Memory Systems

Doug Burger
James R. Goodman
Gurindar S. Sohi

University of Wisconsin-Madison

0.1 Introduction

The memory system serves as the repository of information (data) in a computer system. The processor (also called the central processing unit, or CPU) accesses (reads or loads) data from the memory system, performs computations on them, and stores (writes) them back to memory. The memory system is a collection of storage locations. Each storage location, or memory word, has a numerical address. A collection of storage locations from an address space. Figure 1 shows the essentials of how a processor is connected to a memory system via address, data, and control lines.

When a processor attempts to load the contents of a memory location, the request is very urgent. In virtually all computers, the work soon comes to a halt (in other words, the processor stalls) if the memory request does not return quickly. Modern computers are generally able to continue briefly by overlapping memory requests, but even the most sophisticated computers will frequently exhaust their ability to process data and stall momentarily in the face of long memory delays. Thus, a key performance parameter in the design of any computer, fast or slow, is the effective speed of its memory.

Ideally, the memory system must be both infinitely large so that it can contain an arbitrarily large amount of information, and infinitely fast so that it does not limit the processing unit. Practically, however, this is not possible. There are three properties of memory that are inherently in conflict: speed, capacity, and cost. In general, technology tradeoffs can be employed to optimize any two of the three factors at the expense of the third. Thus it

is possible to have memories that are (1) large and cheap, but not fast; (2) cheap and fast, but small; or (3) large and fast, but expensive. The last of the three is further limited by physical constraints. A large-capacity memory that is very fast is also physically large, and speed-of-light delays place a limit on the speed of such a memory system.

The **latency** (L) of the memory is the delay from when the processor first requests a word from memory until that word arrives and is available for use by the processor. The latency of a memory system is one attribute of performance. The other is **bandwidth** (BW), which is the rate at which information can be transferred from the memory system. The bandwidth and the latency are closely related. If R is the number of requests that the memory can service simultaneously, then:

$$BW = \frac{R}{L} \qquad (1)$$

From Eq. (1) we see that a decrease in the latency will result in an increase in bandwidth, and vice versa, if R is unchanged. We can also see that the bandwidth can be increased by increasing R, if L does not increase proportionately. For example, we can build a memory system that takes 20 ns to service the access of a single 32-bit word. Its latency is 20 ns per 32-bit word, and its bandwidth is

$$\frac{32}{20x10^{-9}} \frac{\text{bits}}{\text{sec}}$$

or 200 Mbytes/s. If the memory system is modified to accept a new (still 20 ns) request for a 32-bit word every 5 ns by overlapping requests, then its bandwidth is

$$\frac{32}{5x10^{-9}} \frac{\text{bits}}{\text{sec}}$$

or 800 Mbytes/s. This memory system must be able to handle four requests at a given time.

Building an ideal memory system (infinite capacity, zero latency and infinite bandwidth, with affordable cost) is not feasible. The challenge is, given the cost and technology constraints, to engineer a memory system whose abilities match the abilities that the processor demands of it. That is, engineering a memory system that performs

as close to an ideal memory system (for the given processing unit) as is possible. For a processor that stalls when it makes a memory request (some current microprocessors are in this category), it is important to engineer a memory system with the lowest possible latency. For those processors that can handle multiple outstanding memory requests (vector processors and high-end CPUs) it is important not only to reduce latency, but also to increase bandwidth (over what is possible by latency reduction alone) by designing a memory system that is capable of servicing multiple requests simultaneously.

Memory hierarchies provide decreased average latency and reduced bandwidth requirements, whereas parallel or **interleaved** memories provide higher bandwidth.

0.2 Memory Hierarchies

Technology does not permit memories that are cheap, large, and fast. By recognizing the nonrandom nature of memory requests, and emphasizing the average rather than worst case latency, it is possible to implement a hierarchical memory system that performs well. A small amount of very fast memory, placed in front of a large, slow memory, can be designed to satisfy most requests at the speed of the small memory. This, in fact, is the primary motivation for the use of registers in the CPU: in this case, the programmer makes sure that the most commonly accessed variables are allocated to registers.

A variety of techniques, employing either hardware, software, or a combination of the two, can be employed to assure that most memory references are satisfied by the faster memory. The foremost of these techniques is the exploitation of the locality of reference principle. This principle captures the fact that some memory locations are referenced much more frequently than others. Spatial locality is the property that an access to a given memory location greatly increases the probability that neighboring locations will be accessed immediately. This is largely, but not exclusively, a result of the tendency to access memory locations sequentially. Temporal locality is the property that an access to a given memory location greatly increases the probability that the same location will be accessed again soon. This is largely, but not exclusively, a result of the high frequency of programs' looping behavior. Particularly for temporal locality, a good predictor of the future is the past: the longer a vari-

able has gone unreferenced, the less likely it is to be accessed soon.

Figure 2 depicts a common construction of a memory hierarchy. At the top of the hierarchy are the CPU registers, which are small and extremely fast. The next level down in the hierarchy is a special, high-speed semiconductor memory, known as a **cache memory**. The cache can actually be divided into multiple distinct levels; most current systems have between one and three levels of cache. Some of the levels of cache may be on the CPU chip itself, they may be on the same module as the CPU, or they may all be entirely distinct. Below the cache is the conventional memory, referred to as main memory, or backing storage. Like a cache, main memory is semiconductor memory, but it is slower, cheaper, and denser than a cache. Below the main memory is the virtual memory, which is generally stored on magnetic or optical disk. Accessing the virtual memory can be tens of thousands of times slower than accessing the main memory, since it involves moving mechanical parts.

As requests go deeper into the memory hierarchy, they encounter levels that are larger (in terms of capacity) and slower than the higher levels (moving left to right in Figure 2). In addition to size and speed, the bandwidth in between adjacent levels in the memory hierarchy is smaller for the lower levels. The bandwidth in between the registers and top cache level, for example, is higher than that between cache and main memory or main memory and virtual memory. Since each level presumably intercepts a fraction of the requests, the bandwidth to the level below need not be as great as that to the intercepting level.

A useful performance parameter is the effective latency. If the needed word is found in a level of the hierarchy, it is a hit; if a request must be sent to the next lower level, the request is said to miss. If the latency L_{HIT} is known in the case of a hit and the latency in the case of a miss is L_{MISS}, the effective latency for that level in the hierarchy can be determined from the hit ratio (H), the fraction of memory accesses that are hits:

$$L_{\text{average}} = L_{\text{HIT}}H + L_{\text{MISS}}(1 - H) \tag{2}$$

The portion of memory accesses that miss is called the miss ratio ($M = 1 - H$). The hit ratio is strongly influenced by the program being executed, but is largely independent of the ratio of cache size to memory size. It is not uncommon for a cache with a capacity of a few thousand bytes to exhibit a hit ratio greater than 90%.

