Chapter 9 Memory Hierarchy
(Revision number 22)

Let us first understand what we mean by memory hierarchy. So far, we have treated the physical memory as a black box. In the implementation of LC-2200 (Chapter 3 and Chapter 5), we treated memory as part of the datapath. There is an implicit assumption in that arrangement, namely, accessing the memory takes the same amount of time as performing any other datapath operation. Let us dig a little deeper into that assumption. Today processor clock speeds have reached the GHz range. This means that the CPU clock cycle time is less than a nanosecond. Let us compare that to the state-of-the-art memory speeds (circa 2009). Physical memory, implemented using *dynamic random access memory (DRAM)* technology, has a cycle time in the range of 100 nanoseconds. We know that the slowest member determines the cycle time of the processor in a pipelined processor implementation. Given that IF and MEM stages of the pipeline access memory, we have to find ways to bridge the 100:1 speed disparity that exists between the CPU and the memory.

It is useful to define two terminologies frequently used in memory systems, namely, *access time* and *cycle time*. The delay between submitting a request to the memory and getting the data is called the access time. On the other hand, the time gap needed between two successive requests to the memory system is called the cycle time. A number of factors are responsible for the disparity between the access time and cycle time. For example, DRAM technology uses a single transistor to store a bit. Reading this bit depletes the charge, and therefore requires a replenishment before the same bit is read again. This is why DRAMs have different cycle time and access time. In addition to the specific technology used to realize the memory system, transmission delays on buses used to connect the processor to the memory system add to the disparity between access time and cycle time.

Let us re-visit the processor datapath of LC-2200. It contains a register file, which is also a kind of memory. The access time of a small 16-element register file is at the speed of the other datapath elements. There are two reasons why this is so. The first reason is that the register-file uses a different technology referred to as *static random access memory (SRAM)*. The virtue of this technology is speed. Succinctly put, SRAM gets its speed advantage over DRAM by using six transistors for storing each bit arranged in such a way that eliminates the need for replenishing the charge after a read. For the same reason, there is no disparity between cycle time and access time for SRAMs As a rule of thumb, SRAM cycle time can be 8 to 16 times faster than DRAMs. As you may have guessed, SRAMs are also bulkier than DRAMs since they use 6 transistors per bit (as opposed to 1 per bit of DRAM), and consume more power for the same reason. Not surprisingly, SRAMs are also considerably more expensive per bit (roughly 8 to 16 times) than DRAMs.

The second and more compelling reason why physical memory is slow compared to register file is the sheer *size*. We usually refer to a register file as a 16-, 32-, or 64-

element entity. On the other hand, we usually specify the size of memory in quantities of Kbytes, Mbytes, or these days in Gbytes. In other words, even if we were to implement physical memory using SRAM technology the larger structure will result in slower access time compared to a register file. A simple state of reality is that independent of the implementation technology (i.e., SRAM or DRAM), you can have high speed or large size but not both. In other words, you cannot have the cake and eat it too!

Pragmatically speaking, SRAMs do not lend themselves to realizing large memory systems due to a variety of reasons, including power consumption, die area, delays that are inevitable with large memories built out of SRAM, and ultimately the sheer cost of this technology for realizing large memories. On the other hand, DRAM technology is much more frugal with respect to power consumption compared to its SRAM counterpart and lends itself to very large scale integration. For example, quoting 2007 numbers, a single DRAM chip may contain up to 256 Mbits with an access time of 70 ns. The virtue of the DRAM technology is the size. Thus, it is economically feasible to realize large memory systems using DRAM technology.

9.1 The Concept of a Cache

It is feasible to have a small amount of fast memory and/or a large amount of slow memory. Ideally, we would like to have the *size* advantage of a *large slow memory* and the *speed* advantage of a *fast small memory*. Given the earlier discussion regarding size and speed, we would choose to implement the small fast memory using SRAM, and the large slow memory using DRAM.

Memory hierarchy comes into play to achieve these twin goals. Figure 9.1 shows the basic idea behind memory hierarchy. *Main Memory* is the physical memory that is visible to the instruction-set of the computer. *Cache*, as the name suggests, is a hidden storage. We already introduced the general idea of caches and its application at various levels of the computer system in Chapter 8 (please see Section 8.6) when we discussed TLBs, which are a special case of caches used for holding address translations. Specifically, in the context of the memory accesses made by the processor, the idea is to stash information brought from the memory in the cache. It is much smaller than the main memory and hence much faster.

