TWO EXTENSIONS TO THE ARCHITECTURE OF THE CONTROL DATA CORPORATION 6000 SERIES

By

John Dykstra

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

1979

ABSTRACT

TWO EXTENSIONS TO THE ARCHITECTURE OF THE CONTROL DATA CORPORATION 6000 SERIES

By

John Dykstra

The computer architecture first developed for the Control Data Corporation 6600 has not been modified in subsequent members of the family. This thesis examines two possible extensions to the 6000 architecture to determine whether it is unusually resistant to modification. One of the extensions considered provides a virtual memory feature. The other supports operation of the system in virtual machine mode. In addition, one possible implementation of the virtual memory feature is described at the register transfer level. No significant problems are detected in the virtual memory scheme, but the inclusion of peripheral processors in the 6000 architecture makes the virtual machine extension extremely awkward.

ACKNOWLEDGEMENTS

This thesis was prepared with the EASYDOC text formatting program, written by David Roderick and maintained by Douglas Nelson. Computer time for the use of the program was supplied through the assistance of Dr. Julian Kateley, Director of the MSU Computer Laboratory.

Two people have contributed not only to the work reported in this thesis, but also to the many other important things in my life. Dr. Lewis Greenberg changed the direction of my career with a job offer, and since then has become a valued professional adviser and personal friend. Pat Seger has helped in so many ways to make each suceeding year of our friendship "better and better." I offer to both of them my sincerest thanks.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER ONE

INTRODUCTION AND OBJECTIVES

When the 6600 was introduced by Control Data Corporation in 1964, it was by far the fastest computer available. During the following decade-and-a-half, CDC has introduced a number of successor systems in three different series, all with the same basic architecture. During the same period, the architectures of other major computer families (DEC PDP-11, IBM System 360/370) have undergone many changes and extensions.

It is natural to ask why, through several changes in implementation and realization, the architecture first developed for the 6600 has remained essentially unchanged. Several possible hypotheses can be advanced:

1. The original architecture was so good that there is little that can be done to improve it;

2. The market served by this family is atypical in that the users of the machine have not desired extensions;

3. There is a need and a desire for extensions, but the architecture presents unusual obstacles to the addition of these extensions;

4. The architecture is amenable to extension, but the implementations and realizations used by CDC for the family do not easily support these extensions;

5. CDC just hasn't had the corporate resources to design a new CPU with extensions. This hypothesis is supported by the length of time CDC stayed with the cordwood CPU designs.

One of the objectives of this study was to test the validity of hypothesis (3) by actually attempting (on paper) to extend the architecture. The other objective was to develop designs for two extensions which would alleviate what the author considers to be serious limitations of the present architecture.

The two extensions selected for detailed consideration are provisions for "virtual memory" and "virtual machine" operation. The ways in which these extensions improve the architecture under consideration are discussed in their respective sections. They are appropriate for testing the extensibility of the architecture because they affect many areas of the architecture, including instruction set, context switching and memory access.

CDC has marketed systems with this architecture under three different names (6000-series, CYBER 70, CYBER 170), and there is no commonly-accepted name for the family. In this paper we shall call it the 6000 family, after the first product line to incorporate the architecture.

Chapter Two of this thesis examines the changes neccessary to add a virtual memory feature to the 6000 architecture, and Chapter Three describes one possible implementation of this extended architecture at the register-transfer level. Chapter Four considers the virtual machine feature and its impact on system architecture. Conclusions from this study are presented in Chapter Five.

CHAPTER_TWO

VIRTUAL_MEMORY_FEATURE

The term "virtual memory" has come to denote a memory system which appears to the processors and I/O devices connected to it to be much larger than it really is. The "virtual" memory apparent to external devices is simulated within the system by a smaller "real" memory through a technique called "paging."

In the classic paging design, both virtual and real memory are divided into blocks called "pages." The real memory of the system is used to hold various pages of the virtual memory, without regard for the real addresses of the pages. Virtual memory pages not currently resident in primary memory are stored on random-access mass storage devices.

A part of the hardware called the "paging mechanism" maps the virtual addresses generated by the processor onto the pages of the real memory. If a virtual page referenced by the processor is not currently present in primary memory, the instruction in progress is interrupted, and a "page fault" is generated. This page fault results in a call to the operating system, which reads the virtual page into real memory from mass storage, and then restarts the interrupted program. The instruction which caused the page fault will complete normally, because its data is now resident in primary memory.

2.1 BENEFITS AND PENALTIES

The possible benefits obtainable from adding a virtual memory capability to the 6000 architecture include:

1. Permitting real memory in the system configuration to exceed 262K. Probably the most serious drawback of the current architecture is its limitation to this quite small memory size. A virtual memory mechanism is one, but not the only, way to evade this limit.

2. Permitting increased multiprogramming without excessive main memory requirements or swapping overhead. As memory sizes get larger, the proportion of the data in memory that will be accessed before the address space is swapped out gets smaller. This useless swapping traffic ties up the swapping device(s) and contributes to memory interference.

3. Capability of running all sizes of programs on any machine that is available, or when large amounts of

memory are defective or unavailable due to maintenance.

4. Permitting sharing of data and program storage between several jobs.

5. Possible support for a virtual machine capability (see Chapter Four).

6. Capability of protecting memory and controlling its usage at the page level.

7. Improved flexibility in re-configuring around failing memory modules.

8. Possible sales advantage in matching features of other manufacturer's machines.

Note that increasing the amount of real memory in a system ((1) above) will have no effect on the amount of memory that can be accessed at any one time by a user job. This is the processor address space, and is limited by the width of the registers and instruction fields of the CPU. Expanding this address space is a non-trivial task, and outside the scope of this study.

Possible disadvantages of a virtual memory capability include:

1. Slight increase in the memory access time due to the paging hardware. At one time, any proposed increase in access time would have been immediately rejected on the grounds that the performance of "scientific" computers is frequently limited by their memory bandwidth, and that therefore every spare nanosecond must be sqeezed out of the memory cycle time. However, it is very possible that the benefits of a virtual memory system would out-weigh the penalty of additional memory access delays. A detailed examination of the tradeoffs depends upon the details of an actual implementation, and so will not be attempted here.

2. Possible thrashing by programs that use a lot of array space. There is an extensive literature on the theoretical and empirical behavior of paging systems, but only a few studies [for example, 1, pp. 405-415] of systems whose memory references include extensive accessing of array-structured data. It would seem that some such programs could cause excessive paging to occur. The studies mentioned above have identified ways of reducing the severity of this problem, but most of these techniques require additional hardware. This topic strays from the objectives of this thesis, and will not be considered further.

3. Paging is really advantagous only in a multiprogramming system, where other useful work can be done while a needed page is moved into real memory. A sizable fraction of 6000 family installations use their machines essentially uniprogrammed, and many of the advantages given for virtual memory would not apply to them. Notice, however, that a uniprogrammed virtual memory system would not show significantly poorer performance as compared to a standard machine of the same real memory size. This is because once all pages of the program were in real memory they would stay there, incurring no additional paging time. The only performance penalty would be due to initial page fetches, and the longer memory cycle time due to paging.

4. Increased PP usage, central memory bandwidth consumption and bank conflicts, due to the paging traffic.

5. Increased hardware and operating system complexity and cost.

In the 6000 architecture there are three sources of references to central memory: the CPU, PP's, and ECS control. (Although ECS control may be considered to be a part of the CPU, its memory usage characteristics are sufficiently different from those of the CPU for it to be treated separately here.) The first sections of this chapter will develop the design of the paging hardware exclusively around CPU memory accesses. Later sections will extend this design to handle memory references from PP's and ECS.

2.2 BASIC PAGING OPERATION

The methodology used to develop the design will consist of beginning with a conventional paging scheme, and modifying it as needed to fit the unusual aspects of the 6000 architecture. The initial structure is similar to many well-known commercial systems, and therefore will be described only briefly.

The paging mechanism may be either enabled or disabled, under the control of a (newly-allocated) bit in the exchange package. When the mechanism is disabled, memory references proceed exactly the way they do in the current architecture. When it is enabled, all memory addresses are interpreted according to a virtual address space implemented by the paging mechanism.

To the program this address space appears contiguous, but the hardware breaks it up into fixed-size blocks called pages. Each page is either resident somewhere in real central memory, resident on some random-access mass storage device, or non-existent. In the later case, a reference to it must be resolved by the operating system.

A memory reference to a page that is resident in main memory is redirected to the proper real memory address by the paging hardware. To locate the page in real memory it uses a data base called the page table, which is kept in real memory. (There is a one-to-one correspondence between virtual address spaces and page tables.) The page table also indicates which pages currently exist only on secondary I/O devices. These are called "non-resident" pages.

A reference to a non-resident page results in a "page fault," which calls a portion of the operating system called the "paging manager." This software moves the page into real memory and restarts the interrupted program so that the memory reference can be retried.

2.3 PAGE TABLES

A page table describes the mapping between virtual and real pages for one virtual address space. Each entry in the table corresponds to one page, and contains a flag indicating whether the page is currently in real memory, and if it is, a pointer to the first word of the page. There may also be additional flags to record page usage and limit the ways in which the page may be accessed. The basic task of the paging hardware, given a virtual address, is to select from the current page table the entry for the page being referenced, and to generate the real address of the referenced word from the pointer in the entry and the original virtual address.

2.3.1 Organization

The first design decision to be made concerns how the page table entries are to be organized, and by implication, how the paging hardware will locate the proper page table entry, given a virtual address. There are two common schemes for page table organization, which here shall be called "associative" and "indexed." Both begin by partitioning the virtual address into two fields. One, called the "page ID," specifies which page contains that address. The other field, called the "word offset," specifies which word within that page should be referenced. Typically, the low-order bits of the virtual address are used as the word offset, so that a page will contain a contiguous portion of the virtual memory.

In the associative paging scheme, each page table entry contains its associated page ID. In most implementations of this scheme, pages that are not in primary memory or are non-existent have no entries in the page table, and page table entries can be in any order. Given a virtual address, the paging hardware generates the real address by associatively searching the page table for an entry that matches the page ID, and then adding the real address contained in that entry to the word offset of the virtual address. The virtual memory mechanism [2, pg. 4-46] of the CDC STAR uses this scheme.

In the indexed scheme, the page table entries are arranged in order of page ID's. The page ID field from the virtual address is used as an index to select the proper entry. From then on, the procedure is identical to that of the associative scheme. There must be entries in the page table for all pages, even those not in primary memory or those that do not exist altogether (although if there is a page table length register then there need not be entries for non-existent pages at the high end of the address space). Many commercial machines have used this scheme.

Perhaps the best known is the IBM System 370 [3, pp. 99-130].