0.3 Cache Memories

The basic unit of construction of a semiconductor memory system is a module or bank. A memory bank, constructed from several memory chips, can service a single request at a time. The time that a bank is busy servicing a request is called the bank busy time. The bank busy time limits the bandwidth of a memory bank. Both caches and main memories are constructed in this fashion, although caches have significantly shorter bank busy times than do main memory banks.

The hardware can dynamically allocate parts of the cache memory for addresses deemed most likely to be accessed soon. The cache contains only redundant copies of the address space, which is wholly contained in the main memory. The cache memory is associative, or content-addressable. In an associative memory, the address of a memory location is stored, along with its content. Rather than reading data directly from a memory location, the cache is given an address and responds by providing data which may or may not be the data requested. When a cache miss occurs, the memory access is then performed with respect to the backing storage, and the cache is updated to include the new data.

The cache is intended to hold the most active portions of the memory, and the hardware dynamically selects portions of main memory to store in the cache. When the cache is full, bringing in new data must be matched by deleting old data. Thus a strategy for cache management is necessary. Cache management strategies exploit the principle of locality. Spatial locality is exploited by the choice of what is brought into the cache. Temporal locality is exploited by the choice of which block is removed. When a cache miss occurs, hardware copies a large, contiguous block of memory into the cache, which includes the requested word. This fixed-size region of memory, known as a cache *line* or *block*, may be as small as a single word, or up to several hundred bytes. A block is a set of contiguous memory locations, the number of which is usually a power of two. A block is said to be aligned if the lowest address in the block is exactly divisible by the block size. That is to say, for a block of size B beginning at location A, the block is aligned if

$$A \text{ modulo } B = 0 \tag{3}$$

Conventional caches require that all blocks be aligned.

When a block is brought into the cache, it is likely that another block must be evicted. The selection of the evicted block is based on an attempt to capture temporal locality. Since prescience is difficult to achieve, other methods are generally used to predict future memory accesses. A least-recently-used (LRU) policy is often the basis for the replacement choice. Other replacement policies are sometimes used, particularly because true LRU replacement requires extensive logic and hardware bookkeeping.

The cache often comprises two conventional memories: the data memory and the tag memory, shown in Figure 3. The address of each cache line contained in the data memory is stored in the tag memory, as well as other information (state information), particularly the fact that a valid cache line is present. The state also keeps track of which cache lines the processor has modified. Each line contained in the data memory is allocated a corresponding entry in the tag memory to indicate the full address of the cache line.

The requirement that the cache memory be associative (content-addressable) complicates the design. Addressing data by content is inherently more complicated than by its address. All the tags must be compared concurrently, of course, because the whole point of the cache is to achieve low latency. The cache can be made simpler, however, by introducing a mapping of memory locations to cache cells. This mapping limits the number of possible cells in which a particular line may reside. The extreme case is known as direct mapping, in which each memory location is mapped to a single location in the cache. Direct mapping makes many aspects of the design simpler, since there is no choice of where the line might reside, and no choice as to which line must be replaced. Direct mapping, however, can result in poor utilization of the cache when two memory locations are alternately accessed and must share a single cache cell.

A hashing algorithm is used to determine the cache address from the memory address. The conventional mapping algorithm consists of a function with the form

$$A_{\text{cache}} = \frac{A_{\text{memory}}\ mod\ \text{cache_size}}{\text{cache_line_size}} \tag{4}$$

where A_{cache} is the address within the cache for main memory location A_{memory}, *cache_size* is the capacity of the

cache in addressable units (usually bytes), and *cache_line_size* is the size of the cache line in addressable units. Since the hashing function is simple bit selection, the tag memory need only contain the part of the address not implied by the hashing function. That is,

$$A_{\text{tag}} = A_{\text{memory}}\ div\ \text{size_of_cache} \tag{5}$$

where A_{tag} is stored in the tag memory and *div* is the integer divide operation. In testing for a match, the complete address of a line stored in the cache can be inferred from the tag and its storage location within the cache.

A *two-way set-associative* cache maps each memory location into either of two locations in the cache and can be constructed essentially as two identical direct-mapped caches. However, both caches must be searched at each memory access, and the appropriate data selected and multiplexed on a tag match (hit). On a miss, a choice must be made between the two possible cache lines as to which is to be replaced. A single LRU bit can be saved for each such pair of lines to remember which line has been accessed more recently. This bit must be toggled to the current state each time either of the cache lines is accessed.

In the same way, an *M-way associative* cache maps each memory location into any of M memory locations in the cache and can be constructed from M identical direct-mapped caches. The problem of maintaining the LRU ordering of M cache lines quickly becomes hard, however, since there are $M!$ possible orderings, so it takes at least

$$\lceil \log_2(M!) \rceil \tag{6}$$

bits to store the ordering. In practice, this requirement limits true LRU replacement to three- or four-way set associativity.

Figure 3 shows how a cache is organized into sets, blocks, and words. The cache shown is a 2-Kbyte, four-way set-associative cache, with 16 sets. Each set consists of four blocks. The cache block size in this example is 32 bytes, so each block contains eight 4-byte words. Also depicted at the bottom Figure 3 is a four-way interleaved main memory system (see the next section for details). Each successive word in the cache block maps into a different main memory bank. Because of the cache's mapping restrictions, each cache block obtained

from main memory will be loaded into its corresponding set, but may appear anywhere within that set.

Write operations require special handling in the cache. If the main memory copy is updated with each write operation—a technique known as write-through or store-through—the writes may force operations to stall while the write operations are completing. This can happen after a series of write operations even if the processor is allowed to proceed before the write to the memory has completed. If the main memory copy is not updated with each write operation—a techniques known as write-back or copy-back or deferred writes—the main memory locations become stale, that is, memory no longer contains the correct values and must not be relied upon to provide data. This is generally permissible, but care must be exercised to make sure that it is always updated before the line is purged from the cache and that the cache is never bypassed. Such a bypass could occur with DMA (*direct memory access*), in which the I/O system writes directly into main memory without the involvement of the processor.

Even for a system that implements write-through, care must be exercised if memory requests may bypass the cache. While the main memory is never stale, a write that bypasses the cache, such as from I/O, could have the effect of making the cached copy stale. A later access by the CPU could then provide an incorrect value. This can only be avoided by making sure that cached entries are invalidated even if the cache is bypassed. The problem is relatively easy to solve for a single processor with I/O, but becomes very difficult to solve for multiple processors, particularly so if multiple caches are involved as well. This is known in general as the cache *coherence* or *consistency* problem.

The cache exploits spatial locality by loading an entire cache line after a miss. This tends to result in bursty traffic to the main memory, since most accesses are filtered out by the cache. After a miss, however, the memory system must provide an entire line at once. Cache memory nicely complements an interleaved, high-bandwidth main memory (described in the next section), since a cache line can be interleaved across many banks in a regular manner—thus avoiding memory conflicts and being loaded rapidly into the cache. The example main memory shown in Figure 3 can provide the entire cache line with two parallel memory accesses.

