consecutive words. Main memory M_2 is similarly divided into one-block regions M_2 (0), M_2 (1), ... M_2 ($s_2 - 1$). With direct mapping, each block M_2 (i) in M_2 is mapped into one specific M_1 (j) in M_1. The set address j is determined from I by the rule

$$\mathbf{J = i \quad (modulo\ s_1)}$$

For example, if $s_1 = 2$ as in Fig. 6.31, every even-address (unshaded) block in M_2 is mapped into M_1 (0) and every odd-address (shaded) block in M_2 is mapped into M_1 (1). The hardware needed to implement direct mapping is fairly simple. The low-order s bits of each block address A form a set address that identifies the unique cache set that can store the block in question. The remaining t high-order bits of A now constitute the tag, and only these bits need be stored in the cache's tag memory.

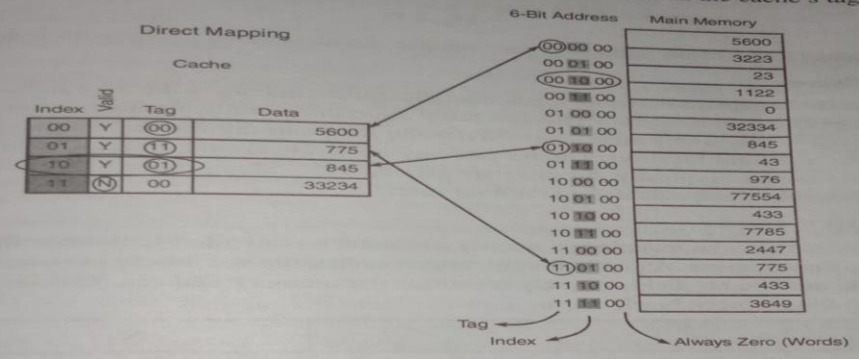


Fig. 6.32. Cache Data Transfer

Consequently, the cache tag memory can be an ordinary RAM that is addressed by the s-bit set-address part of an incoming memory address A. If there are 2^d words per set, then the low-order d bits of A form the displacement address of the word in question within its block. Thus an incoming address has three parts: a t-bit tag, an s-bit set address, and a d-bit displacement.

The main drawback of direct mapping is that the cache's hit ratio drops sharply if two or more frequently used blocks happen to map onto the same region in the cache. This possibility is minimized by the fact that such blocks are relatively for apart in memory-address space. For example, $s_1 = 2^6 = 64$, then only the blocks with addresses i, i + 64, i + 128, i + 1102 can be mapped into the same cache set M_1 (i).

Accessing a Direct-Mapped Cache

Example 1. Show cache addressing for a byte-addressable memory with 32-bit addresses. Cache line W = 16 B. Cache size L = 4096 lines (64 KB).

Solution. Byte offset in line is $\log_2 16 = 4$b. Cache line index is $\log_2 4096 = 12$ b. This leaves 32 – 12 – 4 = 16 b for the tag.

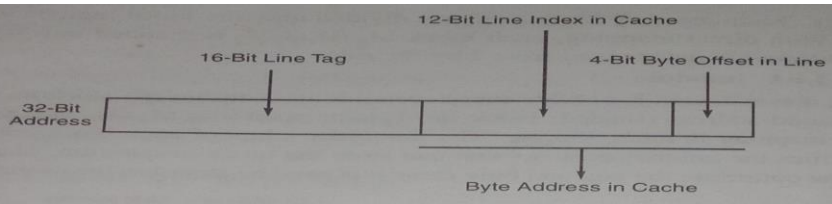


Fig. 6.33

Components of the 32-bit address is an example direct-mapped cache with byte addressing.

Hit and Misses

When the CPU reads from memory, calculate the index and tag is the data in the cache ? Yes – a hit, you're done! Data not in cache ? This is a miss. Read the word from memory, give it to the CPU. *Update* the cache so we won't miss again. Write the data and tag for this memory location to the cache. (Exploits temporal locality). The hit rate and miss rate are the fraction of memory accesses that are hits and misses. Typically, hit rates are around 95%. Many times instructions and data are considered separately when calculating hit/miss rates.

Set Associative Cache

In set associative cache mapping, a memory reference is divided into three fields: TAG, SET, and WORD, as shown above. As with direct-mapped cache, the word field chooses the word within the cache block, and the tag field uniquely identifies the memory address. The set field determines the set to which the memory block maps.

