Microcoded microprocessor simplifies virtual-memory management

When the operating system handles virtual-memory operations the programmer is liberated from clumsy restrictions

by Thomas W. Starnes, *Motorola Ltd., Semiconductor Products Sector, Milton Keynes, England*

☐ Virtual-memory techniques have been implemented successfully in mainframes and minicomptuers in the past. But schemes to extend the memory in microprocessor systems have had two major drawbacks. Either they required an additional microprocessor to perform virtual-memory operations or their software had to be written in overlays within well-defined boundaries. Motorola's MC68010, however, allows the operating system to handle virtual-memory operations, freeing the application programmer from the restrictions previous designs had. To fully support a virtual-memory scheme, the 16-bit microcoded microprocessor has seventeen 32-bit general purpose registers, a 32-bit program counter, a 16-bit status register, a 32-bit vector register, and two 3-bit alternate-function code registers.

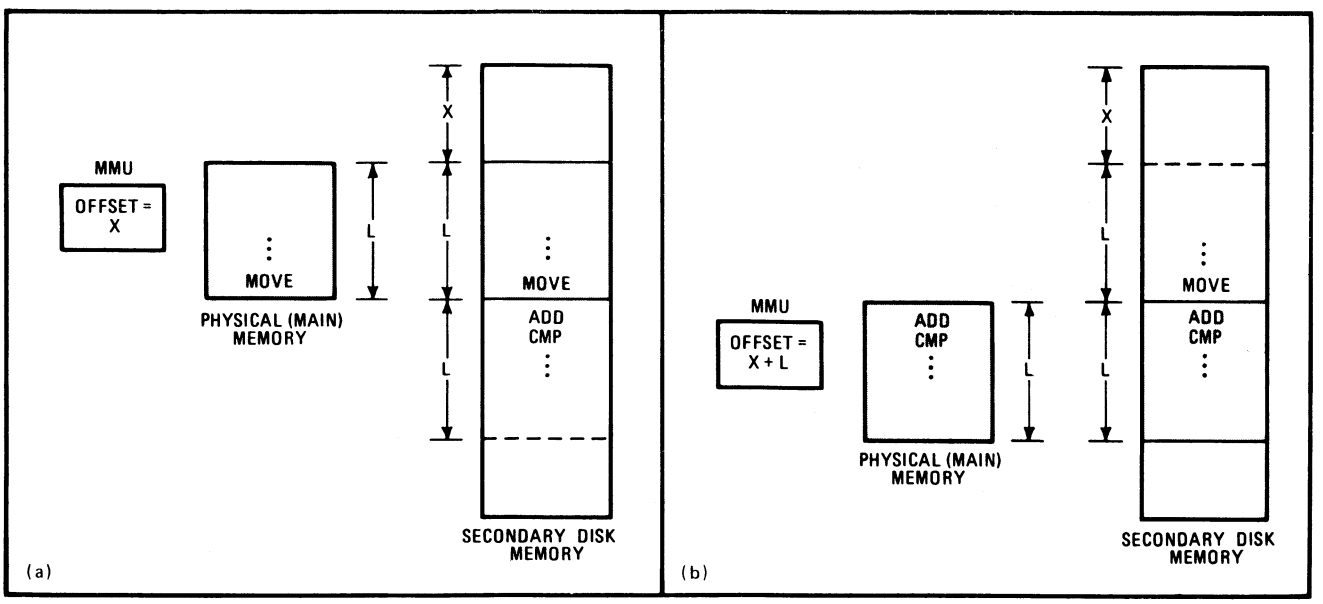
Permanent storage and temporary random-access memory need to be finely balanced to make a system both cost-efficient and operationally sound. Virtual memory combines primary memory—the semiconductor memory to which the processor has direct access—with a secondary storage unit, such as a tape, drum, or hard or floppy disk, to yield a low-cost-per-bit increase in the system's apparent memory. In practice, virtual memory lets a user write a program that is too large to fit comfortably in the available semiconductor memory.

