Label Bus 1 I-O Bus 2 Const Func Spec Mem Target Comment

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

                                      NOP
                                      NOP
                                      NOP
*
*       LOAD THE TOP OF STACK REGISTER
*
TYPE28  MDR     ONTO          0       ADD
        T       INTO                  END
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       READ THE L1 REGISTER
*
TYPE29                                      MPK                PROTECTED
        L1      ONTO  MDR             END
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       LOAD THE L1 REGISTER
*
TYPE30                                      MPK                PROTECTED
        MDR     ONTO          0       ADD
        L1      INTO                  END
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       READ THE L2 REGISTER
*
TYPE31  L2      ONTO  MDR             END
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       LOAD THE L2 REGISTER
*
TYPE32                                      MPK
        MDR     ONTO          0       ADD
        L2      INTO                  END
                                      NOP
                                      NOP
```

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

```
                                      NOP
                                      NOP
                                      NOP
*
*       READ THE  CONSTANT ADDRESS SPACE NUMBER
*
TYPE33                                                  MPK
        CASN    ONTO  MDR             END
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       LOAD THE CONSTANT ADDRESS SPACE NUMBER
*
TYPE34                                                  MPK
        MDR     ONTO          0       ADD
        CASN    INTO                  END
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       READ THE CURRENT CODE ADDRESS SPACE NUMBER
*
TYPE35                                                  MPK
        CCAS    ONTO  MDR             END
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       LOAD THE CURRENT CODE ADDRESS SPACE NUMBER
*
TYPE36                                                  MPK
        MDR     ONTO  DISPL2  0       ADD
        CCAS    INTO                  END
                                      NOP
                                      NOP
                                      NOP
                                      NOP
                                      NOP
*
*       READ THE CURRENT STACK ADDRESS SPACE NUMBER
*
TYPE37                                                  MPK
        CSAS    ONTO  MDR             END
                                      NOP
                                      NOP
                                      NOP
```

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

                                       NOP
                                       NOP
                                       NOP
*
*       LOAD THE CURRENT STACK ADDRESS SPACE NUMBER
*
TYPE38                                        MPK
        MDR     ONTO          0        ADD
        CSAS    INTO                   END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*       READ THE CURRENT PROCESS NO
*
TYPE39  PROCN   ONTO  MDR              END
*
*       MORE CODE FOR THE CONTEXT SWITCH
*
MORE    INST2   ONTO          0        ADD
        PINST2  INTO
        MDR     ONTO  PROCN
        PTIME1  ONTO  WCHDG1
        PTIME2  ONTO  WCHDG2
        PINST1  ONTO  INST1
                                       JUMP              MORE1
*
*       CODE FOR A CONTEXT SWITCH INSTRUCTION
*
TYPE40                                        MPK
        WCHDG1  ONTO          0        ADD
        PTIME1  INTO
        WCHDG2  ONTO          0        ADD
        PTIME2  INTO
        INST1   ONTO          0        ADD
        PINST1  INTO
                                       JUMP              MORE
*
*       READ THE CURRENT SUBSYSTEM NUMBER
*
TYPE41  CSSN    ONTO  MDR              END
*
*    EVEN MORE CODE FOR THE CONTEXT SWITCH
*
MORE1   PINST2  ONTO  INST2
        CSSN    ONTO  DISPL1
        CCAS    ONTO  DISPL2           END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*       LOAD THE CURRENT SUBSYSTEM NUMBER
```


Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

```
*
TYPE42                                        MPK
        MDR     ONTO  DISPL1  0        ADD
        CSSN    INTO                   END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*       READ THE STACK VIOLATION REGISTER
*
TYPE43                                        MPK
        SVR     ONTO  MDR
        SVR     INTO                   END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*       HIT THE LINK WORD
TYPE44                                        MPK
                                       KNLOFF
                                       END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*       GET FIRST WORD OF PTIME
*
TYPE45  WCHDG1  ONTO  MDR              END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*       GET WORD 2 OF PTIME
*
TYPE46  WCHDG2  ONTO  MDR              END
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
                                       NOP
*
*       SET UP WORD 1 OF PTIME
*
```

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

TYPE47                                        MPK
        MDR     ONTO          0         ADD
        WCHDG1  INTO                    END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       SET UP WORD 2 OF PTIME
*
TYPE48                                        MPK
        MDR     ONTO          0         ADD
        WCHDG2  INTO                    END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       GET MOST SIG WORD OF NO INSTRUCTIONS
*
TYPE49  INST1   ONTO  MDR               END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       GET LEAST SIG WORD OF NO OF INSTRUCTIONS
*
TYPE50  INST2   ONTO  MDR               END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       GET MOST SIG WORD OF STACK TIME
*
TYPE51  STIME1  ONTO  MDR               END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       GET LEAST SIG WORD OF STACK TIME
*
```


Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

```
TYPE52  STIME2  ONTO  MDR               END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       UPDATE TOP OF STACK REGISTER
*
TYPE53  T       ONTO          0         ADD
        MDR     ONTO          ACC       ADD
        T       INTO                    END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       UPDATE L1 REGISTER
*
TYPE54                                        MPK
        L1      ONTO          0         ADD
        MDR     ONTO          ACC       ADD
        L1      INTO                    END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       UPDATE L2 REGISTER
*
TYPE55                                        MPK
        L2      ONTO          0         ADD
        MDR     ONTO          ACC       ADD
        L2      INTO                    END
                                        NOP
                                        NOP
                                        NOP
                                        NOP
*
*       UNMAPPED VERSION OF B REGISTER REFERENCE
*       ONLY USED IF INDIRECT AND UNMAP DIRECTIVE ISSUED
*
TYPE56  MAR     ONTO          77777     AND
        DISP    INTO
        C.LEN   ONTO          ACC       SUB     MPL
        C.DISP  ONTO          0         ADD
        DISP    ONTO          ACC       SUB     MPL
                                                      RW
                                        END
                                        NOP
*
*       REFERENCE TO L1 + N
```

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

*
TYPE57  CSAS    ONTO  ADRSPC                                       STACK
        MAR     ONTO          377     AND
        L1      ONTO          ACC     ADD
        DISP    INTO
        L1L     ONTO          ACC     SUB    MPL
                                                  RW
                                      END
                                      NOP
*
*       REFENCE TO L2 + N
*
TYPE58  CSAS    ONTO  ADRSPC                                       STACK
        MAR     ONTO          377     AND
        L2      ONTO          ACC     ADD
        DISP    INTO
        L2L     ONTO          ACC     SUB    MPL
                                                  RW
                                      END
                                      NOP
*
*       CODE ADDRESS SPACE REFERENCE
*
TYPE59  CCAS    ONTO  ADRSPC                                       CODE
        MAR     ONTO  DISP                      RW                 WHERE
                                      END
                                      NOP
*
*      LOAD THE T BASE REGISTER
*
TYPE65                                        MPK
        MDR     ONTO          0       ADD
        TB      INTO                  END
                                      NOP
*
*       DATA ADDRESS SPACE REFERENCE
*       BOTTOM 512 WORDS
*
TYPE60  CASN    ONTO  ADRSPC                                       DATA
        MAR     ONTO  DISP
                                                  RW
                                      END
*
*     WRITE TO THE L1 LIMIT REGISTER
*
TYPE67                                        MPK
        MDR     ONTO          0       ADD
        L1L     INTO                  END
                                      NOP
*
*      READ THE T BASE REGISTER
*
TYPE64  TB      ONTO  MDR               END
        NOP
*

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment
```



```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

*     READ THE L1 LIMIT REGISTER
*
TYPE66  L1L     ONTO  MDR             END
        NOP
*
*      READ THE L2 LIMIT REGISTER
*
TYPE68  L2L     ONTO  MDR             END
        NOP
PUSH1                                 JUMP             ++(T)1
                                      END
TYPE62                                JUMP             CLBVAL
        NOP
PUSH2                                 JUMP             ++(T)2
        NOP
POP1                                  JUMP             (T)--1
        NOP
POP2                                  JUMP             (T)--2
        NOP
*
*     THIS INSTRUCTION CLEARS THE PINST REGISTER
*
TYPE63  PINST1  ONTO          0       AND    MPK
        INST1   INTO
        INST2   INTO                  END
                                      NOP
*
*    WRITE TO THE L2 LIMIT REGISTER
*
TYPE69                                        MPK
        MDR     ONTO          0       ADD
        L2L     INTO                  END
                                      NOP
*
*
*    D E B U G      C O D E
*
*
*   THIS CODE STARTS THE DEBUG SECTION OF THE CONTROL STORE
*
*   IT IS ENTERED WITH A LOAD IMMEDIATE 0 INSTRUCTION
*   WITH THE DEBUG BIT ON
*
*   SECTION1.
*
*         THIS SECTION IS RESPONSIBLE FOR THE
*         TESTING OF THE PROCESSOR REGISTERS
*
*         THE FOLLOWING REGISTERS ARE CYCLED:
*
*               DISPL1
*               DISPL2
*               MDR
*               ADRSPC
*               DISP

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment
```

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

*                SVR
*                PROCN
*
*           IN ADDITION TO THESE, THE FISRT
*           PROCESS IN THE STACK IS TESTED
*
*
*       NB.
*
*             LOCATION 03E0 MUST HAVE AN END STATEMENT
*
*       OPERATION.
*
*         INTERNAL PRESET
*         EXTERNAL PRESET
*         BOOT
*         RUN
*
*         IF ALL IS WELL, CONTROL WILL BE PASSED
*         ON TO THE BOOTSTRAP
*
*         IF AN ERROR OCCURS, THE 2100 WILL EXECUTE A HALT XX
*         INSTRUCTION
*         THE CODE VALUE OF XX WILL ALSO APPEAR IN THE DISPL2
*         REGISTER, AND INDICATES THE CAUSE OF THE ERROR.
*
*
*         IF AN ERROR OCCURS, THE PROCESSOR WILL REMAIN
*         IN DEBUG STATE. THIS CAN BE RESET WITH INTERNAL PRESET
*
*
*
ENTRY   MAR     ONTO        7       AND                         GET LSB
        MAR     INTO        0       NOP     MPO                 =0?
                                    JUMP            TS1         YES
        MAR     INTO        1       NOP     MPO                 =1?
                                    JUMP            TS17        YES
        MAR     INTO        2       NOP     MPO                 =2?
                                    JUMP            CONFIG      YES
*
*
*     SECTION 2.
*
*     THIS CODE INITIALIZES THE SYSTEM BOOTSTRAP
*     IT CONFIGURES THE MAPS AS FOLLOWS:
*
*           100       40        LINKS
*           102       41        CODE
*           200       43        DATA
*           202       0         STACK
*
*
*      IF AN ERROR IS DETECTED IN SECTION 1, SECTION 2 WILL
*      NOT BE ENTERED
*      SECTION 1 MAY BE BYPASSED WITH A REG BIT 2 SET

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment
```



```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment

*
*
*
*       INTERNAL OPERATION IS AS FOLLOWS:
*
*             PROCN = 0
*             ADRSPC = 0
*             DISP = 100
*             CSSN = 0
*             CASN= 1         (ANY PATTERN WILL DO)
*             SVR =0
*             L1L = 77777
*             L2L = 77777
*             TB = 0
*             CCAS= 1
*
*             MDR = 177737
*             WRITE
*             DISP = 102
*             MDR = 177736
*             WRITE
*             DISP = 200
*             MDR = 177734
*             WRITE
*             DISP = 202
*             MDR = 77777
*             WRITE
*             MDR = 124002
*             DEBUG OFF
*             END;
*
CONFIG                                  0       AND
        PROCN   INTO
        TB      INTO
        CCAS    INTO
        CSSN    INTO
        ADRSPC  INTO
        CCAS    ONTO            1       ADD
        CCAS    INTO
        CASN    INTO
        SVR     INTO
*
        PROCN   ONTO            77777   ADD
        L1L     INTO
        L2L     INTO
*
        PROCN   ONTO            7       ADD
        TR1     INTO            1       ADD
        TR1     INTO                    LEFT
        TR1     INTO                    LEFT
        TR1     INTO                    LEFT
        TR1     INTO
        DISP    INTO
        TR1     ONTO                    RIGHT
        MDR     INTO            1       ADD

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem   Target   Comment
```

```
Label   Bus 1   I-O    Bus 2   Const   Func   Spec  Mem    Target  Comment

        PROCN   ONTO           ACC     SUB
        MDR     INTO
                                               WRITE
        MDR     ONTO           1       SUB
        MDR     INTO
        DISP    ONTO           2       ADD
        DISP    INTO
                                               WRITE
        TR1     ONTO                   LEFT
        DISP    INTO
        MDR     ONTO           2       SUB
                MDR     INTO
                                               WRITE
        DISP    ONTO           2       ADD
        DISP    INTO
        PROCN   ONTO           77777   ADD
        MDR     INTO
                                               WRITE
        TR1     ONTO           7       ADD
        TR1     INTO           1       ADD
        TR1     INTO           7       ADD
        TR1     INTO           1       ADD
        TR1     INTO
        TR1     ONTO                   LEFT
        TR1     INTO           7       ADD
        TR1     INTO           1       ADD
        TR1     INTO                   SWAP
        TR1     INTO           2       ADD
        MDR     INTO
                                       DBGOFF
                                       END
*
*       ENTRY TO TEST CODE
*
*
*       TEST CODE - TEST DISPL1 FOR 0
*
TS1                            0       AND                     DISPL1=0
        DISPL1  INTO
        DISPL1  ONTO           ACC     NOP    MPO              DISPL1<>
                                       JUMP           TS2     EQUAL
        DISPL2  INTO           1       ADD                     ERROR 1
        DISPL2  INTO   MDR
                                       JUMP           ERROR   GO ERROR
*
*       TEST CODE - TEST DISPL1 FOR -1
*
TS2                            177400  OR                      DISPL1=-
        DISPL1  INTO           377     OR
        DISPL1  INTO
        DISPL1  ONTO           ACC     NOP    MPO              DISPL2<>
                                       JUMP           TS3     EQUAL
                               0       AND                     ERROR 2
        DISPL2  INTO           2       ADD
        DISPL2  INTO   MDR

Label   Bus 1   I-O    Bus 2   Const   Func   Spec  Mem    Target  Comment
```



```
Label   Bus 1   I-O    Bus 2   Const   Func   Spec  Mem    Target  Comment

                                       JUMP           ERROR   GO ERROR
*
*       TEST CODE - TEST DISPL2 FOR 0
*
TS3                            0       AND                     DISPL2=0
        DISPL2  INTO
        DISPL2  ONTO           ACC     NOP    MPO              DISPL2<>
                                       JUMP           TS4     ALL OK
        DISPL2  INTO           2       ADD                     ERROR 3
        DISPL2  INTO           1       ADD
        DISPL2  INTO   MDR
                                       JUMP           ERROR   GO ERROR
*
*       TEST CODE - TEST  DISPL2FOR -1
*
TS4                            177400  OR                      DISPL2=-
        DISPL2  INTO           377     OR
        DISPL2  INTO
        DISPL2  ONTO           ACC     NOP    MPO              DISPL2<>
                                       JUMP           TS5     ALL OK
                               0       AND                     ERROR 4
        DISPL2  INTO           2       ADD
        DISPL2  INTO                   LEFT
        DISPL2  INTO   MDR
                                       JUMP           ERROR   GO ERROR
*
*       TEST CODE -  TEST MDR FOR 0
*
TS5                            0       AND
        MDR     INTO                                           MDR=0
        MDR     ONTO   DISPL1  ACC     NOP    MPO              MDR<>0
                                       JUMP           TS6     ALL OK
        DISPL2  INTO           2       ADD                     ERROR 5
        DISPL2  INTO                   LEFT
        DISPL2  INTO           1       ADD
        DISPL2  INTO   MDR
                                       JUMP           ERROR   GO ERROR
*
*       TEST CODE - TEST MDR FOR -1
*
TS6                            177400  OR                      MDR=-1
        MDR     INTO           377     OR
        MDR     INTO
        MDR     ONTO   DISPL1  ACC     NOP    MPO              MDR<>-1
                                       JUMP           TS7     ALL OK
                               7       AND                     ERROR 6
        MDR     INTO           1       SUB
        DISPL2  INTO   MDR
                                       JUMP           ERROR   GO ERROR
*
*       TEST CODE - TEST ADRSPC FOR 0
*
TS7                            0       AND                     ADRSPC=0
        ADRSPC  INTO
        ADRSPC  ONTO           ACC     NOP    MPO              ADRSPC<>

