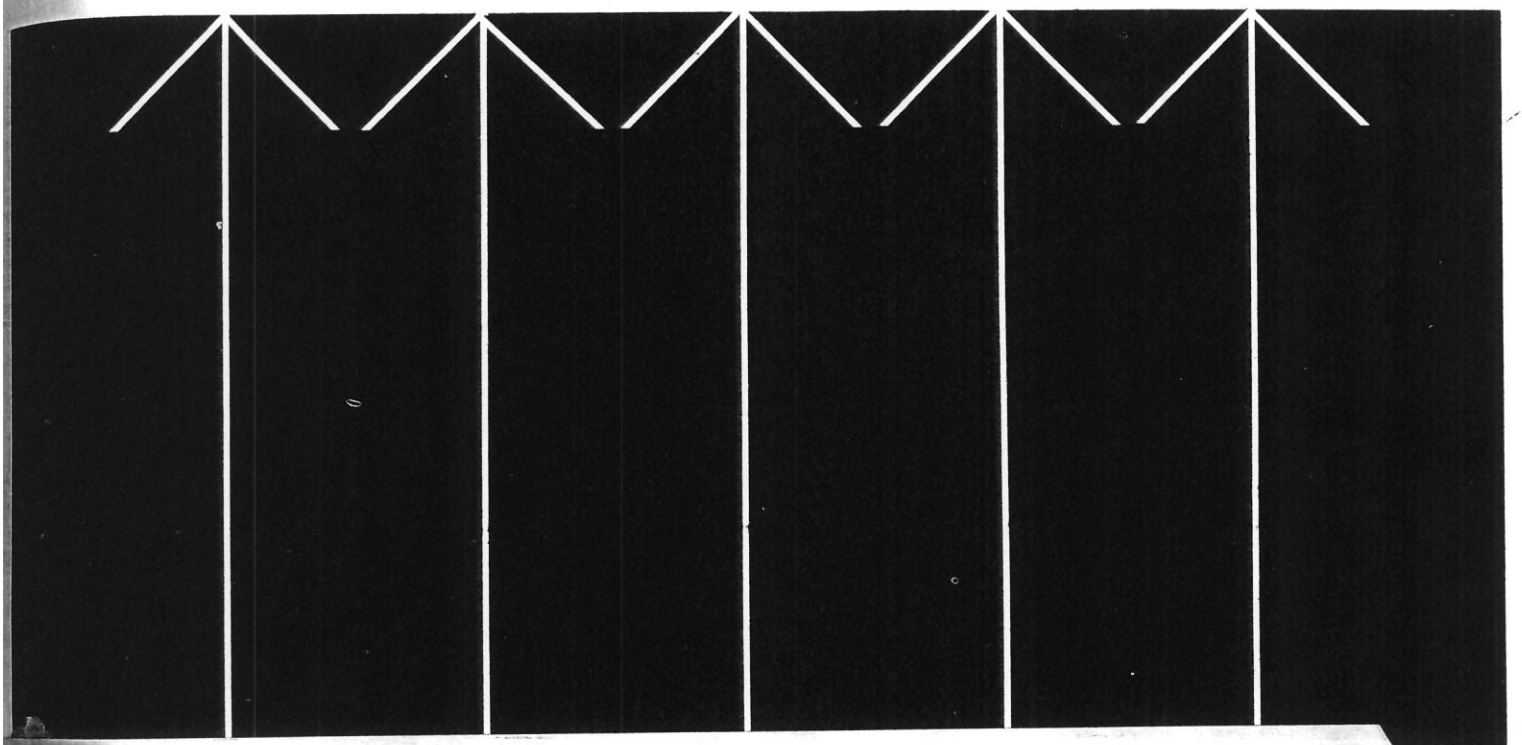
MAPPING DEVICES AND THE M44 DATA PROCESSING SYSTEM

R. A. Nelson



RC 1303 R. A. Nelson October 20, 1964

MAPPING DEVICES AND THE M44 DATA PROCESSING SYSTEM

NOTICE OF DECLASSIFICATION

This Research Report may be released for unrestricted distribution as of April 15, 1969. Please attach this card to your copy of the Report.

Phyllis G. Stigall
Research Publications Officer

MAPPING DEVICES AND THE M44 DATA PROCESSING SYSTEM

by

R. A. Nelson

IBM Watson Research Center
Yorktown Heights, New York

ABSTRACT: A modified 7044 computer is introduced. The computer, called the M44, has a mapping device which assists in the solution of a number of system problems. Several kinds of mapping devices are displayed, with remarks about their advantages and disadvantages. The M44 mapping device is displayed in detail.

Research Report
RC-1303
October 20, 1964

Printed in U. S. A.

INTRODUCTION

A complete report upon a subject such as the machine organization dealt with in this paper has perhaps three aspects to consider. The first might be called the theory of the matter. The second concerns itself with simulation work and related experimental work. And the last would be an evaluation of the subject made from a real life implementation.

This report is restricted to the first of these three aspects. A following report will cover the simulation work to which reference is made from time to time in this paper. One or more evaluation reports will be made at an appropriate time.

This report and the computer introduced here represent the combined work of individuals in Research Departments 440 and 450, in Kingston DS Department 679, and especially, the work of the Experimental Programming Group in Department 440.

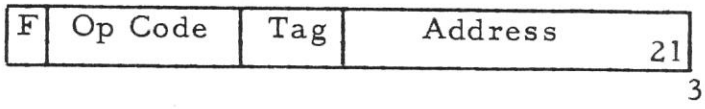
THE M44 DATA PROCESSING SYSTEM

A modified 7044 computer is being established in the Research building. The schedule calls for an installation date of November 1, 1964. The machine has two basic modes. In one mode it is a standard 7044. In the second basic mode the machine is called the M44.

The basic characteristics of the M44 are as follows:

CPU. The CPU has 7 index registers, an instruction counter, and an address register each 21 bits long. Addresses for a 2 million word memory can be generated. There is an additional 22 bit register called the Location Test Register, and an associated mode in which programs will trap upon generating a designated effective address, or alternatively upon generating that address for the purpose of executing a store type operation. The AC and MQ registers are the

normal 7044 registers. The instruction format is



The instruction set is quite similar to that of the 7044. The 7044 instructions which contain decrement parts are altered to skip type instructions. Those which refer to decrement parts have been deleted. There are additional instructions to deal with a variety of modes, and a large number of traps and interrupts have been added to the machine.

Core Store. The M44 will have initially a standard 7044 32K 2 microsecond memory. It will have in addition 96K words of 8 microsecond memory, directly addressable and usable as a computational store. Early in the year there will be an additional half-million words of 8 microsecond computational store added. The core store is available to the 7044 mode through a mode switching instruction and the transmit instruction.

I/O. The M44 will have 3 channels, 14 tape units, printer, card reader and punch, and a 1301 disc file, all available in the 7044 mode as well. In addition, there will be a modified 7750 attached with a so far undetermined number of 1050 consoles. The consoles will be used in a time shared on-line program preparation application, as well as for other purposes.