A good way to determine what information the primary should contain is to allow the natural flow of instructions and references to data to feed primary memory as the executing program demands. This method takes the burden from the application programmer and places it on the processor supported by the operating system and is ideal for virtual memory systems.

Managing data intelligently

Because instructions normally follow in sequence, a program usually executes within a relatively narrow window around the current instruction. That is, if instruction n is executing, chances are that the next instruction to execute is fairly close by, and most likely is n+1. Therefore, if the primary always contains the current instruction and the sections of code immediately before and after it, chances are good that the processor will continue to execute the instructions out of primary memory: the primary's contents are filled with information that falls within a short range of the present instruction.

The MC68010, however, easily manages situations that



1. New page. When the data to be accessed is outside the primary memory's physical boundaries, the correct information block must be copied from secondary memory. To access the ADD instruction (a), a new section must be moved from secondary into main memory (b).

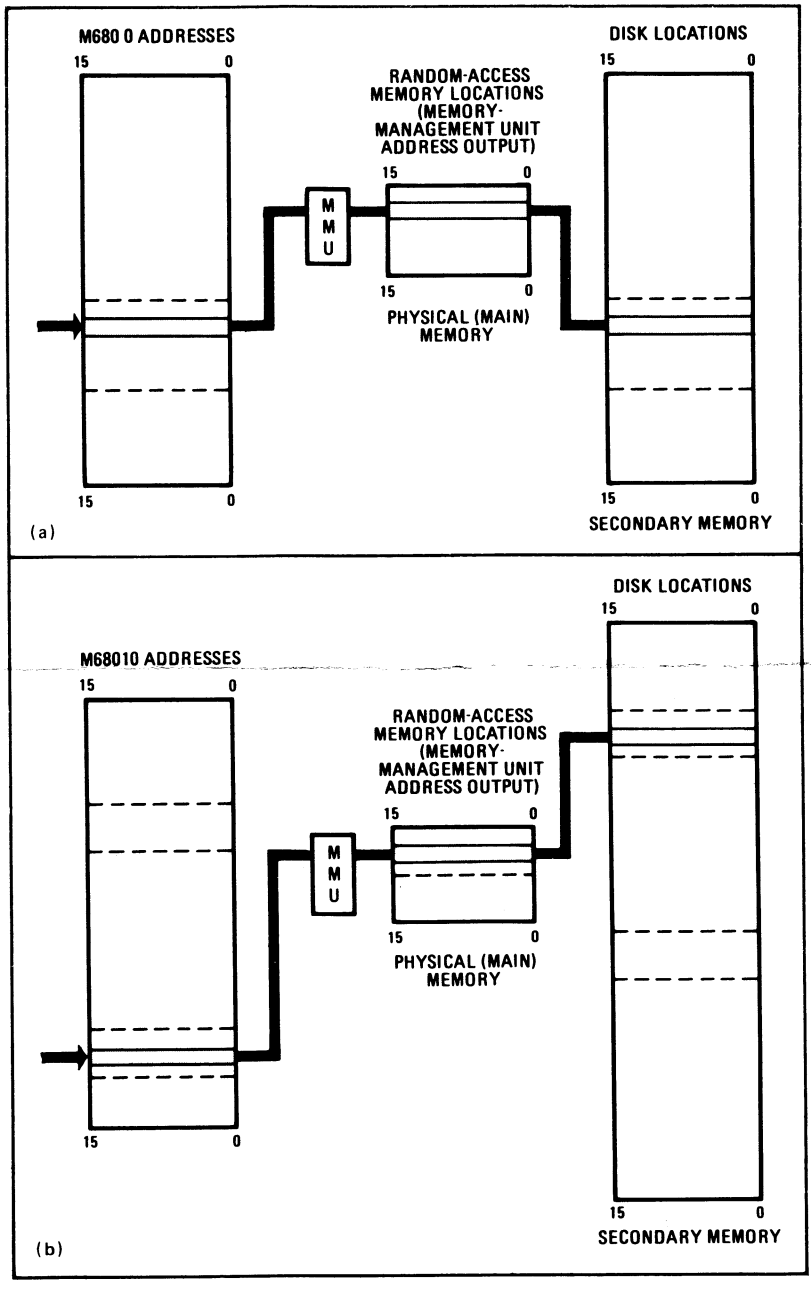
Schemes for managing virtual memory

A number of virtual-memory schemes have been tried previously in microprocessor-based systems; most of them turned out to be clumsy and inefficient, putting additional burdens on either the programmer or the microprocessor, or both. The processes basic to most schemes, however, sound very simple. Typically, (see part a of the figure), when the central processing unit attempts to access a memory location outside the physical memory, a bus-fault signal notifies the processor that something is amiss. At this point, the operating system must determine which block of secondary memory to copy into primary memory. Once the operating system has shifted the proper block of memory and updated the memory manager, the CPU can continue to access the needed information from the physical memory.

A more practical system breaks up memory into more defined pieces and keeps many more active blocks in primary memory (see part b of the figure). One block might contain instructions, another necessary data.

