

# Memory System Design II

In Chapter 6 we introduced the concept of memory hierarchy. We have also characterized a memory hierarchy in terms of the locality of reference and its impact on the average access time. We then moved on to cover the different issues related to the first level of the hierarchy, that is, the cache memory (the reader is advised to carefully review Chapter 6 before proceeding with this chapter). In this chapter, we continue our coverage of the different levels of the memory hierarchy. In particular, we start our discussion with the issues related to the design and analysis of the (main) memory unit. Issues related to *virtual memory* design are then discussed. A brief coverage of the different *read-only memory* (ROM) implementations is provided at the end of the chapter.

## 7.1. MAIN MEMORY

As the name implies, the main memory provides the main storage for a computer. Figure 7.1 shows a typical interface between the main memory and the CPU. Two CPU registers are used to interface the CPU to the main memory. These are the memory address register (MAR) and the memory data register (MDR). The MDR is used to hold the data to be stored and/or retrieved in/from the memory location whose address is held in the MAR.

It is possible to visualize a typical internal main memory structure as consisting of rows and columns of basic cells. Each cell is capable of storing one bit of information. Figure 7.2 provides a conceptual internal organization of a memory chip. In this figure, cells belonging to a given row can be assumed to form the bits of a given memory word. Address lines  $A_{n-1}A_{n-2} \dots A_1A_0$  are used as inputs to the address decoder in order to generate the word select lines  $W_{2^n-1} \dots W_1W_0$ . A given word select line is common to all memory cells in the same row. At any given time, the address decoder activates only one word select line while deactivating the remaining lines. A word select line is used to enable all cells in a row for read or write. Data (bit) lines are used to input or output the contents of cells. Each memory cell is connected to two data lines. A given data line is common to all cells in a given column.

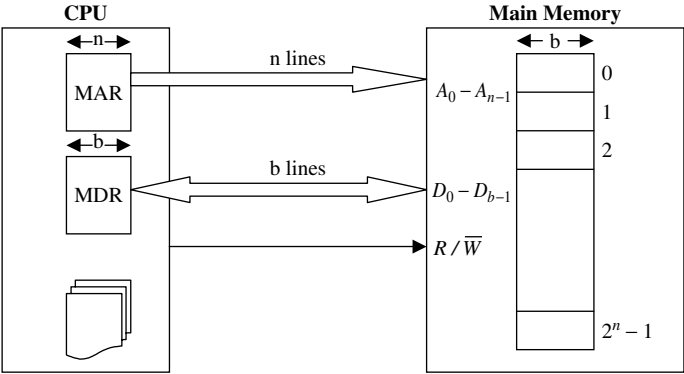


Figure 7.1 A typical CPU and main memory interface

In static CMOS technology, each main memory cell consists of six transistors as shown in Figure 7.3. The six transistor static CMOS memory cell consists of two inverters back to back. It should be noted that the cell could exist in one of the two stable states. For example, if in Figure 7.3  $A = 1$ , then transistor  $N_2$  will be on and point  $B = 0$ , which in turn will cause transistor  $P_1$  to be on, thus causing point  $A = 1$ . This represents a cell stable state, call it state 1. In a similar way

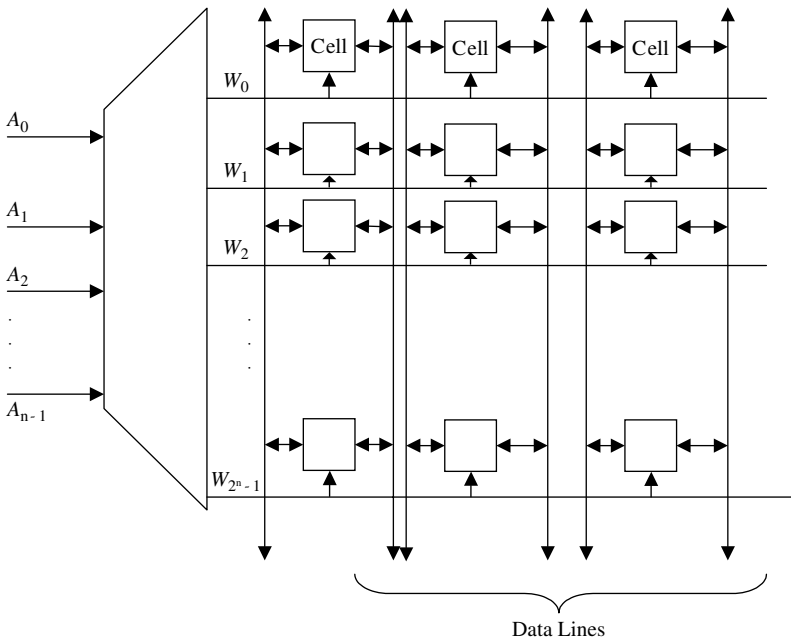
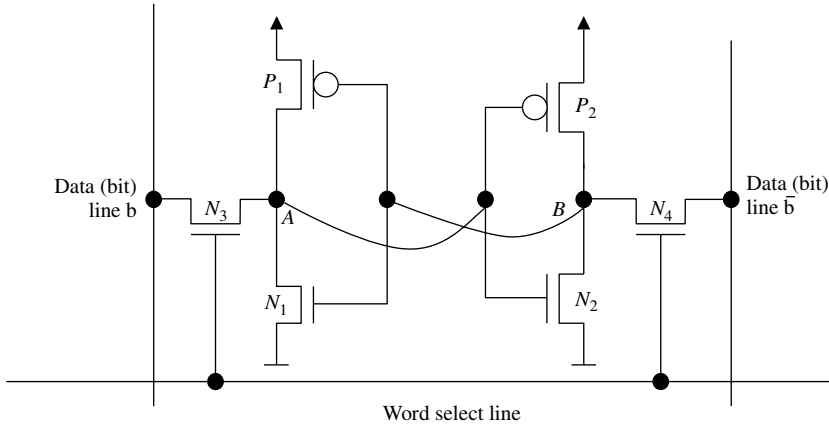


Figure 7.2 A conceptual internal organization of a memory chip



**Figure 7.3** Static CMOS memory cell

one can show that if  $A = 0$ , then  $B = 1$ , which represents the other cell stable state, call it state 0. The two transistors  $N_3$  and  $N_4$  are used to connect the cell to the two data (bit) lines. Normally (if the word select is not activated) these two transistors are turned off, thus protecting the cell from the signal values carried by the data lines. The two transistors are turned on when the word select line is activated. What takes place when the two transistors are turned on will depend on the intended memory operation as shown below.

#### Read operation:

1. Both lines  $b$  and  $\bar{b}$  are precharged high.
2. The word select line is activated, thus turning on both transistors  $N_3$  and  $N_4$ .
3. Depending on the internal value stored in the cell, point  $A(B)$  will lead to the discharge of line  $b(\bar{b})$ .

#### Write operation:

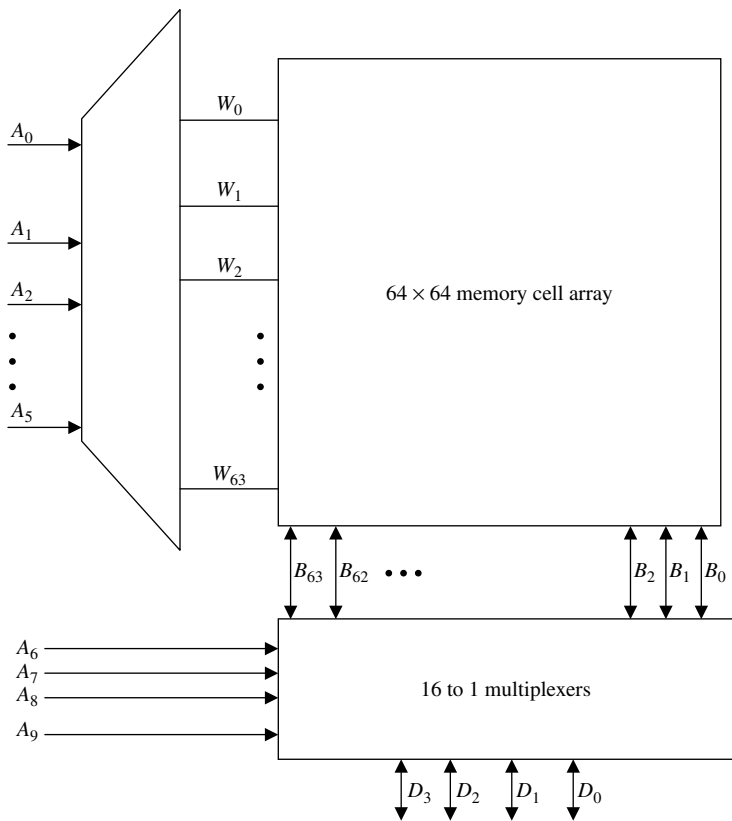
1. The bit lines are precharged such that  $b(\bar{b}) = 1(0)$ .
2. The word select line is activated, thus turning on both transistors  $N_3$  and  $N_4$ .
3. The bit line precharged with 0 will have to force the point  $A(B)$ , which has 1, to 0.