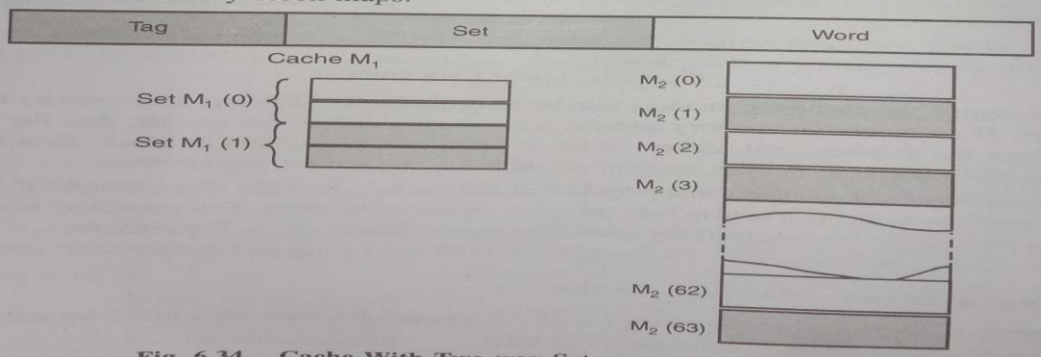


Fig. 6.34. Cache With Two-way Set-associative Addressing

A more general address mapping method for caches, called **set associative,** includes pure associative and direct mapping as special cases. As in direct mapping, blocks in main memory M_2 are grouped into equivalence classes determined by their addresses.M_2 (i), and M_2 (j) are in same

equivalence class E if I=j (modulo s_1 multiblock regions $M_1(0)$, $M_1(1)$,....., $M_1(s_1 - 1)$ called sets, each of which accommodates $k = 2^h$ blocks. A block $M_2(i)$ in M_2 is mapped into set $M_1(h)$, satisfying the condition I=h (modulo s_1). (Fig. 6.34)

Each set $M_1(h)$ in the cache is effectively a small associative memory, so address mapping within each set is associative. This k-way set-associative mapping permits up to k members of the same equivalence class E to be stored in the cache simultaneously, which is not possible with direct mapping. Figure 6.34 illustrates set-associative mapping with cache size $s_1 = 2$, implying that k equals the block capacity of the cache. Intermediate values of k lead to address-mapping methods requiring an intermediate amount of associative hardware. Only small value of k, such as k = 2 or 4, are used in practice which makes it feasible to use low-cost RAM chips.

Fully Associative Cache

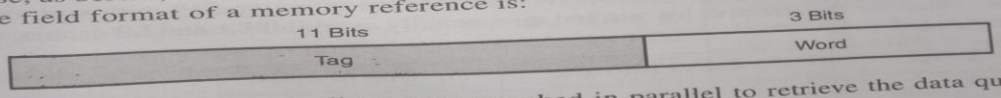
Each main memory block has a specific location to which it maps in cache; when a main memory address is converted to a cache address, the CPU knows exactly where to look in the cache for that memory block by simply examining the bits in the block field. This requires the entire cache to be built from *associative memory* so it can be searched in parallel. That is, a single search must compare the requested tag to *all* tags in cache to determine whether the desired data block is present in cache. Associative memory requires special hardware to allow associative searching, and is, thus, quite expensive.

DO YOU KNOW

Direct mapped cache is not as expensive as other caches because the mapping scheme does not require any searching.

With direct mapping, if a block already occupies the cache location where a new block must be placed, the block currently in cache is removed (it is written back to main memory if it has been modified or simply overwritten if it has not been changed). With fully associative mapping, when cache is full, we need a replacement algorithm to decide which block we wish to throw out of cache (we call this our *victim block*). A simple first-in, first-out algorithm would work, as would a least-recently used algorithm. Instead of placing memory blocks in specific cache locations based on memory address, we could allow a block to go anywhere in cache. In this way, cache would have to fill up before any blocks are evicted. This is how *fully associative* cache works. A memory address is partitioned into only two fields: the tag and the word.