A paging mechanism must operate extremely quickly to avoid degrading system performance, and several existing designs incorporate features toward this end. For example, the associative method as described above requires an associative memory large enough to contain the entire page table. Unfortunately, memories this large are quite expensive, and not exceedingly fast. Instead, the STAR uses a linear search procedure to locate the proper page table entry [2, pg. 4-53]. It compares the page ID field from the virtual address with the contents of the corresponding field from each page table entry in turn until it finds a match or the end of the table is encountered. If a match is found, that entry is moved to the beginning of the page table, and the entries between the first and the original position of the matching entry are moved down one position. The entry is also used to generate the real address for the reference. (To improve efficiency, the STAR implementation keeps the top 16 page table entries in processor registers rather than memory. This scheme essentially implements a cache whose maintenance is automatically performed as part of the overall paging mechanism.)

There is no clear-cut winner in comparisons between the indexed and the associative schemes. The indexed approach is fastest and easiest to implement, but it may require large page tables, only a few of whose entries are actually used. (This characteristic is not a serious problem if the page tables themselves can be paged.) The associative scheme is much more complicated to implement, but will result in smaller page tables, especially if many of the pages in the address space are non-existent or not currently resident in real memory.

If a push-down algorithm like the STAR's is implemented, it is simple for the operating system to identify the least-recently-used page within that address space (not across all system address spaces), simply by choosing the last entry in the page table. This operation is necessary when the paging manager must select a page to move out to secondary storage, and in the indexed scheme is usually approximated through the use of page table entry flags. There is, of course, a penalty for the convienience of the push-down scheme, in the form of additional hardware complexity. This algorithm might also very well be inappropriate for implementations that provide an associative cache for the page table.

As indicated above, the indexed method requires the least amount of hardware, and its only disadvantages are larger page tables, and the need for use-bits in the page table entries for identification of seldom-used pages. These disadvantages are less severe than those of the associative method, which include additional hardware complexity and unsuitability for some implementation approaches. Therefore, the indexed scheme will be selected for use in this design.

2.3.2 Page table entries

The most important component of each page table entry is, of course, the real memory address of the page (assuming that the page is currently resident in real memory). Since one of the goals of this design effort is expanding the allowable system configuration to include more than 262K of real memory, the page address field should be wider than 18 bits. Two possible candidates for the size are 24 bits (two PP bytes) and 30 bits (one-half of a memory word). In the interests of maximum flexibility, and since it is unlikely that PP's will do much manipulation of page table entries, the later size will be chosen.

As mentioned in the previous section, efficiency of the paging mechanism is an important design goal. The paging hardware would operate much faster if the real address for a memory reference could be formed by concatenating the word offset from the virtual address with the page address from the page table entry, rather than by doing an addition. This is possible, of course, only if the low-order bits of the page address corresponding to the word offset are zero. If they are always to be zero, there is no point in including them in the page table entry. Thus, the 30 bit page address in the page table entry will include only the high-order bits of the real address of the page.

Each page table entry must also contain a flag which indicates that the associated page is not currently in real memory. This flag is called the fault flag. Note that when this bit is set, the paging hardware will immediately signal a page fault, and will not use the page address field in any way. This field is therefore free for the monitor software to use in any way that it sees fit. One possible use is to contain the secondary memory address of the page.

When the paging manager moves a page out of real memory to make room for another page, it is best to select a page that will not be referenced again for some time in the future. The best available approximation to this ideal is selection of the least-recently-used (LRU) page. Efficient implementation of this scheme requires the paging hardware to maintain some information about page references.

This information takes the form of use counts in each page table entry. Every time a reference is made to a word within a given page, the use count of the appropriate page table entry is incremented by the paging hardware. If the count was already as high as can be represented in the count field, it is kept at that value, rather than rolled-over to zero.

To approximate a LRU algorithm when selecting a page to move out, the paging manager chooses the page whose use count is lowest. It then resets all use counts in the page table to zero, to prevent the counts from preserving information about events in the distant past.

The correct size for the use count field in the page table entries depends upon the page size selected for the design and the page fault characteristics of the running system. It seems sufficient at this time to allocate 12 bits for the count, since this is far more space than will actually be needed. Particular implementations can use only a part of this field for the count, as long as the remainder is kept zero.

As an additional aid to the paging manager, the hardware should remember whether any word within a particular page has been written into by the CPU. This function is performed by the write flag, which is initialized as clear when the page is first moved into real memory, and set by the paging hardware if a write to that page takes place. When the paging manager selects this page to be moved out it checks the state of the write flag. If the flag is clear, and a copy of the page already exists in secondary memory, there is no need to write out the page.

So far we have defined 44 bits within the page table entry. These fields all fit within a 60 bit memory word with room left over, so the size of each page table entry will be defined to be one memory word.

2.3.3 New CPU registers

Of course, the paging hardware must have some way of finding the page table in memory. The obvious solution is the addition to the CPU of a "page table pointer register," which contains the address of the first word of the page table. The paging hardware adds the page ID to the contents of this register to find the address of the proper page table entry.

CPU exchange jumps usually involve a change in address space, so it is natural to include the page table pointer register in the exchange package. Virtual memory makes obsolete the function performed by the RA register, so its space within the package can be reused for the pointer.

Most programs will use only a portion of the 262K word virtual address space, and it is desirable to shorten the page tables of these address spaces accordingly. A straight-forward solution is to add a "page table length register" to the CPU. Each page ID is compared to the contents of this register before the page table entry is fetched. If the page ID is too large, a bounds error is signaled and the reference abandoned. Thus, the page table need have entries only up to the highest page in the address space that will be referenced. The page table length register is similar in purpose to, and can replace in the exchange package, the FL register of the current architecture.

2.4 PAGE SIZE

Page size is a design parameter that must be determined through the resolution of performance tradeoffs. It is usually chosen to be a power of two to facilitate the partitioning of the virtual address, and the page size of existing machines range from 64 words [4, pg. 150] to 65K words (the larger of two possible page sizes in the CDC STAR [2, pg. 4-46]).

Expected addressing patterns are relevant to one trade-off: If references are fairly sparse across the address space within a moderate interval of time, a small page size avoids the overhead (both in I/O traffic and in real memory space) wasted by bringing in large pages, only a few of whose words will be referenced before they are moved out again.

Page size also affects the cost of the paging hardware. For a given virtual address space size, a small page size results in a large page ID field. It also reduces the number of address bits that need not pass through the paging mechanism, and are therefore immediately available to the rest of the memory system. Use of these "early" bits can often improve the performance of an implementation.

Quantitative analysis of these tradeoffs is beyond the scope of this study. Therefore, a page size of 4096 words will be chosen rather arbitrarily. This size does have the advantage of corresponding to the 12-bit byte size of the PP's, which might simplify some of the PP software.

2.5 PAGE TABLE ADDRESSING

Section 2.3.1 mentioned lengthy page tables as one disadvantage of the indexed method, and proposed paging the page table as a solution. However, the moderately-large page size we have selected reduces the significance of this problem. A 262K virtual field length (the maximum allowed) will require only 64 page table entries. Therefore, it is practical to keep the entire page table in real memory.

Since page tables are so small, there is no need for the page table length register described in Section 2.3.3. It is quite practical for the system to use a fixed page table size of 64 words for all address spaces, and simply set the fault bit of all entries corresponding to out-of-bounds entries. Eliminating the page table length register also frees a 18 bit field in the exchange package for other uses.

Section 2.3.3 described how the page table pointer register is included in the exchange package, but left the exact interpretation of the address contained in that register unspecified. Since we have elected not to page the page table, this address can simply be the real address of the table. If the contents of this register are to replace those of the FL register in the exchange package, the register width will be limited to 18 bits. Thus, all page tables would have to be located in the first 262K of real memory. There is space available in the exchange package with which the register could be widened somewhat, but to expand it to the 30 bit size of the real address field in the page table entries would be difficult. In any event, there is no real need to expand this register beyond 18 bits, for it is unlikely that the entire 262K space would ever be filled with page tables, and with the virtual addressing scheme there is not much else that must go there.

2.6 FAULT MECHANISM

Proper design of the interface between the paging hardware and the monitor at page fault time is crucial to the success of the project. It must be efficient, and it should also promote good structuring of the monitor.

2.6.1 Exchange Address

In the current architecture, exchange jumps are already used to signal other asynchronous events to the CPU, and it seems natural to use this mechanism for page faults too. The first question raised by this choice concerns the exchange address; i.e., the location from which the exchange package for the paging manager is loaded, and the exchange package of the faulting program stored. This location might be the same as used for other types of exchange jumps (the Monitor Address), or could be separate and reserved exclusively for page fault exchange jumps.

This question is closely related to decisions about the structure of the monitor, because exchange packages are equivalent to the entry and exit points of this software. Having one exchange package implies that the monitor has a single entry point, and appears (to user jobs, at least) to be a single block of code. If there is a specialized exchange package to be used after page faults, by implication the system software is divided into two parts--the paging manager and the remainder of the monitor.

The two-exchange package scheme is initially attractive. It isolates all of the support software for the virtual memory feature (the paging manager) from the remainder of the monitor. This would be especially useful if the monitor itself was paged. In addition, page faults could be processed quicker since the paging manager was entered directly rather than through some intermediate code.

Despite these advantages, the single-exchange package scheme will be chosen for this design. This choice is based on an analysis of the best structure for the monitor software.

2.6.2 The Hub

The code that makes up the CPU portion of the monitor can be divided into several "tasks." Each task is concerned with a particular aspect of the monitor's operation, and typically maintains its own tables and local variables. If this concept is expanded to include user job's as tasks, all of the programs to which a CPU can be allocated are tasks. The form of this allocation is either subroutine-like, in which the task keeps the CPU until it is completed, or coroutine-like, in which the task may give up the CPU at various points in its execution, in expectation of getting the CPU back at a later time.

The code that allocates the CPU's to various tasks has several functions to perform besides the elementary one of controlling the CPU's. First, this code implements the various wait states that tasks may pass through as their execution progresses, including waiting for a page fault to be resolved. Secondly, since some monitor tasks may be reentrant, the CPU allocation code must not create a new incarnation of such a task if another incarnation of the same task is waiting on a page fault. Thirdly, in a multiple-CPU system the CPU allocation code is responsible for implementing mutual exclusion between the various CPU's through keeping the CPU away from a task until another CPU is done with it.

Thus, the CPU allocation code controls the operation of the rest of the software, both user and system. It is also one terminus for all transfers of control between the various modules, as shown in Figure 1. This code will therefore be called the "Hub."

Since the CPU allocation module implements software interlocks, simulaneous execution of it by different CPU's must be prevented. This can be achieved through use of the monitor flag already included in the 6000 architecture. This hardware flag is cleared at deadstart time, and toggled by each subsequent exchange jump. If one CPU tries to do an exchange jump while another CPU's monitor flag is set, the exchange jump is delayed until the other's monitor flag is cleared.

