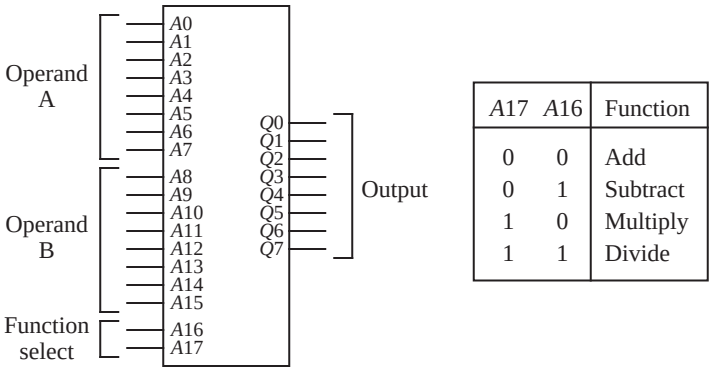


A17	A16	Function
0	0	Add
0	1	Subtract
1	0	Multiply
1	1	Divide

Figure 7-11 A lookup table (LUT) implements an eight-bit ALU.

32-bit operands are standard on computers today, and a corresponding PROM ALU would require $2^{32} \times 2^{32} \times 2^2 = 2^{66}$ words which is prohibitively large.

7.6 Cache Memory

When a program executes on a computer, most of the memory references are made to a small number of locations. Typically, 90% of the execution time of a program is spent in just 10% of the code. This property is known as the **locality principle**. When a program references a memory location, it is likely to reference that same memory location again soon, which is known as **temporal locality**. Similarly, there is **spatial locality**, in which a memory location that is near a recently referenced location is more likely to be referenced than a memory location that is farther away. Temporal locality arises because programs spend much of their time in iteration or in recursion, and thus the same section of code is visited a disproportionately large number of times. Spatial locality arises because data tends to be stored in contiguous locations. Although 10% of the code accounts for the bulk of memory references, accesses within the 10% tend to be clustered. Thus, for a given interval of time, most of memory accesses come from an even smaller set of locations than 10% of a program's size.

Memory access is generally slow when compared with the speed of the central processing unit (CPU), and so the memory poses a significant bottleneck in computer performance. Since most memory references come from a small set of locations, the locality principle can be exploited in order to improve performance. A small but fast **cache memory**, in which the contents of the most commonly accessed locations are maintained, can be placed between the main memory and the CPU. When a program executes, the cache memory is searched first, and the referenced word is accessed in the cache if the word is present. If the

referenced word is not in the cache, then a free location is created in the cache and the referenced word is brought into the cache from the main memory. The word is then accessed in the cache. Although this process takes longer than accessing main memory directly, the overall performance can be improved if a high proportion of memory accesses are satisfied by the cache.

Modern memory systems may have several levels of cache, referred to as Level 1 (L1), Level 2 (L2), and even, in some cases, Level 3 (L3). In most instances the L1 cache is implemented right on the CPU chip. Both the Intel Pentium and the IBM-Motorola PowerPC G3 processors have 32 Kbytes of L1 cache on the CPU chip.

A cache memory is faster than main memory for a number of reasons. Faster electronics can be used, which also results in a greater expense in terms of money, size, and power requirements. Since the cache is small, this increase in cost is relatively small. A cache memory has fewer locations than a main memory, and as a result it has a shallow decoding tree, which reduces the access time. The cache is placed both physically closer and logically closer to the CPU than the main memory, and this placement avoids communication delays over a **shared bus**.

A typical situation is shown in Figure 7-12. A simple computer without a cache

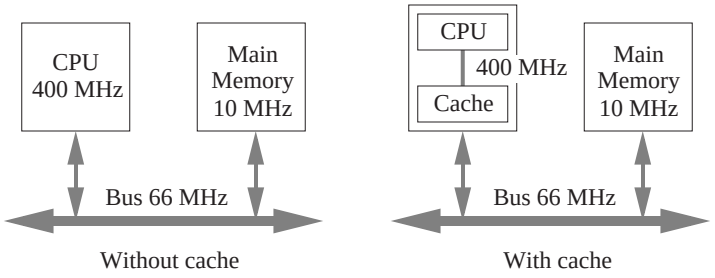


Figure 7-12 Placement of cache in a computer system.

memory is shown in the left side of the figure. This cache-less computer contains a CPU that has a clock speed of 400 MHz, but communicates over a 66 MHz bus to a main memory that supports a lower clock speed of 10 MHz. A few bus cycles are normally needed to synchronize the CPU with the bus, and thus the difference in speed between main memory and the CPU can be as large as a factor of ten or more. A cache memory can be positioned closer to the CPU as shown in the right side of Figure 7-12, so that the CPU sees fast accesses over a 400 MHz direct path to the cache.