Suppose, as before, we have 14-bit memory addresses and a cache with 16 blocks, each block of size 8. The field format of a memory reference is:



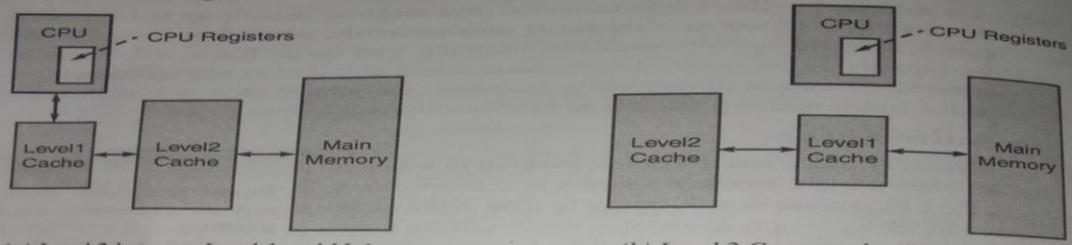
When the cache is searched, all tags are searched in parallel to retrieve the data quickly. This requires special, costly hardware.

Cache Types

Caches are distinguished by the kinds of information they store.

1. Instruction I-Cache Stores Instructions Only,
2. Data or D-cache Stores Data Only.

Separating the stored data in this way recognizes the different access behavior patterns of instructions and data. For example, programs tend to involve few write accesses, and they often exhibit more temporal and spatial locality than the data they process. A cache that stores both instructions and tat is referred to as unified. A split cache, on the other hand, consists of two associated but largely independent units: an I-cache for instructions and D-cache for data. While a unified cache is simpler, a split cache makes it possible to access programs and data concurrently. A split cache can also be designed to manage its I-and D-cache components differently.



(a) Level 2 between Level 1 and Main *(b) Level 2 Connected to "Backside" Bus*

Fig. 6.35. Multi-Level Cache

Caches are also classified by the level they occupy in the memory hierarchy. Early computers employed a single, multichip cache that occupied one level of the hierarchy between the CPU and main memory. Two developments made it desirable to introduce two or more cache levels in high-performance systems: the feasibility of including part of the real memory space on a microprocessor chip and growth in the size (but not the speed) of main memory in typical computers. A level 1 (L1) or primary cache is an efficient way to implement an on-chip memory. An additional memory level can be introduced via an off-chip, level 2 (L2) or secondary cache. The desirability of an L2 cache increases with the size of main memory, assuming that the size of the on-chip, L1 cache is fixed. As main-memory size increases further, even more cache levels may be desirable.

Performance of a Two-Level Cache System

Example 1. A system with L1 and L2 caches has a CPI of 1.2 with no cache miss. There are 1.1 memory accesses on average per instruction. What is the effective CPI with with cache misses factored in ? What are the effective hit rate and miss penalty overall if L1 and L2 caches are modeled as a single cache ?

Level	Local hit rate	Mis penalty
L1	95%	8 cycles
L2	80%	60 cycles

Solution. $C_{eff} = C_{fast} + (1 - h_1)\,[C_{medium} + (1 - h_2)C_{slow}]$

Because C_{fast} is included in the CPI of 1.2, we must account for the rest

$CPI = 1 + 2 + 1.1(1 - 0.95)\,[8 + (1 - 0.8)60] = 1.2 + 1.1 \times 0.05 \times 20 = 2.3$

Overall : hit rate 99% (95% + 80% of 5%), miss penalty 60 cycles.