If the monitor flag is set during execution of the CPU allocation code, simultaneous execution by multiple CPU's is prevented. Since the tasks called by the CPU allocation code are protected through its software interlock function, they can run with the monitor flag

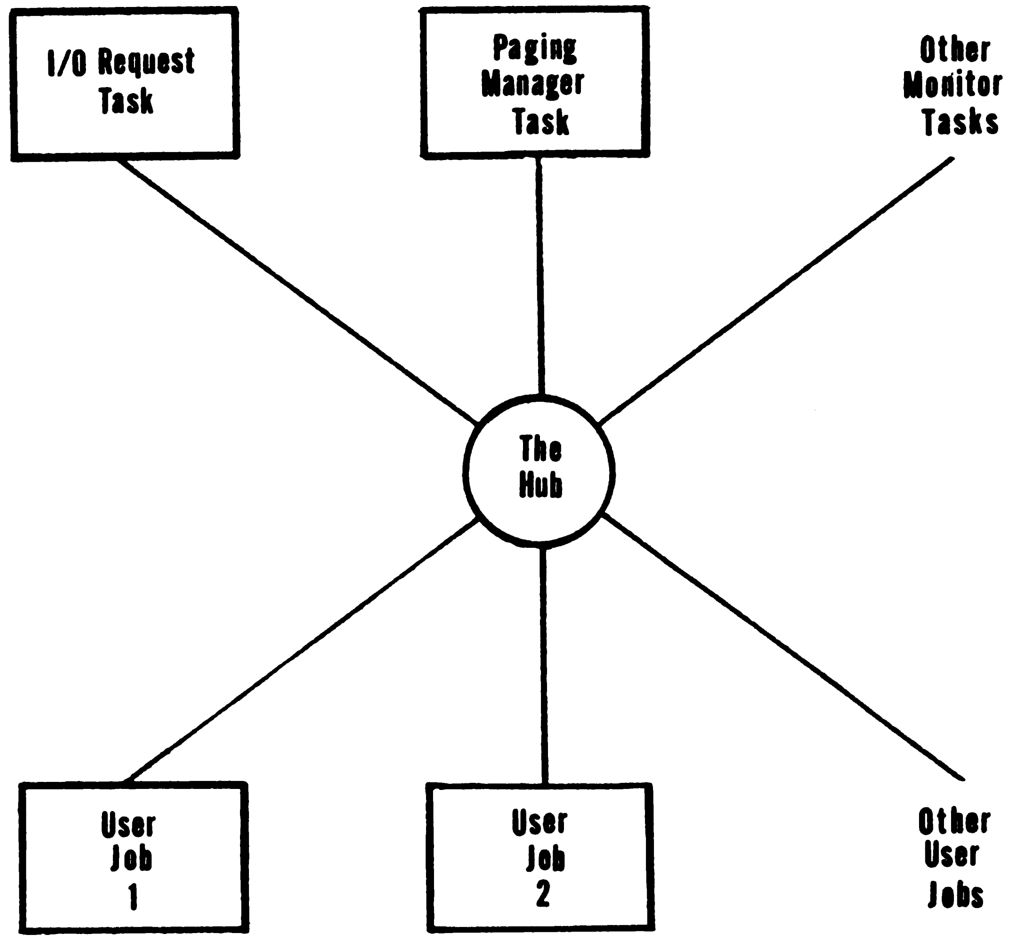

Figure 1: The Hub Concept

cleared. Thus, the amount of time that the Hub is occupied by a CPU is minimized.

This results in the following structure: When a user job is interrupted, the exchange jump begins execution of the Hub, with the monitor flag set. The Hub then selects a new task to execute, and calls that task with an exchange jump, which clears the monitor flag. When the task is completed, or is interrupted for any reason, the CPU returns to the Hub, which selects a new task to run.

Thus, the Hub must be called by the page fault mechanism, just as it is called after any event that causes a monitor exchange jump. The Hub should have single entry point, so page faults will use the same exchange package address as other types of exchange jumps.

(Note: A variation of the Hub concept was independently developed by Lawrence J. Kingsbury, et. al. [5, pg. 2].)

2.6.3 Provisions for Restart

Most processors that use paging execute their instruction streams serially; i.e., each instruction's execution is completed before the next's is begun. Memory reference instructions also frequently include some sort of data manipulation along with their memory reference function. In such systems, an instruction that causes a page fault must be interrupted, and restarted from its beginning when the referenced page has been fetched from secondary memory. This is easily accomplished in such systems by setting the instruction pointer of the interrupted program to point to the instruction that caused the page fault.

This method of restarting after a page fault is not appropriate for the 6000 architecture, because this architecture was designed from the beginning to support parallel execution of instructions. By the time a page fault is recognized, some implementations might have already completed several of the instructions that follow the memory reference in the instruction stream. If the P register was reset after the page fault to point to the faulting instruction, these instructions would be re-executed, with disasterous results.

Fortunately, the parallel design of the processor implies that only _memory references_, not _instructions_, need be restarted. When a page fault occurs, the CPU can complete any non-memory reference instructions that it is currently executing, and leave the P register in its exchange package pointing to the next sequential instruction word. In addition, the CPU must somehow identify in the exchange package the memory reference that caused the page fault.

When this exchange package is reloaded into the CPU after the page fault has been processed, that reference will be restarted, and instruction execution resumed at the next sequential instruction.

All that need be added to the exchange package to provide this restart information is a "reissue" field of 7 bits, each of which is associated with an A/X register pair. (A0/X0 are not used for memory references, and therefore need no flag bit.) A particular bit is set if a memory reference from that register pair caused the page fault. When the exchange package is reloaded into the CPU, the memory reference is reissued, using the address and data information still present in the A/X register pair.

There are several complications that force modifications to this scheme. Most importantly, some parallel implementations of the architecture may allow more than one memory reference to be active at the time a page fault is detected. Also, there may be more than one instruction in a memory word. In the modified fault sequence, the CPU finishes instructions that were in progress when the fault was detected, but does not begin execution of any more. A new two bit field in the exchange package is used to indicate where (in the memory word pointed to by the P register) execution should resume. The system then waits for all outstanding memory references to be completed. If any of these references cause additional page faults, their reissue bits are set in the exchange package, along with that of the reference that caused the original fault. When there are no more active memory references, the page fault exchange jump occurs.

When the exchange package is reloaded into the CPU, the memory reference that caused the page fault will complete normally, as will any other reissued references that access the same page. If any of the reissued references access another page, which is still not in primary memory, another page fault will immediatly occur so that this page can be fetched.

Waiting for all memory references to complete before doing the page fault exchange jump might be a serious waste of time in some implementations. It can be avoided by aborting all outstanding memory references when a page fault is first discovered, and setting the reissue bits so that they all are restarted when the exchange package is ultimately reloaded into the CPU.

2.6.4 Supplying Fault Information

Once the page fault has occured, the paging manager must be able to identify which page it should copy from secondary storage into central memory. This information could be presented in several different forms. The most basic would be the virtual address that caused the fault. Since the word offset is of no use to the paging manager, this could be simplified to the page ID portion of the address. The paging manager would add this to the address of the page table (obtained from the user's exchange package) to determine the address of the faulting page table entry.

This, however, is exactly what has already been done by the paging hardware, and if this address is provided to the monitor by the hardware, the software can go directly to the page table entry. This method will be chosen for the design.

There are several possible methods for transmitting the page table entry address to the paging manager software. It could be placed in a particular CPU register when the Hub's exchange package is loaded. This has the disadvantage of possibly constraining the Hub software in its use of registers. In addition, the Hub has no need for the detailed information about the page fault: All it needs to know is that a page fault has occured, so that it can call the paging manager. If it must pass the detailed information to the paging manager, the length of time the monitor flag is set will increase. This could reduce the perfprmance of the system.

The alternative is providing a new CPU instruction, which loads the address of the faulting page table entry into a register specified by the instruction. Although this method is slightly slower than automatically loading the information into a register, it has none of that method's disadvantages. If the instruction is executed when there is no page fault outstanding, the value loaded into the register will be zero. Thus, the Hub can determine whether a page fault has occured with a simple zero-test.

2.6.5 The CPU Error Sequence

The current architecture defines an error sequence that is performed by the CPU when it detects an error in the program it is executing (undefined operand, memory bounds error, etc.). This sequence includes a write tp the first word of the user address space to store information about the error. This write presents a problem if the page containing this word is not resident in real memory, because there is no way to set the user's exchange package so that the error will be repeated after the page fault is serviced. It seems that the best way to handle this problem is to require the monitor to keep in real memory the first page of any user who is eligible for execution.

2.7 PROCESSOR USAGE

So far we have spoken of the paging manager as a part of the monitor running in the CPU. However, the original system concept for the 6000 family [6, pg. 4] put most of the operating system functions into the PP's, and reserved the CPU for user program execution. Since it is desirable to minimize the amount of CPU time used in paging overhead, and since page fault handling is mainly an I/O operation anyway, it is reasonable to consider assigning primary responsibility for servicing page faults to the PP's.

The initial problem in implementing this scheme concerns how to notify a PP that a page fault has occured. In the current architecture there is no way to "interrupt" a PP. If such a mechanism was added, some method would have to be developed for the hardware to determine which PP's were busy with other tasks and which could be interrupted to perform the paging. Complications such as this make this approach unattractive.

The alternative is the method currently used by 6000 operating systems to handle asynchronous external events--dedicating a PP to watch for event to occur. The falling cost of hardware probably makes this lavish use of PP's acceptable.

Note, however, that not much is gained by handling page faults in the PP's. If the page fault was caused by the CPU (there will be other types of page faults--see below--but the CPU is the most likely source), the CPU might as well be exchange jumped by the fault signal, since the job in execution will have to be blocked until the paging I/O is completed anyway. Once the CPU is exchange jumped into the monitor, it can probably notify the paging service PP of the fault almost as quickly as a direct hardware connection could. In addition, the CPU could direct the paging request to whichever PP it chose.

In summary, there is no significant advantage to allowing PP's to directly receive paging requests. Accordingly, this design will direct the fault information to the CPU, as described in the previous section.

2.8 INTER-ADDRESS SPACE REFERENCES

In the virtual memory system, pages belonging to user address spaces are scattered over real memory, which may exceed the 262K span that the CPU can reference using real addresses, and some user pages may not be resident in real memory at all. This presents a problem to the CPU and PP portions of the monitor, who are often called upon to reference various portions of a user address space in performing monitor functions.

Although both the CPU and the PP portions of the monitor need to reference user address spaces, there are significant differences in the way that the two will perform these references. The CPU portion of the operating system will be executing from its own address space, and most of its data will be located there too. The CPU's references to other address spaces will mostly be limited to fetching and storing operands for system action requests. On the other hand, the PP's execute out of their own memories. They also reference system tables in the monitor address space, but due to their function as I/O processors for the system, their references to user memory frequently take the form of block transfers.

Due to these differences, this section will only consider the CPU side of the problem. A later section will extend the concepts developed herein to the PP's.

Without some assistance from the architecture, the CPU monitor would face serious difficulties when it tried to reference a user address space. Addresses given to the monitor by the user or that the monitor maintains itself would be in virtual form, and these would first have to be translated to real addresses. As the program performed the translation it would have to ensure that the page was currently in real memory, page it in if it was not, and then lock it in primary memory to ensure that its real address did not change. To perform the actual reference the CPU would need several new instructions that would accept real addresses larger than 18 bits. If the CPU subsequently needed to reference additional words (perhaps part of the same table), it would have to check to see if these other words were in the same page, and repeat the entire procedure if they were not.