Conventional caches traditionally could not accept requests while they were servicing a miss request. In other

words, they locked up or blocked when servicing a miss. The growing penalty for cache misses has made it necessary for high-end commodity memory systems to continue to accept (and service) requests from the processor while a miss is being serviced. Some systems are able to service multiple miss requests simultaneously. To allow this mode of operation, the cache design is lockup-free or non-blocking [Kroft, 1981]. Lockup-free caches have one structure for each simultaneous outstanding miss that they can service. This structure holds the information necessary to correctly return the loaded data to the processor, even if the misses come back in a different order than that in which they were sent.

Two factors drive the existence of multiple levels of cache memory in the memory hierarchy: access times and a limited number of transistors on the CPU chip. Larger banks with greater capacity are slower then smaller banks. If the time needed to access the cache limits the clock frequency of the CPU, then the first-level cache size may need to be constrained. Much of the benefit of a large cache may be obtained by placing a small first-level cache above a larger second-level cache; the first is accessed quickly and the second holds more data close to the processor. Since many modern CPUs have caches on the CPU chip itself, the size of the cache is limited by the CPU silicon real-estate. Some CPU designers have assumed that system designers will add large off-chip caches to the one or two levels of caches on the processor chip. The complexity of this part of the memory hierarchy may continue to grow as main memory access penalties continue to increase.

Caches that appear on the CPU chip are manufactured by the CPU vendor. Off-chip caches, however, are a commodity part sold in large volume. An incomplete list of major cache manufacturers is Hitachi, IBM Micro, Micron, Motorola, NEC, Samsung, SGS-Thomson, Sony, and Toshiba. Although most personal computers and all major workstations now contain caches, very high-end machines (such as multi-million dollar supercomputers) do not usually have caches. These ultra-expensive computers can afford to implement their main memory in a comparatively fast semiconductor technology such as static RAM (SRAM), and can afford so many banks that cacheless bandwidth out of the main memory system is sufficient. Massively-parallel processors (MPPs), however, are often constructed out of workstation-like nodes to reduce cost. MPPs therefore contain cache hierarchies similar to those found in the workstations on which the nodes of the MPPs are based.

Cache sizes have been steadily increasing on personal computers and workstations. Intel Pentium-based personal computers come with 8 Kbyte each of instruction and data caches. Two of the Pentium chip sets, manufactured by Intel and OPTi, allow level-two caches ranging from 256 to 512 Kbyte and 64 Kbyte to 2 Mbyte, respectively. The newer Pentium Pro systems also have 8 Kbyte, first-level instruction and data caches, but they also have either a 256 Kbyte or a 512 Kbyte second-level cache on the same module as the processor chip. Higher-end workstations—such as DEC Alpha 21164-based systems—are configured with substantially more cache. The 21164 also has 8 Kbyte, first-level instruction and data caches. Its second-level cache is entirely on-chip, and is 96 Kbyte. The third-level cache is off-chip, and can have a size ranging from 1 Mbyte to 64 Mbyte.

For all desktop machines, cache sizes are likely to continue to grow—although the rate of growth compared to processor speed increases and main memory size increases is unclear.

0.4 Parallel and Interleaved Main Memories

Main memories are comprised of a series of semiconductor memory chips. A number of these chips, like caches, form a *bank*. Multiple memory banks can be connected together to form an **interleaved** (or parallel) memory system. Since each bank can service a request, an interleaved memory system with K banks can service K requests simultaneously, increasing the peak bandwidth of the memory system to K times the bandwidth of a single bank. In most interleaved memory systems, the number of banks is a power of two, that is, $K = 2^k$. An n-bit memory word address is broken into two parts: a k-bit bank number and an m-bit address of a word within a bank. Though the k bits used to select a bank number could be any k bits of the n-bit word address, typical interleaved memory systems use the low-order k address bits to select the bank number; the higher order $m = n - k$ bits of the word address are used to access a word in the selected bank. The reason for using the low-order k bits will be discussed shortly. An interleaved memory system which uses the low-order k bits to select the bank is referred to as a low-order or a standard interleaved memory.

There are two ways of connecting multiple memory banks: simple interleaving and complex interleaving. Sometimes simple interleaving is also referred to as interleaving, and complex interleaving as banking.

Figure 5 shows the structure of a simple interleaved memory system. *m* address bits are simultaneously supplied to every memory bank. All banks are also connected to the same read/write control line (not shown in Figure 5). For a read operation, the banks start the read operation and deposit the data in their latches. Data can then be read from the latches, one by one, by appropriately setting the switch. Meanwhile, the banks could be accessed again, to carry out another read or write operation. For a write operation, the latches are loaded, one by one. When all the latches have been written, their contents can be written into the memory banks by supplying *m* bits of address (they will be written into the same word in each of the different banks). In a simple interleaved memory, all banks are cycled at the same time; each bank starts and completes its individual operations at the same time as every other bank; a new memory cycle can start (for all banks) once the previous cycle is complete. Timing details of the accesses can be found in *The Architecture of Pipelinined Computers*, [Kogge, 1981].

One use of a simple interleaved memory system is to back up a cache memory. To do so, the memory must be able to read blocks of contiguous words (a cache block) and supply them to the cache. If the low-order *k* bits of the address are used to select the bank number, then consecutive words of the block reside in different banks, they can all be read in parallel, and supplied to the cache one by one. If some other address bits are used for bank selection, then multiple words from the block might fall in the same memory bank, requiring multiple accesses to the same bank to fetch the block.

Figure 6 shows the structure of a complex interleaved memory system. In such a system, each bank is set up to operate on its own, independent of the other banks' operation. In this example, Bank 1 could carry out a read operation on a particular memory address, while Bank 2 carries out a write operation on a completely unrelated memory address. (Contrast this with the operation in a simple interleaved memory where all banks are carrying out the same operation, read or write, and the locations accessed within each bank represent a contiguous block of memory.) Complex interleaving is accomplished by providing an address latch and a read/write command line for each bank. The memory controller handles the overall operation of the interleaved memory. The processing unit submits the memory request to the memory controller, which determines the bank that needs to be accessed. The controller then determines if the bank is busy (by monitoring a busy line for each bank). The con-

troller holds the request if the bank is busy, submitting it later when the bank is available to accept the request. When the bank responds to a read request, the switch is set by the controller to accept the request from the bank and forward it to the processing unit. Timing details of the accesses can be found in *The Architecture of Pipelined Computers* [Kogge, 1981].