The internal organization of the memory array should satisfy an important memory design factor, that is, efficient utilization of the memory chip. Consider, for example, a  $1K \times 4$  memory chip. Using the organization shown in Figure 7.2, the memory array should be organized as 1K rows of cells, each consisting of four cells. The chip will then have to have 10 pins for the address and four pins for the data. However, this may not lead to the best utilization of the chip area.

Another possible organization of the memory cell array is as a  $64 \times 64$ , that is, to organize the array in the form of 64 rows, each consisting of 64 cells. In this case, six address lines (forming what is called the row address) will be needed in order to select one of the 64 rows. The remaining four address lines (called the column address) will be used to select the appropriate 4 bits among the available 64 bits constituting a row. Figure 7.4 illustrates this organization.

Another important factor related to the design of the main memory subsystem is the number of chip pins required in an integrated circuit. Consider, for example, the design of a memory subsystem whose capacity is 4K bits. Different organization of the same memory capacity can lead to a different number of chip pins requirement. Table 7.1 illustrates such an observation. It is clear from the table that increasing the number of bits per addressable location results in an increase in the number of pins needed in the integrated circuit.

Another factor pertinent to the design of the main memory subsystem is the required number of memory chips. It is important to realize that the available per chip memory capacity can be a limiting factor in designing memory subsystems.



**Figure 7.4** Efficient internal organization of a 1Kx4 memory chip

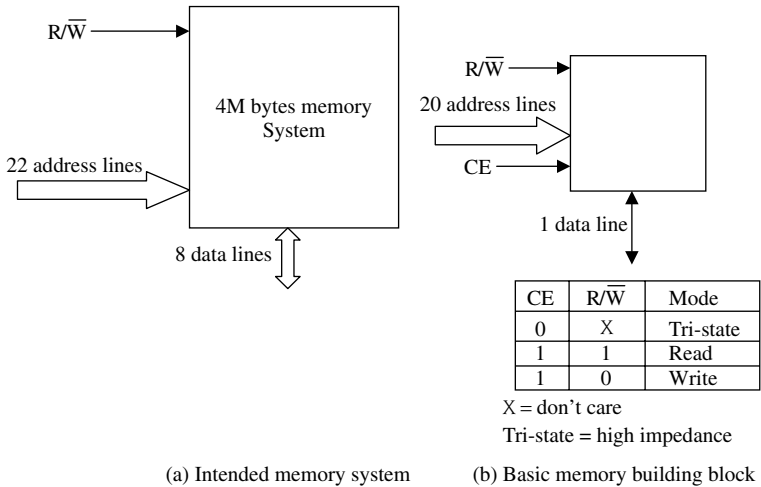
**TABLE 7.1    Impact of Using Different Organizations  
on the Number of Pins**

Organization	Number of needed address lines	Number of needed data lines
4K×1	12	1
1K×4	10	4
512×8	9	8
256×16	8	16

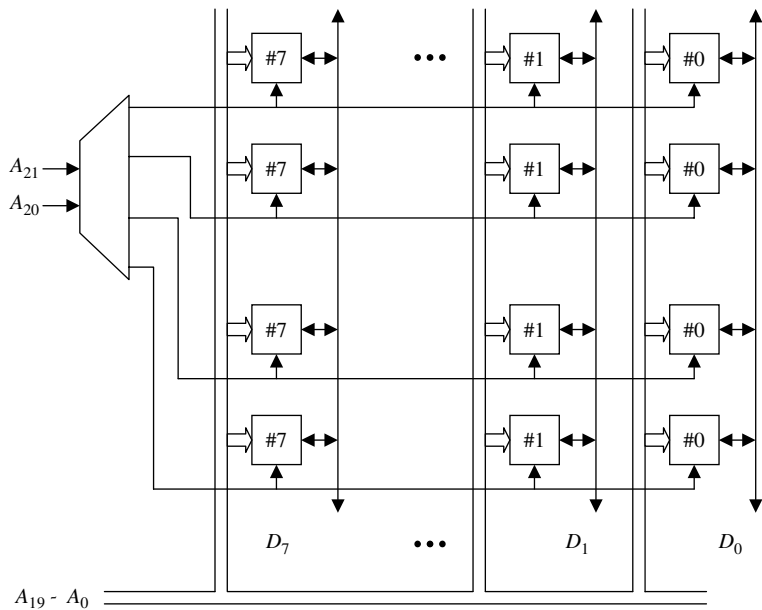
Consider, for example, the design of a 4M bytes main memory subsystem using 1M bit chip. The number of required chips is 32 chips. It should be noted that the number of address lines required for the 4M system is 22, while the number of data lines is 8. Figure 7.5 shows a block diagram for both the intended memory subsystem and the basic building block to be used to construct such a subsystem.

The memory subsystem can be arranged in four rows, each having eight chips. A schematic of such an arrangement is shown in Figure 7.6. In this figure, the least significant 20 address lines  $A_{19} - A_0$  are used to address any of the basic building block 1M single bit chips. The high-order two address lines  $A_{21} - A_{20}$  are used as inputs to a 2–4 decoder in order to generate four enable lines; each is connected to the CE line of the eight chips constituting a row.

The above discussion on main memory system design assumes the use of a six-transistor static random cell. It is possible however to use a one-transistor dynamic cell. Dynamic memory depends on storing logic values using a capacitor together with one transistor that acts as a switch. The use of dynamic memory leads to saving in chip area. However, due to the possibility of decay of the stored values



**Figure 7.5**    Block diagram of a required memory system and its basic building block



**Figure 7.6**    Organization of a 4M 8-bit memory using 1M 1-bit memory chips

(leakage of the stored charge on the capacitor), dynamic memory requires periodical (every few milliseconds) refreshment in order to restore the stored logic values. Figure 7.7 illustrates the dynamic memory array organization. The read/write circuitry in Figure 7.7 performs the functions of sensing the value on the bit line, amplifying it, and refreshing the value stored on the capacitor.

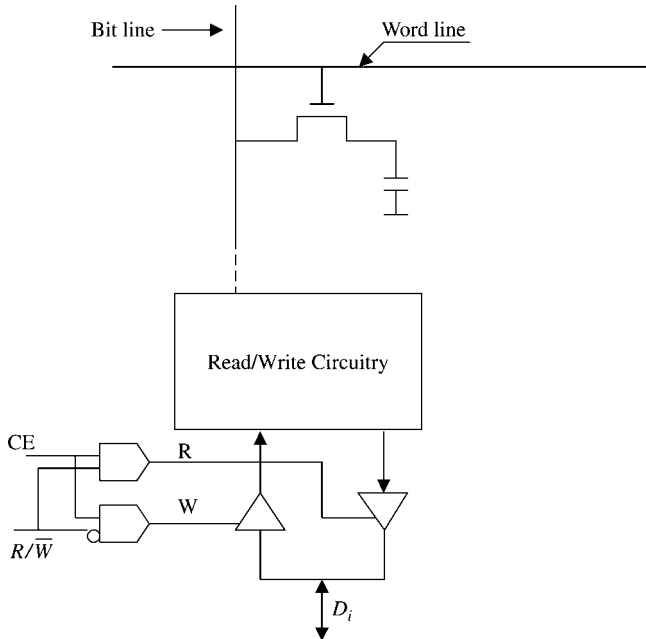
In order to perform a read operation, the bit line is precharged high (same as in static memory) and the word line is activated. That will cause the value stored on the capacitor to appear on the bit line, thus appearing on the data line  $D_i$ . As can be seen, a read operation is destructive; that is, the capacitor is charged to the bit line. Therefore, every read operation is followed by a write operation of the same value.

