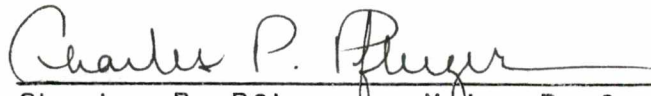
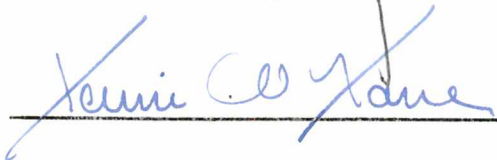


To the Graduate Council:

I am submitting herewith a thesis written by Beverly Cather Zygmunt entitled "Memory Management Facilities for a VAX Operating System Simulator." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.


Charles P. Pfleeger, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:


Vice Provost
and Dean of The Graduate School

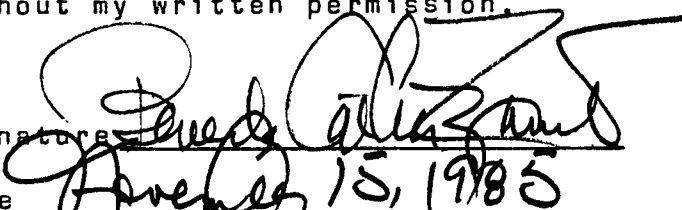
STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature

Date


December 15, 1985

MEMORY MANAGEMENT FACILITIES
FOR A
VAX OPERATING SYSTEM SIMULATOR

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Beverly Cather Zygmunt
December 1985

To Dr. Charles Pfleeger

ABSTRACT

This thesis describes SIMON, a three-part endeavor to provide an ASSIST-V-like operating system simulator for the VAX 11/780.

In this, the second phase of the project, virtual memory capabilities were implemented. In addition, the original addressing scheme was reworked to provide the user with the illusion of logical addresses.

The paper includes an overview of VAX architecture (with an emphasis on memory management), an explanation of SIMON's emulation of the paging hardware, and an example of a program that tests SIMON's memory mapping facilities.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
1.1 Why create a simulator?	1
1.2 How does SIMON work?	2
2. VAX ARCHITECTURE OVERVIEW	5
2.1 Data types, addressing modes, and instruction set	5
2.2 Registers	6
2.3 Stacks	7
2.4 Interrupts and exceptions	10
2.5 Memory management	12
2.6 Protection	18
3. SIMON OVERVIEW	21
3.1 Address relocation	21
3.2 Memory management implementation	23
3.3 Data movement and the extended operations	26
3.4 User constraints	27
3.5 Sample program	28
4. CAVEAT	43
LIST OF REFERENCES	44
VITA	46

LIST OF FIGURES

FIGURE	PAGE
2-1 VAX-11 call frame	10
2-2 Virtual address format	13
2-3 Page table entry format	14
2-4 System space address translation flowchart	16
2-5 Process space vs. system space address translation	17
2-6 Sample Execution	29

CHAPTER 1

INTRODUCTION

1.1 Why create a simulator?

SIMON (SIMulator for a Multi-user Operating system eNvironment) is hardware-emulating software that allows for the testing of operating systems on the VAX 11/780. The concept of such an instructional tool is not new, and other simulators have been used in academic environments for years.

An operating system is a large and complex program that "runs" the computer. In a multi-programming environment like that of the VAX, the operating system manages concurrently-running processes by allocating resources, scheduling tasks, interacting with the I/O devices, and so on. Obviously, a student-written operating system would be on a much smaller scale, and the dedication of a large machine for its testing would not be feasible. Still, there is the need to run and debug such programs, and a simulator is one method of so doing.

SIMON, written in C, accepts as input an operating system written in VAX assembly language. Acting as the VAX hardware, SIMON "executes" the instructions, generates "interrupts", and allows the user all those privileges normally reserved for the operating system (the loading of

the internal registers, for instance). And, since SIMON is itself merely a program, basically no different than any other, there is no conflict between the operating system in residence, and that being tested.

1.2 How does SIMON work?

The creation of SIMON will comprise three separate phases, two of which have now been completed. The first of these was basically concerned with the mechanics of instruction translation and implementation, and the handling of timing, exceptions, and interrupts. The second instituted virtual memory; the third will provide simulated input/output operations. (The only I/O supported by the first phase was register dumps. The second project did not concern itself at all with this area.)

Unlike the simulator ASSIST-V (designed to duplicate the hardware of the IBM 360 family) SIMON does not include an assembler. Rather, it operates on an image file (an assembled and linked VAX MACRO-11 program). Included in this file are all of the user-program instructions and data.

As established in the first phase, SIMON reads this information into an array which represents memory. A pointer (known as the program counter, or PC) into the array is established. As would be expected, movement through the program is accomplished by adjusting the value of this pointer.

Since all of the fundamentals of SIMON are based on this concept of a memory-approximating array, the second project utilized this format when implementing the various I/O operations associated with paging. That is, memory (page frames) is still represented by the single array. A second array represents logical pages stored on disk. Data is moved back and forth between the two.

This, SIMON's second component, incorporates a facility for memory management. Virtual memory, including all the bookkeeping such a scheme entails, is an integral element of most mainframes, many minicomputers, and even some microcomputers.

SIMON has been designed so that it can be run with or without its memory management features. Used without this option, SIMON can still test many aspects of operating system organization. But when its virtual memory capabilities are utilized, SIMON provides experience in even more sophisticated aspects of operating system design, such as the selection and implementation of a page replacement algorithm.

Due primarily to addressing constraints, there has had to be some reinterpretation of the VAX hardware, as represented by SIMON, and the software, in terms of what is expected of the operating system. For example, the VAX 11/780 has over four billion addressable bytes. Operating system addresses fall in the range of approximately two

billion to three billion. The hardware, therefore, expects to see a binary "10" in the high-order two bits of any 32-bit virtual address that references the operating system (or any other segment of system space).

Naturally, the programs that will be tested on SIMON will be neither capable of generating nor so large as to require such addresses. In addition, due to the very structure of VAX assembly language programs, operating system addresses will generally be "lower" than those of the processes they control. In this and other instances that are mentioned further on, compromises were necessarily made between "what should be" and "what could be."

CHAPTER 2

VAX ARCHITECTURE OVERVIEW

2.1 Data types, addressing modes, and instruction set

The basic unit of memory on the VAX 11/780 is the eight-bit byte (although individual bits may be addressed). Multiple-byte blocks also exist: words, two bytes; longwords, four bytes; quadwords, eight bytes; and octawords, 16 bytes.

There are six primary data types. Most of these include variations, which result in a total of 14 different types. These are: integer (byte, word, longword, quadword, and octaword), floating point (F, 32 bits; D and G, both 64 bits; and H, 128 bits), packed decimal string, character string, variable-length bit field, numeric string, and queue.

Except for bit fields, which are referenced relative to some byte address, all of the data types are aligned on a byte boundary.

Characters, which are stored in their seven-digit ASCII representation, are prefixed with a zero bit to accommodate the eight-bit format.

