

MONADS-PC - A Capability-Based Workstation to Support
Software Engineering

John Rosenberg & David Abramson
Department of Computer Science
Monash University
Clayton 3168
Australia

Abstract

The MONADS project was established in 1976 at Monash University, with the main aim of investigating improved techniques for designing and developing large software systems. This paper gives a brief history of the project and then discusses the aims and objectives of the MONADS system. This is followed by a description of the latest phase of the project, the development of a small microprogrammed workstation (called MONADS-PC) specifically designed to support the MONADS architecture.

1 INTRODUCTION

The MONADS project was established in 1976 at Monash University in Melbourne, Australia, with the main aim of investigating improved techniques for designing and developing large software systems. The first phase of the project was the design and implementation of an operating system for a Hewlett Packard 2100A minicomputer, which had been locally modified to support a paged virtual memory and extra addressing modes [1, 2, 3, 4, 5, 6, 7]. This system became known as MONADS I.

From our experience with the project it became obvious that conventional computer hardware was not appropriate for our ideas on software structures. This led us to the MONADS II project, in which we undertook a far more extensive set of modifications to a second HP2100 computer [8, 9].

Although this hardware did allow us to test and demonstrate our ideas in practice, it suffered severely from limitations of the underlying machine especially in the area of addressing which is critical to the MONADS architecture.

We were therefore convinced that the only sensible approach was to build a completely new machine to efficiently implement our ideas in hardware. The design of this machine, to be known as MONADS III, began in 1981. MONADS III was to be comparable in power to a VAX 11/780 and to support a large complement of memory and disk. Although a substantial portion of the design of the MONADS III system was completed [10, 11, 12, 13], the project had to be shelved following the resignation of two of the three chief investigators.

In 1984 the MONADS project was revived and we were again determined to build a MONADS machine. However, changes in technology, especially in the memory and disk areas, now make development of a number of small workstations a more attractive proposition than returning to the MONADS III approach.

Before describing the proposed workstation, known as MONADS-PC, it is instructive to describe the major aims and objectives of the MONADS architecture.

2 AIMS AND OBJECTIVES

The MONADS project has three major design philosophies and these affect the hardware design in several different ways. In this section each of these philosophies is described and the resulting hardware requirements discussed.

2.1 Module Structure

It has long been recognized that one of the keys to successful software engineering is to limit the scope of access to data [14]. Parnas [15] and others [16] have suggested that this can be achieved by decomposing software systems into information-hiding modules. Each software module, according to the information-hiding principle, hides major design and implementation decisions from other modules, so that changes to a specific module have minimal impact on the rest of the system. This implies that alterations to data structures, to algorithms and even to lower-level abstract or real machines can be hidden from the users of a module (unless these involve changes to the module's specification). A good way of achieving this is to insist that all inter-module interfaces are purely procedural and that the private database of a module be accessible only to the procedures of that module.

The MONADS system rigorously enforces the information-hiding principle by not supporting free-standing data-structures (including files [17]) and by forbidding the exportation of

variables from modules [18]. Thus all conventional program units and files are implemented as information hiding modules and all data is accessed via a procedural interface [19]. The implementation of this is described later in this paper.

2.2 Process Structure

Lauer and Needham [20] have proposed two basic models for the decomposition of operating systems into processes. In the first, known as the procedure-oriented or 'in-process' model, operating system functions are requested by procedure calls within the user's process. Thus the only processes in existence on the machine (apart from system management processes such as spoolers) are the user processes. In the second model, known as the process oriented or 'out-of-process' model, each system function or related group of functions is associated with a system process. Requests for system functions in this model are made by passing messages to the appropriate process.

Lauer and Needham have shown that statically the two models can be made to look equivalent and that the static structure of the operating system need not be affected by the choice. However, once the dynamic properties of the two models are considered this duality argument fails. Keedy [21] has argued that the in-process model is superior in that it potentially allows a greater degree of parallelism and has several administrative advantages. We do not have the space to discuss these advantages here, however the interested reader is referred to [21].

