proceeds with little difficulty as long as the second page is also within main memory. If the second page is not in main memory, the required page is transferred into main memory as needed: this is called demand paging, or in some cases the page is transferred in advance by special look-ahead operations. Demand paging is more commonly used. Assuming that main memory is already full, the subsequent operations necessitate a replacement algorithm to determine which page is to be removed, a mapping function to determine where it is to be relocated on secondary storage, and means of keeping track of its new location. Also the location of the desired page in secondary storage must be determined. Since page replacement is a time-consuming operation, either the CPU transfers operation to a different user's program and continues processing, or it sits idle while the transfer is performed. This is a design option set by the desired system cost-performance. Once a page is transferred to main memory, many words within that page will be used more than once, such as in looping operations. Thus one rather slow transfer of information subsequently results in many fast accesses to that information, with a net gain resulting from the clustering of memory references.*

Assuming that memory references do cluster into natural page sizes of reasonable length, it can be deduced from the previous discussion that the fundamental, minimum functions necessary for a virtual memory system are as follows:

1. Page mapping function (Section 9.5).
2. Address translation (Section 9.6).
   a. Word addressing within a page.
   b. Page addressing within both primary and secondary storage.
3. Page replacement algorithm (Section 9.7).

In large, multiprogrammed systems, a number of additional practical functions are needed to produce a feasible, efficient system. Some important practical requirements in a multiuser system are as follows:

a. I/O processor and technique for efficient page relocations.
b. Storage protection.
c. Sharing of pages.

When a required page is not in primary storage, the I/O processor finds and transfers the required page, allowing the CPU to transfer to a different user and continue processing. These overlapping functions combined with cycle stealing for data transfer (Section 9.2) greatly improve the CPU utilization. Yet even without this, some advantage in efficiency over a nonvirtual system can result because of the clustering of memory references,* but the main advantage would be the virtual addressing capability. This is the situation in some small, single user virtual system. In large, multiuser virtual systems, speed and system efficiency are important as well as virtual addressing capability.

* In this case "memory references" means both program instructions and the actual alphanumeric data processed.

The storage protection function ensures that any given page of memory is not inadvertently changed or removed, nor accessed by unauthorized users. In multiuser systems, sharing of pages that contain, for instance, the supervisory program, is highly desirable to conserve memory space. In data based systems, sharing of data between users with proper access rights becomes a fundamental necessity, but this represents a more complex system than that being considered here. In a simple, two-level, multiuser virtual hierarchy, the latter three optional features are of practical importance but are not fundamentally necessary—a workable, but perhaps inefficient system can be envisioned without them. The advantages of an I/O processor, detailed in Section 9.2, are valid here as well. Sharing of pages is a complex issue that is not considered.

To better understand how the fundamental requirements interact, we now discuss a large, multiuser virtual system consisting of a disk as secondary storage and main memory as primary storage. We consider only a multiple virtual address space system: a single virtual address space is fundamentally no different and is just a simpler, limiting case.† The total logical address that addresses secondary storage consists of N_s bits, of which u bits are the user identification bits. These N_s bits can be contained in one or several registers within the CPU—typically, several are used. Each page is divided into 2^{N_r} words or bytes, where N_r represents the lowest order address bits (Fig. 9.3-1). The total number of pages in the virtual disk storage is thus 2^{N_v}, where

$$N_v = N_s - N_r \tag{9.3-1}$$

Since the number of users U must be

$$U = 2^u \tag{9.3-2}$$

* The main advantage is the saving in multiple access times. A nonvirtual system with improper memory segmentation may require numerous accesses to the disk for the same number of words processed, whereas a virtual system may use only one or a few accesses. The clustering of subsequent memory references to pages already present eliminates a large number of disk accesses but not the data transfer time.

† In practical terms, a single virtual space system does not have a separate user ID register (Fig. 9.5-2*b* and Fig. 9.8-2). Rather, the u bits are contained within N_v so each user has a virtual address space which is smaller than the total CPU address N_s but this can still be larger than the actual main memory address n_p.

from: Richard Matick, IBM

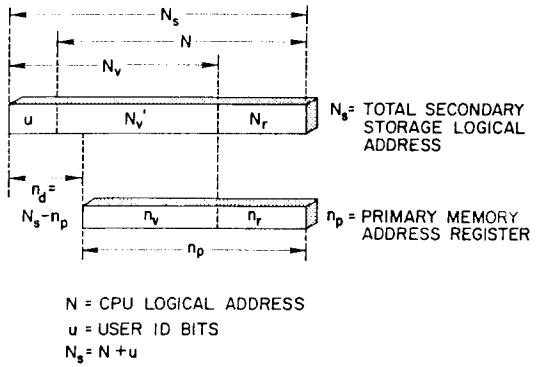

FIGURE 9.3-1 Definitions of address bits in primary and secondary addresses.

each user has a total virtual storage capacity of

$$\text{pages/user} = 2^{N_v'} \tag{9.3-3}$$

where $N_v' = N_v - u = N_s - u - N_r$, as shown schematically in Fig. 9.3-1. This is equivalent to N address bits per user, where N is the standard address length of the CPU as given by column 7 of Table 1.1-1. For instance, on the IBM systems 360/370, $N = 24$ bits and the additional user address bits u are contained in a special register (see Section 9.9). Primary storage is addressed by n_p bits, where

$$n_p < N_s \tag{9.3-4}$$

Hence it can hold only 2^{n_p} words, which is normally considerably less than 2^{N_s}. Since pages are of fixed size,

$$n_v = n_p - n_r \tag{9.3-6}$$

The CPU can reference pages through main memory only. Since in principle all $2^{N_v'}$ pages must eventually be referenced by the CPU, the page slots in main memory must be shared by many virtual pages, at different times, of course. The mapping of this large number of virtual pages into a smaller number of page slots is performed by the mapping function (Section 9.5) as indicated schematically in Fig. 9.3-2.

The addressing of such a system is considerably more complex than for an ordinary memory. Each of the N_s bit combinations represents a unique word in the virtual disk store therefore all these bits *must* be used to address main memory. However the main memory address register can only hold n_p, giving a deficiency of

$$n_d = N_s - n_p \text{ bits} \tag{9.3-7}$$

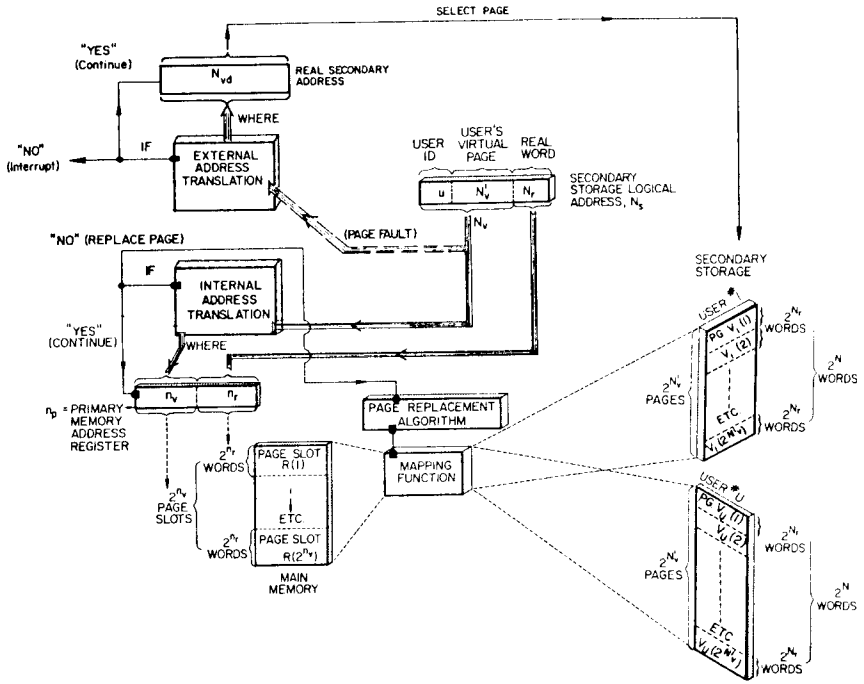

FIGURE 9.3-2 Schematic of basic functional requirements in multiuser virtual memory system, showing information flow during translation of virtual address to primary memory address.

as in Fig. 9.3-1. These deficient bits must be used in addressing main memory, hence this number of bits (but not necessarily these bit positions) *must be decoded externally*. The words or bytes within a page are addressed by the lowest order address bits, denoted as N_r in Fig. 9.3-1. These bits are real and convert directly to n_r in the primary address register (Fig. 9.3-1 or 9.3-2).* Since many virtual pages can occupy that same page slot, the remaining N_v bits must be used to indicate IF and WHERE the desired page resides in main memory. The various parts of the address translation are shown in Fig. 9.3-3. When the IF part produces a "yes," the address residing in the primary address register is used. When a "no" answer is obtained from the IF part, that page does not reside in main memory and a page relocation is necessary. The address that appears in the primary address register is aborted.

* These bits may be added to the contents of an index or base register but nevertheless are converted directly to the real address.

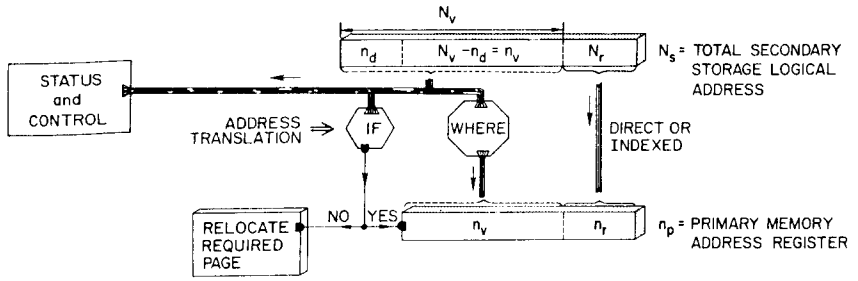

FIGURE 9.3-3 Schematic of translation of virtual logical address N_v into primary address showing IF, WHERE, and DIRECT components.

and now the N_s bits must be converted to the disk (secondary) address in Fig. 9.3-2. A separate external address translation function must be invoked which, as before, provides an IF and a WHERE. When the IF is "yes," the WHERE address is valid. When the IF is "no," that virtual page is not present in the secondary store and a program interrupt is generated. The WHERE function converts the virtual page address N_v into a real disk page address N_{vd}. In principle N_{vd} represents the same number of bits as N_v: hence any of the 2^{N_v} pages of any user can be accessed in secondary storage. Since all words of a page are transferred, the real address bits N_r are incremented by either hardware or software from zero to N_r, and the translation must be concerned only with the page address bits N_v as shown.

When main memory is full, it is necessary to remove one page before a new one can be entered. The page replacement algorithm keeps track of page usage in main memory and decides which page is expendable on request for a new page not present. If this expendable page has been modified, it is transferred back to disk, and only afterward can the new page be transferred. These three fundamental requirements are represented in block diagram form in Fig. 9.3-2 with only some lines of communication and control included. In actual implementation, these functions do not necessarily divide into convenient, separate pieces but often are closely interleaved. The address translation function has two major, separate components: to convert N_s into n_p if that page is resident in main memory; or, when a page fault is obtained, to convert N_s into a disk address. Each of these translations can be done in several fundamental ways, as we see later. The integration of these functions into a system can take various forms as detailed in Sections 9.8 through 9.11. In large multiprogrammed systems, additional status and control functions are often included to indicate whether pages have been changed, to control each user's access rights and sharing of pages, and to implement other practical concerns.

In summary, we can state that the basic functions of Fig. 9.3-2 are necessary and sufficient to make a virtual memory work in a logical sense. From a practical point of view, however, data clustering is needed to make the system efficient. If references to primary storage physical locations occurred in a purely random fashion, page swapping between secondary and primary storage would occur very frequently, greatly reducing the overall system efficiency. Fortunately references to memory locations do not occur at random but cluster into groups called pages. The clustering is not precise, hence page size varies with the miss ratio required, size of primary storage, and other factors. This subject is covered in detail in Section 9.4. Thus the functions in Fig. 9.3-2 are those required to make virtual addressing logically feasible; overlaid on these is the phenomenon of data clustering, which makes the virtual memory economically feasible.

9.4 DATA CLUSTERING, PAGING, AND HIT RATIO

As was mentioned briefly in Section 9.1, early commercial virtual memory systems were dominated by the use of *segments* of *variable* size. This approach evolved because in the early storage hierarchies, where program segments were overlaid into main memory, the segments always appeared in widely varying sizes. In fact, the natural length varied with the problem as well as with the programmer. Thus it seemed natural for a virtual system to allow for variation if efficient operation was to be obtained. Later, however, this notion proved to be incorrect. Though the natural segment length does indeed vary, allowing for this in any memory allocation procedure leads to gross inefficiencies. A number of small, fixed size pages can approximate any segment, and since the memory allocation procedure is much simpler, considerable improvement in overall efficiency can be obtained. The fundamental problem with segmentation is that it requires contiguous words in primary storage and the segments may be of varying size.* A request for transfer of a segment into primary memory requires locating an empty region of the proper size. The empty regions may not singly be large enough, even though their sum may be more than sufficient. Since they are not contiguous, however, they cannot be used, and on a statistical basis, many regions of primary storage may remain unused. In a paged system with fixed page sizes, transferring a page only requires finding or creating an empty page slot in primary storage. This is considerably easier than finding or creating

* Contiguous words are required just as for a page in Section 9.3, since the word within a segment is obtained by catenating the lower order real address bits to the higher order segment address bits.

present in primary storage for many different user programs. Each program is a maximum of 32 pages total size, each page containing 1024 words. The results for the particular set of programs indicate that relatively few pages can give a rather small miss ratio. Four pages result in about a 1% miss ratio, whereas 10 pages can reduce this by 2 orders of magnitude. The conclusion is that relatively few pages of a user's program are required to achieve reasonable miss ratios.

9.5 MAPPING FUNCTIONS

The secondary or virtual store contains a large number of pages that must be mapped or compressed into a smaller number of page slots in primary storage. The mapping function, requirement 1 in Section 9.3, specifies how this mapping is to be done. We consider the various mapping techniques in their general, fundamental form and derive relationships that are useful latter. The mapping function has a very significant effect on the address translation.

In practice, mapping functions are considered in four distinct groups (Conti, 1969): direct, sector, set associative, and fully associative. Fundamentally, they form a continuum of set associative organizations, with the direct and fully associative being two extremes and sectoring a special case.

A point of confusion often arises in understanding mapping functions because the mapping functions are only logical concepts that impart overall organization to the page mapping schemes. A given mapping function can be implemented in different ways and when implemented, it becomes part of the address translation function. These logical concepts can be used to visualize the mapping of logical to logical addresses, or logical address to physical address. The precise meaning of this and its consequences are discussed later in this section. First we deal with the general forms of logical mapping functions, assuming that the page slot allocations and relationships to follow are done in terms of *logical* structure *without* necessarily requiring the same *physical* structure.

Following the nomenclature in Section 9.3 and Figs. 9.3-1 through 9.3-3, secondary store contains a maximum of 2^{N_v} pages, each of 2^{N_r} words, whereas primary storage contains only 2^{n_v} pages of the same size. The primary address register has a deficiency of bits equal to n_d given by (9.3-7) as in Fig. 9.3-1. Since $2^{n_v} \ll 2^{N_v}$ as in Fig. 9.5-1, it is necessary to specify how this compression or mapping is to be done. Any page slot in primary storage must hold many different virtual pages, but at different times of course. The actual number of virtual pages that a logical page slot must accommodate varies with the mapping function; the minimum number, however, is the total

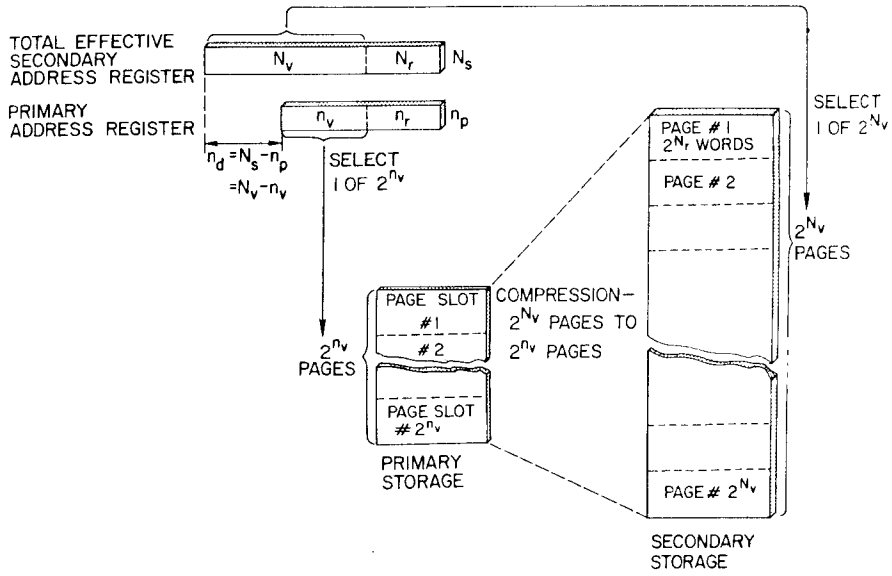

FIGURE 9.5-1 Compression of secondary pages into primary page slots with associated address bits.

page capacity of secondary divided by the total page capacity of primary storage or

$$\frac{\text{minimum number of pages}}{\text{logical page slot}} = \frac{2^{N_v}}{2^{n_v}} = 2^{N_v - n_v} = 2^{n_d} \tag{9.5-1}$$

Thus the minimum number is determined by the deficient bits n_d. The maximum number is the total page capacity of secondary storage, or 2^{N_v}. These two cases represent the extremes of direct and associative mapping, respectively. Between these two limiting cases are any number of possibilities. In a more general form, we can allow a virtual page from secondary storage to reside in any of S page slots in primary storage as in Fig. 9.5-2*a*. These S page slots form a set, and the mapping is the general form of set associative. There are obviously 2^{n_v} total slots; hence the number of possible sets in primary storage is

$$Q = \frac{2^{n_v}}{S} = \frac{2^{n_v}}{2^s} = 2^{n_v - s} = 2^q \text{ sets} \tag{9.5-2}$$

where

$$q = n_v - s \tag{9.5-3}$$

$$S = 2^s \tag{9.5-4}$$

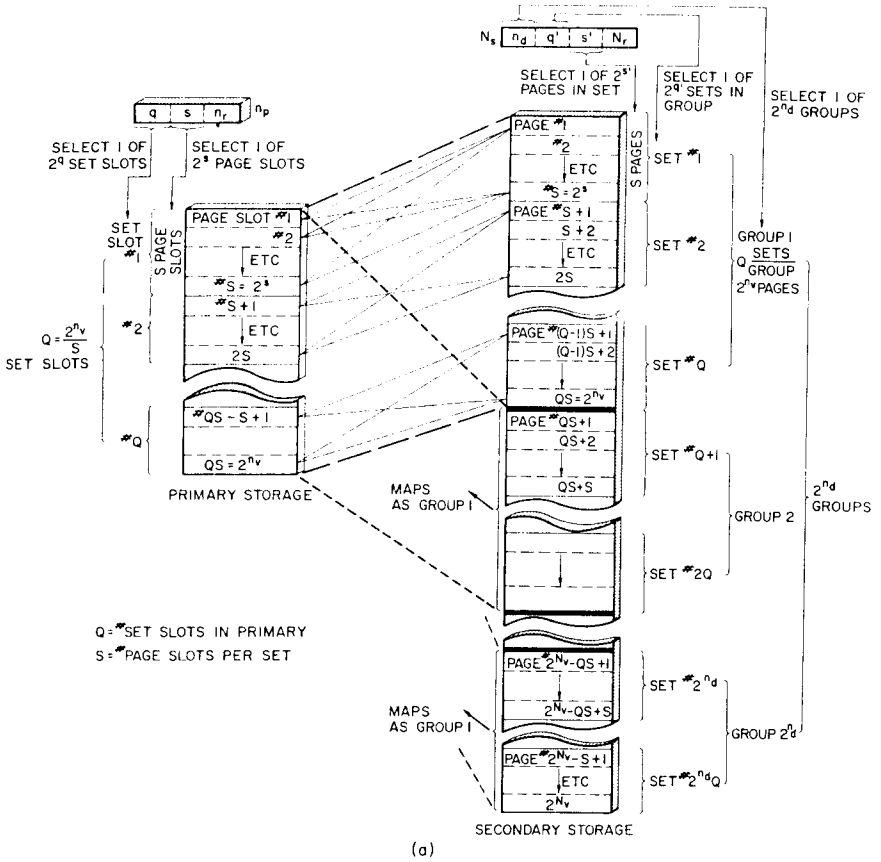

FIGURE 9.5-2 (*a*) General format for mapping secondary to primary storage.

But there are 2^{N_v} virtual pages that must be accommodated in 2^{n_r} page slots. The easiest way to visualize how this might be done is to think of secondary storage as being logically divided into groups of pages, each group of length equal to that of primary storage or 2^{n_r} page slots. Each group contains Q sets and each set contains S pages. Each set slot in primary storage must be shared by the Q^{th} set in secondary storage. Thus set slot 1 is shared by sets 1, $1 + Q, \ldots, 2^{n_d}$. Likewise set slot Q in shared by sets Q, $2Q$, $3Q, \ldots, 2^{n_d}Q$. Pages within a set are associatively mapped into any of the S page slots. All virtual pages are thus assigned to one and only one logical set and can be located in any one of S specific page slots within that set. Sets from different groups can be intermixed within primary storage; therefore not all sets from one given group need be simultaneously resident in primary storage. We

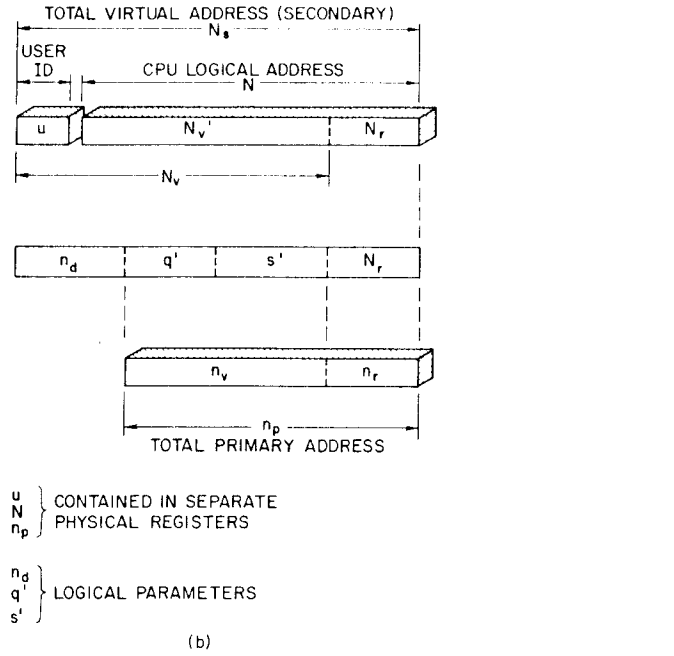

FIGURE 9.5-2 (*b*) Relationship of physical and logical address bit parameters.

may have, for instance, sets 1 from several different groups resident in primary storage, which means that all these virtual pages would have the same q address bits but different n_d address bits.

One reason for such a seemingly complex allocation of pages is that knowledge of which logical set and group a page must occupy constitutes *additional addressing information* that is used by the address translation function. The total number of virtual pages that can possibly reside at different times in any given slot in primary storage is the total number of virtual pages 2^{N_v} divided by the number of sets Q, or

$$\frac{\text{virtual pages}}{\text{logical page slot}} = \frac{2^{N_v}}{Q} = \frac{2^{N_v}}{2^q} = 2^{N_v - q} \tag{9.5-5}$$

But from (9.5-3), $q = n_v - s$; thus substitution gives

$$\frac{\text{possible virtual pages}}{\text{logical page slot}} = 2^{N_v - (n_r - s)} = 2^{n_d + s} \tag{9.5-6}$$

$$= S2^{n_d} \tag{9.5-7}$$

The logical level of organization in secondary storage of Fig. 9.5-2*a* starting from the lowest to the highest is thus 2^{N_r} words per page, $S = 2^s$ pages/set, $Q = 2^q$ sets/group, and 2^{n_d} groups total. The analogous hierarchy in primary storage is 2^{n_r} words/page, $S = 2^s$ page slots/set, and Q set slots/primary storage.

When the physical locations of sets in primary storage are fixed to be equivalent to the logical locations, the word "logical" can be replaced by "physical" in all the above expressions. In the more general case, logical and physical allocations are not equivalent. In the most flexible case, any virtual page can reside anywhere in primary storage, irrespective of the logical structure of the mapping function. In such a case, the total possible virtual pages per physical page slot is the total number of possible virtual pages. Thus for the general case,

$$\frac{\text{possible virtual pages}}{\text{physical page slot}} = 2^{N_v} = 2^{n_d+s+q} \tag{9.5-8}$$

These relationships are important in the address translation process. The smaller the number of slots S per set, or the larger Q, the fewer logical places a page can reside and the easier it will be to find. Likewise larger values of S or smaller Q allow more possible slots per virtual page, and the more difficult the page will be to find. Thus one factor in the selection of a mapping function is the effect on the address translation.

Also important in the choice of a mapping function is the primary slot contention problem. In any mapping scheme, all the virtual pages required for a given problem can, statistically, reside in the same set, and the number of required pages might exceed the size of that set. Thus contention problems between two or more pages for unavailable slots can result in a large miss ratio for that problem, which is undesirable. The probability of contention problems, of course, varies with the number of page slots per set S.