A machine manual will be available in the near future; therefore, a detailed description of the M44 will not be given here.

The M44 is an experimental machine for which a number of projects have been tentatively specified; for example, certain kinds of information retrieval, natural language translation, and the effect of

the hierarchy of computational stores on certain programs which greatly exceed the memory capacity currently available to them. It will be used for experimental programming work in compiler construction and system organization. In addition, the 7044 will be operated with the large store used for system residence and system scratch store, with possible benefits in reduced system overhead.

The M44 will have a mapping device (MD) and an associated mode. The MD and its value in multi-programming, time sharing, and storage allocation are the principal concerns of this paper.

PROCESSING MODES

It is possible to distinguish a number of different modes of utilizing present and proposed large scale data processing machines. Some of them are of particular interest because they present similar problems to the programmer.

a) There is a class of programs characterized by the fact that they occupy most, if not all, of the machines computational resources for very long periods of time. These programs are mainly large scientific programs. In a machine with a computational store which is small relative to the mass of information in the program, a serious storage allocation problem can results. At present, this problem is solved by building into the program a pre-planned solution to every contingency that may arise during the execution of the problem. The input-output programs and related programs become part of the information to be dealt with by the solution. It can be the case that the programmed structure of the store does not reflect the true dynamic requirements of the problem program and the current data set. Thus there is inherent in a pre-planned solution the transmission of much

information which, although necessary to insure correctness, is not used in the particular flow path taken. The extreme difficulties that face the programmer are diminished when possible by accepting limitations on the size of the problem program imposed by the size of the computational store and by using the largest computational store possible. For very large programs, a discontinuity in the size-time curve is accepted, up to a point determined by the reliability of the hardware.

There has been design work done for ultra-high speed computers, which incorporate small, very fast core stores, and larger, slower core stores. In such a hierarchy of stores, the larger ones are sometimes conceived to be only back-up stores to the small, fast computational store, and sometimes the large stores also play the role of computation store. In either case, the small stores, of 4K or fewer words, must be used efficiently in order to derive the tremendous speeds claimed for these computers.

This poses a storage allocation problem of increasing difficulty as the computational store becomes smaller and faster. What is needed is a technique for dynamic allocation which will keep the small high-speed store loaded with the necessary minimal pieces of the program, the contents of the store always reflecting the precise flow of control in the program.

b) The present technique for processing programs on a large computer is, by means of a control program, to place (almost) all of the hardware at the disposal of a program, to cause the execution of the program, and then to reinitialize the control program and reposition the tape containing the standard system functions in order to establish an initial system position prepared for any defined eventuality.

While the typical scientific computing installation may have many large production programs in its work load, it will have many more programs which are small compared to the hardware resources. The maximal claims made by such programs do not include all of the I/O units or all of the core store. Moreover, these claims are indeed maximal in that they include space in the store for instructions, intermediate results, and initial and final values which are highly dependent on the precise data set used in a particular execution of the program. The claims include channels and I/O units, the use of which is also highly dependent on the data set. Of the resources claimed, then, a certain amount will actually be used. Many programs organize themselves into phases, so that the active area of the core store is localized, with intermittent shifts to other areas. Within phases, the organization of programs is usually into subroutines, so that in shorter time intervals there is an even more localized activity in the core store. And at the extreme, the program is proceeding by instruction fetch and data fetch, and can never be said to need all of the core at any moment.

Some recent measurements indicate that for many programs a storage area two or three times as large as necessary is claimed.

All of these remarks about small programs can be seen to apply to large programs also. The programmer attempts to decompose his very large program into what is essentially a set of intersecting small programs, each of which will have the same data-dependent properties of small programs. These properties contribute greatly to the inefficiencies and difficulties of pre-planned storage allocation.

Multi-programming is a processing mode in which a control program attempts to honor the hardware claims of several distinct

programs simultaneously. The object is to keep the CPU busy executing problem program instructions as opposed to its being idle during input/output operations which may arise in a problem program or in a control program which is removing one problem program from the machine and bringing in another. The goal is to increase throughput. The fact that problem programs make maximal claims which must be honored initially, and without variation, tends to preclude successful multi-programming except for programs carefully selected in advance, and so scheduled for execution that some gain results.

What is needed is a technique for dynamically structuring the store with those parts of a problem program which are actually needed during the various moments of execution, and assigning to the balance of the hardware parts of one or more other programs such that they are able to use the computation facilities - in particular, the CPU - should the current principal program allow them to go idle.

A further refinement in multi-programming comes from the observation that many machine runs consist of a compilation and an execution, or an assembly and an execution, and that if multiprogramming could be effectively accomplished, one would discover many copies of the compiler, assembler, the I/O routines, and programs from a math library, existing in the core store at the same time. Therefore the notion of common, re-entrant programs arises: programs which can be entered to work upon one data set, whose execution can be terminated at an arbitrary point, re-initialized, re-entered and executed against a different data set, and after perhaps many different uses, can be caused to recover that first data set and complete its processing. At the same time, it is necessary to keep

relatively small the penalty paid for using such a program instead of an equivalent program structured for complete execution upon a single data set.

There is a secondary definition of multi-programming which covers an application in which the computer responds to one source of real-time transaction. During the periods of idle time, with respect to that source, ordinary batch processing background jobs are accomplished in a conventional way. This type of sequential interleaving of jobs has been implemented a number of times with great success, but has not produced solutions to the problems arising in the expanded definition.

c) Although there are many areas where time-sharing of a machine and man-machine interactions are of great value, we can restrict oursleves to one area which to a great extent displays the problems of immediate interest.

The preparation of problem programs quite often consists of repeated compilations and assemblies each followed by partial or complete execution of the programs against sample data decks. Frequently there are interspersed execution runs for the purpose of obtaining memory dumps of the machine. These pictures of events at certain precise moments during execution are used to uncover program errors. There is a degree of uncertainty in this process as to how thoroughly the program has been debugged.

An arrangement whereby the problem programmer seated at a console and a system debugging program converse about the programmer's problem program intermittently during its execution has been shown to be of great value. The commands issued by the programmer, and the responses on the part of the operating system have

the nature of transactions, of small duration in time, against a large amount of data - the problem program, symbol tables, etc.

Between each transaction there may elapse a relatively large amount of time as the human considers his next request.

A crude picture of time sharing shows a control program which, during these moments of idleness with respect to one console, outputs all of the operational store, inputs a previously outputted store image for a second console, and then permits the execution of a transaction for that console. This procedure is extended to include as large a number of consoles as possible; there being a limitation imposed by the fact that the response time to a console depends upon the number of consoles active, and too long a response time seriously detracts from the human thought processes.

All of the various remarks about problem programs which lead to the consideration of multi-programming can be introduced here. There is a cost in this operating mode in terms of time used for the exchange of core images in order to respond to console requests.

In fact, the larger and more complex the program, the more useful on-line debugging can be, and the greater the cost of exchanging problem programs. This cost is an especially painful burden since the debugging process is often such that no more information is required for a console transaction for a large program than would be required for debugging a small program.