In order to perform a write operation, the intended value is placed on the bit line and the word line is activated. If the intended value is 1, then the capacitor will be charged, while if the intended value is 0, then the capacitor will be discharged. Table 7.2 summarizes the operation of the control circuitry.

**TABLE 7.2    Operation of the Control Circuitry**

CE	$R/\bar{W}$	Operation
0	$\times$	None
1	1	Read
1	0	Write

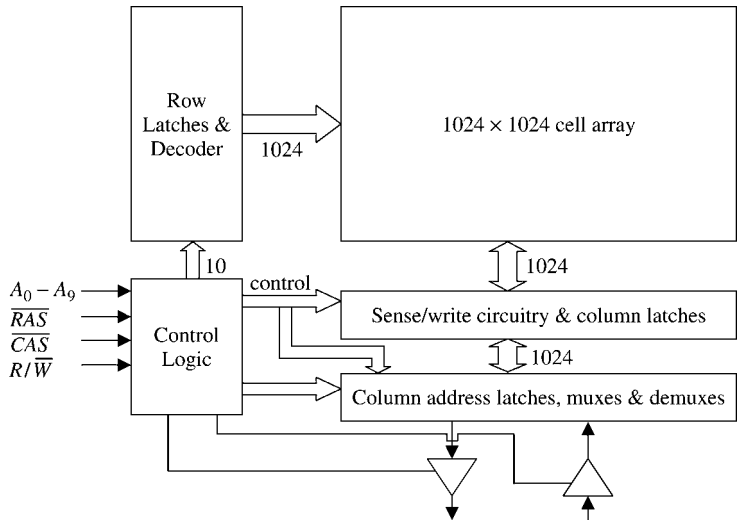
$\times$  = don't care



**Figure 7.7** Dynamic memory array organization

As discussed before, appropriate internal organization of a memory subsystem can lead to a saving in the number of IC pins required, an important IC design factor. In order to reduce the number of pins required for a given dynamic memory subsystem, it is a normal practice (as in the case of static memory) to divide the address lines into row and column address lines. In addition, the row and column address lines are transmitted over the same pins, one after the other in a scheme known as *time-multiplexing*. This can potentially cut the number of address pins required by half. Due to time-multiplexing of address lines, it will be necessary to add two extra control lines, that is, row address strobe ( $\overline{RAS}$ ) and column address strobe ( $\overline{CAS}$ ). These two control lines are used to indicate to the memory chip when the row address lines are valid and when the column address lines are valid, respectively. Consider, for example, the design of a  $1M \times 1$  dynamic memory subsystem. Figure 7.8 shows a possible internal organization of the memory cell array in which the array is organized as a  $1024 \times 1024$ .

It should be noted that only 10 address lines are shown. These are used to multiplex both the rows and columns address lines; each is 10 lines. The rows and columns latches are used to store the row and column addresses for a duration equal to the memory cycle. In this case, a memory access will consist of a  $\overline{RAS}$  and a row address, followed by a  $\overline{CAS}$  and a column address.



**Figure 7.8** A 1024 × 1024 memory array organization

**7.2. VIRTUAL MEMORY**

The concept of virtual memory is in principle similar to that of the cache memory described in Section 6.2. A virtual memory system attempts to optimize the use of the main memory (the higher speed portion) with the hard disk (the lower speed portion). In effect, virtual memory is a technique for using the secondary storage to extend the apparent limited size of the physical memory beyond its actual physical size. It is usually the case that the available physical memory space will not be enough to host all the parts of a given active program. Those parts of the program that are currently active are brought to the main memory while those parts that are not active will be stored on the magnetic disk. If the segment of the program containing the word requested by the processor is not in the main memory at the time of the request, then such segment will have to be brought from the disk to the main memory. The principles employed in the virtual memory design are the same as those employed in the cache memory. The most relevant principle is that of keeping active segments in the high-speed main memory and moving inactive segments back to the hard disk.

Movement of data between the disk and the main memory takes the form of pages. A page is a collection of memory words, which can be moved from the disk to the MM when the processor requests accessing a word on that page. A typical size of a page in modern computers ranges from 2K to 16K bytes. A page fault occurs when the page containing the word required by the processor does not exist in the MM and has to be brought from the disk. The movement of pages of programs or data between the main memory and the disk is totally transparent to the application programmer. The operating system is responsible for such movement of data and programs.



It is useful to mention at this point that although based on similar principles, a significant difference exists between cache and virtual memories. A cache miss can cause a time penalty that is 5 to 10 times as costly as a cache hit. A page fault, on the other hand can be 1000 times as costly as a page hit. It is therefore unreasonable to have the processor wait on a page fault while a page is being transferred to the main memory. This is because thousands of instructions could be executed on a modern processor during page transfer.

The address issued by the processor in order to access a given word does not correspond to the physical memory space. Therefore, such address is called a *virtual (logical) address*. The memory management unit (MMU) is responsible for the translation of virtual addresses to their corresponding physical addresses. Three address translation techniques can be identified. These are *direct-mapping*, *associative-mapping*, and *set-associative-mapping*. In all these techniques, information about the main memory locations and the corresponding virtual pages are kept in a table called the *page table*. The page table is stored in the main memory. Other information kept in the page table includes a bit indicating the validity of a page, modification of a page, and the authority for accessing a page. The *valid bit* is set if the corresponding page is actually loaded into the main memory. Valid bits for all pages are reset when the computer is first powered on. The other control bit that is kept in the page table is the *dirty bit*. It is set if the corresponding page has been altered while residing in the main memory. If while residing in the main memory a given page has not been altered, then its dirty bit will be reset. This can help in deciding whether to write the contents of a page back into the disk (at the time of replacement) or just to override its contents with another page. In the following discussion, we will concentrate on the address translation techniques keeping in mind the use of the different control bits stored in the page table.

### 7.2.1. Direct Mapping

Figure 7.9 illustrates the address translation process according to the direct-mapping technique. In this case, the virtual address issued by the processor is divided into two fields: the virtual page number and the offset fields. If the number of bits in the virtual page number field is  $N$ , then the number of entries in the page table will be  $2^N$ .

The virtual page number field is used to directly address an entry in the page table. If the corresponding page is valid (as indicated by the valid bit), then the contents of the specified page table entry will correspond to the physical page address. The latter is then extracted and concatenated with the offset field in order to form the physical address of the word requested by the processor. If, on the other hand, the specified entry in the page table does not contain a valid physical page number, then this represents a page fault. In this case, the MMU will have to bring the corresponding page from the hard disk, load it into the main memory, and indicate the validity of the page. The translation process is then carried out as explained before.

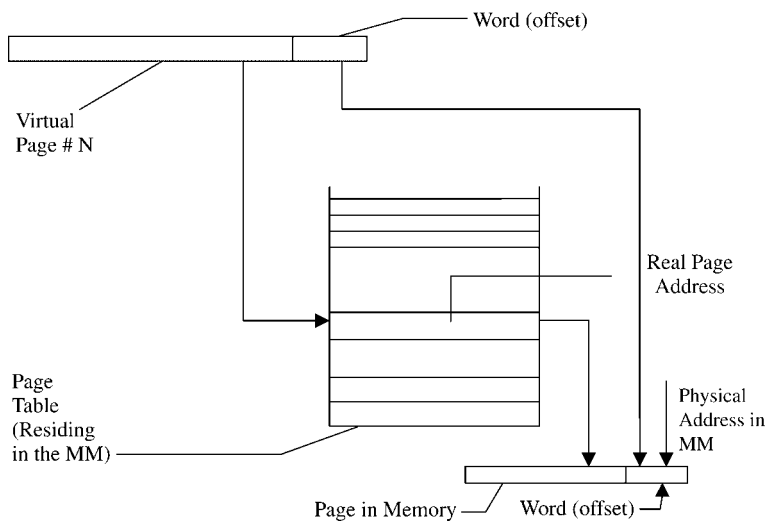


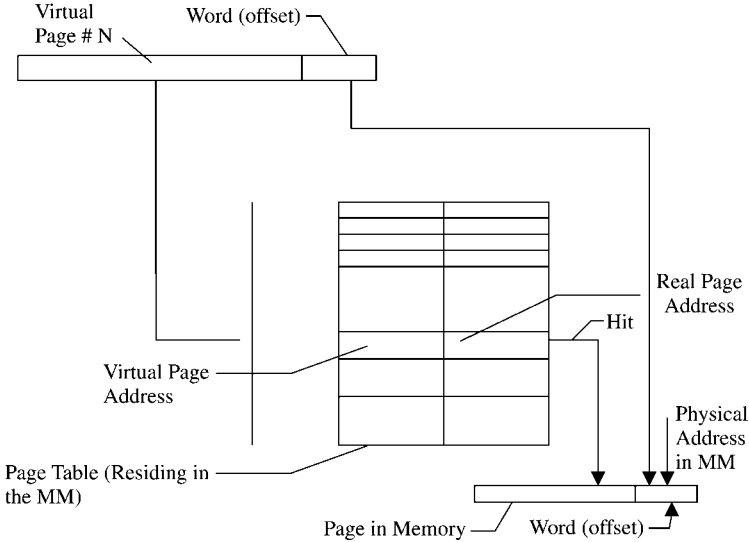
Figure 7.9 Direct-mapping virtual address translation

