The Micro-Architecture of a Capability-Based Computer

*D.A. Abramson** - *J. Rosenberg***

* Department of Computer Science, Monash University, Clayton, Victoria 3168, Australia
** Department of Computer Science, University of Newcastle, N.S.W. 2308, Australia

ABSTRACT

This paper describes the micro-architecture of a microprogrammed workstation called MONADS-PC. The system has been specifically designed to support a very large uniform virtual memory, capability-based addressing and information hiding software modules with procedural interfaces. The paper gives a brief introduction to these topics followed by implementation details of the system.

1. INTRODUCTION

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. The MONADS-PC computer system [1] was developed during 1984 as a workstation to support software engineering research. The 32 bit microprogrammed processor provides hardware support for the decomposition of complex software systems into information-hiding modules [2].

The MONADS architecture is highly structured and places significant demands on the underlying hardware in order to achieve an efficient implementation. This paper discusses the implications of the architecture on the micro machine and describes some of the more novel implementation issues in the MONADS-PC computer. A full technical description of the MONADS-PC micro machine may be found in [3].

2. THE MONADS ARCHITECTURE

There are three central principles in the MONADS philosophy which affect the hardware, and thus are of concern to this paper. The first of these is the support for the decomposition of programs into information hiding modules, the second is the provision of a uniform virtual memory, and the third is the use of capability based addressing.

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 [4]. Parnas [2] and others [5] 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 [6]) and by forbidding the exportation of variables from modules [7]. Thus all conventional program units and files are implemented as information hiding modules and all data is accessed via a procedural interface [8].

In the MONADS-PC system the code and data of a module are tightly coupled. On entry to a module the data and code of the module become addressable, and the previous environment is no longer accessible. Moreover it must be impossible for the code to arbitrarily change the addressing environment. Thus, the procedure CALL instruction must set up the appropriate addressing environment for the called module and the RETURN must reinstate the former environment. Because the CALL and RETURN instructions are the key to the MONADS protection system a hardware (and firmware) supported implementation is essential.

2.2. 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 was first used on the MULTICS system [23] and has since been adopted on at least one commercial machine [9], 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 are discussed in [10]. 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).

2.3. Capability-Based addressing

The idea of capability-based addressing was first proposed by Dennis and Van Horn [11] and can be summarised as follows. Every object in the system is given a unique, non-forgeable name. These names are never reused, even when the object has been deleted. A Capability consists of a name and a set of access rights to the object identified. Capabilities 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 and can only be addressed in accordance with the access rights contained within that capability. 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. Virtual addresses in the MONADS system are embedded in capabilities and thus can never be reused. Consequently addresses must be even larger than required by a uniform virtual memory. An essential feature of capabilities is that they cannot be forged, and thus a user program must be prevented from directly constructing or modifying a virtual address. MONADS-PC provides a suite of system management instructions which manipulate virtual addresses in a well controlled manner.

3. ADDRESSING STRUCTURE

In the MONADS-PC system virtual addresses are 60 bits in size. This very large virtual address space is divided into regions called address spaces. Thus a virtual address consists of two parts, an address space number and an offset (Figure 1). Each address space contains data which is related in some way (e.g. the procedures of a module, segments of a file). The offset identifies a particular byte within the address space.

Address spaces are divided into many segments. Behind the segmentation scheme is a paged virtual memory. However, there is no direct relationship between segments and pages and segments may arbitrarily overlap page boundaries. The details of this scheme are described elsewhere [12]. Each segment is described by a capability which fully defines a contiguous region of virtual memory (Figure 2).

The virtual address (address space number and offset) within the capability defines the start of the segment and the limit defines the end. (Note that segments can not span more than one address space and thus the limit need only be 28 bits.) Executing programs can only access segments for which they have a capability. For efficiency reasons capabilities are never used directly, rather their contents are loaded into *capability registers* before use. The capability registers are loaded by a special instruction, *load-capability-register,* which takes the contents of the capability list entry and places it in the named register. Thus instructions need only name a capability register in order to refer to a segment of data. Access to individual bytes requires the specification of an additional offset within the segment.