If the monitor had to follow this procedure for every reference to a user address space, the software would be both complex and inefficient. The alternative is to use the virtual memory mechanism for these references. This requires some modifications to the Hub/paging manager software discussed in Section 2.6.2, to process page faults from monitor tasks. Once these modifications have been made, entire monitor tasks, not just the user address space references, can run in virtual memory mode, and this will be assumed in the discussion that follows.

One possible scheme using the virtual memory mechanism would consist of duplicating some of the user's page table entries in the monitor's page table. Upon being given an address in the user's address space, the monitor would locate the corresponding entry in the user's page table, and copy this entry into a pre-arranged location in the monitor's page table. It would then be a simple matter for the monitor to compute the address of the target word in its own address space and perform the reference.

Such complications as non-resident pages would be automatically handled by the usual virtual memory mechanisms. If the monitor anticipated referencing more than one word within the same area, it could simply copy enough page table entries to assure that all references would lie inside the block. It could then ignore the page boundaries.

The above scheme would still require the CPU program to do a significant amount of work to perform the cross-address space access. This work could be entirely shifted to the hardware through the implementation of a set of "alternate address space" instructions. One of the operands of these privileged instructions would be a pointer to the page table of the address space the monitor wishes to access. These instructions would execute similarly to the increment instructions, but the value set into the A register would be interpreted by the paging hardware using the alternate page table. Both non-resident pages and page boundaries would be handled automatically by the paging hardware.

This is the scheme that shall be adopted for cross-address space references. Further details of the alternate address space instructions are simple to develop, and will not be considered here.

2.9 MONITOR-MODE PAGING

The solution to the problem of inter-address space references proposed in the previous section requires that the monitor tasks run with the virtual memory feature enabled. However, it is not immediately obvious whether this is practical, or what changes to the architecture might be necessary to make it practical.

There are, of course, other advantages to using virtual memory in the monitor besides facilitating inter-address space references. Perhaps the most important benefit is programming simplicity, since the monitor code can be written with minimal concern for memory management or the overlaying of program segments. An additional benefit is a reduction in the amount of real memory consumed by the monitor, since the size of the working set will almost certainly be smaller than the storage required by an equivalent non-paged monitor.

There will undoubtedly be some monitor tasks that cannot run in virtual memory mode. Deadstart processing and recovery from hardware failure are two of these. However, the Multics operating system [7] has shown that an operating system can be written that normally uses only virtual memory, so the number of these tasks should be quite small.

One potential defect of a paged operating system is inefficiency due to frequent waits for paging. The harmful effects of paging can be minimized, however, by keeping frequently used pages permanently resident in real memory.

Paging monitor tasks puts some new restraints on the interface between these tasks and the Hub. These tasks can now become unrunable for the duration of the paging I/O, and it is up to the Hub to recognize this, and hold subsequent calls to a blocked task until the first task can be completed. Thus, no call to a task may assume that the task will immediately be run.

If either the Hub or the paging manager caused a page fault, there would be no way to process that fault without falling into infinite recursion. Thus, both the paging manager and the Hub, and all data referenced by these modules, must kept resident in real memory at all times.

2.10 ECS

The CPU instruction that starts an ECS transfer uses two addresses--one for central memory and one for extended core storage. We shall leave the ECS addressing scheme as it is, but must decide whether the CM addresses used in ECS transfers will be real or virtual addresses when the CPU is running in virtual mode.

Since ECS is controlled by CPU instructions, it seems most consistent for it to use virtual addresses just as the rest of the CPU instructions do. However, before we accept this choice we must make sure that it does not result in problems when ECS is used as a paging device.

It is true that when the page fault occurs there is no page table entry pointing to the location to which the page is to be read. However, such an entry must be created by the paging manager anyway, to be inserted into the page table of the faulting program. If the entry is formed before the page is read in, it can be used for the ECS transfer too. Thus, ECS can use the virtual memory mechanism without problems.

Since ECS is so closely related to the CPU, it can share the CPU's page table pointer register. ECS transfers never overlap exchange jumps, so the addresses specified by ECS instructions will always be interpreted according to the proper address space.

The fault mechanism developed for the CPU can also be used for ECS transfers. However, the current architecture makes no provision for restarting an ECS transfer at the point at which is was interrupted. Instead, after the paging I/O is completed, the ECS transfer must start again from its beginning. This situation results in two problems. The first is that if a large transfer encounters several page faults, it will have to be restarted from the beginning several times, resulting in much wasted memory traffic.

The second problem is a result of the first. There is the possibility that the paging manager, in processing one page fault, will remove a page from real memory that was referenced earlier in the transfer. If the amount of real memory available is smaller than the size of the transfer, the transfer can never complete without causing a page fault, and therefore will loop forever.

A solution to the second problem is to prohibit ECS transfers larger than the amount of real memory available to the job. This may or may not be practical, depending upon the design of the operating system. It also does nothing about the waste of repeated transfers.

An alternative that solves both problems is to modify the architecture so that interrupted ECS transfers can be restarted from the place at which they were stopped. This can be effected by adding a new field to the exchange package to contain the number of words already transfered. This field would have to be at least 18 bits wide to accomodate the maximum word count that can be specified for these instructions. When the ECS transfer is restarted, the contents of this field would be added to the beginning central memory and ECS addresses to locate the next word to be processed. These addresses would be available in A0 and X0, and the final word count for the transfer could be computed from the ECS transfer instruction, pointed to by the P register.

This change in the architecture has benificial effects aside from its connection with the virtual memory feature, because other types of exchange jumps can now occur during an ECS transfer. Thus, the responsiveness of the CPU part of the system is improved, and possible wasted memory traffic is eliminated.

2.11 PERIPHERAL PROCESSOR MEMORY ACCESSES

So far, it has been decided that both the CPU's and ECS will use virtual addresses for normal memory accesses. It is not immediately obvious whether the same policy should be followed for the third source of memory accesses--the peripheral processors.

The most compelling benefit of passing PP memory references through the paging mechanism would be software simplicity. The entire operating system could operate with virtual addresses, and all real addresses would be confined to the paging manager. The type of address translation software described in Section 2.8 would be unnecessary.

One of the tasks of the PP's is moving memory pages from real memory to secondary memory and vice versa, and the objection may be made that paging I/O cannot take place using virtual addresses. This objection is similar to the one made concerning ECS in the previous section, and as described there, the virtual memory mechanism is no obstacle.

On the whole, then, it appears advantagous for the PP's to use the virtual memory mechanism. The next step is to see whether any problems arise in adding this feature to the architecture.

Each PP in the system can potentially be working with a different address space (and thus a different page table) at the same time. Thus, each PP must have its own page table pointer register. Athough new PP instructions could be defined to load and read this register, it is more efficient to make use of the large number of I/O channel numbers that do not correspond to actual channels. Two of these "pseudo-channels" (to accomodate the 18 bit page table address--see Section 2.5) could be used to both load and read the page table pointer register with the existing PP I/O instructions. To simplify the PP software, the same channel numbers would be used by all PP's, but each PP could reference only its own page table pointer register.

As indicated in Section 2.9, there are undoubtely some situations in which the monitor, including the PP's, must run in real memory mode. Thus, there must be a hardware flag to control which mode the PP's use for central memory accesses. To provide maximum flexiblity, each PP can be given its own flag, and this flag can be set and cleared through the same pseudo-channels used to load the page table pointer registers.

2.11.1 Page Fault Mechanism

It would be possible, of course, to establish the convention that PP's could not reference pages that are not in real memory, and thus could not cause page faults. However, this would force the PP's to invoke the paging manager by software to get the pages they needed to reference. Thus, the paging manager would be invoked by two very different paths to perform the same function. This is bad software design, and should be avoided if possible.

If PP's are to be allowed to cause page faults, it must be decided how the hardware is to process these faults. One approach would make each PP wholly responsible for handling its own page faults. An error exit or indication could be added to each instruction that accessed central memory, which would be invoked if the referenced page was not available. Unfortunately, it is quite difficult to add error exits to the existing PP architecture, and any such scheme would require major changes to existing software.

Another alternative is to suspend any PP memory access that referred to a non-resident page, and convey the fault to another PP for processing. This scheme is essentially the same as that discussed in Section 2.7, and the same objections apply.

The third alternative seems to be by far the best. It is to handle page faults caused by PP memory accesses in essentially the same way as page faults from the CPU or ECS. When a PP referenced a non-resident page, the reference would be suspended, and the CPU exchange-jumped to the paging manager, who would then fetch the page from secondary memory in the usual way.

The only issue that remains to be decided concerns how a PP memory reference is to be suspended if it causes a page fault. One alternative is retry the reference, perhaps with a delay between tries, until the referenced page is resident in real memory. Of course, the CPU would be exchange-jumped only after the original try. This scheme has the advantage of requiring minimal changes in the architecture, but leads to two types of inefficiency. The repeated tries are inefficient, because they increase the workload of the paging hardware, and if there is a delay inserted in the retry cycle, there will be wasted time between when the page is made resident and the next retry.

The alternative requires the PP to wait for a signal from the CPU before retrying the memory access. This signal would be generated by a new instruction added to the CPU's instruction set. Since more than one page fault could be outstanding at any one time, this instruction would have to specify which PP is to be restarted. This in turn implies that the identity of the faulting PP is included in the information passed to the CPU by the hardware when the page fault occurs.

2.11.2 Alternate Address Space References

The PP's share with the CPU portion of the monitor a need to reference two different address spaces--that of the user, and that of the monitor. This requirement can be satisfied with an adaptation of the alternate address space instructions proposed for the CPU in Section 2.8. Two new instructions must be added to the PP instruction set to perform memory reads and writes to the alternate address space. To permit block transfers, these instructions operate similarly to the existing CRM and CWM instructions rather than the CRD and CWD instructions. Since the PP's only user-programmable register, the accumulator, is already used by these instructions, the page table of the alternate address space must be specified by another page table pointer register, loaded through I/O channels just as the primary one is.

2.12 ACCESS CONTROLS

A paging system provides the opportunity to control usage of memory by the system on a page-by-page basis, through access control flags in each page table entry. Access modes that have been supported in commercial systems include execute-only, data-only (no execute), and read-only. In systems that share pages between address spaces a write-only mode is also useful.

There are plenty of unused bits in the page table entries, so it is simple to add access control flags to the entries. The choice of CPU action after a violation of the access controls, and how these controls should be extended to ECS and the PP's, seems less clear.

When the CPU violates the access controls, it could either perform an error sequence similar to that executed after mode errors, or it could cause a page fault exchange jump. In either case, the hardware would have to provide some way for the monitor to determine the cause of the error. Since access violations would seem in most cases to be valid errors rather than expected events, the error sequence seems most appropriate. As part of the existing error sequence the CPU stores information about the error in memory, and there are sufficient unallocated bits in this word to accomodate the additional error information.

