Virtual Memory for Microcomputers

Four New Memory-Management Chips Pave the Way

Stephen Schmitt
2890 Sandhill Rd.
Mason, MI 48854

Not too long ago, a microcomputer system with 32K bytes of memory was considered a luxury. Because memory was very expensive, you took great pains to squeeze, pack, and cram programs into the small amount of memory that you could afford. Today, however, you can buy 256K bytes for less than $500. And new 16- and 32-bit microcomputers feature directly addressable storage spaces that are 100 to 10,000 times larger than those found in 8-bit architectures. Like the pauper who just became rich, how do you handle this vast wealth?

Another drastic change in the microcomputer world deals with software. Multitasking operating systems, high-level languages, and flexible business software have become popular. The problem is now more complex: How do we take advantage of both the increased hardware power *and* the new complex software?

Virtual-memory techniques offer one answer. Virtual memory is an automatic system for controlling very big memories. But special hardware functions are essential for building such a system. And now, single-chip memory-management units (MMUs) have been developed to provide these capabilities for microcomputers.

In the first part of this article, I will introduce some of the basic concepts of virtual memory. Next, I'll compare and evaluate four MMU chips that have recently become available: Intel's iAPX 286, Motorola's MC68451, and Zilog's Z8010 and Z8015. [This survey does not include the National Semiconductor NS16082 MMU for the NS16032 microprocessor. Because of its fairly recent introduction, the part was not evaluated for the review. The NS16082 is another interesting MMU that merits analysis.] Finally, I will discuss some implications and applications of virtual memory in microcomputer systems.

Virtual memory is a powerful concept. It allows you to consider main memory to be very large—much larger than its actual physical size.

Program Folding

Almost every computer system has several types of memory devices that differ in speed and storage capacity. A fundamental tenet of computer technology states that memory price is directly related to its speed. Storage hierarchies thus usually represent an effective compromise between a large, slow, inexpensive memory and a small, expensive one with high access speed. Familiar examples of this are systems with relatively small amounts of fast RAM (random-access read/write memory) and larger, slower, and cheaper disk-storage devices.

Although the cost benefit of such a configuration is substantial, the efficient management of this structure presents a challenge. The movement of data between these two hierarchy levels should be minimal; otherwise, the access time for the slow memory will predominate over the speed of the fast memory. In a typical two-level system, main memory (MM) is

About the Author

Stephen Schmitt has worked for Hewlett-Packard and also taught at Michigan Technological University. He is now doing a review of a version of the Ada programming language for microcomputers.

1000 to 5000 times faster than the magnetic disks used for secondary storage. Thus, disk accesses should be as infrequent as possible if we are to take advantage of the high speed of the main memory.

Another problem is that the relatively small size of the main memory in most systems limits program size. An excessively large program must be broken into parts; each piece is loaded into main memory prior to its processing turn and returned to disk after execution. This technique is called *folding* or *overlaying*, and the task of folding programs is usually a job for the programmer. The problem is that the mechanics of defining separate program parts and adding code to control data transfer between main and secondary memory are usually cumbersome tasks. Also, the additional folding code clouds program logic. Compilers and linkers can simplify the task, but you must still design the overlay framework.

Despite these difficulties, however, folding operations are common. In fact, the word processor I'm using to write this article applies the concept twice. First, the program is too big to fit completely into memory and is divided into three overlays. Second, only a portion of my large text file resides in memory at a given time—the rest is stored on disk.

As you can imagine, manual folding consumes considerable time and effort (as much as 25 to 40 percent of programming costs). But there is a way we can take advantage of the benefits of large, sophisticated programs without spending a tremendous amount of time manually folding them to fit into small memory spaces.

Virtual Memory: Definition

Computer facilities that automatically fold programs and data between two or more memory levels are called virtual-memory systems. Virtual memory is a powerful concept. With it, you can consider main memory to be very large, much larger than its actual physical size. Intermediate files, overlays, and many file-access procedures are no longer necessary. Program logic is simpler and is focused on problem solutions, not critical resource management.

The objective of virtual memory is straightforward: to permit programs with very large address spaces to run at MM speeds. In virtual systems, main memory serves as a window (or group of windows) onto the entire address space held in secondary memory. If the window is big enough, and if it accurately reflects the active part of total memory, the technique works extremely well.

The reason for this is that programs tend to access small portions of memory over fairly long periods of computer time. This is called *clustering* or *locality of reference*. Code loops and manipulations of a data structure are examples of clusters or programs with good locality. A virtual system must detect and maintain in main memory only the *working set* of a program, that is, those locations with high activity. As activity gradually shifts to other memory regions, these areas of secondary storage are automatically accessed and brought into main memory. As you might imagine, a high rate of secondary-storage access will severely degrade performance. This is known as *thrashing*.

Address space in a typical virtual system ranges from 16 megabytes to 64 gigabytes, enough to handle very ambitious programming projects.

Benefits

Let's examine the benefits of virtual-memory management. Foremost, it removes the limit on program size imposed by main-memory size. Address space in a typical virtual system ranges from 16 megabytes to 64 gigabytes—large enough, I dare say, to handle even the most ambitious programming project! And virtual systems offer other advantages:

- Main memory is allocated automatically according to the demands made by a program. The user does not have to estimate memory allocation prior to execution.
- Manual folding is eliminated and replaced, in part, by high-speed hardware. Thus, programming costs tend to be lower.
- Programs execute correctly regardless of actual main-memory size. But note that the execution speed may be affected if the fast store is too small to meet average memory requirements.
- Relocation and task switching are enhanced indirectly.
- Multiprogramming environments have greater flexibility. The problem of deciding the optimal placement of programs in a fixed-size memory is reduced. More programs can execute concurrently because only the active portion of each occupies main memory. While this may induce less efficient use of the total addressable space, more effective use of the main memory is achieved.

Having outlined the motivation for virtual storage, we are ready to explore basic components of system design. The memory-control process must be transparent to normal operation and relatively efficient. On face value, you might doubt if reasonable performance is possible, but research into virtual-memory behavior clearly demonstrates the concept's potential (see reference 2). I hope to show that some practical systems can also have a remarkably simple design.

Virtual-Memory Design: Basic Concepts

Virtual storage systems require a mixture of specialized hardware- and software-control policies. I will focus on architectural features that influence virtual-memory operation. An understanding of intended applications should aid our analysis of MMU products later.