The VAX supports a variety of addressing modes, which fall into two different categories: general mode and branch target addressing. Included in the first group are both

general register and program counter addressing. General register (any register but R15) addressing comprises literal mode, indexed, register, deferred, autodecrement, autoincrement, autoincrement deferred, displacement (byte, word, or longword), and displacement deferred (byte, word, or longword). If the register is R15 (the program counter) these modes are known by different names: immediate (autoincrement), absolute (autoincrement deferred), relative (displacement), and relative deferred (displacement deferred). Branch mode addressing is expressed as byte or word displacement relative to the program counter.

The VAX instruction set exhibits the same diversity seen in the data types and addressing modes, with up to 256 different operations supported. In most instances, one basic instruction will serve as the basis for multiple variations. For example, the add instruction is of the form `ADDwx`, where w can be byte, word, longword, packed, or any of the four floating point types. The number of operands (two or three) is given by x.

2.2 Registers

In native (as opposed to PDP-compatible) mode, more than 200 opcodes are supported by the VAX. Its 16 32-bit general registers can be used with these instructions for temporary storage or as accumulators, pointers, index, or base registers.

Some of these registers have been assigned specific functions. Register 15 is the program counter, while registers 14, 13, and 12 are known, respectively, as the stack pointer, frame pointer, and argument pointer (all explained in more detail later). Other registers have special uses at both the hardware and software level, returning such values as address and length counters, and function results.

In addition to these registers, which can be accessed by the user, there are internal registers whose contents can only be altered by the kernel of the operating system. Among the information that can be found in these privileged registers is the address of the system control block of the process currently in execution. These registers also maintain the addresses and lengths of the page tables, values necessary for memory management.

2.3 Stacks

One notable aspect of VAX-11 architecture is the use of stacks. When subroutine branch or jump instructions, or procedure calls, are executed, the program counter and general registers are saved by the hardware's "pushing" them onto a stack. "Popping" the stack permits the restoration of these values at the time of the return from the subroutine. Register 14 is one of the general-purpose registers that has been designated for special usage. It is

used by the processor as a stack pointer (SP) to the current stack. Stacks on the VAX grow downward (that is, towards lower memory) so that SP is decremented as new items are added to the stack and incremented as they are removed.

Besides being used in subroutine calls and returns, SP is also involved in the activities resulting from interrupts and exceptions. The hardware places such information as the PC and PSL (program status longword) on the appropriate stack (interrupt or kernel) before control is switched to the routines that handle these conditions. The saving of these values allows the operating system to resume execution at the point of interruption. SP serves a similar purpose in the nesting of recursive routines.

Internal registers point to the five stacks maintained by the processor. Which of these stacks will be in use is determined by the current access mode and interrupt stack bit (found in the PSL). As the values of these bits change, the old contents of SP are stored in, and a new value loaded from, the appropriate register.

There is only one interrupt stack; it serves the entire system. Each process, however, has a separate kernel, executive, supervisor, and user stack. (These modes are explained in the section on protection.)

Two other registers with special uses are R12 and R13, that are known as the argument pointer (AP) and frame pointer (FP). These registers are mainly associated with

procedure calls, and the concomitant need for passing arguments to the procedures.

A VAX argument list is an array of longwords, the first of which specifies the number of arguments to follow. Either the addresses of the arguments ("call by reference") or the actual arguments themselves ("call by value") may be used to pass information to a procedure. The two VAX instructions that implement these methods are known, respectively, as CALLG ("general") and CALLS ("stack").

Before control transfers to the called procedure, AP is given the address of the argument list. Execution of a call instruction causes a "call frame" to be created on the user stack. The initial components of this structure are the registers which are to be saved, as denoted by the entry mask. (Besides specifying which registers to store, the entry mask, located just before the procedure's instruction, also contains the overflow-enable bits.) The registers are followed on the stack by the argument pointer, frame pointer, program counter, program status word, entry mask, and a longword associated with condition-handling. FP is then set to point to the top of the call frame. This process is illustrated in Figure 2-1.

An advantage of this multiplicity of stacks is one that will also be mentioned with respect to the page tables: the ability of the operating system to page in and out large amounts of data. In this way, the operating system can

reclaim much space that is used by the individual user tasks, a very important design feature of a machine capable of supporting dozens of simultaneous users.

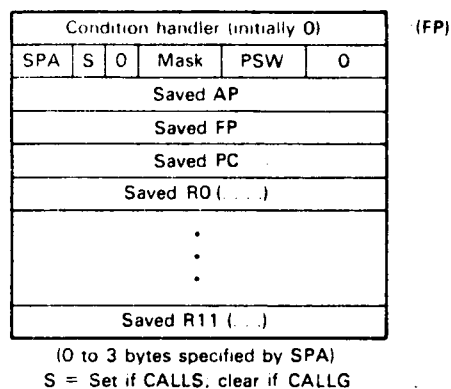


Figure 2-1. VAX-11 call frame

Source: Henry M. Levy and Richard H. Eckhouse, Jr., Computer Programming and Architecture: the VAX-11. Digital Equipment Corp., Bedford, MA, 1980, p.125.

While the kernel and interrupt stacks must always remain in main memory, the system is free to move the others back and forth from disk as needed. This is of particular importance in the case of user stacks, which may grow to be very large.

2.4 Interrupts and exceptions

Two types of events cause the CPU to break the normal flow of program execution and trap to designated operating system routines. These are interrupts and exceptions. As

has already been mentioned, interrupts are handled on a system-wide basis. Typical interrupts include those generated by hardware--I/O devices and timers, for example--as well as those that originate with software. An instance of the latter would be a request for a context switch.

Exceptions, on the other hand, are usually thought of as process-specific. Arithmetic and subscript range traps fall into this category. Some exceptions, however, are system-wide as, for example, machine checks.

The addresses of the routines that will handle these conditions are located at specific locations known as vectors. These addresses are all found within the system control block (SCB). The operating system notifies the hardware of the SCB's location by loading its address into an internal register, the system control block base register.

Interrupt-handling in the VAX-11 family is implemented using a priority scheme with 31 levels. Sixteen of these support hardware interrupts (levels 16-31) while levels 1-15 enable interrupts at the software level. Level zero signifies normal program execution.

An interrupt is generated if its priority is higher than that in the interrupt priority level (IPL) field of the PSL. If so, the PC and PSL are saved on the interrupt stack. The interrupt routine, which operates at kernel

level, accesses this stack using R14, the stack pointer.

The new PC is the address of the appropriate routine. At the conclusion of the routine, the PC and PSL are popped from the stack with the privileged instruction REI (return from exception or interrupt).

Program exceptions also trap to the operating system. In these cases, however, the PC and PSL are pushed onto the kernel stack. Other information pertaining to the cause of the exception may also be placed on the stack.

2.5 Memory management

With its 32-bit addresses, the VAX can permit references to over four billion bytes. The address space is divided in half, with addresses from 2B to 4B called system space, and those from 0 to 2B known collectively as process space. Located in system space is the system region, in which are found the operating system, system-wide data, and page tables, as well as routines for handling interrupts and system service requests. Half of system space (addresses 3B to 4B) is an unused sector, one that has been set aside by Digital for future growth.