7.6.1 *ASSOCIATIVE MAPPED CACHE*

A number of hardware schemes have been developed for translating main memory addresses to cache memory addresses. The user does not need to know about the address translation, which has the advantage that cache memory enhancements can be introduced into a computer without a corresponding need for modifying application software.

The choice of cache mapping scheme affects cost and performance, and there is no single best method that is appropriate for all situations. In this section, an **associative** mapping scheme is studied. Figure 7-13 shows an associative map-

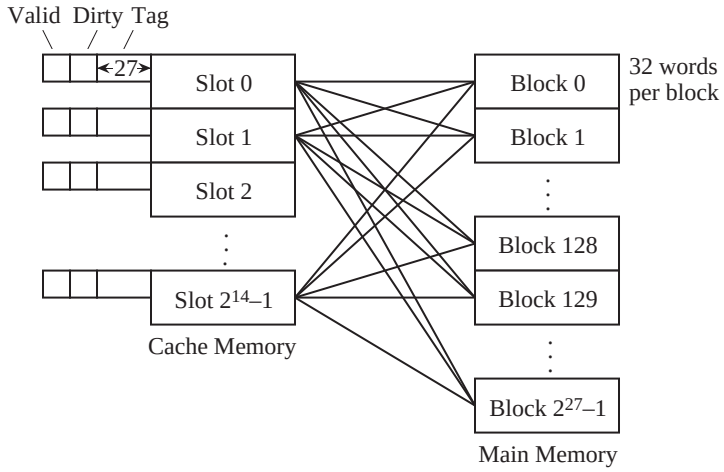


Figure 7-13 An associative mapping scheme for a cache memory.

ping scheme for a 2^{32} word memory space that is divided into 2^{27} **blocks** of 2^5 = 32 words per block. The main memory is not physically partitioned in this way, but this is the view of main memory that the cache sees. Cache blocks, or **cache lines**, as they are also known, typically range in size from 8 to 64 bytes. Data is moved in and out of the cache a line at a time using memory interleaving (discussed earlier).

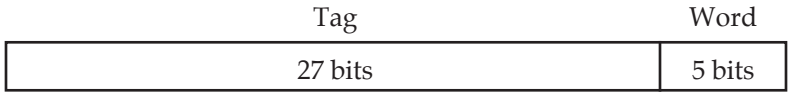
The cache for this example consists of 2^{14} **slots** into which main memory blocks are placed. There are more main memory blocks than there are cache slots, and any one of the 2^{27} main memory blocks can be mapped into each cache slot (with only one block placed in a slot at a time). To keep track of which one of the 2^{27} possible blocks is in each slot, a 27-bit **tag** field is added to each slot which holds an identifier in the range from 0 to $2^{27} - 1$. The tag field is the most signif-

icant 27 bits of the 32-bit memory address presented to the cache. All the tags are stored in a special tag memory where they can be searched in parallel. Whenever a new block is stored in the cache, its tag is stored in the corresponding tag memory location.

When a program is first loaded into main memory, the cache is cleared, and so while a program is executing, a **valid** bit is needed to indicate whether or not the slot holds a block that belongs to the program being executed. There is also a **dirty** bit that keeps track of whether or not a block has been modified while it is in the cache. A slot that is modified must be written back to the main memory before the slot is reused for another block.

A referenced location that is found in the cache results in a **hit**, otherwise, the result is a **miss**. When a program is initially loaded into memory, the valid bits are all set to 0. The first instruction that is executed in the program will therefore cause a miss, since none of the program is in the cache at this point. The block that causes the miss is located in the main memory and is loaded into the cache.