A computer's *address space* (AS) is the legal range of addresses that can be generated by its instruction set. The maximum size of this is determined by the number of bits in the processor's address register. A *logical address* is a memory specification used by the central processor. *Physical addresses*, on the other hand,

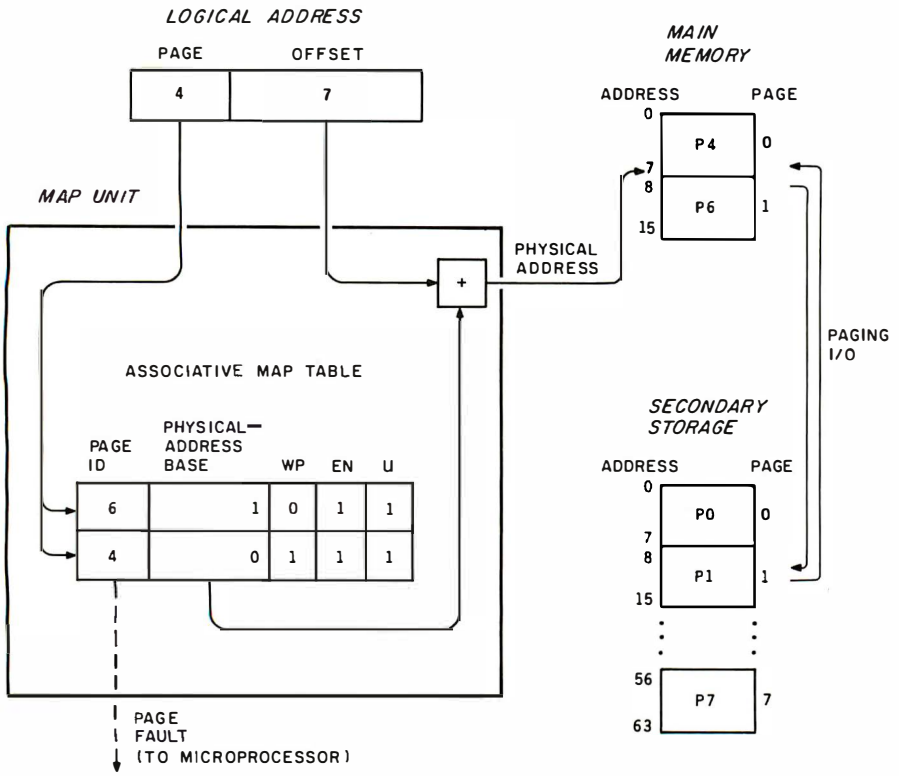

Figure 1: *The associative mapping scheme for virtual-memory systems. The central processing unit specifies a memory location with a two-part logical address that includes a page field and an offset field. The page field provides the value used for searching the map table. All cells in the table are searched in parallel. If a match is found, the selected cell's physical-address-base field is added to the offset field. A page fault occurs when there is no match. In this simple diagram, the second cell matches and translates the logical address. The attribute fields in the above map table are WP (write-protected), EN (valid main-memory page; EN = 0 only when main memory is not full), and U (used; recent page access has occurred).*

describe actual locations in the main memory. In most systems, logical and physical addresses are one and the same (as, for example, in 8-bit microcomputers). With a virtual system, however, the size of the AS can be significantly larger than main memory. The AS may be thought of as occupying a contiguous area in secondary storage; and logical addresses no longer correspond exactly to actual physical RAM locations.

In a virtual system, main memory contains changing portions of AS. At various times, an instruction may address a part of the AS that is not contained in main memory. This is known as a *page fault*. A virtual-memory system must be able to detect a page fault and move the desired part of the AS into main memory. Then, when that part is subsequently addressed by an instruction, the system must be able to translate that part's logical address into its present physical address in main memory. This is known as the *mapping process*. Both these processes, page-fault detection and mapping, are fundamental to every instruction step. They must therefore be performed by high-speed hardware.

A page fault stops execution of the current activity that the central processor is performing (e.g., fetching an instruction or processing operand data) until the absent memory is brought into MM. A page fault is similar to an interrupt except that it may occur partway through instruction processing. Thus, special processing logic is needed to handle partially executed instructions (consider the problems associated with restarting a MOVE BLOCK instruction).

In a virtual system, memory can be

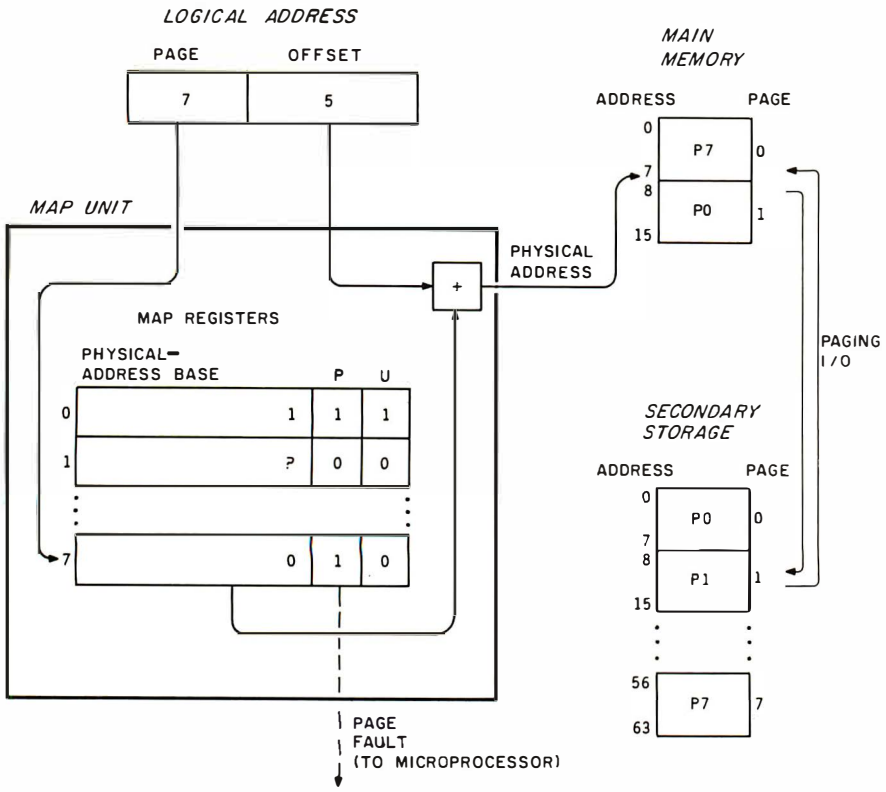

Figure 2: *The mapping-by-address (or mapping-by-register, MBR) scheme for virtual-memory systems. This is similar to associative mapping except that each map cell is a register that refers to a page in secondary storage. In this simple example, the page field in the logical address refers to register 7, which in turn refers to page 0 in main memory. If register 1 had been accessed instead, a page fault would have occurred (P = 0). Page 1 would then be placed into main memory, probably in the space now occupied by page 7, because page 7 has not been accessed recently (U = 0).*

divided into either *pages* or *segments*. A paged policy divides memory (both AS and MM) into equal-size blocks. The rationale for pages relates to the clustering principle. Memory activity occurs in scattered parts or clusters of the AS. By organizing storage into pages, you can "break out" the busy sections and place them in main memory. Paging, like disk blocking, also implies a smaller number of data transfers between disk and main memory.