Process space is divided up among the processes. Each of these individual allotments is composed of two segments, known as P0 and P1 (or, alternatively, program region and control region). P0 contains program images and data. P1 is an area reserved by the system for the user stack and

process-specific data. As a general rule, each process's P0 and P1 are not accessible to other processes. System space, on the other hand, is available to all (through system service calls and the change mode instructions, for example).

The basic memory management unit is the 512-byte page. Addressability is by means of a 32-bit vector, as shown in Figure 2-2. Bits 0 through 9 designate displacement within a page. Bits 31 and 32 specify the address space (P0, P1, or system). The remaining 21 bits denote a virtual page number and serve as a pointer into one of the three page tables (as specified by bits 31 and 32).

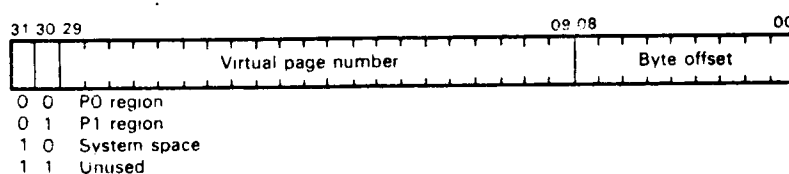


Figure 2-2. Virtual address format

Source: Henry M. Levy and Richard H. Eckhouse, Jr., Computer Programming and Architecture: the VAX-11. Digital Equipment Corp., Bedford, MA, 1980, p.269.

Page tables contain 32-bit (longword) entries. (See Figure 2-3.) Bit 31 is the status or valid bit. If this bit is set, then the page referenced is in physical memory and the low-order 21 bits specify a page frame number. A physical address is obtained by the concatenation of these

bits with the nine-bit displacement given in the virtual address. Bits 21-25 are reserved (for Digital Software), while bit 26 denotes "dirty" pages--those that have been modified. Bits 27-30 specify access rights to the page.

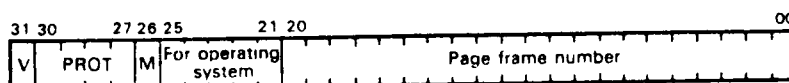


Figure 2-3. Page table entry format

Source: Henry M. Levy and Richard H. Eckhouse, Jr., Computer Programming and Architecture: the VAX-11, Digital Equipment Corp., Bedford, MA, 1980, p.266.

If bit 31 is zero (i.e., not set) bits 0-20 may signify a disk sector identification number or a pointer to a shared page frame. Referring to a page that is on disk, rather than in main memory, will generate a page fault, causing control to transfer to an operating system routine that will write out a page to disk (if necessary), bring in the requested page, and update the appropriate page table.

Addresses of the page tables are maintained in page table base registers. The system page table register contains the physical address of its associated page table. The registers for the process page tables, however, contain virtual addresses. One obvious advantage of this scheme is

that only the system page table must be kept in memory. The per-process tables may be paged in and out as needed.

Associated with each of these three base registers is a length register, whose contents correspond to the number of entries per table. Taken in conjunction with the base registers, they delimit the page tables.

In the case of P1 space, which operates as a stack, addresses move from higher to lower values. The P1 base register denotes the lower bound of the region (as do all the base registers). Consequently, the value in the P1 length register specifies the number of invalid page table entries.

System address translations require at most one look-up, as illustrated by the flowchart seen in Figure 2-4. The virtual page number is added to the base address of the system page table, yielding the physical address of the page table entry (PTE).

Both P0 and P1 addresses can require two look-ups. (See Figure 2-5.) This is due to the fact that their page tables are referenced by virtual addresses. Therefore, for addresses not found in the translation buffer (described below) an initial look-up must occur for the hardware to determine the physical address of the process page table. Once this has been established, the translation process continues as for a system address. Also a part of the address translation mechanism is faster memory known as the

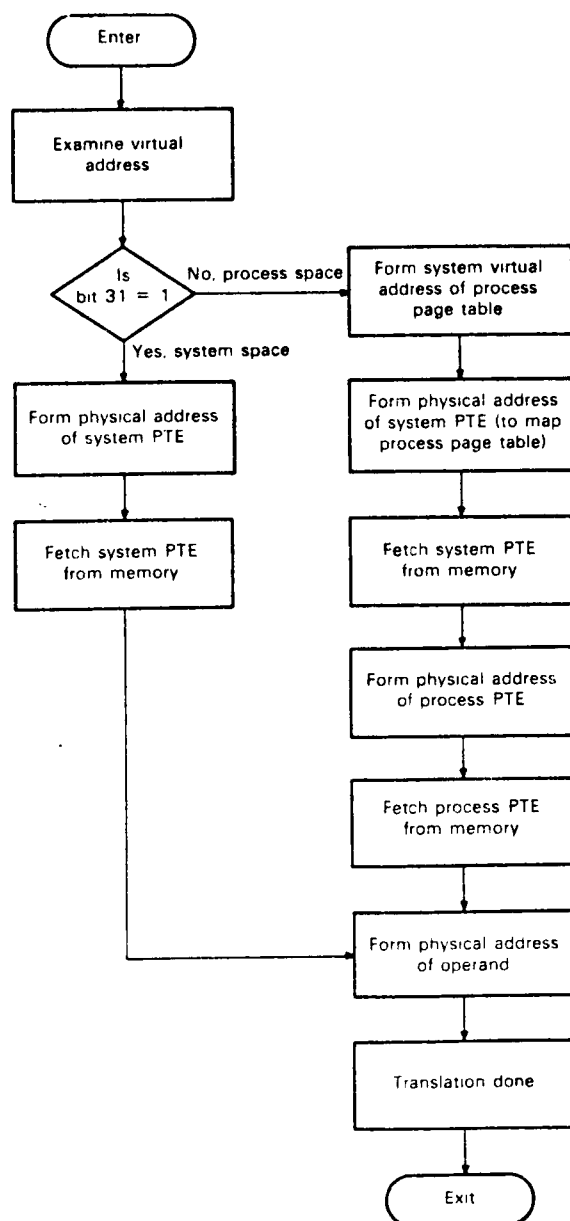


Figure 2-4. System space address translation flowchart

Source: Henry M. Levy and Richard H. Eckhouse, Jr., Computer Programming and Architecture: the VAX-11, Digital Equipment Corp., Bedford, MA, 1980, p. 270.