Since ECS transfers are completely controlled by the CPU program, it seems appropriate to apply access controls to ECS also, and to follow the CPU error sequence if the controls are violated. Access controls might pose a bit of a problem to the paging manager if it used ECS for swapping, since the manager should be able to read and write user pages even if they are marked write-only or execute only. However, the paging manager already has the ability to modify the page table entries, and it can clear these control bits before beginning the transfer. The benefits of applying access controls consistently seems to outweight this slight inconvienience.

It is significantly more difficult to apply access controls to memory accesses from PP's, because these processors do not have a hardware mechanism for aborting their programs. Because of this difficulty, it appears that PP's must excluded from memory access controls.

Execute-only mode usually implies read-only mode. In the 6000 architecture this results in difficulties for code that includes subroutine entry points, because the RJ subroutine call instruction stores its return pointer in the first word of the called subroutine. However, it is a simple matter to exempt this instruction from the read-only restriction.

CHAPTER THREE

VIRTUAL MEMORY IMPLEMENTATION

A machine architecture is not of much value if it cannot be implemented efficiently and economically. This chapter will describe one possible implementation of the virtual memory system described in Chapter Two. The description will be at the register-transfer level, and will ignore considerations related to technology and CPU design.

3.1 SYSTEM DESIGN

Figure 2 depicts the functional blocks of a 6000-architecture machine without the virtual memory capability. All of these blocks have been mentioned earlier except Central Memory Control, labeled as CMC in the diagram. CMC accepts memory references from the CPU's and PP's, and distributes them to the memory array.

The primary change which must be made in this system for the virtual memory feature is the insertion of the paging mechanism, which translates virtual addresses to real addresses, somewhere in the path between each processor and memory.

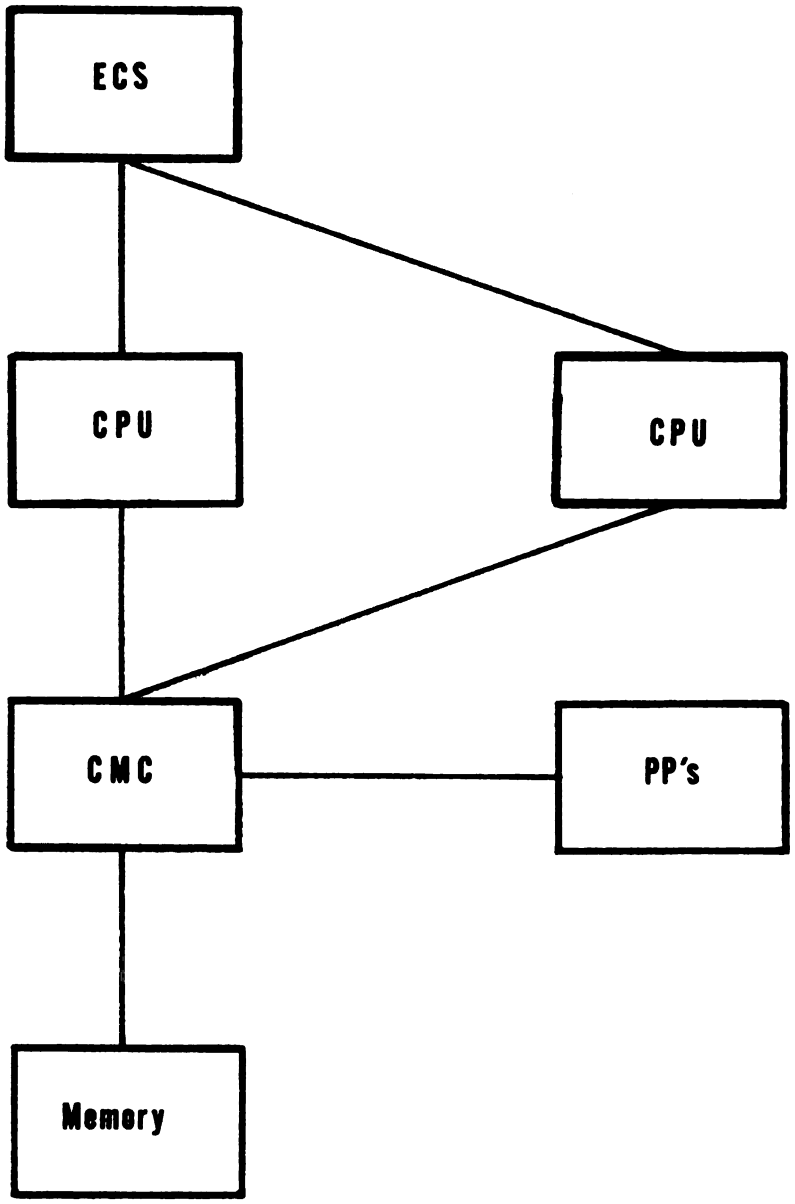

Figure 2: Unmodified System

If the paging manager (which will be abrieviated PM) is inserted between CMC and the memory, only one PM will be needed for the entire system. This arrangement has two primary difficulties. One is that all events in the processors that might change their address spaces (exchange jumps, loads of the PP page table pointer registers, etc.) must be transmitted through CMC to the PM. This increases the complexity of both the CMC and connections to it. The second difficulty is that a centralized PM will contain some components that must be duplicated for each processor in the system. Since different systems might be configured with different numbers of processors, the PM would either have to be customized for each system, or it would have to be designed to accomodate the largest possible system. Either alternative would increase the cost of the unit.

Instead, the arrangement (Figure 3) that will be considered here provides a PM for each CPU, and another PM that is shared by all PP's. Although it might be possible to make all PM's in the system identical, we shall develop different designs for the CPU and the PP PM's, to correspond to the differing characteristics of the two types of processors.

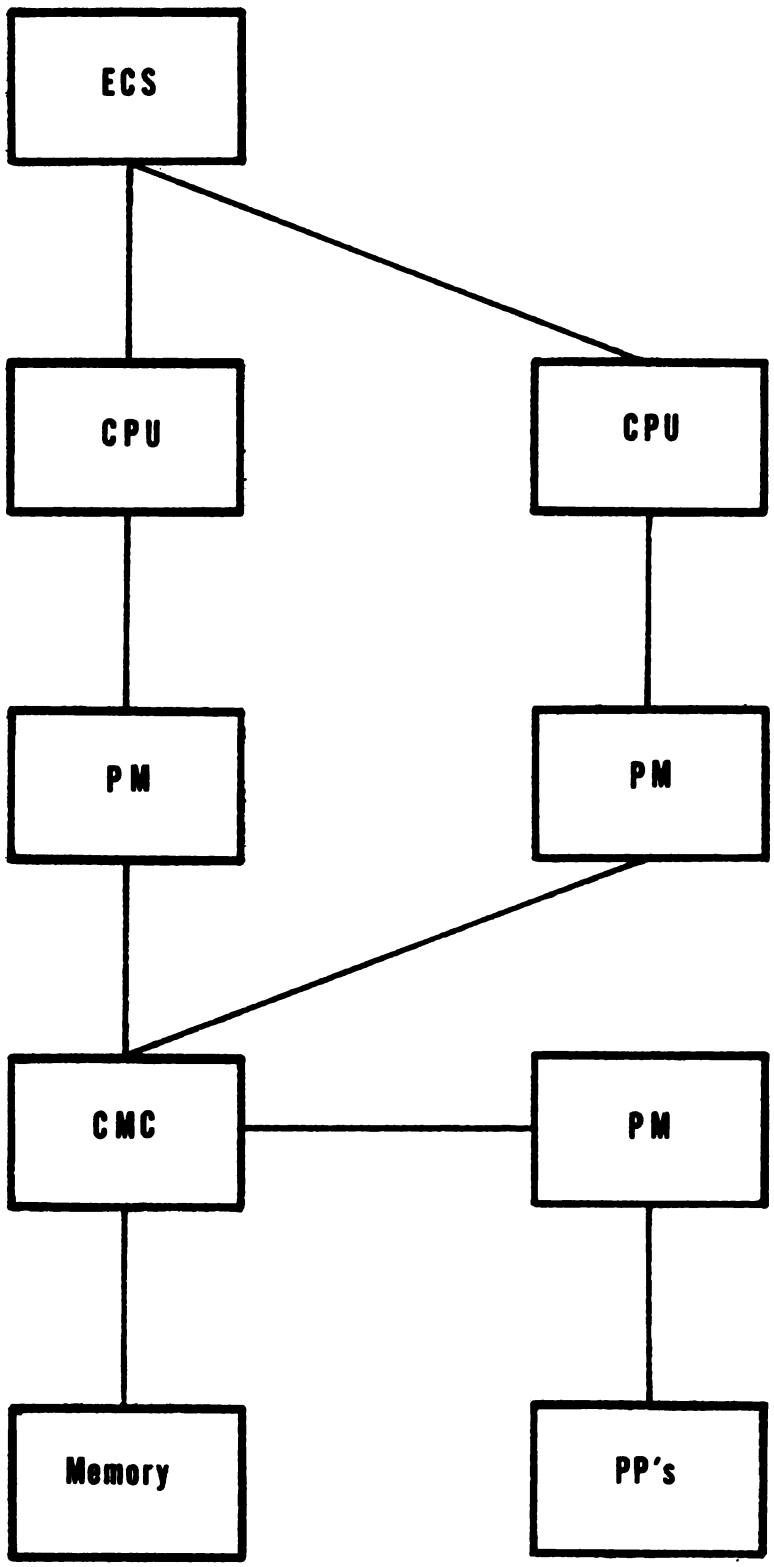

Figure 3: Modified System

3.2 CPU PAGING MECHANISM

A register-transfer-level diagram of the PM to be connected between CPU's and CMC is presented in Figure 4. Connections to the CPU enter at the top of the diagram, and connections to CMC are at the bottom. Only the major data paths are depicted; There are control signals that connect all units within the diagram and the CPU and CMC.

The PM is organized around a high-speed memory of 65 words by 60 bits, called the table memory. This memory is used to contain a copy of the current page table, loaded at exchange jump time. The 65 words of the table are sufficient to contain the page table, plus one additional word used for the alternate address space instructions (see below).