4. INSTRUCTION SET

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.

There is a single accumulator which 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 is used for double length divide and multiply 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 (push/pop) area of the stack and one for fetching words of code from the

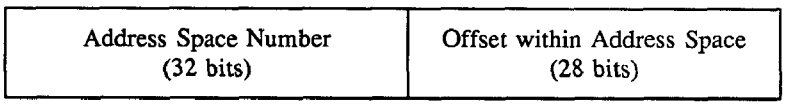

Figure 1 Virtual Address Format

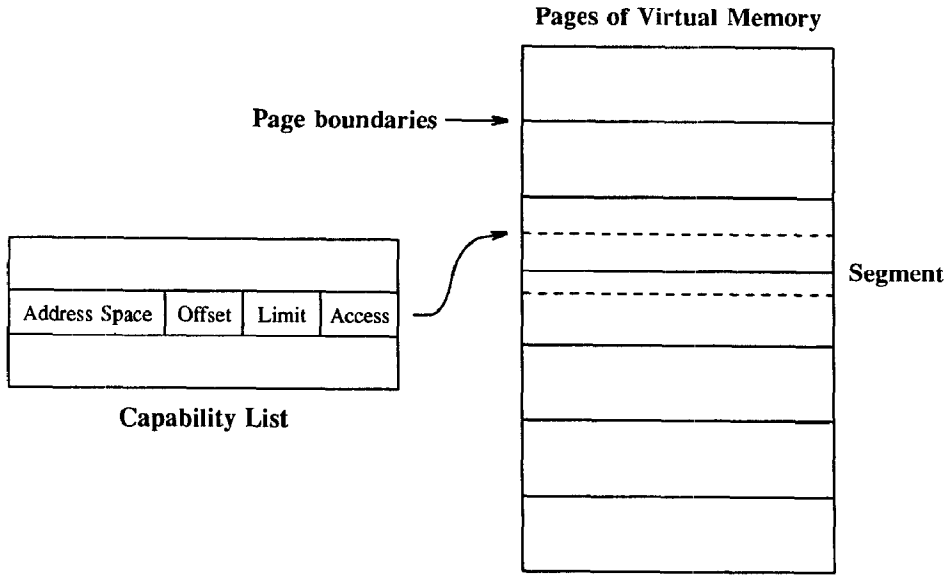

Figure 2 - MONADS Addressing Structure

instruction stream. Finally there is processor status word (PSW) which contains condition codes indicating the result of the last instruction.

MONADS-PC has 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 is added to this offset. The effective address is then compared with the limit value in the capability register. 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, as shown in Figure 3. The *opcode* is used to describe the type of instruction together with the addressing mode. Thus there are five different opcodes for each basic instruction; two for data mode (indexed and non-indexed), two for literal mode (16 bit and 32 bit) and one for top-of-stack mode. The *Cn* field determines which capability register to use, and the *In* field determines which index register to use. The *Offset* field is used for a fixed offset relative to a capability register in data mode, or the literal in 16 bit literal mode. In literal mode the offset is sign extended.

MONADS-PC has the usual complement of arithmetic and logical operations as well as an extensive set of bit manipulation instructions [13, 14]. 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 [15]. Limited arithmetic operations may be performed on the index registers to avoid unnecessary saving and restoring of the accumulator during array indexing operations.

The only byte instructions that are provided are load, store and compare. 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. Many of the complex instructions require more than one operand. The extra operands each use one code word following the instruction word.