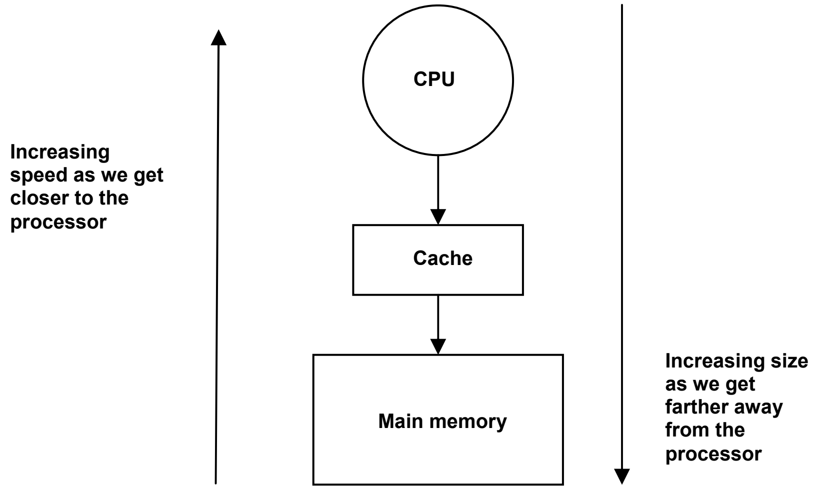

CPU
Cache
Main memory
Increasing speed as we get closer to the processor
Increasing size as we get farther away from the processor

Figure 9.1: A basic memory hierarchy. The figure shows a two-level hierarchy. Modern processors may have a deeper hierarchy (up to 3 levels of caches followed by the main memory).

Our intent is as follows: The CPU looks in the cache for the data it seeks from the main memory. If the data is not there then it retrieves it from the main memory. If the cache is able to service most of the CPU requests then effectively we will be able to get the speed advantage of the cache.

9.2 Principle of Locality

Let us understand why a cache works in the first place. The answer is contained in the principles of locality that we introduced in the previous chapter (see Section 8.4.2).

Stated broadly, a program tends to access a relatively small region of memory irrespective of its actual memory footprint in any given interval of time. While the region of activity may change over time, such changes are gradual. The principle of locality zeroes in on this tendency of programs.

Principle of locality has two dimensions, namely, *spatial* and *temporal*. Spatial locality refers to the high probability of a program accessing *adjacent* memory locations *..., i-3, i-2, i-1, i+1, i+2, i+3, ...*, if it accesses a location *i*. This observation is intuitive. A program's instructions occupy contiguous memory locations. Similarly, data structures such as arrays and records occupy contiguous memory locations. Temporal locality refers to the high probability of a program accessing in the near future, the *same* memory location *i* that it is accessing currently. This observation is intuitive as well considering that a program may be executing a looping construct and thus revisiting the same instructions repeatedly and/or updating the same data structures repeatedly in an iterative algorithm. We will shortly find ways to exploit these locality properties in the cache design.

9.3 Basic terminologies

We will now introduce some intuitive definitions of terms commonly used to describe the performance of memory hierarchies. Before we do that, it would be helpful to remind ourselves of the toolbox and tool tray analogy from Chapter 2. If you needed a tool, you first went and looked in the tool tray. If it is there, it saved you a trip to the toolbox; if not you went to the garage where you keep the toolbox and bring the tool, use it, and put it in the tool tray. Naturally, it is going to be much quicker if you found the tool in the tool tray. Of course, occasionally you may have to return some of the tools back to the toolbox from the tool tray, when the tray starts overflowing. Mathematically speaking, one can associate a probability of finding the tool in the tool tray; one minus this probability gives the odds of going to the toolbox.

Now, we are ready to use this analogy for defining some basic terminologies.

- *Hit*: This term refers to the CPU finding the contents of the memory address in the cache thus saving a trip to the deeper levels of the memory hierarchy, and analogous to finding the tool in the tool tray. *Hit rate (h)* is the probability of such a *successful lookup* of the cache by the CPU.
- *Miss*: This term refers to the CPU *failing* to find what it wants in the cache thus incurring a trip to the deeper levels of the memory hierarchy, and analogous to taking a trip to the toolbox; *Miss rate (m)* is the probability of *missing* in the cache and is equal to *1-h*.
- *Miss penalty*: This is the time penalty associated with servicing a miss at any particular level of the memory hierarchy, and analogous to the time to go to the garage to fetch the missing tool from the tool box.
- *Effective Memory Access Time (EMAT)*: This is the effective access time experienced by the CPU.