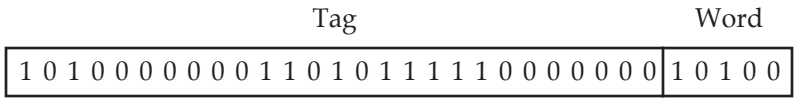
In an associative mapped cache, each main memory block can be mapped to any slot. The mapping from main memory blocks to cache slots is performed by partitioning an address into fields for the tag and the word (also known as the "byte" field) as shown below:



When a reference is made to a main memory address, the cache hardware intercepts the reference and searches the cache tag memory to see if the requested block is in the cache. For each slot, if the valid bit is 1, then the tag field of the referenced address is compared with the tag field of the slot. All of the tags are searched in parallel, using an **associative memory** (which is something different than an associative mapping scheme. See Section 7.8.3 for more on associative memories.) If any tag in the cache tag memory matches the tag field of the memory reference, then the word is taken from the position in the slot specified by the word field. If the referenced word is not found in the cache, then the main memory block that contains the word is brought into the cache and the referenced word is then taken from the cache. The tag, valid, and dirty fields are updated, and the program resumes execution.

Consider how an access to memory location $(A035F014)_{16}$ is mapped to the

cache. The leftmost 27 bits of the address form the tag field, and the remaining five bits form the word field as shown below:



If the addressed word is in the cache, it will be found in word $(14)_{16}$ of a slot that has a tag of $(501AF80)_{16}$, which is made up of the 27 most significant bits of the address. If the addressed word is not in the cache, then the block corresponding to the tag field $(501AF80)_{16}$ will be brought into an available slot in the cache from the main memory, and the memory reference that caused the "cache miss" will then be satisfied from the cache.

Although this mapping scheme is powerful enough to satisfy a wide range of memory access situations, there are two implementation problems that limit performance. First, the process of deciding which slot should be freed when a new block is brought into the cache can be complex. This process requires a significant amount of hardware and introduces delays in memory accesses. A second problem is that when the cache is searched, the tag field of the referenced address must be compared with all 2^{14} tag fields in the cache. (Alternative methods that limit the number of comparisons are described in Sections 7.6.2 and 7.6.3.)

Replacement Policies in Associative Mapped Caches

When a new block needs to be placed in an associative mapped cache, an available slot must be identified. If there are unused slots, such as when a program begins execution, then the first slot with a valid bit of 0 can simply be used. When all of the valid bits for all cache slots are 1, however, then one of the active slots must be freed for the new block. Four replacement policies that are commonly used are: **least recently used** (LRU), **first-in first-out** (FIFO), **least frequently used** (LFU), and **random**. A fifth policy that is used for analysis purposes only, is **optimal**.

For the LRU policy, a time stamp is added to each slot, which is updated when any slot is accessed. When a slot must be freed for a new block, the contents of the least recently used slot, as identified by the age of the corresponding time stamp, are discarded and the new block is written to that slot. The LFU policy works similarly, except that only one slot is updated at a time by incrementing a frequency counter that is attached to each slot. When a slot is needed for a new block, the least frequently used slot is freed. The FIFO policy replaces slots in

round-robin fashion, one after the next in the order of their physical locations in the cache. The random replacement policy simply chooses a slot at random.

The optimal replacement policy is not practical, but is used for comparison purposes to determine how effective other replacement policies are to the best possible. That is, the optimal replacement policy is determined only after a program has already executed, and so it is of little help to a running program.

Studies have shown that the LFU policy is only slightly better than the random policy. The LRU policy can be implemented efficiently, and is sometimes preferred over the others for that reason. A simple implementation of the LRU policy is covered in Section 7.6.7.

Advantages and Disadvantages of the Associative Mapped Cache

The associative mapped cache has the advantage that any main memory block can be placed into any cache slot. This means that regardless of how irregular the data and program references are, if a slot is available for the block, it can be stored in the cache. This results in considerable hardware overhead needed for cache bookkeeping. Each slot must have a 27-bit tag that identifies its location in main memory, and each tag must be searched in parallel. This means that in the example above the tag memory must be 27×2^{14} bits in size, and as described above, there must be a mechanism for searching the tag memory in parallel. Memories that can be searched for their contents, in parallel, are referred to as **associative**, or **content-addressable** memories. We will discuss this kind of memory later in the chapter.

By restricting where each main memory block can be placed in the cache, we can eliminate the need for an associative memory. This kind of cache is referred to as a **direct mapped cache**, which is discussed in the next section.