What is needed in view of the application is a system technique which encourages the transaction aspect of time sharing for debugging purposes, a technique for eliminating redundant copies of popular functions, and a facility for dynamic core allocation.

If a sufficient amount of CPU time is allotted to each console in turn, and if there is a high probability that a console request can be

honored by what is already in the core store, then time sharing can be quite effective and quite economical. If the console request cannot be honored in one time slot, then there are various techniques for interrupting that operation and returning to it later on. If I/O is necessary in order to honor the console request, then it is desirable to introduce the control techniques of multi-programming in order to keep the CPU busy on problem program work.

An important aspect of time-sharing for debugging is that of compatibility. A program is prepared and debugged in the presence of other programs, i.e., debugging and analyzing routines, which will not be present during production runs. The resultant debugged program should be executable in a multi-programming environment during second or third shift, for example. It should be executable as a single program in the event that it requires, as a production program, many hours of machine time. It should be able to play the role of a background job during a period of expected but insufficient console activity. The presence or absence of debugging routines or other problem programs should not influence the construction of the program.

d) There are proposed computer applications where the computation processes are responding to and controlling to some degree events in the real world. These applications permit a burden to be shifted from humans onto the computer, and often permit a degree of efficient control which may be unobtainable by other means. In such applications hardware reliability is extremely important, and failing perfection it must be possible to keep the critical parts of the control process going on even though parts of the computer, such as memory boxes, CPU's, or channels, fail and are removed from operational

status for repair work. The programs in such a system must therefore not be tightly coupled to a particular hardware configuration, but must be executable on a variable configuration with sufficient efficiency to meet the real time requirements.

In the future, some critical real-time systems, and possibly some large scale general purpose installations, will consist of computer organizations which incorporate more than one CPU. For the single large program, it can be seen that there is a problem in organizing the program in such a way that various parts of it can be worked upon simultaneously as though several independent problems were being processed, without the danger of any CPU proceeding beyond certain interlocking points. This organization problem, often called the folding problem, contains many of the problems of multi-programming in that the various parts of the program can be viewed as independent programs which have in common certain sub-programs and in this case certain data areas. Additionally, the resultant organization produced by inspecting the program logic is not likely to be the organization best suited to keeping all of the hardware busy all of the time. There will be differences in storage requirements, and differences in the amount of CPU time required for the various parts. It is likely that a control program should be presented with control information based on near-maximal folding of the program, regardless of the number of CPU's. The control program should then be able to make dynamic core and CPU assignments based on the control information and the current state of the machine.

The notions of multi-programming and real-time processing apply also to multi-processor machines, as do the attendant problems.

An additional problem arises in multi-processing. In the event of a CPU trap or interrupt which brings into execution those parts of

the control program which dynamically schedule the hardware, a second CPU must be prevented from also executing those parts of the control program. However, pre-assignment of one specialized CPU to this function is not adequate since, apart from the possibility of failure of that processor, a CPU-initiated trap leaves that CPU idle; therefore, it is desirable to have the control program activated in the CPU causing the trap.

COMMON PROBLEM

The principal problem which arises in all of these processing modes is the problem of core store allocation.

It is most convenient for the programmer, and for the compiler, to write programs for a machine with a core store large enough so that there is no storage allocation problem. It may be that a program so written has the property that it will not behave well during a production run in a machine without at least as much store as is claimed. Such a program may need at this time all of the core store, and yet we wish to debug the program in the presence of presumably large system routines designed to aid debugging.

For production runs of large programs which behave like collections of sequential smaller programs, and for small programs, we would like to adopt a multi-programming mode of processing. However, channels and CPU's left idle by one program cannot be applied to a second program unless that second program has been assigned space in the store and it prepared to use the CPU for computational purposes or for eliciting the I/O commands which in turn will utilize the idle channels. The problem of keeping several programs in ready status - ready to occupy the CPU - is complicated by the current

necessity of assigning to a program core areas which are contiguous and honor the maximal claims of the program.

When parts of programs, or entire programs, have outlasted their usefulness, areas in the core store become free for use by other programs and data regions. However, these free areas may not represent enough contiguous space to hold waiting program objects. The store then becomes decayed, i.e., it has free areas which cannot be used. This factor produces a scheduling problem for both the programmer with the single large program and the multi-program mode, or real-time mode, control programmer.

For the MP control programmer, storage decay presents the alternatives of waiting until enough contiguous store is available, or restructuring the store, in the event that the hardware lends itself to dynamic relocation. If the control program waits, the hardware is idle. If the core is restructured with some frequency, a cost is incurred which may outweigh the gains in multi-programming. The SP programmer, and the RT control programmer face similar problems in that for quite different reasons the waiting information must be brought in. The result is that there must either be a partial or complete exchange of the contents of the core with the back-up store, or else the store must be restructured, or some combination of both.

An ability on the part of the control program to put a problem program in ready status without honoring the complete storage claims and without the necessity of altering the addressing structure (claims) of the program would be of great value. Properly arranged, it would permit effective multi-programming, improve the efficiency of real-time applications, and enable dynamic allocation of small high-speed stores.

There are two aspects of this kind of allocation and relocation which would be of great value: first, that an incoming piece of program could be placed arbitrarily in the store and made to run correctly, whether it is entering for the first time or not, and second, that other parts of the program not in the store are able to adjust to the new position of one of its fellows, without a time consuming burden being placed on a control program.

Currently implemented techniques for dynamic relocation permit programs to be moved up or down in the core as complete contiguous entities, and so enabling a control program to restructure the contents of the core, or to load a program into the store beginning at an arbitrary origin. In a time sharing system with a large number of consoles active, there is only a slight possibility that a round-robin algorithm will be able to preserve a problem program in ready status from one transaction to the next. While the current technique may permit in certain circumstances the overlapping of input and output of problem programs, it does not diminish the amount of information traveling to and from back up store.

In an MP mode, this type of dynamic relocation eases the burden on the control program to the extent of preserving the address structure of the program while restructuring the store or entering a new program. It does not permit increase of the number of programs in ready status beyond this point.

A related problem is that of program protection. Programs which share the core store must not alter each other, nor should control escape from one program into areas unknown to the programmer. Common subroutines and control programs must not suffer from the execution of undebugged problem programs. Within problem

programs, it is sometimes desirable to have unalterable areas of store, and in certain applications it is required that some areas be alterable only by certain distinct, independent programs, and not by all programs in the store. These considerations are usually implemented by providing protection for core areas, and by insuring that the information for which protection is wanted occupies an area of store which can be so protected. This notion of protection will be found in the M44 also, but imbedded in the mapping device and program address structure, and not related to the physical core store.

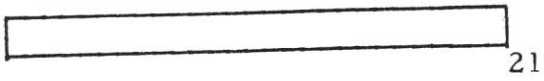
A POSSIBLE SOLUTION

The problems so far touched upon for the various processing modes are the problems of maximal claims, ready status, storage decay, exchange, scheduling, dynamic relocation, common subroutines, protection, programming ease, and processing mode compatibility. A solution may lie in a device based upon a notion of a virtual machine. The device will be called a mapping device. A mapping device is an addressable memory whose principal function in the machine is the monitoring of addresses used by problem programs, and the conversion of these addresses into other addresses. It can be thought of as a kind of indirect addressing which is not specified by the problem program but which is implemented by a control program in order to compensate for permutations made to the ordering of information in the store when no corresponding changes have been made in the problem program addressing structure.