One potential disadvantage of the in-process model is that it requires a higher degree of hardware support. In particular, procedure calls and returns involve a change of domain. For example, when an operating system function is called the operating system code and data must be made addressable on the call and non-addressable on the return. Many systems make this a special case and provide a supervisor-mode which is

enabled on a system call and makes certain areas of memory addressable. However, such a facility will not handle the common situation of a user requesting the operating system to run a program. In this case the operating system is calling the user program and the operating system data should become non-addressable on the call rather than the return. The usual solution to this problem is to call the user program by a simulated 'return' from the operating system and the user program then calls the operating system when it has finished.

The MONADS architecture supports the in-process model by providing a procedure calling mechanism which performs a domain switch on the call and reinstates the caller's domain on the return. The implementation of this mechanism is briefly described later in the paper.

2.3 Uniform Virtual Memory

Most modern computers support a virtual memory. In a virtual memory system the addressing range available to programs is larger than the size of physical memory on the machine and program addresses are mapped onto physical addresses by the hardware. Sections of code or data which are not currently in physical memory are held in a special area on disk. Thus the apparent memory space of the machine is extended by using some of the disk.

In the MONADS architecture the virtual memory structure is extended to the limit. All of the disk is considered to be part of the virtual memory of the machine. Thus there is no conventional filestore and files simply reside in the virtual memory like computational data. This approach, which has been adopted on at least one commercial machine [22], has the major advantage that it avoids duplication of mechanisms. Two obvious examples of such duplication are the dual protection mechanisms which have to be supported (one to protect running programs from each other and the other to guarantee privacy of files) and the dual synchronisation mechanisms (for example semaphores in the computational memory and record

locks in the filestore). In both cases the same fundamental problem is being solved and in both cases the filestore mechanism is clumsier and less efficient because of the lack of hardware support.

There are other areas of duplication (e.g. data accessing techniques) and these have been fully described in [23]. For all of these reasons the MONADS architecture supports a very large uniform virtual memory. A consequence of this is that addresses must also be quite large (large enough to address all of the disks that may be connected to the machine).

3 THE MONADS ARCHITECTURE

The main features of the MONADS architecture are the large addresses required to access the virtual memory, capability-based protection and the procedure calling mechanism. In this section we will briefly describe how these are implemented.

The idea of capability-based addressing was first proposed by Dennis and Van Horn [24] and can be summarised as follows. Every object in the system is given a unique, non-forgable address called a capability. These addresses are never re-used, even when the object has been deleted. Capabilities themselves are protected and cannot be modified or directly generated by programs, rather they are created and transferred under system control. A particular object can only be accessed if a procedure has a capability for the object. Thus a fine grain of protection and control over access to data can be achieved. The MONADS architecture exploits this property of capabilities in order to implement information-hiding modules.

MONADS virtual addresses are 60 bits in size. The address is divided into two parts, an address space number and an offset. This is illustrated in figure 1. An address space is used to hold a related group of segments (e.g. the code of a module, the data of a module or a stack). When an address space is created it is allocated a unique number and this number is never re-used, even

after the address space has been deleted. Address spaces are paged, with a page size of 4096 bytes. The 28 bit offset portion of a virtual address identifies a byte within the address space.

Each address space is divided into a number of segments. Segments are not related to pages or page boundaries. Programs never directly use virtual addresses, rather they address segments via segment lists. A segment list contains a number of capabilities for segments in an address space. The format of each segment list entry is shown in figure 2. Thus a segment list entry is effectively a capability for a window on the address space. The entire structure is illustrated in figure 3.

Each process in MONADS has a stack address space. This is used to hold linkage information on procedure calls, local data for procedures and for temporary storage during expression evaluation. The data on the stack is also held in segments and is addressed via segment lists.