A typical use of a complex interleaved memory system is in a vector processor. In a vector processor, the processing units operate on a vector, for example a portion of a row or a column of a matrix. If consecutive elements of a vector are present in different memory banks, then the memory system can sustain a bandwidth of one element per clock cycle. By arranging the data suitably in memory and using standard interleaving (for example, storing the matrix in row-major order will place consecutive elements in consecutive memory banks), the vector can be accessed at the rate of one element per clock cycle as long as the number of banks is greater than the bank busy time.

Memory systems that are built for current machines vary widely, the price and purpose of the machine being the main determinant of the memory system design. The actual memory chips, which are the components of the memory systems, are generally commodity parts built by a number of manufacturers. The major commodity DRAM manufacturers include (but are certainly not limited to) Hitachi, Fujitsu, LG Semicon, NEC, Oki, Samsung, Texas Instruments, and Toshiba.

The low-end of the price/performance spectrum is the personal computer, presently typified by Intel Pentium systems. Three of the manufacturers of Pentium-compatible chip sets (which include the memory controllers) are Intel, OPTi, and VLSI Technologies. Their controllers provide for memory systems that are simply interleaved, all with minimum bank depths of 256 Kbyte, and maximum system sizes of 192 Mbyte, 128 Mbyte, and 1 Gbyte, respectively.

Both higher-end personal computers and workstations tend to have more main memory than the lower-end systems, although they usually have similar upper limits. Two examples of such systems are workstations built with the DEC Alpha 21164, and servers built with the Intel Pentium Pro. The Alpha systems, using the 21171 chip set, are limited to 128 Mbyte of main memory using 16 Mbit DRAMs, although they will be expandable to

512 Mbyte when 64 Mbit DRAMs are available. Their memory systems are eight-way simply interleaved, providing 128 bits per DRAM access. The Pentium Pro systems support slightly different features. The 82450KX and 82450GX chip sets include memory controllers that allow reads to bypass writes (performing writes when the memory banks are idle). These controllers can also buffer eight outstanding requests simultaneously. The 82450KX controller permits one- or two-way interleaving, and up to 256 Mbyte of memory when 16 Mbit DRAMs are used. The 82450GX chip set is more aggressive, allowing up to four separate (complex-interleaved) memory controllers, each of which can be up to four-way interleaved and have up to 1 Gbyte of memory (again with 16 Mbit DRAMs).

Interleaved memory systems found in high-end vector supercomputers are slight variants on the basic complex interleaved memory system of Figure 6. Such memory systems may have hundreds of banks, with multiple memory controllers that allow multiple independent memory requests to be made every clock cycle. Two examples of modern vector supercomputers are the Cray T-90 series and the NEC SX series. The Cray T-90 models come with varying numbers of processors—up to 32 in the largest configuration. Each of these processors is coupled with 256 Mbyte of memory, split into 16 banks of 16 Mbyte each. The T-90 has complex interleaving among banks. The largest configuration (the T-932) has 32 processors, for a total of 512 banks and 8 Gbyte of main memory. The T-932 can provide a peak of 800 GByte/second bandwidth out of its memory system. NEC's SX-4 product line, their most recent vector supercomputer series, has numerous models. Their largest single-node model (with one processor per node) contains 32 processors, with a maximum of 8 Gbyte of memory, and a peak bandwidth of 512 Gbyte/second out of main memory. Although the sizes of the memory systems are vastly different between workstations and vector machines, the techniques that both use to increase total bandwidth and minimize bank conflicts are similar.

0.5 Virtual Memory

Cache memory contains portions of the main memory in dynamically-allocated cache lines. Since the data portion of the cache memory is itself a conventional memory, each line present in the cache has two addresses

associated with it: its main memory address and its cache address. Thus, the main memory address of a word can be divorced from a particular storage location and abstractly thought of as an element in the address space. The use of a two-level hierarchy—consisting of main memory and a slower, larger disk storage device—evolved by making a clear distinction between the address space and the locations in memory. An address generated during the execution of a program is known as a virtual address, which must be translated to a physical address before it can be accessed in main memory. The total address space is only an abstraction.

A **virtual memory** address is mapped to a physical address, which indicates the location in main memory where the data actually reside [Denning, 1970]. The mapping is maintained through a structure called the page table, which is maintained in software by the operating system. Like the tag memory of a cache memory, the page table is accessed through a virtual address to determine the physical (main memory) address of the entry. Unlike the tag memory, however, the table is usually sorted by virtual addresses, making the translation process a simple matter of an extra memory access to determine the physical address of the desired item. A system maintaining the page table in the way analogous to a cache tag memory is said to have inverted page tables. In addition to the physical address mapped to a virtual page, and an indication of whether the page is present at all, a page table entry often contains other information. For example, the page table may contain the location on the disk where each block of data is stored when not present in main memory.

The virtual memory can be thought of as a collection of blocks. These blocks are often aligned and of fixed size, in which case they are known as pages. Pages are the unit of transfer between the disk and main memory, and are generally larger than a cache line—usually thousands of bytes. *A* typical page size for machines in 1995 is 4Kbyte. A page's virtual address can be broken into two parts: a virtual page number and an offset. The page number specifies which page is to be accessed, and the page offset indicates the distance from the beginning of the page to the indicated address.

A physical address can also be broken into two parts, a physical page number (also called a page frame number) and an offset. This mapping is done at the level of pages, so the page table can be indexed by means of the virtual page number. The page frame number is contained in the page table and is read out during the translation,

along with other information about the page. In most implementations the page offset is the same for a virtual address and the physical address to which it is mapped.

The virtual memory hierarchy is different than the cache/main memory hierarchy in a number of respects, resulting primarily from the fact that there is a much greater difference in latency between accesses to the disk and to main memory. While a typical latency ratio for cache and main memory is one order of magnitude (main memory has a latency ten times larger than the cache), the latency ratio between disk and main memory is often four orders of magnitude or more. This large ratio exists because the disk is a mechanical device—with a latency partially determined by velocity and inertia—whereas main memory is limited only by electronic and energy constraints. Because of the much larger penalty for a page miss, many design decisions are affected by the need to minimize the frequency of misses. When a miss does occur, the processor could be idle for a period during which it could execute tens of thousands of instructions. Rather than stall during this time, as may occur upon a cache miss, the processor invokes the operating system and may switch to a different task. Because the operating system is being invoked anyway, it is convenient to rely on the operating system to set up and maintain the page table, unlike cache memory, where it is done entirely in hardware. The fact that this accounting occurs in the operating system enables the system to use virtual memory to enforce protection on the memory. This ensures that no program can corrupt the data in memory that belong to any other program.

Hardware support provided for a virtual memory system generally includes the ability to translate the virtual addresses provided by the processor into the physical addresses needed to access main memory. Thus, only upon a virtual address miss is the operating system invoked. An important aspect of a computer that implements virtual memory, however, is the necessity of freezing the processor at the point at which a miss occurs, servicing the page table fault, and later returning to continue the execution as if no page fault had occurred. This requirement means either that it must be possible to halt execution at any point—including possibly in the middle of a complex instruction—or that it must be possible to guarantee that all memory accesses will be to pages resident in main memory.