Some CPUs segment memory in an attempt to manage memory use efficiently, but this introduces more problems than it solves. The operating system must determine the primary's best contents. This may be based on the type or sections of code that are expected to be executed and the data that the code needs.

It is the programmer who must provide this information to the operating system each time a portion of code executes, which forces the programmer to match the program's memory requirements to the arrangement of physical memory. This could cause great problems if all the needed segments to be accessed exceeds the primary memory's physical bounds. Segment constraints and size limitations thus restrict the freedom of any of these segments to grow, especially dynamically, and the programmer must be constantly aware of the segment's boundaries. He or she is better off if such considerations are unnecessary.



require an instruction beyond the reach of the primary memory's current contents or require data from a section of memory not in the vicinity of the data recently accessed. In a simplified example (Fig. 1a), if primary memory is L words long and contains an L-sized copy of secondary memory, and the memory block begins at X in the secondary, the primary memory's contents may be described as being bounded by X and X+L.

The central procesing unit knows the primary only as 0 through L. When the processor accesses the memory address at L+1 (ADD), no information can come directly from primary memory. But in order to access this word, which is really at X+L+1, the block of information surrounding X+L+1 must first be copied into the primary. A convenient block may be between X+L and X+L+L, which is the next L-sized block in secondary

memory. This block is now copied into primary memory, enabling the processor to fetch not only the original address at L+1 (ADD) but also the next address at L+2 (CMP) from the primary (Fig. 1b).

In reality, of course, bringing a block of new information into primary memory may be more complex. The programmer must make sure that the data existing in primary memory is not overwritten and lost. To ensure that the old information is safe, the existing block in the primary might first have to be copied back into the secondary. The operating system determines whether secondary memory needs updating, and this action is facilitated by a modified bit in the memory manager. This bit indicates if any write operations have taken place in the memory block after it was moved into the primary.