The main advantage of the direct-mapping technique is its simplicity measured in terms of the direct addressing of the page table entries. Its main disadvantage is the expected large size of the page table. In order to overcome the need for a large page table, the associative-mapping technique, which is explained below, is used.

7.2.2. Associative Mapping

Figure 7.10 illustrates the address translation according to the associative mapping technique. The technique is similar to direct mapping in that the virtual address issued by the processor is divided into two fields: the virtual page number and the offset fields. However, the page table used in associative mapping could be far shorter than its direct mapping counterpart. Every entry in the page table is divided into two parts: the virtual page number and the physical page number. A match is searched (associatively) between the virtual page number field of the address and the virtual page numbers stored in the page table. If a match is found, the corresponding physical page number stored in the page table is extracted and is concatenated with the offset field in order to generate the physical address of the word requested by the processor. If, on the other hand, a match could not be found, then this represents a page fault. In this case, the MMU will have to bring the corresponding page from the hard disk, load it into the main memory, and indicate the validity of the page. The translation process is then carried out as explained before.

The main advantage of the associative-mapping technique is the expected shorter page table (compared to the direct-mapping technique) required for the translation process. Its main disadvantage is the search required for matching the virtual



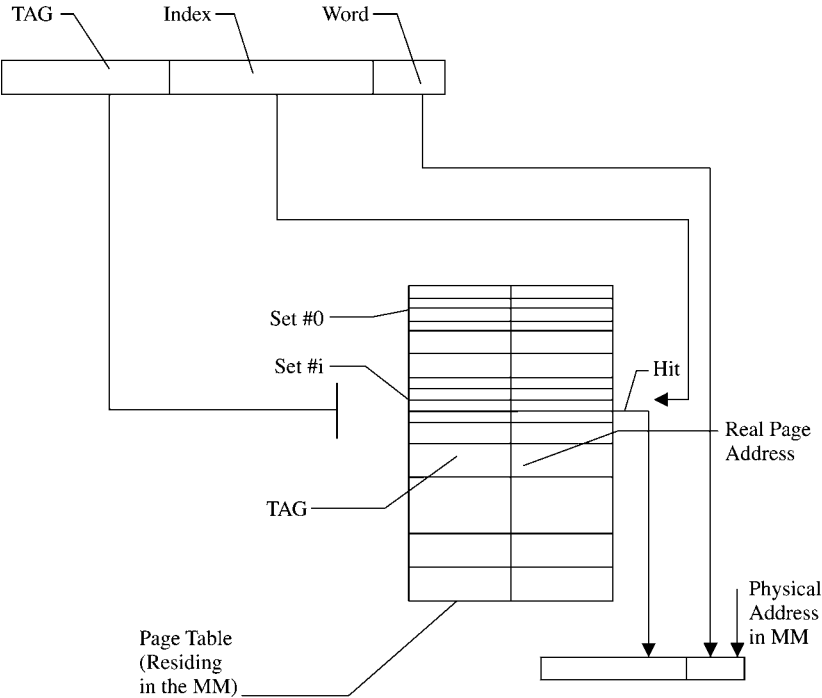
**Figure 7.10** Associative mapping address translation

page number field and all virtual page numbers stored in the page table. Although such a search is done associatively, it requires the use of an added hardware overhead.

A possible compromise between the complexity of the associative mapping and the simplicity of the direct mapping is the set-associative mapping technique. This hybrid technique is explained below.

### 7.2.3. Set-Associative Mapping

Figure 7.11 illustrates the address translation according to the set-associative mapping. In this case, the virtual address issued by the processor is divided into three fields: the tag, the index, and the offset. The page table used in set-associative mapping is divided into sets, each consisting of a number of entries. Each entry in the page table consists of a tag and the corresponding physical page address. Similar to direct mapping, the index field is used to directly determine the set in which a search should be conducted. If the number of bits in the index field is  $S$ , then the number of sets in the page table should be  $2^S$ . Once the set is determined, then a search (similar to associative mapping) is conducted to match the tag field with all entries in that specific set. If a match is found, then the corresponding physical page address is extracted and concatenated with the offset field in order to generate the physical address of the word requested by the processor. If, on the other hand, a match could not be found, then this represents a page fault. In this case, the MMU will have to bring the corresponding page from the hard disk, load it into the main memory, update the corresponding set and indicate the validity of the page. The translation process is then carried out as explained before.



**Figure 7.11** Set-associative mapping address translation

The set-associative-mapping technique strikes a compromise between the inefficiency of direct mapping, in terms of the size of the page table, and excessive hardware overhead of associative mapping. It also enjoys the best of the two techniques: the simplicity of the direct mapping and the efficiency of the associative mapping.

It should be noted that in all the above address translation techniques extra main memory access is required for accessing the page table. This extra main memory access could potentially be saved if a copy of a small portion of the page table can be kept in the MMU. This portion consists of the page table entries that correspond to the most recent accessed pages. In this case, before any address translation is attempted, a search is conducted to find out whether the virtual page number (or the tag) in the virtual address field could be matched. This small portion is kept in the table look-aside buffer (TLB) cache in the MMU. This is explained below.

**7.2.4. Translation Look-Aside Buffer (TLB)**

In most modern computer systems a copy of a small portion of the page table is kept on the processor chip. This portion consists of the page table entries that correspond to the most recently accessed pages. This small portion is kept in the translation look-aside buffer (TLB) cache. A search in the TLB precedes that in the page table. Therefore, the virtual page field is first checked against the entries of the

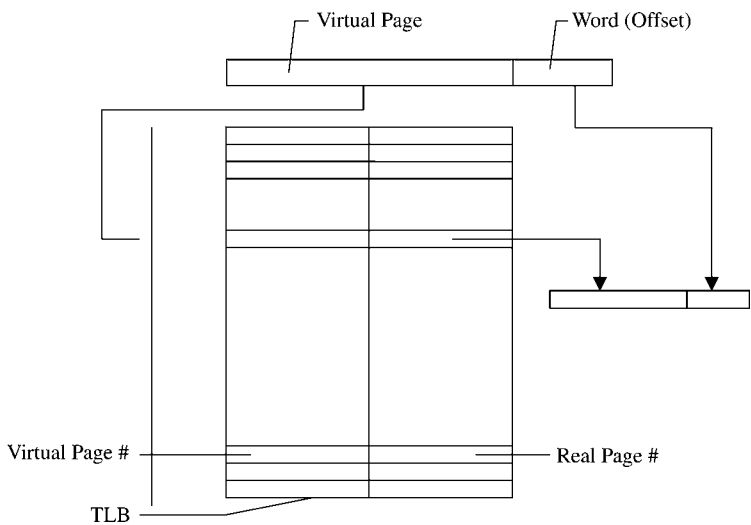
TLB in the hope that a match is found. A hit in the TLB will result in the generation of the physical address of the word requested by the processor, thus saving the extra main memory access required to access the page table. It should be noted that a miss on the TLB is not equivalent to a page fault. Figure 7.12 illustrates the use of the TLB in the virtual address translation process. The typical size of a TLB is in the range of 16 to 64 entries. With this small TLB size, a hit ratio of more than 90% is always possible. Owing to its limited size, the search in the TLB is done associatively, thus reducing the required search time.