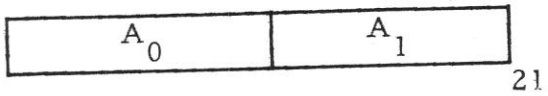
For convenience, we will refer to the problem program addressing structure as an addressing scheme for a virtual machine (VM) and a virtual store (VS), the M44 as the real machine (RM) with a

real core store (RS). The parameters used in the examples apply to the M44 and its mapping device.

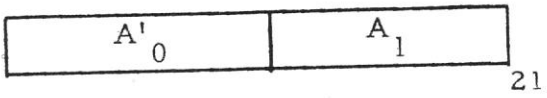
Suppose that the problem program generates a 21-bit effective address



By means of hardware, let the address be decomposed into two pieces A_0 and A_1.



A_0 is used to address the mapping device, which is to respond with a bit field A_0'. Substitute A'_0 for A_0 and let the memory reference proceed.



The problem program addressing range is thus, by hardware, broken up into a set of equal length blocks. A control program may position these blocks in the store in any order convenient for it, and then by making an appropriate entry into the mapping device, cause the problem program to be executed correctly. Since in every processing mode discussed we wish to permit the addressing range of the problem program to exceed the amount of core storage allotted or available to the problem program, a provision for trapping into a control program must be included to cover the case when the mapping device is unable to make a substitution. The control program can now make an exchange with a backup store, and then the CPU can be given back to the problem program.

It is not necessary to allocate two million words of disc storage for each problem program, although for single problem processing

there might be some gain in taking the number of the needed block as, in effect, the disc track number. Since most programs will be of substantially smaller size, it will suffice to allocate only enough disc storage to honor the amount of storage claimed. A disc track number can then be calculated from a small set of core-store / disc-track equivalence pairs, one equivalence pair for each contiguous area of core store.

There is a cost of some CPU time in performing the mapping function. Generally speaking this cost is approximately the cycle time of the mapping memory less any portion of that time which may be buried behind other operations. Since many instruction fetches take place from sequential locations, it is possible to design a device which, for instruction fetches, uses the result of the last instruction fetch map, unless a transfer occurred, or the program sequenced skipped or sequenced out of the last instruction block.

This can be accomplished in a way similar to the technique used on the 7094. In that machine, words are fetched in pairs from the memory, and a redundant instruction fetch is eliminated whenever possible by using the second word of the pair if correct execution can be assured. Since the data word for an instruction is usually far from the instruction, and consecutive data references are also far apart, such a technique would be of little benefit if used for data references.

There is a relationship between memory sizes and speeds. The faster the memory, the smaller it must be. Since we wish the mapping function to be fast, the mapping memory should be small compared to the size of the operational store of the RM.

To use some examples, suppose the VM core to be a two million word store (2^{21}).

block size	nr. of blocks
64	32,768
128	16,384
256	8,192
512	4,096
1,024	2,048
2,048	1,024
4,096	512
8,192	256

Suppose that the processing mode involves running a large problem in a machine with a reasonably large store for computation and a small high speed local store of say 4096 words. We want the mapping device to be appreciably smaller than 4096 words, but for a block size of 128 words, there are 16,384 blocks which may be mentioned. We therefore want a mapping memory which can respond to 16,384 arguments correctly, but which contains fewer than 4096 words. This need is satisfied by an associative, or content-addressed, memory. In the example, an associative memory (AM) of 32 words can describe the 128 blocks representable in the 4K memory. It may be that for some programs, only instructions and program constants would reside in the 4K memory, and data references would go through a different mapping into the slower store. Or fetches may refer to the high speed store, or by dyanmically structuring the high speed store most references may occur there with the less frequent references going to the slower store.

A result of simulation work is that there is a relationship between the block size and the real core size. The more real core that is available, the larger the optimal block size becomes. Suppose that there is enough real core to honor directly every reference in a virtual store. A large problem program might usefully be run out of

the mapping mode in such a machine. But in the debugging stages, and for all those programs that lend themselves to multi-programming, we would want to use the mapping device. In such a case, suppose that the large operational store is used in blocks of size 2096 words. Then the virtual store is divided up into 1024 blocks, since we are multi-programming, we would need to represent say sixteen VMs at one time, or a total of 16,384 blocks. An associative memory for such a store would need to be 1024 words long. However, it may be practical to consider a mapping device of 16K words, if we can use a standard location addressed memory (LA) sufficiently faster than the 2 million word real core. Such a memory would be less expensive than an AM and could provide adequate service for the processing mode. It is this type of MD which is to be used in the M44.

An intermediate device is the hash addressed memory. This scheme attempts to bypass the dollar cost of the AM, and the time cost of a large location addressed memory, by implementing in hardware a hash function applied to the VM block m. and ID m., and using the resultant condensed bit field to directly address a standard memory. The format of the mapping memory word is similar to that of the LA mapping device. Additional information for chaining purposes must be retained in the memory.

After the hash address has been formed and a word fetched, a comparison must be made to determine if the word fetched does indeed describe the block desired. If not, an automatic chaining device is called into play and after a number of additional fetches, a map or trap will occur. This type of memory has an additional disadvantage in that the addition or deletion of information in this memory involves the bookkeeping necessary for keeping the chains properly structured.

As with all hashing schemes, the device responds better with a larger hash domain. The location addressed memory might be viewed as the extreme of a hash memory, the hash function being the identity function. Response time is a function of chain length, which in turn is a function of the size of the hash domain. In simulation work, arbitrarily good results can be obtained by letting the hash addressed memory grow in size.

The principal function of the mapping device is the mapping function. In each of the three mapping devices, an effective address is generated in the CPU, decomposed into two parts: block number (A_0) and word-within-block number (A_1). In order to distinguish between a number of different problem programs allocated space in the store, we wish to include an identification number in the mapping function. This ID number is associated with a virtual machine. It enables a control program to deal concurrently with many different programs. An ID number is therefore concatenated with the block number and the result is used to address the mapping device. The response is either an effective address for referencing the main store, or a trap. The trap logic is discussed separately on page . It is chiefly implemented through a set of bits, called status bits, included in each word in the mapping device.