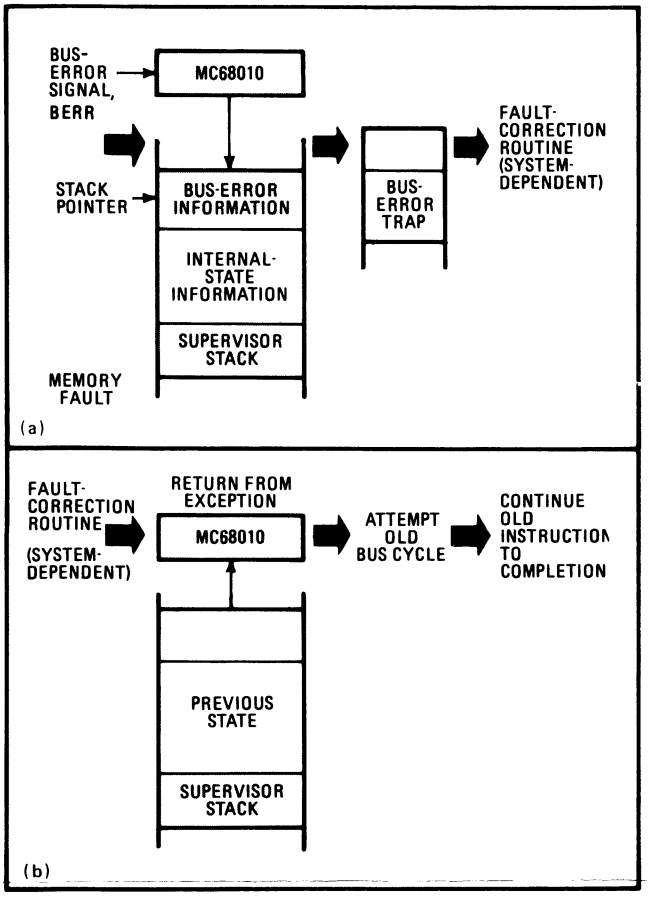
Bus-error recovery

The MC68010 uses instruction continuation to support virtual memory. Basically, this approach involves starting an instruction, flagging a fault to the CPU before the instruction ends, and suspending the instruction in an orderly fashion (Fig. 2a). It then provides a mechanism to allow the instruction to pick up where it left off once the fault is corrected and continue through completion. To do this, when a page fault occurs, the processor stores its internal state; after the page fault is repaired, it restores that internal state and continues to execute the instruction. Though this technique is difficult to implement in random-logic CPUs, it is relatively simple with a microcoded processor.

Instruction continuation appears to be best for the user and frees the application programmer from the restrictions of other schemes. An alternative is instruction restart. Here, the processor must remember the system's exact state before each instruction starts in order to restore the stage if a page fault occurs during execution. In addition, no memory fault is fully recoverable.

To implement intruction continuation, the MC68010 uses a bus-error (BERR) signal—which can cause any and all instructions to abort before even a bus cycle is completed—to warn the processor that there is a problem with the bus cycle in progress. When the MC68010 receives a bus-error signal, it aborts the bus cycle and begins a special internal procedure to suspend the instruction (as opposed to forgetting it, as in instruction restart). It now goes to the supervisor stack, where the operating system works, and places information that will help the operating system determine the cause of the fault on the stack. This information includes the logical address, control information associated with that address, the instruction being executed, the status register, and the program-counter values when the fault was signaled. The operating system can then inspect it to determine the necessary action to correct the bus fault.

In a microcoded machine such as the MC68010, a microinstruction specifies every internal operation associated with any instruction. The microprocessor also has numerous unseen registers, latches, and bits—all of which have information that guide the CPU's operation. The MC68010 also saves all this information, which is needed in order to continue an instruction, on the supervisor stack.



2. Recovery. Using an instruction-continuation technique, Motorola's MC68010 receives a bus-error signal (BERR) when a page fault occurs (a) and then aborts the instruction. It saves all the information on the supervisor stack so that when the CPU returns from the correction routine, the return-from-exception (RTE) instruction reruns the faulted bus cycle and the instruction continues to execute.

The CPU gets external information, such as the type of error from the memory-management unit. If the system uses an error-detecting circuit, it might be polled to determine the nature of the error it found on the data bus. The operating system examines this information to help determine the fault's cause.

As shown in Fig. 2b, when the CPU returns from the correction routine, the return-from-exception (RTE) instruction reloads the MC68010 with the internal state stored on the stack, reruns the faulted bus cycle, and then continues the suspended instruction.

In a worst-case example, neither the operating code nor its source or destination operand are in primary memory when an instruction begins. First, an op-code fetch shows that the instruction is not in the primary. One bus fault and fault-correction routine later, however, the processor re-executes and finds the op code. But trying to fetch the source operand results in a second bus error. Another bus-error routine and the source operand becomes available. A write attempt to the destination location results in yet another bus fault. Again, fault correction can get the destination into primary memory and, after recovery, the processor completes the instruction. □

MOTOROLA ***Semiconductor Products Inc.***

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721 • A SUBSIDIARY OF MOTOROLA INC.

A17767-2 PRINTED IN USA 4-85 IMPERIAL LITHO C30283 5,000

AR214