Label   Bus 1   I-O    Bus 2   Const   Func   Spec  Mem    Target  Comment
```

```
Label   Bus 1   I-O   Bus 2   Const    Func   Spec  Mem   Target   Comment

                                        JUMP                TS8      ALL OK
                              7         AND                          ERROR 7
        DISPL2  INTO  MDR
                                        JUMP                ERROR    GO ERROR
*
*       TEST CODE - TEST ADRSPC FOR -1
*
TS8                           177400    OR                           ADRSPC =
        ADRSPC  INTO          377       OR
        ADRSPC  INTO
        ADRSPC  ONTO          ACC       NOP    MPO                   ADRSPC<>
                                        JUMP                TS9      ALL OK
                              7         AND                          ERROR 8
        DISPL2  INTO          1         ADD
        DISPL2  INTO  MDR
                                        JUMP                ERROR    GO ERROR
*
*       TEST CODE - TEST DISP FOR 0
*
TS9                           0         AND                          DISP=0
        DISP    INTO
        DISP    ONTO          ACC       NOP    MPO                   DISP<>0
                                        JUMP                TS10     ALL OK
                              7         AND                          ERROR 9
        DISPL2  INTO          2         ADD
        DISPL2  INTO  MDR
                                        JUMP                ERROR    GO ERROR
*
*       TEST CODE - TEST DISP FOR -1
*
TS10                          177400    OR                           DISP=-1
        DISP    INTO          377       OR
        DISP    INTO
        DISP    ONTO          ACC       NOP    MPO                   DISP<>-1
                                        JUMP                TS11     ALL OK
                              7         AND                          ERROR 10
        DISPL2  INTO          2         ADD
        DISPL2  INTO          1         ADD
        DISPL2  INTO  MDR
                                        JUMP                ERROR    GO ERROR
*
*        END OF SINGLE REGISTER TESTS
*
*
*       TEST THE REGISTER STACK
*
*       ADRSPC = 200B
*       DISP = 20B          ;START OF STACK REGISTERS
*
TS11                          377       AND                          GET 377
        ADRSPC  INTO                    RIGHT                        GET 177
        ADRSPC  INTO          1         ADD                          GET 200
        ADRSPC  INTO
                              7         AND                          GET 7
        DISP    INTO          1         ADD                          GET 8
```


Label   Bus 1   I-O   Bus 2   Const   Func   Spec Mem   Target   Comment

```
Label   Bus 1   I-O   Bus 2   Const    Func   Spec  Mem   Target   Comment

        DISP    INTO          ACC       LEFT                         GET 20B
        DISP    INTO
*
                              0         AND                          PROCN=0
        PROCN   INTO
*
*       NOW SAVE 0 IN ALL REGISTERS ON STACK
*
        DISP    ONTO          0         ADD                          DISPL1=2
TS11-1  DISPL1  INTO
                              0         AND                          GET 0
        REGSTR  INTO                                                 ^DISPL1=
        DISPL1  ONTO          1         ADD
        ADRSPC  ONTO                    NOP    MPO                   DISPL1<>
                                        JUMP                TS11-4
                                        JUMP                TS11-1   MORE
*
*     NOW TRY TO READ THE REGISTERS BACK
*
TS11-4  DISP    ONTO          0         ADD                          GET 20B
TS11-2  DISPL1  INTO
        REGSTR  ONTO          0         NOP    MPO                   ^DISPL1<
                                        JUMP                TS11-3   ALL OK
        REGSTR  ONTO  ADRSPC                                         SHOW DAT
                              7         AND                          ERROR 11
        DISPL2  INTO          2         ADD
        DISPL2  INTO          2         ADD
        DISPL2  INTO  MDR
                                        JUMP                ERROR    GO ERROR
*
*       INCREMENT THE POINTER - ALL OK
*
TS11-3  DISPL1  ONTO          1         ADD
        ADRSPC  ONTO          ACC       NOP    MPO                   DISPL1=2
                                        JUMP                TS12     NEXT TES
                                        JUMP                TS11-2   GO AGAIN
*
*       END OF TEST FOR ZERO - ALL OK
*
*       NOW TRY TO WRITE 177777 TO ALL REGISTERS
*
TS12    DISP    ONTO          0         ADD                          GET 20B
TS12-1  DISPL1  INTO
                              177400    OR                           GET -1
        REGSTR  INTO          377       OR
        REGSTR  INTO
        DISPL1  ONTO          1         ADD                          INC DISP
        ADRSPC  ONTO          ACC       NOP    MPO                   DISPL1=2
                                        JUMP                TS12-4
                                        JUMP                TS12-1   GO AGAIN
*
*       NOW TRY TO READ THE REGISTERS BACK
*
TS12-4  DISP    ONTO          0         ADD                          DISPL1=2
TS12-2  DISPL1  INTO
```

Label   Bus 1   I-O   Bus 2   Const   Func   Spec Mem   Target   Comment

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec  Mem   Target   Comment

                                177400  OR                      GET -1
        DISPL2  INTO          377     OR
        REGSTR  ONTO          ACC     NOP    MPO              REGSTR<>
                                      JUMP                    ALL OK
        REGSTR  ONTO  ADRSPC                                  SHOW DATA
                                7       AND                   ERROR 12
        DISPL2  INTO          2       ADD
        DISPL2  INTO          2       ADD
        DISPL2  INTO          1       ADD
        DISPL2  INTO  MDR
                                      JUMP                    ERROR
*
*       INCREMENT THE POINTER
*
TS12-3  DISPL1  ONTO          1       ADD                     INC DISP
        ADRSPC  ONTO          ACC     NOP    MPO              DISPL1<>
                                      JUMP                    NEXT TES
                                      JUMP                    GO AGAIN
*
*       END OF -1 TEST
*
*
*
*       TEST THE REGISTER STACK BY WRITING
*       THE REGISTERS ADDRESS IN THE REGISTER
*
TS13    DISP    ONTO          0       ADD                     DISPL1=20
TS13-1  DISPL1  INTO
        REGSTR  INTO                                          SAVE DISP
        DISPL1  ONTO          1       ADD                     INC DISPL
        ADRSPC  ONTO          ACC     NOP    MPO              DISPL1=20
                                      JUMP                    TS13-4
                                      JUMP                    TS13-1  AGAIN
*
*       NOW TRY TO READ THE REGISTER BACK
*
TS13-4  DISP    ONTO          0       ADD                     DISPL1=20
TS13-2  DISPL1  INTO
        REGSTR  ONTO          ACC     NOP    MPO              REGSTR<>D
                                      JUMP                    ALL OK
        REGSTR  ONTO  ADRSPC                                  SHOW DATA
                                7       AND                   ERROR 13
        DISPL2  INTO                  LEFT
        DISPL2  INTO          1       SUB
        DISPL2  INTO  MDR
                                      JUMP                    ERROR   GO ERROR
*
*       INCREMENT POINTER
*
TS13-3  DISPL1  ONTO          1       ADD                     INC POINT
        ADRSPC  ONTO                  NOP    MPO              =200B?
                                      JUMP                    TS14
                                      JUMP                    TS13-2
*
*       END OF REGISTER STACK TEST

Label   Bus 1   I-O   Bus 2   Const   Func   Spec  Mem   Target   Comment
```



```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec  Mem   Target   Comment

*
*       TEST IF WE CAN CLEAR THE SVR?
*
TS14    SVR     INTO                                          SVR=0
        SVR     ONTO  DISPL1  0       NOP    MPO              =0?
                                      JUMP                    TS15
                              7       AND                     ERROR 14
        DISPL2  INTO          ACC     LEFT
        DISPL2  INTO  MDR
                                      JUMP                    ERROR   GO ERROR
*
*       TEST THAT PROCN REGISTER WORKS
*
TS15                          0       AND                     GET 0
        PROCN   INTO
        PROCN   ONTO  DISPL1  0       NOP    MPO              PROCN=0?
                                      JUMP                    TS16    ALL OK
                              7       AND                     ERROR 15
        DISPL2  INTO          ACC     LEFT
        DISPL2  INTO          1       ADD
        DISPL2  INTO  MDR
                                      JUMP                    ERROR   GO ERROR
*
*       TEST WRITE OF 377 INTO PROCN
*
TS16                          377     AND                     GET 377
        PROCN   INTO
        PROCN   ONTO  DISPL1  377     NOP    MPO              PROCN<>37
                                      JUMP                    TS17    ALL OK
                              7       AND                     ERROR 16
        DISPL2  INTO          ACC     LEFT
        DISPL2  INTO          2       ADD
        DISPL2  INTO  MDR
                                      JUMP                    ERROR
*
*       TEST THE DEDICATED MAP UNIT
*       EACH CELL OF THE MAP HAS ITS ADDRESS WRITTEN INTO
*       THE CELL
*
*       DISPL1=400B
*
TS17    ADRSPC  ONTO                  LEFT                    GET 400B
        DISPL1  INTO
                              0       AND                     SEGMENT 0
        ADRSPC  INTO
TS17-1  DISP    INTO                                          DISP 0
                              0       AND                     GET 0
        MDR     INTO                                          FOR 0-
        DISP    ONTO          1       ADD                     DISP+1
        MDR     ONTO          ACC     SUB                     FORM COMP
        MDR     INTO                                          DISP
                                              WRITE           DO WRITE
*
        DISP    ONTO          1       ADD                     FORM DISP
        DISPL1  ONTO          ACC     NOP    MPO              =400B?

Label   Bus 1   I-O   Bus 2   Const   Func   Spec  Mem   Target   Comment
```

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem    Target   Comment

                                      JUMP                 TS17-4
                                      JUMP                 TS17-1   GO AGAIN
*
*    ALL ADDRESSES SAVED - TRY TO READ THEM BACK
*
TS17-4                        0       AND                           AT START
TS17-2  DISP    INTO
                                                    READ            READ MAP
                                      NOP
                                      NOP                           WAIT!
        MDR     ONTO          0       ADD                           GET DATA
        DISP    ONTO          ACC     NOP    MPO                    EQUAL?
                                      JUMP                 TS17-3   YES
        DISPL1  INTO          ACC     NOP                           RETURNED
                              7       AND                           ERROR 17
        DISPL2  INTO                  LEFT
        DISPL2  INTO          1       ADD
        DISPL2  INTO          2       ADD
        DISPL2  INTO  MDR
                                      JUMP                 ERROR    GO ERROR
*
*       INCREMENT THE ADDRESS
*
TS17-3  DISP    ONTO          1       ADD                           INC DSIP
        DISPL1  ONTO                  NOP    MPO                    -400B?
                                      JUMP                 CONFIG
                                      JUMP                 TS17-2   AGAIN
*
*       ERROR REPORTING CODE
*
*       THIS CODE IS ENTERED WHEN AN ERROR IS DETECTED
*       THE ERROR NUMBER IS CONTRAINED IN DISPL2 AND MDR
*       ON EXIT THE MDR CONTAINS A HALT XX INSTRUCTION
*       DEBUG MODE IS NOT TURNED OFF
*
ERROR                         377     AND
        TR1     INTO          1       ADD                           FORM 400
        TR1     INTO                  LEFT                          1000
        TR1     INTO                  LEFT                          2000
        TR1     INTO          77777   ADD
        TR1     INTO          1       ADD
        MDR     ONTO          ACC     OR                            1020XX
        MDR     INTO
                                      END                           RETURN
*
*         THIS INSTRUCTION SETS ALL CAPABILITY REGISTERS INVALID
*
CLBVAL                        0       AND    MPK
        DISPL1  INTO          7       ADD                           GET 7
        DISPL1  INTO          1       ADD
        DISPL1  INTO                  LEFT                          GET 16
        DISPL1  INTO                  LEFT                          GET 32
        DISPL1  INTO                  LEFT                          GET 64
        DISPL1  INTO          7       ADD
        DISPL1  INTO          7       ADD

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem    Target   Comment
```

```
Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem    Target   Comment

        DISPL1  INTO          2       ADD
        DISPL1  INTO                                                GET 80
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND
        REGSTR  INTO
        DISPL1  ONTO          1       ADD
        DISPL1  INTO
*
                ONTO          1       AND

Label   Bus 1   I-O   Bus 2   Const   Func   Spec   Mem    Target   Comment
```

```
Label   Bus 1   I-O    Bus 2   Const   Func   Spec  Mem   Target  Comment

        REGSTR  INTO
        DISPL1  ONTO           1       ADD
        DISPL1  INTO
*
                ONTO           1       AND
        REGSTR  INTO
        DISPL1  ONTO           1       ADD
        DISPL1  INTO
*
                ONTO           1       AND
        REGSTR  INTO
        DISPL1  ONTO           1       ADD
        DISPL1  INTO
*
                ONTO           1       AND
        REGSTR  INTO
        DISPL1  ONTO           1       ADD
        DISPL1  INTO
*
                ONTO           1       AND
        REGSTR  INTO
        DISPL1  ONTO           1       ADD
        DISPL1  INTO
*
                ONTO           1       AND
        REGSTR  INTO
*
*       NOW RESET THE  DISPL1 REGISTER
*
        CSSN    ONTO   DISPL1
                                       END
*
*     CODE TO HANDLE ++(T) FIRST WORD
*
++(T)1  CSAS    ONTO   ADRSPC                                        STACK
        T       ONTO           2       ADD                           T+2
        DISP    INTO
        TB      ONTO           ACC     NOP   MPS
                                               WRITE
                                       END
*
*       CODE TO HANDLE ++(T) SECOND WORD
*
++(T)2  CSAS    ONTO   ADRSPC
        T       ONTO           1       ADD
        DISP    INTO
        TB      ONTO           ACC     NOP   MPS
                                               WRITE
        T       ONTO           2       ADD
        T       INTO
                                       END
*
*    CODE TO HANDLE (T)— FIRST WORD
*
(T)—1   CSAS    ONTO   ADRSPC
```



```
Label   Bus 1   I-O    Bus 2   Const   Func   Spec  Mem   Target  Comment

        T       ONTO   DISP    0       ADD
        TB      ONTO           ACC     NOP   MPS
                                               READ
                                       END
*
*       CODE TO HANDLE (T)— SECOND WORD
*
(T)—2   CSAS    ONTO   ADRSPC
        T       ONTO           1       SUB
        DISP    INTO
        TB      ONTO           ACC     NOP   MPS
                                               READ
        T       ONTO           2       SUB
        T       INTO
                                       END
*
$ORIGIN                 03E0
                                       END
$END
```