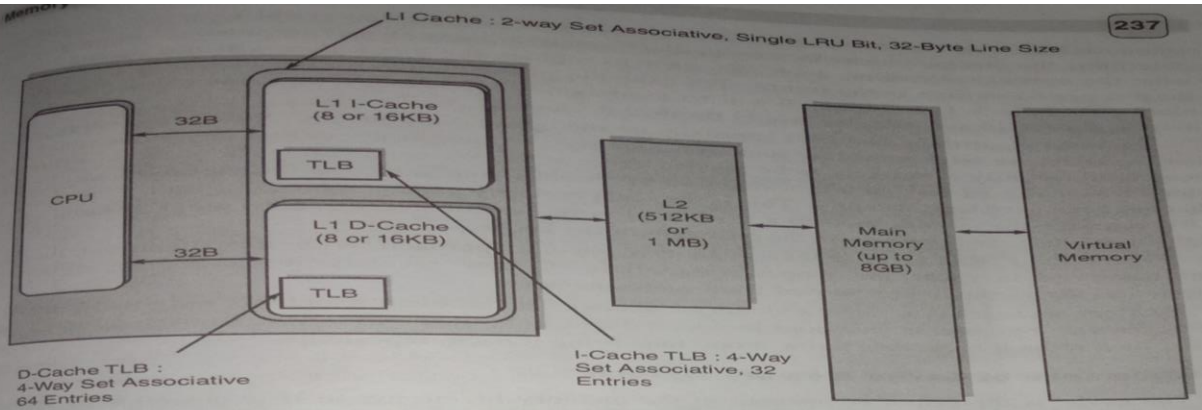


Fig. 6.36. A Pentium Processor-Two-Level Cache

The Pentium architecture supports both paging and segmentation, and they can be used in various combinations including unpaged unsegmented, segmented unpaged, and unsegmented paged. The processor supports two levels of cache (L1 and L2), both having a block size of 32 bytes. The L1 cache is next to the processor, and the L2 cache sits between the processor and memory. The L1 cache is in two parts: and instruction cache (I-cache) and a data cache (D-cache).

6.8. CACHE REPLACEMENT POLICIES

DO YOU KNOW

Direct mapping is no need for a replacement policy because the location for each new block is predetermined.

In a direct-mapped cache, if there is contention for a cache block, there is only one possible action: The existing block is kicked out of cache to make room for the new block. This process is called *replacement*. However, with fully associative cache and set associative cache, we need a replacement algorithm to determine the "victim" block to be removed from cache. When using fully associative cache, there are K possible cache locations (where K is the number of blocks in cache) to which a given main memory block may map. With N-way set associative mapping, a block can map to any of N different blocks within a given set. How do we determine which block in cache should be replaced?

The algorithm for determining replacement is called the *replacement policy*.

There are several popular replacement policies. One that is not practical but that can be used as a benchmark by which to measure all others is the *optimal* algorithm. We like to keep values in cache that will be needed again soon, and throw out blocks that won't be needed again, or that won't be needed for some time. An algorithm that could look into the future to determine the precise blocks to keep or eject based on these two criteria would be best. This is what the optimal algorithm does. We want to replace the block that will not be used for the longest period of time in the future. For example, if the choice for the victim block is between block 0 and block 1, and block 0 will be used again in 5 seconds, whereas block 1 will not be used again for 10 seconds, we would throw out block 1.

We need algorithms that best approximate the optimal algorithm. We have several options. For example, we might consider temporal locality. We might guess that any value that has not been used recently is unlikely to be needed again soon. We can keep track of the last time each block was accessed (assign a timestamp to the block), and select as the victim block the block that has been used least recently. This is the **least *recently used (LRU)*** algorithm. Unfortunately, LRU requires the system to keep a history of accesses for every cache block, which requires significant space and slows down the operation of the cache.

FIRST IN, FIRST OUT (FIFO) is another popular approach. With this algorithm, the block that has been in cache the longest (regardless of how recently it has been used) would be selected as the victim to be removed from cache memory.

Another approach is to select a victim at *random*. The problem with LRU and FIFO is that there are degenerate referencing situations in which they can be made to *thrash* (constantly throw out a block, then bring it back, then throw it out, then bring it back, repeatedly.

Performance of Cache Memory System

The cache is the fastest component in the memory hierarchy, so it is desirable to make the average memory access time Te seen by the CPU as close as possible to access time Tc of the cache.

To achieve this goal, M_1 should satisfy a very high percentage of all memory reference; that is, the cache-hit ratio h should be almost one. A high hit ratio is possible because of the locality-of-reference property discussed earlier.

Hit Ratio - % of memory accesses satisfied by Cache memory system

Let Te: Effective memory access time in Cache memory system
Tc: Cache access time
Tm: Main memory access time

$$Te = Tc + (1 - h)\, Tm$$

For example, $Tc = 0.4\ \mu s$, $Tm = 1.2\ \mu s$, $h = 0.85\%$

Hence: $Te = 0.4 + (1 - 0.85) * 1.2 = 0.58\ \mu s$

Consider a k-way set associative cache M_1 defined by the following parameters; the number of sets s_1, the number of blocks (lines) per set k, and the number of bytes per block (also called the line size) p_1. Recall that the cache is fully associative when $s_1 = 1$ and is direct-mapped when $k = 1$. The number of bytes stored in the cache's data memory, usually referred to as the cache size s_1, is given by the following formula :

$$S_1 = ks_1p_1$$

Or, in words, Cache size = Number of blocks (lines) per set + Number of sets × Number of bytes per block

6.10. VIRTUAL MEMORY

DO YOU KNOW

Virtual memory as an alternate set of memory addresses. Programs use these virtual addresses rather than real addresses to store instructions and data.