Segments are merely pages of variable size. Segments can closely model program units because code modules and data structures vary in size (as do clusters). Trade-offs exist between page and segment organizations. I'll discuss these later. For now, you can ignore the distinction and call both pages.

Several mapping schemes exist for virtual systems. Figures 1 through 3 illustrate three common techniques. In the examples, logical-address fields are composed of two parts: a page field and an offset field. The logical page number is translated into a physical location by the map unit. Adding the offset field to this location forms the complete physical address. A simplistic memory model will be used to show the basic operations of each technique.

Associative mapping is shown in figure 1. The logical-address page field is compared, in parallel, to all page entries in the map table. If an entry matches, its corresponding physical-page address is combined with the offset value to form the complete physical address. A page-fault condition is raised when no match occurs. The problem with associative maps is that they are expensive. High-speed register memory with integrated comparative logic is needed; translation has to be fast and transparent. The associated map has to be as large as

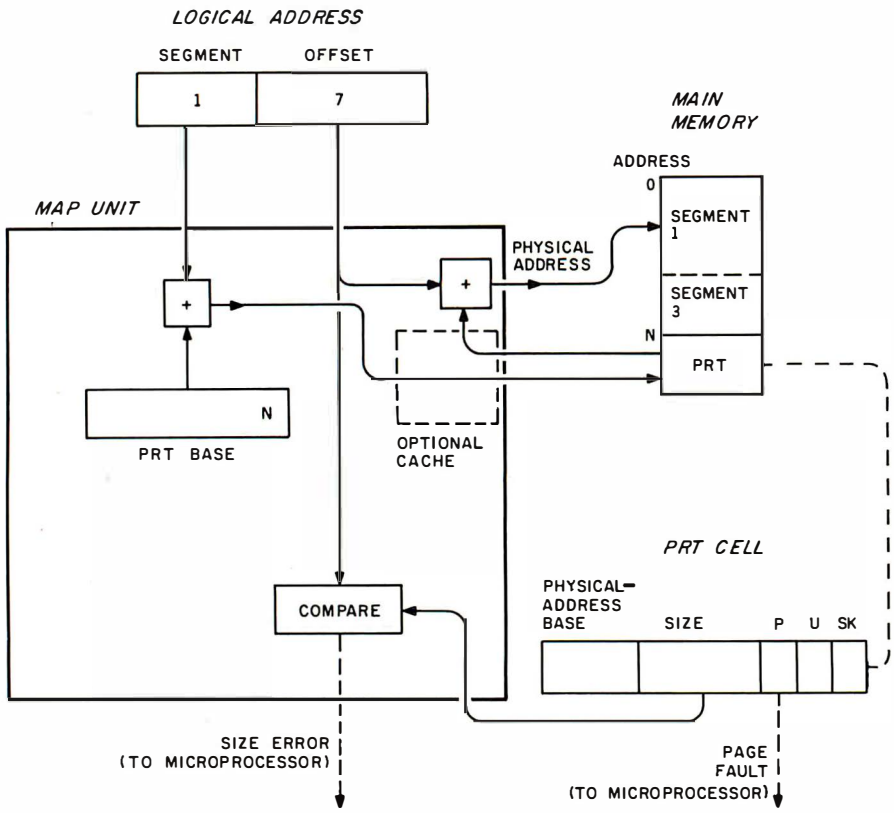

Figure 3: *The segment map table scheme for segmented virtual-memory systems. In this mapping scheme, information on each segment of a program in secondary storage is kept in a program reference table (PRT) in main memory. The location of the PRT is stored in a PRT Base register. In this simple example, the segment field in the logical address is added to the contents of the PRT Base register (N). This refers to a map cell at location N + 1. In this map cell is a physical-address base that is added to the offset to obtain the desired address in main memory. Note that bounds-checking can easily be done by comparing the offset with the size field. Also, note that things can be speeded up by placing the most active map cells in a small associative cache memory. The attribute field SK indicates a stack segment (i.e., the offset orientation is reversed).*

the number of MM page frames. If you change the size of main memory, you have to change the map size accordingly. Associative mapping works best for systems with a large number of AS pages and a moderate-sized, fixed main store.

Figure 2 illustrates *mapping by address* or mapping by register (MBR). In this technique, the logical-address page field refers to an array of high-speed registers. These registers hold status and physical-address information. Mapping by address is analogous to indirect memory addressing except that the registers permit very high processing speeds. Page faults are detected when the addressed register's "present" bit is clear. The problem is that every page in the AS requires a corresponding map register. Fortunately, the economy of conventional registers offsets the mapping array size. Note that the mapping hardware is unaffected by changes in MM size. For relatively small address spaces, mapping by address is quite attractive.

The last technique I'll present applies to segmented systems. Figure 3 defines the operational details. This design is based on the Burroughs Corporation B5500 mainframe. This approach gives you more flexibility, but is slightly more complex and necessitates additional hardware (for the

Field	Abbreviation	Description	Field Size (bits)	Memory Structure (Segment vs Page)
Present	P	Indicates if page is present in main memory	1	Both
Used	U	Indicates if page has been recently accessed	1	Both
Dirty	D	Indicates if page has been modified	1	Fixed-page
Type		Various page properties	1–4	
	RP	Read-protected		Both
	WP	Write-protected		Both
	EX	Executable code		Both
	SH	Shared		Segment
	SK	Stack memory		Segment
	OW	Overflow warning		Both
Size	SZ	Size of segment	16–32	Segment
Priority or mode	PR	Indicates task priority or system context to permit access	1–8	Both
Virtual time	LREF	Time of last access to page	12–32	Both
Task ID	TID	Identifies task(s) that owns the segment	8–16	Segment
Fix	F	Indicates that page is not to be replaced	1	Both
I/O access	IO	Protect or hold page for input/output	1	Both
Enable	EN	Indicates valid pages for access (when main memory is not full)	1	Both

Table 1: *A summary of the control and status information used by virtual-memory mapping units. Most mapping schemes use a subset of these different attribute fields.*