```
Label   Bus 1   I-O    Bus 2   Con
```

Appendix D

This appendix describes the properties of address space zero, which is used to communicate with the MONADS II address translation hardware. This address space is not mapped onto memory, but is recognized as a data pathway to the dedicated map units, the hash table address translator, and a register holding to address of the last page fault.

Each dedicated map table entry is 13 bits in length, and occupies one 16 bit word of the address space. Thus, the four dedicated map units occupy 256 words of the address space. Each hash table cell is 41 bits in length, and is split over 4 16 bits words. Thus, the 1024 word hash table occupies 4096 words of address space zero. The value of the last legal address is saved in a special register. When the operating system wishes to find out which page caused the last page fault it can read this register. The 31 bit address is accessible through 2 words of address space zero. An overall picture of address space zero is shown in Figure 1. The structure of each dedicated map cell entry is shown in Figure 2. The structure of each hash table cell is shown in Figure 3.

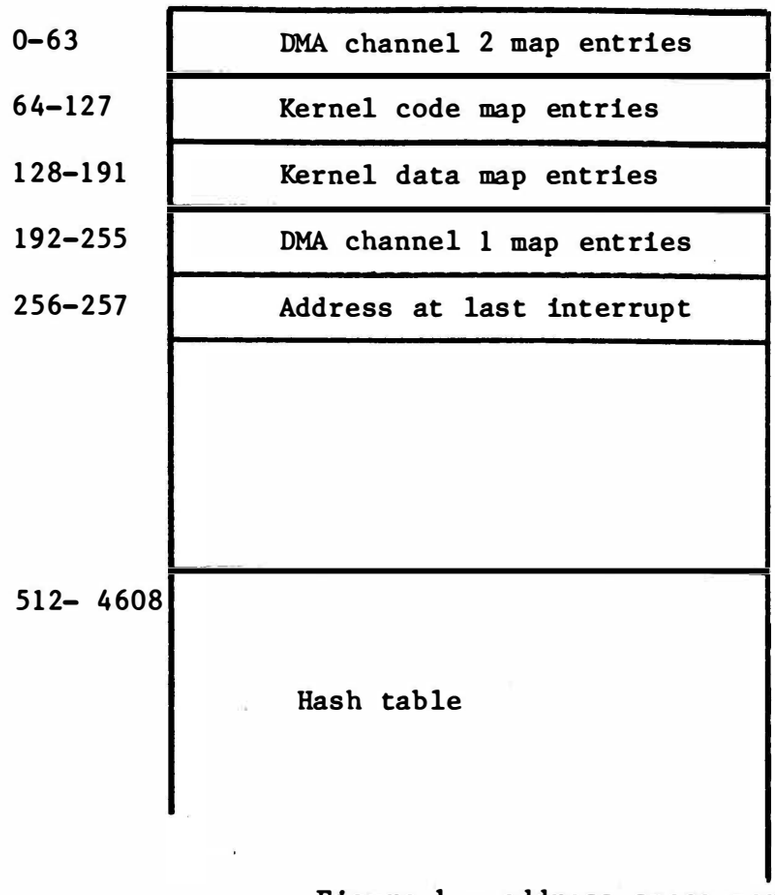


Figure 1 - address space zero

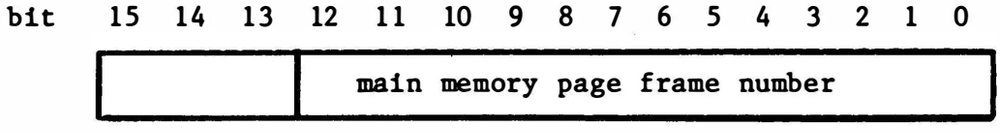
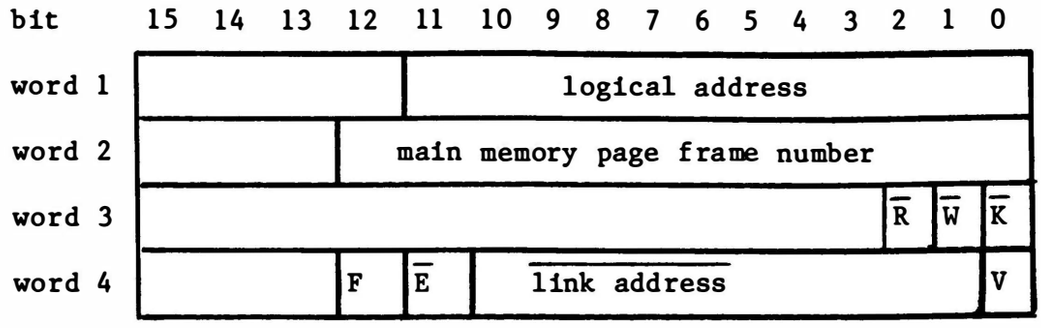


Figure 2 - a dedicated map entry



where:

F is foreigner bit
E is end of chain bit
V is valid bit
R is read access allowed
W is write access allowed
K is kernel access allowed

Figure 3 - a hash table map entry

Appendix E

This appendix contains copies of papers which have been published by the author relating to the work described in this thesis.

Abramson, D.A. (1982b) "Hardware for Capability Based Addressing", Proc. 9th Australian Computer Conference, Hobart.

Abramson, D.A. (1982a) "A Technique for Enhancing Processor Architecture", Proc. 5th Australian Computer Science Conference, Perth (Australian Computer Science Communications 4, 1, pp. 47-57).

Abramson, D.A. (1981) "Hardware Management of a Large Virtual Memory", Proc. 4th Australian Computer Science Conference, Brisbane (Australian Computer Science Communications 3, 1, pp. 1-13).

Hardware for Capability Based Addressing

David Abramson
Department of Computer Science
Monash University
Clayton

This paper examines a number of capability based computer systems and describes some outstanding problems. A new addressing model is proposed which not only alleviates these problems, but which is also efficient, flexible and uniform. The MONADS Series II computer, which implements the new model, is described. Finally, the effectiveness of the new solution is evaluated.

1. INTRODUCTION

Capability based addressing was first proposed by Dennis and Van Horn (1966) as a method for uniformly addressing and protecting objects in a multiprogrammed computer utility. Since that time a number of capability based computer architectures have been implemented, such as the Plessey 250 (England, 1972), Hydra (Wulf et. al., 1981), CAP (Wilkes, 1979), IBM System/38 (IBM, 1978) and the Intel iAPX432 (Intel, 1981), and a number have been proposed, such as the Chicago Magic Number Computer (Shepherd, 1968), and the schemes outlined by Bishop (1977) and Gligor (1978). This paper briefly describes the addressing mechanism common to these capability schemes, and shows how they implement the addressing structure. It then considers some outstanding problems , and proposes an alternative model. To demonstrate that the new model can be efficiently implemented, it was used as the central structure of the MONADS II computer.

2. CAPABILITY BASED ADDRESSING

2.1. Capabilities as an addressing system

A capability is a protected pointer which gives a program the ability to address an object (Fabry, 1974). A capability is logically composed of two fields, <object name> and <access rights>. The name field holds the name of the object which the capability addresses. The access rights field describes the way in which the object may be addressed by that capability. Capabilities possess a number of intrinsic properties:

- the object name is a unique name which defines the object.

- possession of a capability allows a program to address the object.

- there may be several capabilities for an object.

- a capability describes how the object may be addressed.

- a capability is not forgeable.

- object names are never reused, even after the object has been destroyed.

- capabilities facilitate easy sharing of objects.

- capabilities offer different views of the same object.

All current capability systems allow the access rights of a capability to be reduced, and a diminished copy of the capability to be given to another user. These capabilities (known as refined capabilities) then have access to the same object as the master, but with fewer access privileges. A capability may also be refined in range as well as type of access. This type of refinement is useful when a procedure wishes to grant another user access to only part of a data structure (e.g. when passing a parameter by reference).

2.2. Implementing Capability Addressing

Two different methods of addressing capabilities are usually used. One places the capabilities in a small list, called a Capability List (or C-list). The capability may then be addressed by supplying an index value into the list. The other, less commonly used, scheme places

capabilities in tagged memory (IBM, 1978). Both schemes have their advantages and disadvantages, which will not be discussed in this paper.

Because the unique name of an object is very large, and because it may not be reused, the virtual space of a capability system is much larger than the real memory attached to the processor. Thus, the processor hardware must translate a segment name into a real memory address (either main memory or secondary memory) before a reference can proceed. In addition to address translation, the system must maintain logical information about the object, such as its type. Almost all of the capability based processors use a central object table, which holds both the logical information and the mapping information (such as its main memory address and size) about every object in the system. This table is usually split into an active table (for currently addressed objects) and a passive table (for older objects) in an attempt to speed up address translation. Many different table organizations have been used, such as linear lists, directly indexed tables and hash tables and are used in systems such as Hydra, CAL, CAP, Gligor, Intel and Plessey.

Most of these systems place the active object table in main memory. To avoid the speed penality of accessing main memory on every memory reference, most systems apart from CAL, provide some hardware support to speed up address translation. The Plessey 250, Hydra and the Chicago Magic Number Computer use some manual addressing registers which are loaded with the main memory address of a segment before it is addressed. Other systems, such as Intel iPAX432, IBM System/38 and CAP, use automatic address translation caches, which retain the most frequently used object table entries.

Placing the active object table in main memory also limits its size significantly. To avoid this problem, Gligor places the object table in virtual memory. The scheme proposed by Bishop (1977) (and one of the IBM System/38 addressing methods) eliminates the need for a central object table by including in a segment capability a virtual address, which is also a unique name, and a size field. This also has the advantage that object size refinement is easily implemented. In such systems the object type is also placed in the capability.

2.3. Outstanding problems

The existing capability based computers have two main problem areas, memory management and address translation.

2.3.1. Memory Management

Many of the capability systems use a segmented main memory scheme in order to achieve segmented addressing. Unfortunately, this scheme does not cater well for either very large segments or for very small segments. Large segments are awkward because they must be held in contiguous memory. Small segments are inefficient to swap between main and secondary memory because the time taken to initiate the transfer may exceed the time taken to actually transfer the data. These problems have received much attention in the literature (Gligor, 1978; Lanciaux, 1977; Randel 1969; Fabry, 1974; Wilkes, 1979; Keedy, 1980).

Some systems have attempted to use paging as a basis for memory management. Hydra used a paging system by forcing all segments to be one fixed size. This scheme simplifies the memory management task, but does not solve the small and large segment problem. Small segments waste much of the page that they occupy, and large segments can not exist. Thus, this scheme creates even more segments than are logically required, as large segments are constructed from many smaller segments.

Some solutions (cf Gligor and Bishop) have used paging as the memory management model, and have superimposed a segmentation scheme above the virtual memory. Whilst these proposals have solved some of the small and large memory management problems, they still suffer from some memory management problems, as we shall see later.

2.3.2. Address Translation problems

Many of the capability based processors experience significant problems in translating virtual addresses into memory addresses, especially when the system is burdened with many small segments. One source of contention is the central object table which contains an entry for each segment in the system. When the system contains many small segments the size of the central object table becomes excessive, and translation times may be increased. In those systems which have removed the central object table, such as Bishop's, the task of address translation is significantly simplified.

In the processors which have used manual addressing registers with a segmented memory another problem is experienced. Because they hold a main memory address all registers (and all dormant images of registers) of all processors must be modified when main memory is reorganized. Such reorganization is required when segments are brought into and banished from main memory, and requires all absolute pointers to be modified. This overhead is considerable.

3. AIMS OF THE MODEL

The requirements of the model may be summarized in terms of five basic aims: to solve the memory management problems associated with most capability based processors, to solve the address translation problems associated with other capability based systems, to produce a uniform addressing mechanism, to produce an efficient capability addressing mechanism, and to produce a flexible hardware unit. Some of these aims are not shared by the existing capability systems. The first two requirements are associated with the outstanding problems discussed in section 2.3. We shall now consider the other three basic aims in turn.

3.1.1. Uniformity and simplicity

In a true capability based addressing scheme all local and permanent data should be addressed by the same mechanism. Only one way of addressing data should be provided, unlike systems such as the IBM System/38 which provide two different addressing mechanisms.

With one common addressing mechanism the system design becomes much simpler. A simpler design in not only easier to understand, but often yields a more orthogonal and less expensive implementation. Moreover, only one sharing and protection mechanism is required. The model proposed in this paper avoids unnecessary duplication by providing only one way of addressing memory.

3.1.2. Efficiency

The CAL system demonstrated that a capability based addressing scheme requires hardware support for an efficient implementation. Even in those systems which have provided hardware support for addressing memory, the use of capabilities still creates inefficiencies, as described in section 2.3. The model proposed in this paper defines a hardware addressing structure which can be efficiently implemented with current technology. Moreover, the model is capable of implementing many different software structures without significant overheads.

3.1.3. Flexibility

Most processors, both of conventional design and capability based, are designed with a specific addressing structure in mind. For example, the instruction operands in the Intel iAPX432 processor expect a particular C-list structure. The operands of the CAP system expect a different C-list structure. Because these organizations are so well understood by the processor hardware (and firmware) it is unlikely that one processor could efficiently or easily implement the C-list structure of another processor.

The lack of flexibility in some of the existing systems is not a problem, only because the system design does not change significantly at any stage. However, in a research environment a flexible processor is extremely desirable, as it allows the hardware to survive a number of major redesigns of the software ideas. The model proposed in this paper should be capable not only of efficiently implementing a particular addressing structure, but also of implementing any of the other capability addressing structures, such as the different C-lists of CAP, Intel iAPX432 etc. The model can achieve this flexibility by providing a general hardware unit which provides a capability based addressing style, and a small section of software (or firmware if the host machine is microcoded) which understands the addressing structure. If the software ideas change at any stage, then the hardware may remain the same and the software or firmware may be changed.

4. OBJECT ADDRESSING

Most capability based addressing schemes have the property that all addressable objects are treated alike in terms of addressing and protecting. All are addressed via the capability mechanism which the processor uses. Such references can be categorized into two classes, memory segments and high-level objects. High-level objects include I/O devices, data abstractions, program modules (Keedy, 1982) or type managers (Wulf et. al., 1981) etc.

When a memory segment is addressed (via memory reference instructions) the capability mechanism is used to find a segment of memory and make it available to the program. Thus, in a purely segmented system the central object table may contain the main memory address of the segment, and the size of the segment. The access rights field of the capability can then be used to restrict certain operations on the segment. To produce efficient memory references this mechanism is nearly always augmented by some special hardware.

High level objects are also addressed via the capability mechanism. However, the central object table contains information which declares that the object is not a memory segment and requires further software or firmware assistance. (Alternatively, this information may be held in the capability (Lampson, 1976).) These high level objects are not usually addressed by the normal memory reference instructions. Type checking information may then validate the type of instruction against the type of object. For example, a memory segment may be addressed by an add instruction, but a program module is addressed via a call instruction.

From this viewpoint, capability support can be built into a processor in two separate areas: first, a section of hardware which allows efficient manipulation of memory segments; second, a body of software, or firmware, which interprets operations on high level objects. Thus, the knowledge of high level objects need not be built into the processor. The information which usually resides in the central object table about high level objects can now either reside in the

capability for the object (e.g. the type of the object) or can be found in segments associated with the object itself (e.g. with the code which manipulates the object). The implementation of operations on high level objects is left entirely up to the software or firmware concerned. We will now consider the form of the memory segmentation hardware.

5. SEGMENT ADDRESSING

5.1. The basic form of a capability

The virtual memory of the proposed capability based addressing scheme is addressed via a number of capability registers, each of which holds a segment capability. These capability registers are the only addressing mechanism available to the processor. Each register, shown in Figure 1, contains three fields: an address, a length and some access rights. Before we discuss the precise nature of these fields, it will be useful to consider the advantages of a scheme based on registers:

(1) Because of the size of capabilities, they cannot be placed directly in the instruction stream itself. This problem of operand size for addressing memory via capabilities disappears in a register based system because once a register has been loaded with a capability subsequent references need only specify a register number, which is likely to be of the order of four bits.

(2) Registers hide the nature and structure of the logical addressing mechanism from the processor instruction set. The model is invariant to the method of saving capabilities (i.e. C-lists of various structures or tagged protected memory) and the actual structure of a C-list or tagged memory need not be determined at the hardware level (for example, whether the C-list allows tree structures or lattice structures). Thus, the scheme is flexible, because the software structures may be modified without affecting the hardware.

(3) Because registers can uniformly address all kinds of segment, no special registers are required, for example to implement a stack pointer, display registers, etc. Indeed, a combination of a capability register and an index register can be used not only to address data but also to control program sequencing.

(4) Because registers are normally built from high speed logic, they have the same advantages as capability caches (cf. IBM System/38, CAP and Intel iAPX432), but they are generally less expensive and in some cases easier to implement. Because the scheme only translates logical addresses (of the form C-list number and slot number) into capabilities when the register is loaded, it avoids many unnecessary memory references by removing many repeated references to the C-list.

(5) Given the use of registers, protection can be efficiently implemented by allowing only particular instructions (or only instructions executing in a special machine state) to modify their contents. This makes it impossible to modify a capability illegally once it has been placed in a register. The protection of capabilities outside of registers depends on the C-list structure, or tagging mechanism, which the processor provides.

A register based addressing scheme does have some basic disadvantages. First, it requires the compiler or assembler programmers to allocate and deallocate the registers. This problem is not considered serious enough to cancel the advantages of the scheme. First, assembler programmers are far better at judging the working set of a program than a cache, and can choose the correct registers to allocate. Second,

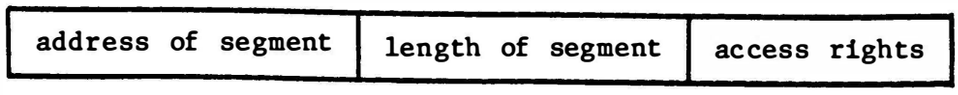


Figure 1 - a capability register

compilers often have to allocate data registers, and have successfully done so for a long time. Addressing registers are no worse to allocate correctly than data registers. Also, the compiler can form conventions which dedicate the use of certain registers. For example, one register may be used for addressing scalars at lexical level zero, whilst another register may be dedicated for addressing data at the current lexical level. Such conventions can help register allocation significantly.

Second, the registers may need to be saved and reloaded when a module is entered by a call instruction, or when a process switch is executed. Similar operations are also required for capability caches. Thus domain changes in the register scheme are not significantly slower than when a cache is used.

5.2. The load-capability-register instruction

Because the logical structure of the addressing mechanism is hidden from the hardware, special software (or firmware) must be written which understands this structure. One such instruction is the load-capability-register instruction. This instruction (or kernel routine if the machine does not possess a microcoded control unit) accepts a capability register number and a program address, and loads the capability found at that address into the register. If the processor uses a C-list for holding capabilities, then the program address may define a C-list number and a slot number. If the system must at some later stage understand a different C-list structure, then only the load-capability-register instruction need be altered. All other data manipulation instructions address their operands via a capability register.