The addressing environment of a process on MONADS is defined by the segment list entries of the segment lists which it can use. The valid segment lists for a process are pointed to by a table held at the base of the stack; in an area of memory not directly addressable by programs. The entries in this table are called bases, and point at segment lists for the following types of data: constant data segments in the code address space, permanent data segments associated with the current module, local data segments held on stack and parameter segments. In fact there are several other segment lists, including multiple local and parameter segment lists to support block structure, which are not relevant to this discussion [25]. Access to a byte in the virtual memory is achieved by specifying a base, segment list entry and an offset within the segment. Naturally, the bases and segment list entries are only modified by special system management instructions.

The MONADS architecture supports up to 64 bases and 1024 segments per segment list. Thus 16

bits are needed to identify uniquely a segment list entry. In order to reduce the size of addresses in instructions, and for efficiency reasons, a set of 16 capability registers are provided. These have the same format as segment list entries and may be loaded from a segment list by a system management instruction. Thus machine instructions can address data by specifying a capability register and an offset.

From the above, it can be seen that the addressing environment at any time is defined by the segment lists pointed to by the bases. This provides very flexible control over access to data. On a procedure call the call mechanism modifies the bases to create the appropriate environment for the called procedure. This environment is derived from the 'red-tape' information held in non-addressable segments in the address space(s) of the called module. Information from this red-tape is also used to create local data segments on the stack for the called procedure. The call mechanism also performs several other useful tasks and these are fully described in [25].

4 MONADS-PC

4.1 Overview

Figure 4 shows a block diagram of the MONADS-PC system. The major components are the microprogrammed central processor, the cache, the address translation unit and the multibus memory and I/O subsystem.

An early decision in the MONADS-PC design was to build our own microprogrammable processor rather than utilizing one of the available microprocessors (e.g. M68000 or National 32032). The main reason for this was the addressing and stack structure of the MONADS architecture. All of these microprocessors generate (relatively) small addresses and naturally do not provide the required capability registers. One possibility was to provide these registers externally to the

microprocessor and to interpret the address generated by the microprocessor as a register number and offset. This is basically the approach that was adopted on MONADS II and from our experience we felt that the amount of hardware required to support this and allow the registers to be loaded, etc. would be similar to that required to build a simple microcoded machine. Also such a structure would prohibit effective use of many of the addressing modes of the microprocessor (e.g. stack addressing) as they would not produce sensible results.

A second reason for rejecting this approach was that none of the available microprocessors provide the procedure calling mechanism required by the MONADS architecture. The procedure call would thus have had to be a system software routine with an obvious loss in efficiency. By building a microprogrammable machine we are able to microcode exactly the required call and return instructions.

A second design decision was that an industry-standard memory and I/O bus would be used. This was so that off-the-shelf memory and input-output controllers could be purchased. The Multibus I [26] was chosen, despite its rather slow speed and lack of support for 32 bit data transfers. This choice was made mainly because of the availability of a large range of disk and serial line controllers. The initial MONADS-PC will have 2 megabytes of dynamic memory, an ST-506 compatible disk controller (capable of supporting up to 4 winchester disks) and an eight port intelligent serial line controller.

4.2 Instruction Set

The main purpose of building the MONADS-PC is to provide a machine that will support good software engineering techniques. We therefore did not want to invest a large amount of effort into designing a complex instruction set, and resolved to keep the machine as simple as possible. In this way we could concentrate on the software engineering aspects of the project.

MONADS-PC is 32 bit machine. It supports only three basic data types, 32 bit integers, 32 bit floating-point numbers and 8 bit characters. Integers and reals must be word aligned, however, this is not considered to be a serious restriction. Most machines which do allow non-aligned data suffer from a severe degradation in performance when such data is accessed and thus the compilers force all words to be aligned.

MONADS-PC is a single accumulator machine. The accumulator is an implicit operand on most arithmetic and logical operations. The accumulator (A) is a 32 bit register. There is also an accumulator extend register (AX) which can be used for double length arithmetic operations. There are three index registers (I1-I3) and sixteen capability registers (C0-C15) of the format described earlier. In addition there are two dedicated capability/index register combinations, one for accessing the computational area of the stack and one for fetching words of code from the instruction stream. Finally there is processor status word (PSW) which contains condition codes indicating the result of the last instruction.

