Simulation as an aid to software transferability

by AARON H. KONSTAM

Trinity University
San Antonio, Texas

and

RONALD G. REINHARD

The Woman's Shop
San Antonio, Texas

ABSTRACT

Transferring software to a new host environment is one of the major problems facing installations wishing to upgrade their computer systems. This study investigates the effectiveness of simulation of an old host environment on a new host machine as a partial solution to the software transferability problem. A simulated environment of a Singer System Ten minicomputer was developed to run on an Alpha Micro microcomputer. The results of the project demonstrate that a simulated environment can be effectively used as an aid in transferring computer operations to a new host machine. It was also found that this technique is particularly suitable when software on the host machine is so dependent on features of the hardware that automated software translation is not feasible. The current generation of microcomputers is shown to be more than adequate to support the simulated environment of a minicomputer-based system.
INTRODUCTION

At present, computing systems are becoming cheaper and smaller, but at the same time faster and more powerful. As older computer systems are rapidly becoming obsolete, many companies have begun to explore ways to transfer their current computing tasks to these newer microcomputer systems.

The major problem in making such a transfer is the difficulty and cost of software conversion.1,2,3 Another problem arises from the testing involved to determine whether the software runs correctly after conversion.4 As for cost, hardware may be getting cheaper, but the labor costs of software generation are steadily rising. One solution to these problems is found in the process of simulation or emulation of the old machine on the new machine. Once the simulation or emulation software is produced, the old software can run on the new machine. If the machines have very different architectures, the old software will probably run more slowly and less efficiently in the simulated environment; so in most cases the simulation provides only an interim solution while new software accomplishing the same tasks can be written for the new machine.

The use of simulated environments to aid software conversion to a new machine is not a new idea. It has been used successfully for at least two decades.5 What is new, however, is the use of microcomputer systems to support such an environment. This paper discusses the process of generating an environment simulating the operation of a Singer System Ten on an Alpha Micro computer system.

The Singer System Ten and the Alpha Micro represent two radically different approaches to hardware structure. Constructing the simulated environment, therefore, was a major task in software design. It was necessary to simulate not only the processor but also the rather peculiar environment in which the System Ten controls its peripherals.

Although the simulation software described in this paper specifically applies only to the two computing systems mentioned above, the problems encountered in simulating one computer system on another computer system are of general interest.

OVERVIEW OF THE SYSTEM TEN COMPUTER

The System Ten is a multitasking, multiprogramming computer capable of executing up to 20 independent programs concurrently. The computer hardware is in total control of the allocation of CPU time and system resources to each program. Each of these 20 jobs is executed in a memory partition of fixed size. Each job can transfer information in and out of the System Ten through a variety of Input/Output Controllers, can store and retrieve information from data files through a File Access Channel, and can access a common memory shared by all the jobs running in the system at that time.

All characters used by the System Ten for both data and instructions are represented by a six-bit subset of the ASCII character set. Character strings of up to 100 bytes can be manipulated by the processor in one instruction.

Numbers on the System Ten are stored in main memory in their ASCII representation. Numeric fields of one to 10 bytes can be manipulated by the processor in one instruction. When the processor is instructed to perform an arithmetic operation, it performs binary coded decimal (BCD) arithmetic on the numeric parts of the characters in each operand and leaves the result as a string of ASCII characters. The System Ten performs integer arithmetic, leaving scaling operations to the programmer.

Address fields are four bytes long. The numeric parts of the four bytes represent the addresses 0000 through 9999 in BCD. A 10,000-byte page address is formed by ASCII bit five in three of the four bytes, and ASCII bit seven in the rightmost character determines whether the address is in common or partition memory.

The instruction word is a fixed-length field 10 characters long and must always be located at a memory address that is divisible by 10. (See Figure 1.)

Main memory on the System Ten is divided into partitions that are byte-addressable. The number of partitions is determined by the number of Input/Output Controllers (IOC) physically resident in the system. Partitions from 1,000 to 80,000 bytes can be allocated by use of a hardware jumper in the IOC of each partition. At least 1,000 bytes of main memory are dedicated to a common memory that can be accessed by programs running in a partition. The first 300 positions of common memory are protected from general memory writes and are used by the hardware operating system to store status and program counters for each of the 20 partitions. Each partition contains three index registers at fixed memory locations; therefore, no special instructions exist for manipulating index registers. Memory addressing within a partition is relative to the beginning of that partition's memory; that is, the first memory position for each partition is location 0000.