To illustrate the effectiveness of the use of a TLB, let us consider the case of using a TLB in a virtual memory system having the following specifications.

Number of entries in the TLB = 16	Associative search time in TLB = 10 ns
Main memory access time = 50 ns	TLB hit ratio = 0.9

The average access time =  $0.9(10 + 50) + 0.1(10 + 2 * 50) = 0.9 * 60 + 0.1 * 110 = 65$  ns. This is to be compared to the 100 ns access time needed in the absence of the TLB. It should be noted that for simplicity, we overlooked the existence of the cache in the above illustration.

It is clear from the above discussion that as more requests for items that do not exist in the main memory (page faults) occur, more pages would have to be brought from the hard disk to the main memory. This will eventually lead to a totally filled main memory. The arrival of any new page from the hard disk to a totally full main memory should promote the following question: Which main memory page should be removed (replaced) in order to make room for the incoming page(s)? Replacement algorithms (policies) are explained next.



**Figure 7.12** Use of the TLB in virtual address translation

It should be noted that Intel's Pentium 4 processor has a 36-bit address bus, which allows for a maximum main memory size of 64 GB. According to Intel's specifications, the virtual memory is 64 TB (65,528 GB). This increases the processor's memory access space from  $2^{36}$  to  $2^{46}$  bytes. This is to be compared to the PowerPC 604 which has two 12-entry, two-way set-associative translation look-aside buffers (TLBs): one for instructions and the other for data. The virtual memory space is therefore  $= 2^{52} = 4$  Peta-bytes.

### 7.2.5. Replacement Algorithms (Policies)

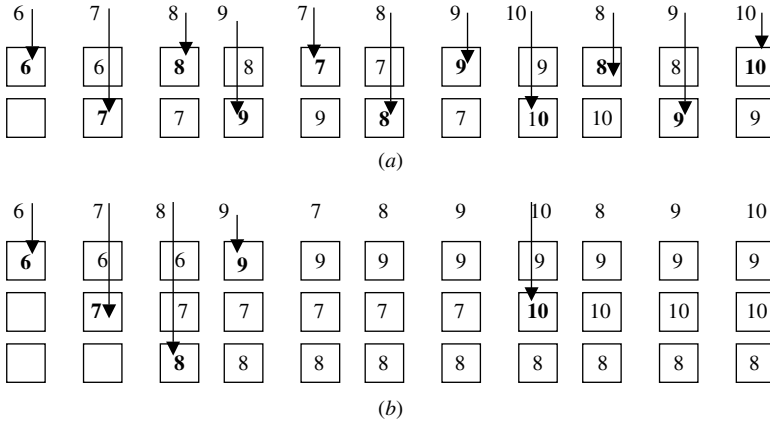
Basic to the implementation of virtual memory is the concept of *demand paging*. This means that the operating system, and not the programmer, controls the swapping of pages in and out of main memory as they are required by the active processes. When a process needs a nonresident page, the operating system must decide which resident page is to be replaced by the requested page. The technique used in the virtual memory that makes this decision is called the *replacement policy*.

There exists a number of possible replacement mechanisms. The main objective in all these mechanisms is to select for removal the page that expectedly will not be referenced in the near future.

**Random Replacement** According to this replacement policy, a page is selected randomly for replacement. This is the simplest replacement mechanism. It can be implemented using a pseudo-random number generator that generates numbers that correspond to all possible page frames. At the time of replacement, the random number generated will indicate the page frame that must be replaced. Although simple, this technique may not result in efficient use of the main memory, that is, a low hit ratio  $h$ . Random replacement has been used in the Intel *i860* family of RISC processor.

**First-In-First-Out (FIFO) Replacement** According to this replacement policy, the page that was loaded before all the others in the main memory is selected for replacement. The basis for page replacement in this technique is the time spent by a given page residing in the main memory regardless of the pattern of usage of that page. This technique is also simple. However, it is expected to result in acceptable performance, measured in terms of the main memory hit ratio, if the page references made by the processor are in strict sequential order. To illustrate the use of the FIFO mechanism, we offer the following example.

**Example** Consider the following reference string of pages made by a processor: 6, 7, 8, 9, 7, 8, 9, 10, 8, 9, 10. In particular, consider two cases: (a) the number of page frames allocated in the main memory is TWO and (b) the number of page frames allocated are THREE. Figure 7.13 illustrates a trace of the reference string for the two cases. As can be seen from the figure, when the number of page frames is TWO, there were 11 page faults (these are shown in bold in the figure). When the number of page frames is increased to THREE, the number of page



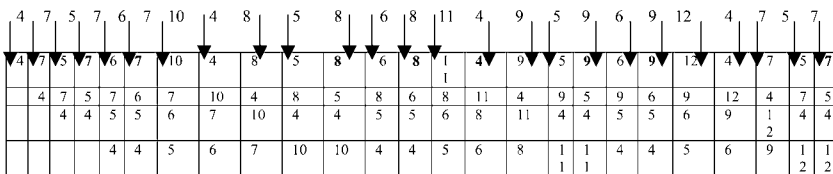
**Figure 7.13** FIFO replacement technique. (a) FIFO replacement using two page frames (#PFs = 11), (b) FIFO replacement using three page frames (#PFs = 5)

faults was reduced to five. Since five pages are referenced, this is the optimum condition. The FIFO policy results in the best (minimum) page faults when the reference string is in strict order of increasing page number references.

**Least Recently Used (LRU) Replacement** According to this technique, page replacement is based on the pattern of usage of a given page residing in the main memory regardless of the time spent in the main memory. The page that has not been referenced for the longest time while residing in the main memory is selected for replacement. The LRU technique matches most programs' characteristics and therefore is expected to result in the best possible performance in terms of the main memory hit ratio. It is, however, more involved compared to other techniques. To illustrate the use of the LRU mechanism, we offer the following example.

**Example** Consider the following reference string of pages made by a processor: 4, 7, 5, 7, 6, 7, 10, 4, 8, 5, 8, 6, 8, 11, 4, 9, 5, 9, 6, 9, 12, 4, 7, 5, 7. Assume that the number of page frames allocated in the main memory is FOUR. Compute the number of page faults generated. The trace of the main memory contents is shown in Figure 7.14. Number of page faults = 17.

In presenting the LRU, we have a particular implementation, called *stack-based LRU*. In this implementation, the most recently accessed page is now represented by



**Figure 7.14** LRU replacement technique

the top page rectangle. The rectangles do not represent specific page frames as they did in the FIFO diagram. Thus, each reference generating a page fault is now on the top row. It should be noted that as more pages are allotted to the program the page references in each row do not change. Only the number of page faults changes. This will make the set of pages in memory for  $n$  page frames be a subset of the set of pages for  $n + 1$  page frames. In fact, the diagram could be considered a STACK data structure with the depth of the stack representing the number of page frames. If a page is not on the stack (i.e., is found at a depth greater than the number of page frames), then a page fault occurs.

**Example** Consider the case of a two-dimensional  $8 \times 8$  array A. The array is stored in row-major order. For THREE page frames, compute how many page faults are generated by the following array-initialization loop. Assume that an LRU replacement algorithm is used and that all frames are initially empty. Assume that the page size is 16.

```
for I = 0 to 7 do
  for J = 0 to 7 do
    A[I, J] = 0;
  End for
End for
```