7.6.2 *DIRECT MAPPED CACHE*

Figure 7-14 shows a direct mapping scheme for a 2^{32} word memory. As before, the memory is divided into 2^{27} blocks of $2^5 = 32$ words per block, and the cache consists of 2^{14} slots. There are more main memory blocks than there are cache slots, and a total of $2^{27}/2^{14} = 2^{13}$ main memory blocks can be mapped onto each cache slot. In order to keep track of which of the 2^{13} possible blocks is in each slot, a 13-bit tag field is added to each slot which holds an identifier in the range

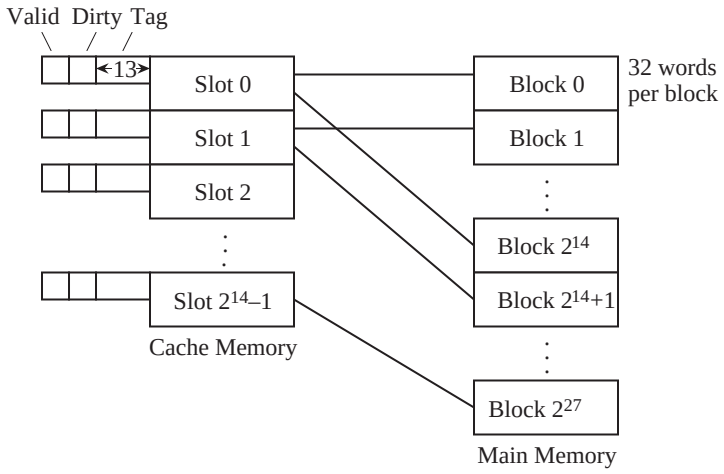


Figure 7-14 A direct mapping scheme for cache memory.

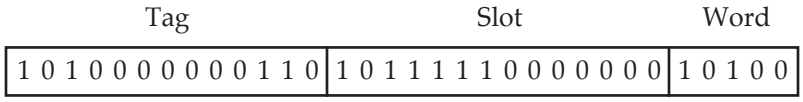
from 0 to $2^{13} - 1$.

This scheme is called “direct mapping” because each cache slot corresponds to an explicit set of main memory blocks. For a direct mapped cache, each main memory block can be mapped to only one slot, but each slot can receive more than one block. The mapping from main memory blocks to cache slots is performed by partitioning an address into fields for the tag, the slot, and the word as shown below:

Tag	Slot	Word
13 bits	14 bits	5 bits

The 32-bit main memory address is partitioned into a 13-bit tag field, followed by a 14-bit slot field, followed by a five-bit word field. When a reference is made to a main memory address, the slot field identifies in which of the 2^{14} slots the block will be found if it is in the cache. If the valid bit is 1, then the tag field of the referenced address is compared with the tag field of the slot. If the tag fields are the same, then the word is taken from the position in the slot specified by the word field. If the valid bit is 1 but the tag fields are not the same, then the slot is written back to main memory if the dirty bit is set, and the corresponding main memory block is then read into the slot. For a program that has just started execution, the valid bit will be 0, and so the block is simply written to the slot. The valid bit for the block is then set to 1, and the program resumes execution.

Consider how an access to memory location $(A035F014)_{16}$ is mapped to the cache. The bit pattern is partitioned according to the word format shown above. The leftmost 13 bits form the tag field, the next 14 bits form the slot field, and the remaining five bits form the word field as shown below:



If the addressed word is in the cache, it will be found in word $(14)_{16}$ of slot $(2F80)_{16}$, which will have a tag of $(1406)_{16}$.

Advantages and Disadvantages of the Direct Mapped Cache

The direct mapped cache is a relatively simple scheme to implement. The tag memory in the example above is only 13×2^{14} bits in size, less than half of the associative mapped cache. Furthermore, there is no need for an associative search, since the slot field of the main memory address from the CPU is used to "direct" the comparison to the single slot where the block will be if it is indeed in the cache.

This simplicity comes at a cost. Consider what happens when a program references locations that are 2^{19} words apart, which is the size of the cache. This pattern can arise naturally if a matrix is stored in memory by rows and is accessed by columns. Every memory reference will result in a miss, which will cause an entire block to be read into the cache even though only a single word is used. Worse still, only a small fraction of the available cache memory will actually be used.

Now it may seem that any programmer who writes a program this way deserves the resulting poor performance, but in fact, fast matrix calculations use power-of-two dimensions (which allows shift operations to replace costly multiplications and divisions for array indexing), and so the worst-case scenario of accessing memory locations that are 2^{19} addresses apart is not all that unlikely. To avoid this situation without paying the high implementation price of a fully associative cache memory, the **set associative mapping** scheme can be used, which combines aspects of both direct mapping and associative mapping. Set associative mapping, which is also known as **set-direct mapping**, is described in the next section.