As described above, virtual memory requires two memory accesses to fetch a single entry from memory, one

into the page table to map the virtual address into the physical address, and the second to fetch the actual data. This process can be sped up in a variety of ways. First, a special-purpose cache memory to store the active portion of the page table can be used to speed up the first access. This special-purpose cache is usually called a translation lookaside buffer (TLB). Second, if the system also employs a cache memory, it may be possible to overlap the access of the cache memory with the access to the TLB, ideally allowing the requested item to be accessed in a single cache access time. The two accesses can be fully overlapped if the virtual address supplies sufficient information to fetch the data from the cache before the virtual-to-physical address translation has been accomplished. This is true for an M-way set associative cache of capacity C if the following relationship holds:

$$\text{Page_size} \geq \frac{C}{M} \tag{7}$$

For such a cache, the index into the cache can be determined strictly from the page offset. Since the virtual page offset is identical to the physical page offset, no translation is necessary, and the cache can be accessed concurrently with the TLB. The physical address must be obtained before the tag can be compared, of course.

An alternative method applicable to a system containing both virtual memory and a cache is to store the virtual address in the tag memory instead of the physical address. This technique introduces consistency problems in virtual memory systems that either permit more than a single address space, or allow a single physical page to be mapped to more than one single virtual page. This problem is known as the aliasing problem.

Chapter 102 is devoted to virtual memory, and contains significantly more material on this topic for the interested reader.

Research Issues

Research is occurring on all levels of the memory hierarchy. At the register level, researchers are exploring techniques to provide more registers than are architecturally visible to the compiler. A large volume of work exists (and is occurring) for cache optimizations and alternate cache organizations. For instance, modern processors now commonly split the top level of the cache into separate physical caches, one for instructions (code) and one for program data. Due to the increasing cost of cache misses (in terms of processor cycles), some research

trades-off increasing the complexity of the cache for reducing the miss rate. Two examples of cache research from opposite ends of the hardware/software spectrum are blocking [Lam, 1991] and skewed-associative caches [Seznec, 1993]. Blocking is a software technique in which the programmer or compiler reorganizes algorithms to work on subsets of data that are smaller than the cache, instead of streaming entire large data structures repeatedly through the cache. This reorganization greatly improves temporal locality. The skewed-associative cache is one example of a host of hardware techniques that map blocks into the cache differently, with the goal of reducing misses from set conflicts. In skewed-associative caches, either one of two hashing functions may determine where a block should be placed in the cache, as opposed to just the one hashing function (low-order index bits) that traditional caches use. An important cache-related research topic is prefetching [Mowry, 1992], in which the processor issues requests for data well before the data are actually needed. Speculative prefetching is also a current research topic. In speculative prefetching, prefetches are issued based on guesses as to which data will be needed soon. Other cache-related research examines placing special structures in parallel with the cache, trying to optimize for workloads that do not lend themselves well to caches. Stream buffers [Jouppi, 1990] are one such example. A stream buffer automatically detects when a linear access through a data structure is occurring. The stream buffer issues multiple sequential prefetches upon detection of a linear array access.

Much of the ongoing research on main memory involves improving the bandwidth from the memory system without greatly increasing the number of banks. Multiple banks are expensive, particularly with the large and growing capacity of modern DRAM chips. Rambus [Rambus Inc., 1992] and Ramlink [IEEE Computer Society, 1993] are two such examples.

Research issues associated with improving the performance of the virtual memory system fall under the domain of operating system research. One proposed strategy for reducing page faults allows each running program to specify its own page replacement algorithm, enabling each program to optimize the choice of page replacements based on its reference pattern [Engler et al., 1995]. Other recent research focuses on improving the performance of the TLB. Two techniques for doing this are the use of a two-level TLB (the motivation is similar to that for a two-level cache), and the use of superpages [Talluri, 1994]. With superpages, each TLB entry may

represent a mapping for more than one consecutive page, thus increasing the total address range that a fixed number of TLB entries may cover.

Summary

A computer's memory system is the repository for all the information that the CPU uses and produces. A perfect memory system is one that can immediately supply any datum that the CPU requests. This ideal memory is not implementable, however, as the three factors of memory capacity, speed, and cost are directly in opposition.

By staging smaller, faster memories in front of larger, slower, and cheaper memories, the performance of the memory system may approach that of a perfect memory system—at a reasonable cost. The memory hierarchies of modern general-purpose computers generally contain registers at the top, followed by one or more levels of cache memory, main memory, and virtual memory on a magnetic or optical disk.

Performance of a memory system is measured in terms of latency and bandwidth. The latency of a memory request is how long it takes the memory system to produce the result of the request. The bandwidth of a memory system is the rate at which the memory system can accept requests and produce results. The memory hierarchy improves average latency by quickly returning results that are found in the higher levels of the hierarchy. The memory hierarchy generally reduces bandwidth requirements by intercepting a fraction of the memory requests at higher levels of the hierarchy. Some machines—such as high-performance vector machines—may have fewer levels in the hierarchy, increasing memory cost for better predictability and performance. Some of these machines contain no caches at all, relying on large arrays of main memory banks to supply very high bandwidth, with pipelined accesses of operands that mitigate the adverse performance impact of long latencies.

Cache memories are a general solution to improving the performance of a memory system. Although caches are smaller than typical main memory sizes, they ideally contain the most frequently-accessed portions of main memory. By keeping the most heavily-used data near the CPU, caches can service a large fraction of the requests without accessing main memory (the fraction serviced is called the hit rate). Caches assume locality of reference to work well transparently—they assume that accessed memory words will be accessed again quickly

(temporal locality), and that memory words adjacent to an accessed word will be accessed soon after the access in question (spatial locality). When the CPU issues a request for a datum not in the cache (a cache miss), the cache loads that datum and some number of adjacent data (a cache block) into itself from main memory.

To reduce cache misses, some caches are associative—a cache may place a given block in one of several places, collectively called a set. This set is content-addressable; a block may or may not be accessed based on an address tag, one of which is coupled with each block. When a new block is brought into a set and the set is full, the cache's replacement policy dictates which of the old blocks should be removed from the cache to make room for the new block. Most caches use an approximation of least-recently-used (LRU) replacement, in which the block last accessed farthest in the past is the one that the cache replaces.

Main memory, or backing store, consists of banks of dense semiconductor memory. Since each memory chip has a small off-chip bandwidth, rows of these chips are placed together to form a bank, and multiple banks are used to increase the total bandwidth out of main memory. When a bank is accessed, it remains busy for a period of time, during which the processor may make no other accesses to that bank. By increasing the number of interleaved (parallel) banks, the chance that the processor issues two conflicting requests to the same bank is reduced.