The mapping device machine now looks like this: there is a control program which can address and use all of the machine's hardware. It is very small. When it wishes to begin or resume execution of problem program i (virtual machine i), it puts the integer i in an ID register, loads the CPU with the initial or current quantities for program i, and then simultaneously enters the mapping mode, the problem (privileged instruction) mode, and transfers to the

indicated instruction for program i. The problem program's addresses are modified by the mapping device during execution. From time to time, traps to the control program may occur. When a trap occurs, the machine leaves the program and mapping modes. A typical trap is a select trap, which occurs when the problem program begins to execute an I/O instruction. The control program will ensure correct execution of the I/O operation. Another trap would occur if the problem program attempts to alter the contents of the mapping device. A mapping mode trap may occur indicating the absence of a block of the virtual store. The control program bring in that required block, alter the contents of the mapping device, and then resume execution of the problem program. This activity may include an output operation for a block chosen for replacement, and a good replacement algorithm is critical at this point. During any of the I/O services rendered by the control program, it may be the case that the problem program cannot continue using the CPU until the I/O operation has been completed. In that event, the contents of the CPU are stored, and then the CPU may be initialized for virtual machine j. The ID register is loaded with the integer j and the execution of that program can then be resumed, at least until the control program receives an interrupt signaling the end of the I/O activity for program i.

ASSOCIATIVE MEMORY MAPPING DEVICE DETAILS (Diagrams on Pages 23 and 24)

The associative memory contains an entry for each block in the computational store occupied by problem programs. The argument register of the AM is loaded with ones in its ID field. Each time a

mapping cycle is to be performed, the block number is automatically loaded into the argument register A_0 field and a search operation proceeds as follows: for every bit that is a one in the mask register, that column is searched for agreement with the corresponding bit in the argument register. With the ID field treated as a position field rather than an integer field, the ID portion of the Mask Register shows a single bit in position j for the j^{th} program. The effect of this arrangement is to permit shared blocks to indicate this property by filling out the ID field of an AM word with ones for each VM for which the block is valid. The Real Core block number is supplied implicitly; agreement on the i^{th} word causing the integer i to be read out. The ID field can be used as an integer field, but the intersecting store property is lost unless provision is made to permit intersecting only restricted areas such as the low order or high order parts of the virtual memories. Then a test of the block number can be made and if it falls in the proper range, the ID field search could be suppressed. This technique requires all virtual stores to have the same contents in these intersected areas.

A replacement algorithm for an associative memory can be implemented in the mapping device itself. An example is the following scheme which has been simulated and appears to be very promising.

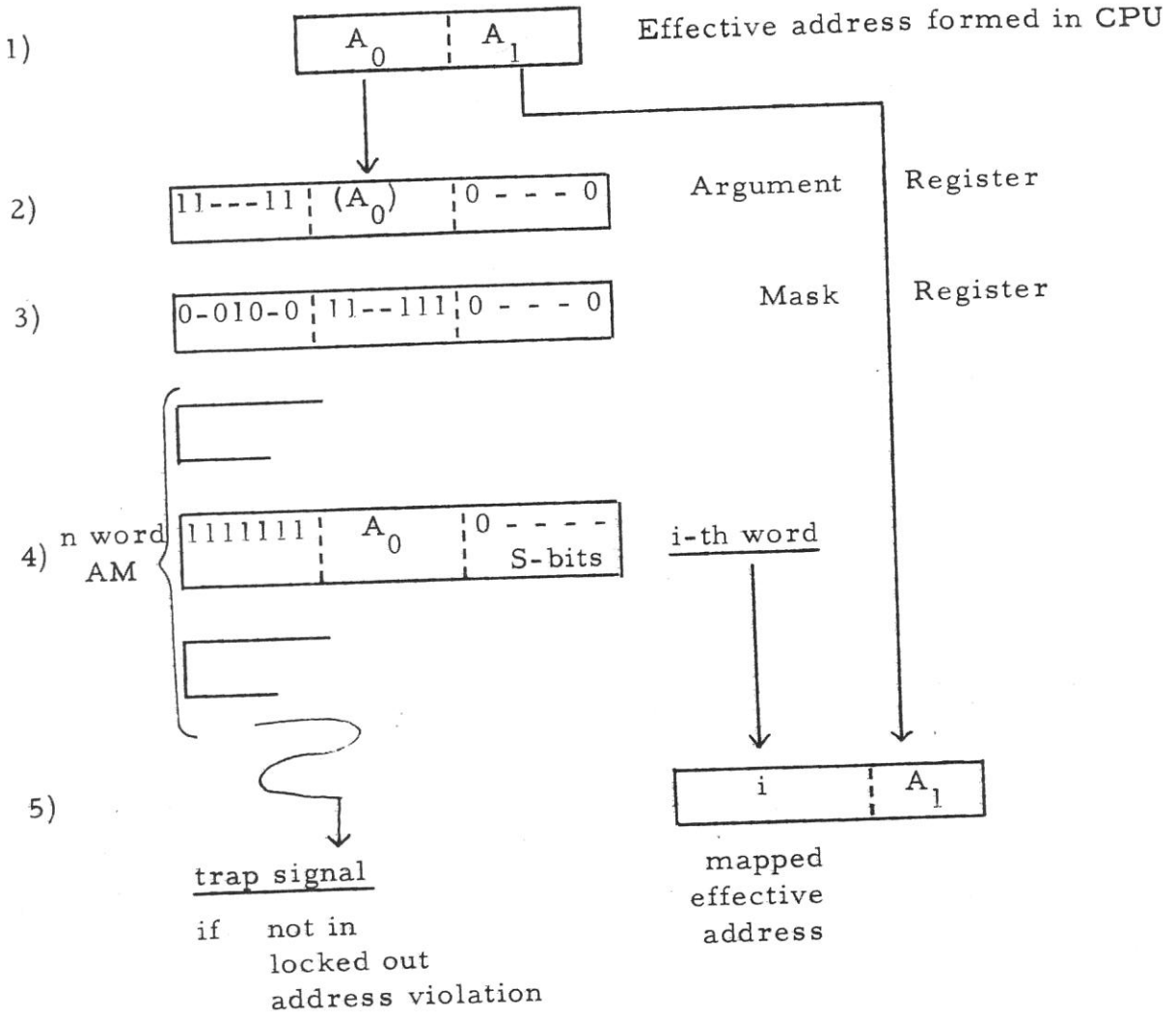
In the format of the AM word, let there be two (status) bits, an alteration bit A which is set to one whenever the corresponding block is altered, and a second bit, the reference bit R which is set to one whenever a successful reference is made to the corresponding block. Additionally, whenever all of the reference bits in the AM are on, let them be reset to zero. At any moment, and in particular

at replacement time, the following values are possible.

A	R
0	0
1	0
0	1
1	1

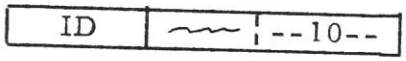
Due to the reset of the reference bits, there must always be at least one AM word with bit values 0, 0 or 1, 0. We wish to choose an entry with values 0, 0. If there are none, then we wish to choose an entry with values 1, 0. The algorithm clearly prevents overwriting the block from which the last instruction was fetched. Unlike the mapping function, this search can result in multiple matches. The simulation work mentioned previously dictates that we should choose randomly from the set of matches. A technique which approximates a random choice is the following (page 24). Those words which match after a search instruction is executed are indicated by ones in a sense vector. Corresponding to the sense vector is a replacement vector which contains a single bit set to one at that word position which corresponds to the last block chosen for replacement. The replacement vector bit is circulated until it occupies a position corresponding to a one in the sense vector. The first such position is used. The block is now taken as the block to be replaced. For a time-sharing application, this algorithm may not be effective. The priority requirements of the application may override the past history as shown by the status bits. However, the active reference bit is still useful in suppressing unnecessary output.