Virtual memory was developed in approximately 1959-1962, at the University of Manchester for the Atlas Computer, completed in 1962. In 1961; Burroughs released the B5000, the first commercial computer with virtual memory. When the program is actually executed, the logical/virtual addresses are converted into real memory addresses.

Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space is available equal to the totality of the auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory.

Before the development of the virtual memory technique, programmers in the 1940s and 1950s had to manage directly two-level storage such as main memory or RAM and secondary memory in the form of hard disks or earlier, magnetic drums. The VM concept enlarges the address space, the set of addresses a program can utilize. Virtual memory might contain twice as many addresses as main memory.

Objective : When a computer is executing many programs at the same time, Virtual memory make the computer to share memory efficiently. Eliminate a restriction that a computer works in memory which is small and be limited. When many programs is running at the same time, by distributing each suitable memory area to each program, VM protect programs to interfere each other in each memory area.

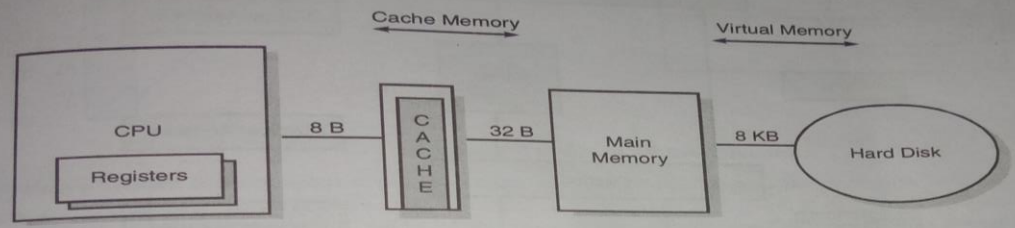
Working Principle of VM

(*i*) To facilitate copying virtual memory into real memory, the operating system divides virtual memory into **pages**, each of which contains a fixed number of addresses. Each page is stored on a disk until it is needed. When the page is needed, the operating system copies it

from disk to main memory, translating the virtual addresses into real addresses. A *physical address* is the actual memory address of physical memory.

(*ii*) Programs create *virtual addresses* that are *mapped* to physical addresses by the memory manager. *Page faults* occur when a logical address requires that a page be brought in from disk. *Memory fragmentation* occurs when the paging process results in the creation of small, unusable clusters of memory addresses.

A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.



Memory Parameters	Register	Cache	Memory	Disk Memory
1. Size:	200 B	32KB-4MB	128 MB	20 GB
2. Speed:	2 ns	4 ns	60 NS	8 ms
3. INR/Byte:	--------	2000/MB	INR 100/MB	INR 30/MB
4. Block Size:	8 B	32 B	8 KB	

Larger, Slower, Cheaper

Fig. 6.37. Virtual Memory in Memory Hierarchy

6.11. MEMORY MANAGEMENT

Definition Memory management is the hardware base that makes a virtual memory system possible and which allows software to reference physical memory by virtual addresses, quite often more than once. It accomplishes this through the use of page and page tables.

Use a section of memory to translate virtual addresses into physical addresses via a series of table lookups.

The software that handles the page fault is generally part of an operating system and the hardware that detects this situation.

The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management. Logical Address generated by the CPU; also referred to as virtual address. Physical address seen by the memory unit.

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme. Main memory and virtual memory are divided into equal sized pages. The entire address space required by a process need not be in memory at once. Some parts can be on disk, while others are in main memory.

Further, the pages allocated to a process do not need to be stored contiguously— either on disk or in memory. In this way, only the needed pages are in memory at any time, the unnecessary pages are in slower disk storage.

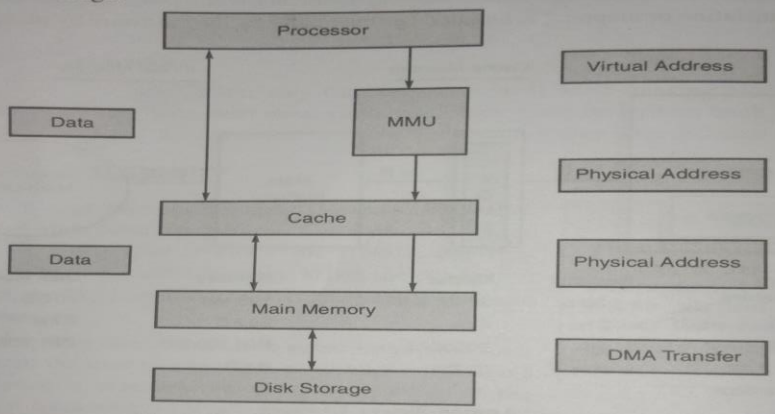


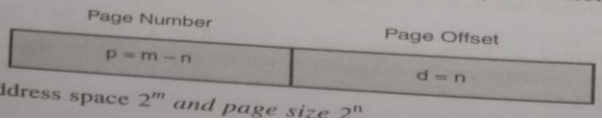
Fig. 6.38. Virtual Memory Concept Diagram