Systems generally require a greater number of memory locations than are available in the main memory (i.e., a larger address space). The entire address space that the CPU uses is kept on large magnetic or optical disks; this is called the virtual address space, or virtual memory. The most frequently-used sections of the virtual memory are kept in main memory (physical memory), and are moved back and forth in units called pages. The place at which a virtual address lies in main memory is called its physical address. Since a much larger address space (virtual memory) is mapped onto a much smaller one (physical memory), the CPU must translate the memory addresses issued by a program (virtual addresses) into their corresponding locations in physical memory (physical addresses). This mapping is maintained in a memory structure called the page table. When the CPU attempts to access a virtual address that does not have a corresponding entry in physical memory, a page fault occurs. Since a page fault requires an access to a slow mechanical storage device (such as a disk), the CPU usually

switches to a different task while the needed page is read from the disk.

Every memory request issued by the CPU requires an address translation, which in turn requires an access to the page table stored in memory. A translation lookaside buffer (TLB) is used to reduce the number of page table lookups. The most frequent virtual-to-physical mappings are kept in the TLB, which is a small associative memory tightly coupled with the CPU. If the needed mapping is found in the TLB, the translation is performed quickly and no access to the page table need be made. Virtual memory allows systems to run larger or more programs than are able to fit in main memory, enhancing the capabilities of the system.

Defining Terms

Bandwidth: The rate at which the memory system can service requests.

Cache memory: A small, fast, redundant memory used to store the most frequently accessed parts of the main memory.

Interleaving: Technique for connecting multiple memory modules together in order to improve the bandwidth of the memory system.

Latency: The time between the initiation of a memory request and its completion.

Memory hierarchy: Successive levels of different types of memory, which attempt to approximate a single large, fast, and cheap memory structure.

Virtual memory: A memory space implemented by storing the more-frequently-accessed parts in main memory and less-frequently-accessed parts on disk.

References

Denning, P. J. 1970. “Virtual memory,” *Computing Surveys*, vol. 2, no. 3, pp. 153-170.

Engler, D. R., Kaashoek, M. F., O’Toole, J. Jr. 1995. “Exokernel: An Operating System Architecture for Application-Level Resource Management,” *Proc. 15th Symposium on Operating Systems Principles*, pp. 251-266.

Hennessy, J. L. and Patterson, D. A. 1990. *Computer Architecture: A Quantitative Approach*, 1st ed. Morgan Kaufmann Publishers, San Mateo, CA.

Hill, M. D. 1988. “A case for direct-mapped caches,” *IEEE Computer*, vol. 21, no. 12.

IEEE Computer Society. 1993. *IEEE Standard for High-Bandwidth Memory Interface Based on SCI Signaling Technology (RamLink),* Draft 1.00 IEEE P1596.4-199X.

Jouppi, N. 1990. “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” *Proc. 17th Annual International Symposium on Computer Architecture*, pp. 364-373.

Kogge, P. M. 1981. *The Architecture of Pipelined Computers*, New York: McGraw-Hill.

Kroft, D. 1981. “Lockup-Free Instruction Fetch/Prefetch Cache Organization,” *Proc. 8th Annual International Symposium on Computer Architecture*, pp. 81-87.

Lam, M.S.,Rothberg, E. E., and Wolf, M. E. 1991. “The Cache Performance and Optimizations of Blocked Algorithms,” *Proc. 4th Annual Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 63-74.

Mowry, T. C., Lam, M. S., Gupta, A. 1992. “Design and Evaluation of a Compiler Algorithm for Prefetching,” *Proc. 5th Annual Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73.

Rambus, Inc. 1992. *Rambus Architectural Overview*, Mountain View, CA.

Seznec, A. 1993. “A case for two-way skewed-associative caches,” *Proc. 20th International Symposium on Computer Architecture*, pp. 169-178.

Smith, A. J. 1986. “Bibliography and readings on CPU cache memories and related topics,” *ACM SIGARCH Computer Architecture News*, vol. 14, no 1, pp. 22-42.

Smith, A. J. 1991. “Second bibliography on cache memories,” *ACM SIGARCH Computer Architecture News*, vol. 19, no 4, pp. 154-182.

Talluri, M. and Hill, M. D. 1994. “Surpassing the TLB Performance of Superpages with Less Operating System Support,” *Proc. Sixth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 171-182.

Further Information

Some general information on the design of memory systems is available in *High-Speed Memory Systems by* A. V. Pohm and O. P. Agarwal.

Computer Architecture: A Quantitative Approach by John Hennessy and David Patterson contains a detailed discussion on the interaction between memory systems and computer architecture.

For information on memory system research, the recent proceedings of the *International Symposium on Computer Architecture* contain annual research papers in computer architecture, many of which focus on the memory system. To obtain copies, contact the IEEE Computer Society Press, at 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264.

FIGURE CAPTIONS

Figure 1. The memory interface

Source: Dorf, R. C. 1992. *The Electrical Engineering Handbook*, 1st ed., p. 1928. CRC Press, Inc., Boca Raton, FL. With permission.

Figure 2. A memory hierarchy

Source: Dorf, R. C. 1992. *The Electrical Engineering Handbook*, 1st ed., p. 1932. CRC Press, Inc., Boca Raton, FL. With permission.

Figure 3. Components of a cache memory.

Source: Hill, M. D. 1988. "A case for direct-mapped caches," *IEEE Computer*, vol. 21, no. 12, p. 27. IEEE Computer Society, New York, NY. With permission.

Figure 4. Organization of a cache.

Figure 5. A simple interleaved memory system. (*Source*: Adapted from Kogge, 1981.)

Source: Kogge, P. M. 1981. *The Architecture of Pipelined Computers*, 1st ed., p. 41. McGraw-Hill, Inc., New York, NY. With permission.

Figure 6. A complex interleaved memory system. (*Source*: Adapted from Kogge, 1981.)

Source: Kogge, P. M. 1981. *The Architecture of Pipelined Computers*, 1st ed., p. 42. McGraw-Hill, Inc., New York, NY. With permission.

Figure 7. Virtual-to-physical address translation

Source: Dorf, R. C. 1992. *The Electrical Engineering Handbook*, 1st ed., p. 1935. CRC Press, Inc., Boca Raton, FL. With permission.