5.3. Representation of a capability

A memory capability, shown in Figure 1, is composed of three sections: an address, a length and a set of access rights. The key difference between these registers and those of the other manual addressing register schemes is that our capability uses a virtual address, rather than a main memory address. As described in section 2.3, the use of main memory addresses both causes difficulties in re-organizing memory and also means that the main memory must be segmented. Apart from the difficulties of organizing a segmented memory, a central object table is required to map segment addresses onto main memory addresses which causes further problems related to the size of the mapping table, as discussed in section 2.3. The use of a virtual address in the capability registers avoids these problems. First, the memory can be physically reorganized without affecting the addresses held in registers. Second, the memory does not have to be segmented (from the viewpoint of the memory management system). This removes the problems of a segmented memory, and also means that the system does not need a central object table.

The length field of the capability holds the length of the segment, and must be large enough to allow large segments. Ideally, this field is the same size as the virtual address. However, it may be considerably less without being restrictive.

The access rights field must allow operations to be performed or restricted, such as read only, write only, read-write, execute etc. These can be encoded in a bit pattern.

6. VIRTUAL MEMORY

This section comprises three subsections. The first defines the nature of the virtual memory required by the capability register addressing scheme. The second describes a memory management model which provides some of the required attributes, and the third shows what modifications are necessary to this model to provide a virtual memory which may be addressed by the registers described in section 5.

6.1. Requirements of the virtual memory

In this section we will examine the requirements of the virtual memory which is used by the model. They are as follows:

(1) Virtual addresses should be large and unique. When a segment is created it consumes a range of virtual addresses, which eventually reside in C-lists and capability registers. When a segment is deleted, the address may either be found and destroyed, or never reused. A large addressing range means that it is not necessary to reuse addresses, saving on the number of addresses which need to be found and deleted.

(2) The virtual memory must be the only memory mechanism. This uniform treatment of memory means that all data, files and code, are present in the same virtual memory without support from a separate file store. This technique was pioneered in MULTICS (Organick, 1972) and has been used in other capability systems with many advantages (Rosenberg and Keedy, 1981).

(3) The tables, or mechanism, used to translate virtual addresses to main memory addresses should not affect the way in which the virtual memory management software organizes the secondary memory. This condition is not met in many existing systems, such as MULTICS. The page table structure which is used by the hardware, or firmware, to translate virtual addresses into main memory addresses is also used by the software to locate pages in secondary memory. If the software wishes to change the table format then the hardware may also need to be modified. Greater flexibility is desirable because better secondary storage methods may be devised after the hardware has been built. Thus secondary memory address translation and main memory address translation should be independent.

(4) Virtual memory management should be simple. If virtual addresses are ever reused, the virtual space may become fragmented due to objects being created and destroyed. Both Gligor and Bishop propose the use of large paged virtual memories for holding segments. Gligor packs segments into virtual space in a random manner, whereas Bishop places common segments in areas, or groups. The first scheme, whilst conceptually simple, means that the virtual space may become very fragmented in time. Bishop's scheme does not totally avoid this problem, as areas themselves are variable in size. The virtual memory should be organized so that if addresses are ever reused, the memory can be reorganized without massive data manipulation.

(5) The virtual memory should efficiently support both large and small segments. This problem is vastly simplified by implementing the segmentation at the register level. It then only becomes necessary for the virtual space to hold both large and small areas. All of the models previously discussed fail to provide an acceptable mechanism.

(6) Real memory management should be simple. Unlike the segmented schemes of some capability systems, the model can choose another main memory organization without losing the logical advantages of segmentation. Thus a simpler main memory scheme can be used instead of the complex and inefficient segmented scheme.

Unfortunately most virtual memory systems fail to provide a suitable virtual memory which supports these requirements. Another scheme, not previously discussed, allows a conventional processor to efficiently support small and large segments. The next section will discuss this model.

6.2. A small segment model

Keedy (1980) proposes a memory management model which allows a conventional processor to support both large and small segments without many of the associated inefficiencies. The scheme uses capabilities which hold a virtual address, segment length and access rights. The virtual address is further composed of an address space number and an offset within the address space. Each offset is composed of a page number and a within page displacement.

Address translation is performed via a number of tables. An address space list is consulted to find the location of the page table for the space. The page table reveals either the main memory address of the pages or the secondary memory addresses. This model is similar to various paged and segmented schemes and thus could be supported by a processor similar in nature to MULTICS. Unlike MULTICS, however, this model can support items 4,5 and 6 of the model aims, namely simple real and virtual memory management and support for small and large segments. All the advantages of the scheme are discussed in Keedy (1980). However, the following are particularly relevant:

6.2.1. Simple real memory management

The main memory is far easier to manage in this model than the segmented solutions because memory is allocated in fixed size pages. Provided some reference locality is experienced, several independently addressed and protected segments can be packed into a single address space, and the amount of space lost to internal fragmentation is on average only half a page per address space rather than half a page per segment (or more for small segments). Thus while internal fragmentation is not entirely eliminated, the amount of space wasted in this way can be greatly reduced.

6.2.2. Simple virtual memory management

The virtual memory is easier to manage than that of Gligor or Bishop because the virtual space is allocated in fixed size units, namely address spaces. Typically, because of reference locality, all the segments of a module are placed together in one or more address spaces. If the module is deleted, and all old addresses within the space are collected and destroyed, then the address of the address space may be reused. Because the address spaces are all of the same size, the hole left in the virtual space is not of a variable size, unlike those of Bishop and Gligor.

Even though the address spaces are all of a fixed size, spaces smaller than the maximum size do not actually require this fixed amount of disk space to be allocated. Thus, the scheme does not require any more disk space or page table space than other schemes.

6.2.3. Support for small and large segments

The scheme does not use a large central object table, but rather a smaller address space list, and can therefore support many small segments efficiently. As more segments are added to an address space, the address space list will remain the same size, and not grow like the central object tables in many of the capability systems. Moreover, provided that a reasonable amount of locality of reference is exhibited, many small related segments may be placed in one page, reducing the amount of wasted space and making segment swapping more efficient. Large segments may be composed of many pages. Because only those pages actually being addressed are held in main memory the scheme does not have the large segment problems experienced in segmented schemes.

Thus, the scheme solves both the memory management problems and the address translation problems associated with many small and large segments. However, the model in this form does not support requirements 1,2 and 3 of the model aims, namely large unique virtual addresses, a uniform memory and separate main and secondary memory address translation systems. The next section shows how the model can be modified and used to provide a virtual memory with all the required attributes.

6.3. Applying the memory management model

Requirements 1,2 and 3 of the model demanded a large uniform virtual memory and a separate main and secondary memory address translation system. A large uniform uniquely addressed memory which holds all data and files implies an address size of the order of 64 bits, as used in some other capability systems. The model described in section 6.2 implies an address size comparable to processors such as the ICL2900 (Keedy, 1977), MULTICS etc, and of the order of 32 bits because it uses page tables in main memory for address translation. Unfortunately, a simple scaling up of the tables is not possible because the large address is 2^32 times that of the conventional address. Moreover, the table structure would be used for both main memory and secondary memory address translation, contrary to the requirements set out in 6.1 (3). Thus, in order to use the memory model, the address size must be expanded to about 64 bits in size and another address translation mechanism must be found. We can consider a number of the techniques used by other capability based computers.

Gligor's addressing scheme assumes the presence of a robust virtual memory without indicating how to provide such a mechanism. Bishop attempts to use conventional page tables to translate addresses. This technique is unsuitable because of the size of the directly indexed page table. For the same reasons, the page and segment tables proposed by Keedy, and used by the ICL2900 series, MULTICS, Prime 750 (Prime) etc, are unsuitable because of the space required for the tables, and the time taken to translate an address.

The best form of address translation for an address of this size is the associative technique used by Atlas (Kilburn et. al. 1962), IBM System/38, MU6-G (Edwards et. al. 1980), and MONADS II (Abramson, 1981). These methods only attempt to translate addresses for those pages resident in main memory, and leave the software free to organize the

secondary memory translation tables in any suitable way (Rosenberg and Keedy, 1981).

Thus, by increasing the address size to 64 bits and by using an associative address translation scheme the Keedy model can provide an acceptable virtual memory for our capability model. The new addressing scheme is summarized in Figure 2.

7. MONADS

7.1. Background

The MONADS II computer was built in the Department of Computer Science at Monash University in 1980. The processor is constructed above an HP2100A minicomputer using the technique described in Abramson (1982), and uses the addressing structure described in this paper as the method for addressing memory.

7.2. Addressing structure

The MONADS II system supports the capability register scheme in two ways. First, the processor provides a virtual space of 2^31 words. This consists of 2^16 separate address spaces, in the sense described in this paper, each of 32k words. While a full scale capability system would ideally require more and larger address spaces (e.g. 2^32 by 2^32), the MONADS II addressing range is sufficient to demonstrate the principles involved and to support a pilot system.

Second, the processor provides 16 sets of registers. Thus, the system can efficiently support process-switching between 16 processes. Each register set includes 16 standard capability registers, six special

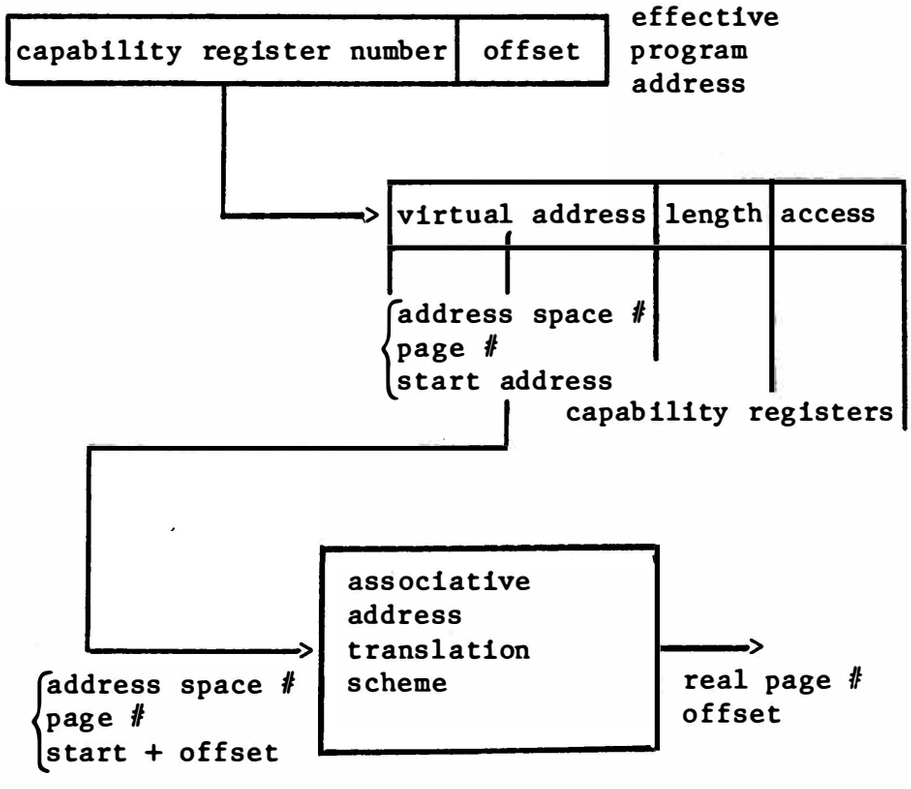


Figure 2 - The new addressing model

capability registers intended to address code, constants, base leaf links (Hewlett Packard, 1970) and scalars on the stack, and various other associated registers.

7.3. The capability registers

Each of the 16 processes executing on the HP2100A is provided with 16 capability registers for addressing segments of memory. Each register is composed of 4 16 bit words, as follows:

word 1: Address space number - 16 bits
word 2: Displacement within address - 15 bits
word 3: length of segment - 16 bits
word 4: access bits - read,write,kernel,invalid - 4 bits

The address space number defines one of the 32k word addressing regions in virtual memory. The displacement is used to mark the start of the segment in the address space. The length field marks the end of the segment in the address space. The read and write bits determine whether the segment may be read from or written into. The kernel bit specifies that the segment may only be addressed if the processor is in kernel mode. The invalid bit prevents the register from being used, and is set when a register is uninitialized. A capability register can only be loaded when the processor is in kernel mode (e.g. executing a load capability register instruction) and thus its contents are protected from corruption. Because the HP2100A only has 16 bit data pathways, four write cycles are required to set up each register. The HP2100A microcode provides a load capability register instruction of the type described in section 5.2.

A capability register may be used as an operand of any of the HP2100A memory reference instructions. When used, the 31 bit address is treated as a paged virtual address. This virtual address is mapped onto the main memory by the MONADS II address translation hardware, described in Abramson (1981). The displacement field is checked against the length field, and an interrupt is sent to the HP2100A if a violation occurs. If the mode of access is contrary to the read or write bits, or the kernel bit is set and the processor is not in kernel mode, or a register is invalid, an interrupt is sent to the HP2100A.

The displacement held in the register may be modified by two different methods. In the first, a small constant offset in the range 0 - 7 may be dynamically added to the value in the register. Alternatively a value held in a modifier register can be used to index into a segment defined by a capability register.

7.4. The load-capability-register instruction

There is not sufficient space in this paper to describe the details of the MONADS C-list structure. However, MONADS II uses a microcoded instruction to map this structure onto the capability registers. Before a segment can be addressed, the capability must be loaded into one of the 16 capability registers. Instructions may then address memory by specifying the 4 bit capability register number. Other instructions are provided for managing high level objects such as information hiding modules.

8. CONCLUSION

Many systems have difficulty in managing a memory addressed by capabilities. The model avoids many of these problems by using the memory management model described in section 6. In systems which use a central object table for segment address translation, the size of the table may become excessive if the processor addresses many small segments. The model proposed in this paper avoids this problem by eliminating the object table altogether. The scheme is uniform in two respects. First, the capability registers are the only way of addressing memory. Second, all data, regardless of its size or lifetime, is stored in the virtual memory. The efficiency of the solution is dependent on two main factors. First, sufficient capability registers must be available to contain the working set of the process (Denning, 1980). Capability registers must be allocated sensibly, and some hardware support is provided for domain changes. Second, an associative address translation scheme must be used. All of these are true in MONADS II. The hardware proposed in this paper was designed to be flexible enough to survive a number of changes in software ideas. Whilst there is not space in this paper to demonstrate the flexibility, the addressing model has actually been applied to a number of different C-list structures quite successfully. In each of these the model was capable of implementing a different addressing structure with only a different load-capability-register instruction and new high level object instructions.

ACKNOWLEDGEMENTS

This work was only possible after many hours of discussion with Les Keedy and John Rosenberg to whom I am eternally grateful. This paper would never have been finished without many more hours of help from Les Keedy. I am also grateful to the rest of the MONADS group who contributed significantly to the MONADS II processor and associated software.

REFERENCES

Abramson, D. (1981) "Hardware Management of a Large Virtual Memory", Proc. 4th Australian Computer Science Conference, Brisbane (Australian Computer Science Communications 3, 1, pp. 1-13).

Abramson, D. (1982) "A Technique for Enhancing Processor Architecture", Proc. 5th Australian Computer Science Conference, Perth (Australian Computer Science Communications 4, 1, pp. 47-57).

Bishop,P (1977) "Computer systems with a very large address space and garbage collection" PhD Thesis, MIT.

Denning P.J., (1980) "Working sets past and present" IEEE Transactions on Software Engineering, Vol SE-6 Number 1 pp 64 - 84.

Dennis.J, Van Horn,E (1966) "Programming semantics for multiprogrammed computations" Comms of ACM, Vol 9, No 3 pp 143-155.

Edwards D.B.G, Knowles A.E and Woods J.V. (1980) "The MU6-G: A new design to achieve mainframe performance from a mini sized computer", Proc. of the 7'th Annual Symposium on Computer Architecture, pp 161-167.

England,D.M. (1972) "Architectural features of the system 250" Infotech

state of the art report 14 on Operating systems. pp 395-426.

Fabry, R.S. (1974) "Capability Based Addressing" Comms. of ACM, Vol 17, Num 7, pp 403-412

Gligor,V. (1978) "Architectural implications of abstract data type implementations" Dept of Computer Science internal report, University of Maryland. TR-659.

Hewlett Packard (1970) "A Pocket Guide to the HP2100A minicomputer", Hewlett Packard Co., California, U.S.A.

Intel. (1981) "Introduction to the iAPX432 architecture" Intel corporation manual 171821-001.

IBM. (1978) "IBM System/38 Technical developments" IBM Corporation.

Keedy J.L. (1977) "An outline of the ICL2900 Series system architecture" The Australian Computer Journal Vol 9 Number 2, pp 53 - 62.

Keedy,J.L. (1980) "Paging and small segments: a memory management model" Proceedings of 8th World Computer Conference IFIP-80.

Keedy, J.L. "The MONADS View of Software Modules", Proc. 9th Australian Computer Conference, Hobart.

Kilburn T.,Edwards D.B.E.,Lanigan M.J. and Sumner F.H. (1962) "One Level Storage System", I.R.E Trans. Electronic Computation, EC-11, No 2, pp 223-234

Lampson,B., Sturgis,H (1976) "Reflections on an Operating System design" Comms of ACM, Vol 19, No 5, pp 251-265.

Lanciaux D, Schiller L, Wulf W "Supporting small objects in a capability system" Carnegie Mellon University, Internal report, Dec 1977.

Organick E.I. (1972) "The MULTICS System: An examination of its structure", MIT Press, Cambridge MAS & London.

Prime. "The System architecture reference guide" PDR 3060.

Randel, B. (1969) "A note on storage fragmentation and program segmentation" Comms. of ACM, Vol 12, Num 7, pp 365-372