7.6.3 *SET ASSOCIATIVE MAPPED CACHE*

The set associative mapping scheme combines the simplicity of direct mapping with the flexibility of associative mapping. Set associative mapping is more practical than fully associative mapping because the associative portion is limited to just a few slots that make up a set, as illustrated in Figure 7-15. For this example,

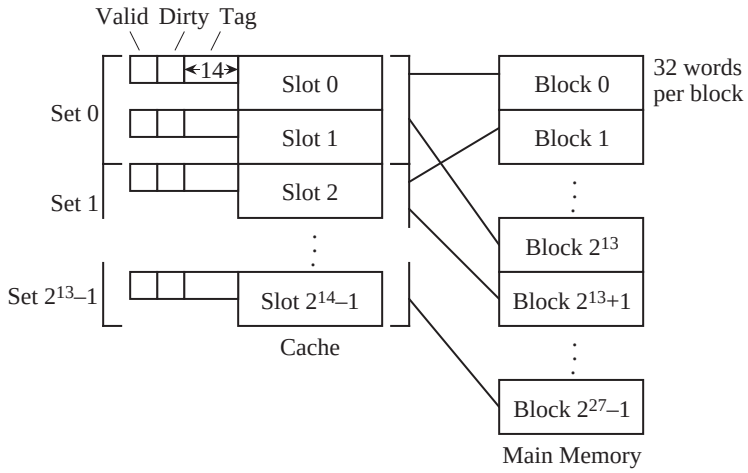


Figure 7-15 A set associative mapping scheme for a cache memory.

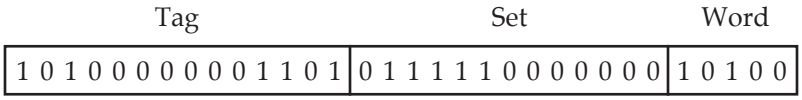
two blocks make up a set, and so it is a two-way set associative cache. If there are four blocks per set, then it is a four-way set associative cache.

Since there are 2^{14} slots in the cache, there are $2^{14}/2 = 2^{13}$ sets. When an address is mapped to a set, the direct mapping scheme is used, and then associative mapping is used within a set. The format for an address has 13 bits in the set field, which identifies the set in which the addressed word will be found if it is in the cache. There are five bits for the word field as before and there is a 14-bit tag field that together make up the remaining 32 bits of the address as shown below:

Tag	Set	Word
14 bits	13 bits	5 bits

As an example of how the set associative cache views a main memory address, consider again the address $(A035F014)_{16}$. The leftmost 14 bits form the tag field, followed by 13 bits for the set field, followed by five bits for the word field

as shown below:



As before, the partitioning of the address field is known only to the cache, and the rest of the computer is oblivious to any address translation.

Advantages and Disadvantages of the Set Associative Mapped Cache

In the example above, the tag memory increases only slightly from the direct mapping example, to 13×2^{14} bits, and only two tags need to be searched for each memory reference. The set associative cache is almost universally used in today's microprocessors.

7.6.4 *CACHE PERFORMANCE*

Notice that we can readily replace the cache direct mapping hardware with associative or set associative mapping hardware, without making any other changes to the computer or the software. Only the runtime performance will change between methods.

Runtime performance is the purpose behind using a cache memory, and there are a number of issues that need to be addressed as to what triggers a word or block to be moved between the cache and the main memory. Cache read and write policies are summarized in Figure 7-16. The policies depend upon whether or not the requested word is in the cache. If a cache read operation is taking place, and the referenced data is in the cache, then there is a "cache hit" and the referenced data is immediately forwarded to the CPU. When a cache miss occurs, then the entire block that contains the referenced word is read into the cache.

In some cache organizations, the word that causes the miss is immediately forwarded to the CPU as soon as it is read into the cache, rather than waiting for the remainder of the cache slot to be filled, which is known as a **load-through** operation. For a non-interleaved main memory, if the word occurs in the last position of the block, then no performance gain is realized since the entire slot is brought in before load-through can take place. For an interleaved main memory, the order of accesses can be organized so that a load-through operation will always result in a performance gain.