The various distinct types of mapping function can be obtained by varying the value of S. Fully associative mapping results when S is its maximum value of 2^{n_r} and Q is its minimum value of 1. Direct mapping is just the opposite, resulting when S is its minimum value of 1 or $s = 0$ and $Q = 2^{n_r}$. Set associative occurs for any values of S and Q between these extremes. Sector mapping is just a special case of fully associative mapping with the page enlarged to encompass many pages, called sectors. Thus it is apparent that the address parameters q and s are logical entities that can be selected by the designer. They can be taken from the physical address registers, and the value of each is determined by the arrangement of the physical wires connecting the registers. The relationship between the various physical and logical address bits is shown in Fig. 9.5-2*b*. Each of the mapping functions is now considered separately.

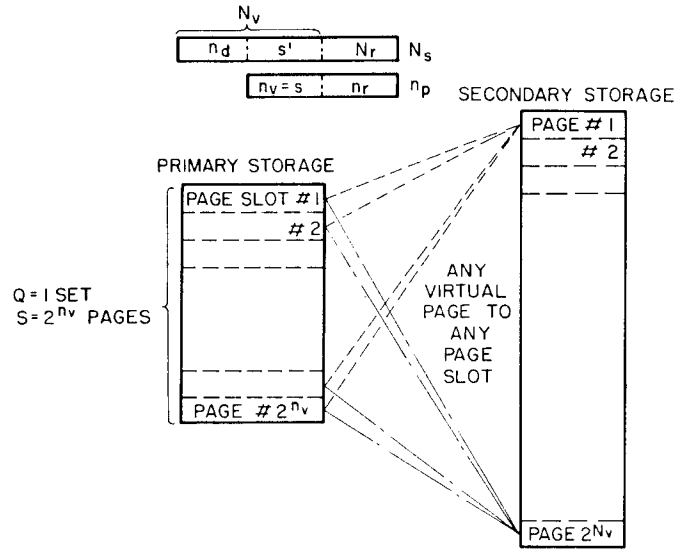

FIGURE 9.5-3 Schematic of fully associative mapping: $S = 2^{n_r}$ page slots set; $s = n_r$, $Q = 1$, $q = 0$.

9.5.1 Fully Associative Mapping: $S = 2^{n_r}$, $Q = 1$, $s = n_r$, $q = 0$

If we allow S to become its maximum value of 2^{n_r}, there can only be one set in primary storage (i.e., $Q = 1$). Under these circumstances, any virtual page can be mapped logically into any page slot, and the mapping becomes fully associative (Fig. 9.5-3). This mapping is the most general and provides the minimum probability for page slot contention problems. Two virtual pages can contend for a page slot only when the pages required simultaneously for a given problem exceeds 2^{n_r}, which is most unlikely. Hence fully associative mapping provides the largest hit ratio for a given problem on a given virtual system and is the most desired mapping function. However an associative type of compare is required in the address translation (Section 9.6), making this the most difficult mapping function to implement.

9.5.2 Direct Mapping: $S = 1$, $Q = 2^{n_r}$, $s = 0$, $q = n_r$

One way to completely avoid associative searching and greatly simplify the address translation is to let $S = 1$, giving only one page slot per set. This is referred to as direct mapping, illustrated in Fig. 9.5-4. Any given page in secondary storage can reside logically only in a specific page slot in primary storage. Using the rules previously described, the first logical page slot in

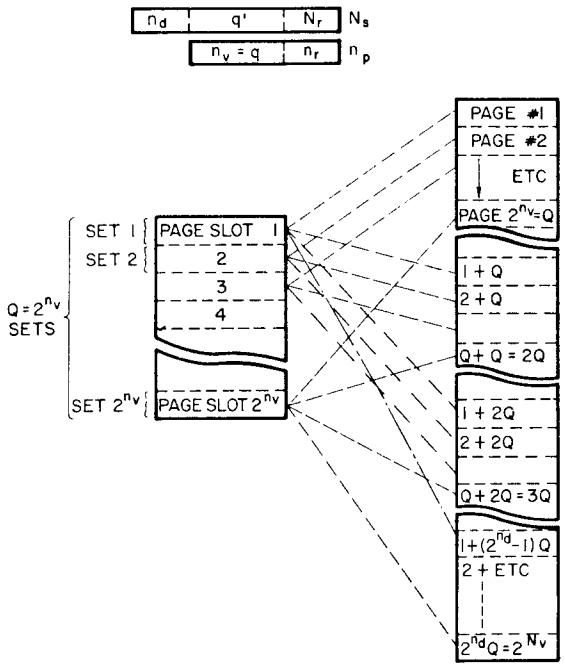

FIGURE 9.5-4 Schematic of direct mapping; $S = 1$ page slot set; $Q = 2^{n_v}$ sets, $s = 0$, $q = n_v$.

primary storage is assigned to hold virtual page 1, or $1 + Q$, $1 + 2Q$, . . . , up to page $1 + (2^{n_d} - 1)2^{n_r}$. Likewise, the second page slot is assigned virtual pages 2, $2 + Q$, $2 + 2Q$, and so on, as shown, where $Q = 2^{n_v}$. When the logical and physical allocations are identical, primary storage will physically contain the pages as shown.

A serious disadvantage of direct mapping is the high probability of primary slot contention. For example, suppose a problem is being processed using arrays or matrices. If each array is in a separate page with array A_{ijk} in page 1 and array M_{ijk} in page $1 + Q$, a problem of the form

$$\sum C_1 A_{ijk} + C_2 M_{ijk} = F_{ijk} \tag{9.5-9}$$

would require a transfer of one of the arrays for every evaluation of F_{ijk}, since both arrays are needed for each point. But only one of these arrays can be present in primary storage at any time, giving a low hit ratio and very inefficient operation. It could be argued that the data could, or should, be organized so that this does not happen. This is possible in principle, but it

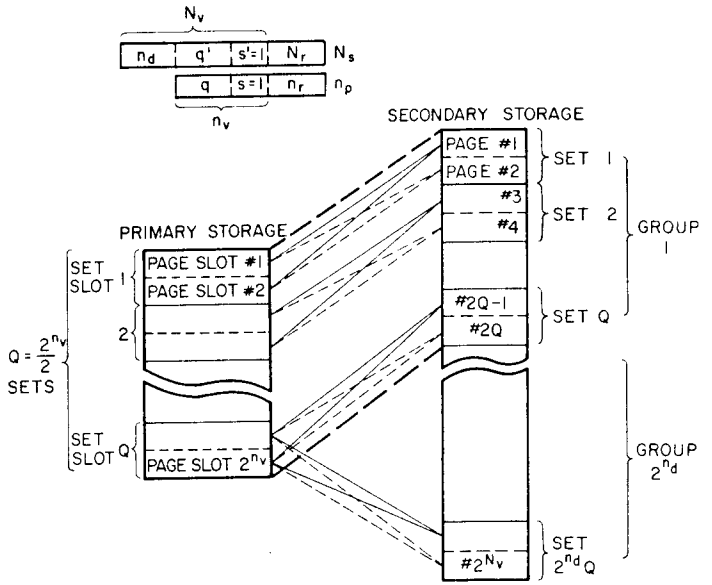

FIGURE 9.5-5 Schematic of set associative mapping showing 2 page slots set; $S = 2$, $s = 1$, $Q = 2^{n_v}/2$, $q = n_v - 1$.

is not always known beforehand how the data will be used. Even when known, such organization presents many formidable problems to the system allocation of storage, requiring "hand tuning" for efficient operation. Therefore direct mapping of the secondary to primary logical address space is never used.

9.5.3 Set Associative Mapping

In fully associative mapping, the slot contention problem is minimized but the address translation problem is maximized. Direct mapping does just the opposite; contention problems are maximized and the address translation problem is minimized. Set associative mapping represents a compromise between these two. The general form is that of Fig. 9.5-2*a*. Some typical values are $S = 2$ or 4 page slots per set. The case for $S = 2$ is presented in Fig. 9.5-5. A virtual page can now reside in either of two logical page slots in primary storage. Hence the contention problem arising with direct mapping in solving (9.5-9) is eliminated. However other contention problems can and do arise. Larger values for S are desirable, when possible. A case for $S = 4$ is described in Section 9.11.2.

9.5.4 Sector Mapping

Sector mapping is a special case of fully associative mapping. In fact, if we ignore the labeling of what constitutes a page, the two are fundamentally identical. In practice there are some differences arising from the way the words are divided and decoded.

Sector mapping can be approached in two ways. The simplest method is to consider that the pages in Fig. 9.5-3 are increased in size to encompass many previous pages. If we wish to maintain our definition of a page (i.e., the number of words that is approximately the natural block size), we can just redefine the groupings. Instead of larger pages, we now define a sector to consist of several pages (Fig. 9.5-6). Both primary and secondary storage are broken into sectors of Z pages, and sectors are associatively mapped into primary storage. Obviously a sector of several natural pages has now taken the place of a single page in the fully associative mapping. All the previous rules pertaining to pages now apply to sectors. Note, however, that for a given physical implementation, the value of S, the number of page slots per set, takes on a slightly modified meaning and is reduced by the factor of $1/Z$. In other words, the number of associative compares is now $2^{n_v}/Z$, since only the presence or absence of the sector is required. In practice, this has considerable merit to the extent that the number of sectors can be quite reasonable (e.g., 16); thus if tags are used for sector identification, the tags can be kept in a small associative memory that is very fast and not excessively expensive.

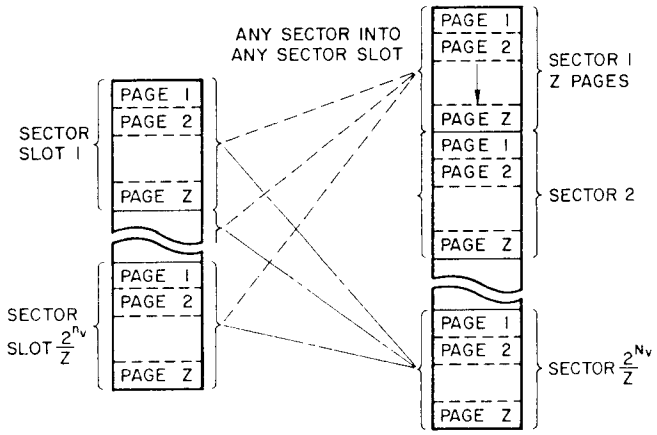

FIGURE 9.5-6 General form of sector mapping with Z pages per sector.

This sector technique using tags has been used with some modifications in the IBM system 360 model 85 cache. The essential idea remains the same – namely, to reduce the size of associative compare hardware while giving the flexibility of an associative mapping. The difficulty with this scheme is that Z must be reasonably large (16 or more), thus the number of sectors in primary storage becomes too small, resulting in a lower average hit ratio.

9.5.5 Logical versus Physical Address Space of Primary Storage

The mapping functions have been discussed in terms of secondary to primary logical address space, as previously indicated. The logical mapping represents the relationship between the symbolic addresses of page slots and can be done independently of any physical address mapping. The descriptions and figures require some conceptual representation of the primary page slots, and these might give the *appearance* of being the actual physical locations. Although this can indeed be the case, it need not be, and in general is not. The confusion between logical and physical mapping comes about because in our minds, we can picture page slots into which virtual pages may be placed, and they appear to have some physical relationship to other page slots. We can quickly address any specified page slot by visual inspection, and we tend to overlook the fact that we have performed a complex address translation function. Yet such a visual search is a sequential associative addressing function. In a computer, the same or other addressing operations can be performed but must be specifically implemented in hardware, just as our visual search is implemented with our eyes and the associative functions of the brain. Thus in our minds, the mapping function appears as though it were mapping secondary store to the physical primary page slots. But when we implement the mapping in the computer, we have removed the associative function provided by the human brain and must supply it in some other manner. This is then the primary logical to physical mapping, which is an essential part of the overall mapping function.

To understand the fundamentally important difference between logical versus physical address and mapping, let us consider the problem of mapping a deck of ordinary playing cards into a set of boxes. For simplicity, we assume that there are only four boxes into which cards can be placed. Also, since an ordinary deck of 52 cards is not a power of 2, we use only the highest 32 cards, from aces to sevens. The cards essentially represent the secondary virtual pages, and the boxes are the primary store page slots. The exact physical address of each box is in the item of importance ultimately, and this has not yet been specified. Let us "logically" label the boxes in some random fashion a, b, c, d by attaching tags (Fig. 9.5-7). The mapping is assumed to be set associative with $s = 1$, which means that a given card can

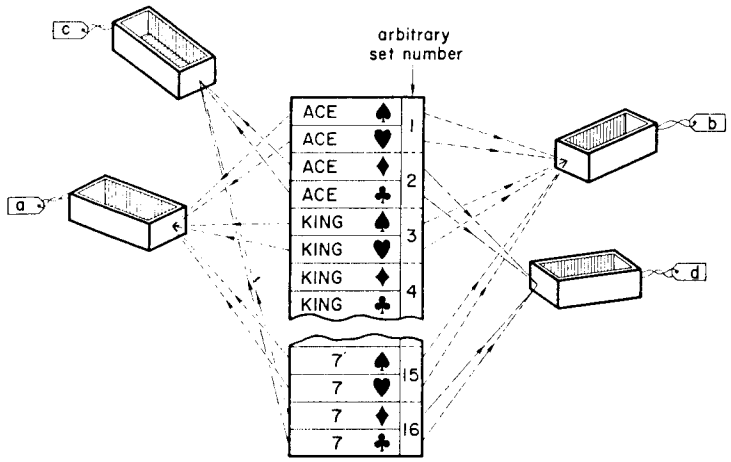

FIGURE 9.5-7 Logical mapping of cards into boxes for set associative mapping with $s = 1$ and four primary storage boxes.

reside in only one of two specifically labeled boxes as shown. For simplicity, we arrange the deck with the highest value card on top (ace of spades) and the lowest card on the bottom (seven of clubs). Since logical mapping is totally independent of physical address, it makes no difference which cards are mapped into which "logical" boxes, as long as we are consistent and obey the mapping rules. Thus we arrange the logical mapping so that the first set, consisting of the aces of spades and hearts, can reside only in the boxes tagged *a* and *b*; the second set, consisting of the aces of diamonds and clubs into boxes *c* and *d*; set 3, consisting of the kings of spades and hearts into boxes tagged *a* and *b*, and so on, similar to that in Fig. 9.5-2*a*. This information about how cards are mapped must be stored *somewhere*. In this very simple case, we may be able to remenber the mapping, so it is stored in our brains. As a rule, however, the mapping is too complex to remember, therefore let us store this information on a sheet of paper (second, third, and fourth columns of Fig. 9.5-8*a*). To completely understand the mapping function and the kind of information that is required, we must attempt to find various cards in boxes, which invokes some address translation operations. However, we do these only in a very general way without specifying the actual implementation. Section 9.6 demonstrates the various ways in which the address translation function can be implemented in terms of the mapping information.

Continuing with the example, we specify that only one card may reside in a box at any one time and start moving the cards in random fashion into

32 Cards

BINARY ID Value $X_5X_4X_3$	Suit X_2X_1	LOGICAL NAME		SET #	LOGICAL WHERE WHERE POSSIBLE	WHERE ACTUAL IF	Tag
111	11	ACE	S	1	a – b	Yes	b
111	10		H			No	
111	01		D	2	c – d	No	
111	00		C			No	
110	11	KING	S	3	a – b		
110	10		H				
110	01		D	4	c – d		
110	00		C				
101	11	QUEEN	S	5	a – b	Yes	a
101	10		H			No	
			D	6		No	
			C				
		etc.					
000	11	7	S	15	a – b	No	
000	10		H			No	
000	01		D	16	c – d	Yes	d
000	00		C			Yes	c

S – Spades
H – Hearts
D – Diamonds
C – Clubs

(a) Logical Map of Cards to Tags
[Map 2]

LOGICAL Tag WHERE	PHYSICAL Shelf #	WHERE Binary Address
a	3	11
b	2	10
c	1	01
d	0	00

(b) Primary Logical to Physical Mapping at given time
[Map 3]

FIGURE 9.5-8 Mapping information for cards in boxes.

and out of the boxes. At any given instant of time, the four boxes contain four cards. We know from the mapping function which cards may be in which boxes, assuming no mistakes in the swapping process. Suppose that we wish to know IF and WHERE the ace of spades is residing in one of the boxes; a quick check on Fig. 9.5-8*a* shows that it is useless to look into boxes *c* and *d*. Hence we need to associatively search only boxes *a* and *b*. If this associative search fails to provide a match, the IF is a "no" and the WHERE is irrelevant. A "yes" match supplies the IF and WHERE directly. Only two associative searches are needed because the boxes are logically tagged and the tags are visible on the boxes. Suppose we remove the tags from the boxes so that the required mapping information is contained only on the piece of paper of Fig. 9.5-8. Since the boxes are identical, it may be difficult determining which two boxes to search for the ace of spades. But *we remember* that box *a* is on the lower left and box *b* on the upper right; thus we might search these boxes. However we have made a very important assumption, namely, that there is a "direct" relationship between the logical tags and physical location of the boxes; that is, the logical to physical mapping of the boxes (primary) is direct. If this relationship is true, our search is valid.

Suppose, however, that we rearranged the boxes in some random fashion as we swapped cards. Now with the logical tags on the piece of paper only, there is no way to find the physical location of the ace of spades except by a completely associative search of all four boxes. If we had provided some mapping of the logical tags to physical location of the boxes, only two associative searches would be needed to locate the ace of spades. For instance, suppose we keep the boxes on four shelves and specify that boxes *a* and *b* are always on the top two shelves and *c* and *d* are always on the bottom two shelves. Now there is no need to search the bottom two shelves, which makes the search simpler. This additional information about where the boxes physically reside must be stored somewhere (e.g., in the primary logical to physical mapping of Fig. 9.5-8*b*). There are various techniques for implementing this mapping, but then it becomes part of the address translation. The essential point is that in the most general sense, the secondary to primary mapping function is a logical map. When the physical position of the boxes is specified by the use of tags stored directly on the boxes, the logical map is directly converted into a physical map. When the tags are stored separately from the associated boxes, an indirect conversion to the physical location of each box is required. Both cases of directly or separately stored tags involve the use of an additional mapping of the logical tag to physical location of the box, the former represents direct mapping and the latter a form of associative mapping. In either case, this logical to physical map is a fundamental requirement.

The case of finding IF and WHERE the ace of spades might be in the boxes requires a minimum of two associative searches plus the second, third, and fourth columns of Fig. 9.5-8*a* and the map of Fig. 9.5-8*b*. It is possible to avoid associative searches completely by storing more information within the mapping tables. Suppose we provide a fifth column in Fig. 9.5-8*a* called "WHERE ACTUAL," specifying IF and WHERE any card actually resides logically within a box. Thus the ace of spades and three other cards have a "yes" for IF and a logical box location WHERE each is. The boxes logical to physical translation must make use of Fig. 9.5-8*b* as before. Thus the location of the ace of spades requires direct addressing only: a reference to its binary address, shown as 111-11 in Fig. 9.5-8*a*, yields "yes," "tag *b*" for IF and WHERE from the last column; an access to tag *b* in Fig. 9.5-8*b* yields shelf 2 (note that any of the four shelves just as easily could have been indicated here). The price we pay for the elimination of associative searches is the need for more stored mapping information.

In a completely analogous manner, we can show that the initial orientation of the entire deck of cards as in Fig. 9.5-7 is a logical orientation, and the initial selection of a physical card from the deck (secondary) before being placed in a box requires a mapping of the secondary logical to secondary

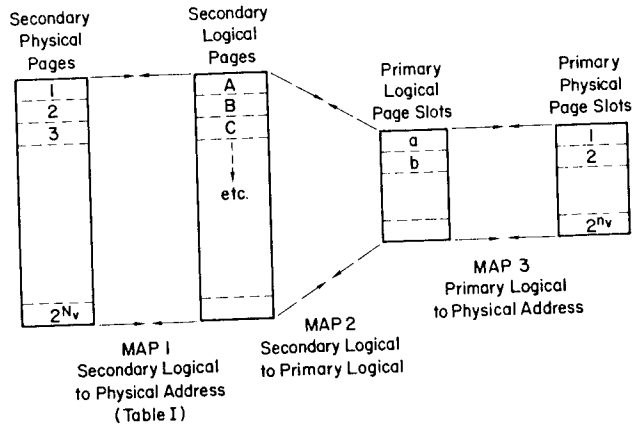

FIGURE 9.5-9 Schematic of logical and physical address spaces showing three fundamental maps required for a two-level virtual memory hierarchy.

physical location. In other words, we picture the logical orientation of the cards as in Fig. 9.5-7, but given a shuffled deck, we would need a logical to physical mapping to find the ace of spades, and so on. Thus we can conclude that any two-level virtual hierarchy requires three mapping functions. One map is required for each level to map that level's logical to physical space, and one map is required between levels. Figure 9.5-9 shows these maps schematically for a disk–main memory type of two-level hierarchy. Map 1 is generally provided by Table I discussed in Section 9.8. In a cache–main memory type of two-level hierarchy without a disk–main hierarchy, there are no pages in the main memory that are visible to the user. The CPU (secondary) logical address is converted into a physical main memory address by the compiler, assembler, link editor, and so on; thus Map 1 in Fig. 9.5-9 is essentially a direct mapping in such a case. The internal memory physical pages are mapped into the logical cache pages with an analogous Map 2, and likewise the cache logical to physical mapping with an analogous Map 3.

In a three-level hierarchy, containing a disk–main virtual store and a main–cache hierarchy, the cache is paged out of main memory just as in the two-level hierarchy. However additional mappings are obviously required. One logical to physical map for each level and one map between each level results in a minimum of five maps for this case. The disk and main memory mappings would be as indicated in Fig. 9.5-9, with disk being the secondary, the main memory the primary, and the total CPU logical address the equivalent to the "secondary logical pages." When a cache is inserted into

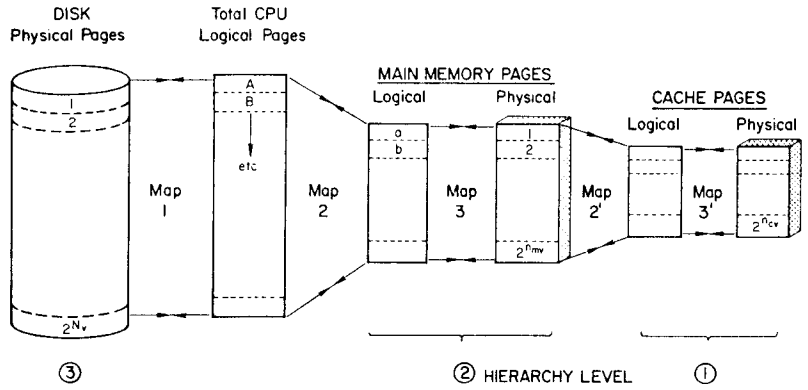

FIGURE 9.5-10 Schematic of logical and physical address space showing various maps required for a typical three-level virtual memory hierarchy.

this system, its position in the mapping scheme must be determined. Fundamentally, the logical cache pages can be mapped into the cache starting from the CPU logical pages, from the main memory logical pages, or from main real (physical) pages. However since the CPU logical pages are so much more numerous, and since they must be mapped into the main memory logical space anyway, it is advantageous to start either with the main logical or physical pages. For various practical reasons, the latter are often used (see Section 9.12). The schematic of the various maps for such a case appears in Fig. 9.5-10. In all systems, Maps 2, 2', 3, and 3' are implemented as an integral part of the address translation function discussed in Section 9.6. All three maps can take on any of the possible forms discussed previously from direct to fully associative. The degree of associativity of one does not affect the associativity of the others, since each is completely independent.

Some of the important conclusions about mapping of physical secondary pages to physical primary pages in a two-level hierarchy can be summarized as follows.

1. Three mapping functions are a fundamental requirement.
2. Contention problems for primary page slots are determined only by the secondary logical to primary logical mapping function and become less severe as the degree of associativity 2^s increases.
3. The mapping of primary logical to physical address has no bearing on the page slot contention problem; more generally, all three maps are independent of each other in principle.

9.6 ADDRESS TRANSLATION FOR A TWO-LEVEL VIRTUAL HIERARCHY

To understand virtual memory addressing, familiarity with the concepts and mapping functions discussed in Section 9.5 is imperative.

The functions performed by the address translation were outlined in Section 9.3. The N_s bits must be converted into an IF and WHERE real address. As shown in Fig. 9.3-3, N_r is directly translated, hence only the virtual page address N_v need be considered. When the desired page happens to be resident in primary storage, the IF translation produces a "yes" and the correct, real n_v must be provided by the WHERE translation. This is essentially an implementation of Maps 2 and 3 of Fig. 9.5-9. When the desired page happens not to be resident in primary storage, the IF produces a "no" and the real address of that page in secondary storage must be found. This is an implementation of Map 1 of Fig. 9.5-9. In a two-level hierarchy, the conversion of N_v into n_v when the desired page happens to be resident (i.e., IF = "yes") becomes the critical factor in the overall speed of address translation. If the entire process is to run at an effectively fast cycle time, this part of the translation should not introduce excessively large delays. We concentrate mainly on this, the most important part of the translation function, and its associated Maps 2 and 3 of Fig. 9.5-9. We consider the various fundamental schemes in considerable detail and derive important tradeoffs and relationships.