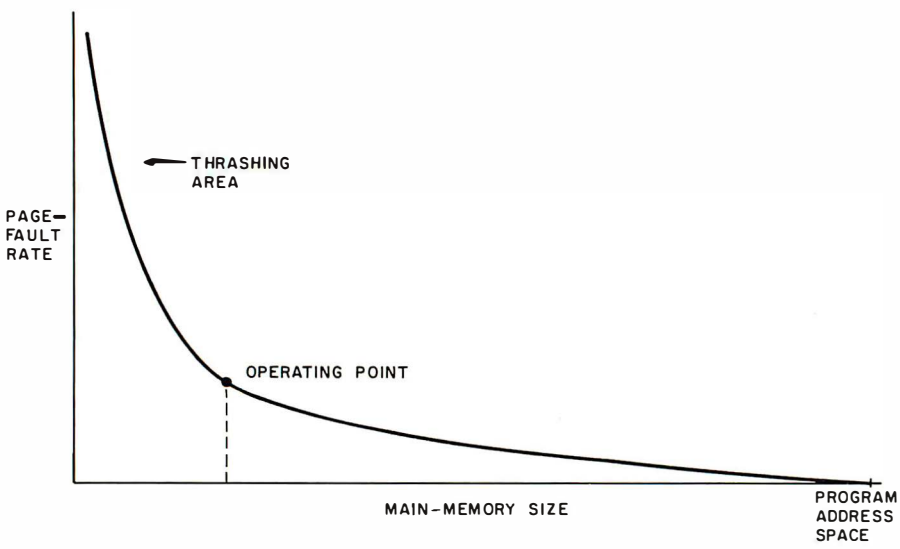

Figure 4: *A graph showing how the page-fault rate (i.e., the rate of accesses to pages not present in main memory) is related to the size of main memory. The operating point is the memory size sufficient to hold a program's most frequently accessed routines—its working set. Adding memory past this point has little effect on the page-fault rate. Of course, as the needs of a program change, the operating point will shift.*

segment-size field and the *program reference table* or PRT). A segment field replaces the logical-address page field. You can have any number of system segments; MM and AS sizes do not constrain the choice. Translation is similar to mapping by address. The program reference table contains the size and location for each segment. It resides in main memory for two reasons: (1) to allow a large number of segments, and (2) to avoid the high cost of registers that would be wide enough to hold extra mapping data. The table acts like mapping registers, with the segment field—not the page field—providing the index address. At first glance, it looks as if the scheme is twice as slow as the others because every reference needs two memory accesses (one for the PRT and one for the actual data). But this problem can be handled by putting a small buffer-register set in the MMU. Keeping copies of the most active PRT entries in this high-speed buffer greatly increases mapping speed.

Besides translating addresses, mapping units also provide other functions. They hold information to aid memory management and data protection. Table 1 is a list of information found in various mapping systems. Each virtual system uses a subset of these items, determined by the particular mapping scheme used and the memory-control functions.

Up to now, attention has centered on the hardware aspects of virtual systems. Now, let's consider the memory-management and software requirements for virtual systems.

Paging Policies

As mentioned previously, a minimal amount of secondary-storage access is central to a virtual system's viability. Figure 4 shows a graph of disk activity versus main-memory size allocated to a program. From the graph, we see that given enough main memory, disk access approaches zero. However, a primary aim of virtual memory is to provide a huge address space while minimizing expensive main memory. To satisfy both conditions, you must operate at a point just below the "knee" of the

curve (labeled operating point). The locality principle states that the amount of MM needed at any given time is but a small portion of AS. Hence, if you keep the active cluster or working set in memory, thrashing and main-memory needs are minimized.

Sounds simple, doesn't it? Alas, a few minor problems "gum up" matters. For openers, measuring a system's working set is a dynamic process. The size and contents of a working set change rapidly. Keeping track of working sets involves considerable time, resources, and problems. Just how is this working set determined? At what times do we change the working set to reflect locality movement? What happens when several programs are running or there is program I/O? All these problems are handled by a *paging policy*.

Basically, a paging policy does three things:

- Fetching—decides when to transfer pages from secondary storage to main memory
- Placement—determines which MM page frame should hold the fetched page
- Replacement—when main memory is full, chooses which MM page frame should be replaced by the fetched page

In regard to fetching, research has found that demand paging is generally best. When a page fault takes place, you fetch the desired page from secondary storage. The placement decision is resolved automatically by mapping hardware. The last issue, however, choosing which page to replace, is the hard part. The replacement policy affects how well we address the other concerns mentioned above.

Page-replacement techniques, which determine the set of main-memory pages, are all approximation algorithms. This is so because you can't calculate the best page to remove without some future knowledge of which page will be required. The optimal algorithm (OPT), or Belady's algorithm, replaces the active page that is next referenced the furthest into the future. Even though totally impractical, it is a benchmark for comparing other techniques.

Page-Replacement Algorithms

Virtual-system performance is very much dependent on the page-replacement technique that is used. Because the process selects departing pages, it indirectly determines the pages remaining in main memory. If the algorithm closely models a system's actual working-set memory demand, few page swaps will result. Algorithms usually base removal choices on prior reference activity, because the locality principle implies that past behavior approximates future needs (at least over short time periods).

Here I will discuss four specific page-replacement algorithms: Least recently used (LRU), Clock, Generalized working set (WS), and WS-Clock.

The LRU Algorithm

This page-replacement policy is conceptually related to the optimal algorithm. Instead of selecting the page with the furthest time until next access, you pick the page whose last reference occurred longest ago. If not used for a long time, the probability that the page will soon be referenced is small. When a page fault happens, you scan the map cell for each page and replace the page having the smallest (oldest) virtual time.

To implement LRU, the memory-management hardware must support two features: a virtual time register in the map unit and the ability to update the page access time during address translation. The time register should be wide enough to ensure sufficient resolution. In addition, the necessity of associating a time stamp with every memory reference dictates high-speed logic and added map complexity.

The LRU algorithm works well. Its performance is much better than that of an arbitrary replacement policy or many other paging policies. But because LRU is a *global policy*, it can exhibit anomalies in multitasking systems. For example, global LRU tends to save pages of the task last executed

and favors jobs with smaller locality; low-priority tasks and large programs may experience reduced throughput.

Computer systems using LRU include the CDC Star-100 and the Multics drum-to-disk control. Microcomputer MMU parts lack the hardware mechanisms needed for a "pure" LRU policy.

The Clock Algorithm

This algorithm is a variation of the LRU algorithm. Main-memory pages are logically ordered in a circular list. You can envision each page as a unit marking of a clock face. A pointer or hand always points to the last page replaced. On a page fault, you advance the pointer clockwise to the succeeding page. Then you check and clear that page's *used* bit. If the bit was set (i.e., the page was used recently), scanning continues; otherwise, the frame is not recently used and replaceable. If the replaceable page has been changed (or *dirty*), you must schedule it for transfer back to secondary memory. Scanning stops when a clean, not recently used page is found, and the pointer is left at the chosen page. A replaceable page is not processed if accessed before the disk transfer.