The Input/Output Controllers handle all peripherals except disk and tape drives. Each device on the IOC is assigned a unique one-digit identifier. Device zero is normally a CRT used for program loading and execution. A device can only be accessed by a program residing in the partition associated with the IOC controlling that device.

The System Ten contains one File Access Channel (FAC) that can control up to four magnetic tape drives and up to 16
The Alpha Micro System is a multitasking, multiprogramming system based on the Motorola MC68000 microprocessor. The largest model will concurrently handle up to 60 users.

Although the Alpha Micro software and System Ten hardware approaches to multiprogramming differ considerably, they are conceptually similar. Both employ a fixed memory structure, set by system initialization on the Alpha Micro and by hardware jumpers on the System Ten. Both the System Memory on the Alpha Micro and the Common Memory on the System Ten are usable by all jobs running in the system as a means of sharing programs, data, and communication between jobs. The Alpha Micro and System Ten both service their jobs in a round-robin fashion, sharing the CPU time among jobs.

The project described in this paper utilizes an Alpha Micro AM-1042 to support the simulated environment. This system consists of the Alpha Micro 100/L processor, 512 kilobytes of main memory, one 32-megabyte Winchester-type disk drive, one videotape recorder/player for backup, two Ampex D81 CRTs, one Texas Instruments 810 printer, and the AMOS/L version of the Alpha Micro Operating System.

SIMULATION OF THE SYSTEM TEN INSTRUCTION SET

The System Ten has five addressing modes: Absolute, Indexed, Indirect, Indirect/Indexed, and Immediate (ADD ADDRESS Instruction only).

Five instructions are used for the manipulation of numeric fields. These five—ADD, SUBTRACT, MULTIPLY, DIVIDE, and FORM NUMERIC FIELD—compose what are known as the two-length instructions. In this form the A and B operand may each be from one to 10 digits long and may use any addressing mode except Immediate. The algebraic sign of the operands and results of these operations is indicated by bit seven of the rightmost digit of the operand or result. If the bit is on, the number is negative; otherwise the number is positive. These instructions perform their functions from right to left in the operand fields. The simulator performs these operations in a manner similar to the System Ten, with one exception. The System Ten executes on the operands in place, whereas the simulator copies the operands into a work area, performs the operation, and copies the result back into the target operand field. The instructions perform the following operations:

1. ADD—Algebraically adds the numeric contents of the A operand to the numeric contents of the B operand.
2. SUBTRACT—Algebraically subtracts the numeric contents of the A operand from the numeric contents of the B operand.
3. MULTIPLY—Calculates the algebraic product of the A and B operands and develops the result in the B operand.
4. DIVIDE—Calculates the algebraic quotient of the A and B operands and develops the quotient in the rightmost positions of the B operand and the remainder in the leftmost positions of the B operand.
5. FORM NUMERIC FIELD—Moves the numeric portions of the A operand to the B operand, leaving the B operand to contain the numeric value in the form used by the arithmetic instructions.

The System Ten supports five instructions used for the manipulation of character fields. These instructions—MOVE CHARACTER, MOVE NUMERIC, EXCHANGE, COMPARE, and EDIT—make up the System Ten one-length instructions. In this form, the A and B operands are of the same length, from one to 100 characters long, and may be any addressing mode except Immediate. These instructions perform their functions from left to right in the operand fields.
The simulator performs these operations in the same manner as the System Ten:

1. MOVE CHARACTER—Transfers the characters in the A operand to the corresponding characters of the B operand.
2. MOVE NUMERIC—Transfers the numeric bits of the characters in the A operand to the numeric bits of the characters in the B operand.
3. EXCHANGE—Interchanges the characters in the A operand with the corresponding characters in the B operand.
4. COMPARE—Compares the character in the A operand with the corresponding characters in the B operand and sets the condition code register to reflect the relationship between the operands.
5. EDIT—Moves numeric parts in the A operand to the B operand. The B operand identifies a control field, which contains characters to control the suppression of leading zeros; insertion of check protection characters; and insertion of punctuation characters such as commas, hyphens, decimal points, and a sign indicator.