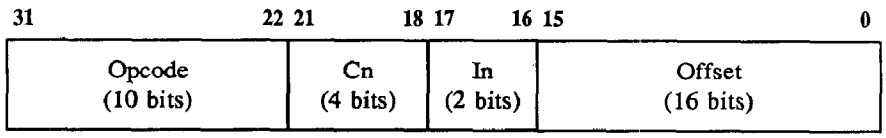

Figure 3 - A MONADS-PC instruction

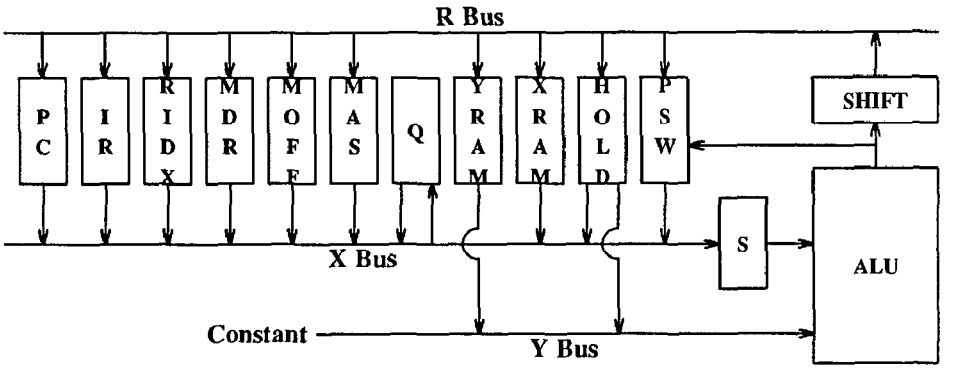

Figure 4 - the MONADS-PC bus structure

5. IMPLEMENTATION

5.1. Central Processor

The main processor consists of a microprogrammed control unit and a regular three bus arithmetic unit. The basic cycle time is 160 nanoseconds. However, some instructions take 200 ns. The buses are all 32 bits wide and connect the registers and ALU as shown in Figure 4. The X and Y buses form the two inputs to the ALU. The data on the X bus can optionally be shifted by a scaler (*S*) which is useful for turning word addresses into byte addresses. The output of the ALU may be optionally shifted or rotated before it is returned to the R bus. A constant from the current microinstruction may be placed on the Y bus.

The central processor has 9 individual registers and a high speed memory (2048 x 32 bits) for holding the program context. The PC holds the offset within the current procedure for the next macro instruction and may be incremented without using the ALU. The IR holds the current macro instruction. MDR is used to hold data to be written to virtual memory or data read from virtual memory. MAS holds the most significant 32 bits of the virtual address to be used for the next memory reference (The address space number). MOFF holds the least significant 28 bits (The offset). Q is a general purpose shift register used for multiply and divide operations. HOLD is a temporary holding register. HOLD is necessary because it is not possible to read from the RAM file and return a result to the RAM file in the one cycle, due to timing restrictions.

The high speed memory (X RAM and Y RAM) is configured as a dual ported register file. When data is written to the file, it is written to the *same* address in both the X RAM and the Y RAM. (This discipline is actually enforced by the micro-assembler [16, 17], not the hardware). Thus the contents of corresponding words in X RAM and Y RAM are the same. However, when data is read onto the X and Y buses, it is possible to address different locations in each memory at the same time. Such a scheme allows the ALU to operate on two register values to produce a result in one clock cycle.

The dual ported register file is used to hold information about the current process, including the capability registers, and the usage is summarised in figure 5. The *general register* section (64 registers) contains the dedicated code/stack capability registers, the index registers, the accumulator, special kernel registers and 32 temporary registers. Most of these registers are not visible at the macro instruction level. The rest of the dual ported register file contains 31 banks of 16 capability registers. The first 7 banks are used by the operating system kernel and the remaining 24 banks are used by user processes as described later.