3.2.1 Exchange Jump Sequence

When the CPU begins an exchange jump, the paging mechanism must save the contents of the previous address space's page table, contained in the table memory. Each word of the table is sent to CMC with an address formed by adding the page table address supplied by the CPU to the contents of the counter. This counter is incremented for each word. Since the real addresses accepted by the CMC are 42

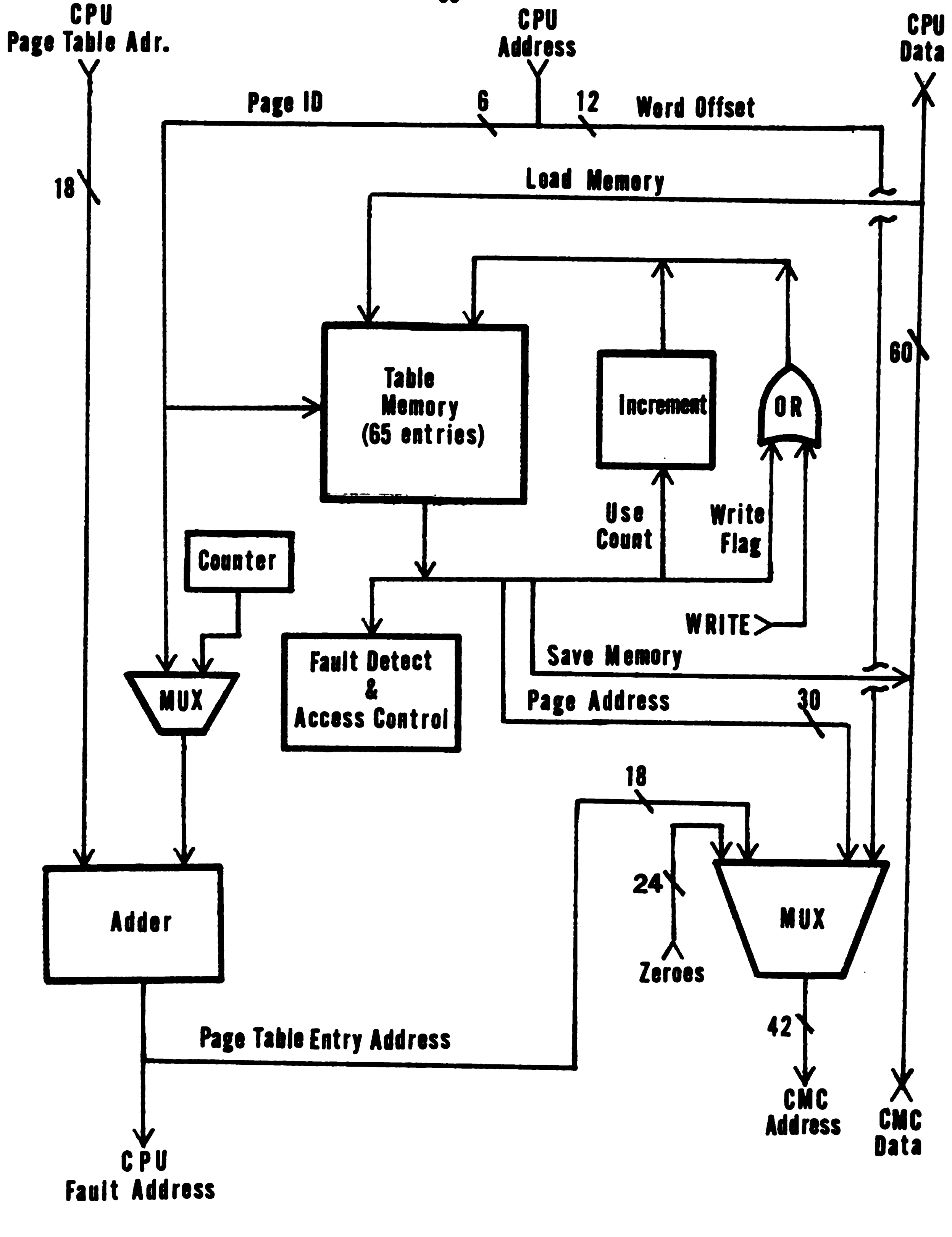

Figure 4: CPU Paging Mechanism

bits wide (30 bit page pointer plus 12 bit word offset), the output of the adder is concatenated with 24 zero bits.

After the old page table has been saved in memory, the CPU examines the paging mode bit of the new exchange package. If the bit is clear, the PM is disabled, and its part of the exchange jump is finished. If the paging mode bit is set, the PM reads the new page table into table memory. The addresses send to CMC for each read are formed in the way described in the last paragraph.

3.2.2 Normal Memory References

Normal memory references begin with the CPU supplying the virtual address on the address bus, and if this is a write operation, the data word on the data bus. The PM splits the virtual address into its word offset and page ID components, and the word offset is immediately sent to the CMC. The page ID simultaneously goes to several destinations.

The page ID is used to address one of the page table entries contained in the table memory. The access control bits from the entry are checked against the type of memory reference in progress. Any access violation is signaled to the CPU, and aborts the memory reference.

The fault bit from the page table entry is then checked. If it is set, a page fault is signaled to the CPU. An adder computes the address of the faulting page table entry from the page ID and the contents of the page table pointer register. This address is sent to the CPU over the utility data path, and will ultimately be passed to the paging manager software.

If none of these errors has been detected, the page address field from the page table entry is sent to CMC. This field, concatenated with the word offset, forms the real address of the referenced memory word.

Two fields of the table memory word addressed by the page ID are updated by every memory reference. The use count is increased by one, assuming the maximum use count has not already be reached. If the CPU memory reference was a write, the write flag in the entry is set. If the CPU memory reference was a read, the current value of the write flag is retained.

3.2.3 Alternate Address Space References

The CPU alternate address space instructions do not use the standard page table. Instead, during execution of these instructions the CPU places on the utility data path the address of the alternate page table, specified as an operand of the instruction. Address and write data information is transmitted from the CPU as for normal memory references.

The word offset and write data from the CPU are temporarily blocked from the CMC. Instead, a real address formed by adding the page table address and the page ID is sent to the CMC with a read request. This fetches the appropriate page table entry, which is returned on the data bus. The PM stores this entry into the 65th word of table memory. It then performs a normal memory reference cycle, except that the page ID is ignored, and the 65th table entry is always used.

Since the alternate address space instructions will probably be used only by the monitor, it is not absolutely necessary for the page table entry used by the alternate address space instructions to be updated. If at a later time it does become necessary, this update can be performed through a simple extension of the above mechanism.

3.3 PP PAGING MECHANISM

PP memory references will tend to be more localized within an address space than CPU references. In addition, there are far more PP's than CPU's in a system. For that reason, a different paging mechanism (Figure 5) is used to connect the PP's to CMC than is used by the CPU's.

The data received from the PP chassis for each memory reference include the number of the PP originating the reference, the contents of the page table pointer register (normal or alternate) to be used, and the virtual address and write data. The information sent to the CMC is identical to that generated by the CPU PM; i.e., real address and write data. There is also a path to carry the addresses of faulting page table entries to the CPU.

The PM is organized around memories for the page table pointers, page ID's and page table entries. These memories contain one entry for each PP in the PP chassis, and are addressed by the PP number. They are used to remember the last page table entry used by each PP.

A memory reference begins by comparing the current page table pointer and page ID aginst the contents of the corresponding memory entries. If either does not match, a new page table entry must be fetched. First, the entry

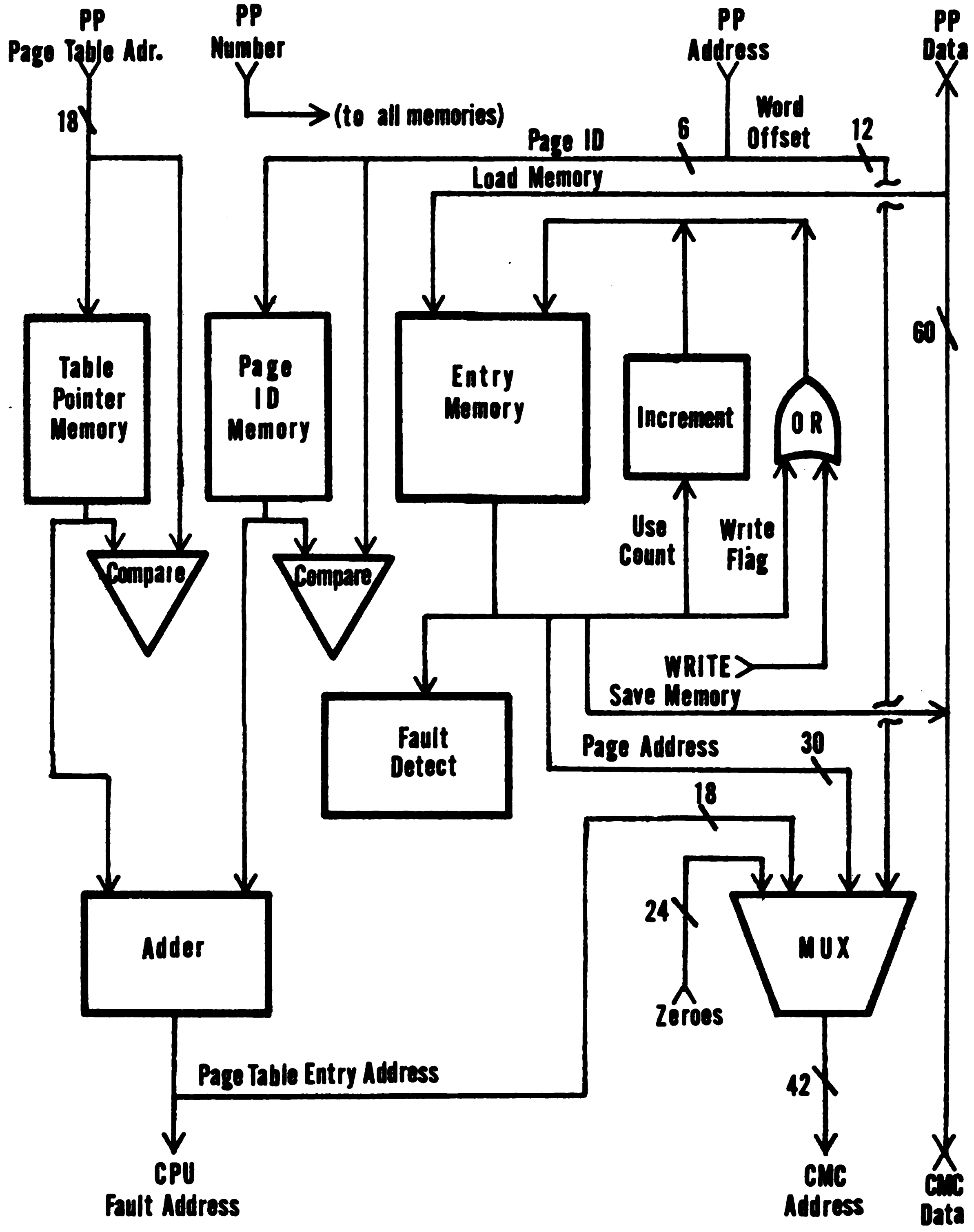

Figure 5: PP Paging Mechanism

currently contained in the memories is written back to real memory, using an address formed by adding the current contents of the page table pointer and the page ID registers. The new values for page table pointer and page ID are then written into their appropriate memory slots, and a read request sent to CMC using an address formed from the new contents of the two memories. The page table entry returned by CMC is placed in the PM memory.

Whether or not a new page table entry was fetched, the next step in the memory reference is to examine the page fault flag in the entry. If this bit is set, a page fault signal is sent to the CPU, accompanied by the address of the entry as computed by the adder. Otherwise, the real address for the memory reference is formed by concatenating the page address from the page table entry with the word offset from the virtual address. This address is sent with the write data to CMC to perform the memory access.