Rosenberg J., Keedy J.L (1981) "Software Management of a Large Virtual Memory " Proc. 4th Australian Computer Science Conference, Brisbane (Australian Computer Science Communications 3, 1, pp 173-181.

Shepherd J.H., (1968) "The principle design features of the multi-computer Chicago Magic Number Computer" ICR quarterly report 19, Nov 1968, University of Chicago.

Wulf,W et al (1981) "HYDRA/Cmmp An experimental system" , McGraw-Hill

Wilkes,M.V.,Needham,R.M. (1979) "The Cambridge CAP computer and its operating system" North Holland.

A Technique for Enhancing Processor Architecture

D. Abramson
Dept. of Computer Science,
Monash University.

Abstract:

The MONADS II computer implements an architecture with a large segmented and paged virtual memory, an 'inprocess' stack organization and a capability based addressing scheme.

The processor is constructed around a 16 bit minicomputer, a HP2100A.

This paper describes the techniques used in the MONADS II processor to enhance a primitive architecture and proposes this technique as a general method of constructing research processors cheaply and quickly.

1. Introduction

Since the time that Charles Babbage designed his mechanical analytical engine in 1837, many new and different computer architectures have been proposed. The rapid changes in technique have enabled many structures to be built which previously could not be implemented. Moreover, the view of computer architecture has altered dramatically and has been affected by computer language research, providing architectures capable of directly supporting some high level languages [1] and operating systems.

Currently, much research is being conducted into architectures which, whilst basically VonNeuman, have new and different memory organizations (such as capability based addressing [2]) and which can manipulate higher level data constructs (such as sets and queues).

Many of these ideas are often only designed and documented at a conceptual level and are never actually implemented as the basic structure of a new processor. Unfortunately, many major design flaws are not discovered until an attempt is made to implement the design. Moreover, some designs cannot be implemented at all. Thus, a real implementation determines both that the ideas are basically sound and that they can be efficiently built with the available techniques.

A problem often faced is how to realize a new computer architecture in an environment in which resources are both expensive and limited, as in many Universities and research institutes.

This paper discusses some of the common practices and proposes an interesting technique.

2. Realizing a new architecture

A system designer is presented with two alternatives when attempting to implement a new architecture. First, the architecture can be incorporated into a totally new computer system. This approach, whilst logically the more desirable, often involves many more hours than may superficially appear necessary.

Extra devices (such as interfaces and controllers) must be constructed purely to operate the new processor. Some may require a large amount of design effort; effort which is not directly connected to the original architectural aims. Many software packages must then be developed, such as assemblers, compilers, loaders and bootstraps.

Consequently, the project often grows in size where 'large group management problems' are encountered. Much of this extra effort appears to be directed to the devices which must communicate with the processor, rather than to the processor itself. Thus, because of the extra effort involved, the full scale production of a new computer simply to test out some architectural enhancements is often not viable in a research environment.

The second alternative consists of modifying or using an existing computer system (called the 'source' architecture) in order to test out a new architectural design (called the 'target' architecture). This approach has the advantage that the design time and effort may be dramatically reduced. However, great care must be exercised to prevent the source architecture from restricting the scope and effectiveness of the target.

3. Using an existing computer system

Three different techniques may be used when the target architecture is constructed on top of a simpler source machine. First, an environment may be constructed in software. Second, if the source processor uses a microcoded control unit, the target may be implemented in firmware. Third, the actual hardware of the source processor may be modified to implement the target architecture.

3.1. A Software Emulation

This solution may take a number of forms. The most general is to produce a program (called the interpreter) which interprets instructions for the target machine. The interpreter emulates the fetch-execute cycle of the target processor, and executes target instructions by using small sections of source instructions. Interpreting the new architecture offers many advantages. Because the interpreter is a program, often written in a high level language, it may be easily modified. Complex debugging and monitoring aids may be incorporated in the design, allowing the designers to measure and judge the effectiveness of the new processor. At the same time as emulating the target architecture, the source machine may be executing many other programs.

This approach also has some major disadvantages. The ultimate execution speed of the target processor is often far too slow to support realistic tests . Moreover, it is not always obvious whether efficient hardware can later be constructed, somewhat diminishing the effectiveness of the implementation.

A slightly more efficient software emulation involves another different body of code (called the Kernel) which attempts to provide a normal source machine program with attributes from the target processor. Programs for the target machine are compiled into source machine instructions. When a target machine operation is required which cannot be directly translated into a short sequence of source instructions,a call to the kernel is executed, which performs the task and returns control to the source program.

Whilst far more efficient than an interpreter, the kernel solution tends to highlight the architectural features of both the source processor and the target, often with disastrous effects. (Such an example is found in [4]). Moreover, this technique may not be able to manage a target machine which is dramatically different in design from the source. Thus, a target program may be reduced mainly to kernel calls and appear the same as an interpretive solution.

Because many source instructions may be required to emulate a target instruction, the speed of the kernel is often far too slow to support a realistic test environment.

Many different types of kernel have been written. A good review is found in [16].

A common disadvantage is that both the kernel and the interpreter often occupy large amounts of memory and may reduce the space available for user programs significantly.

The advantages of these solutions are mostly logical. An interpretive solution can usually emulate the target architecture successfully. The disadvantages are mostly practical. Poor execution speed often makes the model useless.

In spite of the disadvantages, many architectures have been successfully emulated in software. Most, however, have failed to provide a usable, long term computer utility [4, 5] because of poor efficiency.

3.2. A Firmware Implementation

Another technique used is to emulate the target architecture in firmware (or microcode). This solution is clearly only applicable if the source machine uses a microcoded control unit and possesses a writable control store.

The internal microcycle of most processors is several times faster than their fetch-execute cycle. Consequently, target machine instructions can be much more efficiently emulated with microcode than with software. Because new instructions can be placed in writable control store, the processor can continue to execute normal source machine programs at the same time as target programs.

Unfortunately, most processors provide only a small writable control store and, more importantly, a limited number of uncommitted operation codes. Thus, it is usually difficult to microcode all of the operations required by the target machine.

Even when sufficient store and entry points are available, this technique often encounters another important problem. Many target instructions may implicitly require storage space, which must be provided by the source machine mainstore. (An obvious example is the implementation of a virtual memory system, which requires page tables in order to translate addresses). In many cases the fact that target operations are implemented in microcode may not be sufficient to make them efficient. The operations may be limited in speed by the time taken to scan or search various data structures which, if built into hardware, would have used much faster store and searching strategies. (examples of such an address translation system are found in [6] and [7]).

In addition, the structure of the micro instruction is usually designed for the source instruction set, not the target. Consequently, it is often quite difficult to write the target microcode on the source machine.

Thus, a firmware emulation, whilst much more efficient than a kernel or interpretive solution, is often still too slow to provide a usable system. Moreover, the implementation often leaves too much of the source processor architecture visible, affecting the attributes and view of the target architecture.

In the situation where speed is important, the only solution may be to provide special hardware.

3.3. Modifying the source hardware

The third possibility is to modify the hardware of an existing machine. Clearly, this technique can offer the best performance. Traditionally, however, this method is only used when the target architecture does not differ greatly from the source architecture.

Small changes such as small modifications to the instruction set, adding virtual memory hardware [8] and detecting extra error modes [5], have been done successfully. Each of these changes, however, has not introduced major architectural enhancements to the source processor.

In fact, it is clear that the major changes possible with an emulation environment are not always possible when modifying the hardware of an existing machine.

The technique is often rejected because it may alter the environment for normal source machine programs as well as target machine machine programs, dedicating the use of the source machine.

In spite of the disadvantages and practical difficulties a number of architectural changes have been achieved by hardware changes. The next section examines some of the more common hardware modification schemes used.

4. Hardware modifications

Many specific changes are possible when the processor design is modified. These depend upon the internal implementation of the processor itself, and will not be considered further. This section concentrates on some of the more general modification techniques available.

4.1. Processor Configurations

Most computer systems can be divided into two main parts, the CPU and the memory, connected usually by a 'clean' set of interface signals, shown in Figure 1.

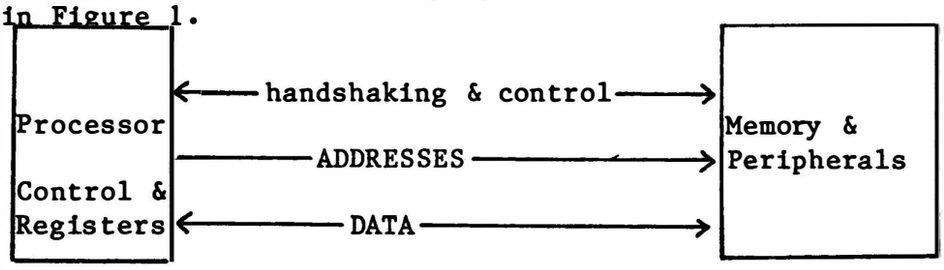


figure 1.

The signals involved in the interface can typically be divided into three sections; addresses, data and control/ handshaking information. The CPU communicates with the memory mostly by read and write commands. When the CPU executes a read operation control information is generated together with an address pattern. The CPU may then wait for data, which is passed back over the data pathway. When a write is executed data is sent with the address to the memory unit

The connections between the CPU and memory section may be generalized to form a system bus which connects to devices other than the memory.

It is the 'clean' nature of the interface between CPU and memory which is often employed when architectural enhancements are introduced.

4.2. Breaking the address bus

One technique used to enhance the architecture of the source processor is to introduce extra logic into the address pathway between the CPU and the memory, shown in Figure 2.

If the architectural enhancement is the addition of a virtual memory system, then the extra logic may be used to modify, or translate, the processor addresses before they reach the memory. Such a system is described in [8].

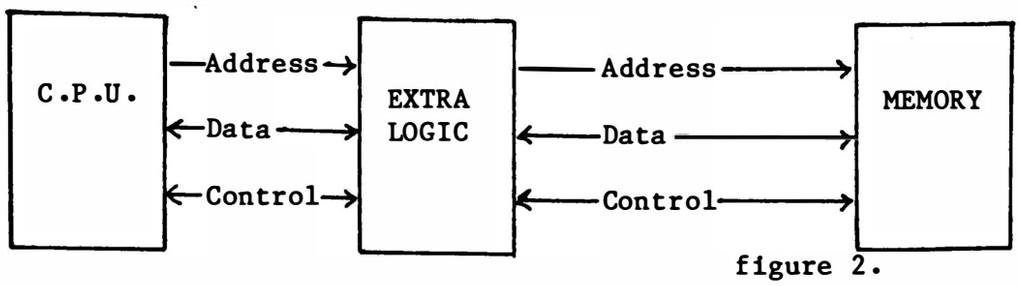


figure 2.

If however, the target architecture is to include more registers, these may be assigned addresses and placed in the extra logic. Read and write commands directed to these addresses are 'stolen' by the extra logic and may never reach the memory.

The extra logic in some systems appears as a block of memory, but the data in the locations is calculated by the logic rather than being the previously saved values. Such a system is described in [9] to implement a stack mechanism and addressing registers.

Many systems have been constructed which place special significance upon certain addresses within the address space. Many rely on special addresses for performing I/O operations. All, however, only 'steal' a limited number of addresses for such operations, and perform very specific operations. None of these systems make dramatic architectural changes. Such systems do, however, suggest that treating the addresses from a source processor in a special way may be used as a general mechanism for enhancing an existing machine architecture. The next section proposes such a model.

5. A general model

The systems discussed in section 4 used the processor addresses in various ways. If rather than using a dedicated piece of extra logic, another fast processor is placed in the address path, a general mechanism for dramatic architectural enhancements is created. In such a scheme, the processor addresses are treated as instructions by another, small fast processor, the intermediate processor, as shown in Figure 3. These new instructions may be tailored to the target architecture.

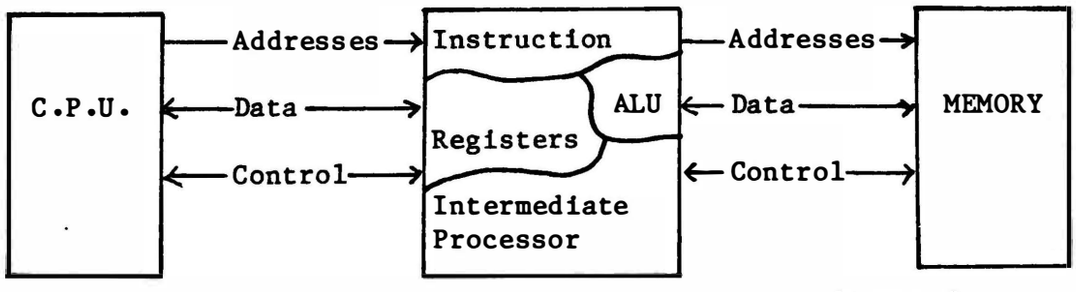


figure 3.

The intermediate processor reinterprets all of the CPU addresses, and executes them as though they were instructions. Some may be sent to the memory unit, whilst others may be used internally.

The intermediate processor appears as a piece of memory to the source processor.

The model possesses some particularly notable attributes.

i) Many new operation codes are available, thus many new target operations may be supported. The potential number of codes available is the size of the address space.

ii) Because the intermediate processor is a general processor many different target operations may be attempted, from very simple memory references to complex data manipulation.

iii) Extra target architecture registers may be located in the structure of the intermediate processor, and can be manipulated by read and write commands from the source processor.

iv) Normal memory references can be made to proceed from the source processor to the memory with very little delay.

v) Complex target operation may be added to the source without major modifications to the source processor hardware. Thus, the source processor may be a mainframe, a minicomputer or possibly even a microprocessor.

vi) The new architecture is partly transportable among source processors. Most of the target architecture is housed within the intermediate processor itself.

vii) The intermediate processor may be removed, or made logically transparent; thus it is not difficult to allow the source processor to execute normal source programs instead of target machine programs.

viii) The target architecture inherits all of the input/output devices, controllers, communications, frame and power supplies from the source processor. This vastly reduces the amount of effort required to implement a working target architecture.

ix) Depending upon the address interpretations it may be possible to execute source programs on the new target machine. At the very least, these programs can execute on another source processor of the same type. Thus the assemblers, compilers and loaders already available for the source processor may be modified to produce code for the target architecture. Consequently, some software development may be avoided.

x) Because the intermediate processor only consists of a central processor unit it may be easily constructed, possibly from bit slice components.

The next section examines the MONADS II processor, which was designed and built using this general model.

6. MONADS II

The MONADS project began in 1976 with the intention of investigating methods for developing large software systems. An operating system [10-14] was written to execute on MONADS I, a modified HP2100 minicomputer [8] [9].

The principles underlying the design of the operating system extend far beyond that system and can be applied to any large software system.

During the development of the MONADS I system it became obvious that the available hardware was not entirely suitable for the MONADS software structures. This prompted the building of a second computer, MONADS II, which is designed around the general model proposed in Section 5, and uses a HP2100A minicomputer as the source processor.

6.1. The HP2100

The HP2100 [15] is typical of many 16 bit minicomputers of the same era, and incorporates a microcoded control unit, some general accumulators and 32k words of memory. Addresses are constructed from a word of 16 bits, 15 of which form the memory address. The top bit represents whether addresses are to be used indirectly. [15, page 2-8].

Physically, the processor is divided into three areas; the central processor itself, the input output section and the memory section.

The interface between the memory section and the processor consists of 15 address bits, 16 bidirectional data bits and a number of control signals. The memory section is self contained and uses 16 bit words of core memory.

6.2. The MONADS II Architecture

It is beyond the scope of this paper to describe the architecture of the MONADS II system. It is, however, easy to demonstrate that the target architecture could never be efficiently emulated, either by software or firmware, totally within the HP2100A.

The MONADS II processor supports a number of processes directly in hardware and provides each process with 124 extra 16 bit registers. Some of these are used as capability registers to address a segmented virtual memory with 31 bit virtual addresses. The processor also supports the MONADS subsystem [13] and inprocess [12, 13] architecture with a protected stack structure. Most of these concepts are so alien to the HP2100A that an emulation would be extremely inefficient.

6.3. The MONADS II system

The MONADS II system comprises four sections; The HP2100A processor and I/O logic, the intermediate processor, the virtual memory manager and the system mainstore. The old HP2100A core controller has been removed and the interface is used to communicate with the intermediate processor, as shown in Figure 4.

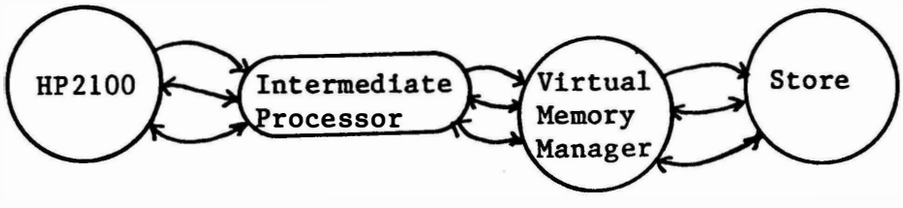


figure 4.

6.3.1. The HP2100 processor

The changes made to the HP2100 engine itself were minimal. Some of these were essential for the correct operation of MONADS II, whilst others were made for efficiency reasons. Four changes were made.

First, the microcode control store of the HP2100A was increased in size from 1024, 24 bit words to 4096 words. This modification whilst not essential, simplified the implementation and improved the efficiency of the operating system.

Second, the direct memory access (DMA) logic was modified to communicate directly with the virtual memory manager. This modification was essential, and guaranteed that the DMA system would always receive immediate service.

Third, minor changes were made to the interrupt logic to allow the intermediate processor to interrupt the HP2100A and abort the current instruction.

Fourth, small changes were made to the control signals between the HP2100A and the intermediate processor to make them asynchronous with respect to each other.

The core controller board and associated cards were removed from the frame and an interface to the intermediate processor substituted.

6.3.2. The intermediate processor

The intermediate processor is fast microcoded processor. Each instruction from the source processor is interpreted by a stream of microcode. It accepts all HP2100 addresses and reinterprets them according to the following rule:

```
if address is direct and =< 777B  then read from current data
  segment else
if address is indirect and =< 777B  then read from current link
  segment else
if address >= 1000B  and =< 1377B  then read from stack frame 1 else
if address >= 1400B  and =< 1777B  then read from stack frame 2 else
if address >= 2000B  and =< 75777B  then read from current code
  segment else
if address >= 76000B  and =< 77777B  then use another special set of
  interpretations.
```

The special interpretations allow the HP2100 to perform many other operations, including modifying registers, addressing the top of the stack, using push and pop operations on the stack, using and loading the capability registers, changing processes, reading the time, setting process time limits and addressing constants. The details of the address interpretations are beyond the scope of this paper and range widely in complexity.The simplest instruction manipulates a register whereas the most complex performs byte operations on word oriented segments. The intermediate processor is described in more detail in [17].

6.3.3. The memory manager and real memory

The intermediate processor is capable of expanding the 15 bit HP2100A address to a 31 bit virtual address. The virtual memory manager translates this address into a mainstore address and is described in detail in [18].

7. Achievements

The intermediate processor was designed by one person over a period of months, built in about 6 weeks and tested in about 2 weeks. The processor was implemented for two reasons.

First, and most obvious, to provide the MONADS group with a new processor capable of supporting the goals of the MONADS project.

Second, to determine whether the general model proposed in section 5 was realistic.

Both of these objectives were successfully met. The MONADS group is currently using the MONADS II system for software development. Moreover, the fact that such a complex architecture was developed efficiently on a very simple computer demonstrates that the model is

indeed realistic. The implementation appears to support the advantages cited in Section 5. The number of internal modifications made to the HP2100 was small. Whilst the processor is effectively dedicated to the MONADS project it would be possible to make the intermediate processor logically transparent and allow the HP2100A to execute normal programs.

Providing certain rules were obeyed it would also be possible to move the intermediate processor to another 16 bit minicomputer, with very little change to the intermediate processor.

Initially, a hardware diagnostic and monitor system was developed. This system was written in normal HP assembler and compiled and linked using the normal DOS-M assembler and loader software.

The effective speed possible within the intermediate processor would suggest that the implementation chosen is far more efficient than an interpretive or firmware solution. The design of the intermediate processor is significantly less complex than the design of a complete CPU module, and avoids all of the extra device logic described in Section 2.

An unexpected advantage was found in the initial bootstrap of the system. The intermediate processor has a special instruction which, when executed, sets up the address translation hardware and performs internal register diagnostics.

The most notable disadvantage of this technique, like the firmware solution, is that the source architecture is still somewhat visible. For example, the data paths in the target machine are still 16 bits wide. In spite of the simplicity of the source machine, these features did not appear to limit the target too significantly.

8. Conclusions

The success of the implementation of MONADS II clearly demonstrates that the model developed in Section 5 is realistic.

The end result of this research is a usable implementation of a new computer architecture.

Acknowledgements.

The design of the intermediate processor was only possible through many hours of discussion with Professor Chris Wallace, Dr. Les Keedy and Dr. John Rosenberg.

The technical staff, namely Mr. David Duke and Mr. Steve Garrison, did an excellent job of constructing the intermediate processor and memory management unit.

The bugs would never have been found without the help of Mr. Brian Wallis and the rest of the MONADS group.

References.

[1] Western Digital (1979) - "Pascal MICROENGINE Reference Manual" March.

[2] Fabry, R.S. (1974) - "Capability Based Addressing", Comms. of ACM, Vol. 17, No. 7 pp 403-412.

[3] Ramamohanarao, K. (1980) -"A New Model For Job Management Systems" PhD. Thesis. Monash University.

[4] Lampson, B., Sturgis, H. (1976) - "Reflections on an Operating System Design", Comms. of ACM, Vol. 19, No. 5, pp 251-265.

[5] Wulf, W. et al (1981) - "Hydra/Cmmp An Experimental System", McGraw-Hill.

[6] Sitton, W.G. & Wear, L.L. (1974) - "A Virtual Memory System for the Hewlett-Packard HP2100A", ACM 7th Annual workshop on Microprogramming, pp 119 - 121.

[7] D'Hautcourt-Carette, Francoise (1971), "A Micro programmed Virtual Memory for the Eclipse", SIGMICRO, ACM, June.

[8] Hagan, R. (1977) - "Virtual memory hardware for a HP2100A Minicomputer", M.Sc. Thesis, Monash University.

[9] Wallace,C.S. (1978) "Memory and Addressing Extensions to a HP2100A", Proc. of the 8th Australian Computer Conference.

[10] Keedy, J.L. (1978) - "The MONADS Operating System", Proc. of 8th Australian Computer Conference.

[11] Rosenberg, J. and Keedy, J.L. (1978) - "The MONADS hardware Kernel", proc. of 8th Australian Computer Conference.

[12] Ramamohanarao, K. and Keedy, J.L. (1978) "Job Management in The MONADS Operating System", Proc. of 8th Australian Computer Conference.

[13] Richards, I. and Keedy, J.L., (1978) "Subsystem Management in the MONADS Operating System" Proc. of 8th Australian Computer Conference.

[14] Georgiades, A., Richards, I. and Keedy, J.L. (1978) "A File System for the MONADS Operating System" Proc of 8th Australian Computer Conference.

[15] Hewlett Packard, "Hp2100A Pocket Guide", Hewlett-Packard Company, California, U.S.A.

[16] Rosenberg,J.(1979) "The Concept of a Hardware Kernel" PhD.Thesis Monash University.

[17] Abramson,D.A(1980) "A Users Guide to the MONADS Extended Hardware" MONADS Internal Report No 9.

[18] Abramson,D.A.(1980) "Hardware Management of a Large Virtual Memory". Proceedings of ACSC4, pp1-13.

HARDWARE MANAGEMENT OF A LARGE VIRTUAL MEMORY.

D. Abramson,
Dept. of Computer Science,
Monash University.

ABSTRACT

The MONADS II Computer was built in the Computer Science department at Monash University in 1980. Among its many features, the Series II utilizes a capability based addressing scheme, in a large virtual memory.

Standard techniques unfortunately fail to provide an efficient mechanism for translating the Series II virtual addresses into main memory addresses.

Another technique is proposed for mapping very large addresses, which operates very efficiently for a relatively low cost.

The address translation units of two other computers are examined and are compared to the Series II unit.

1.0 Introduction

1.1 The MONADS Project.

The MONADS project began in 1976, with the intention of investigating methods for developing large software systems. An operating system ([1]-[6]) was implemented to execute on the MONADS Series I computer; a modified HP2100A minicomputer [2], [7]. The principles underlying the design of the operating system extend far beyond that system, and indeed can be applied to the development of any large software system.

During the development of the Series I system, it became evident that the available hardware was not entirely suitable for supporting the MONADS software structures. This prompted the building of a second computer, the MONADS Series II [8][9], which is partly based on a vastly modified HP2100A minicomputer.

1.2 MONADS Series II.

The MONADS Series II processor was designed to support an environment sympathetic to the philosophy of the MONADS project. It provides constructs for efficiently managing and supporting the key features of MONADS, some of which are not well 'understood' by other computers, including the Series I processor. This paper describes one of these areas, the Virtual Memory System.

Section 2 portrays a logical view of virtual memory systems whilst Section 3 comments on some standard implementations and their problems. Section 4 describes the Series II address translation unit and Section 5 compares it with two other virtual memory systems.

2.0 Logical View of Memory

2.1 Series II Virtual Memory.

It was demonstrated by the Multics designers [10] in 1964 that it was highly desirable to treat the memory of a processor in a homogeneous manner. From a user's viewpoint, there is no conceptual difference between a block of core (or semiconductor) memory and a block of disc memory; the only practical difference being the way in which the data is retrieved and the relative speed at which it is returned.

It was therefore decided that the Series II processor would provide the user (and system) with a very large virtual memory, and like multics, draw no distinction between fast and slow memory (or between files and arrays).

It was also deemed desirable that programs could be broken into their logical sections, or segments, so that these segments could be treated separately.

Thus the Virtual memory of the Series II is designed as a very large segmented memory (as this is now the only form of storage). Each segment is paged to simplify the enormous task of managing a segmented memory [15].

Address translation is the process of mapping an address as viewed from the processor onto the physical address required by the main memory system. If the address to be translated is not resident in main memory a page fault interrupt is caused, and the supervisor program fetches the page into mainstore.

The virtual address is typically divided into two sections, a virtual page number (possibly including a segment identifier) and an offset within page. The address translator maps the virtual page number onto a main memory page number, which is combined with the unaffected offset to form a main memory address. The model translation process is shown in Figure 1.

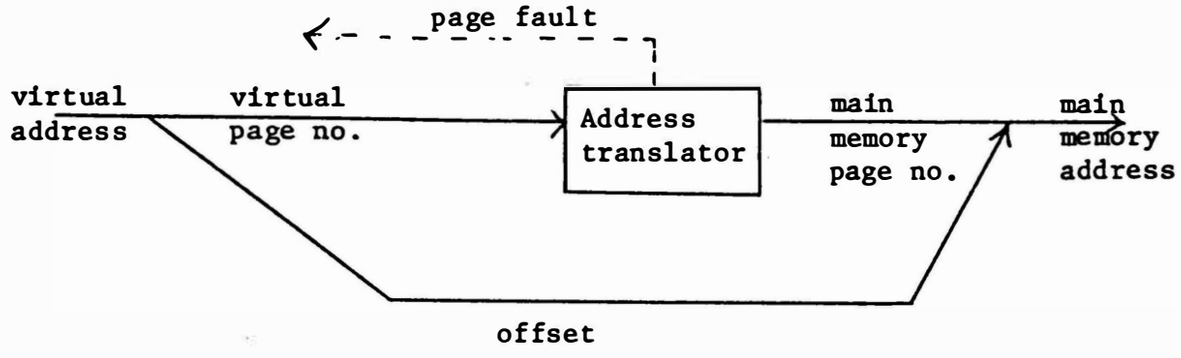


FIGURE 1

3.0 Virtual Memory Systems

3.1 Virtual Memory Categories.

For the purpose of implementing address translation hardware, virtual memory systems may be divided into three categories:

(1) Small Virtual memories
(2) Large Virtual memories with small main memories
(3) Large Virtual memories with large main memories.

3.1.1 Category (1) refers to systems where the address space viewed from a program is small enough to allow the address translation tables to be held directly in the hardware [7] [11]. Some such systems allow the operating system to swap these tables into and out of the hardware, so that individual processes may execute their own isolated address spaces.

These systems are relatively unaffected by changes in the size of the main memory, as this only alters the width of the translation tables. However, they are greatly affected by the size of the virtual address space, as this alters the length of the translation tables. A simple address translation unit is shown in Figure 2.

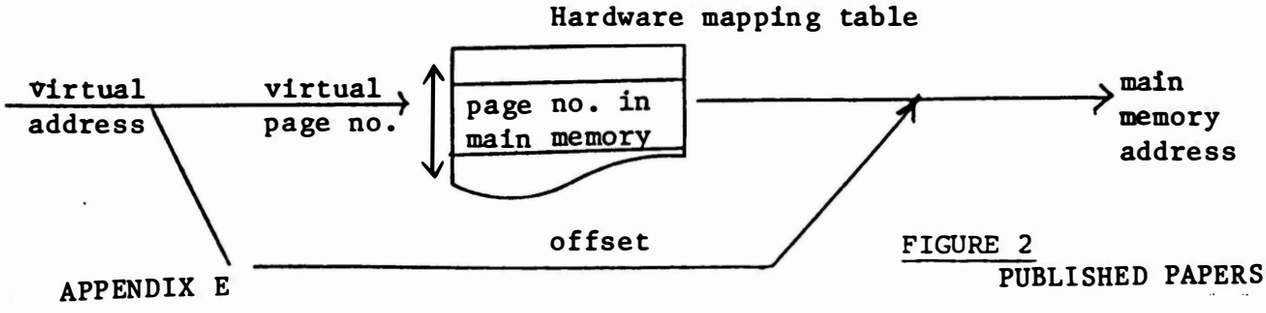


FIGURE 2

3.1.2 In systems with a large virtual memory and a small main memory (c.f. Atlas [12]), address translation is typically accomplished by utilising an associative memory to hold the translation tables. Each page of main memory is associated with one page address register, which holds the virtual address pertaining to that main memory page.