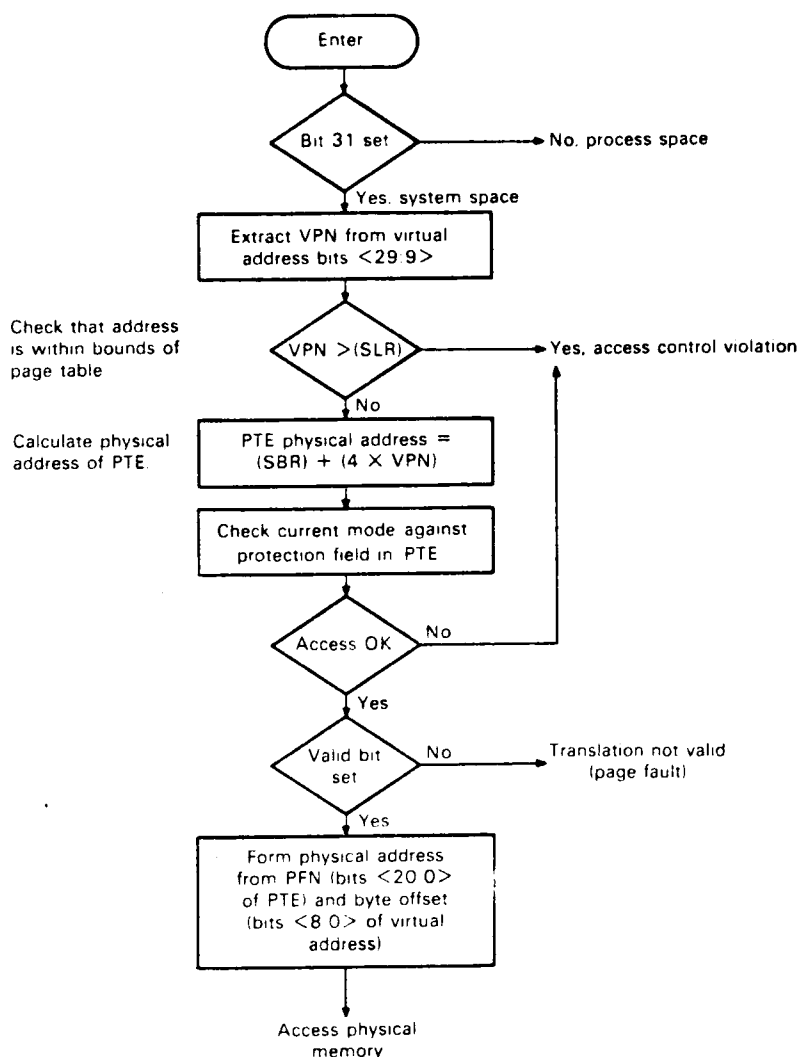


Figure 2-5. Process space vs. system space address translation

Source: Henry M. Levy and Richard H. Eckhouse, Jr., Computer Programming and Architecture: the VAX-11, Digital Equipment Corp., Bedford, MA, 1980, p. 274.

translation buffer. Kept in this area, in order to provide quicker memory access, are the 128 most recently used virtual address translations, 64 each for system and process space.

2.6 Protection

Protection on the VAX is provided at the page level, and is a function of the current access mode. Associated with each page, and located in the corresponding PTE, are four protection bits. These bits specify either no access, read only, or read and write access for each of the four modes: kernel, executive, supervisor, and user. All pages that can be accessed by any given mode are also accessible to those modes with more privilege.

Kernel mode is the most privileged mode, one that controls such system-wide activities as job scheduling, paging, interrupt handling, etc. It is the kernel that initiates jobs by loading the process context register.

The executive mode, one level below the kernel, is responsible for most of the system service calls, including those for the I/O routines known as RMS (record management services).

One level below the executive is the supervisor, associated with the command interpreter and other user services. The least privileged mode is that of the user. User level is the current mode during the running of user

programs and in the accessing of such facilities as compilers and system utilities.

Processes obtain operating system services by moving temporarily to a more privileged mode. This can be accomplished by either a call to a more privileged routine or via the change mode instruction (which can only be used to move to a more privileged mode). In either case, such a request traps (causes an interrupt) to the operating system. If the system permits the change, the old stack pointer is stored and a new one activated. Pushed onto the stack of the new mode are the PSL and the PC. The REI instruction returns that process to its former mode.

Typically, such routines apply the probe instructions PROBER and PROBEW to insure that any memory location referenced was accessible to the previous mode. This method allows a process to operate with more privilege, yet prohibits it from reading and/or writing unauthorized areas.

Access violations occur when any mode attempts to access a page to which it has insufficient privilege. This can occur in one of two ways: if the page referenced lies outside the page table (i.e., the page number, which acts as an index into the page table, exceeds the length of the table) or if the protection bits of the PTE show less access than requested. Such a violation causes a trap to an operating system routine. Placed on the interrupt stack are the PSL, PC, the virtual address that caused the fault, and

the fault parameter word. Included in this latter structure are bits that signify whether or not the intended access was a write, if the fault occurred during a reference to the process (rather than system) page table, and if the violation was a length violation.

If an access violation and a page fault are both the result of an address reference, the former takes precedence. This prevents a process (or even the operating system itself) from generating the fault of a page which it does not own.

CHAPTER 3

SIMON OVERVIEW

As was mentioned in the introduction, SIMON's first phase was primarily concerned with the instruction translation process, timing, and the simulation of exceptions and interrupts. SIMON uses a one-dimensional array to approximate memory, and pointer variables to represent registers. SIMON "loads" the user program into "memory" and, beginning with the program entry point, proceeds to "execute" the user program by extracting opcodes and operands from the instruction stream. A timing factor associated with each instruction is used to duplicate the effects of an internal clock. Exceptions and interrupts are basically the result of flags which, when set, cause SIMON to reset the program counter to the address of specific operating system routines. Addresses are based on the memory array address.

3.1 Address relocation

The starting point in the writing of SIMON's memory management component was the revision of the original addressing scheme. In the first phase actual physical addresses, as generated by SIMON's C program, were visible to the user. That is, if the initial entry point were 100 bytes into the user program, and the address of the memory

array were 25,000, the PC at the start of execution would be 25,100. Available to the user ["user" throughout this chapter refers to the student operating system, not a user process] was a relocation factor, which was this first PC. If used as an adjustment value, it resulted in negative addresses "above" the entry point.

As it was desirable (for setting up the page tables and to more closely approximate the VAX) to have addresses begin at zero, a new relocation factor, one not visible to the user, was established. Its value is the memory array address.

Whether or not the user decides to use memory mapping, all the addresses to which this student system now has access are as just described. These values are not, of course, valid addresses that SIMON can use. Therefore, whenever a reference is made to an address, this relocation factor is added back for SIMON's use, and then deducted again before the address is given back to the user.

In the case of branch mode addressing, the original scheme was unaltered. That is because, as these addresses are not available to the user, there is no need to maintain the logical facade. Indeed, since register 15 is not accessible to the user, it never holds a relocated address. To maintain consistency, however, (PC - relocation factor) is pushed onto stacks during interrupts, printed in register dumps, etc. If this value is ever received by SIMON (as in

an REI), the relocation factor is added internally for SIMON's use, as is done with all addresses.

3.2 Memory management implementation

The enabling of memory mapping on the VAX is a function of the MAPEN (map enable) bit, defined by privileged register hex 56. This bit is turned on (i.e., given a value of one) when the system is booted. Normally, MAPEN is turned off (by placing a zero in the internal register) only during system testing. When MAPEN equals zero, the high-order address bits are ignored, and the virtual addresses map directly to physical addresses. In addition, no access checking occurs: each mode is allowed all access privileges.