There are 4 ways of addressing a word in the file. The first allows one scalar register value to be extracted. In this case the address of the register is directly encoded in the microinstruction. The second allows an index register to be extracted. In this case the address of the register is formed from the *In* field of the macro instruction register (IR). The third allows selected parts of a capability

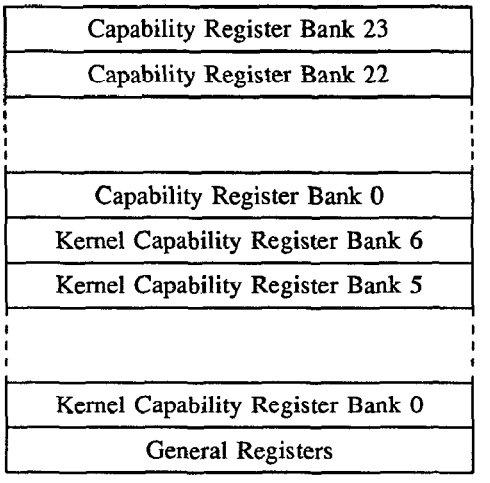

Figure 5 - the dual ported register file

register to be extracted. The register address is formed from the *Cn* field of the macro instruction register (IR), the current micro instruction, (which part of the register is required i.e. base, limit, offset or access rights, is encoded in the current micro instruction) and the contents of the RIDX register. The RIDX register is used as a bank select to determine which bank of 16 capability registers to use. The last mode allows a register to be selected depending only on the contents of the RIDX register. This mode provides an indirection mechanism for addressing RAM register values, and is useful for context save and restore operations, as well as machine diagnostics.

5.2. The Control Section

The bus structure described in the last section is controlled by a microprogrammed control unit. The control store has 8192 112-bit microwords and and thus a large amount of parallelism is possible. The microword has 63 individual fields which can be broadly placed in 8 categories. These are control of the X bus registers, control of the Y bus registers, control of the R bus registers (for saving data), control of the arithmetic and logic unit, control of the shifter, control of the microcode sequencer, control of memory, address translation and the cache, constants and jump target addresses. The structure of the microprogram control unit is shown in figure 6.

The control system uses a single pipelined scheme in which the next micro instruction is being fetched whilst the current one is being executed. The pipeline may be interrupted when a control transfer instruction is executed. The programmer has the ability to select a *branch* which makes the next instruction a no-operation, or one which will execute the next instruction. This feature has been used extensively in the microcode for the MONADS-PC macro instruction interpreter, and there are very few places in which the pipeline is broken.

The sequencer does not provide explicit support for microcode subroutines. However, it is possible to perform subroutine calls and returns using a special branch instruction. The control store address register may be reloaded with the contents of the R Bus, thus target addresses may be computed dynamically. Subroutines may be called using the normal branch instruction providing the return address is saved in one of the temporary registers. Since the return address is a compile time constant it is possible to save the return address using a microcode literal load. The return can be executed by placing the return address register value on the R Bus and executing a JUMP-ON-R-BUS micro instruction. Nested subroutines may be supported by choosing the return address registers so that each subroutine uses a different register. This scheme is considerably simpler than the conventional stack implementation used on many sequencers [18]. It provides flexibility in that there is no fixed limit on the depth of calls allowed (e.g. return addresses could even be held in memory if there are insufficient registers). Also the same mechanism can be used to implement microcode case statements and allows for non-hierarchical call and return sequences.

Another simplification made to the sequencer was in the area of interrupt detection. Because there is no complex micro-subroutine call facility, it is not possible to take interrupts after the completion of any micro instruction. Instead a special branch instruction is provided which tests the interrupt status and branches if an interrupt is pending. The MONADS-PC instruction interpreter checks for interrupts before fetching each macro instruction, and also at regular intervals during the execution of complex instructions. Such a scheme not only simplifies the hardware, but also indicates when the microcode is prepared to accept an interrupt, simplifying instruction restart.