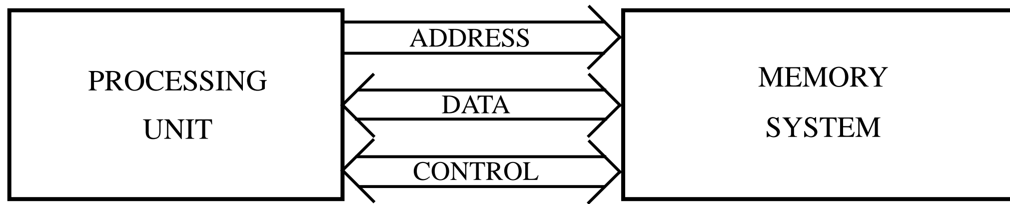
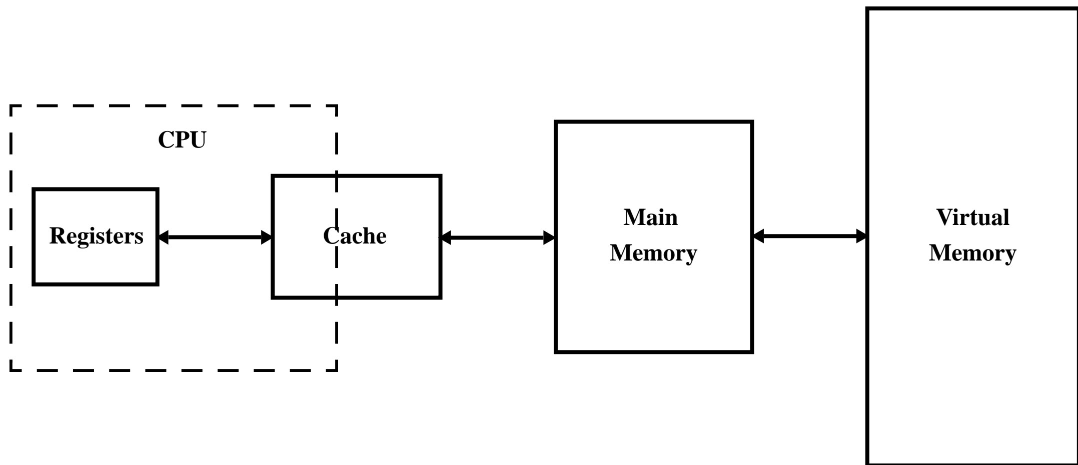


FIGURE 1



Very Fast (Electronic speeds)	Very Fast (Electronic speeds)	Fast (Electronic Speeds)	Very Slow (Mechanical Speeds)
Semiconductor SRAM	Semiconductor SRAM	Semiconductor DRAM	Magnetic/Optical
Tiny 128 bytes - 4Kbytes	Small 32 Kbytes - 4 Mbytes	Large 4 Mbytes - 512 Mbytes	Very Large 40 MBytes - 8 Gbytes