When invoked by a user program, SIMON assumes that MAPEN is not set. As mentioned in the introduction, memory is simply an array. Until MAPEN is turned on, an access violation is generated if an address does not 1) represent one of the 16 general-purpose registers, 2) equal a given literal value (that specified in literal mode addressing), or 3) fall within the array bounds. The only two values placed on the (interrupt) stack by SIMON when such a violation occurs are the PC and PSL. Once memory mapping is enabled, all four expected values (PC, PSL, faulting virtual address, and fault parameter word) are pushed onto the stack when either of the two page fault violations occurs.

As has been described, the PC is an index into the memory array. As such, its value continually changes as operands are extracted. To enable recovery from an interrupt, a "previous PC" counter is maintained to reset the PC to the beginning of the instruction.

Before a user sets MAPEN, certain parameters (the locations and lengths of the page tables, among others) must be passed to SIMON via the privileged registers. While VAX hardware expects, in the case of the system page table, a physical address, SIMON accepts a relative one.

It is mandatory that, when a translation not valid fault occurs, the system page table, kernel stack, interrupt stack, and paging routine be resident in main memory. SIMON will initially bring in two pages (the first page of the user program and the page containing the entry point). The system page table and both stacks will usually be located on the former; a paging routine, rudimentary or otherwise, on the latter.

SIMON will place logical page zero in frame zero, and the page with the entry point in frame one. It is the responsibility of the user to set the valid bits and indicate these physical addresses in the page table entries.

As noted in the section on VAX architecture, all VAX addresses fall into one of three valid categories, depending on the values of bits 31 and 30. P0 addresses should begin with "00", P1 addresses with "01", and system addresses with

"10". Any other address falls into an area reserved by Digital for future expansion.

Unlike the VAX scheme, however, in which system addresses are larger than P1 addresses which, in turn, are larger than P0 addresses, SIMON delineates the regions as follows: system space, addresses zero to hex 1FFF; P1, hex 2000 to 3FFF; and P0, hex 4000 and larger. The user must, when taking advantage of the memory management facilities, generate these addresses which will represent the P0 and P1 regions.

This is one of the only times when it is safe to alter the location counter, and it must be done as illustrated. The reason for this rigid and rather contrived method is a problem associated with the image file. When one changes the value of this counter, especially by a "large" amount, the assembler often compresses the contents of the image file: that is, the contents of address hex 4000 might be located immediately after the bytes associated with address hex 1200 (assuming that is the last addresses in system space).

As will be mentioned more at the end of this chapter, there is no documentation available that would explain how this is done and, while it is a logical assumption that such information can be found in the image file, repeated attempts to discover it were unsuccessful. As the example shows, there was no difficulty with the alteration of the

location counter in either the macro or just prior to the entry point. But, as can be seen, only small address changes resulted. If problems are encountered in attempting to locate the entry point at the start of a new page, the user should remember that storage can occur anywhere in the program. Moving the process page tables, for example, to the end of the operating system may provide the space needed.

3.3 Data movement and the extended operations

Physical memory in SIMON (once memory mapping is enabled) comprises 10 page frames, each of 512 bytes. As mentioned in the introduction, these frames are merely a section of the memory array.

Disk memory is a second array, which initially holds a copy of the user program as it appears at the start of memory management. The disk memory array is updated with the extended operation XOPC 3. When this instruction is executed, register zero should hold the number of the page frame to be written out to secondary storage.

SIMON always reads and writes from disk in a sequential manner: that is, logical page three would be written to the fourth 512 bytes of the disk array. For this reason, the 21 bits of the page table entry that could specify a disk address are meaningless in this context. In a similar manner, XOPC 4 causes a copy of the page specified by

register zero to be copied into the frame given in register one.

If an invalid page frame number [PFN] is given in either of these two instructions, no data movement occurs. If, however, a non-existent PFN appears in a page table entry, an access violation occurs. (How this eventuality is handled by the VAX is not clear. It was decided to treat it as a fault so that recovery by the user program was possible).

The third extended operation to be introduced in this project. XOPC 0, is a dummy instruction that acts as a flag to SIMON when the P1 and P0 addresses are created.

3.4 User constraints

The only restrictions new to this phase result from SIMON's need to find specific information in the image file. The program name (that of the PSECT) must match the name of the VMS file. Other than that this name cannot be "MAIN", the only other requirement is that it must not be more than eight characters in length and should be unique. (Since SIMON will look for this name in the image file when searching for the program length, a program name consisting of a single letter would not be good choice.)

The other constraint is that already mentioned in the section on memory management: the necessity of generating addresses that will be recognized by SIMON as P0 and P1.

While the example shows only one label in P1, multiple labels are allowed. However, the user must ensure that, when XOPC 0 is given, the address at that point is hex 4000.

3.5 Sample program

In order to tie this information together, and provide a working example for the user interested in SIMON's memory management capabilities, a sample program is included (Figure 2-6). What follows is a step-by-step explanation of the code.

1. The XOPC macro was established in phase one. It must be included if the user wishes to make use of the extended operations (a requirement for memory management).
2. The first items in the storage section are the page tables. While the process page tables can be placed anywhere within the program, the system page table (as well as the kernel and interrupt stacks) must be located on one of the two pages SIMON brings in when MAPEN is set: page zero and the page containing the entry point (page three in this example). It is the responsibility of the user to set the valid bit (bit 31) and denote the physical location (bits 0-20) in the PTE of each of these pages.


```

00000600 05C8      42      . = 1536
C000      0600      43      .entry test, m<iv, dv>
0602      44      ; adjust location counter so entry point
0602      45      ; and page fault routine on same page
0602      46      ;*--- load privileged registers with values needed for paging -----*
0602      47      ;
0602      48      ;
0607      49      ;
060A      50      ;
0610      51      ;
0615      52      ;
0618      53      ;
0619      54      ;
061E      55      ;
0623      56      ;
0626      57      ;
0629      58      ;
062C      59      ;
062C      60      ;*--- load privileged registers with stack pointers -----*
062C      61      ;
062C      62      ;
0631      63      ;
0634      64      ;
0639      65      ;
063C      66      ;
0641      67      ;
0644      68      ;
0648      69      ;
064E      70      ;
0653      71      ;
0656      72      ;
0659      73      ;*--- load scb with interrupt vectors -----*
0659      74      ;
0659      75      ;
0659      76      ;
065E      77      ;
0661      78      ;
0669      79      ;

51      F9FA CF DE 0602      47      movl spt, r1
12      51 DA 0607      48      mtr r1, #-x12
51      04 00 060A      49      movl #4, r1
13      51 DA 0610      50      mtr r1, #-x13
F9FC CF DE 0610      51      movl p1pt, r1
10      51 DA 0615      52      mtr r1, #-x10
11      51 DA 0618      53      movl #15, r1
11      51 DA 0619      54      mtr r1, #-x11
F9F2 CF DE 061E      55      movl p0pt, r1
08      51 DA 0623      56      mtr r1, #-x8
51      01 00 0626      57      movl #1, r1
09      51 DA 0629      58      mtr r1, #-x9

51      FA17 CF DE 062C      59      movl ketk+47, r1
00      51 DA 062C      60      mtr r1, #-x00
FAAB CF DE 0634      64      movl esk+11, r1
01      51 DA 0639      65      mtr r1, #-x01
FAD3 CF DE 063C      66      movl esk+11, r1
02      51 DA 0641      67      mtr r1, #-x02
00003F0B'EF DE 0644      68      movl usk+11, r1
03      51 DA 0648      69      mtr r1, #-x03
FAP1 CF DE 064E      70      movl isk+11, r1
04      51 DA 0653      71      mtr r1, #-x04
5E      04 DB 0656      72      mtr #-x04, sp

51      F568 CF DE 0659      73      movl scb, r1
17      51 DA 0659      74      mtr r1, #-x17
000006A8'EF DE 0659      75      movl tnv, x24(r1)
00000740'EF DE 0669      76      movl access, -x20(r1)

24 A1      000006A8'EF DE 0659      77      ; load address of system control block
20 A1      00000740'EF DE 0669      78      ; translation not valid routine
                                79      ; access violation routine