CHAPTER_FOUR

VIRTUAL_MACHINE_FEATURE

A virtual machine system is one in which each user is presented with an interface that in almost all aspects is identical to that of a dedicated hardware system. The resemblance between each user's virtual machine and a real machine is close enough that the user can utilize a standard operation system to control his virtual machine.

4.1 BENEFITS AND PENALTIES

Possible benefits of a virtual machine capability include:

1. Easing or eliminating the need for program conversion between various operating systems;
2. Allowing system software to be developed and tested without tying up an entire machine;
3. Facilitating achievment of security and privacy goals by providing a well-defined and easily-controllable interface to user jobs. [8, pp. 270-278]

4. Matching features of other manufacturer's machines.

Some of the possible disadvantages of such an extension include:

1. Additional hardware and software complexity. (Software support for the I/O portion of a virtual machine capability would be especially complex--see below.)

2. Significantly increased system overhead;

3. Possible constraints on the characteristics of I/O devices that could be attached to system (see section on virtual I/O);

4. Difficulty in supporting the system console of virtual machines.

Despite the similarity in names, it is not absolutely necessary for a system with the virtual machine capability described in this chapter to have the virtual memory feature described in Chapter Two. However, as noted in that chapter, a machine without the virtual memory feature can have a maximum of 262K of memory. This limits both the memory size that can be configured into a virtual machine, and the number of virtual machine memories that can be present in central memory at the same time. This in turn impacts system peformance by reducing the level of multiprogramming possible. In addition, a non-virtual memory system requires that the memory of each virtual machine be transfered in and out of central memory as a whole. This is quite inefficient when only a small portion of that memory may be accessed before the memory is swapped out again.

Despite these considerations, the discussion that follows will assume that neither the real machine or the virtual machines to be simulated on it will have virtual memory. This is done to avoid needless complexity, and so that both architectual extensions considered in this thesis will begin from the same base machine. The modifications needed for virtual memory are straight-forward.

The goal of virtual machine support features is to present the full architecture to each virtual machine, while protecting other virtual machines and the real machine's monitor from all users. For the CPU, the extensions consist primarily of memory relocation/protection and mode extensions. The PP's require load/dump facilities, virtual I/O support, and central memory relocation/protection.

4.2 CPU MODE EXTENSIONS

Since it is intended to be capable of running standard operating systems, the virtual CPU has both user and monitor modes. There is also a mode under which the software implementing the virtual machines can run, which will be called "real monitor mode." To allow the hardware to run unmodified operating systems in non-virtual mode, there is also a mode, called "real user mode," which operates identically to user mode in the current architecture. (See Figure 6)

There are also two varieties of exchange jumps, differing in purpose and in the contents of their exchange packages. A virtual exchange jump moves the machine between virtual user and virtual monitor modes. Since this exchange jump implements part of the virtual machine architecture, its exchange package is identical to that of the current architecture. A real exchange jump moves the CPU between virtual mode (either virtual user or virtual monitor) and real monitor mode. Here the exchange package may be extended to contain information relevant to the operation of the virtual machine support features. An exchange jump between real monitor and real user modes will be considered to be a special case of the real exchange jump in which the virtual machine information in the exchange package defaults to zero.

	Monitor	User
Real	Real Monitor Program	Compatable User Program
Virtual	User's Monitor Program	User's Problem Program

Figure 6: Mode Combinations

Relatively little state information need be added to the CPU to implement this system of modes. Without constraining actual implementations, a flip-flop within the CPU can be considered to indicate whether the processor is in user or monitor mode (either virtual or real). A virtual exchange jump complements the value of this flag. When a real exchange jump occurs, the old value of the flag is stored as part of the exchange package put into memory, and the flag is set according to the contents of the new exchange package. This flag is used to determine which form the CPU Central Exchange Jump instruction takes.

There is also a "virtual" flip-flop which is set when the CPU is in either virtual user or virtual monitor modes, and cleared when the CPU is in either of the other two modes. Its value is neither stored into nor set from an exchange package, but is complemented by a real exchange jump. This flag performs the CPU interlock function assigned to the monitor mode flag in the current architecture, i.e., only one CPU in a multiple-CPU configuration can have its virtual mode flag cleared simultaneously. The flag also controls operation of the memory relocation and protection circuitry.

4.3 CPU MEMORY RELOCATION AND PROTECTION

To allow efficient multi-programming of the real system, it must be possible for more than one virtual machine's memory to be present in real central memory at the same time. In addition, the contents of each one of these virtual memories must be protected from the actions of other virtual machines. This implies that some sort of additional relocation and protection mechanism must be available to the real machine. The mechanism already present in the architecture (implemented using the RA and FL registers) cannot be used because it must remain available for use by the virtual machines.

The most straight-forward solution to this problem consists of adding another RA and another FL register to the processor, and using these for all memory references from a given virtual machine. These registers, which will be referred to as the "super-RA" and "super-FL" registers, are used as follows: If the machine is in real mode, neither register is used. If the machine is in virtual monitor mode, the contents of the super-FL register are compared to each memory address. An out-of-bounds address results in a real exchange jump. If the address is valid, the contents of the super-RA register are added to the memory address and the result is sent to real central memory.

If the processor is in virtual user mode, the contents of the FL register are compared to each memory address. If the limit is exceeded, a virtual exchange jump results. Otherwise, the contents of the RA register are added to the address, and the result compared to the contents of the super-FL. Again, if this limit is exceeded, a real exchange jump is generated. If the limit is not exceeded, the contents of the super-FL register are added to the result of the previous addition, and the final sum passed on to real central memory as the reference address.

Since the contents of the super-FL and super-RA register are associated with a given virtual machine, it is natural to load and store them as part of the real exchange package. There are two 18 bit fields available in the exchange package for this purpose.

4.4 VIRTUAL PP'S

The 6000 architecture is rather unique in that a system contains multiple processors of two quite different types. This poses some difficult problems for the designer of a virtual machine feature for this architecture. A similar case is that of VM/370, whose channels can be considered to be separate processors. Even though these processors are quite limited in power, and execute out of the same memory as the CPU, their support in IBM's virtual machine system requires a significant amount of software. For example, virtual channel programs must be "translated" before they can be executed by real channels. [3, pg. 124]

The goal of permitting all present 6000-family operating systems to run without modification on a virtual machine puts rather stringent restrictions on the implementation of virtual PP's. The hardware cannot make any assumptions about the contents or behavior of the PP software, for example, nor significantly slow its execution.

The execution time requirement seems to rule out interpretation of PP software by the CPU or another PP. The only alternative remaining is direct execution of PP software by real PP's, in a manner similar to that of virtual CPU implementation. When a virtual machine is scheduled for execution on the real machine, some of the real PP's will be assigned to emulate the virtual machine's PP's, just as the real CPU will be assigned to emulate the virtual machine's CPU. Since one real PP is needed to emulate each virtual PP, and the virtual machine must have a full complement (10 or 20) of PP's, and since the real machine must have some PP's left over for its own use, the real machine must have a large number of PP's.

Many changes must be made to the PP architecture to support virtual mode execution on PP's. First, the system must have some way to load and store the contents of all virtual PP memories, their A and P registers, and the status of the channels, all without cooperation from the PP software. This mechanism will be used to swap a virtual machine in and out of the real machine, and simulate the deadstart sequence. The system must also have some way to intercept transactions on the virtual I/O channels and redirect and interpret them. Also needed is relocation/protection hardware, similar to that used on the CPU, to limit central memory references to this virtual machine's central memory.

Each real PP must contain a virtual flag whose meaning is similar to the CPU's virtual mode flag. When set, this flag indicates to the PP hardware that it should use the hardware features added to support virtual operation.

4.4.1 Virtual PP Load/Dump Instructions

The dump and load operations are used to prepare the PP portion of a virtual machine for execution on the real machine, and to terminate that execution. The dump operation stops execution of all PP's devoted tp the virtual machine (if they have not already been stopped--see below), and saves the state of those PP's in central memory. The load operation takes the information saved by a dump and reloads it into real PP's. This dump/load cycle must be invisible to the virtual PP software.

The destination of the information from a PP dump could be an I/O device or ECS. However, the most general design, and also the one most consistent with the remainder of the architecture, is for the transfer to reference a buffer in central memory. From there the data can be moved to an external I/O device if necessary.

An important question to be decided concerns how the dump operation is initiated. The CPU analog of the PP dump is the exchange jump, since it too saves part of the state of the virtual machine, and it therefore appears attractive to link PP dumps with CPU exchange jumps. If a dump operation was initated when a real exchange jump moved the CPU from virtual to real mode, the entire processor state of the virtual machine would be saved in a single operation.

Unfortunately, many needless dump/load cycles would occur if this approach was adopted. Many exchange jumps into the real monitor would not result in a change in the virtual machine, and would end with the same machine being loaded and execution resumed.

These wasted dump/load cycles would harm system performance in two ways. First, both the dump and the load involve the transfer of a sizable amount of information between the PP's and central memory, and thus would reduce the memory bandwidth available to the rest of the system. Secondly, while the dump operation could presumably be overlapped in time with execution of the CPU monitor program, it is doubtful that the load operation could be. Thus, the duration of the load, which due to the number of memory references required is not small, would have to be added to the amount of time required for a task switch. This would have a severe impact on the efficiency of the system software.

The alternative of letting the CPU monitor invoke the dump operation through execution of a new instruction seems to avoid these difficulties. The major disadvantage of this approach is that due to the delayed start of the dump operation (as compared with starting it with the monitor exchange jump), the monitor may have to wait for the dump to be completed before beginning the loading of a new virtual machine. However, the duration of this delay could probably be minimized through proper design of the monitor program.

If a virtual machine's PP's are not automatically dumped when a real CPU exchange jump occurs, that exchange jump leaves the virtual system with its PP's still executing, but its CPU effectively stalled. Most operating systems that might be run on the virtual machine could tolerate this state of affairs, but some, especially those that assume that the CPU immediately responds to a PP-initiated exchange jump, might fail. Since we cannot make any assumptions about the software to be run on the virtual machine, and since it is more consistent with the architecture to do so, we will halt virtual PP execution when a real exchange jump takes the CPU away from the virtual machine, and resume PP execution when the CPU is returned. This can easily be done by allowing virtual PP execution only when the CPU's virtual mode flag is set.

Also to be decided is how the hardware locates the area in memory to/from which the dump/load transfer is to be done. Since the operation is to be initiated by a CPU instruction, it seems natural for that instruction to specify the memory block as one of its operands.

The format of the load/dump block is not a major concern. The space occupied by one PP's record should probably be a integer number of central memory words to simplify the job of any program that must analyze these records. The data to be included for each PP are the PP memory (4096 12-bit bytes), the contents of the P register (12 bits) and the A register (18 bits). For each virtual machine, the state of the I/O channels (2 bits for active and full flags and 12 bits for contents per channel) and the contents of the Status and Control Register must be saved.