MONADS-PC has only three basic addressing modes, data mode, literal mode and top-of-stack mode. In data mode a capability register, offset and index register are specified. The capability register effectively points to a window on the virtual address space, the offset is an offset from the start of this window and the index register (scaled by four for instructions with word operands) is added to this offset. Thus data mode can be used to access any byte in the addressing environment of the current process. Literal mode is used to generate a constant literal. Top-of-stack mode has different effects depending on whether it is used as a read operand or a write operand. On a read the data is obtained from the top of the stack and the stack is popped, on a write the data is pushed onto the stack. For certain instructions (e.g. index register operations) the accumulator and index registers may be specified as operands.

All instructions in MONADS-PC occupy an integral number of words, most being one word long. MONADS-PC has the usual complement of arithmetic and logical operations as well as an extensive set of bit manipulation instructions [27]. These are used to manipulate sets which are used extensively by the operating system modules. Instructions are provided to allow the accumulator to be viewed as an extension to the top-of-stack, in particular there is a 'push and reload accumulator' instruction. This assists in efficient evaluation of arithmetic expressions [28]. Limited arithmetic operations may be performed on the index registers to avoid unnecessary saving and restoring of the accumulator during array index operations.

At this stage it is envisaged that the only byte instructions that will be provided will be load, store and compare. It will be possible later to provide support for variable length strings. There is a general multiple word/byte move instruction. Control instructions include the usual branches on conditions, a simple jump and link instruction and the procedure call and return instructions. In addition there is a large set of system management instructions to manipulate bases, segment lists and the capability registers in a controlled manner.

4.3 The Micromachine

MONADS-PC is based on a 32 bit microprogrammable processor [29]. When we embarked on the design it was hoped that we would be able to make considerable use of the AMD 2900 series bit-slice components [30] both for the ALU and the control logic. Unfortunately this turned out not to be practical. The 2901/3 ALU chip has only 16 internal registers. The MONADS architecture requires sixty-four 32-bit registers simply to hold the capability registers. Although extra registers can be added externally to the 2901 this becomes quite cumbersome. The 2910 sequencer was rejected because of the limited control store size it will support (4096 words). MONADS-PC makes

extensive use of microcode for system management functions and thus we wanted the flexibility to add more control memory at a later stage. The 2909/2911 series sequencers were rejected mainly because they were too slow. It is very difficult to achieve reasonable speed without employing several levels of pipe-lining which makes coding more complex, or using very fast control store (less than 70 nanosecond access time) which is not available in high density memory components.

For these reasons the MONADS-PC micro-machine is built using standard MSI and LSI chips. It has a highly horizontal structure with a microword of approximately 100 bits. The basic cycle time is 140 nanoseconds, although for conditional branch instructions this is stretched to 175 nanoseconds. The initial implementation will have 8K words of 100 nanosecond writable control store. With this configuration the fetch and execution of an indexed add to accumulator instruction will take 11 microcycles (ignoring memory access time which is zero if the instruction and data are in the cache) or about 1.65 microseconds.

A horizontal structure was chosen because of the obvious efficiency advantages in not needing to decode the microinstructions. However, we were concerned about the difficulty of developing large amounts of microcode for a micromachine with such a large number of bits per word. In order to simplify this task we developed a general-purpose micro-assembler which allows the code to be written for a pseudo-vertical machine and then maps this vertical code onto the real horizontal structure [31].

4.4 Address Translation and the Cache

The MONADS-PC processor generates 60 bit virtual addresses. These are translated into physical addresses by the address translation unit (ATU). The ATU contains a hash table with imbedded overflow. At all times the table contains an entry for every occupied physical page frame. The structure of each cell in the hash table is shown in figure 5. This table is used on a memory

access as follows. The page number portion of the virtual address (48 bits) is hashed to produce an address within the hash table. The key portion of the cell is compared to the page number, if it matches then the physical page number is extracted. Otherwise the link field is used to obtain the address of the next cell in the chain. This process continues until either a match is found or the end-of-chain is reached. The foreign bit indicates that a cell is part of a chain rather than being a primary entry point. This process of chasing chains is controlled by a (hardware) state machine. In order to minimise chain length it is desirable that the hash table be sparsely occupied. We are aiming for an occupancy ratio of four to one, which should require, on average, 1.1 probes to translate an address. Thus the processor can view the ATU as a black box which, given a virtual address, either reads/writes the data or causes a page fault. This type of address translation scheme is further described in [32].