```

Figure 2-6 (continued)

```

0671 80 ; *--- clear translation buffer, enable paging -----*
0671 81 ;
0671 82 ;
0671 83      movl    #0,r1
0671 84      mtpcr   r1,#x57      ;tbia (translation buffer invalidate all)
0674 85      movl    #1,r1
0678 86      mtpcr   r1,#x56      ;set mapen (map enable)
067E 87 ;
0685 88 ; *--- load chm interrupt vector (done here to generate an o/s page fault) ---*
0685 89 ;
0685 90      movl    scb,r2      ;load address of system control block
068A 91      movl    chg_mode,x44(r2) ;change mode (to executive) routine
0692 92 ;
0692 93 ; *--- prepare for user execution -----*
0692 94 ;
0692 95      movl    pcb,r1      ;address of user process control block
0697 96      mtpcr   r1,#x16      ;loaded into pcbb
069A 97      pushl   #-x03c000a0 ;push user pal onto (interrupt) stack
06A0 98      pushal   user      ;push user program address (pc) on stack
06A6 99      ret            ;begin user program--this will cause
06A7 100 ;                an access violation and, after this
06A7 101 ;                is processed, another page fault
06A7 102 ;
06A7 103 ;
06A7 104 ; *--- page fault routine -----*
06A7 105 ;
06A7 106      .align   long
06A8 107      popr    #-M<r6,r7> ;remove fpw and faulting address
06AC 108      xopc     2          ;save this address
06AF 109      movl    r7,r6      ;initialize counter
06B2 110      clrl   r5        ; for comparison to determine page #
06B4 111      movl    #512,r8
06B8 112
06B8 113      loop:
06C2 114          subl   #512,r7
06C4 115          incl   r5
06C7 116          cmpl   r7,r8
06C9 117          bgeq   loop
06CC 118          movl   r5,r3
06D0 119          cmpl   r6,#x2000
06D3 120          bgeq   p1
06D5 121          movl   spt,r6 ;yes
06D7 122          movl   #7,r1 ;no, bring in @ of system page table
06D9          brb    skip ;this page will go in frame 7
06DD

```

Figure 2-6 (continued)

```

00004000 8F 56 01 06DF 124 p1:      cmpl  r6,#'x4000      ;is the region p0?
125      bgequ p0      ;yes
126      movel  p1pt,r6 ;no, p1. get p1 page table @
127      subl2  #16,r5  ;determine page number relative to p1
128      subl3  r5,#15,r5 ;p1 a stack. recompute page number with
129      ;      respect to bottom of stack
130      movl   #8,r1    ;this page will go in frame 8
131      brb    skip
132      movel  p0pt,r6  ;address of p0 page table
133      subl2  #32,r5   ;page number relative to p0
134      ;
135      ;this page will go into frame 7, one that has been used before. a check is
136      ;made to see if page one (that currently in the frame) has been modified
137      ;
138      movel  spt,r7    ;address of system page table
139      addl2  #4,r7     ;displacement of page 1 within spt
140      movl   (r7),r8   ;show that this page has been modified
141      xopc   2
142      bitl   #'x04000000,r8 ;was this page modified?
143      bneq   no_write  ;no. no need to rewrite to disk
144      movl   #7,r0     ;yes. ready to write out page one
145      xopc   3
146      no_write: nop
147      movl   #7,r1     ;place this page in frame 7
148      ;
149      ; the low-order 21 bits, when referencing a page that is on disk, have no
150      ; meaning under SIMON. the use of 1's in those positions makes it easy to
151      ; spot pages that are not in memory. here they are zeroed out, preparatory
152      ; to receiving the page frame number.
153      ;
154      skip:  bicl2  #'x001fffff,(r6)[r5] ;set the page frame bits (r1 holds the
155      bicl2  r1,(r6)[r5] ;page frame number) and the valid bit
156      ;
157      bicl2  #'x800000000,(r6)[r5]
158      xopc   2
159      movl   r3,r0      ;load r0 with (logical) page number of
160      ;      page to be brought in
161      xopc   4
162      rel

```

Figure 2-6 (continued)

```

0740 163 ; *--- access violation routine -----*
0740 164 ;
0740 165 ;
0740 166 .align long
0740 167 access: popr #M<r6,r7> ;pop fpw and faulting address
0744 168 xopc 2
0747 169 ;
0747 170 ; rewrite the page table entry so user has access to this page
0747 171 ;
0747 172 clrl r5
0747 173 movl #-x201ffff,pOpt[r5]
0749 174 movl r7,r1 ;load r1 with the faulting address
0753 175 mtprr1,#x58 ;notify hardware of change in pte (tbis)
0756 176 rei
0750 177 ;
075E 178 ; *--- change mode routine -----*
075E 179 ;
075E 180 .align long
075E 181 chg_mode: movl (SP)+,r7 ;which routine is requested?
0760 182 cmpl #0,r7 ;this program really offers no routine,
0763 183 ; simply a mode change
0766 184 bneq error ;if anything but zero, it's a mistake
0768 185 rei ;return
0769 186 error: xopc 1 ;stop the program
076C 187 ;
076C 188 ; *--- system service procedure -----*
076C 189 ;
076C 190 ;
076C 191 ; this routine (which could be thought of as a mail facility) is one in which
076C 192 ; a request is made to have information sent from one location to another.
076C 193 ; the addresses are probed to insure that the caller (who now is operating at
076C 194 ; executive level) had access to them while at his former privilege.
076C 195 ;
076C 196 .entry send,"M<r5,r6,r7>
076C 197 chme #0 ;change mode to executive
076E 198 movl 4(AP),r5 ;remove source and destination
0770 199 movl 8(AP),r6 ; addresses from the stack
0774 200
0778 201 prober #0,#4,(r6) ;test access of source address
0778 202 movpel r7 ;move in pel to test zero condition
077C 203 xopc 2 ; code bit--set by probe
077E 204 bitb #-x04,r7 ; if not accessible
0781 205 beql no_send
0784 206