Studies indicate that the Clock algorithm closely simulates LRU replacement, and the hardware needed is inexpensive. As implied above, only 2 flag bits per cell are required (changed and used). Software complexity and overhead are small. Calculations are trivial and the average number of scans per page fault is a fraction of total map size. Many successful mainframe systems, including the IBM 370 and Multics, use Clock algorithms. However, you should note that deficiencies of LRU apply equally to Clock. The technique offers a simple mechanism and good efficiency; but, as you shall see, other paging algorithms exhibit even better performance characteristics.

The WS Algorithm

This page-replacement algorithm represents the most practical policy according to empirical studies. WS is so named because it approximates the working-set locality model. In WS, any page referenced within a specified time (designated as Θ) is regarded as a member of the working set. Real working sets of course have variable durations, but if the WS time-control value (Θ) is properly chosen, a real working-set model can be closely approximated.

I will briefly highlight WS operation (see reference 1 for details). The WS policy defines a working set (W) to be those pages of the AS that have been referenced within the previous Θ time units. In order to determine when a page (p) in main memory is no longer in W, and thus is replaceable, we need two things: (1) a procedure to calculate a time value (L) equal to the owning task's current execution time (ET) minus the last reference time for every AS page, LREF(p), and (2) a scan mechanism to check for values of L greater than or equal to Θ. Calculating L can be done with page-frame counter registers. When the page is accessed, its counter register is cleared. Then, at fixed intervals, a global broadcast pulse increments all the counters. The scan operation can run at various times (e.g., at fixed intervals or when a page fault occurs). Pages marked as replaceable become part of the available pool (AP). The page-replacement algorithm merely selects some page from the AP and replaces it. If the AP is empty, the system must suspend a task to free pages.

Although WS accurately models dynamic-memory demands, the computational overhead and extra hardware support it requires diminish the algorithm's viability. Space for the LREF(p) field can effectively double page-table size. Moreover, the counter mechanism is relatively expensive. Scanning requires inspection of each map cell at regular intervals, and AP maintenance adds more control functions. On the plus side, the local scope of WS enforces more consistent multitasking management. And pure WS simulations perform better than other policies. Research systems have implemented practical WS schemes and observed substantial

improvements over Clock techniques. The parameter Θ allows you to "tune" a virtual system for different applications. Also, investigations have found that use of the constant value Θ deviates less than 10 percent from an optimal WS.

WS is technically appealing, but design difficulties detract from its advantages. Like LRU, classical WS is impractical for microcomputer application. But the next technique surveyed approaches WS performance and is feasible for microcomputers.

The WS-Clock Algorithm

WS-Clock combines the best properties of Clock and WS. Additionally, the new strengths offset some problems of the separate procedures:

- the extra scanning required by WS is replaced by a simple Clock mechanism
- WS-Clock is a local policy
- LREF(p) registers are needed only for main-memory pages, not for every AS page
- the available pool (AP) is eliminated

This algorithm organizes page frames in a circular list like Clock. The clock pointer identifies the page replaced during the last scan. When a page fault occurs, the scan advances clockwise to the next page. The used bit is checked and cleared. If the bit was set, you reset the page's LREF(p) to the owning task's accumulated execution time (ET). Otherwise, if the used bit is clear and if $L = ET - LREF(p) >= \Theta$, you remove the page from W. If dirty, a replaceable page is scheduled for disk transfer and not replaced. Scanning halts when you encounter a clean, replaceable page.

WS-Clock approximates WS replacement, and W for the two policies becomes equivalent when the task executes for Θ units of time. Performance differences appear negligible and can be ignored. Thus, WS-Clock approaches WS behavior with a significantly simpler mechanism. The average number of frames examined per page fault compares favorably with WS. And to implement WS-Clock, you need minimal map hardware: a used bit, a dirty bit, an LREF field, and a task ID descriptor.

Multitasking and Load Control

In a multitasking system, virtual-memory management must coincide with general resource-sharing policies. The problem is that in a dynamic multiprocessing environment, wide variations in programming level and memory demand occur. Every active task consumes a portion of total memory (in both MM and AS). At some point, adding another task will push MM demand past the ideal operating point in figure 4 and trigger the onset of thrashing. Left unchecked, throughput diminishes rapidly. Accordingly, you must either prevent the "overcommitment" of memory or recognize the condition and make corrective adjustments. This is done by a load-control policy.

Function or Characteristic	Memory Structure: Segments	Memory Structure: Fixed Pages
Flexibility	+ Can model pages, has more design options, promotes protection, data sharing, and so on	− Lessens "tuning" options, trade-off exists between efficient I/O blocking and memory waste
Hardware requirements	− Cost and complexity fairly high	+ Lower cost, simpler hardware
VS(1) systems	+ Better security	+ Simpler design
VS(n) systems	+ Segments are mandatory for VS(n) designs	0 Not applicable
Mapping unit		
Map-register size	− More space needed for size fields	+ Minimal
Table size	+ Fewer frames needed if segments are large	0 Larger pages result in memory waste
Memory management overhead	− Extra logic needed to close "holes" and to manage extra attributes	+ Placement simple
Allocation policy	+ Make segments the desired size	0 More complex, but no main-memory waste
Protection	+ Superior—facilitated by extra attribute fields	− Pages do not correspond to program "objects"
Sharing	+ Direct support	− Much harder
I/O efficiency	0 Depends on segment size	+ Facilitated by relating page size to disk block size
Memory usage		
Internal fragmentation	+ None	− Some main memory wasted
External fragmentation	0 Can reclaim with "garbage collection"	+ None
Page-fault rate	+ Lower rate	0 Depends on page size

Table 2: *A comparison of the relative strengths and weaknesses of segmented and fixed-page memory organizations. The +/− column indicates approximate merit: (+) good; (−) poor; (0) okay or does not apply.*

Load control is sensitive to page-replacement strategy. Local strategies estimate each task's independent memory needs and allocate sufficient main storage to hold the locality set. Global page-replacement strategies discriminate in favor of the most recent task's memory set and can lead to thrashing.

System Design: Issues and Options

Virtual memory reflects a composite of hardware, resource management, and programming processes. We now turn our attention to alternatives that can affect overall microcomputer system design.

VS(1)

A fundamental system decision is whether you treat the large virtual address space as a shared resource divided among the several active jobs or whether each process is provided a separate AS. The first class, termed VS(1), extends the idea of a conventional operating system where supervisor, system resources, and user tasks occupy one large address space. Software compatibility with nonvirtual systems is a major benefit of this system. System complexity is minimized and a single mapping table can define AS structure.

VS(n)

VS(n) systems give each executing task a unique AS. To support this feature, every job has its own mapping table. Typically, a mapping-table origin register (MTOR) points to the mapping table of a running task. When switching tasks, you change the MTOR and reload the mapping hardware. A VS(n) system gives you, in effect, several virtual machines, each using the same physical resources, running concurrently. VS(n) systems also give you improved system integrity and data protection.