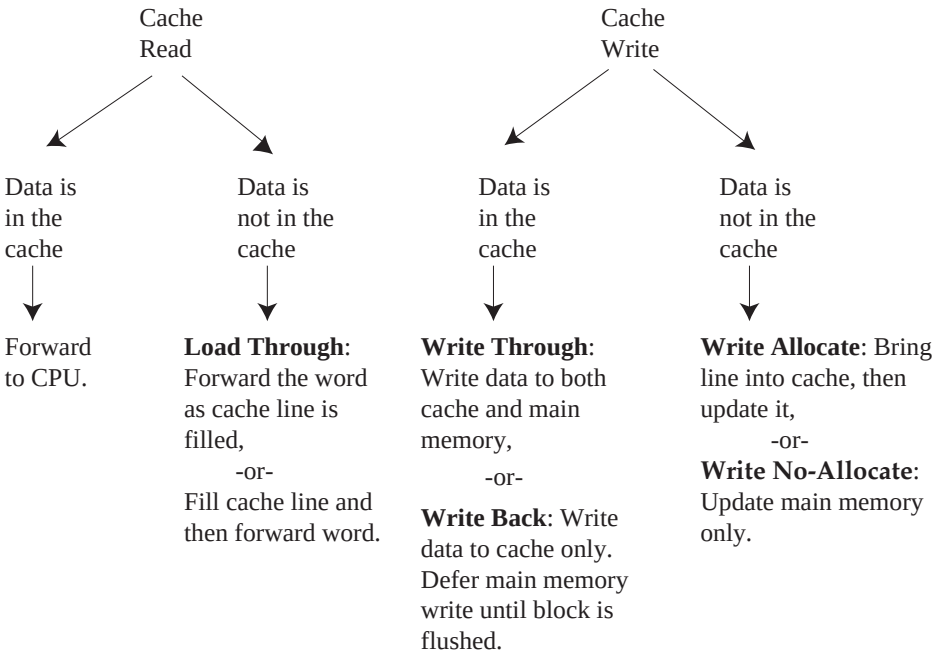


Figure 7-16 Cache read and write policies.

For write operations, if the word is in the cache, then there may be *two* copies of the word, one in the cache, and one in main memory. If both are updated simultaneously, this is referred to as **write-through**. If the write is deferred until the cache line is flushed from the cache, this is referred to as **write-back**. Even if the data item is not in the cache when the write occurs, there is the choice of bringing the block containing the word into the cache and then updating it, known as **write-allocate**, or to update it in main memory without involving the cache, known as **write-no-allocate**.

Some computers have separate caches for instructions and data, which is a variation of a configuration known as the **Harvard architecture** (also known as a **split cache**), in which instructions and data are stored in separate sections of memory. Since instruction slots can never be dirty (unless we write self-modifying code, which is rare these days), an instruction cache is simpler than a data cache. In support of this configuration, observations have shown that most of the memory traffic moves away from main memory rather than toward it. Statistically, there is only one write to memory for every four read operations from memory. One reason for this is that instructions in an executing program are only read from the main memory, and are never written to the memory except by

the system loader. Another reason is that operations on data typically involve reading two operands and storing a single result, which means there are two read operations for every write operation. A cache that only handles reads, while sending writes directly to main memory can thus also be effective, although not necessarily as effective as a fully functional cache.

As to which cache read and write policies are best, there is no simple answer. The organization of a cache is optimized for each computer architecture and the mix of programs that the computer executes. Cache organization and cache sizes are normally determined by the results of simulation runs that expose the nature of memory traffic.

7.6.5 *HIT RATIOS AND EFFECTIVE ACCESS TIMES*

Two measures that characterize the performance of a cache memory are the **hit ratio** and the **effective access time**. The hit ratio is computed by dividing the number of times referenced words are found in the cache by the total number of memory references. The effective access time is computed by dividing the total time spent accessing memory (summing the main memory and cache access times) by the total number of memory references. The corresponding equations are given below:

$$Hit\ ratio = \frac{No.\ times\ referenced\ words\ are\ in\ cache}{Total\ number\ of\ memory\ accesses}$$

$$Eff.\ access\ time = \frac{(\#\ hits)(Time\ per\ hit) + (\#\ misses)(Time\ per\ miss)}{Total\ number\ of\ memory\ access}$$

Consider computing the hit ratio and the effective access time for a program running on a computer that has a direct mapped cache with four 16-word slots. The layout of the cache and the main memory are shown in Figure 7-17. The cache access time is 80 ns, and the time for transferring a main memory block to the cache is 2500 ns. Assume that load-through is used in this architecture and that the cache is initially empty. A sample program executes from memory locations 48 – 95, and then loops 10 times from 15 – 31 before halting.