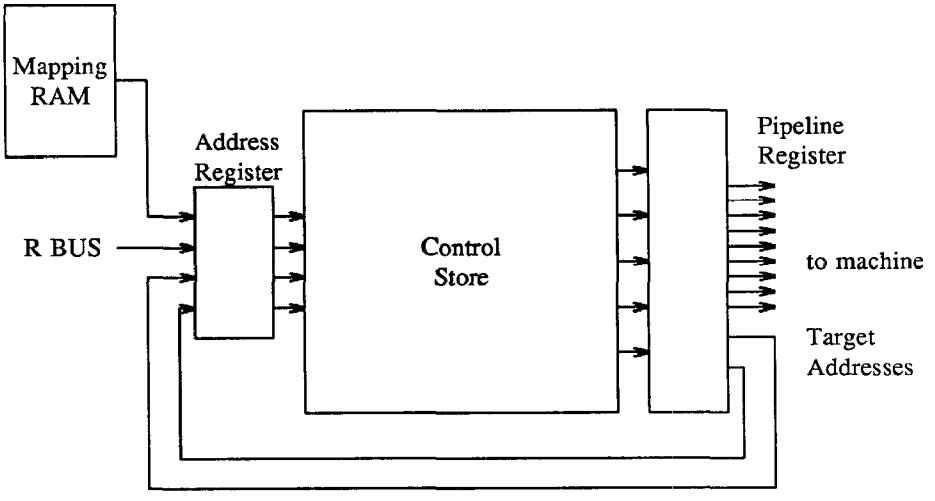

Figure 6 - the control unit

The starting address for the microcode interpreter for a particular instruction is found in a mapping memory. A special micro instruction causes the control store address register to be reloaded from the mapping memory. The mapping memory address is taken from the memory data register so that once a macro instruction has been fetched it is possible to transfer control to the correct microcode before the instruction has been placed in the macro instruction register.

A group of general branch instructions allows the control store address register to be reloaded conditionally with a value from the micro instruction register. Two 16 bit fields in the micro instruction are allocated for branch targets and constants which may be placed on the Y bus. Certain branch instructions require only one target, leaving the other free for 16 bit constants. A few branch instructions require two branch targets and choose the appropriate target conditionally. In such cases no constants may be placed on the Y bus. If neither of the branch targets is used then a 32 bit constant may be placed on the Y bus.

Main memory references are controlled by two fields of the micro instruction. One field is responsible for starting a memory operation (START-MEMORY-OP), and the other is responsible for waiting for the completion of an outstanding operation (SYNCHRONIZE). Before a memory operation can be started the microcode must place an address in the virtual address registers (MAS and MOFF) and, in the case of a write, data must be placed in the MDR register. When the memory cycle is started these registers are copied to a set of shadow registers thus allowing , MAS, MOFF and MDR to be reloaded before the next memory reference.

If both the START-MEMORY-OP and the SYNCHRONIZE fields are set in the same micro instruction then the memory operation will be started and execution of the micro instruction will be frozen until the cycle has completed. If only the START-MEMORY-OP is issued then the memory reference will be started and the micro sequencer will continue regardless of the status of the memory operation. A subsequent SYNCHRONIZE operation will cause the micro sequencer to wait until the operation has completed, or continue if the operation has already completed.

Since the memory operation is performed on a virtual address, a translation step is included before the main memory access can be started [19]. The translation and memory access are performed by an autonomous state machine. If the machine detects a page fault then the micro program must take corrective action, otherwise no action is necessary. When the SYNCHRONIZE instruction is issued the status from the state machine is tested, and if an error is detected a micro branch is executed. Thus the micro code sequence is only disturbed when a page fault or memory error is detected. This scheme allows pipelining between the processor and the memory subsystem, and is used extensively in the MONADS-PC system microcode.

5.3. The MONADS-PC Macro Instruction Interpreter

Each procedure of a program executing on MONADS-PC is contained in a separate segment of virtual memory. The base and limit addresses of the segment are held in registers of the dual ported memory. When a code word is fetched the program counter is added to the code segment base and a memory reference started. The program counter is compared to the segment limit. When the code word has been fetched it is placed in the instruction register and a mapped branch micro instruction executed. The simple macro instruction format makes instruction decoding easy.