The System Ten has two instructions that are used for manipulating the four-byte memory address fields. These instructions are simulated in the same manner as performed on the System Ten:

1. ADD ADDRESS— Adds the address bits of the A operand to the address bits of the B operand.
2. MOVE ADDRESS—Moves the address bits of the A operand to the address bits of the B operand.

The System Ten BRANCH instruction allows the program to change the path of execution. The BRANCH may be unconditional, dependent on the result of a previous instruction, dependent on a request for service from an input device, or a link to a subroutine.

The System Ten has one instruction, SET MODE, which may be used to inhibit partition switching, to perform a system reset, and to allow changes to the protected area of common. This permits partition zero to change the program counters and status of other partitions, thus forcing another partition to load and begin execution of a program.

Two instructions are used by the System Ten to transfer data between memory and peripherals or storage devices: the READ and WRITE instructions. The System Ten supports two types of input and output devices: Terminal devices (CRTs and Printers) and File devices (Disk and Tape drives).

DESIGN OF THE SIMULATOR

Prior to developing the simulation of the System Ten on the Alpha Micro, other System Ten replacement alternatives were explored. The replacements generally entailed redesign of System Ten to employ state-of-the-art technology. These approaches allow the user to continue using the current software with conversion efforts ranging from no conversion at all to a moderate conversion that uses a totally different disk management facility. By maintaining the System Ten operating environment, a user would not be able to use the extensive software base existing on other machines, such as comprehensive word processing, spreadsheet programs, program development aids, and application packages.

Consideration was also given to developing a software translation program at the source program level. The only common language ever successfully developed on the System Ten was RPGII, and its use was relatively small. Therefore, most System Ten software, both system and application, has been developed in System Ten assembly language. Given the System Ten method for handling arithmetic operations and the programming tricks often required to write working software, the possibility of writing an effective software translation program was practically nil.

Obviously, the primary goal in any simulation project of this nature is to imitate the operations of the source computer on the target computer as efficiently as possible. To accomplish this goal, the System Ten simulator was designed to follow closely the original System Ten hardware implementation. A secondary goal was to allow the concurrent operation of simulation and native modes on the Alpha Micro. This is an important consideration when the simulation process is approached as a conversion aid rather than as a permanent solution to the problems of changing computer hardware. This goal was attained by designing the simulator to observe AMOS/L system conventions and therefore to maintain the integrity of the AMOS/L operating environment.

Although the System Ten approach to processing information and performing multitasking is radically different from most of the current computers available, it still has several of the same hardware functions, which must be simulated: Memory Access, Fetch and Instruction Decode, and Instruction Execution.

Memory Access is normally the easiest one of these hardware functions to simulate. Within the simulator program, memory exists as a large variable, and access to memory is accomplished by indexing from the beginning of the variable. It is important to maintain starting and ending memory addresses to trap memory access violations (e.g., attempts to access beyond the end of memory). In simulating the System Ten, memory access also involves converting the four-byte BCD memory address to a binary address for accessing the memory variable.

The simulated Fetch and Instruction Decode routine performs the same functions as the equivalent hardware operation: retrieving the next instruction and maintaining the program counter. As an instruction is fetched, parts of the instruction are decoded and placed into registers and variables for use during instruction execution. Subsequent to retrieving the entire instruction, the remaining functions of the decode routine are performed, including operation code validation, address modification (e.g., indexing, indirect addressing), conversion of instruction address to actual simulated memory address, memory address violation checks, operand length validations, and passing program control to the proper instruction execution routine. Since the System Ten has a fixed instruction word of 10 bytes for its 16 instructions, the Fetch cycle was relatively easy to simulate. In attempting to simulate...
other computers, which may have variable-length instruction words, it would be necessary to fetch the operand code, determine operand length by using an operation table or operation length algorithm, and fetch the remaining portion of the instruction. On the other hand, the Decode cycle was more complex to simulate because it performs the memory accessing verifications and conversions described above for both instruction operands. In the case of indirect and indexed operands, as many as five address conversions may be required for a single operand. When simulating computers using binary memory addressing, the time spent in address conversion would be significantly reduced.