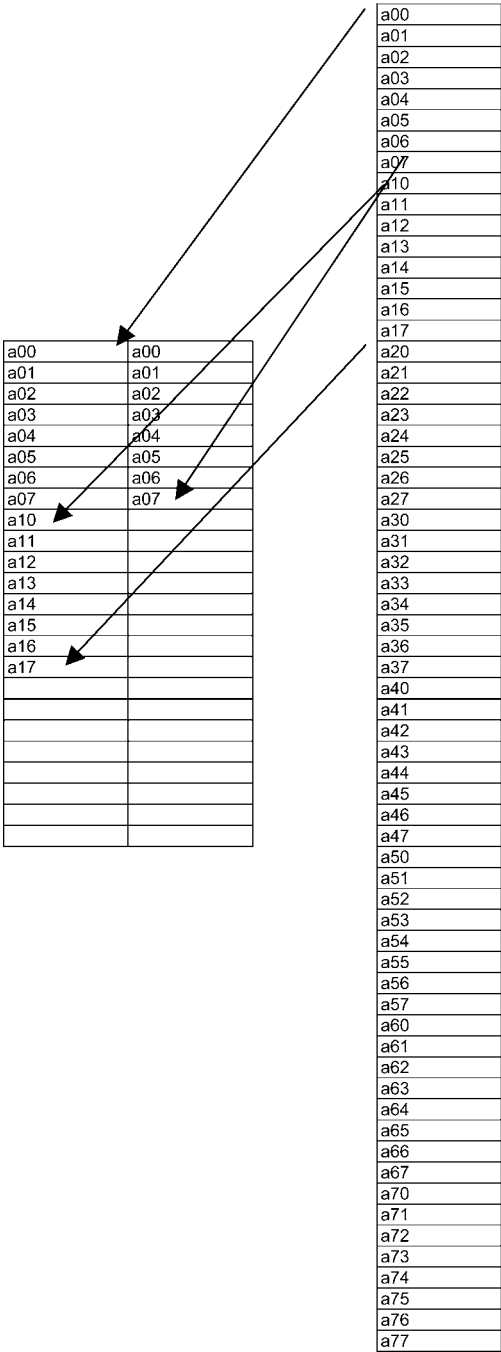
The arrangement of the array elements in the secondary storage is shown in Figure 7.15. The sequence of requests for the array elements in the first TWO external loop executions is as follows:

```
I = 0
  J = 0, 1, 2, 3, 4, 5, 6, 7
     $a_{00}, a_{01}, a_{02}, a_{03}, a_{04}, a_{05}, a_{06}, a_{07}$       The number of page faults (PFs) = 1
I = 1
  J = 0, 1, 2, 3, 4, 5, 6, 7
     $a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}$       The number of page faults (PFs) = 1
```

From the above analysis, it is clear that there will be one PF in every external loop execution. This makes the total number of PFs be 8. It should be noted that if the array was stored column-major, then every internal loop execution would generate eight page faults, thus causing the total number of PFs to become 64.

**Clock Replacement Algorithm** This is a modified FIFO algorithm. It takes into account both the time spent by a page residing in the main memory (similar to the FIFO) and the pattern of usage of the page (similar to the LRU). The technique is therefore sometimes called the First-In-Not-Used-First-Out (FINUFO). In keeping track of both the time and the usage, the technique uses a pointer to





**Figure 7.15** Arrangement of array elements in secondary storage and main memory built up

indicate where to place the incoming page and a used bit to indicate the usage of a given page. The technique can be explained using the following three steps.

1. If the used bit = 1, then reset bit, increment pointer and repeat.
2. If the used bit = 0, then replace corresponding page and increment pointer.
3. The used bit is SET if the page is referenced after the initial loading.

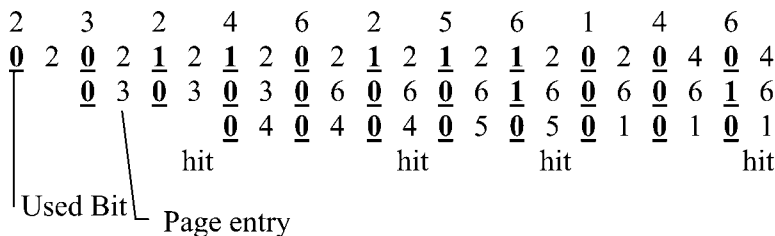
**Example** Consider the following page requests (Fig. 7.16) in a THREE-page frames MM system using the FINUFO technique: 2,3,2,4,6,2,5,6,1,4,6. Estimate the hit ratio. The estimated Hit Ratio = 4/11.

### 7.2.6. Virtual Memory Systems with Cache Memory

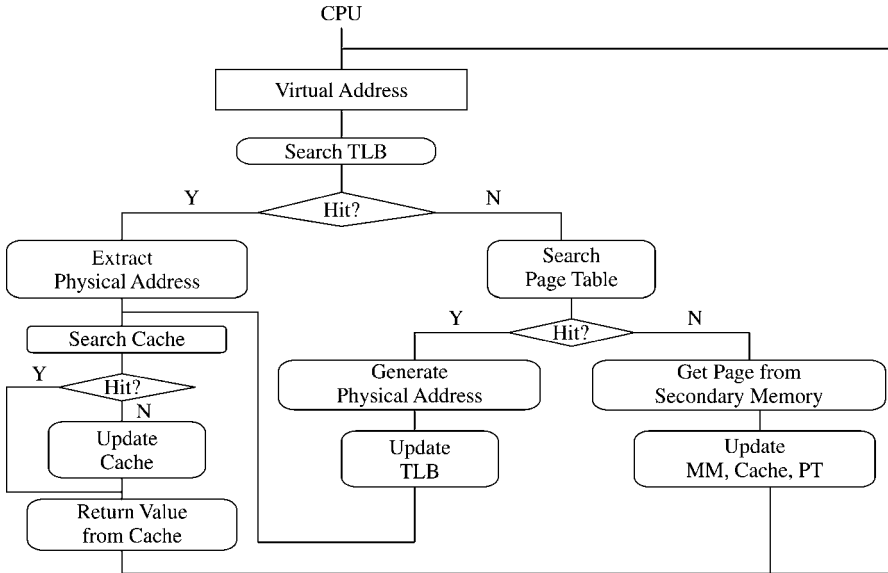
A typical computer system will contain a cache, a virtual memory, and a TLB. When a virtual address is received from the processor, a number of different scenarios can occur, each dependent on the availability of the requested item in the cache, the main memory, or the secondary storage. Figure 7.17 shows a general flow diagram for the different scenarios.

The first level of address translation checks for a match between the received virtual address and the virtual addresses stored in the TLB. If a match occurs (TLB hit) then the corresponding physical address is obtained. This physical address can then be used to access the cache. If a match occurs (cache hit) then the element requested by the processor can be sent from the cache to the processor. If, on the other hand, a cache miss occurs, then the block containing the targeted element is copied from the main memory into the cache (as discussed before) and the requested element is sent to the processor.

The above scenario assumes a TLB hit. If a TLB miss occurs, then the page table (PT) is searched for the existence of the page containing the targeted element in the main memory. If a PT hit occurs, then the corresponding physical address is generated (as discussed before) and a search is conducted for the block containing the requested element (as discussed above). This will require updating the TLB. If on the other hand a PT miss takes place (indicating a page fault), then the page containing the targeted element is copied from the disk into the main memory, a block is copied into the cache, and the element is sent to the processor. This last



**Figure 7.16** FINUFO replacement technique



**Figure 7.17** Memory hierarchy accesses scenarios

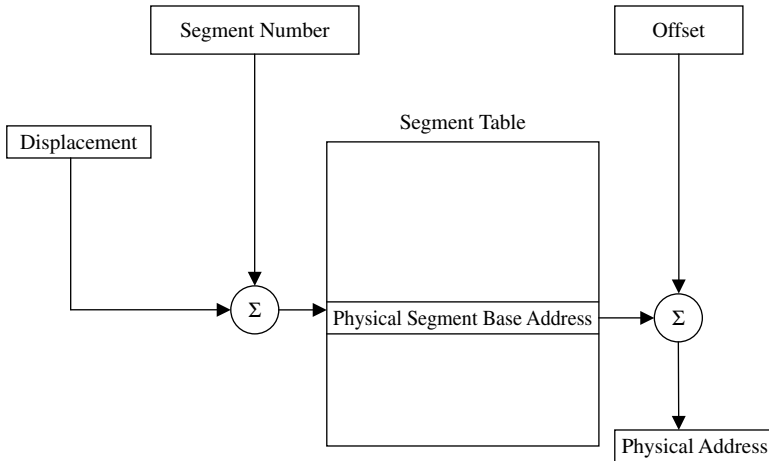
scenario will require updating the page table, the main memory, and the cache. A subsequent request of that virtual address by the processor will result in updating the TLB.

### 7.2.7. Segmentation

A segment is a block of contiguous locations of varying size. Segments are used by the operating system (OS) to relocate complete programs in the main and the disk memory. Segments can be shared between programs. They provide means for protection from unauthorized access and/or execution. It is not possible to enter segments from other segments unless the access has been specifically allowed. Data segments and code segments are separated. It should also not be possible to alter information in the code segment while fetching an instruction nor should it be possible to execute data in a data segment.