Associative Memory - Mapping Function

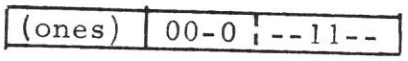


Associative Memory - Replacement Function

1) 'Search' Instr. executed in CPU

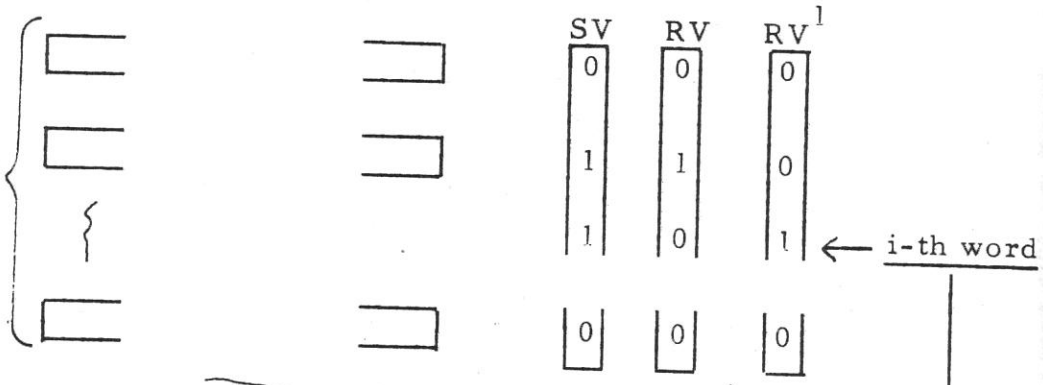
2) 

Arg. Register

3) 

Mask Register

4) n words



5) trap signal
if no match

i

MD entry, RM block nr. (location) to be replaced

A recent proposal by Blaauw leads to the following. Suppose that we contemplate having a very large AM to service a very large core store. If indeed large problem programs behave like sequences of small problem programs, or if many programs are in ready status, then for appreciable periods of time some of the AM words will not be used. Suppose now that we place the contents of the proposed AM in the computational store and then reduce the size of the AM. A trap from the AM does not necessarily mean that the block is not in the computational store. It may be in, and a word describing its location may be found in the core table. It is now necessary to replace only the directory word in the AM, without any corresponding input/output.

The directory replacement or exchange, can be accomplished by hardware, and an algorithm similar to the one previously described could be used for this purpose. True block replacement could not be done except by program steps, unless the allocation and reference bits for the core table were kept in special registers.

The size of the reduced AM cannot be given at the moment. But if an ultra-high speed computer can usefully use a 32 word AM to service a 4K memory, then perhaps a general purpose computer could use a similar size. In fact, there is some evidence from the simulation work that, for some current problem programs, a 16 word AM would cause directory replacement as infrequently as 5% of the total number of references in non-trivial cases. This would cause very little degradation in the mapping service, and largely retain the speed of the AM. A further remark about the small AM will be made in the next section.

M44 LOCATION ADDRESSED MAPPING DEVICE (Diagram on Page 29)

A location addressed memory (standard core memory) must, like the associative memory, respond to a very large number of arguments. But unlike the AM it must contain an entry for each argument, and is therefore quite large. A virtual store address of 21 bits and a block size of 2^{11} words leaves a block number field of 10 bits. To deal with 16 (2^4) virtual machines concurrently, a mapping memory size of $2^{(10+4)}$ words (16K) is necessary. Such a device will not work well with small high speed stores, but can work effectively with extremely large slower stores of several hundred thousand words or more, such as the large 8 microsecond memories which are to form part of the M44 system.

A format of a word in a location-addressed memory is given on page 29. It differs from that of an AM word in several respects. The ID field and VM block number are missing: they are used in addressing the word. The intersection property of the ID field is obtained by multiple entries in the mapping device, or by block magnitude inspection with, when appropriate, deletion of the ID field before fetching from the memory. Because this memory always responds to its argument, an additional status bit must be included to indicate whether or not the block is in fact in the real core. A cross index to a table of contents is presumed to be necessary, and the RS block number must be explicitly contained in the word.

The replacement algorithm discussed with the associative memory cannot be implemented in the location addressed memory. Apart from a possible 2-bit AM for this purpose, such an algorithm can only be obtained by, in effect, reading into the computational core all pertinent words of this mapping device, including all multiple entry words, and then simulating the algorithm by a program. For

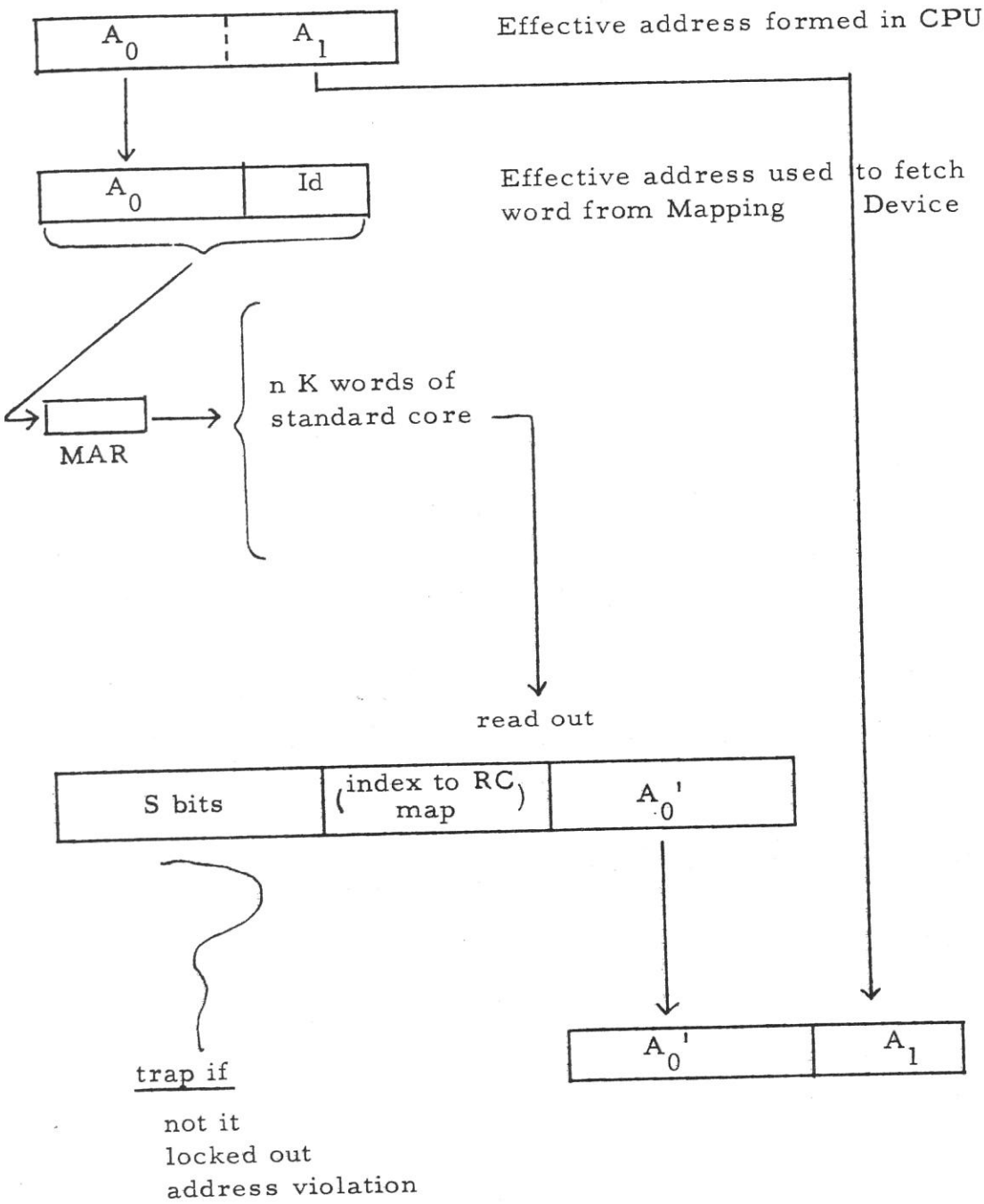
evaluation purposes such a programmed algorithm would be valuable, but as a viable algorithm there may be too much time lost in using the CPU for this purpose.