FIGURE 2

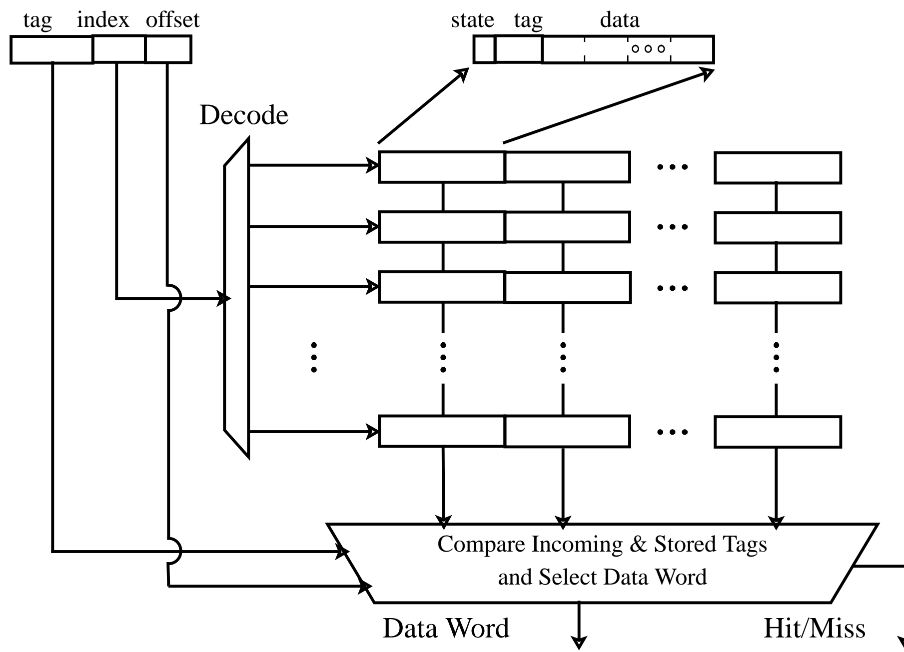


FIGURE 3

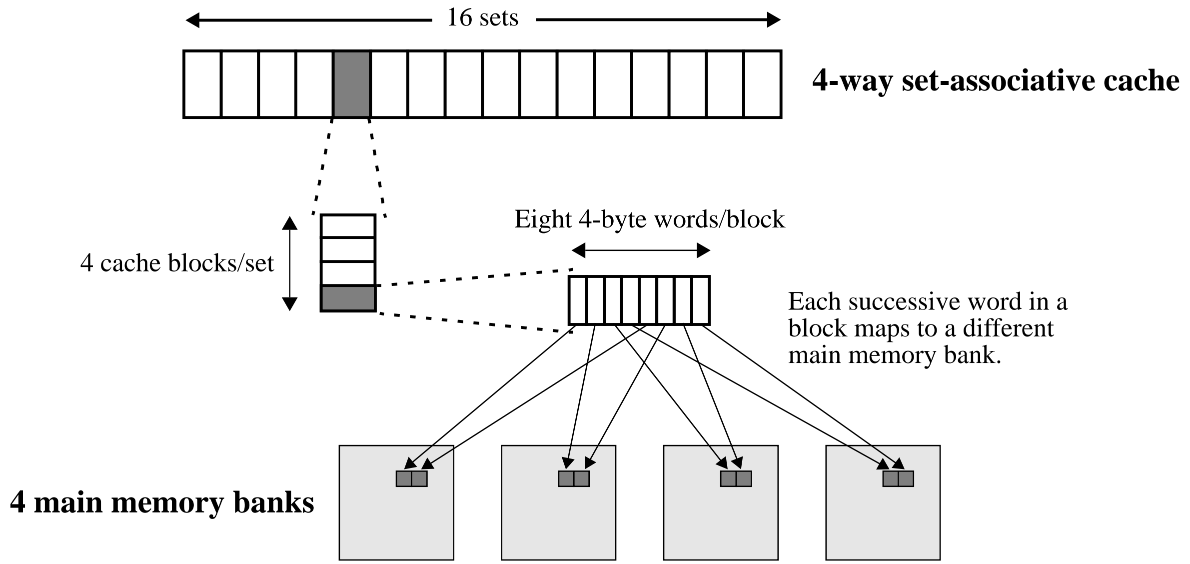


FIGURE 4

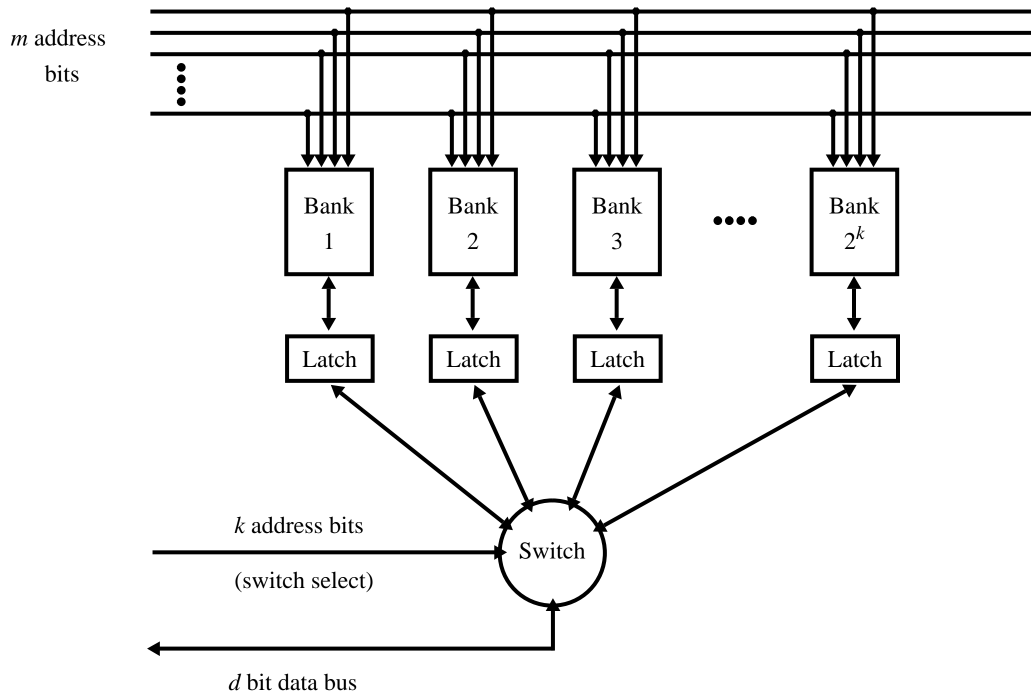


FIGURE 5

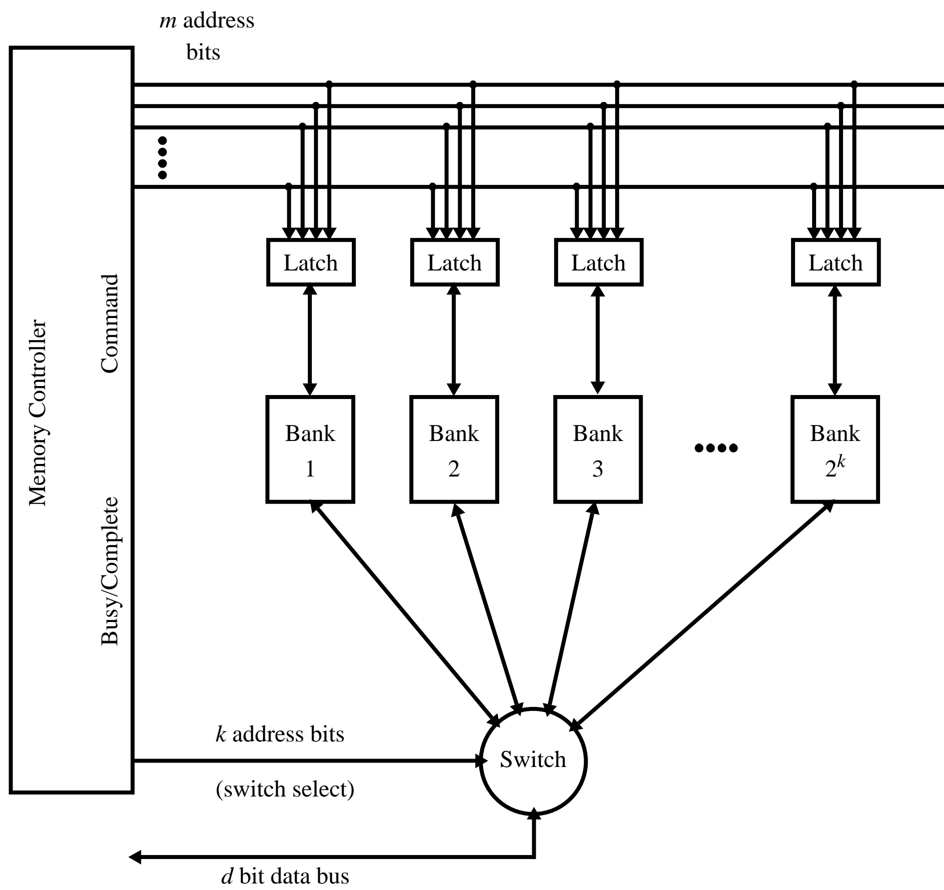


FIGURE 6

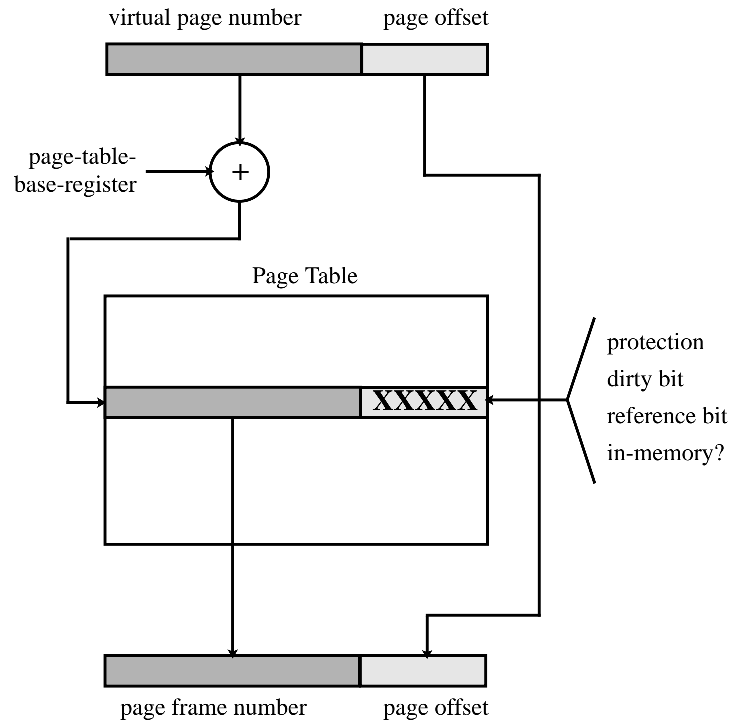


FIGURE 7