Having made the decision to use Multibus, we had some concerns about the time taken to perform memory accesses. The Multibus only supports 16 bit transfers and thus most memory requests will require two bus cycles. The cache was included in the design to try to reduce the effect of the slow memory access on the overall performance of the processor. The cache holds both code and data and is one-way set-associative on virtual addresses. It operates in parallel with the ATU and uses a write-through update strategy. In the initial implementation the cache will be 16K bytes long, and we anticipate a hit-rate of at least 95% [33], although this is difficult to estimate given the MONADS architecture.

4.5 Software

Fortunately much of the software from earlier MONADS projects will be able to be used on MONADS-PC. In particular a Pascal compiler and a compiler for a modified form of Modula 2 [34] are available. These use a common code generator [35]

and a new code generator for MONADS-PC is currently being developed. It is envisaged that a small part of the kernel will be written in assembler and that the rest of the system code will be written in a high-level language, probably Module 2.

5 CONCLUSION

At the time of writing, the hardware for MONADS-PC is still at the design and construction stage. The microassembler [36] and much of the microcode has been written. An assembler for MONADS-PC machine code is under construction [37] and a new code generator is being developed. We view this initial implementation as a model for further research into computer architectures with support for good software engineering principles. In particular we are interested in designing a VLSI version of the MONADS machine.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the contribution to this work from Prof. J.L. Keedy, the instigator of the MONADS project.

REFERENCES

1. Georgiades, A., Richards, I. and Keedy, J.L. "A File System for the MONADS Operating System", Proc. 8th. Australian Computer Conference, pp 547-558, Canberra, 1978.
2. Keedy, J.L. "The MONADS Operating System", Proc. 8th. Australian Computer Conference, pp. 903-910, Canberra, 1978.
3. Ramamohanarao, K. and Keedy, J.L. "Job Management in the MONADS Operating System", Proc. 8th. Australian Computer Conference, pp 1476-1488, Canberra, 1978.
4. Richards, I. and Keedy, J.L. "Subsystem Management in the MONADS Operating System", Proc. 8th. Australian Computer Conference, pp 1520-1529, Canberra, 1978.

Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, 1985.