9.6.1 Translation of Logical Address to Primary Storage Address

Irrespective of the mapping function used, the address translator must decode all N_v bits of the secondary storage logical address. Only a WHERE function is performed in ordinary nonvirtual decoding, since in such cases each logical address has the same length as the primary memory address register and uniquely specifies a memory word. In virtual address decoding, the WHERE function is more complex and the additional IF function must be performed. More generally, ordinary nonvirtual decoding could be thought of as virtual decoding with the IF functions always set in the "yes" state. However the given nonvirtual address will always be unique and real, which allows for a simple implementation, since a one-to-one correspondence exists between the address and the physical location. In virtual addressing, this correspondence is lost. As indicated in the general mapping form of Fig. 9.5-2*a*, a given page slot in primary storage can contain different pages at different times. Although the virtual page address N_v uniquely specifies the virtual page in secondary storage, N_v by itself is not sufficient to specify

IF and WHERE that page resides in primary storage. Thus *additional information* must be stored in some manner to tell how the mapping is performed, and this constitutes the mapping function. The *manner* in which this information is stored and *used* constitutes the address translation function. A mapping function is only a conceptual, logical view of how page slots are to be shared, and when implemented, it becomes part of the address translation function. Clearly these two must work closely together. As previously indicated, we must implement both Maps 2 and 3 of Fig. 9.5-9 — the secondary to primary logical address map and the primary logical to physical address map. The address translation must provide the IF and WHERE for both maps. However the logical IF is easily converted into a physical IF (e.g., by the hard-wired "Yes/No" control line on the primary memory address register as in Fig. 9.3-2). There are numerous ways of implementing this conversion, but it is relatively straightforward, and we concentrate here on the fundamentals of the address translation function to provide the logical IF, logical WHERE, and physical WHERE. The exact form of the address translation function varies in a continuous manner from a pure table requiring look-up for each request, to a simple associative directory, assuming a general form of mapping as in Fig. 9.5-2*a*. With such a mapping function, some form of association must be established between the real and the virtual page, irrespective of the method of address translation. The table form of translation provides this association by storing the real location of every virtual page, hence requires considerable storage. The associative directory provides this association very directly but requires a great deal of associative logic. Between these two methods lies a continuum of translation techniques requiring varying amounts of storage and associative logic, as might be expected. In fact this represents the fundamental tradeoffs in the choice of an address translation function: the table look-up is relatively slow because the table is stored in primary memory and requires additional memory access. The directory is small, thus can be fast, but it is expensive because of the necessary associative hardware.

The example of placing cards in boxes in Section 9.5 did not specify the details of implementing the mapping function because it is carried out by the address translation function. To see the range of possible schemes, let us continue with the "cards in boxes" example and consider two possible approaches, representing the end points of a continuum. After the example, we investigate address translation in its general form and derive important relationships.

Section 9.5 partially described these two cases of a table and an associative directory, but without the details and some remaining important considerations. We already know that certain mapping information must be stored somewhere, the exact amount varying with the address translation scheme

chosen. Let us store this information in a general random access memory called the "tag store" and see how much and what information it must actually contain. We arbitrarily define a table scheme as one for which no associative compares are needed and a directory as one for which associative compares are needed. If we are to avoid associative compares and use direct decoding instead, some form of Fig. 9.5-8*a* and *b* must be stored. We initially assume set associative mapping for both Maps 2 and 3. The tag store will have to contain at least one tag in the form of a table entry or word for each card. If we use a 5 bit address $X_5, \ldots, X_1$ for each of the 32 cards, the logical name of each is replaced by an address of the tag store: hence the column labeled "Logical name" in Fig. 9.5-8*a* is unnecessary. To determine IF and WHERE a given card resides, the last column must be included. In addition, the information of Map 3, Fig. 9.5-8*b*, must somehow be incorporated. Note that the last column of Fig. 9.5-8*a* contains the logical tag for each of the four boxes and is redundant with the first column of Fig. 9.5-8*b*. Hence the physical WHERE of Map 3 can be included directly within the last column of Fig. 9.5-8*a* in place of the logical WHERE tag. This greatly simplifies both mapping information and address translation. Furthermore, since each of the four cards resident in a box now has the physical box location (i.e., shelf address) stored with its entry in the tag store, this address can be any of the four shelves within invoking any difficulties. Hence the box logical to physical mapping can be fully associative as easily as set associative, so the former is chosen. This does not necessarily have an effect on the secondary to primary logical Map 2. If the latter is set associative, we do *not* need to know where it possibly can reside for address translation because its address is already given by the physical WHERE. To swap a new card into a box, however, we would have to specify which logical boxes are permitted. But this is totally unnecessary: if the address translation does not need to know the set, we may as well make this mapping fully associative and eliminate the complexity of mapping and replacement. Thus columns 3 and 4 of Fig. 9.5-8*a* are unnecessary, leaving only the first column contained in the address decoding network of the tag store and the last column as the actual stored information. The IF function can be done with a one-bit logical operation called a flag, and the WHERE function must be able to specify at least one of 2^{n_p} pages in primary storage, which requires n_p bits, or 2 in this case. The general form of this table (Fig. 9.6-1) represents that actually used, with some modification, in virtual memory systems as detailed in Section 9.9. The logical flag "IF" is shown converted to a physical IF by a direct enable input to the primary address decoder.

The associative directory can be approached in many different ways. If we study the table of Fig. 9.5-8*a*, we note that most of the entries contain "no" for the IF function. In fact, only four cards can ever be resident in a box:

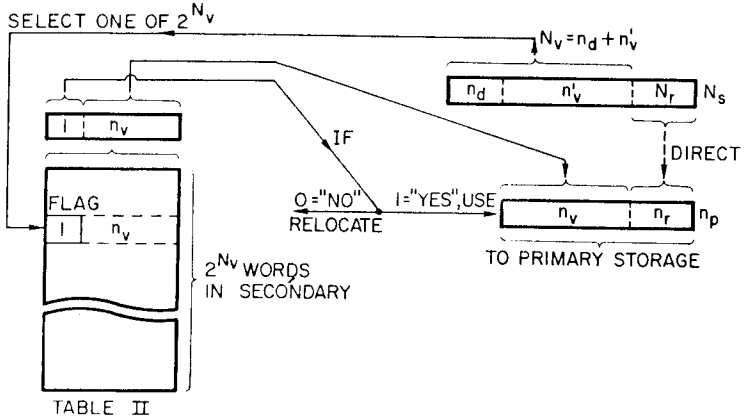

FIGURE 9.6-1 Table address translation showing fully associative mapping for Maps 2 and 3.

thus it is necessary to store information about these four boxes only, which greatly reduces the size of the tag store. In other words, instead of storing information about each card, most of which is "no," we store information only about each box. The difficulty is that many different cards can reside in any given box, and although the required storage is reduced, the process of address translation becomes more complex.

We assume a fully associative mapping, giving $s = n_v = 2$. The tag store must now contain only one entry or word per box to implement Map 2. Since we do not know a priori the identity of the card in any box, this must be stored in the tag store word, which requires 5 bits of $X_5, \ldots, X_1$. This is equivalent to N_v or $n_d + n_v$ bits, as is evident from Fig. 9.3-3 and 9.5-2*b*. To implement Map 3, we can do as previously with the table and store the real physical address of the box as part of the tag store word. The tag store then becomes a fully associatively addressed directory (Fig. 9.6-2). The 5 bit ID of each of the cards resident in a box would be stored in the N_v portion of each word. An associative match on each of these produces one "match" condition that is equivalent to IF = "yes," and the physical WHERE address of that box is the n_v portion of that match word. For instance, if we want to test for the ace of spades whose ID is 11111, each word of the tag store is matched against this ID. The second word in Fig. 9.6-2 will give a match, and the n_v portion indicates shelf 2 (binary 10), as originally indicated in Fig. 9.5-8. Thus Maps 2 and 3 are both contained in the directory, and both are fully associative in this example. The tag store is much smaller than previously with the table scheme, but now it must perform associative selection capabilities, which is more difficult.

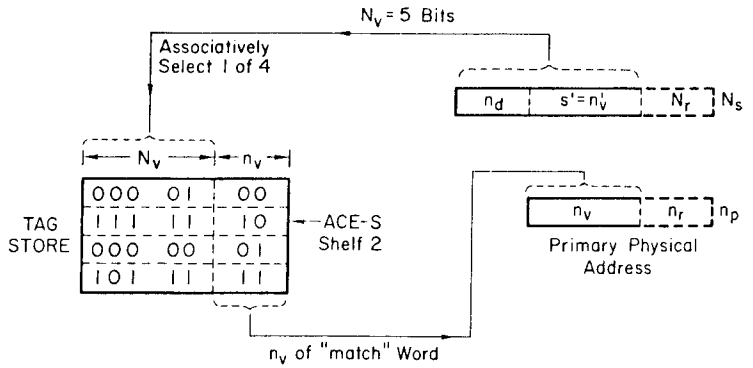

FIGURE 9.6-2 Fully associative tag store (directory) for address translation.

We now consider the address translation in its general form to see the "continuum" of schemes and subtle tradeoffs. Some profound conclusions that become evident are summarized later. For the general case, we assume the mapping to be the general set associative form of Fig. 9.5-2*a*. In our two specific examples of "cards in boxes," we saw that the amount and character of the additional stored information could vary considerably, depending on the implementation of the mapping function. Therefore let us assume that the required information is stored as 2^e logical entries or words of as yet unspecified length, in some undefined tag store. These entries contain all tags and other information needed to implement Maps 2 and 3, as well as the address translation function. Let us first determine the bounds on the number of such logical entries, the amount of information that must be stored in each, and the various ways in which this information can be used to determine IF (logical) and WHERE (logical and physical) the real page exists in primary storage. We let e be some fraction of the total secondary virtual page bits as in Fig. 9.6-3, where the maximum value is $e \leq N_v$. The minimum value for e can be deduced easily. Since there are 2^{n_v} page slots in primary storage, it is necessary to store some information concerning at least each of these slots with one logical entry per page slot. Thus e must be at least as large as n_v, and the bounds on e are

$$N_v \geq e \geq n_v \tag{9.6-1}$$

Obviously the two previous cases of "cards in boxes" were the ends points $e = N_v$ for the table and $e = n_v$ for the directory. Let us now consider the various possible cases for some general value of e between the bounds of (9.6-1). First note that the associativity of the secondary logical to primary logical mapping functions, specified by s, can vary independent of e, that is,

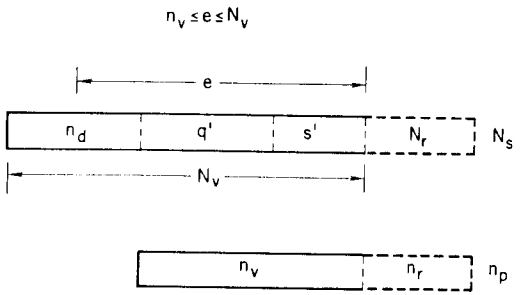

FIGURE 9.6-3 Relationship of tag store addressing bits e to the secondary and primary page-addressing bits N_v and n_r, for the general case.

any degree of associativity can be translated by any value of e satisfying (9.6-1). This is fundamentally true for the logical mapping, but the method of implementation forces certain relationships between e and s in terms of the number of bits that must be associatively compared. For the general case with $e < N_v$, the number of stored logical entries is *less* than one per virtual page by the amount $2^{N_v - e}$. To find IF and WHERE a given N_v address is resident in primary storage, this "unstored" information of length $N_v - e$ bits must be stored in each logical entry. In addition, for a given mapping of associativity 2^s, a required page might possibly reside in any of these 2^s slots of a set. Hence s bits must also be stored in each logical entry of the tag store to assist in the logical IF and WHERE determination. Completing the search therefore requires

$$\text{number of associative compares} = 2^s \tag{9.6-2}$$

$$\text{number of bits/compare} = N_v - e + s \tag{9.6-3}$$

These fundamental requirements are valid irrespective of the primary logical to physical mapping or the remaining implementation. These associative compares provide only the logical IF and WHERE, the former signaled by a "yes/no" match condition. This can be easily converted to a physical IF by using a simple logic gate to activate or deactivate the primary address register as in Fig. 9.3-2. Other techniques are possible, but we assume the method just described in subsequent cases.

The primary logical to physical mapping (Map 3) can be done as previously, by storing the real address of length n_r bits within each of the 2^e entries. As before, this makes Map 3 fully associative without affecting the overall address translation, irrespective of the value of s used in Map 2. We do not have to make Map 3 fully associative: if a set associative map of associativity $S_2 = 2^{s_2}$ is used, only $n_r - s_2$ bits need be stored in this portion

of the tag store. Since this seldom saves many bits and greatly complicates the system, a fully associative Map 3 storing n_r bits is typically used. A direct map could also be used, which would require no stored bits but only a direct *enable* or similar function on a page of a semiconductor chip (see Section 9.13). A direct map 3 becomes very important in one special case of a fully associative Map 2 as discussed later.

Now we have a tag store consisting of 2^e entries or words, each storing $N_v - e + s$ bits, which are associatively compared to provide the logical IF and WHERE for Map 2, and each contains also from 0 to n_r bits for Map 3 depending on the associativity of the latter. The general scheme is given in Fig. 9.6-4*a*, assuming that Map 3 is fully associative and Map 2 is of associativity 2^s. Since at least 2^s entries must be associatively addressed, 2^{e-s} can be directly addressed. The address translation procedure would be as follows. For a given virtual page address N_v, the higher order bits $e - s = n_d - (N_v - e) + q'$ are used to directly address one of the possible sets in the tag store; only 2^q of these can have valid entries, since there are only 2^q sets in primary storage. Comparison of the $N_v - e + s$ bits stored within each of the 2^s entries of the directly selected set with the same bit positions from N_v produces a "yes" or "no" match. If "yes," the n_r from the matched

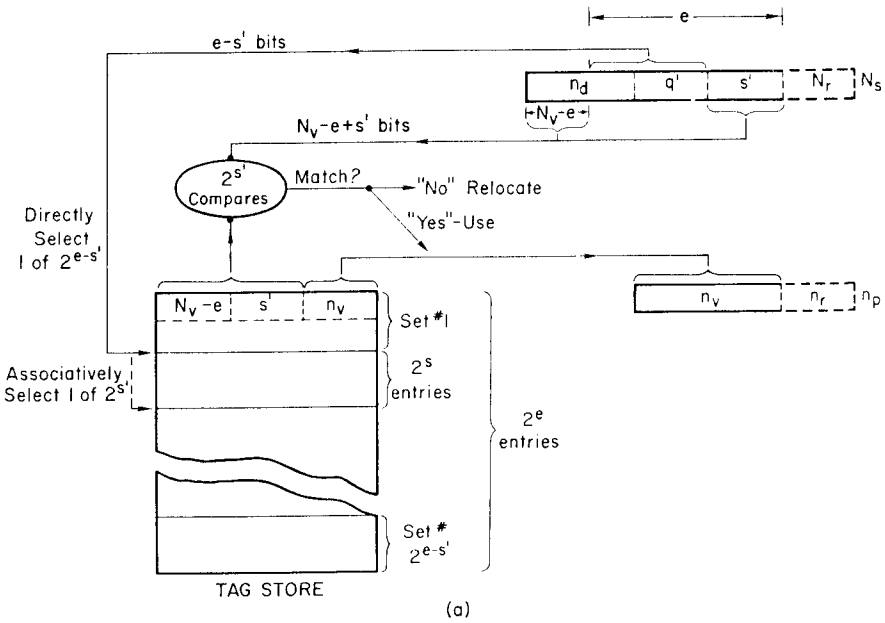

FIGURE 9.6-4 (*a*) General case of a tag store directory with 2^e logical entries.

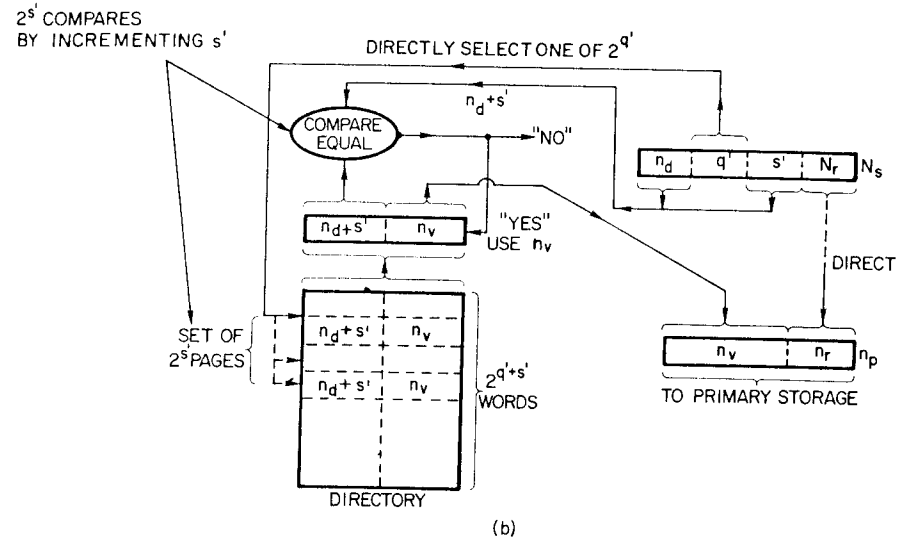

FIGURE 9.6-4 (*b*) Example of minimum tag store directory implementing a set associative Map 2 and fully associative Map 3 showing $2^{s'}$ sequential compares.

entry provides the real WHERE address for the primary store address register as shown. In this case, the tag store must be accessed first to find the real n_v and subsequently the primary store is accessed using $n_v + n_r$ to obtained the desired byte or word. A specific example for $e = n_v$, set associative Map 2 and fully associative Map 3 appears in Fig. 9.6-4*b*.

A fundamental relationship of considerable importance is the total number of *possible* pages in secondary storage that share the *same* logical set in the tag store. This is simply the total number of secondary pages divided by the number of logical sets in the tag store, or

$$\text{secondary pages sharing each tag store set} = \frac{2^{N_v}}{2^{e-s}} = 2^{N_v - e + s} \qquad (9.6\text{-}4)$$

This relation is used later to obtain a number of important conclusions about the tag store.

9.6.2 Special Cases of the Tag Store

We should be able to obtain previous examples of the table and fully associative directory from this general scheme by letting e take on the values N_v and n_v, respectively. Let us try these to see some important consequences.

CASE 1

When $e = N_v$ and $s > 0$, the tag store contains 2^{N_v} entries. Following the general operation already given, a search requires a direct decoding to one out of 2^{e-s} or $2^{N_v - s}$ entries, and 2^s associative compares on $N_v - e + s = s$ bits. The result of a correct match must produce n_v bits for the real page address, assuming Map 3 to be fully associative.

Equation 9.6-4 indicates that the total number of possible secondary pages sharing the same logical set in the tag store is 2^s pages. But since this is also the maximum number of logical page slots in a set, any number of pages of any mix can be accommodated simultaneously in the tag store. Hence the logical mapping is fully associative for any value of s in this special case. Let us choose $s = 0$, to permit the number of associative compares to reduce to $2^0 = 1$, and the number of bits to be compared to reduce to zero. This results in a direct decoding to one of 2^{N_v} or the entire table, storage of n_v real address bits, and one associative compare over zero bits. The latter represents the degenerate case of merely specifying whether that page is resident. It could be done by performing a logical check on whether an n_v address is present. It is simpler to provide a one-bit logical flag; hence we have reduced the tag store to the previous table scheme.

CASE 2

When $e = n_v$ and $s > 0$, the tag store contains 2^{n_v} entries. A search requires a direct selection of one of $2^{e-s} = 2^{n_v - s}$, 2^s associative compares on $N_v - n_v + s = n_d + s$ bits, and a match must produce the real n_v bits. This is a specific case of a set associative directory such as that in Fig. 9.6-4*b*. If we let $s = n_v$, the direct selection of one of $2^{n_v - s}$ reduces to zero and 2^{n_v} associative compares are required on $n_d + s = n_d + n_v = N_v$ bits. This is the fully associative directory previously discussed.

A fundamental conclusion of profound consequences is that the associativity of the secondary to primary logical mapping, Map 2, provided by a general tag store, hence the required number of associative compares, is fixed at 2^s for all values of e except for the special case of the table where $e = N_v$. The fundamental reason for this follows very directly from (9.6-4) and results from contention problems for *logical* page slots. Even when the real primary store is not full, contention problems can arise because the pages required *simultaneously might happen* to be all from the *same logical set*, and the *number required* might be greater than 2^s. For instance, for any tag store with $N_v - e \geq 1$ or $e \leq N_v - 1$, the number of secondary pages sharing a logical set, given by (9.6-4), is obviously greater than 2^s. But by chance, this larger number could be needed simultaneously in that logical set, whereas only 2^s can be accommodated within the tag store (i.e., the

degree of associativity is 2^s). Hence a new contention problem may arise within the tag store even though the primary store has empty page slots. Set associative mapping, therefore, introduces a second kind of contention problem, one for tag store space, in addition to the ordinary contention arising because the primary store is smaller than secondary storage. We did not have this contention problem in the special case of $e = N_v$ because as previously discussed, the maximum number of secondary pages sharing a logical set is 2^s, which exactly equals that which can be accommodated. Hence except for the special case, the associativity of the general tag store with 2^e entries is determined only by s.

The above conclusion raises the question, What properties of the address translation are affected by changing e? As the number of entries e varies, only the number of bits that must be associatively decoded varies as

$$N_v - e + s \tag{9.6-5}$$

As e increases, the number of bits associatively compared decreases, and vice versa, Thus we merely trade the number of tag store entries for word size and associative bits compared. The minimum number of bits required in any tag store scheme is just the number of logical entries times the number of bits per entry. These various expressions and other important relations are summarized in Fig. 9.6-5 for the general case, the table, and the minimum sized directory. The table can require considerably more storage than the minimum directory. The ratio of total storage capacity for these two (from Fig. 9.6-5, assuming Map 3 to be fully associative) is

$$\text{bit capacity} \frac{\text{table}}{\text{directory}} = \frac{(1 + n_v)2^{N_v}}{(n_d + s + n_v)2^{n_v}} = \frac{1 + n_v}{N_v + s} 2^{n_d} \tag{9.6-6}$$

Some typical values might be

$$N_v = 18, \qquad n_v = 6, \qquad s = 2\ (q = 4), \qquad n_d = 12$$

Substitution of these into (9.6-6) gives the ratio as $0.35 \times 2^{12} = 1434$, a substantial difference showing the value of the minimum directory. However we have said little about the hardware characteristics of the tag store for each case. Typically for the table implementation, the tag store uses primary (main) memory, since only random access addressing is required and the table can become quite large. A fundamental requirement of this table is that the entries be logically contiguous, since successive binary values of N_v refer to successive logical entries in the table. Thus even if many entries are blank, they *cannot* be removed from the table to make the table shorter.* The table

* Except for the rare case that all entires above a certain binary address are blank or the user requests a smaller total number of pages, which automatically makes the table smaller.

TAG STORE Description	NUMBER LOGICAL ENTRIES	NUMBER LOGICAL ENTRIES DECODED Directly	NUMBER LOGICAL ENTRIES DECODED Assoc.	NUMBER BITS COMPARED ASSOCIATIVELY	MINIMUM NUMBER BITS PER ENTRY Map 2	MINIMUM NUMBER BITS PER ENTRY Map 3	MINIMUM STORAGE CAPACITY ≠
A. General TAG STORE	2^e *	2^{e-s}	2^s	$N_v - e + s$	$N_v - e + s$	0 to n_v	$(N_v - e + s + n_v)2^e$
B. Minimum TAG STORE ($e = n_v$)	2^{n_v} *	$2^{n_v - s} = 2^q$	2^s	$n_d + s$	$n_d + s$	0 to n_v	$(N_v + s)2^{n_v}$
C. TABLE ($e = N_v$)	2^{N_v}	2^{N_v}	0	0	1	0 to n_v	$(1 + n_v)2^{N_v}$

* Can be 2^{e-s} physical words – see section 9.12
≠ Assuming Map 3 = n_v & excluding control bits

FIGURE 9.6-5 Comparison of tag store characteristics for different implementations.

must contain one entry for each possible logical (virtual) page. A method for reducing the amount of this table that must be stored in primary memory is described in Section 9.8, and examples are given in Section 9.9. For the minimum directory implementation, the tag store is usually a separate random access memory because of the required associative compares. Further discussions concerning the various design considerations are given in Sections 9.8 and 9.10.

It is clear that the choice of mapping function has a significant effect on the method of address translation. This effect can be linked to a telephone directory in which names are listed in various ways. If we assume that no two names are identical, a purely alphabetical listing is like direct mapping: there is one and only one possible logical position for each unique name, and it is easily found. If the listing is purely random or has no structure, this is comparable to fully associative mapping and requires the capability for an associative search on every name, to be able to find the final phone number. Set associative mapping would be analogous to listing together all names say, starting with A, then all those starting with B, etc., but no order within the A's or B's, etc. Hence the number of associative searches equals the number of names under the letter of interest, which is identical to the number S of possible logical names per set.

In all the above examples we have emphasized the use of fully associative mapping for the primary logical to physical Map 3, even though in principle, direct or set associative mapping can be used. The reason is practical in that a direct or set associative Map 3 would involve additional complexity in the actual page swapping and would greatly limit the ability to incorporate operating systems and memory expansion. Thus a fully associative Map 3 is

most desirable. A very special and important case arises in which this Map 3 has the appearance of being fully associative even though it is a direct mapping, in fact can be a hard-wired direct mapping. This situation occurs when the secondary to primary logical mapping (Map 2) is fully associative: that is, $s = n_v$. In such a case, *any* virtual page can reside in *any* logical page slot. If a direct Map 3 is used, logical and physical page slots become identical. Now, any virtual page can reside in any physical page slot as desired, and Map 3 has no effect on this capability. This principle is discussed more fully in Section 9.13 with example.