For Instruction Execution, with the exception of the ADD and SUBTRACT instructions, each simulated System Ten instruction has its own routine to perform the desired operation. Since the ADD and SUBTRACT executions are so similar, there is only one routine to perform these operations. Each routine has the effective Alpha Micro address passed to it from the Decode function and is responsible for manipulating the data as required and for setting the proper condition code on the basis of the result of the operation. Upon completion, each Instruction Execution routine returns to the Fetch routine.

In addition to the three hardware functions described above, other aspects of the source computer design will influence the simulator design. The System Ten hardware, for example, is in control of partition management and time allocation and therefore requires a partition Switch Cycle. This cycle is generally not present in the operation of other CPUs, but in this case it must be simulated so that the multitasking feature of the System Ten can be faithfully modeled. In addition to its other functions, the Fetch routine is responsible for maintaining simulated instruction times and determining when a partition switch is required. When a switch is required, the Fetch routine releases control to the Switch routine, which is responsible for updating certain portions of System Ten status, finding the next partition in sequence to receive control, retrieving that partition program counter, and releasing control back to the Fetch routine to begin program execution for the new partition.

The System Ten instructions that manipulate data work strictly with memory resident operands. There are no registers and therefore no register instructions. It should be noted that the System Ten allows indexing and therefore does have index registers; however, the index registers reside at fixed locations in partition memory and are treated as memory operands. In designing simulators for computers that have true index registers, the simulation of registers and register operations would have to be addressed. Perhaps the most difficult task of simulator development resides in imitating input and output, especially terminal input and output. Although it is possible to attach a variety of terminal devices to the System Ten, the simulator developed in this project allowed only CRT input and output and printer output. The System Ten interfaces very intimately with its peripherals and creates situations very difficult to simulate without totally simulating the peripheral in software. In effect, the latter was necessary to ensure CRT input and output compatibility with existing System Ten application software.

Since the simulation mode in this project was designed to coexist with native operations on the new system, it was important to prevent collisions in accessing system resources. Since the Alpha Micro uses a print spooler for almost all printing tasks, the simulator was designed to channel all printer output to a spool file and to recognize a special (unused) form of the System Ten WRITE instruction to allow the print file to be closed and submitted to the spooler for actual printing. This also allows a printer to be available for each simulated partition, whereas on the System Ten a manually operated peripheral switch is required to attach a printer to a partition. It is also important to note that the simulator has the responsibility for assigning the print file names. By using a combination of partition number, date, and time, unique names are assigned to avoid name collisions between partitions or the same partition at different times.

The coexistence of simulated and native modes was also taken into consideration when designing the simulated disk drive interface. The simulator was written to use the random file facility of AMOS/1. The System Ten uses a fixed-disk sector of 100 bytes. The Alpha Micro uses a fixed-disk sector of 512 bytes. The simulator maps five System Ten sectors into one Alpha Micro sector, thereby wasting 12 bytes per sector. After the six-byte BCD System Ten disk address is converted to a binary number, a simple calculation determines the record number and displacement within the Alpha Micro file. To increase efficiency, the simulated disk read checks the last block read against the block to be read to prevent extra disk seeks. The simulated disk write, however, always performs a disk write to maintain the integrity of the data.

PROBLEMS ENCOUNTERED IN SIMULATOR DEVELOPMENT

It should be relatively easy to design and develop a simulation of one computer in another, providing that the target computer has at least the capacities and capabilities of the source computer. There are, however, three major areas in which problems may be encountered: deviations in simulated and actual hardware operations necessitated by differences in the source and target computers; interfacing input and output; and idiosyncrasies of the source computer.