We record the events as the program executes as shown in Figure 7-18. Since the memory is initially empty, the first instruction that executes causes a miss. A miss thus occurs at location 48, which causes main memory block #3 to be read into cache slot #3. This first memory access takes 2500 ns to complete. Load-through is used for this example, and so the word that causes the miss at location 48 is passed directly to the CPU while the rest of the block is loaded into the cache

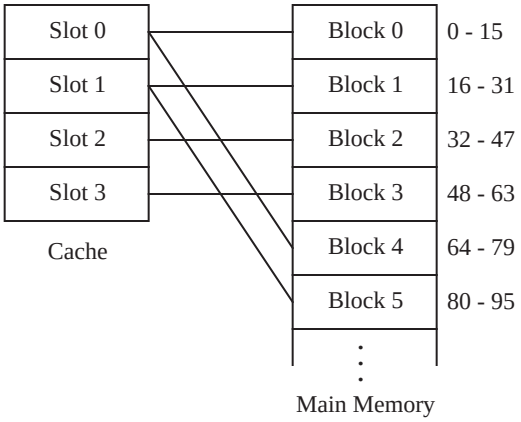


Figure 7-17 An example of a direct mapped cache memory.

Event	Location	Time	Comment
1 miss	48	2500ns	Memory block 3 to cache slot 3
15 hits	49-63	80ns×15=1200ns	
1 miss	64	2500ns	Memory block 4 to cache slot 0
15 hits	65-79	80ns×15=1200ns	
1 miss	80	2500ns	Memory block 5 to cache slot 1
15 hits	81-95	80ns×15=1200ns	
1 miss	15	2500ns	Memory block 0 to cache slot 0
1 miss	16	2500ns	Memory block 1 to cache slot 1
15 hits	17-31	80ns×15=1200ns	
9 hits	15	80ns×9=720ns	Last nine iterations of loop
144 hits	16-31	80ns×144=12,240ns	Last nine iterations of loop
Total hits = 213		Total misses = 5	

Figure 7-18 A table of events for a program executing on an architecture with a small direct mapped cache memory.

slot. The next event consists of 15 hits for locations 49 through 63. The events that follow are recorded in a similar manner, and the result is a total of 213 hits and five misses. The total number of accesses is 213 + 5 = 218. The hit ratio and effective access time are computed as shown below:

$$Hit\ ratio = \frac{213}{218} = 97.7\%$$

$$Effective Access Time = \frac{(213)(80ns) + (5)(2500ns)}{218} = 136ns$$

Although the hit ratio is 97.6%, the effective access time for this example is

almost 75% longer than the cache access time. This is due to the large amount of time spent in accessing a block from main memory.

7.6.6 *MULTILEVEL CACHES*

As the sizes of silicon ICs have increased, and the packing density of components on ICs has increased, it has become possible to include cache memory on the same IC as the processor. Since the on-chip processing speed is faster than the speed of communication between chips, an on-chip cache can be faster than an off-chip cache. Current technology is not dense enough to allow the entire cache to be placed on the same chip as the processor, however. For this reason, **multilevel caches** have been developed, in which the fastest level of the cache, L1, is on the same chip as the processor, and the remaining cache is placed off of the processor chip. Data and instruction caches are separately maintained in the L1 cache. The L2 cache is unified, which means that the same cache holds both data and instructions.

In order to compute the hit ratio and effective access time for a multilevel cache, the hits and misses must be recorded among both caches. Equations that represent the overall hit ratio and the overall effective access time for a two-level cache are shown below. H_1 is the hit ratio for the on-chip cache, H_2 is the hit ratio for the off-chip cache, and T_{EFF} is the overall effective access time. The method can be extended to any number of levels.

$$H_1 = \frac{\textit{No. times accessed word is in on-chip cache}}{\textit{Total number of memory accesses}}$$

$$H_2 = \frac{\textit{No. times accessed word is in off-chip cache}}{\textit{No. times accessed word is not in on-chip cache}}$$

$$T_{EFF} = (\textit{No. on-chip cache hits})(\textit{On-chip cache hit time}) + (\textit{No. off-chip cache hits})(\textit{Off-chip cache hit time}) + (\textit{No. off-chip cache misses})(\textit{Off-chip cache miss time}) / \textit{Total number of memory accesses}$$

7.6.7 *CACHE MANAGEMENT*

Management of a cache memory presents a complex problem to the system programmer. If a given memory location represents an I/O port, as it may in memory-mapped systems, then it probably should not appear in the cache at all. If it is cached, the value in the I/O port may change, and this change will not be reflected in the value of the data stored in the cache. This is known as “stale”

data: the copy that is in the cache is "stale" compared with the value in main memory. Likewise, in shared-memory multiprocessor environments (see Chapter 10), where more than one processor may access the same main memory, either the cached value or the value in main memory may become stale due to the activity of one or more of the CPUs. At a minimum, the cache in a multiprocessor environment should implement a write-through policy for those cache lines which map to shared memory locations.