Some important conclusions about address translation are as follows.

1. The address translation function cannot be implemented without including the mapping function for both Maps 2 and 3.
2. Translation schemes form a continuum of tag stores of varying number of entries e, with the table and minimum directory at the two extremes for e equal to N_v and n_v, respectively.
3. Any secondary to primary logical mapping with associativity of degree 2^s must perform 2^s associative compares on $N_v - e + s$ bits: a special case occurs for $e = N_v$, which yields a table of any degree of associativity.
4. Increasing the number of entries e in the tag store only reduces the number of bits that must be associatively compared; it does not change the required number of such compares.
5. Implementation of the logical to physical IF function is very simple.
6. Implementation of the primary logical to physical WHERE map is much easier than the secondary logical to primary logical WHERE map, and the former can typically be fully associative as easily as any other mapping.
7. When the secondary logical to primary logical mapping function is fully associative, the primary logical to physical map can be direct while giving the appearance of being fully associative in the sense that complete freedom is provided for the physical location of any page in the primary store. This is a special case.

Fundamentally, the address translation function is required only to perform the IF and WHERE translation of virtual to real pages. In practical systems it is also desirable to provide a certain amount of "status and control" information. For instance, if no changes (i.e., no writing) have been made within a given page, it can be erased when it is to be swapped out of primary memory. In addition, storage protection in the form of access rights can be implemented within the address translation function if desired. Additional information stored within the previous necessary logical entries of the tag store can control access and sharing of pages among various users. Examples

are given in Section 9.9. When the address translation is accomplished with a directory, the need for speed dictates that the information for the replacement algorithm be contained within the logical entries or an equivalent hardware scheme (see Section 9.7). Thus the various schemes for storing the additional information for address translation sometimes contain status and control information in addition to the IF and WHERE information.

9.7 REPLACEMENT ALGORITHMS AND IMPLEMENTATION

Replacement algorithms can be placed arbitrarily into two broad categories: (*a*) algorithms that *do not* use historical information to determine which page to remove from primary, and (*b*) algorithms that *do* use historical information for replacement.

The first category does not require storing any information about page referencing, hence essentially includes random or near-random replacement algorithms. The second category requires storage of some kind of information about page referencing, depending on the exact nature of the algorithm. This represents the bulk of algorithms—first in, first out (FIFO), least recently used (LRU), least frequency used (LFU), and so on. The implementation of the first type—say, a near-random replacement—is simple in principle, requiring only a pseudo-random number generator. The second type requires additional storage hardware and logic processing functions, both of which can vary considerably depending on the exact nature of the algorithm and sometimes are quite complex.

Intuitively it would seem that if data clustering (Section 9.4) occurs, page replacement should be based on some history of page usage. This generally seems to be the case, but there are many exceptions wherein a random replacement can give a better hit ratio. Unfortunately, the best algorithm cannot be derived from first principles but must be obtained from simulation of actual job streams as was done in Section 9.4. Various studies (Belady, 1966) have resulted in the conclusion that no one best algorithm exists; rather, certain algorithms are best for particular classes of problems and worse for other classes. As a result, disk–main memory virtual systems use a "not recently used" type of algorithm that is an approximation to the LRU algorithm. This can be implemented in many ways. Intuitively it seems reasonable to suppose that the more historical bits that are maintained about page usage, the better would be the choice of page to be replaced. However this requires more stored information and updating, which is expensive and time-consuming. Hence a tradeoff is necessary. In the selection of a page for replacement, one would expect that if a page has not been referenced over a certain time period, it is less likely to be needed next than pages that have

been referenced. Hence a reference bit R for a certain time period is necessary. But before selecting a page for removal, it must be recognized that if a page has not been modified (i.e., has not been written into), it need not be written back onto disk but simply can be overlaid (i.e., erased). If it has been modified, however, it *must* be placed back on the disk before erasure. Since this requires considerably more time than just overlaying, a second status bit is also stored with the previous R "reference" bit. The second bit M specifies whether the page has been modified while residing in main. A page that has not been modified is more eligible for replacement than one that has been modified. In addition, locking bits are often used to protect storage when a page is undergoing swapping, or for other reasons, which we ignore for now.

In principle, the replacement algorithm can be implemented with the R and M bits alone. Whenever a page is referenced, its associated R bit is automatically set to "1"; likewise, M is set to "1" when that page is modified. Eligibility for replacement then goes as "not modified and not referenced," "not modified but referenced," and so on. The time period over which the "not referenced" criterion exists requires some consideration. The M bit is not affected because once modified, a page must be so indicated for the entire time it is resident in main memory. The time period over which the R bit is evaluated can be a fixed or variable period.

Theoretically we do not have to evaluate pages for replacement until a page fault occurs in main memory. At this time, we can scan all pages for replacement, select one, and reset all R bits to zero, indicating a new time period. We may introduce additional historical bits H, to indicate the number of such successive scan intervals over which each page was not referenced. One historical H bit would allow indication of usage during the previous scan interval, two bits over four intervals, and so on. For such a case at scan time, any page with $R = 0$ would have the "unreferenced interval" H bits incremented by binary "1" and R would remain unchanged. If R were "1," indicating "referenced," the H bits would be turned off and likewise R would be turned off. A large binary value for the H bits would indicate a large number of successive intervals over which that page was not used. Of course the opposite polarity could also be used. Eligibility of pages for replacement is then modified to include these H bits, pages with a large binary value of successive "not referenced" intervals (large H) being more eligible than those with smaller values. The scan interval in this case would be variable, since it is initiated by a page fault. This scanning can also be done periodically during fixed intervals. For various practical reasons, the latter scheme is often used in large commercial systems. There are, of course, numerous possible variations of this procedure as well as other replacement procedures. The cost/performance tradeoffs vary with the system as well as the manufacturer, hence the details can differ substantially. This type of algorithm is often referred to as an approximation to the LRU algorithm, but it is a gross approximation at best. It really replaces pages *not* recently used in a somewhat random fashion according to the simple referenced/modified priority scheme. Considerable logical processing is required, but much less than for even slightly more complex algorithms. The implementation can be in hardware or software, but more of the latter is common.

In the cache–main memory type of virtual system, which typically uses set associative mapping, a required page that must be brought into the cache can go only into a fixed logical set. Hence a page must be removed from *that* logical set. Typically an LRU-type algorithm for each set is stored and processed with separate hardware. This is basically the technique used on the IBM 195 and 168.

Of the various types of algorithm in use, the LRU has intuitive appeal as an average technique over a large job stream, despite many exceptions. Since a pure LRU over a large number of pages is rather difficult to implement, some approximation of the type described earlier is used in main–disk virtual memories. However the mapping of a cache paged out of main memory is typically set associative (see Sections 9.10 and 9.11) with very few page slots per set. An LRU algorithm is typically implemented in hardware for each set of the cache. The following discussion considers some of the fundamental aspects of implementing a pure LRU in a general case.

9.7.1 LRU Algorithm Implementation

To implement any general ordering type of algorithm, some means must be provided within the system for keeping track of each page usage with respect to all other pages. In other words, all pages must be continually ordered among themselves for each new page reference. This calls for two fundamental pieces of information: (*A*) the address of the pages within a given ranking order, and (*B*) the relative order or ranking of page usage among themselves. In conjunction with these two pieces of information, two fundamental operations are necessary:

A'. *Search Rank for Address.* Search usage information and produce address of page with required ranking.

B'. *Update the Usage Information.* Enter new page slot usage into the ranking after each page reference.

These operations can be implemented with the use of associative or nonassociative functions.

9.7.2 LRU via Associative Functions

In general, there are two basic ways to implement an associative ordering algorithm. We can store the address of the pages within a stack of registers

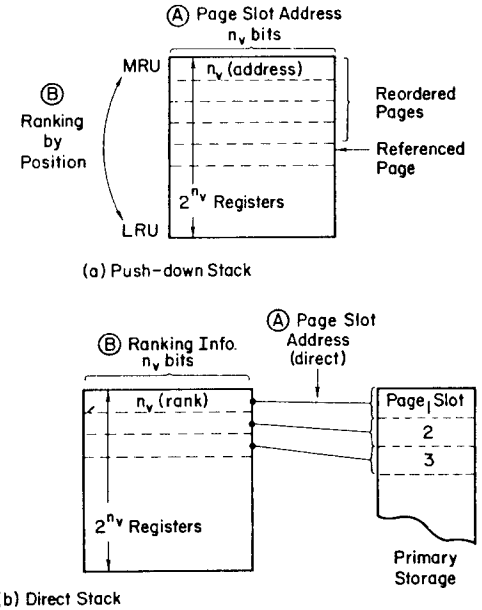

FIGURE 9.7-1 Two fundamental schemes for implementing LRU replacement algorithms by way of the associative function.

and perform the ordering by the arrangement of the stack (i.e., order from top to bottom of the stack via a push-down (or up) stack) Fig. 9.7-1*a*. An alternative is to store the ordering within a stack of registers and allow each register to have a direct, one-to-one correspondence to each page slot in primary storage. For instance, the top of the stack contains ranking (usage) information for page slot 1, and the bottom register contains the ranking for page slot 2^{n_v} (Fig. 9.7-1*b*). Thus for the very general implementation, we can store the page slot address in registers and perform the ranking externally by ordering the registers from top to bottom (push-down stack) or, alternatively, we can store the ranking within the register, allowing the page address to be determined externally by the physical correspondence to page slots. If an ordered stack* is used as in Fig. 9.7-1*a*, the bottom (or top) of the stack contains the identity of the least recently used page and the top contains the most recently used page. Under these circumstances, B is determined by the physical ordering of the stack and requires no extra information. In such a case, A, the identity of the pages within the stack, requires a number

* The stack can be a list in main memory or special hardware registers.

of bits given by

$$\text{identity bits/page slot} = \log_2 p = n_v \tag{9.7-1}$$

where p is the number of pages requiring ordering. The stack would have $p = 2^{n_v}$ entries each of the length given. The fundamental operation A' of finding the LRU page is very simple and requires only reading the address from the bottom entry. However, operation B' is not so simple. A new reference to any page within the stack, except the most recently used page, involves considerable logical reordering of the stack. For instance, the ranking or actual register corresponding to the referenced page must first be found, and thus requires some kind of associative compare, either serial by register or all in parallel. After the correct ranking position is found and the register position stored, this page slot address must be moved to the top register (MRU), and all registers from the top, down to the previous register position of the referenced page, must be pushed down by one ranking position.

In the alternative scheme of Fig. 9.7-1*b*, the situation is not much better. We still need 2^{n_v} registers, one for each page slot, and n_v bits per register, to be able to rank these from 1 to 2^{n_v}. Note that the same *number* of bits is stored in both schemes, but the information is different (an address in the first case and a ranking order in the second). The operation of entering new ranking or usage information (i.e., B') is very easy for the referenced page it now becomes the highest rank (i.e., MRU). However reordering these pages between the previous MRU and the previous ranking of the referenced page becomes quite complex. An associative search must be made on each entry to see whether it requires reordering; if so, it is lowered by one. The operation of finding a page slot to be replaced (i.e., A') also requires an associative search to determine which of the registers contain the desired ranking position (e.g., LRU rank).

9.7.3 LRU via Nonassociative Functions

The above method of implementing the LRU algorithm are only two fundamental ways that require associative compare capabilities. These associative techniques minimize the number of additional bits required for large numbers of pages, but the hardware becomes expensive and slow. It is possible to eliminate the need for associative compares, but for large numbers of pages, the additional stored information becomes large. Let us first study the basic technique, then compare the additional hardware required versus that in the previous associative schemes.

In the nonassociative scheme, each page is paired with every other page. If there are p pages to be ordered, the total number of pairs of pages is p pages for the first one of the pair and $p - 1$ for the second, or a total of

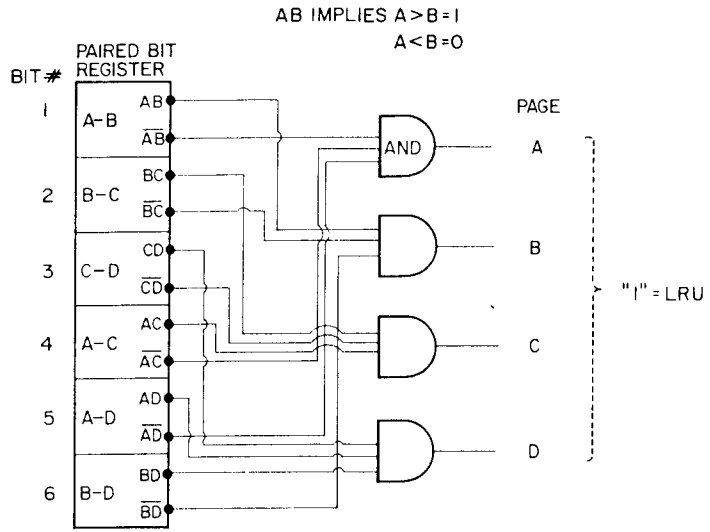

FIGURE 9.7-2 Nonassociative implementation of LRU algorithm with pairwise inequality bits.

$p(p-1)$ pairs. However there are many duplicates, since pairs such as AB are identical to BA. Hence the number of distinct pairs is half the value just given, or

$$\text{number of distinct pairs} = \frac{p(p-1)}{2} \tag{9.7-2}$$

where p = total number of pages.

One ranking bit is stored in separate hardware for each pair. The bit is, say, "1" if A is more recently used than B and "0" if A is less recently used than B. The total number of such stored ranking bits is thus equal to the number of pairs of (9.7-2). The general idea for the ranking is simply that these bits form a pair-wise inequality that uniquely specifies the LRU page. For instance, if we have three pages and we know $A > B$, $B < C$, and $A > C$, we can deduce that B is the lowest entry. The question now is, How can we implement this in nonassociative hardware? It is actually quite simple. Each page has one AND logic gate associated with it (Fig. 9.7-2) for the example with four pages. Each AND receives an input from a "pair bit" that has that page for one of its pairs. For instance, page A must have an input from the pair bits A–B, A–C, and A–D. Similarly page B must have an input from pair bits A–B, B–C, B–D, and so on. Thus the fan-in must be $p-1$. If we specify that the AND gate with an output signal indicates the LRU or lowest

ranking page, we must connect either the pair-bit or its complement, depending on which page in the pair, being more recently used, gives a "1" output. For the pairing in Fig. 9.7-2, if $A > B = 1$, $A < B = 0$, $B > C = 1$, $B < C = 0$, and so on, the complements must be used on pages for which that page is the higher ranking in the pair as shown. Obviously searching for the LRU page is trivial compared to the associative case. The LRU page is always specified by the AND gate that is "on."

Let us now determine the number of stored pair-bits required for this nonassociative scheme for a system that has, say, 256 pages, all of which are ordered. From (9.7-2), the additional storage required is

$$\text{pair-bits} = \frac{256(255)}{2} = 32{,}640 \text{ bits}$$

In addition, 256 AND gates each with an equivalent logic tree fan-in of 255 are required. For the associative scheme, (9.7-1) indicates that the total number of stored bits is only

$$\text{total associative bits} = p \log_2 p$$
$$= 256(8) = 2048 \text{ bits}$$

This is more than 15 times less than the nonassociative scheme but requires associative hardware. If one uses set associative mapping, only the LRU within each set must be maintained. For instance, for a four way associative set (i.e., $s = 2$), only 6 bits/set are required as in Fig. 9.7-2. Such a scheme is actually used on the IBM model 195 cache (Section 9.11.2). Other schemes are possible for reducing the total number of stored bits, but the logic can become complex. For instance, it is possible to store one string of bits specifying the entire ranking of all pages relative to one another. For p pages, there are a maximum of p factorial combinations or rankings relative to one another. The number of bits required to specify any one relative ranking of all pages is $\log_2 p!$. This can be substantially less than that for the previous cases. The scheme takes on the form of a decoding tree in which one logical combination must be decoded from the given bit stream. For large p, the logic operations become complex for searching (decoding) and, especially, for updating after a reference.

There are, of course, many possible ways to implement replacement algorithms, but all require additional hardware and time. In addition, the updating after a page reference, which must be performed by the storage control unit, can become quite complex even in the scheme of Fig. 9.7-2. The IBM 370 models 195 and 165 cache memories use the latter type of set LRU implementation, whereas the model 168 cache uses a stored address type of stack register similar in principle to that of Fig. 9.7-1*a*.

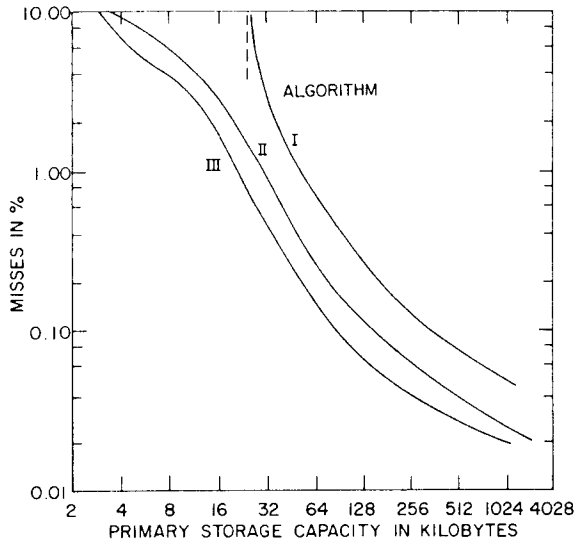

FIGURE 9.7-3 Percentage of misses versus primary storage capacity for various algorithms, showing hypothetical general cases.

9.7.4 Effect of Replacement Algorithm on Hit Ratio

The variation in hit ratio for different replacement algorithms is a matter of fundamental importance. Since it is a function of many variables, we give some general trends* and demonstrate a basic technique for circumventing the problem.

If the miss ratio is plotted as a function of primary storage capacity for a fixed page size, a curve similar to one of those in Fig. 9.4-2 is obtained for any replacement algorithm. The miss ratio must obviously approach 100% as primary storage capacity approaches zero, and it must approach zero as primary capacity approaches the total capacity required by the problem or job stream. If we consider these curves as applied to specific users in the job stream, the primary capacity axis becomes a measure of the number of pages allotted to that user. If we were to plot similar curves for different replacement algorithms, the results might be something like the hypothetical curves of Fig. 9.7-3. All show a general improvement with increasing storage capacity. For a specific user, one of the algorithms (e.g., I in Fig. 9.7-3) might require a certain minimum number of pages to achieve any reasonable miss

* See Belady (1966) for discussions and performance of various algorithms relative to the optimum for specific cases.

ratio, because of the nature of the problem. This could arise under the conditions described by the example in Section 9.14 or similar cases. If the allotted pages were in this "minimum" neighborhood on the horizontal axis, a different replacement algorithm could give a much improved miss ratio. If the algorithm is fixed, as is usually the case, an improvement can also be obtained by increasing the number of pages or primary bytes allotted to that user. Unfortunately in a given multiprogrammed system, this reduces the number of pages available to other users. Thus while the hit ratio may be improved for the first user, the overall system hit ratio and through-put may degenerate. Theoretically, dynamic fine tuning of the system hit ratio is possible but is usually too complex to be feasible.

9.7.5 Working Set Replacement Algorithm

Denning (1962) attempts to dynamically assign the number of page slots of primary storage allotted to a given user in terms of past history. The working set for a given user is defined as the set of pages that are referenced in a given period of time. In other words, referring to Fig. 9.4-1*a* and assuming that the address references refer to pages rather than words, the working set would be the total number of different pages appearing in the trace for a single user over some specified number of time intervals, usually referred to as the window size. The longer the time period, the larger the working set and hit ratio for a given user, but the poorer the hit ratio will be for another user. If the time period is too small, a very poor hit ratio will result. The major problem is then determining the correct value of the time period. Additional fundamental statistics are required to ascertain whether the optimum time period is substantially smaller than the problem running time to be useful. Also needed is information on variations between problem sizes and problem types.

9.8 VIRTUAL MEMORY SYSTEM DESIGN CONSIDERATIONS

Using the previous sections, let us now consider the conceptual design of a virtual memory system consisting of main memory and a disk-like backing store. Remember that the primary goal is to bridge the capacity gap between main memory and disk. However the overall speed must be reasonable, or the primary goal is of little value.

The system is assumed to be identical to that of Fig. 9.3-2, consisting of $U = 2^u$ users, each with a virtual store of 2^{N_i} pages or 2^N words as shown.

Each of the three fundamental requirements must be implemented with the design consideration in mind. The choice of page size and replacement

algorithm depends on problem statistics and is not amenable to any analysis from first principles. The results of Section 9.4 indicate that the LRU algorithm can give adequate miss ratios. However the LRU algorithm is difficult to implement, and an approximation to LRU is commonly used.

The required miss ratio for a given system can be determined only by simulation. In a large system, a miss causes transfer of control of the CPU to a new user whose pages are in primary storage. The transfer of control may require tens, hundreds, or thousands of CPU execution cycles. Although time-consuming, it nevertheless introduces much less delay than the page access and relocation time. However, maintaining a high CPU utilization requires a small miss ratio—in the range of 0.02% or less. To achieve this ratio, the results of Section 9.4 indicate that page sizes of 1K′ to 4K′ bytes, with a minimum of 256K′ bytes of useful primary storage, are required for the job streams and parameters used. The actual values change for different job streams, but the difference is not orders of magnitude. These parameters represent typical values that are adequate for many cases. The two address translation functions of Fig. 9.3-2 (external and internal) can be implemented separately; thus the type of tag store to be used must be decided for each. The external translation, which converts N_s into a real disk address, is required only on a page fault in main memory. This occurs rather rarely, and when encountered, the CPU transfers to a different user already resident in main memory. Thus speed is not a major concern, and this translation can be implemented economically with a table, called Table I as in Fig. 9.8-1. This table contains a maximum of 2^{N_r} contiguous entries, one for each virtual page. A software algorithm uses the N_r address to access the table. Each

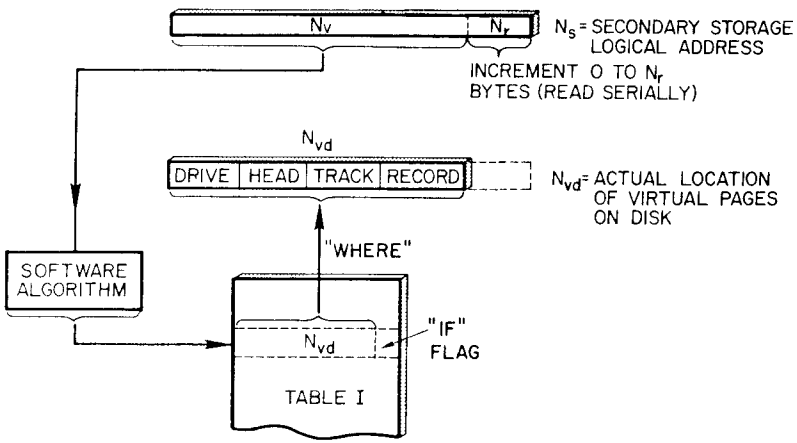

FIGURE 9.8-1 Translation of logical address N_s to actual disk page address using Table I.

entry contains an IF flag, the actual WHERE address, and any other desired control functions. Such an implementation automatically makes Map 1 in Fig. 9.5-10 fully associative. Since Table I potentially can be very large, it is normally stored on disk and transferred into main memory when a user is entered. In principle, Table I can remain on disk (located, e.g., as the first file on the first disk, etc.). This would conserve main memory space but at the expense of a sequential search of the disk via the I/O processor. The latter is slow but would be feasible if sufficient time were available between page faults. Since this is not usually the case, Table I would have to be entered when needed.

The internal translation, which converts N_s into n_p, is required on every memory reference; hence speed is of vital concern. In addition, maximum flexibility in physical page location within main memory is necessary, making a fully associative Map 3 in Fig. 9.5-10 essential. The need for speed suggests the use of a fast, minimum tag store (directory) with a fully associative Map 3 and set associative Map 2 in Fig. 9.5-10. However this directory would need 2^{n_r} logical entries, and the associativity would have to be rather large for a good hit ratio. In other words, the directory would be large and would have to perform many associative compares, which would make the directory slow and very expensive. The use of a table translation scheme stored in main memory would be relatively inexpensive and would provide fully associative mapping. However every CPU reference to main memory would require a minimum of two or more main memory cycles, which is a severe speed penalty. To circumvent this dilemma, large commercial systems typically compromise, storing a few of the most recently used pages in a very small partial directory for speed, while using a table as backup. Examples are given in Sections 9.9 and 9.12. The partial directory is usually quite straightforward and need not be detailed here. The table translation, however, becomes somewhat complex because of practical considerations.