As described earlier there are three addressing modes for each instruction. The addressing mode is encoded in the operation code of the instruction, thus there is a separate section of microcode for each addressing mode of each instruction. The most common addressing mode, indexed data mode, requires the following control sequence in order to generate a virtual address:

```
(1) capability_register.address_space -> MAS
    check capability_register.access_rights

(2) capability_register.base + IR.offset -> HOLD

(3) index_register + HOLD -> MOFF

(4) if (MOFF - capability_register.limit) > 0
        cause an error
    else
        perform memory operation.
```

Thus four machine cycles are required for every memory reference instruction.

In addition to the basic instruction set, there are many system management instructions [20]. The most important of these are the external module CALL and RETURN instructions. These instructions make use of the banks of capability registers in order to reduce the number of times the registers are invalidated and reloaded.

When an external module CALL is executed the addressing domain of the calling module must be invalidated and the environment of the called module must be created. On the return the domain of the called module must be invalidated and the previous environment recreated. Such a scheme can be enforced by saving the capability registers and then invalidating them on a CALL, and by reinstating them on the return. The main disadvantage of this scheme is that the CALL and RETURN require large amounts of data to be saved in and fetched from main memory. MONADS-PC makes use of the multiple banks of registers to reduce the main memory traffic.

The call and return instructions use the 24 banks of capability registers in a stack-like manner to avoid saving and restoring the registers on *every* CALL and RETURN. The scheme adopted is somewhat similar to the data register management scheme used by Patterson [21].

The MONADS-PC system provides many more banks than should be required to handle a normal nesting of modules. Thus, the dual ported memory is further divided into areas of banks of capability registers. An area is allocated to each user process executing on the machine, thus registers need not be reloaded when a process context switch occurs. If there are more user processes than areas an allocation algorithm must be applied when a context switch is executed.

5.4. Diagnostic Pathways

Many of the registers in the central processor are connected together through a serial shift chain. This allows a diagnostic processor (Motorola 6809) to read the contents of the registers and to insert a value in any register. A special diagnostic port also has control of the micro sequencer, and can halt, single step and run microcode. The pipeline register can also be loaded; thus it is possible to insert and execute a single micro instruction. The diagnostic processor can receive further instructions from another host processor which executes a large diagnostic program. This program can examine any register, load micro programs, load macro code programs and perform operating system functions. If the program needs to address a register which is not connected to the serial shift chain (e.g. the registers in the dual ported memory) it can insert a micro instruction in the pipeline register which copies data from the inaccessable register to one in the shift chain.

The 6809 microprocessor also is able to read the microcode from disk, load the control store, load the kernel from disk and start the operating system.

6. CONCLUSION

A prototype MONADS-PC computer has been constructed and is operational. The system has 8 serial lines, 2 Mbytes of memory and a modest amount of disk storage. Initial software includes an operating system kernel which handles memory management, process management and input-output and Pascal and C compilers. Work is continuing on higher level operating system modules and other language compilers.

In many respects the MONADS-PC micro architecture resembles that of other conventional machines. However, a number of unusual features are necessary for the efficient execution of the software architecture. These are the virtual memory management hardware, the capability registers, the microcode pipelining and the memory synchronization scheme. The capability registers, which are necessary in order the decrease the memory traffic and instruction size, are efficiently implemented using the dual ported memory arrangement. The use of branches which do not break the instruction pipeline are essential in order to guarantee the speed of the macro instruction interpreter. Similarly the memory synchronization scheme allows a maximum amount of work to be performed whilst the memory subsystem is busy.

The sequencer was not constructed from commercial bit slice components because of their lack of speed. However, the sequencer which is built from MSI components does not occupy any more space than a bit slice implementation, and is much faster. The subroutine calling mechanism and interrupt scheme were found to be quite effective in spite of their simplicity. Architectures other than MONADS-PC may benefit from faster but simpler micro sequencers.