### 7.2.8. Segment Address Translation

In order to support segmentation, the address issued by the processor should consist of a segment number (base) and a displacement (or an offset) within the segment. Address translation is performed directly via a segment table. The starting address of the targeted segment is obtained by adding the segment number to the contents of the segment table pointer. One important content of the segment table is the



**Figure 7.18** Segment address translation

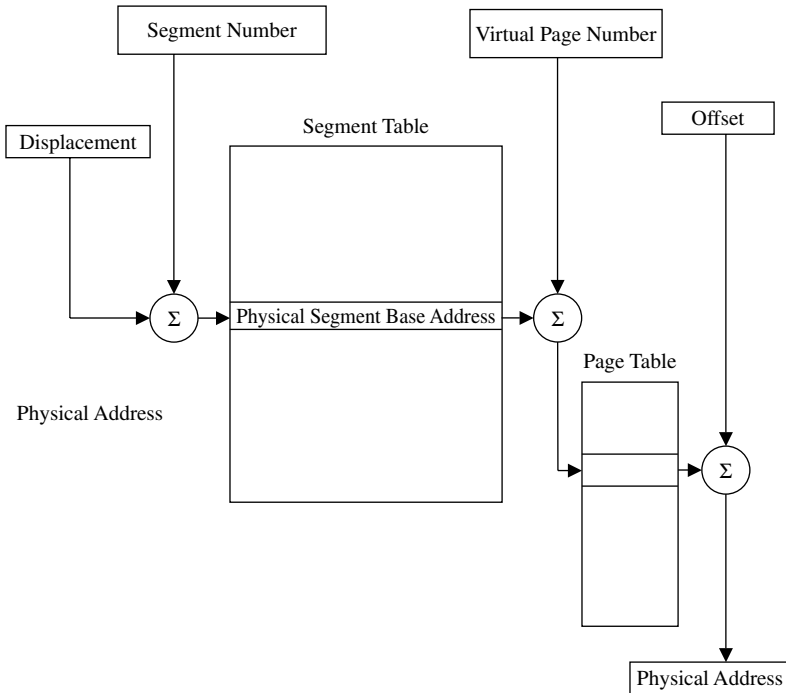
physical segment base address. Adding the latter to the offset yields the required physical address. Figure 7.18 illustrates the segment address translation process.

Possible additional information included in the segment table includes:

1. Segment length
2. Memory protection (read-only, execute-only, system-only, and so on)
3. Replacement algorithm (similar to those used in the paged systems)
4. Placement algorithm (finding a suitable place in the main memory to hold the incoming segment). Examples include
  - (a) First fit
  - (b) Best fit
  - (c) Worst fit

### 7.2.9. Paged Segmentation

Both segmentation and paging are combined in most systems. Each segment is divided into a number of equal sized pages. The basic unit of transfer of data between the main memory and the disk is the page, that is, at any given time, the main memory may consist of pages from various segments. In this case, the *virtual address* is divided into a *segment number*, a *page number*, and *displacement within the page*. Address translation is the same as explained above except that the physical segment base address obtained from the segment table is now added to the virtual page number in order to obtain the appropriate entry in the page table. The output of the page table is the page physical address, which when concatenated with the word field of the virtual address results in the physical address. Figure 7.19 illustrates the paged segmentation address translation.



**Figure 7.19** Paged segmentation address translation

### 7.2.10. Pentium Memory Management

In the Pentium processor, both segmentation and paging are individually available and can also be disabled. Four distinct views of the memory exist:

1. Unsegmented unpagged memory
2. Unsegmented paged memory
3. segmented unpagged memory
4. segmented paged memory

For segmentation, the 16-bit segment number (two of which are used for protection) and the 32-bit offset produce a segmented virtual address space equal to  $2^{46} = 64$  terabytes. The virtual address space is divided into two parts: one half, that is,  $8K \times 4$  GB, is global and shared by all processes, and the other half is local and is distinct for each process.

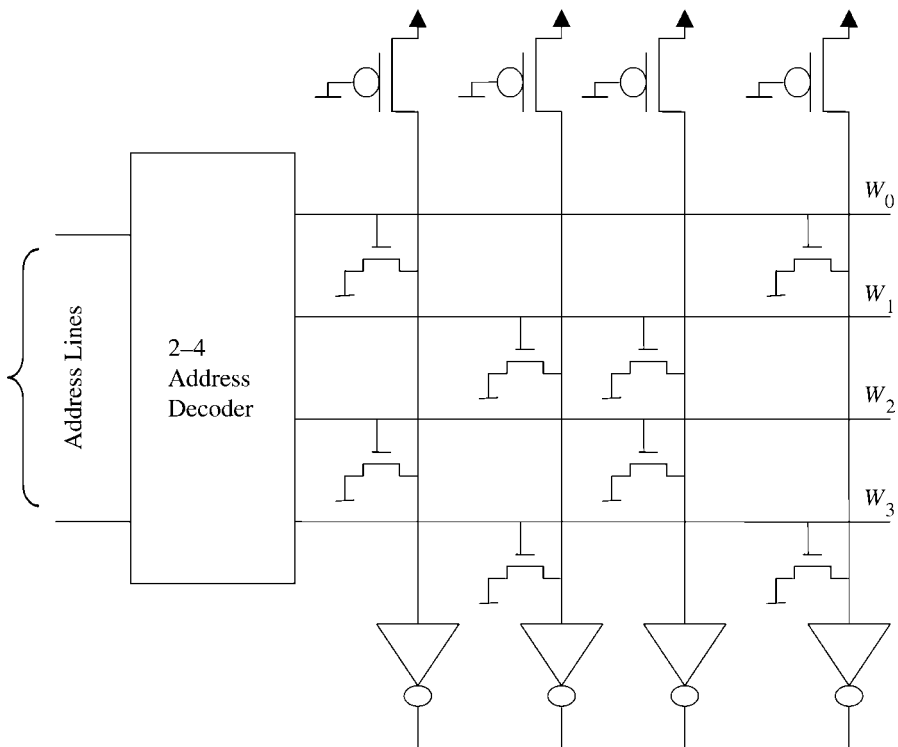
For paging, a two-level table lookup paging system is used. First level is a page directory with 1024 entries, that is, 1024 page groups, each with its own page table and each FOUR MB in length. Each page table contains 1024 entries; each entry corresponds to a single 4 KB page.

### 7.3. READ-ONLY MEMORY

Random access as well as cache memories are examples of *volatile* memories. A volatile storage is defined as one that loses its contents when power is turned off. *Nonvolatile* memory storages are those that retain the stored information if power is turned off. As there is a need for volatile storage there is also a need for nonvolatile storage. Computer system boot subroutines, microcode control, and video game cartridges are a few examples of computer software that require the use of nonvolatile storage. Read-only memory (ROM) can also be used to realize combinational logic functions.

The technology used for implementing ROM chips has evolved over the years. Early implementations of ROMs were called *mask-programmed ROMs*. In this case, a made-to-order one time ROM is programmed according to a specific encoding pattern supplied by the user. The structure of a  $4 \times 4$  CMOS ROM chip is shown in Figure 7.20.

In this figure an n-type transistor is placed where a 1 is to be stored. A two-to-four address decoder is used to create four word lines; each is used to activate a row of transistors. When a 1 appears on the word line, the corresponding transistors will be turned on, thus pulling the corresponding bit line to 0. An inverter at the output of the



**Figure 7.20** Example of a  $4 \times 4$  CMOS ROM chip

**TABLE 7.3 Patterns Stored at Four ROM Locations**

Address lines	Word line activated	Output pattern
00	$W_0$	1001
01	$W_1$	0110
10	$W_2$	1010
11	$W_3$	0101

bit lines is used to output a 1 at the output of those pulled down bit lines. Table 7.3 shows the patterns stored at each of the four ROM locations.

Mask-programmed ROMs are primarily used to store machine microcode, desktop bootstrap loaders, and video game cartridges. Because they can be programmed only once by the manufacturer, mask-programmed ROMs are inflexible. If the user would like to program his/her ROM on site, then a different type of ROM, called the Programmable ROM (PROM) should be used. In this case, fuses, instead of transistors, are placed at the intersection of word and bit lines. The user can program the PROM by selectively blowing up the appropriate fuses. This can be done by allowing a high current to flow in those particular fuses, thus causing them to blow up. This process is known as “burning the ROM.”