Page Size

An important design consideration is block memory structure. If you select a fixed-page structure, you must determine the number and size of page frames. Secondary storage transfers data in fixed-size units. Hence, for efficient paging memory frames should be an integer multiple of a disk block. Big pages reduce disk overhead and map hardware. On the other side of the coin, however, a large number of small pages lowers page-fault rates and increases the number of locality sets. Some compromise is in order. My research indicates that 1K- to 4K-byte pages are considered optimal.

A segmented address space reflects programming features such as scope rules, data encapsulation, modularity, and so on. Pages, being constant in size, usually waste some memory, a condition termed *internal fragmentation* (e.g., a 5K-byte program takes two pages in a system with 4K-byte pages—3K bytes are unused). Although segments avoid this problem, they are prey to a form of waste called *external fragmentation*. Because variable-size units are allocated, program termination leaves holes of unused space. You must close up these areas periodically to provide sufficient space for large segments. Consequently, the procedures to reclaim these holes add to operating overhead. In general, segmented schemes offer more flexible designs while page organizations make for easier memory management. Table 2 summarizes trade-offs between page and segment organizations.

Many other topics related to virtual memory have not been covered: operating-system interaction, I/O considerations, page locking for nonswappable memory, disk-access properties, and more—the subject is rather deep. However, the topics

FUNCTIONAL CHARACTERISTICS	MEMORY-MANAGEMENT UNITS Intel iAPX 286	Motorola MC68451	Zilog Z8010	Zilog Z8015/PMMU
Address-translation delay (10 MHz)	0–1.5 μs	100 ns	60 ns	70 ns
Supports multiple MMUs?	No	Yes	Yes	Yes
No. of MMUs needed to map address space	1	1	2	Depends on MM size
No. of unique address spaces possible (no. of users)	Unlimited	256	8	8
VS(n) support?	Yes	Partial	No	No
Priority levels	4	0	0	0
User/supervisor modes available?	No, uses priority levels	Yes	Yes	Yes
Data sharing?	Yes	Yes	Limited	No
MMU control method	Special instructions	I/O program	I/O program	I/O program
Fault restart data	None	Limited	Moderate	Extensive
Control and status attribute names (dash indicates not supported)				
Present	P	—	CPUI	—
Used	A	U	Ref	Ref
Dirty	—	M	Chg	Chg
Write-protected	W	WP	RD	RD
Read-protected	R	—	—	—
Executable code	E	—	Exc	Exc
Shared	—	AST	—	—
Stacked memory	ED	—	DIRW	DIRW
I/O access	—	—	—	DMAI
Overflow warning	—	—	DIRW	DIRW
Virtual time	LREF	—	—	—
Task ID	(Yes)	(Yes)	(Yes)	(Yes)
Fix	F	—	—	—
Enable	—	E	—	Valid

Table 3: *A comparison of the functional characteristics of the four surveyed memory-management units.*

covered should give you the perspective to analyze the capabilities of the new memory-management units for microcomputer virtual-memory systems.

Memory Management

The need for a memory-management unit (MMU) derives from two concerns: efficient control of large memories and support for multiprocessing environments. We can summarize the major goals of memory management as follows:

- Memory allocation—Allocation policies determine what portions of memory are committed to particular tasks. Address translation allows you to treat physically separate blocks as logically contiguous. Dynamic allocation, which adds memory during execution, is a valuable feature. A virtual system's large address space makes allocation less of a concern.
- Program relocation—Relocation hardware permits a program to load anywhere in physical memory without changing the logical addresses. Systems that swap tasks to disk may need to relocate a program when it's reloaded.
- Protection—This prevents inadvertent or unauthorized destruction of data. Also, one task cannot interfere with another's operation.
- Data sharing—Controlled access to common data or code.
- Multitasking—Several tasks can logically occupy main memory during a given time frame.

Virtual memory is just one of several approaches to memory management. Another approach is *dynamic mapping*, a technique used to extend the address space of limited-address machines (e.g., the HP-1000 or DEC PDP-11 minicomputers).

A Survey of MMU Chips

The main thing we'll look for when examining these MMU products is how well they implement virtual-memory concepts. We'll review four products: Zilog's Z8010 and Z8015, the Motorola MC68451, and Intel's iAPX 286 processor/MMU. The Zilog chips are to be used primarily with the Z8003 16-bit microprocessor for virtual storage. The MC68451 MMU is designed to work with the soon-to-be-released MC68010 processor. Finally, the iAPX 286 represents a combination of both an 8086-compatible processor and an integral memory-management unit.

The Z8010, MC68451, and iAPX 286 feature segmented-memory architectures. The Z8015, however, is designed specifically for a fixed-page virtual-memory system. All four units support a 16-megabyte physical-address space. Logical-address spaces range from 8 megabytes for the Zilog chips to a whopping 1 gigabyte in the iAPX 286. Tables 3 and 4 compare the basic properties of these chips on a point-by-point basis. As the tables show, there's quite a bit of diversity. For each MMU, I'll point out its unique characteristics, operation, and programming details. Later I'll describe some applications for virtual memory.

The Zilog Z8010

The Z8010 was one of the first single-chip MMU devices on the market. As a consequence, it has a few flaws that have been corrected on newer products. In fact, virtual memory appears to have been an afterthought for this chip because you will need extra hardware to handle the Z8000 microprocessor's page-fault procedure. Still, the product does have good protection features, and it directly supports a supervisor mode for operating-system functions. Another virtue is its fast translation time.

The Z8000 architecture defines logical addresses for 128 segments,

Text continued on page 234

PHYSICAL CHARACTERISTICS	MEMORY-MANAGEMENT UNITS Intel iAPX 286	Motorola MC68451	Zilog Z8010	Zilog Z8015/PMMU
No. of pins	68	64	48	48
Integrated processor?	Yes (8086)	No	No	No
Dimensions (mm)	24 by 24	22 by 81	15 by 51	15 by 51
IC process	HMOS	HMOS	NMOS	NMOS
Power (W)	3	1	1.5	1.5
Compatible processors				
Model	Integral	MC68010	Z8001/3	Z8003/4
Clock rates (MHz)	8, 10	4, 6, 8, 10	4, 6, 10	4, 6, 10
Cost per unit (10 MHz)	$237	$111	$383	$137
Logical addresses				
Virtual size	1 gigabyte	16 megabytes	8 megabytes	8 megabytes
Address width (bits)	32	24	23	23
Memory structure				
Segment or page?	Segment	Segment	Segment	Page
Size field (bits)	16 limit	16 mask	8 limit	NA
Size range	1 byte – 64K bytes	256 bytes – 16 megabytes	256 bytes – 64K bytes	2K bytes
Resolution	1 byte	Power of 2	256 blocks	Adjustable
Page boundary	1 byte	256 bytes	256 bytes	2K bytes
Map organization (mapping scheme)	Segment map table	Associative lookup	Mapped by address	Associative lookup
Maximum no. of pages or segments per MMU	16,384	32	64	64
Map cell width (bits)	64	72	32	32
Physical-address base field width (bits)	24	16	16	13
Attribute field (bits)	8	8	8	7
No. of high-speed registers	4	32	64	64
MMU address-generation unit	24-bit adder	16-bit logical	16-bit adder	13-bit concatenation
Global control-register set	1 byte (7 bits)	4 bytes	3 bytes	3 bytes
Global status-register set	0 (status pushed on stack for faults)	18 bytes	6 bytes	9 bytes