For these reasons, and others, most modern processor architectures allow the system programmer to have some measure of control over the cache. For example, the Motorola PPC 601 processor's cache, which normally enforces a write-back policy, can be set to a write-through policy on a per-line basis. Other instructions allow individual lines to be specified as noncacheable, or to be marked as invalid, loaded, or flushed.

Internal to the cache, replacement policies (for associative and set-associative caches) need to be implemented efficiently. An efficient implementation of the LRU replacement policy can be achieved with the **Neat Little LRU Algorithm** (origin unknown). Continuing with the cache example used in Section 7.6.5, we construct a matrix in which there is a row and a column for every slot in the cache, as shown in Figure 7-19. Initially, all of the cells are set to 0. Each time

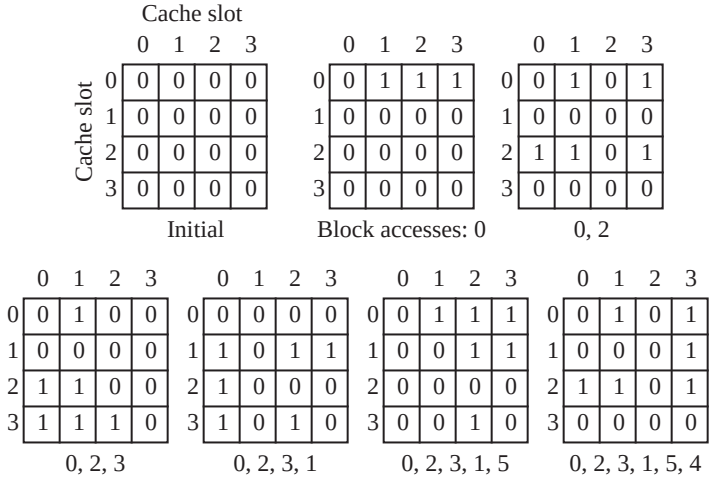


Figure 7-19 A sequence is shown for the Neat Little LRU Algorithm for a cache with four slots. Main memory blocks are accessed in the sequence: 0, 2, 3, 1, 5, 4.

that a slot is accessed, 1's are written into each cell in the row of the table that

corresponds to that slot. 0's are then written into each cell in the column that corresponds to that slot. Whenever a slot is needed, the row that contains all 0's is the oldest and is used next. At the beginning of the process, more than one row will contain all 0's, and so a tie-breaking mechanism is needed. The first row with all 0's is one method that will work, which we use here.

The example shown in Figure 7-19 shows the configuration of the matrix as blocks are accessed in the order: 0, 2, 3, 1, 5, 4. Initially, the matrix is filled with 0's. After a reference is made to block 0, the row corresponding to block 0 is filled with 1's and the column corresponding to block 0 is filled with 0's. For this example, block 0 happens to be placed in slot 0, but for other situations, block 0 can be placed in any slot. The process continues until all cache slots are in use at the end of the sequence: 0, 2, 3, 1. In order to bring the next block (5) into the cache, a slot must be freed. The row for slot 0 contains 0's, and so it is the least recently used slot. Block 5 is then brought into slot 0. Similarly, when block 4 is brought into the cache, slot 2 is overwritten.

7.7 Virtual Memory

Despite the enormous advancements in creating ever larger memories in smaller areas, computer memory is still like closet space, in the sense that we can never have enough of it. An economical method of extending the apparent size of the main memory is to augment it with disk space, which is one aspect of **virtual memory** that we cover in this section. Disk storage appears near the bottom of the memory hierarchy, with a lower cost per bit than main memory, and so it is reasonable to use disk storage to hold the portions of a program or data sets that do not entirely fit into the main memory. In a different aspect of virtual memory, complex address mapping schemes are supported, which give greater flexibility in how the memory is used. We explore these aspects of virtual memory below.

7.7.1 *OVERLAYS*

An early approach of using disk storage to augment the main memory made use of **overlays**, in which an executing program overwrites its own code with other code as needed. In this scenario, the programmer has the responsibility of managing memory usage. Figure 7-20 shows an example in which a program contains a main routine and three subroutines *A*, *B*, and *C*. The physical memory is smaller than the size of the program, but is larger than any single routine. A strategy for managing memory using overlays is to modify the program so that it keeps track of which subroutines are in memory, and reads in subroutine code as