```

Figure 2-6 (continued)

```

66 04 00 0C 0778 201      prober #0,#4,(r6)      ;test access of source address
57 04 57 0C 077C 202      movpal r7      ;move in pal to test zero condition
57 04 93 07E 203      xopc 2      ; code bit--set by probe
18 13 0781 204      bitb #x04,r7      ; if not accessible
18 13 0784 205      beql no_send      ;
65 04 00 00 0786 206      probew #0,#4,(r5)      ;source ok. test access of destination
57 0C 078A 207      movpal r7      ;check zero bit again
57 04 93 078F 208      xopc 2      ;
57 04 0A 13 0792 209      bitb #x04,r7      ;bit in right-most byte
67 66 00 0794 210      beql no_send      ;if set, access not allowed--do not send
50 04 0797 211      movl (r6),(r7)      ;both accesses allowable. send info
50 04 0799 212      clrl r0      ;set condition code
03 11 079C 213      xopc 2      ;
50 01 00 079E 214      brb return      ;
07A1 215      no_send: movl #1,r0      ;set condition code
07A2 216      return: ret      ;
07A2 217      ;
07A2 218      ;--- generate p1 and p0 addresses -----
07A2 219      ;
00003F00 07A2 220      . = 16336
00004000 3F00 221      uatkl .blkl 12
4000 222      xopc 0
4003 223      ;
4003 224      ;--- user storage -----
4003 225      ;
4003 226      .align long
4004 227      numarg: .long 3
4008 228      num1: .long 15
400C 229      num2: .long 25
00004014 230      answer: .blkl 1
F1 AF 0F 4014 231      user: pushal num1
C680 CF DF 4017 232      pushal tnv
C74C CF 02 F8 401B 233      calle #2,send
DE AF FA 4020 234      xopc 2
57 E2 AF 00 402B 235      callg numarg,abs_val
402F 236      movl answer,r7
4032 237      xopc 2
4035 238      xopc 1
4035 239      ;
4035 240      ;--- subroutine abs_val -----
4035 241      ;
0010 4035 242      .entry abs_val,"M<r4>
54 04 AC 08 AC C3 4037 243      subl3 8(AP),4(AP),r4      ;subtract second value from first
06 18 403D 244      bgeq noteq      ;if the answer is not positive
54 08 AC 04 AC C3 403F 245      subl3 4(AP),8(AP),r4      ; subtract first from the second
0C AC 54 D0 4045 246      movl r4,12(AP)      ; and store the result
04 4049 247      ret
404A 248      .end test

```

Figure 2-6 (continued)

PAGE FAULT--TRANSLATION NOT VALID:
PC: 692

*** REGISTER DUMP ***
R0: cccccccc R1: 00000001 R2: 000001c8 R3: cccccccc R4: cccccccc
R5: cccccccc R6: 00000004 R7: 0000020c R8: cccccccc R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 0000003f
PC: 000006ac CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***
R0: cccccccc R1: 00000007 R2: 000001c8 R3: cccccccc R4: cccccccc
R5: 00000001 R6: 00000000 R7: 0000000c R8: 00000200 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 0000003f
PC: 00000736 CONDITION CODES: N = 1 Z = 0 V = 0 C = 1

ACCESS VIOLATION exception generated;
PC: 4014;

*** REGISTER DUMP ***
R0: 00000001 R1: 00000168 R2: 000001c8 R3: cccccccc R4: cccccccc
R5: 00000001 R6: 00000000 R7: 00004014 R8: 00000200 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 0000002f
PC: 00000744 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

PAGE FAULT--TRANSLATION NOT VALID:
PC: 4014

*** REGISTER DUMP ***
R0: 00000001 R1: 00004014 R2: 000001c8 R3: cccccccc R4: cccccccc
R5: 00000000 R6: 00000002 R7: 00004014 R8: 00000200 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 0000002f
PC: 000006ac CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

Figure 2-6 (continued)

```

R0: 00000001 R1: 00004014 R2: 000001c8 R3: 00000020 R4: cccccccc
R5: 00000000 R6: 00000014 R7: 00000004 R8: c4000007 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 0000002f
PC: 0000070c CONDITION CODES: N = 1 Z = 0 V = 0 C = 1

```

```

R0: 00000007 R1: 00000007 R2: 000001c8 R3: 00000020 R4: cccccccc
R5: 00000000 R6: 00000014 R7: 00000004 R8: c4000007 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 0000002f
PC: 00000736 CONDITION CODES: N = 1 Z = 0 V = 0 C = 1

```

PAGE FAULT--TRANSLATION NOT VALID:
PC: 4017

```

R0: 00000020 R1: 00000007 R2: 000001c8 R3: 00000020 R4: cccccccc
R5: 00000000 R6: 00000006 R7: 00003fd7 R8: c4000007 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 0000002f
PC: 000006ec CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

```

```

R0: 00000020 R1: 00000008 R2: 000001c8 R3: 0000001f R4: cccccccc
R5: 00000000 R6: 00000010 R7: 00000fd7 R8: 00000200 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 0000002f
PC: 00000736 CONDITION CODES: N = 1 Z = 0 V = 0 C = 0

```

Figure 2-6 (continued)

```

R0: 0000001f R1: 00000008 R2: 000001c8 R3: 0000001f R4: cccccccc
R5: 000006a8 R6: 00004008 R7: 00c00000 R8: 00000200 R9: cccccccc
R10: cccccccc R11: cccccccc AP: 00003fcf FP: 00003fac SP: 00003fac
PC: 0000077e CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***

R0: 0000001f R1: 00000008 R2: 000001c8 R3: 0000001f R4: cccccccc
R5: 000006a8 R6: 00004008 R7: 00c00004 R8: 00000200 R9: cccccccc
R10: cccccccc R11: cccccccc AP: 00003fcf FP: 00003fac SP: 00003fac
PC: 0000078c CONDITION CODES: N = 0 Z = 1 V = 0 C = 0

*** REGISTER DUMP ***

R0: 00000001 R1: 00000008 R2: 000001c8 R3: 0000001f R4: cccccccc
R5: 00000000 R6: 00000010 R7: 000001d7 R8: 00000200 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 00003fdb
PC: 00004020 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***

R0: 00000001 R1: 00000008 R2: 000001c8 R3: 0000001f R4: cccccccc
R5: 00000000 R6: 00000010 R7: 0000000a R8: 00000200 R9: cccccccc
R10: cccccccc R11: cccccccc AP: cccccccc FP: cccccccc SP: 00003fdb
PC: 0000402f CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

```

Figure 2-6 (continued)

3. The protection bits of the only PTE in the P0 page table specify no access for user mode. This has been done to cause an access violation when the process program begins. Storage for the stacks, process control block (PCB), and system control block (SCB) follow standard form.
4. Before memory mapping can be enabled, the locations and lengths of the page tables must be given to the hardware via the privileged registers. It should be noted that, in the case of P1 space (which operates as a stack) the value loaded into the length register specifies the number of invalid pages.
5. In the next section, the privileged registers are loaded with the addresses of the bottoms of the stacks (since they are empty at this point). As the operating system will use the REI (return from exception or interrupt) instruction to begin the process program, the current stack pointer is set to that of the interrupt stack.
6. The addresses of the two routines that handle page fault interrupts (access violation and translation not valid) must be given to SIMON before the setting of MAPEN.