Although it allows for some added flexibility, PROM is still restricted by the fact that it can only be programmed once (by the user). A third type of ROM, called Erasable PROM (EPROM), is reprogrammable; that is, it allows stored data to be erased and new data to be stored. In order to provide such flexibility, EPROMs are constructed using a special type of transistors. These transistors are able to assume one of two statuses, normal or disabled. A disabled transistor acts like a switch that is turned off all the time. A normal transistor can be programmed to become open all the time by inducing a certain amount of charge to be trapped under its gate. A disabled transistor can become normal again by removing the induced charge. This requires exposing those transistors to ultraviolet light. Exposing the EPROM chip to such ultraviolet light will lead to the erasure of the entire chip contents. This is considered a major drawback of EPROMs. Both PROMs and EPROMs are used in prototyping, of moderate size systems.

Flash EPROMs (FEPRoms) have emerged as strong contenders to EPROMs. This is because FEPRoms are more compact, faster, and removable compared to EPROMs. The erasure time of a FEPRom is far faster than that of an EPROM.

A different type of ROM, which overcomes the drawback of the EPROM, is the Electrically EPROM or EEPROM. In this case, the erasure of the EPROM can be done electrically and, moreover, selectively; that is, only the contents of selective cells can be erased, leaving the other cells’ contents untouched. Both FEPRoms and EEPROMs are used in applications requiring occasional updating of information, such as Programmable TVs, VCR, and automobiles.

Table 7.4 summarizes the main characteristics of the different types of ROM discussed above.

**TABLE 7.4    Characteristics of Different ROM Implementations**

ROM type	Cost	Programmability	Typical applications
Mask-programmed ROM	Truly inexpensive	Once at manufacture	Microcode
PROM	Inexpensive	Once on site	Prototyping
EPROM	Moderate	Many times	Prototyping
FEPR0M	Expensive	Many times	VCR & TVs
EEPROM	Truly expensive	Many times	VCR & TVs

**7.4. SUMMARY**

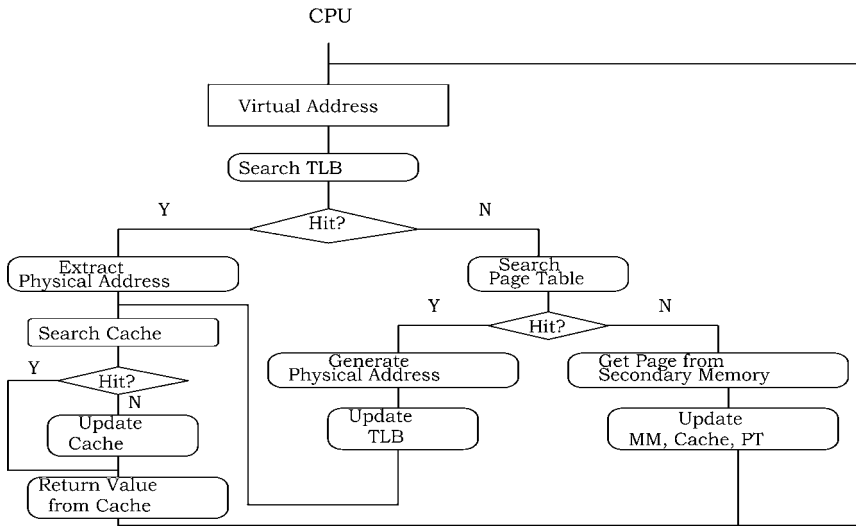
The discussion in this chapter has been a continuation of that conducted in Chapter 6. In particular, this chapter has been dedicated to cover the design aspects that relate to the internal and external organization of the main memory. The design of a static RAM cell was introduced with emphasis on the read and write operations. Our discussion on virtual memory started with the issues related to address translation. Three address translation techniques were discussed and compared. These are the direct, associative, and the set-associative techniques. The use of a TLB to improve the average access time was explained. Three replacement techniques were introduced. These are the FIFO, LRU, and clock replacement. Segmented paged systems were also introduced. Our discussion on virtual memory ended with an explanation of the virtual memory aspects of the Pentium IV processor. Toward the end of the chapter, we have touched on a number of implementations for ROMs.

**EXERCISES**

1. Consider the case of a computer system employing both a cache and a paged virtual memory as shown below (Fig. 7.21). One can analyze this system through identifying FIVE combinations of accesses. What are these combinations? Determine the probability and the access time in each case assuming the following information. Compute also the overall average access time.

TLB address translation and search	25 ns
Cache search time to determine whether address in cache	25 ns
Cache access time	25 ns
Main memory access time	250 ns
Hard disk access time	100 ms
TLB hit ratio	0.9
Cache hit ratio	0.95
Main memory hit ratio	0.8
2. A 64×64 array of words (elements) is to be “normalized” as follows. For each row, the largest element is found and all elements of the row are divided by





**Figure 7.21** Computer system with cache and paged virtual memory

this maximum value. Assume that each page in the virtual memory consists of 64 words, and that 2K words of the main memory are allocated for storing data during this computation. Suppose that it takes 100 ms to load a page from the disk into the main memory when a page fault occurs.

- (a) Write a simple piece of code (in a notational form) that can perform the above job.
  - (b) How many page faults would occur if the elements of the array are stored in column order in the virtual memory?
  - (c) How many page faults would occur if the elements of the array are stored in row order in the virtual memory?
  - (d) Estimate the total time needed to perform this normalization for both arrangements (b) and (c).
3. Design a  $64\text{M} \times 8\text{-bit}$  memory using a number of  $16\text{M} \times 1\text{-bit}$  static RAM chips. Assume that each individual chip has a chip select ( $\overline{\text{CS}}$ ) line and a read/write ( $R/\overline{W}$ ) line. Compute the number of chips required and show a complete connection diagram of the designed memory.
  4. Consider the following stream of page requests: 1,2,3,4,5,1,2,3,4,5,1,2,3,4,5. Assume that the main memory consists of FOUR page frames. Show a trace of the status of the page frames in the MM and estimate the hit ratio assuming each of the following page replacement algorithms.
    - (a) FIFO
    - (b) LRU
    - (c) FINUFO
  5. Consider the case of a two-dimensional  $20 \times 20$  array A. The array is stored column-major. For FIVE main memory page frames, compute how many

*page faults* are generated by the following array-initialization loop. Assume that *LRU* replacement algorithm is used and that all frames are initially empty. Assume also that the page size is 40 elements.

```
for I = 1 to 20 do
  for J = 1 to 20 do
    A[I, J] = 0;
```

6. In this problem you are asked to pick a real-life computer memory system that uses both caching and virtual memory schemes. Your job is to apply all the knowledge that you have gained in this chapter concerning “Memory System Design and Analysis” in describing and analyzing your selected system. Examples that you may pick include but are not limited to Intel Pentium 4, The PowerPC, Alpha AXP 21064, and so on. Use examples and illustrations to support your analysis and be as specific as possible. Remember that the emphasis should be on the analysis and the basic design issues involved. You can use any reference material but make sure that you include it in your reference list.

## REFERENCES AND FURTHER READING

- S. D. Burd, *Systems Architecture*, 3rd ed., Thomson Learning Ltd, Boston, 2001.
- H. Cragon, *Memory Systems and Pipelined Processors*, Jones and Bartlett: Sudbury, MA, 1996.
- V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed., McGraw-Hill, NY, 2002.
- J. L. Hennessy, and Patterson, D. A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 1996.
- V. P. Heuring, and H. F. Jordan, *Computer Systems Design and Architecture*, Addison-Wesley, NJ, USA, 1997.
- D. A. Patterson, and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann, San Mateo, CA, 1994.
- H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, Amsterdam, Netherlands, 1987.
- B. Wilkinson, *Computer Architecture: Design and Performance*, Prentice-Hall, Hertfordshire, UK, 1996.

## Websites

<http://www.sysopt.com>

<http://www.intel.com>

<http://www.AcerHardware.com>

<http://www.pmc-sierra.com/products/details/rm7000a>

[http://physinfo.ulb.ac.be/divers\\_html/PowerPC\\_Programming\\_Info/into\\_to\\_ppc/ppc2\\_hardware.html](http://physinfo.ulb.ac.be/divers_html/PowerPC_Programming_Info/into_to_ppc/ppc2_hardware.html)