Translation is accomplished by the simultaneous comparison of the contents of each page address register with the page number part of the virtual address; the matching register then pointing to the page in main memory is the required one. Contrary to category (]), this technique is relatively unaffected by changes in the size of the virtual memory. However, it is greatly affected by the size of the main memory, as this alters the number of page address registers and comparators required. Such a scheme is described in Figure 3.

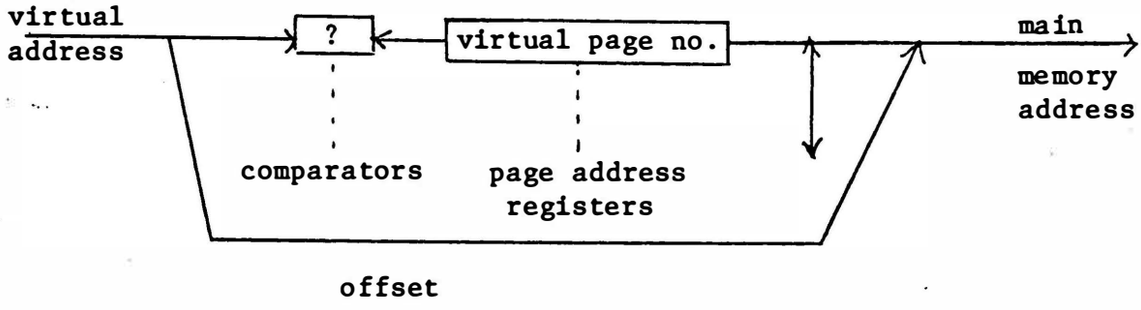


FIGURE 3

The use of an associative memory cannot, unfortunately, be extended to provide translation for category (3) systems due to the number of page address registers and comparators required.

3.1.3 The classical solution for systems with large virtual memories and large main memories (category 3) [10] [13] involves the use of page tables, (and segment tables) which are held either in main memory or virtual memory. These tables, which are maintained and searched by system software and firmware, offer a multi level indexed address translation table.

To achieve respectable translation times, this mechanism is usually augmented by a small associative memory, which holds the most recently used page table entries. This scheme is shown in Figure 4.

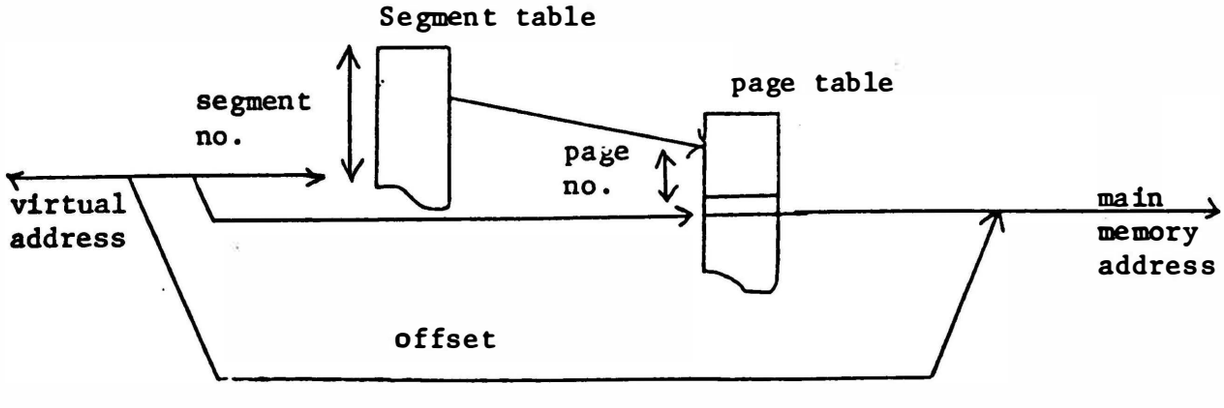


FIGURE 4

3.2 This classical solution would unfortunately fail to provide an efficient address translation mechanism for the Series II processor. The addressing style of Series II (which is capability based)[22] would introduce a very large number of small segments [14]. In addition, the page tables, which would be too large to reside in main memory, would be forced into virtual memory. This would greatly complicate the address translation software, and cuase the translation process to become extremely inefficient.

3.3 Logically, the solution adopted by systems such as Atlas (category (2) systems) would offer the best performance. However, the production of an associative memory capable of managing the many thousand pages of main memory, has not yet become feasible.

The next section describes an address translation unit which emulates a large associative memory very efficiently.

4.0 The Series II Virtual Memory System

4.1 The Virtual Address.

The virtual memory of the Series II processor is addressed by a 31 bit virtual address, which is unforgeable and unique across the system. From the viewpoint of the virtual memory hardware, this address <segment no (16 bits), page no, (6 bits) displacement (9 bits)> may be considered as 22 bits of virtual page identifier, and 9 bits of within page displacement, <virtual page no (22 bits), displacement (9 bits)>

Because the address is unique, the mapping hardware need never concern itself with the identity of the executing process and the system software need never swap mapping entries when switching processes.

4.2 The Physical Address.

The Series II physical address, for the main memory, consists of 13 bits of page number and 9 bits of within page displacement. This 22 bit address <page no (13 bits), displacement (4 bits)> allows a maximum main memory size of 8 MB.

4.3 Control of the Virtual Memory.

Control of the virtual memory may be divided into three sections, namely:

(1) Mapping Hardware
(2) Swap out software
(3) Swap in software.

Sections (2) and (3) are responsible for moving pages out of and into main memory respectively, and are managed by the Series II Hardware Kernel [16]. This software is fully described in a companion paper [17]. The remainder of this paper is concerned only with section (1).

4.4 Mapping Hardware.

The Mapping Hardware is solely responsible for mapping the 31 bit virtual address onto a 22 bit physical address. The external appearance is characterized in Figure 5.

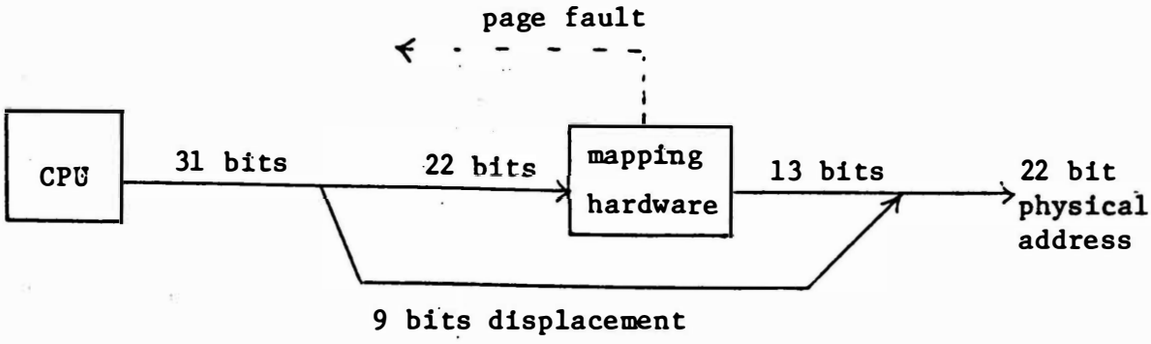


FIGURE 5

When presented with a virtual address, the mapping hardware may either produce a physical address, or a page fault signal. If the page required is in main memory, a physical address will be produced. However, if the page required is in secondary memory, a page fault interrupt will be signalled to the system, in order that remedial action may be taken.

4.5 Internal Operation.

The mapping hardware is organised as a high speed sparsely occupied hash table, with embedded overflow chains, as characterized in Figure 6.

The unit consists of three main components, a hashing unit, a hash table and a comparator.

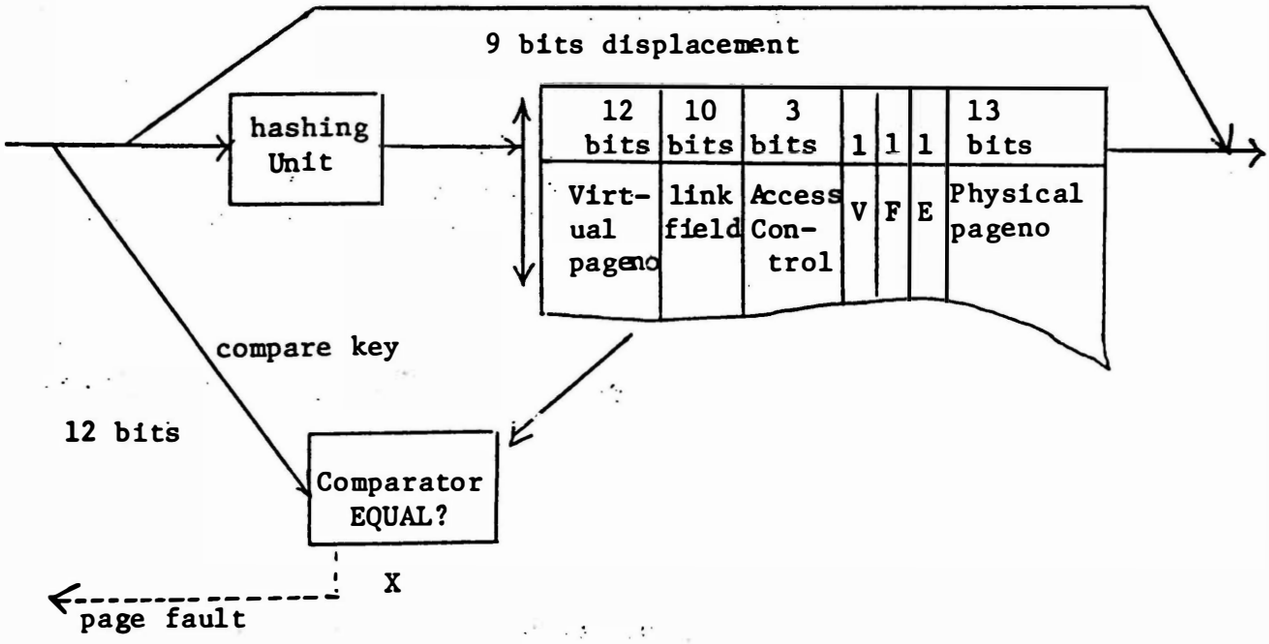


FIGURE 6

4.5.1 Hashing Unit.

The hashing unit accepts a 22 bit virtual page number and generates a 10 bits uniformly distributed index into the hash table.

The current version of the hashing unit uses low order bits from both the segment field and the page field of the virtual address (See 4.9.2.)

4.5.2 Hash Table.

The hash table is implemented using very high speed bipolar memory. Each cell is addressed by a 10 bit hash key, and contains seven fields:

(1)	virtual address identifier		12 bits
(2)	Physical page number		13 bits
(3)	Access field		3 bits
(4)	Valid field	(V)	1 bit
(5)	Link field		10 bits
(6)	Foreigner field	(F)	1 bit
(7)	end of chain field	(E)	1 bit

4.5.2.1 Virtual Address Identifier.

This field contains the remaining 12 bits of the virtual page number not used by the hashing function. All other information in the cell pertains to this virtual page.

4.5.2.2 Physical Page number.

The field holds the page number to which the virtual page is mapped.

4.5.2.3 Access Control Field

This field consists of three access control bits, controlling read, write and execute access respectively.

If a reference to a page contravenes any of the access control bits, an interrupt is generated and the reference is aborted.

4.5.2.4 Valid field.

This field specifies whether the virtual address identifier field contains a valid address. If an address hashes to an invalid cell, a page fault signal is generated.

4.5.2.5 Link field.

If more than one virtual address hashes to a given cell (i.e. clashes occur), an overflow chain is maintained by using another unused cell in the hash table. This field holds the address of the next cell in the overflow chain. It is maintained by the kernel software and is searched by the mapping hardware.

4.5.2.6 Foreigner field.

An address is foreign to a cell if its hash key value is not equal to the cell number in which it is held. If an address is foreign, the foreign bit is set. This bit is required because the Virtual address identifier field does not hold the entire virtual page number.

4.5.2.7 End of chain field.

This bit is set to signify that the cell is at the end of a link chain.

4.5.3 The comparator.

This unit tests the virtual address identifier field, and those bits of the virtual page number not used by the hashing unit, for equality. If equal, the physical page number field value is used as the translated page number.

4.6 Mapping Unit Operation.

Operation of the mapping unit is summarised in Figure 7. The entire translation process is managed by the mapping hardware, including the following of overflow chains. Control is returned to the software once the memory reference is completed (or after a page fault).

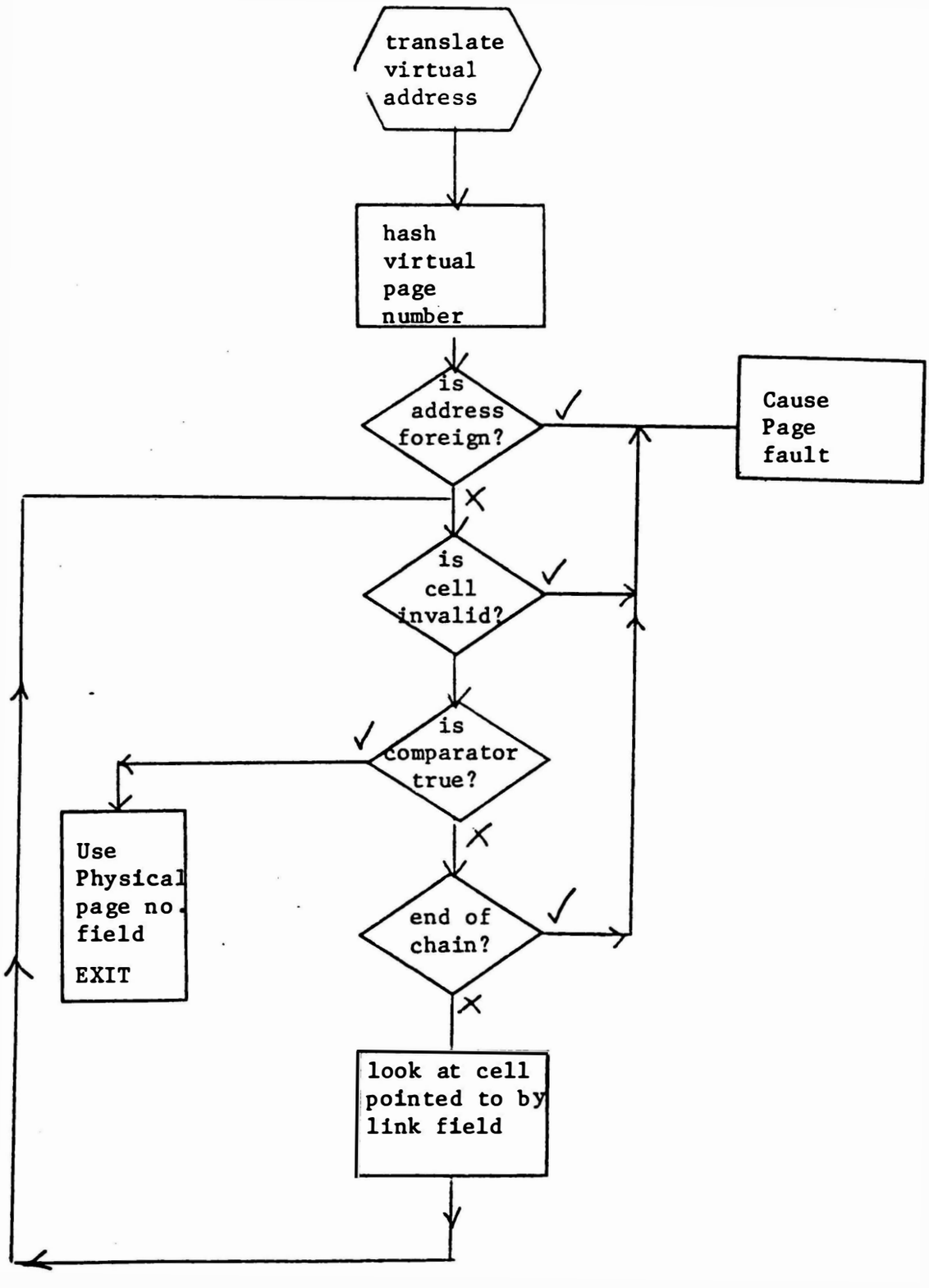


FIGURE 7

4.7 'Peek' Operation

For certain critical system software, it is important to know whether a page reference will cause a page fault. This may be achieved with the 'peek' operation.

When a peek operation is performed on a virtual address, the mapping unit attempts to translate the address. The software may then examine a Series II status register (SVR) to determine whether the reference would have caused a page fault. If the peek operation indicates that no page fault would have occured, then the reference is regarded as a normal (non peek) reference.

However, if the status register indicates a page fault condition, then the page must be fetched into main memory.

4.8 Control of the Mapping Hardware.

To the software responsible for initializing and maintaining the data in the mapping hardware, the hash table and associated registers appear in a special segment, the memory control segment (segment Ø). Values may be saved into or read from the various fields of the hash table by executing memory reference instructions on this segment.

4.8.1 Insertion and deletion.

Algorithms for inserting and deleting entries from the translation unit are fully defined in [23], and thus are not described here. Correct operation of the hardware demands that the software responsible for insertions and deletions conforms to these algorithms.

4.9 Performance of the Translation Unit.

4.9.1 Loading Factor.

A potential danger with using a hash table is that the number of collisions to any one cell (or clashes), and the average chain length. may become unacceptably high. Acceptable performance can, however, be obtained if the hash table is sparsely occupied (i.e. low loading factor). Providing that the hashing unit generates a uniform distribution of hash keys, the expected number of probes to retrieve an item in the hash table can be calculated from

$$E = 1 + \alpha/2$$

where α = loading factor [18], [23].

The current version of the Series II processor has a hash table size four times the number of pages in physical memory, so in this version $E = 1.125$ which is acceptably low.

(Note that for a true associative memory, $E = 1$)

4.9.2 Hashing Function.

The hash table performance is also affected by the efficiency of the hashing function, which should guarantee a uniform distribution of hash keys. The current version of the hashing unit uses a combination of low order bits from both the segment field and the page field of the virtual address. Should this function yield poor results, experiments may be made with more complex hashing functions.

Figure 11 shows the timing delays inherent in the Series II address translation unit. It can be seen that the minimum access time will be

$$t_{min} = \emptyset + 50 + 50 + (300\text{~}700)\ ns$$

$$= 400\text{~}800\ ns$$

On average

$$t_{av} = (400\text{~}800) + (E-1) \times 100$$

$$= 412\text{~}812\ ns$$

The variation in the main memory time is dependent on the cycle stealing of the refresh hardware for the dynamic memories used.

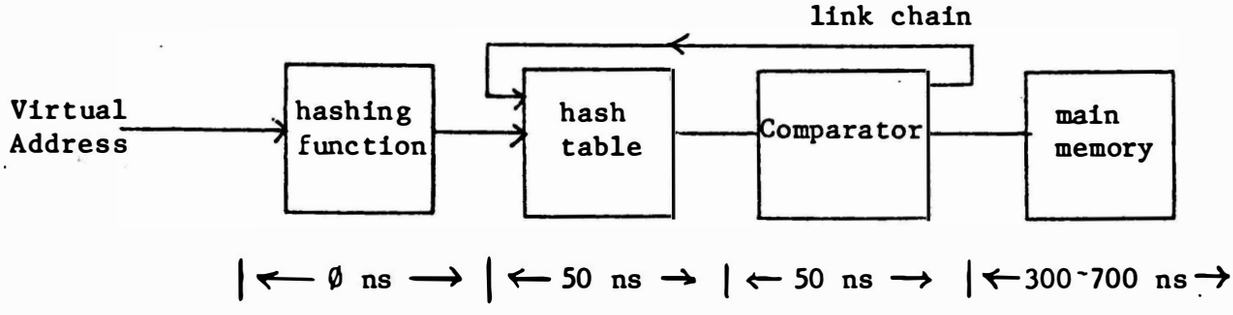


FIGURE 8

4.9.4 Expansion of main memory.

To maintain acceptable performance, the value E must be kept low, thus on addition of main memory the hash table size must be expanded proportionally. This increase in size does not necessarily affect any of the fields within the hash table. If the hash table is divided into blocks, links may be restricted to the 'block' of hash table in which they exist.

4.9.5 Optimization.

Performance of the hash table may be optimized by overlapping the comparison of the virtual page identifier in the current cell to the virtual page number, with the fetch of the cell linked to the current cell. This optimization, however, has little effect if the value E-1 is low.

5.0 Alternative Solutions

Two other computers have made attempts at solving the problem of translating very long virtual addresses, both using unconventional techniques.

5.1 MU6-G [19]

The MU6-G was recently developed at Manchester University as a "high performance machine useful for general and scientific applications". Among its many features, MU6-G includes a Memory Access Controller, or MAC, for translating virtual to physical addresses.

Memory Managment

The virtual address format in the MU6-G processor is as follows