The location addressed mapping device to be used in the M44 is a 16K 2 microsecond standard core memory. The M44 and its MD have the property that the block size can be varied from 2^7 words to 2^{12} words without hardware changes. A detailed flow description of the MD is given in an appendix. A mask register and an ID register, each 6 bits in size, control the decomposition of the effective address and the concatenation of an ID number. The block size is not altered during execution times. The purpose of this facility is to permit the experimental evaluation of a permanent or semi-permanent block size. An unfortunate aspect of this facility is that the number of problem programs representable in the MD is a function of the block size chosen. It is thought that this will not inhibit the experimental work however.

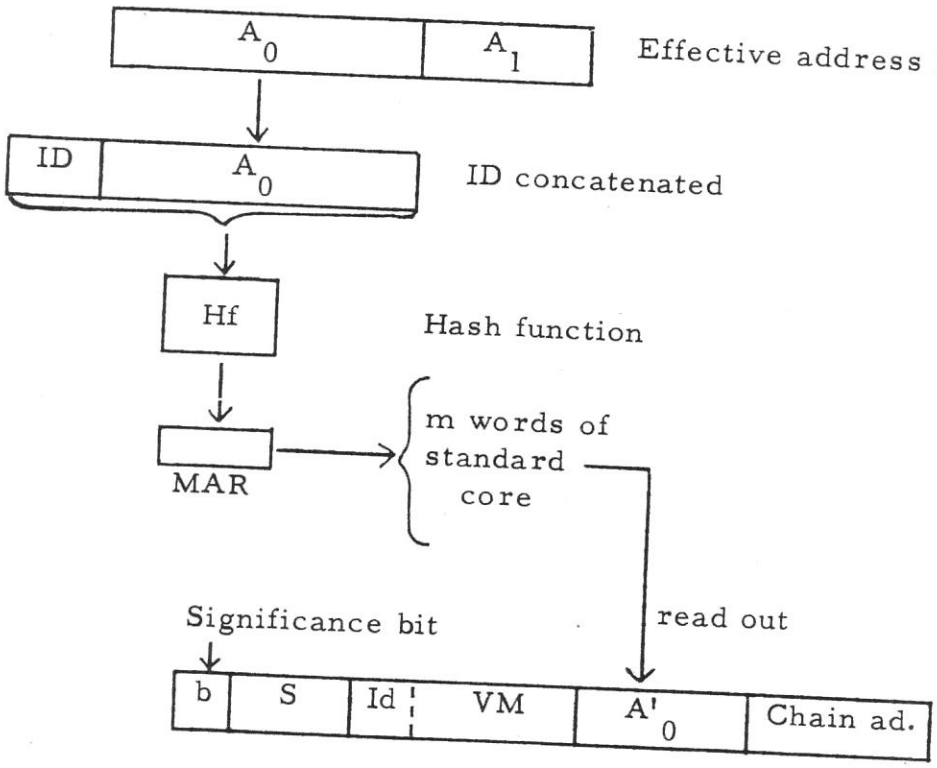
On a previous page, a technique for simulating a large AM with a small AM and some tabular information was described. The directory exchange needed, if the tabular information is indeed the contents of a complete AM, will necessitate a table search since the core store is not content-addressable. The tabular information might however be a virtual store directory, and its contents identical to the contents of the location addressed memory. Now automatic replacement between the AM and the LA devices is possible. This larger directory might well be set aside as a stand-alone memory, in order to permit possible overlapping of various machine functions. The AM and the location-addressed MD now forms a compound mapping device which could be used in any of the processing modes previously described.

Under appropriate and realistic assumptions, the cost in CPU performance due to mapping can be held below 20 percent, and perhaps substantially below that figure. This cost is to be more than regained due to decreased costs elsewhere in the system.

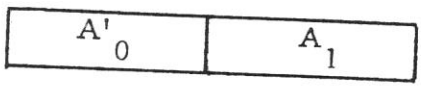
Location Addressed Memory - Mapping Function



Hash Addressed Memory



- trap if b = 0 (not in, no chain)
- b = 1, compare Id, VM:Id, A_0, equal, go to (X)
- unequal, put chain ad. in MAR and fetch again

(X) use 

or trap depending upon S-bits

STATUS BITS IN THE M44 MAPPING DEVICE

Since an important part of the M44 project is to be analysis, measurement, and evaluation, a number of information bits are included in the MD to facilitate the gathering of information and the simulation of a variety of uses and parameters for the MD.

There will be delivered to the MD along with each block number to be mapped, a signal which indicates an I-time (instruction fetch) reference and a signal which indicates an active (store type) reference.

The status bits and their significance:

	1	0
S_0	block in	block not in
S_1	can be used	locked out
S_2	read only	can be altered
S_3	referenced	not referenced
S_4	altered	not altered
S_5	I-time reference	no I-time reference
S_6	conditionally protected	not CP
S_7	privileged	not privileged
S_8	used	not used

S_0 indicates whether or not the block is in fact in the computational store (not in AM mapping device).

S_1 prevents a block's use even when in the store - for example, if the block has some I/O activity, it may be desirable to suppress references.

S_2 a one in this position and an active reference signal will cause a trap.

S_3 a successful map results in this bit being set to one.

S_4 a successful map with an active reference signal causes S_4 to be set to one.

S_5 an I-time signal causes this bit to be turned on. The purpose of this bit is for information gathering.

S_6 this bit is initialized by the control program and associated with those blocks of the VM which are read-only to the problem program but not to the system programs. It is used with the I-time signal and S_7.