Page Tables : One for kernel addresses or more for user space processes. Page Table Entry (PTE) one per virtual page. When a process generates a virtual address, the operating system translates it into a physical memory address. To accomplish this, the virtual address is divided into two fields: A *page* field, and an *offset* field. The page field determines the page location of the address, and the offset indicates the location of the address within the page.The logical page number is translated into a physical page frame through a lookup in the page table.

Address generated by CPU is divided into :

Page number (*p*) – used as an index into a *page table* which contains base address of each page in physical memory.

Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit.



For given logical address space 2^m *and page size* 2^n

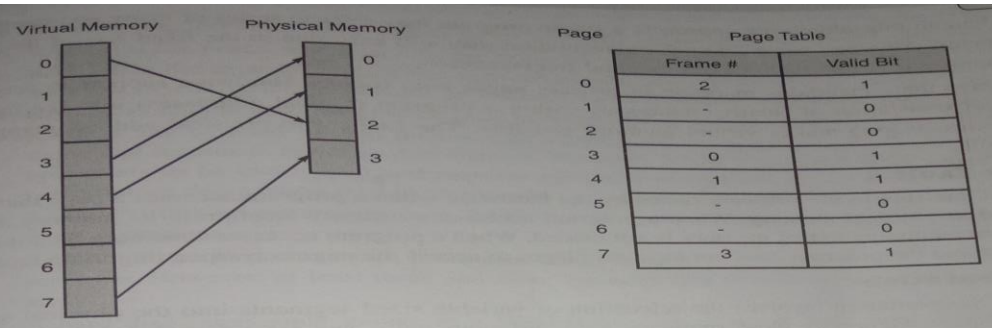


Fig. 6.39. Paging Concepts in Virtual Memory

Paging provides a somewhat easier interface for programs, in that its operation tends to be more automatic and thus transparent. Each unit of transfer, referred to as a page, is of a fixed size and swapped by the virtual memory manager outside of the program's control. Instead of utilizing a segment/offset addressing approach, as seen in segmentation, paging uses a linear sequence of virtual addresses which are mapped to physical memory as necessary.

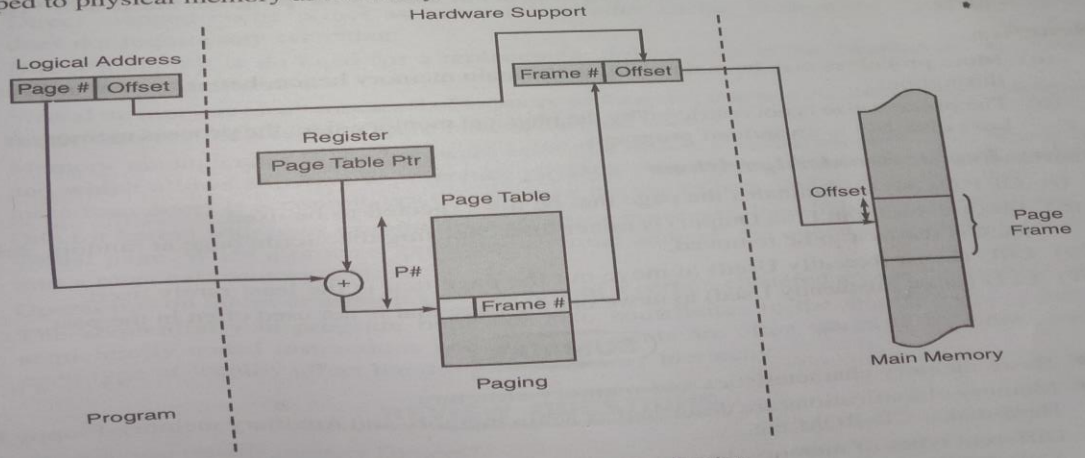


Fig. 6.40. Hardware For Paging