Table 4: *A comparison of the physical characteristics of the four surveyed memory-management units. All four have the same physical-memory limit: 16 megabytes.*

and the Z8010 has 64 map registers. If you use the mapping-by-address technique, you will need two MMUs to map AS. The user/supervisor flag can be used as an extra addressing bit to increase memory size to the full 16 megabytes. With this type of organization, four MMU chips are necessary (i.e., 128 segments in two separate address spaces).

You can assign four protection attributes: read-only, data/code, system reserved, and I/O enable. If you do sophisticated I/O processing, you'll appreciate the I/O flag. Another feature, the direction and warning (DIRW) attribute, indicates the orientation for a stack segment. When set, offsets can be negative. DIRW also provides a warning if you are accessing the last 256 bytes of the segment. The warning allows the system to dynamically extend segments during execution, making allocation procedures easier to implement.

Programming the MMU is accomplished through 22 special I/O instructions. By placing the MMU into command mode, you can manipulate map cells and global status registers in a manner similar to programming DMA or peripheral controllers. Z8000 instructions permit you to send a block of commands and data to speed up the process.

Provisions for virtual memory are marginal at best. Only three attribute flags aid paging policies: present, accessed, and changed. Up to eight separate users are possible, but this requires additional MMUs and external hardware. The Z8010's limited number of segments has two major drawbacks. First, it discourages small segment size. The segment-size resolution may not reflect program modularity (studies indicate that median module size is about 50 words). More important, the number of pages it can handle may be insufficient for working-set purposes. Although you can share data, utility is minimal. Many of the benefits of a segmented design are not fully realized.

The Clock page-replacement policy would probably work well with the Z8010. Without an LREF field and strong multiuser support, WS-Clock is likely to be inefficient. Poor sharing and task switching make a VS(n) design impractical. Time required for MMU programming and the page-fault recovery procedure is partially offset by translation speed. I can't see the Z8010 finding much use outside of systems with few users or nonvirtual environments.

The Zilog Z8015

The Z8015 is most notable for its paged-memory strategy. Although markedly different in mapping and logical block structure, most of its features borrow heavily from the Z8010 design. Protection, programming, and multiprocessing components are virtually identical. This MMU's main selling point is the simplified allocation and storage mechanism inherent in a paged system. Also, several mistakes found in the Z8010 are corrected in the Z8015.

The Z8015 employs an associative lookup mapping scheme. Each Z8015 MMU chip can map 64 pages, each 2K bytes in length. Thus, each unit directly maps 128K bytes. Up to 64 units can be grouped together, giving you a total of 4096 page frames (8 megabytes)—ample room for system expansion.

If you don't like 2K-byte pages, simple wiring alterations allow different size options. Page attributes are the same as a Z8010's except that I/O enable is omitted—too bad, it's a handy feature. Translation time is about 15 percent slower.

As with the Z8010, you should stick to the Clock replacement algorithm and a VS(1) design. The MMU supplies all the information necessary

to recover from a page fault; extra hardware is not required. The increased number of page frames is noteworthy: you can achieve a higher degree of multitasking, and it is easier to expand storage. But with the Z8015's fixed-page policy, you sacrifice some degree of flexibility. Adequate support for virtual memory and system security yield the components of a virtual microcomputer. On top of all this, you can use the Z8015 to implement dynamic mapping for the Z8004, the 16-bit-address version of the Z8000 processor.

The Motorola MC68451

Motorola has come up with an excellent memory-control product. The MC68451 furnishes the horsepower to construct a serious virtual computer system. In combination with the MC68000, which I think is the best 16-bit microprocessor around, the MC68451 is quite impressive. Before getting too worked up, however, I should mention that I do have a few reservations about the device. Despite this, the MMU uses some intriguing methods to build the functions that constitute a virtual-memory scheme.

Like the Z8015, the MC68451 relies on associative mapping. Address translation takes place in two stages consisting of an address range and user-space comparison. A clever technique accomplishes both lookup and segment bounds-checking in one fell swoop. Normally, logical page numbers and each map cell's page number are compared bit-wise for a match. Instead, the MC68451 employs a mask field that selects which address bits to check against a map cell's page number. The mask effectively turns some of these bits into "don't cares." The end result is that several logical-address page values can map to a single cell entry.

Motorola's excellent MC68451 furnishes the horsepower to construct a serious virtual computer system.

Each map cell contains a user-space number and associated mask. A valid memory reference must match the active processor user number. The masking function allows a range of user numbers (or just one) to use the same segment; sharing among users becomes almost a trivial task.

One worrisome point is the map-table size. You get merely 32 map cells per MMU, and physical limitations restrict you to a total of eight MMU devices (or 256 cells). You need more than that. Also, the MMU design is complex; it has too many internal registers and multiple MMU coordination is complicated. You have your work cut out for you programming this hardware.

Status registers contain used and changed bits to aid virtual paging routines. But there's no provision for an LREF attribute. Curiously, some bits cause an interrupt to be generated whenever reference is made to the segment. The purpose of this feature eludes me—maybe it's for debugging.

Protection and data typing are practically nonexistent. You can only write-inhibit segments. Any distinction between code and data or user and supervisor happens at the processor level—this makes security less effective.

The MC68451's multiuser facilities and fast context switching make WS-Clock a possible paging strategy. But you must build an extra data structure to hold reference times to support this approach. Though the MC68451 is short on facilities to fully protect data resources, you are compensated by more features for virtual memory. Also, its architecture delivers functions conducive to VS(n) designs.

The Intel iAPX 286

The Intel iAPX 286 constitutes a complete virtual-memory processor that I believe supplies the best features available in the microcomputer world today. What I consider most amazing is the fact that the memory-control unit is practically identical to a Burroughs B5500 mainframe system—a highly touted segmented architecture. Benefits are numerous and problems sparse. To summarize, I'll list the principal advantages:

- it has an integrated processor/MMU design using the 8086 microprocessor, one of the most popular 16-bit processors, making it compatible with existing software
- it has a gigantic (1-gigabyte) address space
- memory segments can be sized with a resolution of 1 byte, making it ideal for program modules
- it features advanced data-protection measures: four priority levels and several data attributes
- it can completely support WS-Clock and VS(n) designs

The iAPX 286's local data table (LDT) register points to a map table residing in MM. There's also a global equivalent of this (GDT) for shared segments. You can define up to 16,384 active segments per user. The number of users is almost boundless. Four fast internal processor registers hold the most recently accessed segment descriptors (called a cache). If you reference a map cell held in one of these registers, there's no translation-time penalty.