Table Translation: Design Considerations

In principle, the table translation of the virtual address N_s into a real address in main memory can be accomplished with a single table. This table (referred to as Table II) is fundamentally as shown in Fig. 9.6-1, although this represents a very general means of address translation. For a virtual memory system as outlined in Section 9.3 and specifically in Fig. 9.3-2, one important requirement that does not appear in the scheme of Fig. 9.6-1 is the means each user employs to address the memory in terms of the full CPU logical address N. To see this, the reader should study Fig. 9.5-2*b*, which gives a complete identification of all address bits used previously. The total virtual or secondary store logical address N_s consists of the processor logical address N, and the user ID bits u. In other words, $N_s = u + N$. Not only must each

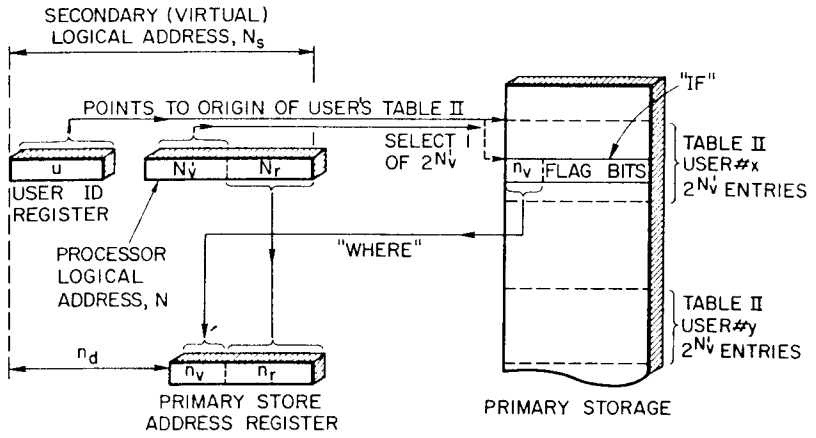

FIGURE 9.8-2 Translation of virtual address N_s into real, primary store address using single table for each user.

user be able to address main memory as if it consisted of 2^N words, but the system must contain the capability to be transferred from one user to another when a page fault is encountered. The simplest way to achieve this is shown in Fig. 9.8-2. The user ID bits are contained in a separate register. These bits give the real, origin address of that user's Table II in main memory. The virtual address part of the CPU logical address, namely, N_v', then increments to one of the entries in the table. This entry contains a minimum of n_v plus one flag bit as specified in Fig. 9.6-5, where $n_v = q + s$. Thus the higher order bits

$$u + N_v' = n_d + q + s' \tag{9.8-1}$$

do indeed point to the table as shown in Fig. 9.6-1. Each entry in the table must contain the higher order address bits n_r for *each page* of that user. Since each user must be able to use the full CPU logical address length, each user has $2^{N_v'}$ pages as in Fig. 9.3-2. Thus Table II must contain $2^{N_v'}$ entries for each user. In the implementation in Fig. 9.8-2, when the processor transfers to a new user, the ID bits are changed in the user ID register and then point to user y, for instance. Thus each user current on the system must have a Table II stored in main memory. The following analysis reveals that it is generally not possible to store Table II as a single table; hence it is divided into smaller parts. Nevertheless, the concept of address translation using tables is identical in principle to that achieved with a single table as shown. If Table I is stored in main memory for the active users, it can be addressed in a manner very similar to that used for Table II.

To see why a large table and only a partial tag directory is used for the internal address translation as well as why Table II is divided into parts, let us make the approximate calculation of the additional storage capacity and number of associative compares required for a full-sized directory and table translation. This requires some estimates of the values of various parameters. If a virtual memory is to be effective, the total external storage should be considerably larger than main memory by a factor of 1000 or more. We will assume a minimum value of

$$2^{N_s} \geq 1024 \times 2^{n_p} \qquad \text{or} \qquad N_s \geq \log_2 1024 + n_p = 10 + n_p$$

Using the definition of n_d given by (9.3-7), we have

$$n_d = N_s - n_p \geq 10 \text{ bits}$$

For further parameters of the system, let us assume that $n_r = 10$ bits (i.e., 1K′ words/page) and main memory has the minimum size of 256K′ words or $n_p = 18$ bits. Since $N_s = n_p + n_d$, the number of secondary storage address bits is $N_s = 18 + 10 = 28$ bits (this in somewhat on the low side). We further assume that the mapping is fully associative or $q = 0$, $s = n_v = n_p - n_r = 8$, and that each user can have $2^{12} = 4096$ pages maximum.

The various parameters are thus

$$n_p = n_v + n_r = 8 + 10 = 18$$

$$s = n_v = 8 \qquad q = 0 \qquad n_d = 10 \qquad u = 6$$

$$N_s = n_d + n_p = 10 + 18 = 28$$

$$N_v = n_v + n_d = 18 \qquad N_v' = 12$$

From Fig. 9.6-5, the minimum storage capacity for full translation for all $2^6 = 64$ users is

Directory Storage

$$2^{n_v} = 2^8 = 256 \text{ words}$$

$$n_d + 2s = 10 + 16 = 26 \text{ bits/word}$$

Table II Storage

$$2^{N_v} = 2^{18} = 256\text{K}' \text{ words}$$

$$1 + s = 9 \text{ bits/word}$$

The table storage requires a 100% increase in addressable words or entries at a minimum of 2 bytes/entry, assuming 8 bits/byte. Note that a similar storage capacity would be required for Table I if it is all stored in main memory. The full directory requires one-thousandth as many addressable

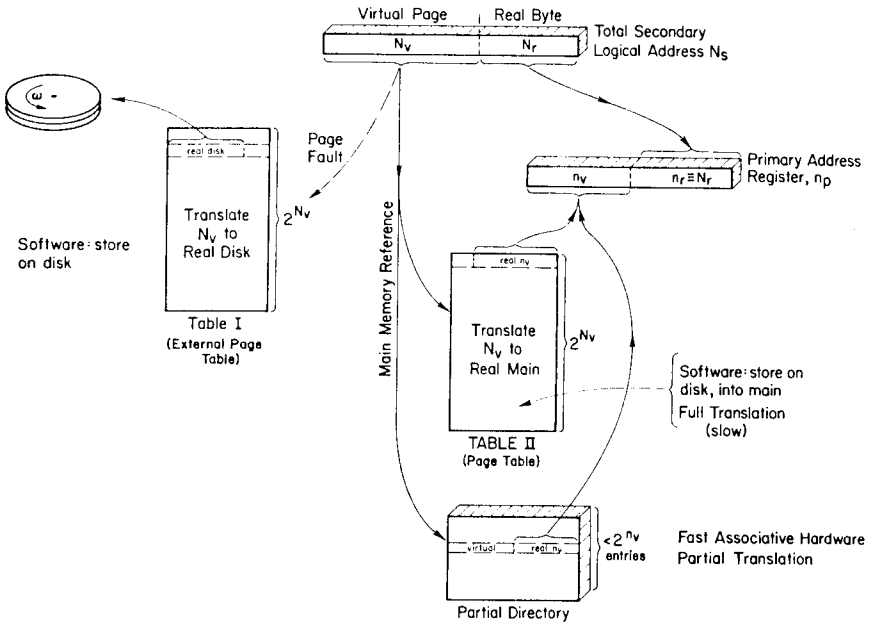

FIGURE 9.8-3 Address translation on typical virtual memory system using tables and partial directory.

words as the main memory, with a minimum of 4 bytes/entry. The latter thus seems more reasonable except that $2^s = 256$ associative compares are required, whereas the table needs no associative compares. Both methods appear to be rather expensive, and the designer is faced with a dilema that is solved by the use of both, as previously indicated. This solution is shown schematically in Fig. 9.8-3 for typical commercial systems. A CPU reference to main memory initiates the address translation simultaneously to the small, fast, partial directory and to the slow Table II. Since the partial directory contains only a few (e.g., 8 or 16) of the most recently referenced pages, a page may be in main memory but without an entry in the partial directory. If the latter is true, the translation proceeds by way of Table II. The real and virtual addresses of the page are then entered in the partial directory, since it is most likely to be referenced next. If the required page already was present in the partial directory, the Table II access would be aborted. When the required page is not resident in main memory, a page fault occurs. Then Table I must be accessed to obtain the real address as discussed in Section 9.3.

For the partial directory scheme to be effect, it must have a high hit ratio. A small hit ratio would require excessive translations via Table II in main memory, which would seriously deteriorate system speed. The partial directory must contain enough entries to achieve a high hit ratio, thus it is an important design parameter.

The above analysis assumed that all 64 users used the maximum number of virtual pages and all 64 tables were stored in main memory. Two serious objections are first, few users actually use the full virtual memory size, second, in the normal course of operation, programs are completed and new users must be entered. Thus it would be desirable to make these tables pageable just like other data, to permit relocation as required and also to provide a means for acquiring a variable table length. For the most general case, a pageable table (Table I or II) requires a table hierarchy and automatically provides a means for variable table length. The need for a table hierarchy can easily be understood as follows. In general, the maximum size of Table II for any given user can be much larger than one main memory page slot. If Table II is divided into many such page sizes, *physically* each page can reside in any real main memory page slot, but *logically* the table must be contiguous. Thus some means is necessary to associate the logical contiguity with the random physical locations. This can be done with a small, additional table that maintains a listing of the real origin of any portion (page) of Table II. If this additional small table exceeds a page slot size, there must be another level of this table hierarchy. The number of levels required in this table hierarchy for accommodating a single *logical* Table II for each user can be deduced from a knowledge of the page slot size of 2^{n_r} bytes or words, the maximum number of virtual pages per user of 2^{N_v}, and the necessary number of bytes or words per table entry B_e. Let us assume for simplicity that B_e can be expressed as a power of 2, or $B_e = 2^{n_e}$. A page slot then holds a number of table entries given by

$$\text{number of table entries per page slot} = \frac{2^{N_r}}{B_e} = 2^{N_r - n_e} \qquad (9.8\text{-}1)$$

In effect, the table entries in this page slot are addressed by $N_r - n_e$ address bits. If we assume that any additional levels in this table hierarchy also require B_e bytes or words per entry, the number of levels can be determined by the number of groups of $N_r - n_e$ address bits required in the total virtual address per user N_v'. Thus the number of levels in this table hierarchy is the higher integer value of the ratio of total table address bits divided by the effective address bits per page slot, or

$$\text{number of levels in page table hierarchy} = \left\lceil \frac{N_v'}{N_r - n_e} \right\rceil \qquad (9.8\text{-}2)$$

Obviously a one-level table results when $N_v' \leq N_r - n_c$. In the previous example we had $N_v' = 12$, $N_r = 10$, and $n_c = 1$. Substituting these into (9.8-2) gives

$$\text{number of levels} = \left\lceil \frac{12}{9} \right\rceil = 2$$

Thus our one long contiguous logical table of Fig. 9.8-2 for each user would require another level to contain the origin of the various pageable parts of the actual table. The next section indicates that the virtual address space and page sizes result in the need for a two-level table in IBM 360/370 virtual system and a three-level table in Multics.

If the external page table, Table I, is stored and paged into main memory, it must be handled in a manner analogous to that given. The number of bytes or words per entry may be different as a result of additional information which is conveniently stored in these tables.

It should be clearly understood that these translation tables are not fundamentally necessary. The address translation can be done in hardware with a tag store. Most commercial virtual memory systems use such tables created by software and paged into main memory because they provide a flexible, economical scheme. In addition to address translation, these tables can provide such other functions as the control of access rights and sharing of pages among users. These multiple functions are often incorporated into the table because of convenience and economy. Fundamentally, they can also be done in other ways with hardware or software. The details of such subjects require a consideration of the overall architecture of a virtual computing system that is beyond the scope of this chapter.

The replacement algorithm typically used in virtual memory systems is some form of a "not recently used" algorithm discussed in Section 9.7. The page usage and control information can be stored in hardware/software-created tables, or both; typically combinations of both are necessary. The various commercial and supervisory systems offer wide variations in details, but not in principle.

Current virtual memory systems tend to implement considerable amounts of the necessary functions in software for economy. However this has created enormous supervisory programs and tables that not only are complex, but consume ever-increasing amounts of main memory. It can be shown that all these functions can be implemented in associative-type hardware either in a separate array or on-chip as in Section 9.13, while still providing overall system flexibility and fine-tuning capability. This hardware would be highly desirable if sufficiently inexpensive and reasonably fast. Eventually the new high density, low cost FET may take over more and more of these functions.

9.9 EXAMPLES OF VIRTUAL MEMORY SYSTEMS

Examples of a few large virtual memory systems are included here.

9.9.1 Ferranti Altas

One of the early paged virtual memories, the Ferranti Atlas, used a 98,304 word magnetic drum secondary store paged into 16K′ core primary store.* The total logical address length, N_s was 20 bits, allowing a maximum of 1M′ words for all users. This was the total size of secondary storage permitted on the system, and it was all addressed randomly by the 20 bit logical address. Hence there was only one mode of addressing, and the system represented the first operating example of a one-level store. Secondary storage was divided into 2048 pages of 512 words/page. Thus the logical address consists of 9 lower order real bits and 11 higher order virtual bits (i.e., $N_r = 9$, $N_v = 11$, as in Fig. 9.9-1. Main memory (primary) thus contained a maximum of only 32 page slots, which requires a higher order address length of $n_c = 5$ bits. Hence the address translation must convert the 11 bits of N_v into an IF (yes/no) command and a 5 bit page slot address specifying WHERE the desired page is in core. This is accomplished by a tag directory. Each of the 32 page slots in primary store is assigned one register or word of the directory. The 11 bit virtual address tag N_v of the correct page stored in each primary slot is maintained in the directory.† For translating a given logical address, the 11 N_v bits are associatively compared with the 11 bits of all 32 tags in the directory. If no match is found, a relocate cycle is initiated. If a match is found, the register (i.e., page slot number) of the directory is flagged, and its address, expressed as 5 bits binary, is used as the higher order bits n_c of the primary store address register as shown. The 9 real bits complete the 14-bit address required to select one of 16K′ words in primary storage. When a miss occurs, location of the desired page on the drum is accomplished by use of a page directory (i.e., Table I of Section 9.8). This table stores the 11 bit logical page address and the associated correct drum address. The 11 bit logical address of each page is scanned sequentially until the correct drum address is found. Subsequently that address is used to read the desired page into primary storage. This tag directory is similar to the cache directory used in the IBM model 85 in that the binary address of the entry in the tag directory serves as the higher order address bits of the primary address register. In other words, since there is a one-to-one correspondence between the position of the tag in the directory and the position of the page (or sector in model 85) in the primary store, in essence Map 3 (Fig. 9.5-9) is direct.

* In addition, there was approximately 4M words of tape storage for I-O.

† A twelfth bit for storage protect is also stored in the directory.

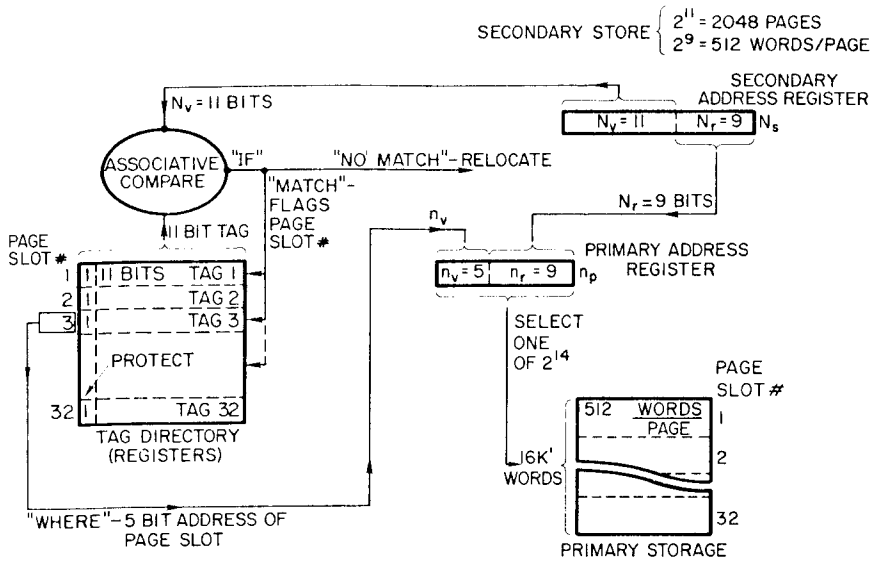

FIGURE 9.9-1 Schematic of virtual paging scheme on Ferranti Atlas.

To determine which page to remove from core to make room for a new page, the Atlas system used a learning program, part of which is similar to the LRU algorithm. Each of the 32 page slots in primary storage has associated with it a "use" bit stored in a separate register. Hence there are a total of 32 "use" bits. A given bit is set to "1" whenever a word from the current page in that slot is used. The learning program reads these bits destructively, setting them to "0" but storing them in a new list in a subsidiary memory (private store). These "use" bits are read every time 1024 instructions have been executed as determined by an instruction counter.* These bits are added to the subsidiary store list, then used by the program to update two sets of times also stored in the subsidiary memory. These times consist of 32 values for t and 32 values for T, one of each for each page slot in primary storage. The value of t is the length of time† since the current page in that page slot was last used: T is the length of time of the last previous period of inactivity of that same page. A page is then removed from primary store based on three simple tests in the following order of priority: remove

* This is not real time, since I/O interrupts can cause real time to be any value.
† Time measured in 1024 instruction intervals.

the page that has

1. $t > T + 1$.
2. Maximum value of $(T - t)$ with $t \neq 0$.
3. Maximum value of T with all $t = 0$.

The first rule selects the page that has been out of use for the longest time, hence is equivalent to LRU except that in this case it is possible for no page to qualify. Rule 2 ignore pages in current use ($t = 0$) and selects the page that will not be needed by the program for the longest time. If the first two rules fail, rule 3 ensures that one page will be selected and also that if this page is immediately required again, T becomes 0 and the same mistake will not be repeated on the next page replacement.

When a page is relocated back to the drum, a means is also provided to store the value of T so that when this page is once again called to primary storage, the value of T is set in the subsidiary memory.

The Atlas was not multiprogrammed in the sense that program control was not transferred when a page miss occurred. Nevertheless, it was partitioned such that output from a previously completed program or input for a new program could share main memory during idle cycles (e.g., page misses) of a third program that was being processed.

This machine thus represents a one-level store of a very simple type. One limitation is the small number of page slots. Also, and more serious, is the maximum of 1M′ logical words to be divided among all programs if there are several programs resident at one time. It is more desirable to allow each user to have the full 20 bit addressing capability of the system and to multiprogram the entire logical address space. However this could not be done because there was no user ID register, hence no "u" bits as in Fig. 9.3-2. This feature evolved in later systems.

9.9.2 IBM System 360/370 Virtual Memory Fundamentals

The IBM virtual memory systems implement the various requirements of Section 9.3 in a manner very similar to that outlined in Section 9.8 in the following way:*

1. *Page Mapping Function.* This is fully associative both from disk† to main memory and from main memory back to disk.

2a. *Word Addressing.* Pages consist of from 1K′ to 4K′ bytes, depending on model.

* This section is not intended to be a full description of IBM systems; it merely indicates how the fundamentals are implemented.
† In some systems the secondary store may be drums, disks, or a combination.

2b. *Page Address Translation.* This makes use of a partial tag directory and table schemes to capitalize on the strengths of each while minimizing the respective weaknesses. The tables make use of a Table I and Table II as described in Section 9.8, and both are implemented entirely in software.

3. *Page Replacement Algorithm*: An approximation to the LRU algorithm is implemented in various ways, depending on the system.

a. I/O Processor. This function is provided by the channel coupled to the control units as explained in Section 9.2. The channel is a basic part of system 360 and 370 architecture, with or without virtual storage.

b. Storage Protection. This is provided in the form of keys associated with each 4K′ word module of main memory. The modules and keys are assigned by the operating system and only protect supervisory programs from unauthorized usage. They do not protect users from each other and are required irrespective of the virtual memory.

c. Sharing of Pages. Global sharing of supervisory programs is implemented by way of Table II.

We now cover the address translation in more detail with particular reference to the system 360 model 67, although the basic concepts pervade the 360/370 virtual systems. The CPU logical address is 24 bits, (32 bit version available on 360/M67). This provides a virtual store of $2^{24} = 16\text{M}'$ bytes/user. The user ID bits in Fig. 9.8-2 are maintained in a separate 32 bit register as shown.

The maximum value of n_p is 24 bits, which would allow 16M′ bytes of main memory. However n_p is generally much smaller but has a lower limit for each system (e.g., for the model 67, $18 \leq n_p < 24$). Some typical parameters for a system might be as follows.

Since fully associative mapping is used,

$s = n_v \leq 12 \qquad q' \equiv q = 0 \qquad s' \equiv N_v' = 12$

$s' \neq s$ except in special case of $s =$ maximum value

$n_p = 18$ or 256K′ byte main memory

$n_r = n_p - n_r = 6$ bits

$N_r \equiv n_r =$ 10 to 12 bits or 1K′ to 4K′ bytes/page

$N = 24$ defined by system 360/370 architecture

In principle, the address translation function is implemented exactly as as shown in Fig. 9.8-3. To allow a relatively large number of page references to be done fast, only 8 of the most recently used pages are kept in a partial directory, as illustrated in Fig. 9.9-2 for the model 67. This associative directory compares all the s' (i.e., N_v') bits to determine a match: hence u bits are

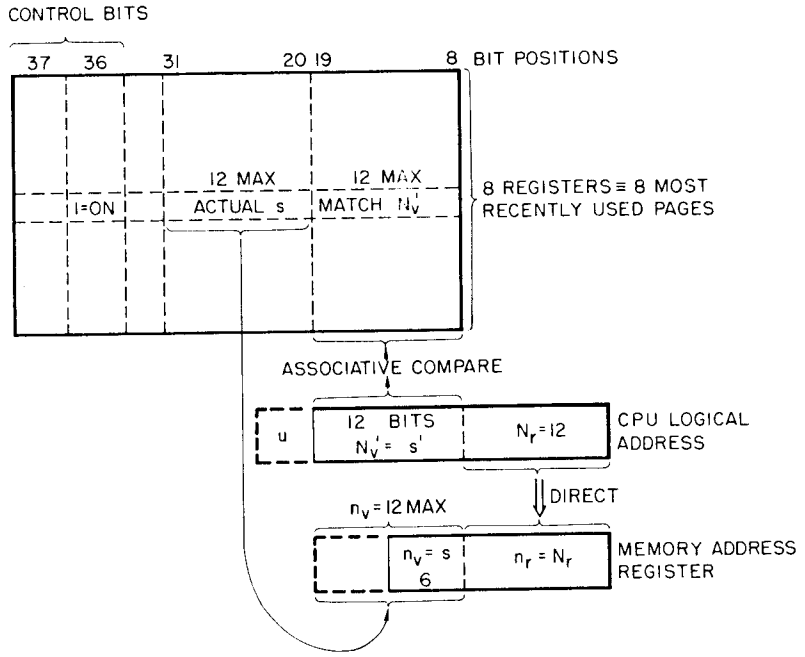

FIGURE 9.9-2 IBM 360, model 67, address translation, using small partial directory for the eight most recently used pages.

unaccounted for. These bits *must* be decoded or an invalid page could be referenced by the s bits found in a matching word. These u bits are taken care of by the use of control bits 36 and 37 of the associative register. Whenever a specific user is first given control of the CPU, these two control bits are set to "0" in all words of the register, meaning none of the pages is valid. The given user must first reference pages through the tables, after which time the associative register is loaded with the virtual address $N_v' = s'$ and the correct value of s for that page of that user as shown. This correct value for s is obtained from the page table, to be described shortly. When a page address is loaded in the register, the control bits 36 and 37 are both set to "1" to indicate a valid page for that user. A user must reference eight different pages through the tables before the partial directory is full. When all entries are full, such that all bits 37 equal "1," they are all set equal to "0" and bit 36 of all is still "1." A subsequent reference to an already present page will reset bit 37 to "1" for that page. Thus bit 36 indicates the presence of a valid page, whereas bit 37 indicates usage. In essence, then, bit 36 is the control

bit that indirectly decodes the u bits while bit 37 is used as a means for replacing entries. For instance, if a page reference is found to not exist in the directory, the first entry with bit 37 equal to "0" is replaced by the currently demanded page. Note that this does *not* constitute the page-swapping algorithm from secondary disk to main memory, rather, only replacement within the directory. This small directory does *not* perform a full address translation as would the directory of Fig. 9.6-4*a* or *b*. The currently demanded page may be in main memory (in the table described below) but not in the directory. If the CPU control switches between users too frequently, of course, the partial directory is of little value. Hence a high hit ratio is desired.

The main memory obviously holds more than eight pages. If the desired page is not one of those contained in the partial directory, the IF and WHERE are determined from a Table II much as that shown in Figs. 9.6-1 and 9.8-2. Actually the table search is initiated simultaneously with the partial directory as in Fig. 9.8-3. Table I and II can each be organized as one list as described in Section 9.8. However the need for pageability requires a multilevel table hierarchy. The number of levels is given by (9.8-2). For the current case, the real page address N_r can be from 10 to 12 bits, the number of virtual pages per user N'_v from 14 to 12, respectively, and $n_e = 1$ (2 bytes/table entry). Substituting these into (9.8-2) gives for IBM 360/370 table hierarchy

$$\text{number of Levels} = \left\lceil \frac{14}{10-1} \right\rceil \quad \text{to} \quad \left\lceil \frac{12}{12-1} \right\rceil$$

$$= 2$$

This two-level table hierarchy is organized as follows. Referring to Fig. 9.9-3, assuming $N_r = 12$ bits, the N'_v bits of the CPU logical address are divided into a 4 bit segment and an 8 bit page portion as shown to form a segment table and multiple page tables. Each user can have up to $2^4 = 16$ segments, with each segment having $2^8 = 256$ pages for a total of 4K′ pages per user as required. Each of the 16 segments of the segment table contains the origin address of one page table for that user.

The address translation by way of this table hierarchy proceeds as follows. The user ID register contains the 24 bit address of the origin of a given user's segment table in main memory. The 4 bit portion of the CPU logical address represents the lower order address bits and indicates how deep the desired word is within the table. The so-located word contains a length portion, a page table origin address, and a control bit. The control bit indicates whether the required segment of pages is actually present. The length portion is compared with the 8 bit page portion of the CPU logical word only to save time. The comparison is done very fast in CPU hardware and tells whether the page table is large enough to possibly hold the page identification bits.