4.4.2 Central Memory Relocation/Protection

The address space seen by the virtual PP should be the same seen by the CPU in virtual mode. Therefore it is natural to pass PP memory references through the protection/relocation mechanism already used by the CPU. Since virtual PP execution occurs only when the real CPU is running in virtual mode, this presents no implementation difficulties.

4.4.3 Virtual I/O

A virtual machine requires I/O devices for input, output and mass storage. Since it is usually not practical to dedicate an entire real I/O device to a single virtual machine, these I/O devices must be virtual too. For example, the real operating system might split a real disk unit into many smaller virtual disks. Virtual unit-record I/O might be spooled onto the real system's mass storage.

(Note that there are occasions on which a virtual machine will operate dedicated real I/O devices. For example, a virtual machine might operate a graphic display unit or line concentrator just as it would if it was executing in real mode. Real devices dedicated to a virtual machine must not share the I/O channel with devices used by other virtual machines or the real monitor to avoid protection problems.)

There are rather difficult problems to be overcome in providing virtual I/O, because the PP software that will be running on the virtual machine is rather intimately involved in I/O operations. Transactions over the PP's I/O channels are at a relatively low level, and the PP software enforces strict timing constraints. To "translate" these transactions into a form that can be used by real I/O devices, the real system must intercept the transactions sent over the virtual channels, piece together the low-level transactions into higher-level operations, and finally convert these operations into a form that can be executed on the real machine. While the real system is performing these operations, it must simulate the timing that would be seen by the virtual PP if a real device was actually doing the I/O.

There are several possible approaches to satisfying these requirements. The one most similar to methods used by other architectures is to intercept I/O transactions sent over virtual channels, probably through a real PP external to the virtual machine's PP set. (The "interception" would resemble conventional inter-PP communication through channels.) The I/O transactions are then redirected by the "translator PP" to the real I/O device that is simulating the virtual one.

4.4.3.1 Translator PP's

In considering the translator PP method, the first matter to be decided concerns channel numbers. As noted above, the easiest way for the translator PP to intercept the I/O transactions of the virtual PP is for it to simply read them off the channel. Unfortunately, the two PP's cannot use the same number for this channel. The virtual PP's will always do I/O on channels numbered in the low end of the possible range, in order to be compatable with existing software. For the same reason, real PP's will have to use these channel numbers for real I/O to external devices. Thus, the low-numbered virtual channels must be mapped by the hardware to high numbers for the real PP's.

Only one addition must be made to the standard channel design for this new usage. That is some way for the translator PP to determine whether a full state on the channel is due to a function or a data output. This could be accomplished by using one of the unallocated PP instruction opcodes for a conditional jump flag that tests the function flag.

The dump operation discussed in Section 4.4.1 provides a way for the state of the PP portion of the virtual machine to be saved when the machine is removed from execution. Similarly, some way must be found to save the I/O state information maintained by the translator PP's. The most obvious method is for the real operating system to send a signal to the translator PP's (via one of the usual software mechanisms) when the virtual machine is to be suspended. The PP's would then write the information to part of the central memory block reserved for the status of this machine.

Unfortunately, this method is very inefficient. The principal location of the inefficiencies is the translator PP. The software in this processor would have to constantly check for an idle-down request from the operating system. This checking would slow down the time-critical code involved in simulating the virtual devices.

In order to avoid having to check for idle-down between every transaction with the virtual PP, the following idle-down scheme must be used. When the real operating system wants to remove a virtual machine from execution, it sets a flag notfying the translator PP's working on that machine of the idle-down. As each translator reaches a convenient stopping point, it checks this flag, writes out the status of the virtual I/O devices it is handling to the machine state block, and acknowledges the idle-down request. The CPU and PP's of the virtual machine are not stopped until the last translator PP has acknowledged the idle-down. In this way, the virtual PP's can complete all I/O operations in progress.

It would be attractive to do the saving of I/O status with hardware. Unfortunately, it is quite difficult for hardware to examine the memory of translator PP's and separate virtual machine state information, which must be saved and ultimately restored, from real machine state information, which need not be saved, and must not be restored.

Checking the idle-down flag will slow down the time-dependent code in the translator PP. In addition, this code has the usual interpretation characteristic of requiring many instruction executions to simulate a single virtual operation. In order to make such a system practical, it seems obvious that the processors performing as translator PP's must run at a much faster instruction rate than the processors performing as virtual PP's. This should be relatively easy to achieve, since the instruction timings on which existing software is based are the result of a long-outdated technology. The exact execution time ratio between the virtual PP's and the translator PP's is an implementation matter, but must probably be at least 10:1.

4.4.3.2 Intelligent I/O Controllers

The falling cost of processing power has resulted in an increasing level of intelligence being available to peripheral controllers. This trend offers an attractive alternative to the translator PP scheme presented above. In this proposal, virtual PP's communicate directly with actual I/O controllers, who take over the task of translating the I/O requests into real I/O operations. There is no need for support from real PP's during virtual machine execution.

The channels of the real machine are divided into two groups. The "fixed" channels are numbered and used identically to the current 6000 architecture, and are connected to all I/O devices. The "movable" channels are used only for virtual I/O, and need only be connected to the virtual PP's. Normally each movable channel is connected only to one piece of equipment--one that has a virtual I/O capability. The movable channels are so named because, under control of the real operating system, they can be mapped to any of the virtual channels. This mapping is established when the configuration of the virtual machine is first specified. More than one movable channel may be mapped to the same virtual channel.

When the virtual machine is readied for execution, the real operating system loads the mapping between virtual and movable channels into the hardware, and sends the saved I/O state of the virtual machine to the controllers over the fixed channels. As the virtual machine executes, the controllers translate its virtual I/O requests into real I/O operations. When the virtual machine is to be removed from execution, an idle-down request is sent to the controllers over the fixed channels. When the controllers finish the I/O operations in progress, they send the updated I/O state of the virtual machine back over the fixed channels, and prepare themselves for the next virtual machine.

This scheme greatly improves the efficiency of virtual I/O operations over the translator PP method. Unfortunately, there is a significant hardware cost involved. First, of course, there is the requirement for new device controllers, each containing a rather high level of processing power. In addition, there are many more channels to connect the mainframe with these controllers, and the mainframe must contain some sort of "switchboard" to connect the movable channels to the virtual PP channels.

A possible implementation of the real channels that would minimize the impact of these requirements is based on a fiber-optic communication path. This path would serve as all of the fixed and movable channels through a time-division-multiplexed scheme, and would be daisy-chained to all of the I/O devices. The switchboard mentioned above would be implemented by sending to each controller the time-slot to which it should listen. Each fixed channel and each virtual channel would be assigned a fixed slot in which to broadcast and listen.

4.5 VIRTUAL OPERATOR'S CONSOLE

CDC machines are rather unique in the industry in having a fully-interactive graphics terminal as their standard console device, rather than a simple teletypewriter. This presents problems when running virtual systems, each of which has its own virtual console, which must be accessible to the user of the system.

If the user's console has graphics capabilities, then it would probably be possible for the virtual system to translate the output of the PP driving the virtual console (DSD) into a form that could instead drive the user's terminal. Since this would be rather constraining, and would require a high-bandwidth transmission path between the system and each user, there would probably also have to be software in the virtual system to allow any type of terminal to examine simulated display screens and generate type-in's. These requirements would add a significant amount of overhead to the system. (Note that this software would have to be responsive to the user even while the virtual machine is swapped out.)

Another solution to this problem would require abandonment of the customary operation of a virtual system, in which each user has his own virtual machine. Instead, there would be only a few virtual machines running on the real system, each supporting a number of users. Since there are so few virtual machines, each could be assigned its own real console, or a single console could be switched among several machines. The primary advantages of a virtual system (no need for operating system conversion, and easy checkout of system software) would be retained.

4.6 VIRTUAL DEADSTART

Most of the implementation of deadstart on a virtual machine is straight-forward. The primary question that must be answered concerns how the virtual PP's are to be put into their deadstart state. One alternative would be to create a status block of a standard form which results in the desired behavior, and then loading this block into the PP's with the standard load operation. The alternative is a hardware operation that sets the PP's into the standard deadstart state, except that the channels read by the PP's are virtual ones, not real ones.

The tradeoffs are fairly obvious. The first scheme takes advantage of existing features of the architecture, but requires extra central memory and CPU time for creation of the state block. The second scheme requires a small amount of additional hardware, but avoids the disadvantages of the first. It would almost certainly be faster, too.

These trade-offs are so closely matched that the choice would have to be made on the basis of implementation considerations. Perhaps the architectural features implied by the second scheme could be made optional, with the real operating system adapting to the features of the machine on which it is executing.

CHAPTER FIVE

CONCLUSIONS

The virtual memory and virtual machine designs presented in the previous chapters were undertaken to evaluate the flexibility of the 6000 architecture. The results were mixed.

Design of the virtual memory feature was relatively straight-forward. The only unusual aspects of the design were the result of the parallel operation of the CPU's, and the presence of two different types of processors (CPU's and PP's) referencing the same virtual memory. However, neither of these unusual aspects made the resulting design unworkable. The trial implementation presented in Chapter Three confirmed this.

The design of the virtual machine feature was less successful. The CPU portion of the 6000 architecture adapted easily to the requirements of such a system, but the PP's presented major problems. Most of these problems arose from the simplicity of the I/O channels connected to the PP's, and the resulting hardware dependence of the PP software. Two approaches to the problem (translator PP's, and the use of intelligent I/O controllers) were examined, but both appeared to be unacceptably complex and expensive. Due to these problems, a trial implementation of the virtual machine feature was not attempted.

The use of programmable peripheral processors to control I/O devices was an effective design choice in the early 1960's when the 6000 architecture was first developed. However, the presence of these highly hardware-dependent processors now poses the most serious obstacle to expansion of the architecture to take advantage of state-of-the-art technology.

REFERENCES

1. Fischer, Patrick C., and Robert L. Probert. Storage Reorganization Techniques for Matrix Computation in a Paging Environment. Communications of the ACM, 22, 7 (July, 1979), pp. 405-415.

2. Control Data Corporation. Control Data STAR-100 Computer System Hardware Reference Manual. Publication Number 60256000, Revision 5, 1973.

3. Parmelee, R. P., T. I. Peterson, C. C. Tillman, and D. J. Hatfield. Virtual Storage and Virtual Machine Concepts. IBM Systems Journal, 11, 2, pp. 99-130, 1972.

4. Digital Equipment Corporation. PDP-11/45 Processor Handbook. 1971.

5. Roderick, David K. CPUMTR. Michigan State University Computer Laboratory 6000 SCOPE Memo 120, 1977.

6. Members of the Instructional Staff of the Control Data Institute. 6000 Series Introduction and Peripheral Processors Training Manual, Second Edition, Publication Number 60250400, 1968.

7. Organick, Elliott Irving. The Multics System. MIT Press, 1972.

8. Donovan, J. J. and S. E. Madnick. Virtual Machine Advantages in Security, Integrity, and Decision Support Systems. IBM Systems Journal, 15, 3, pp. 270-278, 1976.