Because the iAPX 286, like the 8086, uses segment registers, your program must load these registers prior to memory access. Only branching instructions allow you to specify a full 32-bit virtual address. Segment-register management has two negative aspects: segment-control code clutters a program and net translation time grows. Compilers normally solve the first problem. The time problem is more of a nuisance.

The iAPX 286 features excellent virtual-memory support. Map cells have 16 undefined bits that you can use for several purposes. For instance, 12 to 14 bits would be sufficient for an LREF field. You could also allocate a fix bit to lock special pages into MM (e.g., a supervisor kernel program or an LDT table). Used and segment present bits are supplied, but a changed bit is noticeably absent.

The Intel product represents a superb tool for building virtual memory. Drawbacks are minor: small cache size, the need for segment-register management, and no changed bit. And the device is easy to program. Instead of an I/O program to operate a separate MMU, you do simple loads and stores of memory. The iAPX 286's integrated approach, its numerous features, and its regular design comprise an impressive computing engine.

Evaluation

Which MMU should you choose? The answer depends on various factors. Above all else, the companion microprocessor sways this decision. Get a system with a processor you like—it influences your software and operating-system selections. Performance requirements and intended applications are important. Is the multitasking level a factor? Do you want the flexibility of segmentation or the simplicity of paging? What software policies or peripheral components fit your needs? Table 5 compares the MMU products. The evaluation offers a yardstick for

	MEMORY-MANAGEMENT UNITS			
OVERALL RATINGS	Intel iAPX 286	Motorola MC68451	Zilog Z8010	Zilog Z8015/PMMU
Virtual-memory features	Excellent	Good	Poor	Average
Support for the Clock page-replacement algorithm	Excellent	Excellent	Good	Excellent
Support for the WS-Clock algorithm	Excellent	Good	Poor	Poor
VS(n) architecture	Excellent	Good	Poor	NA
Device features and performance				
Translation speed	Excellent	Poor	Good	Average
Address space	Excellent	Good	Poor	Average
Block resolution	Excellent	Good	Average	NA
Mapping strategy	Good	Excellent	Average	Excellent
Companion processor				
Popularity	Excellent	Good	Average	Average
Architectural design	Average	Excellent	Good	Good
Multiuser capability	Excellent	Excellent	Average	Average
Design flexibility	Good	Good	Average	Poor
Expansion potential	NA	Good	Average	Good
Protection features	Excellent	Average	Good	Good
Ease of programming	Excellent	Average	Good	Good
Hardware requirements	Excellent	Good	Poor	Good
Complexity (board level)	Excellent	Good	Poor	Average
Page-fault overhead	Good	Excellent	Poor	Poor

Table 5: *An overall comparison of the four memory-management units surveyed in this article. This evaluation highlights the differences between each MMU and gives you an idea of the application possibilities.*

analyzing potential applications. You can draw your own conclusions on how well each device addresses virtual-memory concepts.

New Horizons

Virtual memory opens up a whole new world for microcomputer systems. The expanded address space accommodates traditional large-scale software applications: database-management systems, sophisticated operating systems, and complex high-level-language translators. Moreover, some unique applications of virtual memory exist for microcomputer systems.

Virtual storage streamlines database operations. For example, you don't have to use complex file-access techniques to locate data. A 1-gigabyte address space defines an enormous amount of information. Applications can be much bigger and retrieval time much faster.

A VS(n) design has very exciting implications for microcomputers. You may have observed the trend among microcomputer vendors to offer a choice of several of the leading operating systems with their hardware. With a VS(n) organization, several different operating systems could run in the separate logical-address spaces concurrently. Think of it, CP/M for one user, Xenix for another, Oasis-16 over there. The IBM 370/VM (virtual machine) applies this approach with good success.

Virtual microcomputers give you many other unique software avenues. Consider the memory needs of a 1K-by 1K-byte color graphics system. With various shades and colors, you will quickly consume 1 megabyte of memory. And how about the trend toward integrated business environments? Word processing, report generation, spreadsheet analysis, electronic filing, etc., collectively take a sizable amount of storage.

When will virtual microcomputers be available? They are right now. Altos Computer Systems, Integrated Business Computers (IBC), and Plexus all feature virtual systems designed around the Motorola MC68000 processor. IBM has been looking at the iAPX 286 with some interest, and its future microcomputer systems should prove interesting. The trend is just beginning.

Conclusion

Virtual memory will play a vital role in the evolution of microprocessor-based computer systems. The memory-management units I've reviewed here lay the foundation by supplying the essential hardware components.

Some people claim that virtual memory fails to provide good performance. I disagree. A carefully designed unit, properly tuned (e.g., with the proper Θ parameter for the WS-Clock algorithm), should actually improve a system's operation. MMUs, sophisticated microprocessors, and simple management policies collectively supply the elements that make virtual systems viable and inevitable. A wider range of advanced applications becomes feasible.

Which MMU is the best is not totally clear. Obviously, the iAPX 286 offers some outstanding features. However, your intended applications and software considerations should figure prominently when you make your determination.

A big memory space presents a new software frontier. With these new MMU devices, it will soon be possible to have the processing power of a mainframe in the size of a desktop. ■

References

1. Carr, R. and J. Hennessy. "WSCLOCK—A Simple and Effective Algorithm for Virtual Memory Management." *Proceedings of the 8th Symposium on Operating Systems Principles*, ACM, Vol. 15, No. 5, December 1981, pp. 87–95.
2. Denning, P. "Working Sets Past and Present." *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, January 1980, pp. 64–84.
3. Easton, M. and P. Franaszek. "Use Bit Scanning in Replacement Decisions." *IEEE Transactions on Software Engineering*, Vol. C-28, No. 2, February 1979, pp. 133–141.
4. Hellerman, H. and T. Conroy. *Computer System Performance*. New York: McGraw-Hill, 1975.
5. *Intel iAPX 286 Preliminary Users Manual*. Santa Clara, CA: Intel Corporation, 1981.
6. *Motorola MC68451, Advance Information*. Austin, TX: Motorola Inc., 1981.
7. *Z8015 Paged Memory Management Unit, Product Specification*. Campbell, CA: Zilog Inc., 1981.
8. *Zilog 1982/83 Data Book*. Campbell, CA: Zilog Inc., 1982.