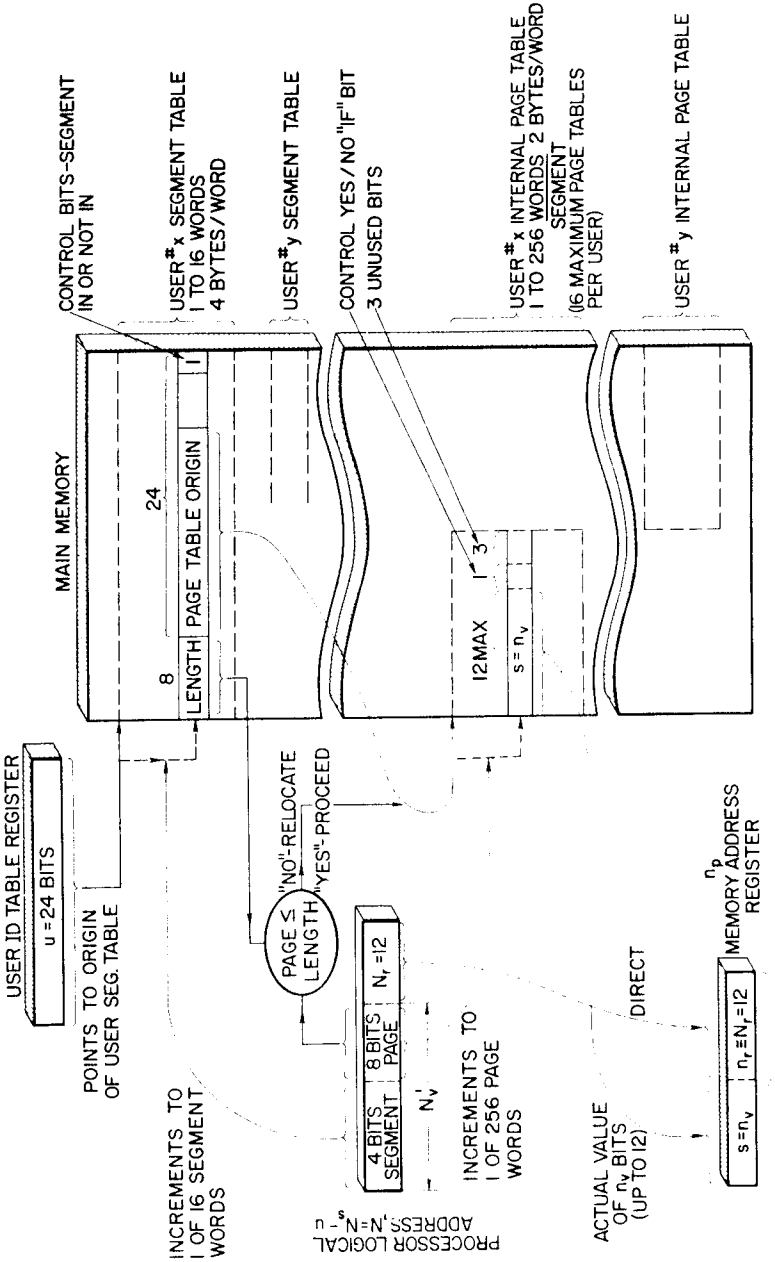

FIGURE 9.9-3 IBM 360/370 virtual memory showing address translation using segment and page tables.

If yes, the page table portion of that segment word points to the origin of the page table and the 8 bit page portion of the CPU logical address indicates how far into the table to go. The so-located word then produces the one-bit yes/no or IF control as well as the correct s bits, which are then placed into the memory address register for subsequent reference to the correct page. The lower order N_r bits in the CPU logical address translate directly into the n_r bits of the memory address register, hence the table hierarchy works exactly as the single table in Fig. 9.8-2. The external page Table I is treated in exactly the same way. For convenience Table I is appended to the bottom of each "internal page table" in Fig. 9.9-3 (not shown). Note that for the example given, the Table II slot occupies a maximum of 256 entries at 2 bytes/entry or 512 bytes. Since the assumed page size is 4K′ bytes, the remaining portion is allotted to Table I. When a page fault occurs, Table I is addressed as in Table II except the 8 bit page address is displaced by the appropriate amount. This seemingly complex system provides overall flexibility in virtual system implementation and compatibility among different models. The supervisory program is granted complete flexibility for control of the primary storage consumed by the various segments and page tables. These tables may be stored permanently for each current user or swapped dynamically as needed. IBM 360/370 batch processing systems generally load the full page tables, both Tables I and II, at the time a user is made active in the CPU. In time-shared systems, when a user is in a wait state his complete Table II, or only part, may be removed from primary storage if necessary. All these possibilities are design options that must be evaluated in terms of the actual system and type of job stream environment. Different types of virtual supervisory system, IBM CP67, VS1, VS2, MVS, TSS, as well as Multics and others handle these tradeoffs in different ways. There does not appear to be one best approach. In any case, the overall architecture of a hierarchy for Table II provides considerable flexibility.

A two-level table also facilitates global sharing of pages among users, although other techniques may be used. Two users may share a page by having the "page table origin" portion of each user's segment table contain the same information, hence pointing to the same page table in main memory. Sharing could be done with a one-level address table but becomes unwieldy.

A fundamental characteristic of this system is that the segment table consists of contiguous words in main memory. Consequently, if a user's storage requirement overflows from one segment into another, the higher order segment address bits are incremented by one to allow addressing of the next segment. In other words, addressing between segments is continuous. One disadvantage is that the user is limited to a finite number of segments, namely, 16 maximum in this case.

Since all references to main memory must proceed by way of the memory

address registers, the pointers form the user's ID register pointing to a segment table, the segment table word pointing to a page table, and page table pointing to actual page must be correctly loaded and aligned in the memory address register. This requires some control logic but is rather straightforward in principle and is not considered further. The replacement algorithm is essentially the "not recently used" algorithm described in Section 9.7 and is primarily software implemented. The various supervisory programs use varying number of historical H bits, called unreferenced internal count (e.g., VS1 uses 1 bit; MVS uses 8 bits). Storage protection makes use of a storage key that is associated with each 2048 byte block of main memory. Each program is assigned a protection key that is stored as part of the status word, or in a separate array. A memory reference into any specific block of main memory is permitted only if the user's protection key matches the storage key, or if the former is zero. These keys only protect the supervisory programs from unauthorized access.

9.9.3 Honeywell Multics Virtual Memory (645 Processor)

The intention behind the original Multics was to allow a user to program any problem with essentially an infinite number of segments. The system has evolved over the years, taking on various forms. In one form, the Honeywell Multics (Multics, 1972; Organick, 1972) allows a user to program up to 2^{18} segments with 2^{18} words/segment. In other words, the processor effective logical address is 36 bits long and each user may program as if a main memory of 2^{36} bits were available. But the main memory address register can contain a maximum of 24 bits and the real main memory may be smaller than this (e.g., 2^{18} words, or 18 bits effective main memory address register as in the previous example). Thus it is necessary to translate these 36 logical address bits plus a user ID into a real address of 18 to 24 bits specifying WHERE the required word might be and IF it does in fact reside there. This address translation function, assuming the logical address to be already available, is performed in a manner very similar to that in Figs. 9.8-2 and 9.8-3 but with some practical differences in detail. Memory and storage are divided into pages of fixed length (1024 words). The 10 lower order address bits of the logical address are real; hence as in all previous cases, these bits address one of the words in a page. This leaves $N_r' = 36 - 10 = 26$ bits/user for addressing pages. Since the table used for address translation must be pageable, a multilevel table hierarchy is required. Since $N_r = 10$ and $N_r' = 26$ bits, assuming $n_e = 1$ bit (2 bytes table entry), the number of levels for Multics, from (9.8-2), is

$$\text{number of Level 9} = \left\lceil \frac{26}{10 - 1} \right\rceil = 3$$

Obviously the larger virtual address has resulted in a more complex table hierarchy. The Multics address translation (Fig. 9.9-4) is implemented as follows. The origin of each user's first table is contained in the so-called descriptor base register, which is the user ID register described previously. This register contains the real core address of the first table for address translation and a field L_1 which specifies the length of the descriptor segment table in main memory. The processor logical address consists of two major components $s + i$, each of 18 bits for 36 bits total. This logical address is divided into four components, namely

$$s + i = s_x + s_y + i_x + i_y$$

These are used in conjunction with the descriptor base register (user ID) as follows. Referring to Fig. 9.9-4, the length bits L_1 of the descriptor base register are compared with the s bits to determine whether the first set of segment tables is long enough to contain the desired information.* If "yes," the u bits are catenated to the s_x bits to give the origin address of the page table PT_1 of the descriptor segment.† A flag bit F indicates presence or absence of the desired entry; F = ON indicates "not present." This flag provides one part of the "IF" translation function. If the flag is off, the contents of this word provide the origin address of the page of the descriptor segment table P_1, and the s_y bits increment to the correct entry in that page. The entry in this table contains the origin address of the page table, a length L_2, which specifies the length of the segment, an access field ACC, which specifies the access rights of this segment, and a flag bit F as before. The ACC field provides storage protection at the segment level. The second flag bit provides further "IF" translation. Assuming that the F bit does not generate a fault, the binary value of the L_2 bits is compared with the i bits to determine if the next table PT_2 is long enough to contain the desired information, as before. If L_2 is less than i, a miss is generated. If L_2 is greater than i, the ACC field is compared with the operation code of the instruction to determine whether the specified operation is permitted. If "yes," the origin field points to the page table origin and the i_x bits select one of 256 words in the page table PT_2. This page table likewise specifies the true higher order address bits of the required page; the 10 lower order address bits i_y, which are real and equivalent to N_r previously defined, give the correct word. The last two address bit fields constitute the final memory address as shown in n_p.

This table translation works in principle just as that described in Section 9.8 and previously except for details of the control bits and number of components into which the CPU logical address is divided. Because of the large

* These tables need not be filled.

† The nomenclature is that used by the Multics system.

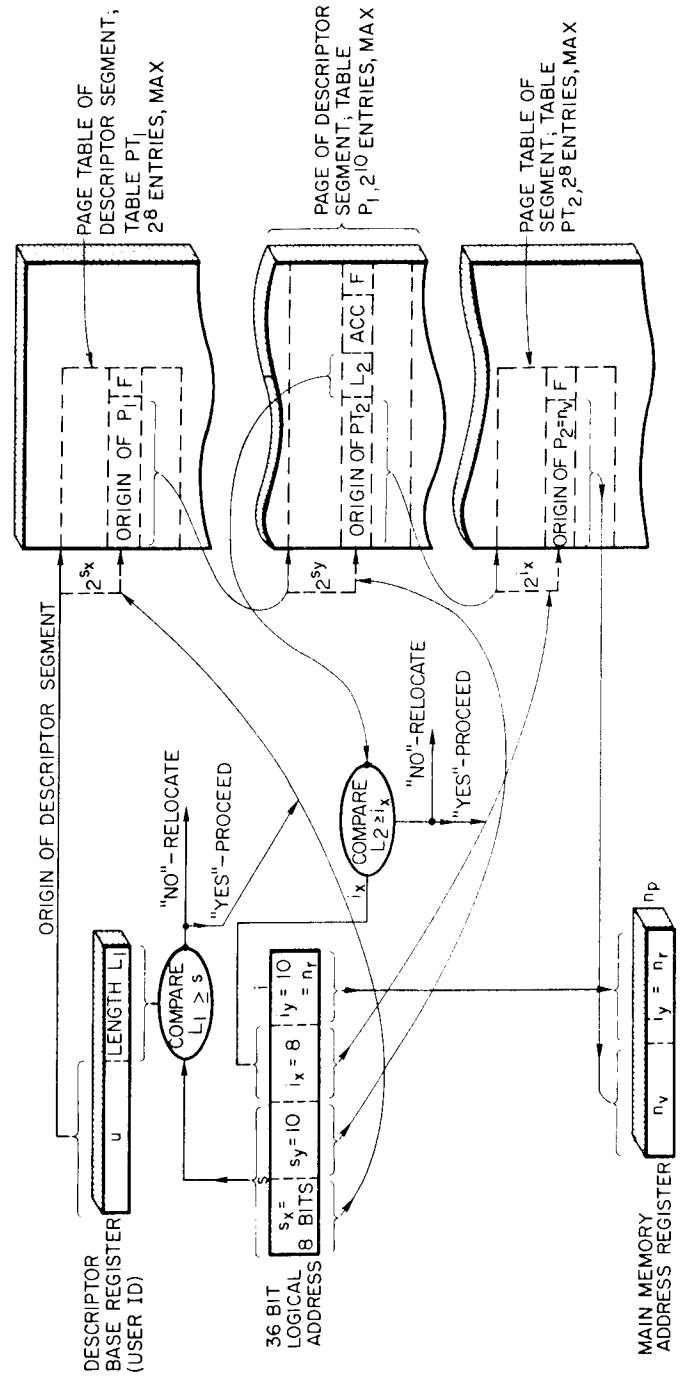

FIGURE 9.9-4 Honeywell Multics virtual memory showing address translation using three tables.

size (36 bits), of the logical address, this translation uses three tables; which are stored in main memory as needed. The address translation by way of these tables can be very slow, requiring three memory accesses plus a final access to the desired word. To speed up the translation process, a small, partial directory stores several of the most recently used pages just as in Fig. 9.8-3.

The implementation of Multics in the 6000 series machines works much as the 645 except two small partial directories are used (one to store several segment descriptor words and one to store several page table words); also the field formats of these table words have been changed somewhat for more efficient operation (MULTICS, 1972).

In the Multics system, the Table I required for keeping track of all segments on the secondary storage is maintained in a file hierarchy called *directories*. As with other similar systems, this directory hierarchy is implemented entirely in software under direction of the supervisory program.

The original Multics on the 645 supported two page sizes of 64 and 1024 words, the choice being left to the programmer. This reflected the historical concern of variable segmentation versus fixed page size. The 6000 series processors support only one page size (1024 words), although page size can be changed by field modification to any power of 2 from 64 to 4096 words.

9.9.4 Comparison of IBM and Multics Virtual Systems

We have already seen that the virtual memory address translation is nearly identical for the IBM and Multics virtual systems. The major difference comes about from the use of symbolic segment referencing in Multics, whereas IBM systems use linear segmentation. This basic difference gives rise to rather different ways of generating the logical address within the processor. The difference in address formation has some effect on the underlying virtual memory subsystems but only results in some practical differences in details such as the organization and amount of information stored in the various tables (essentially Table I and Table II), the manner in which these tables are searched, and the method of sharing and protecting segments.

In Multics, the symbolic segment names are converted by the supervisory system to segment numbers—namely, the address s—in a manner that prevents the programmer from knowing what value of s will be assigned. Hence two successively referenced symbolic segments do not get successive s numbers except in rare coincidences. Such a scheme provides complete generality of segment names within a procedure segment and is desirable from a logical point of view. However it can lead to practical problems. Suppose, for instance, that a large problem is being executed which requires more than one segment (i.e., more than 2^{18} pages). In particular, suppose two segments are required. An instruction in segment 1 may branch to a

word in segment 2 and then wish to come back. The programmer knows that in his program (virtual space) this required word is a certain number of memory references from the branch instruction. In logical memory space, however, since the branch must cross the segment boundary, the actual number of memory references cannot be known to the programmer beforehand. This is true because the linkage mechanism assigns the two segments noncontiguous locations in the descriptor segment table. Hence the programmer cannot use a simple indexing scheme to cross the segment boundaries but rather must use a more complex method of indirect addressing. In the IBM systems, during execution time, each user's segments are loaded contiguously in his separate segment table, ensuring that the boundaries between segments are contiguous: that is, the logical address of the last page of the first segment C is contiguous with the logical address of the first page of the second segment $C + 1$. Thus segment boundaries can be crossed with simple indexing.* It can be seen that this segment linkage difference is completely independent of the virtual memory subsystem. In fact, this difference comes about mainly from the method of compilation (or assembling) and loading of the source program. In the IBM systems, segments are assigned specific numbers and predetermined positions in the user's segment table. Consecutive segments are loaded into consecutive positions, hence are linearly related. In Multics a position in the segment table (descriptor segment) – that is, a value for s in Fig. 9.9-4—is not assigned until execution time; therefore the positions of segments in the table are not related in any way. These types of segmentation are known as linear (IBM) and symbolic (Multics).

In a nutshell we can say that the Multics and IBM virtual memory systems differ mainly in the way the general logical address is formed, with Multics maintaining symbolic segment representation and IBM using linear segmentation. The virtual address translation, mapping, paging, and replacement display practical rather than fundamental differences.

9.10 CACHE MEMORY SYSTEM DESIGN CONSIDERATIONS

A cache memory system represents a type of memory hierarchy that attempts to bridge the CPU–main memory speed gap by the use of a very small, high speed random access memory whose cost per bit is roughly 10 times that of main memory [see (1.3-1)] but whose total cost is relatively small because of the small size.

The cache–main memory hierarchy is really a virtual hierarchy of a limited variety, since the CPU now can reference stored data only through the cache.

* This continuity across segment boundaries exists only for the logical addresses. The real physical address is seldom continuous but is of no concern.

and main memory assumes the role of secondary storage. The only difference is that here the secondary store is directly addressable out of the main memory address register, whereas when secondary storage is a disk, a completely different accessing method must be initiated by way of the I/O processor and controller. Otherwise, the same concepts are used for both and the same fundamental problems arise. However the method of implementation is usually different because of the high speed demanded by the entire cache - main memory hierarchy. Thus this hierarchy usually has the addressing, paging, and other extra requirements implemented in special hardware, whereas a multiuser virtual memory is usually implemented by a combination of hardware and software features.

To understand the detailed organizational aspects of cache, it is essential to keep in mind that the primary requirement is speed, particularly the access and cycle time to a given page. Thus we have a demand for high speed both in the basic cycle time of the cache and in the address translation function. Speed is also of some concern in the page replacement function for moving pages out of and into the cache. One can imagine that the use of special logic control function similar to the I/O processor in Section 9.2 could be used to free the CPU during a page replacement. In other words, the time required for page replacement could be overlapped with CPU operation by cycle stealing during contention references or by available open periods for memory references due to the problem statistics (e.g., "multiply" may require 3 to 10 or more CPU cycles between memory references). Unfortunately, transferring control to another user's page requires a considerable amount of processing in itself, which can be greater than the time the CPU remains idle during a page transfer. This is true only when the speed difference between cache and main memory is not loo large, when a wide data path between the two can be achieved, or both. Section 9.4 indicated that a page size of about 64 bytes is adequate for cache. The data bus width can be 8 or 16 bytes per memory reference; if the latter, only four main memory cycle times are interleaved to achieve the page relocation. This can greatly speed up the system.

The remainder of this section explains how the need for speed influences the organization of the address translation. Before proceeding, the reader should be thoroughly familiar with the general operation of virtual memory of Section 9.3, especially the fundamentals of mapping and address translation of Sections 9.5 and 9.6. We make use of the fundamental system diagram of Fig. 9.3-2. The only difference is that secondary storage is now main memory, the cache being primary storage. We can use the same notations and definitions for address bits as previously.

As before, the lower order bits $N_r = n_r$ are real. The problem is to translate the higher order N_v bits to specify IF the desired page is present in the cache and if so WHERE. Both these should be done on one memory cycle if at

all possible. This eliminates the use of a table (Fig. 9.6-1) for translating the $n_d + s'$ bits of N_v, leaving only tag directory schemes. Again, because of the need for versatility in physical page location, Map 3 of Fig. 9.5-9 should be fully associative, suggesting the tag store schemes of Fig. 9.6-4. We assume a minimum directory with $e = n_v$. For Map 2 of Fig. 9.5-9, S compares are required, and since S depends on the mapping function, this is a major decision. Direct mapping with $S = 1$ or $s = 0$ is unworkable because of page slot contention problems. Fully associative mapping would have to be done with an associative memory, which is not only expensive but would be slower than a random access memory. Hence a set associative mapping seems more feasible. The value of S must be chosen such that a page or word access can be done in one cycle. This seems to be contradictory because S associative compares must be made simultaneously, which implies at least a partially associative memory. In other words, we must make S compares in one cycle. How can this be done without an associative directory? Non-associative memory requires that the compares be made external to the array. Simultaneous compares require that all S tags be available on one directory fetch. Together these requirements mean that a physical word must be composed of S logical words, each logical word with its own tag as in Fig. 9.10-1. Each physical word of the directory is thus composed of S logical words. Obviously while there are only 2^q physical words, there are still $2^{q+s} = 2^{n_v}$ logical words, as required in Fig. 9.6-5. A set is now specified by one physical word of the directory, and when one such word is selected by the q' address bit of the logical address, all S tags appear in the directory output buffer register. Since we do not know which if any of the logical words is correct, S compares are still required, as shown. These can be done simultaneously, and when a "yes" is obtained, the correct page origin address in cache is immediately available. Thus retrieval of the desired word requires one access to the page directory and one access to the cache itself. If no match gives a "yes" answer, the required page is not present and a relocate cycle is immediately initiated. This entire system is identical in principle to that of Fig. 9.6-4 except for the use of several logical words per physical word, which allows several simultaneous external compares. The compare functions can be implemented in high speed CPU register technology, hence introduce very small additional delay.

One might ask, Why not make the tag directory contain more entries? Why not let $e = N_v$, reducing the number of associatively compared bits to s (see Fig. 9.6-5)? This can be done in principle, but since primary storage only contains $2^q = Q$ set slots, there would be a considerable number of wasted entries in the table. Only 2^q entries could contain useful information at any one time, the others being essentially blank. Since directories are expensive, it is desirable to reduce the size as much as possible by removing

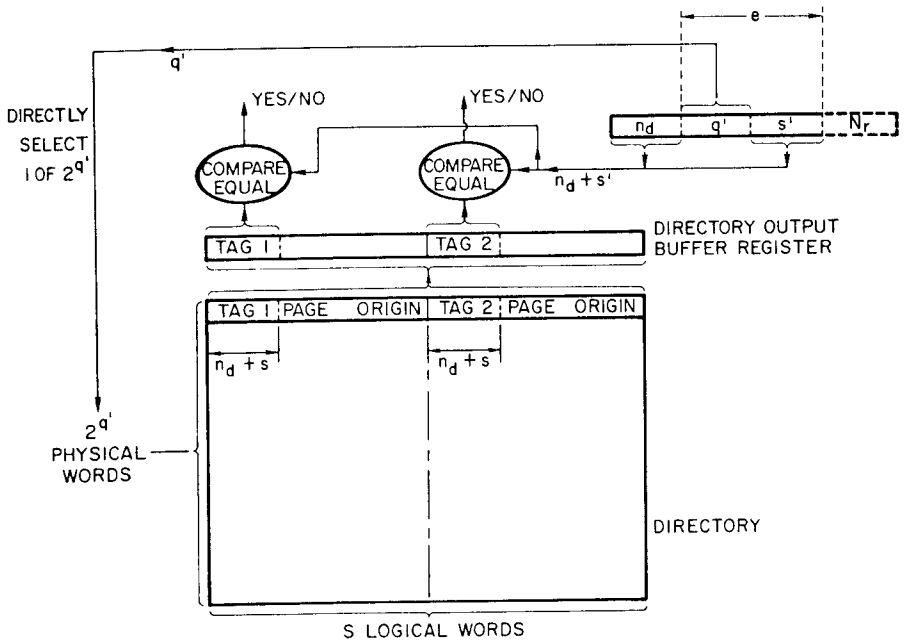

FIGURE 9.10-1 Tag directory decoding using multiple logical words per physical word for high speed with $e = n_t$ and set associative Map 2.

unnecessary, unused entries, which is achieved by reducing the value of e. For example, assuming $q' = n_d = 7$ bits (e.g., Fig. 9.11-2), if we use a directory with $e = N_r$, this requires $2^{14} = 16K'$ physical words. Each word must be capable of storing $2^5(s' + n_t)$ bits, but most entries will be empty at any one time. The scheme of Fig. 9.10-1 (or Fig. 9.11-2) with $e = n_t$ obviously is smaller because normally it will be full, representing only the information part of the 16K′ directory.

The need for speed in a cache hierarchy, coupled with only about one order of magnitude difference in speed and cost of cache versus main memory, raises two other important design considerations: "store-through" and "load-through." In store-through, whenever a change is made in a page residing in the cache, the same change is made in that page as it resides in main memory. Thus when a page is to be removed from cache, it can be simply erased instead of recopied into main memory. Since statistically only some bytes are changed, store-through can be more efficient and is often employed in practical systems. Load-through allows the data from main memory, as they are being paged into the cache, to be available simultaneously to the CPU without waiting for a full transfer and cache read cycle.

In essence, scratch-pad memories (Section 9.1) operate as a cachelike virtual memory system except the page is only one word long. The primary store or scratch pad is very much smaller than secondary storage, hence the addressing of the scratch pad requires both an IF and a WHERE function similar to those described in Section 9.3. As originally conceived, scratch pads were very small (in the range of 256 words), to introduce only small additional costs.

9.11 EXAMPLES OF CACHE MEMORY SYSTEMS

Early computing systems such as the Univac 1110 used fast scratch-pad memories to bridge the CPU–main memory speed gap as pointed out in Section 9.1. The first computer to incorporate a complete system using paged cache was the IBM 360/85.