```
< process no.   (8 bits)
  segment no.   (8 bits),
  block no.     (7 bits),
  bit no.       (14 bits)>
```

MU6-G associates one Page Address Register (PAR) with each physical page, as in Atlas [12]. However, rather than using a fully associative mechanism, which would be prohibitively expensive, a sequential search is made of these registers.

PARs are organized in banks of 256 locations, which gives an average translation time a little under 6μs. This unacceptably high time is reduced by the use of a translation look aside buffer. If the main memory cache on MU6-G is ignored, a mainstore access time of 750 ns is achieved.

5.2 The IBM System/38

The IBM System 38 [20] [21] is the most recent computer in the IBM range, and was developed by the General Systems Division in 1978.

Like the Series II, the System/38 translates an extremely long virtual address into a smaller mainstore address.

The translation process involves the use of a sparse hash table in the main memory of the System/38. Two tables are involved; the hash index table and the page directory. The page directory serves as an overflow table, unlike the Series II address translator which uses overflow chains embedded in the hash table. In addition, the hash index table, which is equivalent to the hash table in Series II, is held in the main memory of the system/38. This memory is far slower than the high speed bipolar RAM used in Series II.

Acceptable translation times are only achieved by the addition of a translation look aside buffer.

No timing values have been published for the System/38.

5.3 Conclusion.

Both MU6-G and System/38 require the use of high speed look aside buffers to achieve respectable translation times. The design of such buffers is usually similar to the design of the Series II mapping unit, i.e. as high speed hash tables, but with no overflow strategy, and usually of much smaller size. When the lookaside buffers in MU6-G and System/38 fail to translate an address, some form of slower overflow strategy is utilized. When a clash condition in the Series II mapping unit occurs, (which is less likely than a lookaside buffer failure) a high speed search is made within the hash table.

The paper has demonstrated that the address translation technique utilized by the MONADS Series II computer is practical, and offers very acceptable performance. It suggests that the Series II translation unit should offer equal or superior performance to the MU6-G or System/38 translation units.

Acknowledgments

The author wishes to acknowledge the following people, without whose help this work would never have been possible.

Professor Chris Wallace,
Dr. Les. Keedy,
Dr. John Rosenberg,
Mr. Brian Wallis.

References

[1] Keedy, J.L. (1978) "The MONADS Operating System", Proc. of 8th Australian Computer Conference.

[2] Wallace, C.S. (1978) "Memory and Addressing Extensions to a HP2100A", Proc. of the 8th Australian Computer Conference.

[3] Rosenberg, J. and Keedy, J.L. (1978) "The MONADS Hardware Kernel", Proc. of the 8th Australian Computer Conference.

[4] Ramamohanarao, K. and Keedy, J.L. (1978) "Job Management in the MONADS Operating System", Proc. of the 8th Australian Computer Conference.

[5] Richards, I. and Keedy, J.L. (1978) "Subsystem management in the MONADS Operating System", Proc. of the 8th Australian Computer Conference.

[6] Georgiades, A., Richards, I. and Keedy, J.L. (1978) "A File System for the MONADS Operating System", Proc. of the 8th Australian Computer Conference.

[7] Hagan, R.A. (1980), MSc. Thesis "Virtual Memory Hardware for a HP2100A Minicomputer".

[8] Abramson, D.A. (1980) "A Users Guide to the MONADS Extended Hardware" MONADS Internal Report No. 9.

[9] Abramson, D.A. (1980) "A Users Guide to the MONADS Memory Management Hardware", MONADS Internal Report No. 10.

[10] Organick, E.I.(1972) "The MULTICS System: An Examination of its Structure", MIT Press, Cambridge, MAS. & London.

[11] Hewlett Packard, "The HP21MX Reference Manual"

[12] Kilburn, T., Edwards, D.B.E., Lanigan, M.J. and Sumner, F.H. (1962) "One Level Storage System", I.R.E. Trans. Electronic Computation, EC-11 No. 2, pp 223-234.

[13] Prime, "The System Architecture Reference Guide", PDR 3060, Section 2.

[14] Keedy, J.L. (1980) "Paging and Small Segments: A Memory Management Model", Proc. of IFIP World Congress 1980.

[15] Randell, B. (1969), "A Note on Storage Fragmentation and Program Segmentation", Comms. of A.C.M., July 1969, Vol. 12, No. 7, pp 365-372.

References (contd.)

[16] Wallis, B. (1980) "A Hardware Kernel of the MONADS Series II computer", Honours Report - Dept. of Computer Science, Monash University.

[17] Rosenberg, J. and Keedy, J.L. (1981) "Software Management of a Large Virtual Memory", Proc. of ACSC 4, Brisbane 1981.

[18] Morris, R. (1968) "Scatter Storage Techniques", Comms. of A.C.M., Jan. 1968, pp 38-43.

[19] Edwards, D.B.G., Knowles, A.E. and Woods, J.V. (1980) "The MU6-G. A New Design to Achieve Mainframe Performance from a Mini Sized Computer", Proc. of the 7th Annual Symposium on Computer Architecture, 1980 pp 161 - 167.

[20] Houdek, M.E. and Mitchell, G.R. "Translating a Large Virtual Address" IBM System/38 Technical Developments (1978) pp 19-21.

[21] Hoffman, R.L. and Soltis, F.G. "Hardware Organization of the System/38" IBM System/38 Technical Developments (1978) pp 19-21.

[22] Fabry, R.S. (1974) "Capability-based Addressing", Comms. of A.C.M., July 1974, Vol. 17, No. 7, pp 403-411.

[23] Knuth, D.E., "The Art of Computer Programming", Vol. 3, Section 6.4, pp 506 - 549.

[Editor's note: This paper and its companion, "Software Management of a Large Virtual Memory", by J. Rosenberg and J.L. Keedy, together fall within the page limit for two contributed papers.]

BIBLIOGRAPHY

Abramson, D.A. (1980) "A Users Guide to the MONADS Extended Hardware" MONADS Internal Report No 9. Monash University.

Abramson, D.A. (1981) "Hardware Management of a Large Virtual Memory", Proc. 4th Australian Computer Science Conference, Brisbane (Australian Computer Science Communications 3, 1, pp. 1-13).

Abramson, D.A. (1982a) "A Technique for Enhancing Processor Architecture", Proc. 5th Australian Computer Science Conference, Perth (Australian Computer Science Communications 4, 1, pp. 47-57).

Abramson, D.A. (1982b) "Hardware for Capability Based Addressing", Proc. 9th Australian Computer Conference, Hobart.

Abramson, D.A. and Rosenberg J. (1982) "Hardware Support for Program Debuggers", (in preparation).

Batson, A.P. and Brundage, R.E. (1977) "Segment Sizes and Lifetimes in Algol 60 Programs" Comm. ACM. Vol 20 Num 1, pp. 36-44.

Belady, L.A., Parmelee, R.P. and Scalzi, C.A. (1981) "The IBM History of Memory Management Technology", IBM Journal of Research and Development, Vol 25, Num 5, September 1981.

Belgard, R. (1976) "A Generalized Virtual Memory Package for B1700 Interpreter Writers" SIGMICRO Dec 1976.

Bird, A. (1982) Honours Report, Department of Computer Science, Monash University. (in preparation).

Bishop, P. (1977) "Computer Systems with a Very Large Address Space and Garbage Collection" PhD Thesis, MIT (MIT TCS TR-178).

Cohen, S.J. (1973) "A Virtual Memory Facility for Emulation" SIGMICRO Jan 1973.

D'Hautcourt-Carrette, F. (1977) "A Micro-programmed Virtual Memory for

the Eclipse" SIGMICRO June 1977.

Dahl, O.J.,Myhrhaug, B. and Nygaard, K. (1968) "The Simula 67 Common Base Language", Norwegian Computer Centre, Oslo.

Data General (1974) "Programmers Reference Manual, Eclipse Computer", Number 015-000024-02, Data General Corporation, 1974.

Dawson, P. (1982) " A Users Guide to the MONADS II Debugger", Department of Computer Science, Monash University.

Denning, P.J. (1968) "The Working Set Model for Program Behaviour" Comm. ACM, Vol 11, Num 5, pp. 323-333.

Denning, P.J. (1970) "Virtual memory" Computing Surveys, Vol 2, Num 3, pp. 153-189.

Denning, P.J. (1980) "Working Sets: Past and Present" IEEE Trans. of software engineering. Vol SE-6, Num 1, pp. 64-84.

Dennis, J.B. and VanHorn, E.C. (1966) "Programming Semantics for Multiprogrammed Computations" Comm. ACM, Vol 9, Num 3, pp. 143-155.

Digital Equipment Corp. (1979) "VAX 11 Architecture Handbook", Digital Equipment Corporation.

Edwards, D.B.G., Knowles, A.E. and Woods, J.V. (1980) "MU6-G: A New Design to Achieve Mainframe Performance from a Mini Sized Computer" Proc. of 7'th Annual symposium on computer architecture, pp. 161-167.

England, D.M. (1972) "Architectural Features of System 250" Infotech state of the art report 14, Operating systems, pp. 395-426.

Evans, D.C. and Leclerc, J.Y. (1967) "Address Mapping and Control of Access in an Interacive Computer" Proc. of 1967 Spring Joint Computer Conference, pp. 23-30.

Fabry, R.S. (1974) "Capability Based Addressing" Comm. ACM, Vol 17, Num 7, pp. 403-412.

Feustal, E.A. (1972) "The Rice Research Computer - a Tagged Architecture" AFIPS conference, Vol 40, pp. 369-377.

Fotheringham, J. (1961) "Dynamic Storage Allocation in the Atlas Computer including an automatic use of backing store" Comm. ACM, Vol 4, Num 10, pp. 435-436.

Gehringer, E.F. (1979) "Functionality and Performance in Capability-based Operating Systems", Ph.D. Thesis, Purdue University, May 1979.

Gehringer, E.F. and Chansler, R.J. (1981) "STAROS User and System Structure Manual", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Gehringer, E.F. and Keedy, J.L. (1982) "Tagged Architecture: How Compelling are its Advantages?", (submitted for publication).

Gligor, V. (1978) "Architectural Implications of Abstract Data Type Implementations" Department of Computer Science, University of Maryland, TR-659.

Hagan, R. (1977) "Virtual Memory Hardware for a HP2100A Minicomputer", M.Sc. Thesis, Monash University.

Hagan, R.A. and Wallace, C.S. (1979) "A Virtual Memory System for the Hewlett Packard 2100A", ACM Computer Architecture News, 6, 5, pp. 5-13.

Houdek, M.E., Soltis, F.G. and Hoffman, R.L.(1981) "IBM System Support for Capability Based Addressing" 8'th SIGARCH symposium on computer architecture, pp. 341-348.

Hewlett Packard (1972) "A Pocket Guide for the HP2100 Mini-computer",

Hewlett Packard Co., California, U.S.A.

Hewlett Packard (1974) Hewlett Packard Journal, October, 1974.

IBM (1978) "IBM System/38 Technical Developments" General Systems Division.

IBM (1980) "IBM System/38 Functional Concepts Manual" File S38-01.

ICL (1971) "J-Level Disc Operating System, Technical Publication 4558, International Computers Ltd, Putney UK, 1971.

ICL (1976) "Central Processors: the ICL 1900 Series" ICL Technical Publication, 4412, May 1976.

Intel (1981a) "Object Based Computer Architecture" Computer Architecture News, ACM, 1981.

Intel (1981b) "Introduction to the iAPX432 Architecture", Intel Corp. Manual Order No. 171821-001.

Keedy J.L. (1977) "An Outline of the ICL2900 Series System Architecture", Australian Computer Journal, 9, 2, pp. 53-62.

Keedy, J.L. (1978) "The MONADS Operating System", Proc. 8th Australian Computer Conference, Canberra, pp. 903-910.

Keedy, J.L. (1980) "Paging and Small Segments: A Memory Management Model", Proc.8th World Computer Congress, IFIP-80, Melbourne, pp. 337-342.

Keedy, J.L. (1981) "A Progress Report on the MONADS Project" Australian Computer Science Communications, 3, 2, pp. 270-277.

Keedy, J.L. (1982a) "The MONADS View of Software Modules", Proc. 9th Australian Computer Conference, Hobart.

BIBLIOGRAPHY

Keedy, J.L. (1982b) "Support for Information-Hiding Modules in the MONADS Architecture" (submitted for publication).

Keedy, (1982c) "Module Capability Management in the MONADS Systems", (submitted for publication).

Keedy, J.L., Abramson, D., Rosenberg, J. and Rowe, D.M. (1982) "The MONADS Project Stage 2: Hardware Designed to Support Software Engineering Techniques", Proc. 9th Australian Computer Conference, Hobart.

Keedy, J.L. and Ramamohanarao, K. (1979) "A Job Management Model for In-Process Systems", MONADS Report No. 7, Monash University.

Keedy, J.L. and Richards, I. (1982) "A Software Engineering View of Files", Australian Computer Journal, 14, 2.

Keedy, J.L. and Rosenberg, J. (1981) "Information Hiding - A Key to Successful Software Engineering", Proc. Conference on Computers in Engineering, 1981, Institution of Engineers, Australia, Publication No. 81/8, pp. 1-5.

Keedy, J.L. and Rosenberg, J. (1982a) "The MONADS III Computer Design: A System to Support Software Engineering" (submitted for publication).

Keedy, J.L. and Rosenberg, J. (1982b) "Architectural Support for Software in the MONADS III Computer Design", Proc. 12th Annual Conference Gesellschaft fuer Informatik, Kaiserslautern, 1982.

Keedy, J.L., Rosenberg, J., Abramson D. and Rowe, D.M. (1982) "A Comparison of the MONADS II and III Computer Systems", Proc. 9th Australian Computer Conference, Hobart.

Knuth, D.E. (1978) "The Art of Computer Programming: Fundamental Algorithms", Addison Wesley Publishing Company, Second Edition, 1978, pp. 435-456.

BIBLIOGRAPHY

Kohonen, T. (1978) "Associative Memory - a System Theoretical Approach" Springer-Verlag, Berlin, Heidelberg, New York.

Kohonen, T. (1980) "Content Addressable Memories" Springer-Verlag, Berlin, Heidelberg, New York.

Kilburn T., Edwards D.B.E., Lanigan M.J. and Sumner F.H. (1962) "One Level Storage System", I.R.E Trans. Electronic Computation, EC-11, No 2, pp. 223-234.

Lampson, B. and Sturgis, H. (1976) "Reflections on an Operating System Design" Comm. ACM, Vol 19, No 5, pp. 251-265.

Lanciaux D., Schiller L. and Wulf W. (1976) "Supporting Small Objects in a Capability System" Carnegie-Mellon University ,Internal report , Dec 1977.

Liskov, B. and Zilles, S. (1974) "Programming with Abstract Data Types", Proc. A.C.M. Sigplan Conf On Very High Level Languages, A.C.M., Sigplan Notices, Vol 9, Num 4, pp. 50-59.

Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. (1977) "Abstraction Mechanisms in CLU", Comm. ACM, 20, 8,pp. 564-576.

Morris, R. (1968) "Scatter Storage Techniques", Comm. ACM, Jan, 1968, pp. 38-43.

Morris, J.B. (1972) "Demand Paging through Utilization of Working Sets in the MANIAC II", Comm. ACM, Vol 6, Num 1, pp. 1-17.

Myers, G.J. (1978a) "Advances in Computer Architecture", New York: Wiley-Interscience, 1978.

Myers, G.J. (1978b) "Storage Concepts in a Software Reliability Directed Computer Archhitecture" Proc. Fifth Annual Symp. on Computer Architecture, New York, ACM, pp. 107-113.

BIBLIOGRAPHY

Myers, G.J. and Buckingham, B.R.S. (1980) "A Hardware Implementation of Capability Based Addressing" Computer Architecture News, October, 1980, pp. 12-24.

Needham, R.M. (1977) "The CAP Project - an Interim Evaluation" Proc. of 6'th ACM symposium on Operating System principles, pp. 17-22.

Organick, E.I. (1972) "The MULTICS System: An Examination of its Structure", MIT Press, Cambridge, Mass.

Organick, E.I. (1973) "Computer Systems Organization, the B5700/6700 Series", Academic Press, New York.

Parnas, D.L. (1971) "Information Distribution Aspects of Design Methodology", Proc. 5th World Computer Congress, IFIP-71, pp. 339-344.

Parnas, D.L. (1972) "On the Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, 15, 12, pp. 1053-1058.

Patterson, G. (1981) "MONADS Command Language Interpreter", Department of Computer Science, Monash University, Honours Report.

Prime. (1979) "The System Architecture Reference Guide" PDR 3060.

Ramamohanarao, K. (1980) "A New Model for Job Management Systems", Ph.D. Thesis, Monash University.

Ramamohanarao, K. and Keedy, J.L. (1978) "Job Management in the MONADS Operating System", Proc. 8th Australian Computer Conference, Canberra, pp. 1476-1488.

Ramamohanarao, K. and Sacks-Davis, R. (1981) "Hardware Address Translation for Machines with a Large Virtual Memory", Information Processing Letters, 13, 1, pp. 23-29.

Randell, B. (1969) "A Note on Storage Fragmentation and Program

Segmentation" Comm. ACM, Vol 12, Num 7, pp. 365-372.

Rees, S. (1981) "The MONADS Series II Assembler Manual", Department of Computer Science, MONADS II Technical Report I, Monash University.

Richards, I. (1982) "The Organisation and Protection of Information in a Computer Utility", Ph.D. Thesis, Monash University, 1982.

Richards, I. and Keedy, J.L. (1978) "Subsystem Management in the MONADS Operating System", Proc. 8th Australian Computer Conference, Canberra, pp. 1520-1529.

Rosenberg, J. (1979) "The Concept of a Hardware Kernel and its Implementation on a Minicomputer", Ph.D. Thesis, Dept. of Computer Science, Monash University.

Rosenberg, J. and Keedy, J.L. (1978) "The MONADS Hardware Kernel", Proc. 8th Australian Computer Conference, Canberra, pp. 1542-1552.

Rosenberg, J. and Keedy, J.L. (1981a) "Software Management of a Large Virtual Memory", Proc. 4th Australian Computer Science Conference, Brisbane, pp. 173-181.

Rosenberg, J. and Keedy, J.L. (1981b) "Information Hiding - A Case Study", Proc. Conference on Computers in Engineering, 1981, Institution of Engineers, Australia, Publication No. 81/8, pp. 6-9.

Rosenberg, J., Rowe, D.M. and Keedy, J.L. (1982) "An Overview of the MONADS Series III Architecture", Proc. 5th Australian Computer Science Conference, Perth, pp. 58-67.

Shepherd, J.H., (1968) "The principle Design Features of the Multi-computer Chicago Magic Number Computer" ICR quarterly report 19, Nov 1968, University of Chicago.

Sitton, W.G. and Wear, L.L. (1974) - "A Virtual Memory System for the Hewlett-Packard HP2100A", ACM 7th Annual workshop on

Microprogramming, pp. 119 - 121.

Strecker, W.D. (1978) "Cache Memories for the PDP-11 Family Computers" Chapter 10, "The PDP-11 family" "Computer engineering - a DEC view of hardware system design" Digital Press.

Tanenbaum, A. (1979) "A Method for Implementing Paged Segmented Virtual Memories in Microprogrammed Computers" SIGOPS, ACM Vol 13, Num 2, pp. 26-32.

Wallace, C.S. (1978) "Memory and Addressing Extensions to a HP2100A" Proc. 8th Australian Computer Conference, pp. 1796-1811, Canberra.

Wallis, B.R. (1980) "A Hardware Kernel for the MONADS II Computer", Honours Thesis, Department of Computer Science, Monash University.

Wilkes, M.V. and Needham, R.M. (1979) "The Cambridge CAP Computer and its Operating System", North Holland, Oxford.

Wilkes, M.W. (1980) "A New Hardware Capability Architecture" Operating System Reviews, April 1980, pp. 17-20.

Wirth, N. (1977) "Modula - A Language for Modular Programming", Software-Practice and Experience, 7, 1, pp. 3.

Wulf, W.A. et. al. (1974) "HYDRA: The Kernel of a Multiprocessor Operating System", Comm. ACM, 17, 3, pp. 336-345.

Wulf, W.A., London, R. and Shaw, M. (1976) "An Introduction to the Construction and Verification of Alphard Programs" IEEE Transactions on Software Engineering, SE-2, 4, pp. 253-264.

Wulf, W.A., Levin, R. and Harbison, S.P. (1981) "HYDRA/C.mmp: An Experimental Computer System", McGraw-Hill, New York.

Yngve, V.H. (1968) "The Chicago Magic Number Computer" ICR quarterly report 19, Nov 1968, University of Chicago.

BIBLIOGRAPHY