7. Since the translation buffer is checked whenever an address is referenced, the buffer must be cleared prior to initiating memory management. With all requisite values loaded, and the buffer free of extraneous information, MAPEN can now be set.
8. The addresses in the storage section were maneuvered so that this offset into the SCB is on a page not in memory, causing a page fault. In actual practice, all of the address vectors could have been loaded at the same time.
9. The operating system is now ready to turn control over to the process program. This is done by loading the process control block base register and pushing the beginning address (PC) and PSL onto the interrupt stack. These two values will be removed by the REI instruction.
10. Placed on the interrupt stack (by the hardware) when a page fault occurs are four parameters. Two of them are removed here: the fault parameter word and faulting address. This leaves the PC and PSL at the time of the interrupt on the top of the stack. The loop determines the logical page number of the address causing the fault.

11. If the address lies in system space, the page will go into frame seven. If the address is in P1 (pages 16-31) the P1 page number is computed. So, as in this example, logical page 31 becomes P1 page 15. As can be seen, there is only one entry in the P1 page table. This anomaly can be explained by the fact, already mentioned, that P1 operates as a stack. The P1 page number is recomputed to obtain a value relative to the bottom (higher-address) end of P1. R1 is given the PFN where this page will go.
12. A similar computation is done if the address is in P0. As P0 begins on logical page 32, that amount is subtracted to give the P0-relative page number. While there are still seven free page frames, a swap is done here to illustrate the method. This page will go into frame seven, one that currently holds an operating system page. The displacement of page one (four bytes) into the system page table is obtained. The PTE is brought into R8 for examination, and the modify bit is tested. Since it has been set, the page is written out to disk with the XOPC 3 instruction. Note that R0 hold the PFN of the page to be written.

13. Next, the physical address (PFN) is placed into bits 0-20 of the PTE and the valid bit is set. XOPC 4 is issued, and the logical page specified in R0 is brought into the frame given in R1. The REI pops the PC (which points at the instruction that generated the fault) and PSL from the stack, returning control to the point at which it was interrupted.
14. As with the translation not valid routine, that for access violation begins by removing the fault parameter word and faulting address from the interrupt stack. Since the fault occurred because the process program has no rights to its page, the P0 PTE is modified to correct this. If the hardware were not notified of this change (via the TBIS instruction) the user program would go into a loop as, on return to the process program, the translation buffer would always find the original (no access) PTE.
15. The change mode routine simply changes the mode to executive. The value associated with this instruction (in this case, zero) acts as an index to various services that could be incorporated at this point.

16. This operating system subroutine is one that can be called by those operating in a less privileged mode (in this case, user). As this routine is used to transfer data, a check is made of the addresses placed on the stack by the call. Only if the caller had access rights to both of them in his previous mode is the movement permitted. A condition code of one is returned in R0 to show that the request was unsuccessful.
17. The location counter is adjusted here. It is crucial that XOPC 0 (which is a dummy instruction) be located at the first address in P0, hex 4000 (decimal 16384).
18. The process program begins with a call to an operating system routine. Two addresses are sent, one to which the process has no access rights. As can be seen in the register dumps, the zero bit of the PSL reflects this status. The second call is to a process subroutine that calculates the absolute value of the difference of its two arguments. The final register dump shows that this routine worked. XOPC 1 halts program execution.

CHAPTER 4

CAVEAT

A problem that became evident during the writing of SIMON's second phase concerns the image file. As already noted, SIMON "reads" this file, extracting from it all the data needed to run the user program.

When a new version [4.1] of the operating system was installed, it quickly became evident that the format of the image file had been altered. For example, whereas the image file length used to occupy byte 65 of the image file, it is now found in byte 69.

While it is unlikely that future upgrades to the system will be as extensive as this last one (which eliminated the inceptive PDP-11 code), there is still the possibility that modification of the image file could again occur.

It is very probable that there are system structures which contain all the requisite information. If it were possible to access these, there would be no need to rely on a strictly-structured image file. Since requests to Digital for such information have been unproductive, however, the original scheme must suffice.

LIST OF REFERENCES

LIST OF REFERENCES

- Baase, Sara. VAX-11 Assembly Language Programming. Prentice Hall, Inc., Englewood Cliffs, NJ, 1983.
- Bell, C. Gordon, J. Craig Mudge, and John E. McHamara. Computer Engineering. Digital Press, Bedford, MA, 1978.
- Levy, Henry M. and Richard Eckhouse. Computer Programming and Architecture: The VAX-11. Digital Press, Bedford, MA, 1980.
- Marlino, Phyllis J. "A VAX Simulator for Developing Operating Systems", Master's Thesis, The University of Tennessee, Knoxville, 1985.
- Pfleeger, C.P. and C.E. Hughes. "ASSIST-V: An Environment Simulator for IBM 360 Systems Software Development", IEEE Transactions on Software Engineering, November 1978, pp. 526-530.
- Pressman, Michael H. Assembly Language Programming for the VAX-11. Mayfield Publishing Co., Palo Alto, CA, 1985.
- Siewivrek, Daniel P., C. Gordon Bell, and Allen Newell. Computer Structures: Principles and Examples. McGraw-Hill, New York, NY, 1982.
- VAX Architecture Handbook. Digital Equipment Corp., Maynard, MA, 1981.
- VAX Hardware Handbook. Digital Equipment Corp., Maynard, MA, 1982.
- VAX Technical Survey. Digital Equipment Corp., Maynard, MA, 1982.

VITA

Beverly Cather Zygmunt was born on April 17, 1948, in Winchester, Virginia. She attended two years of grade school in that city, and then moved with her family in 1956 to Saint Petersburg, Florida.

Ms. Zygmunt completed her elementary and secondary education there, graduating from Boca Ciega High School in 1966. She entered the University of South Florida in Tampa that fall, and received a Bachelor of Arts degree in Education in June, 1970.

After teaching for three years in the Hillsborough County public school system, she lived for a year in Europe. Before moving to Madisonville, Tennessee, where she now resides, Ms. Zygmunt spent two years in the Peace Corps. She lectured in pure mathematics at Mombasa Polytechnic Institute in Mombasa, Kenya.

She entered the graduate program in Computer Science at the University of Tennessee in 1983, and held a teaching assistantship. Ms. Zygmunt was awarded her Master of Science degree on December 10, 1985. She is employed by the Computing and Telecommunications Division of Oak Ridge National Laboratory.