9.11.1 IBM 360/85* Cache

To achieve flexibility and high hit ratios, associative mapping is desirable. However fully associative mapping requires a large number of associative compares. A cache must be accessed at high speed, necessitating an associative memory for address translation. Since this is expensive, the model 85 uses sector mapping as a compromise between these conflicting demands. Main memory and the cache are divided into sectors of 16 blocks/sector. Each block, which is almost equivalent to our previously defined page, consists of 64 bytes/block. Hence each sector consists of 1024 bytes (Fig. 9.11-1). Blocks or pages are mapped in sequential order within all sectors. Hence the location of any byte in a sector is fixed so that the 10 lower order address bits are real. In essence, the six lowest order bits specify one of 64 bytes within a block and the next four bits represent one of 16 blocks within that sector.

When a miss occurs requiring data transfer from main memory to cache, only the desired block is transferred. If all 16 blocks of a sector were transferred, additional, unnecessary delays and degradation in performance would result. The tag directory is a small associative memory built of CPU register technology for speed.

For a given logical address, the 14 high order bits uniquely specify the sector in main memory. Since there are only 16 sectors in the cache, these 14 bits must be decoded into an IF and WHERE function. This is accomplished with the tag directory in the following way. An associative compare is performed on all 16 tags of the directory. If no match occurs, a relocate

* There is no disk- main but only a main cache virtual memory. For a more complete description of this system and the cache, see *IBM* (1968).

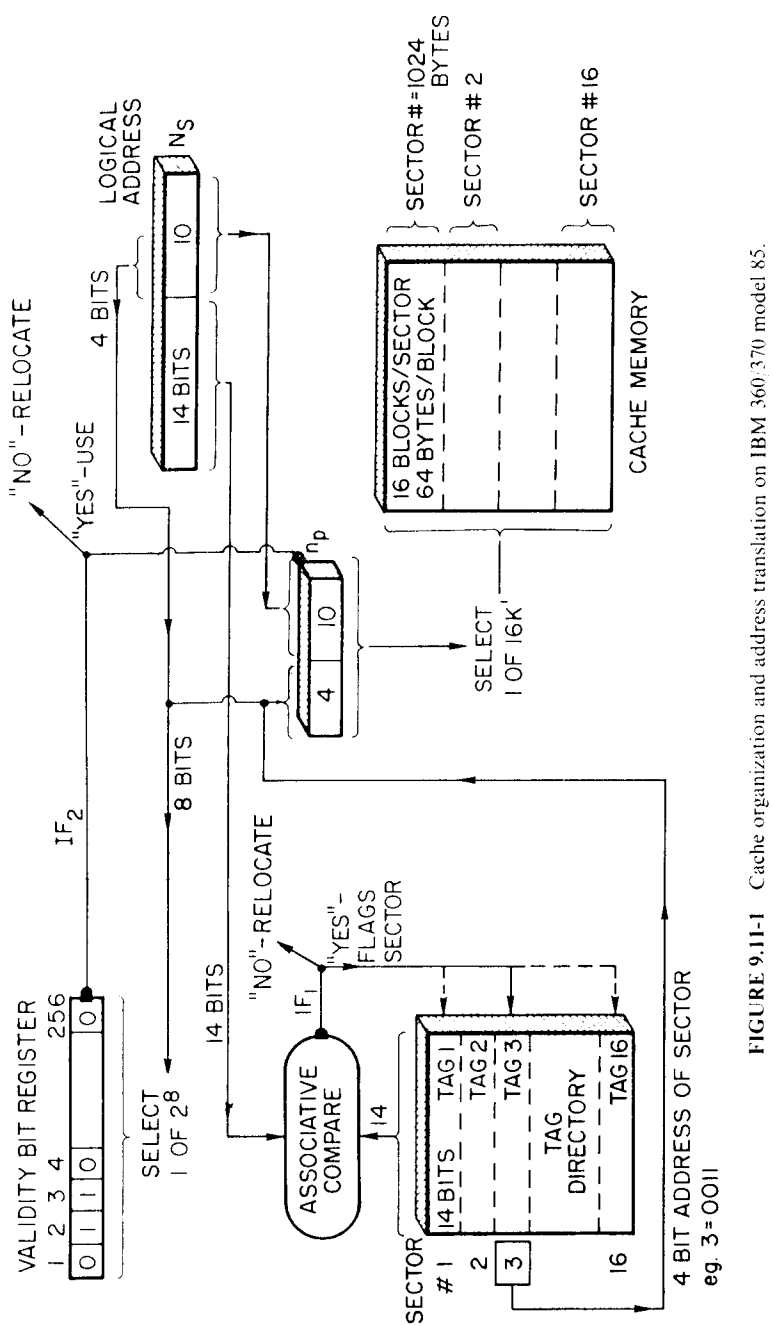

FIGURE 9.11-1 Cache organization and address translation on IBM 360/370 model 85.

cycle is initiated. If a match occurs, the IF determination is "yes" for that sector (but not necessarily for the required block). This represents the first part of the IF translation, indicated as IF_1 in Fig. 9.11-1. The number of the physical word of the directory for which the match occurs specifies the physical origin in cache of that sector. This number, in binary form, represents the four higher order address bits of the cache, the 10 lower order bits are real as shown. Since not all blocks of a sector need be present at one time, a further translation step is necessary to determine IF the required block is present. This is done by storing a "validity" bit for each block in cache, hence requires 256 such bits. Each validity bit, normally "0," is set to "1" when a block is placed in the cache and reset to "0" when a sector is reassigned. The remaining IF translation (IF_2 in Fig. 9.11-1) is then accomplished by testing the proper bit for the specified block: "1" = yes, "0" = no. Thus two processor cycles are required to fetch a given byte from the cache. The first cycle simultaneously searches the tag directory and the validity bits to determine IF and WHERE the data are in the cache. The second cycle fetches the data out of the address specified by the first cycle in combination with the real address bits. These validity bits are stored in CPU logic registers. Selection of the correct bit is done by using the four bits decoded from the tag directory plus the four higher order bits of the 10 real bits of N_s as in Fig. 9.11-1.

When a miss occurs, the least recently used sector is replaced. This is determined by a logical stack that maintains a logical ordering of sectors in terms of usage. When a sector is referenced, its number is moved to the top of the stack. Hence the bottom of the stack always specifies the least recently used sector. This logical stack, like the validity bits, is implemented entirely in CPU logic hardware. The model 85 cache uses both store-through and load-through to increase overall speed.

The bus width for transfer of data from main memory to cache is 16 bytes in parallel because the physical words of main memory are 8 bytes long, and two units are paired to give and effective word size of 16 bytes or 128 bits. Since a block is 64 bytes, four main memory words are required. Each of these is stored automatically in four separate memory modules, which can be separately accessed (i.e., four-way interleaved at time intervals equal to the cache time). Hence transfer time is approximately one main memory access time plus four cache times or about $0.880 + 4(0.080) = 1.2$ μsec.

Performance studies (Liptay, 1968) on 19 job streams of about 250,000 instructions each and simulated on the model 85 cache gave a hit ratio of from 92 to 99 %, with a mean of 96.8 %. Though adequate in many cases, the sector mapping did not provide the high hit ratios desired for such large systems. The difficulty is that when a miss occurs and the required block belongs to a sector not present in the cache, an entire sector must be removed.

This means that up to 16 blocks are removed, and even though being the least recently used sector, one of these 16 removed blocks has a high probability of being referenced subsequently. Nevertheless this system demonstrated the power and utility of the cache concept. The subsequent IBM models 360/195, 370/158, and 370/168 use set associative mapping for their cache–main memory hierarchies.

9.11.2 IBM System 360 Model 195 Cache*

This cache memory consists of a random access array using the organization of Fig. 4.9-5. The cycle time of the array itself is 54 nsec, which equals the basic processor clock cycle time. Pages are 64 bytes long, since this length is more optimum, as previously described. The cache capacity is 32K′ bytes, or 512 pages maximum. Main memory is 1M′ to 4M′ bytes of core storage with 756 nsec cycle time. A physical word consists of 8 bytes; thus one access to a physical word every 756 nsec produces 8 logical words or 8 bytes. The mapping function is set associative with four page slots per set.

As indicated in Section 9.10, two design factors are especially important. First, it is desirable to perform a memory reference with only one access time to the cache, plus a minimum of other delays for address translation and data transfer. Second, when a miss occurs, it is desirable to have a fast page transfer from main memory to cache, to minimize any possible subsequent delays. Considerable concurrency exists in this system by way of pipeline processing, as well as up to 10 instructions simultaneously in various stages of decoding and execution. Thus one cannot easily specify exactly what happens on, say, a miss, without analyzing an entire instruction stream, yet fast page transfer is desirable on an average basis. The first criterion results in the use of a tag directory and associative compares for address translation.

The second criterion is fulfilled because the main memory is at least eight-way interleaved,† so that each of the eight modules can be accessed separately. In this way the 8 physical words of 8 bytes each, which equals one page, can be transferred into the cache at the cycle time of the cache. If the main memory were not interleaved, 8 cycles of 756 nsec each would be required to transfer one page, an intolerably long time.

To allow fast address translation without consuming large amounts of additional storage for the translation, the address translation is done exactly as shown in Fig. 9.10-1 but with $S = 4$ pages/set, hence 4 logical words per physical word. The directory (Fig. 9.11-2) contains 128 physical words, with each word storing four tags of length $n_d + s'$ and the correct $s + q$ bits

* There is no disk main but only a main cache virtual memory.

† 2M′ and 4M′ byte versions are 16-way interleaved.

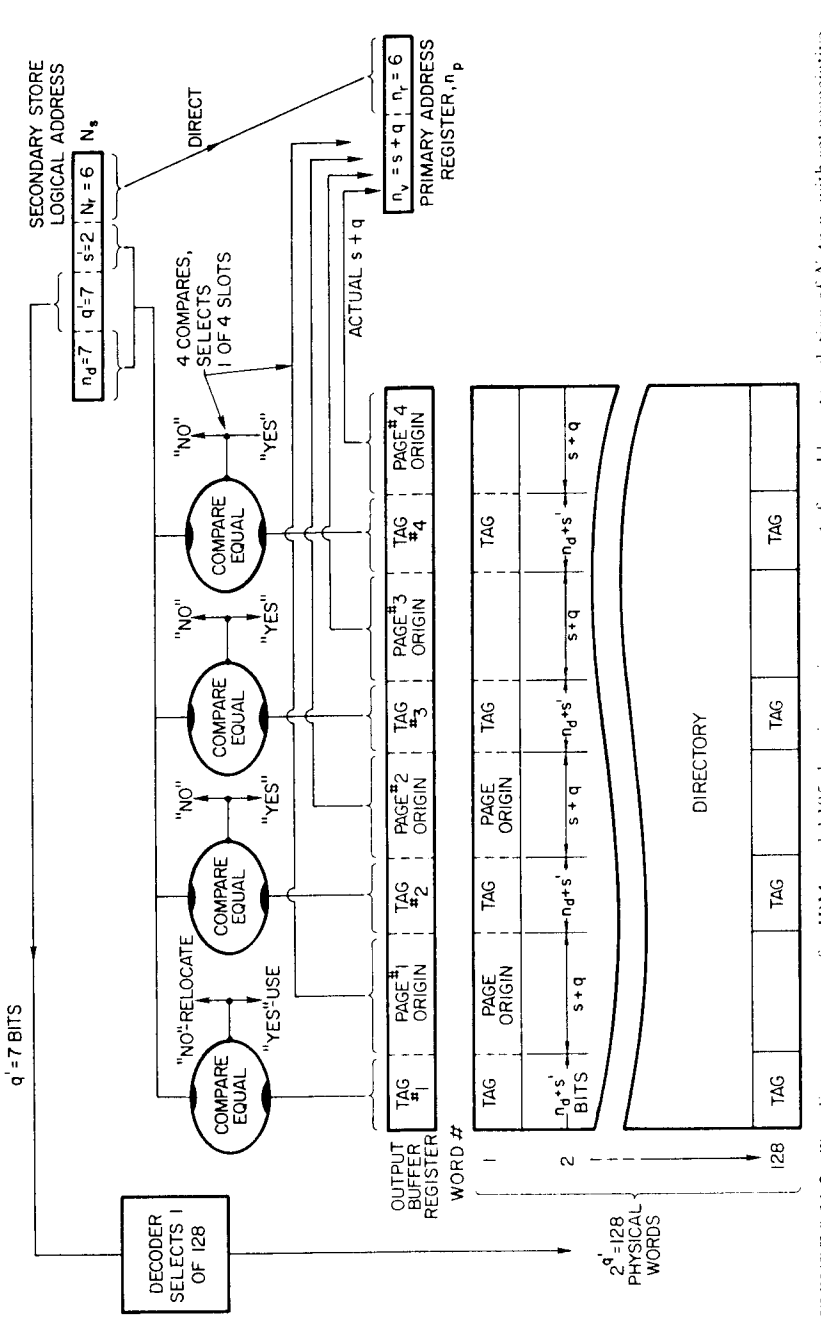

FIGURE 9.11-2 Tag directory system for IBM model 195 showing various components for address translation of N_s to n_p with set associative mapping.

associated with each tag. The virtual q' bits are used to select one of the 128 words, which is read into the output buffer register, yielding four possible locations. Four logical compares are simultaneously made of the $n_d + s'$ bits of the virtual address with each of the four tags. Only one of these can give an equal or "yes," which then gates the correct page location bits $s + q$ into the primary address register. Since the n_r bits are real, the translation is complete. If none of the four compares gives a "yes," a page transfer is initiated.

Although the cycle time of the cache array itself is 54 nsec and equals the processor clock cycle time, a full cycle of the system requires three such cycles for a total access time of 162 nsec. The first cycle gates the information from the processor to the directory, accesses the directory, and obtains the correct page address. The second cycle accesses the cache and produces the required word in the cache buffer register.* The third cycle gates the data to the proper section of the processor as required for processing. Hence the data became available for use by the processor only after three machine cycles. Since any page origin $(s + q)$ can be contained in the directory, pages can reside anywhere in the cache so Map 3′ of Fig. 9.5-10 is fully associative.

Even though the directory performs associative compares, it is not an associative memory in the strict sense. The directory is a high speed, random access memory with only one word accessed at a time. The associative logic is external to the array. It would be desirable to have a fully associative directory, which would allow fully associative mapping, since better overall paging efficiency could be obtained. Associative mapping requires a smaller cache for a given miss ratio objective than set associative mapping. Furthermore, the fully associative design is less sensitive to the particular job stream being processed, since it tends to optimize itself into the map most appropriate for the job. Fully associative design is more expensive, however, as well as slower for typical cache designs (Meade, 1971).

The replacement algorithm is the LRU procedure previously discussed. Since a page can reside in only one of four possible page slots, the least recently used page of each set must be tracked. In any design using LRU replacement, the order of all elements relative to one another in terms of usage must be stored. For four elements or four pages per set, there are four factorial or 24 possible combinations of relative usage. This requires 5 bits minimum plus an occupancy control bit, giving a 6 bit usage control function for each of the 128 sets. These are stored in special hardware with separate control and updating algorithms. Two other important features used in the model 195 to increase the overall speed are store-through and load-through. Store-through merely ensures that when a change is made in the cache, it is

* This register is contained in the buffer storage control unit.

simultaneously made in main memory. Thus when a miss occurs and the cache is full, the least recently used page within the proper set can be erased without having to write possible changes back into main memory. Load-through arranges the order of page data transfer when a miss occurs. It ensures that the referenced 8 byte word of the desired page is retrieved first and simultaneously provided to the processor and cache. The remaining bytes are then transferred, furnishing the necessary information to the processor with minimum delay.

The operation of the cache is hardware controlled by a storage control unit and is not program addressable. The user addresses main memory as usual, and the cache operation is transparent to the user. The user's logical address is equivalent to a main memory or in this case a secondary storage address. Table I, needed in Section 9.8 for keeping track of virtual pages on the disk or in this case in main memory, is unnecessary because this information is contained within the user's program.

Performance evaluation of the model 195 cache (Murphey and Wade, 1970) using 17 job segments indicates that the effective cycle time of the hierarchy is about 162 nsec, with occasional increases to 175 nsec. The 17 segment job stream contained a mixture of commercial processing and moderate amounts of decimal arithmetic, scientific, engineering, and systems-type processing (sorting, assembling, compile, link edit). The average hit ratio for the buffer during simulated processing of these 17 segments was 99.6%. Smaller hit ratios give quite adequate performance.

9.11.3 CDC 7600 Memory Hierarchy (Cyber 70 Model 76)

Although the operation of the memory hierarchy in the CDC 7600 system is quite different in detail from that of the cache concept, it serves the same purpose—to speed up the effective or apparent cycle time of a large, slow main memory with a smaller, faster memory. This hierarchy has certain aspects similar to the older scratch-pad concept and certain concepts similar to the cache, hence it could be classified as an alternative to both. This system is not a paged virtual system in the fundamental sense for several reasons. (*a*) The operation of the hierarchy is *not* transparent to the user, in fact must be programmed in detail (i.e., no automatic address translation nor relocation hardware). (*b*) The number of words that are block transferred between secondary and primary storage can vary at the discretion of the programmer (i.e., pages are not of fixed size). Because of these differences, greater involvement is required from the programmer, but more freedom is also provided for fine tuning the system to individual problems. This freedom is often useful in scientific problems for which this system was specifically designed.

The memory hierarchy within the central procesing unit consists basically of a large core memory (LCM), which is the secondary store in this hierarchy. This memory has a maximum capacity of 512K′ words of 64 bits/word.* The complete read/write cycle requires 64 CPU clock cycles of 27.5 nsec each (see Table 1.1-1) or 1.75 μsec. This memory is backed by a small core memory (SCM) with a maximum capacity of 64K′ words × 65 bits/word.† This smaller primary store has a read/write cycle time of 10 CPU clock periods or 0.275 μsec and an access time of 4 clock periods or 0.110 μsec. This is considerably slower than the CPU clock period but is compensated by interleaving of modules. The SCM is organized logically into a maximum of 32 independent modules or banks. Since the read/write cycle time is a maximum of 10 clock pulses, 10 banks can be in operation at one time. Under ideal conditions with data that can be perfectly interleaved, the effective cycle time is thus one-tenth that of any module; that is, the successively operating banks deliver one memory reference on successive clock pulses. Hence an effective access rate of 27.5 nsec is obtained under ideal conditions. In random addressing, far fewer banks, typically only four, are in operation, giving a longer effective cycle time of the primary store.‡

In operation, block transfers of data from the large secondary store to the small primary store require that the program specify the length of the block and the beginning address in both stores. Each of these three parameters is placed in separate CPU registers,§ thus acting as base registers for simplifying subsequent word addressing during programming. The 60 bit words are read and copied from consecutive addresses at the rate of one word per CPU clock period. All other activity is stopped during this data transfer except for I/O word requests, which can proceed simultaneously. Exactly the same circumstances exist when data are transferred in the reverse direction, from the smaller memory back to the larger store. Thus it is apparent that the operation of the primary store is not transparent to the user and in fact must be programmed in detail.

The architecture of this entire system is very different from that of previously described systems. The central processor is augmented by 10 peripheral processing units, each with its own memory of 4096 words × 12 bits (plus one parity bit) per word. These peripheral processors act both as I/O controllers and as local computers as described in Section 9.2. This architecture represents one approach to the problem of speeding up the entire

* This is a 60 bit word plus 4 parity bits.

† This is a 60 bit word plus 5 parity bits.

‡ This is quite different from the cache, which has a basic read/write cycle time equal to one CPU cycle time.

§ The total block length is obtained by adding an 18 bit field from the instruction for BLOCK COPY to the register containing the block length.

computational process. Increasing the effective speed of main memory is one important aspect of this larger problem.

9.12 VIRTUAL MEMORY SYSTEMS WITH CACHE: THREE-LEVEL HIERARCHY

The foregoing discussions and examples have dealt with systems that had either a disk–main memory *or* a cache-type of virtual store. In large multiprogrammed systems, it is often desirable to combine both features for better efficiency and versatility. Since the overall operation of such systems becomes very involved with the overall computer architecture, we consider only a few basic ideas pertinent to operation of the various storage levels. Assume a three-level hierarchy in which the main memory is paged out of a disk and the cache is paged out of main memory. To maintain consistency with previous definitions but to provide a distinction between various level, we define the various address bits as follows. Referring to Fig. 9.5-2*b*, we let n_m and n_c represent the total address register capacities of main and cache, respectively, with $n_m > n_c$; N_s and N remain as previously, but now we have

$$\text{main} \qquad n_m = n_{mv} + n_{mr} \tag{9.12a-1}$$

$$\text{cache} \qquad n_c = n_{cv} + n_{cr} \tag{9.12a-2}$$

where n_{mv} and n_{cv} are the page address bits and n_{mr} and n_{cr} are the real byte reference bits. Section 9.4 stated that page sizes for disk to main should be in the range of 1K′ to 4K′ bytes, whereas 32 to 64 bytes is more reasonable for cache.* So the real address bits using the nomenclature just given would be

$$N_r = n_{mr} \sim 10 \text{ to } 12 \text{ bits}$$

$$n_{mr} > n_{cr} \sim 5 \text{ to } 6 \text{ bits}$$

In principle this presents no problem and need not change the translation scheme. We are normally given a total logical address $N_s = u + N$, and ideally this byte should reside in the cache as well as main memory. Thus it is necessary only to translate all the N_s bits into n_c with an IF and WHERE part. This can be the same as that described in Sections 9.10 and 9.11 for translation from main to cache, except for some practical differences. First, N_s is so much larger than n_c that the directory becomes large and expensive. The major question is, Can it be simplified?

* In IBM manuals cache pages are called "blocks." Also, main memory page slots are called "frames"; the term "page slot" refers only to those on disk.

Before answering this, let us analyze the operation of the three-level hierarchy more closely. If we decode N_s into n_c by the use of a directory, ideally the IF part of the translation will be "yes" (i.e., a hit). However, when a "miss" occurs it is necessary to determine IF and WHERE this N_s address resides in main memory. Since the processing *cannot transfer to a new user on a miss to the cache, it is most desirable to initiate* the address translation of N_s to main memory *simultaneously* with that to the cache. A hit to the cache can then abort the main memory decoding, and a miss will provide faster fetching from main. So far, this requires nothing different from the previous separate pieces of the hierarchy. The translation of the total logical address to main memory physical address can be done as in Section 9.8 or 9.9, using a Table II assisted by a small partial directory. The simultaneous translation of the same logical address to the cache physical address can be done with a full directory as in Section 9.10 or 9.11. Since such a three-level hierarchy is useful only on rather large systems, both the partial and full directories begin to require large amounts of (expensive) storage capacity. Thus ways to reduce these amounts are desirable and are possible, since the two stages of translation of N_s to n_m and n_c have in common certain elements that can be used to advantage. As a result, the address translation for such three-level hierarchies appears to be more complex, though in principle it is simpler. We shall try to show this, then relate to an example using a large system, the IBM 370/168 virtual memory system with a cache. Theoretically, the partial directory could be expanded to perform a full translation of N_v into the real n_{mr} with no change in the three-level translation scheme.

Let us continue with the idea of the two stages of decoding to main and cache simultaneously, using a partial directory to be called a translation lookaside buffer (TLB) for the former and a full directory to be called buffer address array (BAA) for the latter. The former TLB must decode the N_v page address to give an IF equal "yes" or "no," and a WHERE, in terms of the real main page address bits n_{mr}. The BAA must take $N_v + N_r - n_{cr}$ bits and convert these into n_{cr} bits for WHERE and a yes/no for IF. Note from Figs. 9.6-2 and 9.6-5 that the number of bits that must be stored and associatively compared in the minimum tag store directory increases with the size of N_v. But we really need not separately translate $N_v + N_r - n_{cr}$ bits in the BAA for the following reason. If a portion of a page is in the cache, it must be somewhere in main memory, with an entry either in the TLB or in Table II. *Such being the case, it is necessary only to translate the real value of n_{mr}, which is already available from the TLB or Table II, into the real cache address.* This is the first fundamental principle of such a three-level hierarchy, and it greatly reduces the number of bits that must be associatively compared in the BAA.

Another general principle, mentioned now and described in the example later, is as follows. There are different page sizes in main memory and cache; however even though the cache pages, which are called *blocks* or *lines*, must be in contiguous memory locations in main memory, they may be set associatively mapped into cache. This permits a simple translation of the bits $N_r - n_{cr}$.

We now examine the IBM 370/168, primarily to see how the address translation is implemented in terms of the fundamentals presented in this chapter.* No attempt is made to be complete: rather simplification of such systems into fundamental principles can greatly assist in achieving an overall understanding. As always, there are two general, fundamental problems in address translation that cannot be avoided but only designed around. First, as the size and versatility of the virtual store increases, maintaining overall efficiency tends to increase the number of associative compares and the number of bits that are associatively compared; second, these increases make the translation slower and/or more expensive. Hence the general design approach is to minimize both factors. These problems should be kept in mind in the following example.

IBM 370/168

For a large multiprogrammed system with a fully associative disk–main memory mapping function, it is desirable to make the partial directory, which assists Table II, larger than the eight entries of Fig. 9.9-2, to improve the hit ratio and to accommodate several user's pages. However if a full directory were used, the large number of associative compares of $2^{n_{mr}}$, and the large number of bits compared associatively (namely, $N_v - n_{mr}$) would make this impractical. The model 168 circumvents these two problems while providing an effectively large partial directory (TLB) in the following way. The large value of $N_v - n_{mr}$ (i.e., n_d or $N_s - n_m$) is reduced *inside the CPU* by reducing the large user ID address to a 3 bit STO ID. The large number of associative compares is reduced by combining the N'_v bits with the STO ID, using a hash decoding that produces an address to directly address the TLB directory for WHERE and requiring only one associative compare (on four fields) for IF.