Since the machine is still in the prototype stage, not many bench mark programs have been executed. However, early indications are that MONADS-PC will execute Pascal programs at about the speed of a VAX 11/750. There are many areas in which MONADS machines could be made faster than MONADS-PC. One of these is the address generation stage which is performed on every instruction fetch and most instruction executions. An alternative scheme has already been proposed [22]. The use of macro instruction pipelining, macro instruction addressable register stacks [21] and special purpose hardware for system management instructions would also increase the speed of the processor significantly.

REFERENCES

1 Rosenberg, J. and Abramson, D. (1985): "MONADS-PC - A Capability-Based Workstation to Support Software Engineering", *Proc. 18th. Annual Hawaii International Conference on System Sciences,* Honolulu.

2. Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules", *Comm. ACM,* 15, 12, pp 1053-1058.

3. Abramson, D.A. "MONADS-PC Micro Architecture Manual", MONADS-PC Technical Report 2, Department of Computer Science, Monash University, 1984.

4. Parnas, D.L. "Information Distribution Aspects of Design Methodology", *Proc. 5th. World Computer Congress,* IFIP-71, pp 339-344.

5. 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.

6. Keedy, J.L. and Richards, I. "A Software Engineering View of Files", *Australian Computer Journal,* 14, 2, 1982.

7. Keedy, J.L. "On the Exportation of Variables", *Australian Computer Journal,* 12, 1, pp 23-27, 1980.

8. Keedy, J.L. "The MONADS View of Software Modules", *Proc. 9th. Australian Computer Conference,* pp 560-574, Hobart, 1982.

9. Houdek, M.E. and Mitchell, G.R. "Translating a Large Virtual Address", IBM System/38 Technical Developments, pp 19-21, 1978.

10. Rosenberg, J. and Keedy, J.L. "Software Management of a Large Virtual Memory", *Proc. 4th. Australian Computer Science Conference,* pp 173-181, Brisbane, 1981.

11. Dennis, J.B. and Van Horn, E.C. "Programming Semantics for Multiprogrammed Computations", *Comm. ACM,* 9,3, pp 143-155, 1966.

12. Keedy, J.L., 1980. Paging and Small Segments: A Memory Management Model. Proc. 8th. World Computer Congress, IFIP-80, Melbourne, pp. 337-342.

13. Rosenberg, J. "MONADS-PC Instruction Set", MONADS-PC Technical Report 1, Department of Computer Science, Monash University, 1984.

14. Rosenberg, J. "MONADS-PC Assembler Manual", MONADS-PC Technical Report 3, Department of Computer Science, Monash University, 1984.

15. Keedy, J.L. "An Instruction Set for Evaluating Expressions", *IEEE Transactions on Computers,* Vol. C-32, 5, pp 476-478, 1983.

16. Abramson, D.A. and Rosenberg, J. "A Vertical User Interface to Horizontal Microcode", Proceedings of 8th Australian Computer Sciences Conference, Melbourne, Australian Computer Sciences Communications, Vol7, No 1, 1985.

17. Tuke, M. "MONADS-PC Microassembler Specification", MONADS-PC Technical Report 4, Department of Computer Science, Monash University, 1984.

18. Advanced Micro Devices "Am2900 Family 1983 Data Book", Advanced Micro Devices, 1983.

19. Abramson, D.A. "Hardware Management of a Large Virtual Memory", *Proc. 4th. Australian Computer Science Conference,* pp 1-13, Brisbane, 1981.

20. Rosenberg, J. "MONADS-PC System Management Instructions", MONADS-PC Technical Report 5, Department of Computer Science, Monash University, 1984.

21. Patterson D. (1985) "Reduced Instruction Set Computers", Communications of the ACM, Vol 28, No 1, pp 8, 21.

22. Abramson, D.A. and Rosenberg, J (1985) "Supporting a Capability-based Architecture in Silicon", The 4th Australian Micro-electronics Conference, Sydney, May 1985.

23. Organick, E.I. (1972) "The Multics System: An Examination of its Structure", Cambridge, Mass., M.I.T. Press, 1972.