5. Rosenberg, J. and Keedy, J.L. "The MONADS Hardware Kernel", Proc. 8th. Australian Computer Conference, pp 1542-1552, Canberra, 1978.
6. Rosenberg, J. "The Concept of a Hardware Kernel and its Implementation on a Minicomputer", Ph.D. Thesis, Department of Computer Science, Monash University, 1979.
7. Wallace, C.S. "Memory and Addressing Extensions to a HP2100A", Proc. 8th. Australian Computer Conference, pp 1796-1811, Canberra, 1978.
8. Abramson, D.A. "Hardware for Capability Based Addressing", Proc. 9th. Australian Computer Conference, pp 101-115, Hobart, 1982.
9. Abramson, D.A. "Computer Hardware to Support Capability Based Addressing in a Large Virtual Memory", Ph.D. Thesis, Department of Computer Science, Monash University, 1982.
10. Rosenberg, J., Rowe, D.M. and Keedy, J.L. "An Overview of the MONADS Series III Architecture", Proc. 5th. Australian Computer Science Conference, pp 58-67, Perth, 1982.
11. Keedy, J.L., Abramson, D.A., Rosenberg, J. and Rowe, D.M. "A Comparison of the MONADS II and III Computer Systems", Proc. 9th. Australian Computer Conference, pp 581-587, Hobart, 1982.
12. Rosenberg, J. "The MONADS Series III Instruction Set", Proc. 9th. Australian Computer Conference, pp 704-718, Hobart, 1982.
13. Rowe, D.M. "MONADS III Inter-unit Communication", Proc. 9th. Australian Computer Conference, pp 719-729, Hobart, 1982.
14. Parnas, D.L. "Information Distribution Aspects of Design Methodology", Proc. 5th. World Computer Congress, IFIP-71, pp 339-344.
15. Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, 15, 12, pp 1053-1058.
16. Jones, A.K. "The Object Model, a Conceptual Tool for Structuring Software", in Bayer et al, "Operating Systems, An Advanced Course", Lecture Notes in Computer Science, 60, Springer Verlag, Berlin, 1978, pp 7-16.
17. Keedy, J.L. and Richards, I. "A Software Engineering View of Files", Australian Computer Journal, 14, 2, 1982.
18. Keedy, J.L. "On the Exportation of Variables", Australian Computer Journal, 12, 1, pp 23-27, 1980.
19. Keedy, J.L. "The MONADS View of Software Modules", Proc. 9th. Australian Computer Conference, pp 560-574, Hobart, 1982.
20. Lauer, H.C. and Needham, R.M. "On the Duality of Operating System Structures", ACM Operating Systems Review, 13, 2, pp 3-19, 1979.
21. Keedy, J.L. "A Comparison of Two Process Structuring Models", MONADS Report No. 4, Monash University, 1980.
22. Houdek, M.E. and Mitchell, G.R. "Translating a Large Virtual Address", IBM System/38 Technical Developments, pp 19-21, 1978.
23. Rosenberg, J. and Keedy, J.L. "Software Management of a Large Virtual Memory", Proc. 4th. Australian Computer Science Conference, pp 173-181, Brisbane, 1981.
24. Dennis, J.B. and Van Horn, E.C. "Programming Semantics for Multiprogrammed Computations", Comm. ACM, 9,3, pp 143-155, 1966.
25. Keedy, J.L. "Support for Software Engineering in the MONADS Computer Architecture", Ph.D. Thesis, Department of Computer Science, Monash University, 1982.
26. Intel Corporation "Intel Multibus Specification", Manual Order Number 9800683-02, Intel Corporation, 1978.
27. Rosenberg, J. "MONADS-PC Instruction Set", MONADS-PC Technical Report 1, Department of Computer Science, Monash University, 1984.

Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, 1985.

28. Keedy, J.L. "An Instruction Set for Evaluating Expressions", IEEE Transactions on Computers, Vol. C-32, 5, pp 476-478, 1983.
29. Abramson, D.A. "MONADS-PC Micro Architecture Manual", MONADS-PC Technical Report 2, Department of Computer Science, Monash University, 1984.
30. Advanced Micro Devices "Am2900 Family 1983 Data Book", Advanced Micro Devices, 1983.
31. Abramson, D.A. and Rosenberg, J. "A Vertical User Interface to Horizontal Microcode", 1984 (submitted for publication).
32. Abramson, D.A. "Hardware Management of a Large Virtual Memory", Proc. 4th. Australian Computer Science Conference, pp 1-13, Brisbane, 1981.
33. Strecker, W.D. "Cache Memories for the PDP-11 Family of Computers", chapter 10, "The PDP-11 Family", "Computer Engineering - A DEC View of Hardware System Design", Digital Press, 1978.
34. Wirth, N. "MODULA-2", ETH, Zurich, March, 1980.
35. Rosenberg, J. and Keedy, J.L. "Information Hiding - A Case Study", Proc. Conference on Computers in Engineering, Institution of Engineers, Australia, Publication No. 81/8, pp 6-9, 1981.
36. Tuke, M. "MONADS-PC Microassembler Specification", MONADS-PC Technical Report 4, Department of Computer Science, Monash University, 1984.
37. Rosenberg, J. "MONADS-PC Assembler Manual", MONADS-PC Technical Report 3, Department of Computer Science, Monash University, 1984.

