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 micro-coded 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.

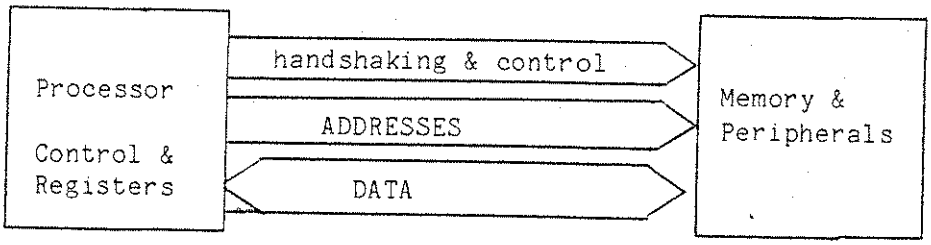


figure 1.

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].

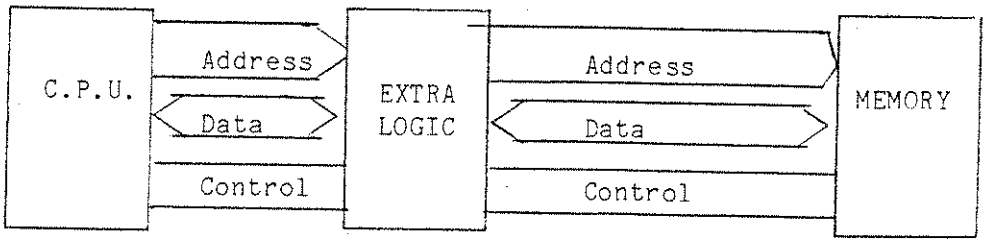


figure 2.

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.

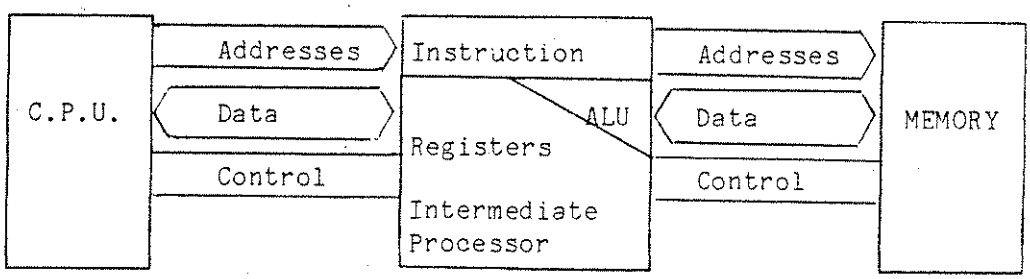


figure 3.

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.

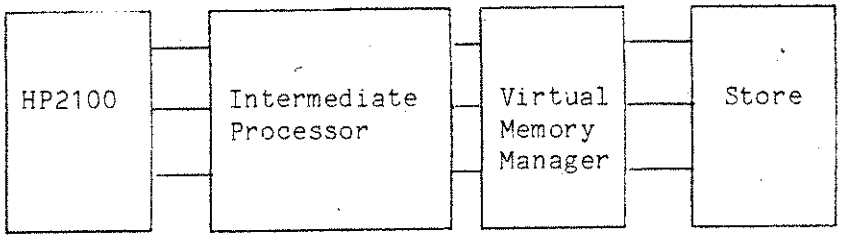


figure 4.

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.

AUSTRALIAN COMPUTER SCIENCE

COMMUNICATIONS

Volume 4, Number 1

February 1982

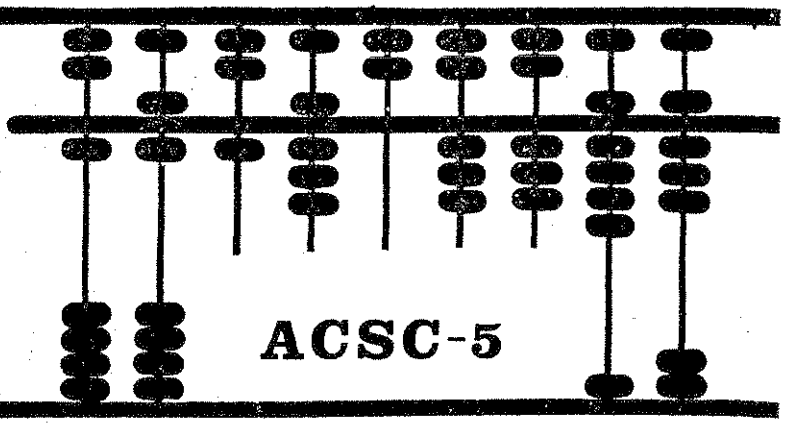
PROCEEDINGS OF THE

FIFTH AUSTRALIAN COMPUTER SCIENCE CONFERENCE

Department of Computer Science

University of Western Australia

Nedlands, WA 6009, Australia



ISSN 0157-3055