S_7 at I-time the contents of this position initialize a trigger P. During the following cycles up to the next I-time, if an active signal is present the referenced block has S_6 set to one, then trigger P must also be set to one (privileged to alter that conditionally protected block) else a trap occurs.

S_8 the location addressed memory is a virtual store directory. It contains an entry for every block in the virtual store. On the supposition that S_3 will be reset by programming in experimental replacement algorithms, this bit is used to indicate whether or not blocks of the virtual store has ever been used. In the event that it has not been used, the input of the block from the disc can be deleted.

Although not currently in the implementation plans for the M44, the status bits can be used to prevent erroneous transfers from the problem program to the system routines, in much the same manner as the conditional protection feature works.

A trap signal causes information to be stored in the trap area of the M44 which in turn enables the operating system to take appropriate action.

Program Protection in the M44

The program protection features of the total system are segregated into three parts. First, the protection of the M44 control

program is the responsibility of that program. It should not so load an MD word that a problem program can refer to it. Second, the inclusion of ID numbers in the mapping function prevents cross references between programs except where it is specifically desired. These cross references are thought of as occurring in intersecting virtual stores. Lastly, protection inside a single virtual store is available, and in the M44 system is related to the protection of the virtual machine's operating system from problem programs.

THE 44X MACHINE

The rules for writing programs which can be correctly executed by the M44 and its control program have been embodied in the description of a virtual machine called the 44X.

The precise definition of this machine enables the construction of a modular operating system (called MOS in the M44 project) which by ensuring execution according to the definition of the 44X can correctly execute any 44X program. MOS does not need to know anything about the nature of the program in the 44X. This ability is in fact the only function of MOS. However, the 2,000 word long MOS will be extended in the M44 to include information gathering routines, and its modularity will be used to exchange, from time to time, the replacement algorithms and the time-sharing algorithm for evaluation purposes.

The 44X is defined as a machine with a 21 bit address field and a 2 million word core store, all of which may be used. Its CPU is the M44 CPU but its instruction repertoire does not include the M44 instructions for mapping device work and other privileged instructions. Such instructions will trap if execution is attempted when the M44 is in the problem mode.

Although the implementation of the M44/44X relationship is based only on the definition of the 44X, it is convenient in particular applications to tell the M44 control program MOS something about certain programs in the 44X.

The 44X has an extremely large address range and it is convenient to consider that the 44X operating system sits permanently in that store. This operating system, for the standard scientific installation, will consist of routines which compile, assemble, debug, maintain disc files, and other programs of that kind. The operating system will be of a conventional kind, and would contain no evidence of multi-programming activities. Although the operating system might be large in an absolute sense, it is small with respect to the size of the virtual store, taking perhaps 5 or 10 percent of the space. But because of its absolute size it is desirable not to reload the program for each new job. Therefore the operating system will be written as much as possible as a read-only program. Those parts which are read-only will be assigned core areas which are protected from alteration by the problem program, or by errors in the read-only programs. The balance of the system, those few parts or collections of single instructions, and the data areas necessary to the system, will be assigned an area which is not alteration-protected from the operating system, but is protected from the balance of the store. This second area, insofar as instructions are concerned, will need re-reading or initialization from the read-only part or from an I/O unit before beginning a new job. Relating this protection to the Mapping Device, the read-only blocks have status bits S2 and S7 set to one. S6 has no influence. The data areas, which may contain instructions, have S2 set to zero, and S6 and S7 set to one.

The operating system is assigned to a fixed location in the 44X and in fact the same fixed location in all 44X's. The read-only parts of the operating system are identical for each 44X. The problem of common, re-entrant subroutines can now easily be solved by the technique for intersecting the 44X stores in those areas where the core contents are identical. A 44X core block which is taken from this read only area can then play its role with respect to many 44X's if the appropriate mapping device entry or entries, point to that block. It is possible to declare arbitrary intersections of 44X core storages, and also to permit some intersected areas to be modified only by certain selected 44X machines.

There are a few other points of contact between the M44 and the 44X loaded with its operating system. For example, the proper handling of a halt instruction in a program which is to be debugged on-line and also executed on-line includes a decision made by the control program.

Since the operating system is to reside permanently in the 44X store, it is possible that erroneous transfers may occur into the system area from a problem program. An additional degree of protection may be necessary to prevent such transfers from taking place.

PROJECT GOALS

An important aspect of the M44/44X project is the construction of an integrated operating system. The 44X system will provide facilities for compilation, assembly, execution, file maintenance, and debugging. All of these facilities are available either through a console conceived to be attached to a 44X, or through normal batch processing techniques. The system is to be integrated in the sense

that a single copy of a function such as a conversion program will be used by the compiler, the assembler, the debugging routines, and by the problem programs. It is hoped, for example, that very few instructions will exist which are only executed if a complete compilation in the ordinary sense is done. The modularity of this system is important for planned experimental work in compilation techniques.

The mapping device will facilitate the implementation of the 44X and its system. However, it is to the measurement and evaluation work related to the MD that the project is principally devoted.

In a time sharing application, it is anticipated that the solution to the common subroutine problem will diminish the system overhead in terms of CPU time for execution and channel time for exchange of memory. The system residue left by the last console transaction will be usable by the next console and its transaction. The ability to begin to execute a problem program before it is brought in from the back-up store will diminish the amount of information brought in. A sufficient number of blocks to honor the transaction will need to be brought in, and not the entire program. While a transaction is being carried out for one console, the capability to effectively multi-program will be exploited by beginning when possible to deal with transactions which may be waiting in a queue.

The shortening of problem programs under a block measurement should enable a larger number of programs to be in ready status, and non-trivially so. A marked increase in throughput for a batch processing mode should be possible.

The techniques for dynamic allocation of large programs to small stores will be investigated, and evaluated with respect to preplanned allocation schemes. The system has the property that arbitrary core

structuring can be forced by preplanned insertion of certain instructions in 44X programs. These instructions can be added to a problem program by a compiler, and can cause input/output to start ahead of time to obtain CPU/channel overlap.

FOR PUBLICATION IN THE FUTURE

A number of reports are planned for publication soon. They are:

A report on the simulation work referred to in this paper.

An M44 manual.

A 44X manual.

A user's manual for timesharing operation.

A description of the 44X operating system.

Measurements of the M44/44X system including a description of MOS.

REFERENCES

1. Corbato, F. J., et al, "An Experimental Time-Sharing System", Proceedings of SJCC, 1962 (AFIPS).

2. Hellerman, H., "On the Organization of a Multiprogramming-Multiprocessing System", RC-522, September 5, 1961.

3. Nelson, R. A., "An Experimental Data Processing Machine", Proceedings of ITL Programming, December 10, 1962.

4. Nelson, R. A., "Problems in Automatic Storage Allocation", RC-601, November 27, 1961.

5. O'Neill, R. W., "Some Notes on the Absolute Core Location Problem", RC-600, January 3, 1962.

6. Roberts, M. de V., "Associative Memories and the One-Level Store", RC-807, September 26, 1962.

7. Tapscott, R. P., "An Algorithm for the Reduction of Intermediate Storage Device Utilization Costs", RC-599, December 22, 1961.