BIOGRAPHIES

John Rosenberg

John Rosenberg has been involved with the MONADS project since its inception in 1976. His Ph.D. thesis related to the design and implementation of the kernel for the MONADS I system. Dr. Rosenberg was a Lecturer in the Department of Computer Science at Monash University until 1982 at which time he accepted a position as Senior Consultant with Fenwick Software. In 1984 he returned to Monash University and began work with Dr. Abramson on the design of the MONADS-PC system. Dr. Rosenberg's research interests include computer architecture, microprogramming, software engineering and operating systems.

David Abramson

David Abramson became involved in the MONADS project in 1979. He was responsible for the design and implementation of the MONADS II hardware, and this was the basis of his Ph.D. thesis. In 1982 Dr. Abramson accepted a Lectureship in the Department of Computer Science at Monash University. His interests include hardware, computer architecture, microprogramming and VLSI systems.

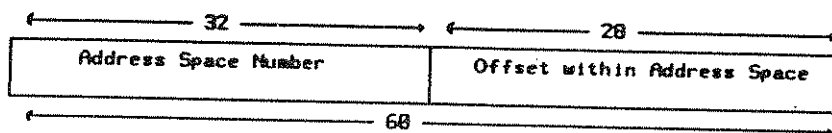


Figure 1 - A Virtual Address



Figure 2 - A Segment List Entry

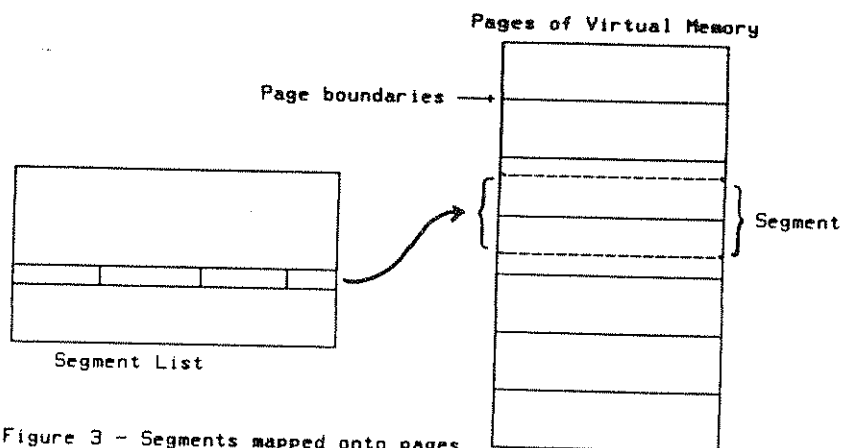


Figure 3 - Segments mapped onto pages

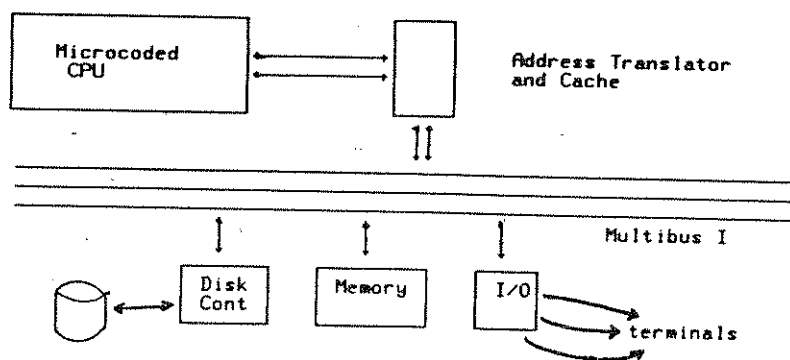


Figure 4 - The MONADS-PC Configuration

Virtual Page Key	Phys Page #	F	E	R	M	C	Link Pointer
------------------	-------------	---	---	---	---	---	--------------

Where:

F denotes cell is foreign
 E denotes cell is end of overflow chain
 R denotes page is read-only
 M is set by hardware when page has been modified
 C denotes page cannot be placed in cache

Figure 5 - A cell of the Address Translator Table