Part of the design considerations in planning a simulator involve determining the most efficient method to perform the simulation on the target machine. Though it may be possible to follow the original hardware design exactly, the extra instructions required may produce an inefficient simulation. It therefore becomes necessary to deviate from the original hardware design, but it is extremely important to ensure that the proper results are generated. One example of this phenomenon in the System Ten simulator involves performing the arithmetic functions. Although the MC68000 processor has BCD add and subtract instructions, they address one byte as two BCD digits rather than as an ASCII character. The System Ten design also allows the two operands to be of different lengths, from one to 10 bytes. To accommodate these conditions, the simulator copies the operands for the lengths specified in the instruction into work areas, performs 10-digit BCD
arithmetic, and copies the proper result to the target operand for the proper length.

As mentioned above, interfacing input and output is probably the most difficult part of implementing a simulator. As an example of the type of input and output problems encountered in the System Ten simulation, an output operation to a CRT that causes the screen to scroll places a 3 in the condition code register of the processor. This condition code can be used to determine whether the terminal is a hard-copy work station (which will not set the condition, since there is no screen to scroll) or a CRT. If the device is a CRT, this condition can also be used to determine if the CRT has 20 or 24 lines on the screen. Since most terminals currently available do not provide this type of feedback, it becomes necessary to perform terminal simulation as well as a processor simulation.

Another difficulty of implementing a simulator involves duplicating the idiosyncrasies of the hardware, which will often be used by operating and application systems. The CRT test just described is an example of one System Ten idiosyncrasy. Another example involves the use of overlapping operand fields, which provide consistent and predictable results but are often used by programmers to perform data transformations whose purpose is difficult to understand. The EDIT instruction, normally used for inserting punctuation characters into numeric fields to provide a formatted output, has been cleverly used to produce the absolute value of a field by overlapping the source and target operands. Still another example uses an obscure feature of the System Ten partition switching operation whereby a shared subroutine can safely execute self-modifying code in lieu of longer reentrant code. For the simulator to allow this, it was necessary to duplicate the relative execution time of each instruction, accumulate the simulated elapsed time, and allow a partition switch to occur at the exact place that the System Ten would have switched partitions.

To assist in solving these problems it is extremely helpful to have debugging aids on both the source and target machines. These aids will normally take the form of trace programs, which allow single-step and continuous tracing. It is also invaluable to build a trace function into the initial version of the simulator that can produce a display of each instruction as it is executed. When this trace is compared to the trace displays from the source computer, it should be easy to spot discrepancies, although in some cases the comparison will be very time-consuming.

RESULTS

Following extensive research into the design and structure of the System Ten and the Alpha Micro, approximately two person-months were required to bring the software simulation from initial design to installation of the first production version. The simulator, written in AMOS/L MC68000 assembly language, requires approximately 18Kb of memory plus the memory required to simulate System Ten partition and common memory.

During the initial design phase of the System Ten simulator program, it was determined that the simulated environment would be considerably slower than native System Ten operation. This was due to the inordinate amount of time that would be spent encoding and decoding the unique System Ten memory addressing scheme and to the significant programming involved in simulating the System Ten arithmetic and partition switching functions. In reality, the overall simulator throughput is twice as slow as the native System Ten used for the comparison. Certain arithmetic intensive applications proved to be even slower, whereas the difference in the data entry type of task appeared to be nominal. It was also determined that adding more than three or four simulated partitions began to cause a noticeable degradation in both the simulated and native system operations. This degradation proved to be the result of the terminal simulation required to support the System Ten environment rather than overhead in the processor simulator. The area of terminal simulation appears to be the only area of the simulation that can be targeted for additional development to improve efficiency.

In the particular installation where the simulated environment is in use, the software transfer plan consists of (1) rewriting the three major applications that consume approximately 80% of the processing load into the native Alpha Micro environment and (2) using the simulation to perform the remaining 12 applications until they can be scheduled to be rewritten. This allows all production work to continue, while essential new application development can also be accomplished.

CONCLUSIONS

Simulated environments can be used effectively as aids in transferring computer operations to a new host machine. At best, this approach may offer only an interim solution to the software transferability problem. But it does allow the data processing installation to continue uninterrupted while software for the new host system is generated.

This is especially true in the case of the System Ten, where software is so dependent on features of the hardware that automated or semiautomated software translation is not feasible.

It is also evident that current generation microcomputers are more than adequate to support a computer environment that simulates that of a minicomputer-based system.

REFERENCES