More specifically, the principles of the model 168 are as follows. The total size of N_v, which equals $u + N'_v$ bits, is rather large, representing many *possible* logical pages for many users. Of this possible size, only a small fraction will ever be used at any one time by the current users resident in the system. Thus it would seem desirable and expedient to reduce the potentially large size of N_v to a more manageable size. The model 168 accomplishes this in two steps. In the first step the logical user ID, consisting of u bits

* Additional details can be obtained from manuals [IBM].

representing millions of potential users ($u \geq 24$ bits), is reduced to a more manageable size. The full user logical IDs of only the six CPU-active users are stored in a STO address array consisting of six registers of u bits each. Associated with each register is a 3 bit STO ID. Thus the u bits are reduced to 3 bits represented by one of the addresses of the STO array as in Fig. 9.12-1. When a transfer to a second user is performed, an associative search of the array produces the 3 bit active STO ID of that user, if a match occurs. If no match occurs, the new user is entered, assigned one of the six possible STO ID values, and the TLB directory must be cleared of all such IDs that referred to the removed user. In the second step, the active STO ID is combined with the higher order N'_v bits of the CPU logical address in a hash decoder to produce a smaller address to directly select the TLB as shown. The hashed address need not be unique either for the current user nor among various users. In other words, the nature of hash decoding can produce the same TLB entry address for different users, or for different logical pages of the same user.

This nonunique hash address problem is circumvented by the use of a one-step associative compare as follows. The direct selection of one of 64 TLB entries yields six data fields; two fields are different STO ID bits (plus storage protect keys) for two different users who happen to hash code to the same entry, or alternatively are the same STO ID. Similarly, two fields contain the same, or alternatively different logical addresses $N - n_{mr}$ that happen also to code to the same entry. An associative comparison on these four fields is required to select one of the correct "real n_{mr}" address bits, very much like the directory scheme of Fig. 9.11-2. In this case, however, a "yes" match must be obtained simultaneously on both the STO ID and the corresponding logical page address, to permit use of one of the real n_{mr} addresses.

In this implementation the TLB allows only for two-way redundancy in the hash coding, which usually is sufficient. If greater redundancy occurs by chance in the hash decoding, no match is obtained in some cases and the translation must proceed by way of Table II. Incidentally, since hash decoder is also used to initially store a user's entry in the TLB, once established, a given logical address for that user will always decode to the same TLB entry. The primary purpose of this hashing procedure is to randomize virtual page allocations within the TLB so fundamentally, hashing is not necessary. The actual hash decoder is relatively simple. The large CPU virtual address and STO ID bit positions are merely arranged into six columns and added. No carry is used between columns but only the absolute value, giving the 6 bit TLB entry address. Obviously the above translation scheme has reduced the number of bits to be compared associatively through the use of the STO stack array and has reduced the number of associative compares by the use of hash decoding and providing only for two-way redundancy. This TLB and

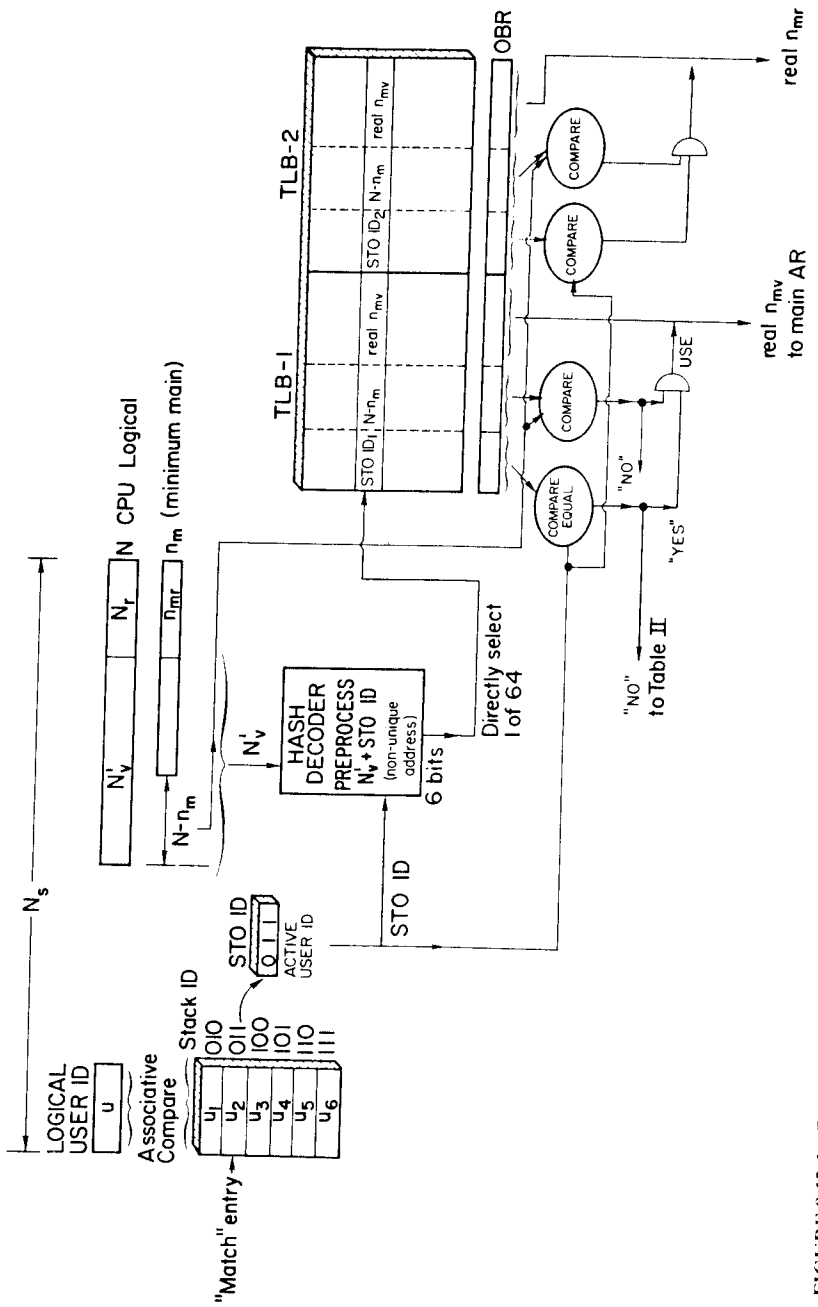

FIGURE 9.12-1 Translation of virtual address to real main memory address via hash decoder and partial directory (TLB) on IBM 370/168.

associated logic has many features in common with the directories of Figs. 9.6-4*b* and 9.11-2. However it should be clearly understood that the TLB holds only a small percentage of the more recently used pages of the current active users similar to the model 67 directory of Fig. 9.9-2. As such, it does *not* perform the complete address translation for main memory. A Table II similar to that of Fig. 9.6-1 performs the complete address translation and serves as backup to the TLB. On the other hand, the cache directory, described below, must perform a complete address translation, which it does in a manner similar to that illustrated in Figs. 9.6-4*b* and 9.11-2.

The cache address decoding is initiated at the same time as the decoding of the main address. In effect, we must decode all of N_s into n_c. However the first fundamental principle of a three-level hierarchy specifies that the cache may use the real n_{mr} address bits instead of N_r. Since these bits must be produced anyway, they might just as well be used to simplify the cache translation function.

The decoding of the cache is done with a tag store that is identical in principle to that of Figs. 9.10-1 and 9.11-2 except for some minor changes in detail. We now show how all the total logical address bits N_s are decoded in conjunction with a tag store directory or BAA to yield the cache address n_c. The cache blocks (pages) are 32 bytes, giving real bits of $n_{cr} = 5$; the mapping is eight-way set associative, thus $s' = 3$ bits. Referring to Fig. 9.12-2, the five lower order bits of N_r are real in the cache, hence convert directly to n_{cr} as shown. The remaining six higher order bits of N_r, labeled q', are real in main memory, hence must be in contiguous addresses there. In the cache storage array, however, the corresponding blocks can belong to different virtual pages. Thus these six q' bits may also be real in the cache address, but the actual page to which each address belongs must be determined. Hence these q' bits go directly into the cache address register, but they also must be used in decoding the s' bits by selecting one of $2^{q'}$ in the BAA. The s' bits of the cache logical address plus the deficient bits are decoded exactly as in Fig. 9.11-2 except that these bits are first converted into the real main virtual page address n_{mr} as in Fig. 9.12-1. Then these bits are used to do the eight-way associative compares on the real $n_d + s'$ bits in the BAA as shown. Since the real bits of the main memory address are used to select the set in the BAA directory, as well as for the associative compare, the mapping goes from physical main to logical cache as in Fig. 9.5-10 rather than from logical main to logical cache.

A simpler way to understand this translation is to merely think of the entire process as being identical to that of Fig. 9.11-2, but with two minor changes. First, the $n_d + s'$ bits are preprocessed because of the interaction with main memory. Second, rather than making Map 3′ of Fig. 9.5-10 fully associative, which requires $s + q$ bits, we may make it *set associative* with

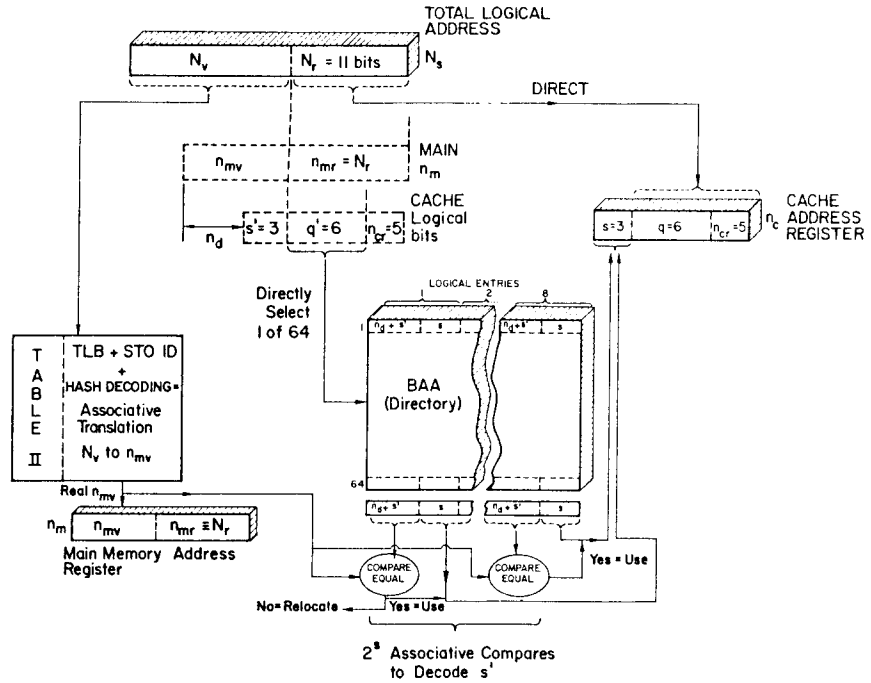

FIGURE 9.12-2 Translation of real main memory address to real cache address via a directory (BAA) on IBM 370 168.

no difficulties. Hence we do *not* store q bits within the page origin of the directory as is done in Fig. 9.11-2, but rather only the s bits; the q bits are obtained directly from N_s. Hence the model 168 cache uses a tag store directory with set associative Maps 2^s and 3 and real main memory address bits for the associative compare bits $n_d + s'$.

Note that in addressing the cache, a minimum of two cycles is required as previously (Fig. 9.11-2). The first cycle interrogates the TLB and BAA simultaneously to obtain the real cache s bits; the second cycle then accesses the cache itself. If only read operations were required from the cache, it would be possible to do this in approximately one cycle by accessing a full set in the cache (i.e., all possible s bit locations) simultaneously with the interrogation of the directory. The directory would then specify which, if any, of 2^s cache pages is correct. The latter logical operation can be done very fast; hence the entire access is performed in one cache cycle. For writing, however, we must first determine the correct s bits, then write into the cache on a second cycle.

Rather than having two different accessing schemes, two cycles are used for all accesses.

In principle, the three-level hierarchy translation of the Amdahl 470 V/6 works as just described, although there are some differences in details.

9.13 ASSOCIATIVE MEMORY APPLICATIONS IN VIRTUAL MEMORY HIERARCHIES

It was pointed out in Section 9.5 that fully associative mapping of the blocks between primary and secondary storage provides a system that is less sensitive to the particular job stream being run than any of the other mapping functions. Fully associative mapping is used in commercial virtual memory systems, but the address translation is done with tables because of the high cost of associative type hardware. However the table scheme is slow, and cache systems use set associative mapping as a compromise between cost and speed. In this section we assume that fully associative mapping is used, and we investigate the various applications of an associative memory in the address translation function. The complexity and speed of associative memory, which determine the cost, are discussed in Chapter 4 and are not considered here.

Section 9.6 presented the fundamental ways in which address translation could be carried out, and these should be understood before proceeding. Referring to Fig. 9.3-1, when pages of fixed size are used, the maximum number of bits that require decoding is $N_v = N_s - N_r$, where N_r are the real bits. Since $n_r = N_r$, primary storage contains 2^{n_r} directly addressed words within each page, but each of the 2^{n_v} pages must be indirectly addressed (e.g., with an associative memory). It is easily deduced from this that we would not use a totally associative memory for primary unless n_r were very small (e.g., 0 or 1). However n_r is generally from 6 to 12 bits; thus an associative primary memory would be uneconomical and slow. In other words, the n_r bits need not be decoded associatively, and doing this becomes increasingly wasteful as n_r increases. Thus the first general conclusion is that *it is fundamentally unnecessary to make primary storage entirely associative.*

The next question is, Where and how might associative memory be useful? There are two basic choices: use a fully associative directory or use a hybrid associative primary memory. The former is no different from the directories previously discussed, except with $s = n_v$, 2^{n_v} associative compares are required for each page reference. In addition, when a match is found, the real s bits must be transferred to the primary address register for a subsequent reference to the desired page (Fig. 9.13-1). The storing and transfer of the s bits privides a fully associative Map 3 (Fig. 9.5-9) as discussed in Section 9.6. Recall, however, that when Map 2 is fully associative, Map 3 can be a direct

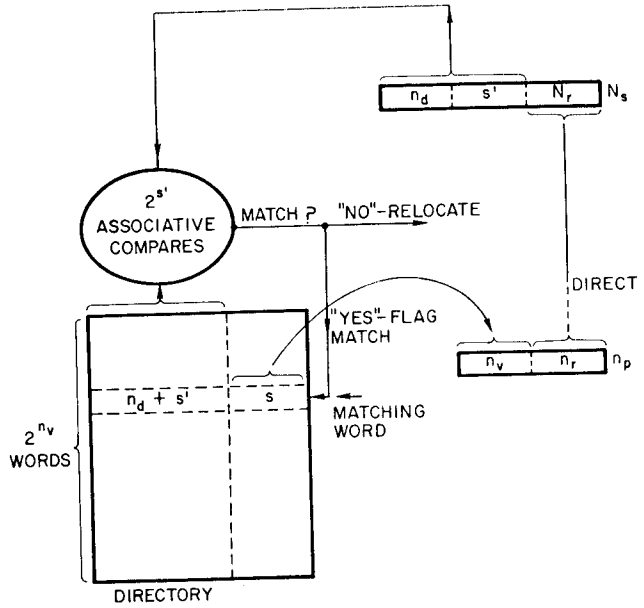

FIGURE 9.13-1 Fully associative directory using $n_d + 2s$ bits word; $s = n_v$.

mapping and still provide the appearance of a fully associative map. The directory can have hard-wired direct enable lines to primary pages in place of transferring the s bits, thus eliminating the second access. This is one of the principles used in the hybrid primary store. Rather than fabricating a separate directory, the associative functions can be placed close to the proper pages, as in Fig. 9.13-2. Large-scale integration allows either of these schemes to be implemented with a number of possible configuration. Let us consider some of the advantages and disadvantages of each approach. Assume for simplicity that fully decoded $2\frac{1}{2}$D semiconductor chips, similar to those of Fig. 4.9-4, are available, and each chip contains a full page of 2^{n_r} bytes or words. The real address bits must then be broken into two parts n_{r1} and n_{r2} as in Fig. 9.13-2—one part to select one of $2^{n_{r1}}$ word lines and the other part to select one of $2^{n_{r2}}$ segments of bit/sense lines, each segment of b bits as in Fig. 4.9-4. As detailed in Section 4.9, these address bits are paralleled to all chips, hence require an additional chip selection, which is provided by the ENABLE input to the chip decoder as shown. The ENABLE input permits a very simple implementation of the associative translation function. Along with the other devices, each chip has a separate register fabricated that performs a single compare on all N_r bits, and this compare is done simultaneously on all chips. Only one chip can have a match, and this match

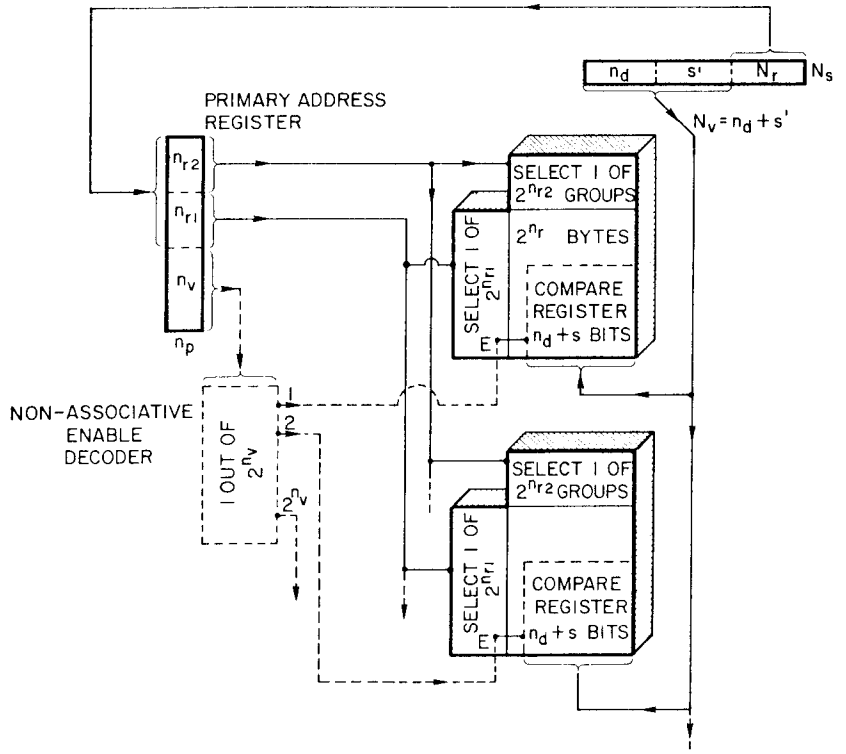

FIGURE 9.13-2 Hybrid primary store with on-chip associative address translation.

produces an immediate ENABLE signal on the chip decoder as shown. It is *not* necessary in this case to first provide the s bits in the primary address register for ENABLE decoding, as would be required in the scheme of Fig. 9.13-1, since this is actually done directly. Of course nonassociative decoding of the n_r or s bits in the primary address register would still be done by a separate decoder and a separate input to the same ENABLE function on each chip* (dashed lines in Fig. 9.13-2). Hence only one access to the primary store is needed, and if the on-chip compare register is fast, the entire translation could be fast. The decoding of the real n_r bits for the word and bit/sense line of each chip can be performed simultaneously with the associative decoding of the N_v chip selection bits. One serious disadvantage of this on-chip hybrid scheme is that an additional N_v pin connections must be made to

* The two inputs to the ENABLE terminal would be ORed together.

every chip.* The identical scheme can be implemented with off-chip associative compares and direct Enable signals to each page, and only one additional pin per chip is required.

The direct page ENABLE signal necessary in the above scheme can be obtained from an associative memory array that provides $\bar{F}$ type flags as detailed in Section 4.12. A simple inverter NOR gate with $\bar{F}$ as input for each page would provide the ENABLE signal immediately. Note that if Map 3 were fully associative as in Fig. 9.13-1, rather than direct with a direct ENABLE as previously, an associative memory array such as that described in Section 4.12, which performs the compare functions internal to the array, would require two cycles to fetch the real n_v page address. The first cycle would provide the match flag and the second cycle would read the match word to obtain n_v. This second cycle can be avoided only by doing the compare functions external to the array as in Figs. 9.10-1 and 9.11-2. This still necessitates a subsequent decoding of the n_v bits in the primary address register, and both requirements are avoid by the direct ENABLE implementation for Map 3 as in Fig. 9.13-2.

The page usage and control information required for each page could also be included in the on-chip associative register, which uses the same direct page ENABLE line. Such a scheme would have considerable advantages.

9.14 POTENTIAL PITFALLS IN PROGRAMMING VIRTUAL MEMORY SYSTEMS

Even though virtual systems can relieve the user of much detailed programming, there are potential pitfalls which can sometimes seriously degrade the problem solution. We wish to show by example that the programming is not always optimized, and in some cases the user must pay close attention to data organization. Generally the key problem is to minimize the number of page swaps—in other words, to maximize the hit ratio required for problem solution. The following case demonstrates that the number of page swaps can depend critically on the organization of arrays and the method of iterating or looping the array elements.

Let us assume that an engineering problem to be solved involves three-dimensional arrays. A virtual memory system is available having the following characteristics.

* We would also need some method of signaling that no match has occurred on any chip to initiate a relocate cycle. Conceivably this could be done on the existing sense lines, but an additional pin might be needed.

1. Demand paging. "LRU" page replacement algorithm.
2. Page size = 2048 words, 40 bits/word.
3. Maximum allotment of 8 pages of physical main memory per user (i.e., 16,384 words/user).
4. Virtual memory on disk (any size).

The problem requires evaluation of the function.

$$F_2 = \sum_{\text{all } I,J,K} (C_1 A_{IJK} + C_2 A_{IJK}^2)$$

The 3D array A_{IJK} is organized as follows. Each element A_{IJK} occupies one word; I varies from 1 to 2048, J varies from 1 to 4, and K from 1 to 2. The pages are made up of segmented listings of the array elements, starting page 1 with $A_{111}, A_{211}, A_{311}, \ldots, A_{2048,1,1}$; page 2 with $A_{112}, A_{212}, A_{312}$, and so on, through page 8, with $A_{142}, A_{2,4,2}, \ldots, A_{2048,4,2}$. The pages may be placed in any manner on one disk surface.

Let us first do this problem by looping or iterating on subscript K with I and J constant, then J, and finally I. A Fortran program that would accomplish this is as follows:

```
      DIMENSIONS A(2048, 4, 2).
      COMMENT—ARRAY A OCCUPIES 8 PAGES.
      F1 = 0.
      DO 2 I = 1, 2048.
      DO 2 J = 1, 4.
      DO 2 K = 1, 2.
1     F1 = C1 × A(IJK) + C2 × (A(IJK))^2.
2     F2 = F2 + F1.
      COMMENT—F2 SUMS F1 OVER ALL ARRAY ELEMENTS.
      PRINT F2.
      COMMENT—PROGRAM AND ANSWER REQUIRES 1000
      WORDS MAXIMUM.
```

Note. In this Fortran program, K loop is done first, then J, then I.

Let us now determine the number of page swaps executed. We will see that considerable page swapping is required because of poor data organization. Significant improvement will be seen to be possible by simply rearranging the order of I, J, and K within the DO loops, or by reorganizing the arrays.

One page is required for the program and work space to hold constants and intermediate results in the calculations. Thus there are only seven pages available to hold alphanumeric data. Page numbers are determined by the JK subscripts only since each page contains all I from 1 to 2048 words. The page numbers and corresponding J K subscripts are listed in Table 9.14-1.

TABLE 9.14-1 Page Numbers and Corresponding Subscript Designations

Page Number	J	K
1	1	1
2	1	2
3	2	1
4	2	2
5	3	1
6	3	2
7	4	1
8	4	2

The program first increments K from 1 to 2 with $J = 1$ and $I = 1$, which requires pages 1 and 2. Next J is incremented progressively from 1 to 4 requiring pages 3, 4, 5, 6, 7, and 8. Since only seven pages can be present at one time, and these are assumed to be already loaded, page 1, the least recently used, is removed and page 8 is transferred in. Next I is incremented to 2 and the interactions on J and K are repeated. This requires pages through 8 successively. But page 1 is out and page 2 is the least recently used. Thus page 2 is replaced by page 1. But 2 is needed next, and 3 is now the least recently used. Therefore another swap of 3 with 2 takes place. The process continues, requiring a total of seven page swaps. As the end, the pages are back in the original order, and when I is incremented to 3, only one page swap is required. On $I = 4$, however, again seven page swaps are required. Hence when I is an even number, seven page swaps are required, whereas when I is an odd number ,only one swap is needed. Thus a total of $1024 + (7 \times 1024) = 8K'$ swaps are needed, a rather formidable amount.

The situations can be greatly improved simply by reversing the order of the DO loops to

```
DO 2 K = 1, 2
DO 2 J = 1, 4
DO 2 I = 1, 2048
```

Now all 2048 interations on a total page are performed before incrementing to a new page. After the seventh page is complete, page 8 must be swapped in as before, but only one page swap is required for the entire program.

Clearly the ordering of page references can be very important. Of course the problem could be eliminated by allotting more pages per user, but this is not always possible, and the programmer must be aware of these limitations.