  This has two components:
  (a) time to lookup the cache to see if the memory location that the CPU is looking for is already there, defined as the *cache access time* or *hit time*, and
  (b) upon a miss in the cache, the time to go to the deeper levels of the memory hierarchy to fetch the missing memory location, defined as the *miss penalty*.
  The CPU is always going to incur the first component on every memory access. The second component, namely, miss penalty is governed by the access time to the deeper levels of the memory hierarchy, and is measured in terms of the number of CPU clock cycles that the processor has to be idle waiting for the miss to be serviced. The miss penalty depends on a number of factors including the organization of the cache and the details of the main memory system design. These factors will become more apparent in later sections of this chapter. Since the CPU incurs this penalty only on a miss, to compute the second component we need to condition the miss penalty with the probability (quantified by the miss rate *m*) that the memory location that the CPU is looking for is not presently in the cache.

Thus, if m, Tc, and Tm, be the cache miss rate, the cache access time, and the miss penalty, respectively.

$$\boxed{EMAT = Tc + m * Tm} \tag{1}$$

9.4 Multilevel Memory Hierarchy

As it turns out modern processors employ multiple levels of caches. For example, a state-of-the-art processor (circa 2006) has at least 2 levels of caches on-chip, referred to as *first-level (L1)*, *second-level (L2) caches*. You may be wondering if both the first and second level caches are on-chip, why not make it a single big cache? The answer lies in the fact that we are trying to take care of two different concerns simultaneously: fast access time to the cache (i.e., hit time), and lower miss rate. On the one hand, we want the hit time to be as small as possible to keep pace with the clock cycle time of the processor. Since the size of the cache has a direct impact on the access time, this suggests that the cache should be small. On the other hand, the growing gap between the processor cycle time and main memory access time suggests that the cache should be large. Addressing these twin concerns leads to multilevel caches. The first level cache is optimized for speed to keep pace with the clock cycle time of the processor, and hence is small. The speed of the second level cache only affects the miss penalty incurred by the first level cache and does not directly affect the clock cycle time of the processor. Therefore, the second level cache design focuses on reducing the miss rate and hence is large. The processor sits in an integrated circuit board, referred to as *motherboard*, which has the main (physical) memory[1]. High-end CPUs intended for enterprise class machines (database and web servers) even have a large off-chip *third-level cache (L3)*. For considerations of speed of access time, these multiple levels of the caches are all usually implemented with SRAM technology.

For reasons outlined above, the sizes of the caches become bigger as we move away from the processor. If S_i is the size of the cache at level i, then,

$$S_{i+n} > S_{i+n-1} > \ldots. > S_2 > S_1$$

Correspondingly, the access times also increase as we move from the processor. If T_i is the access time at level i, then,

$$T_{i+n} > T_{i+n-1} > \ldots. > T_2 > T_1$$

Generalizing the EMAT terminology to the memory hierarchy as a whole, let T_i and m_i be the access time and miss rate for any level of the memory hierarchy, respectively. The effective memory access time for any level i of the memory hierarchy is given by the recursive formula:

$$\boxed{\text{EMAT}_i = T_i + m_i * \text{EMAT}_{i+1}} \tag{2}$$

[1] The typical access times and sizes for these different levels of the cache hierarchy change with advances in technology.

We are now ready to generalize the concept of memory hierarchy:

> *Memory hierarchy is defined as all the storage containing either instructions and/or data that a processor accesses either directly or indirectly.*

By direct access we mean that the storage is visible to the ISA. By indirect access we mean that it is not visible to the ISA. Figure 9.2 illustrates this definition. In Chapter 2, we introduced registers which are the fastest and closest data storage available to the processor for direct access from the ISA. Typically, load/store instructions, and arithmetic/logic instructions in the ISA access the registers. L1, L2, and L3 are different levels of caches that a processor (usually) implicitly accesses every time it accesses main memory for bringing in an instruction or data. Usually L1 and L2 are on-chip, while L3 is off-chip. The main memory serves as the storage for instructions and data of the program. The processor explicitly accesses the main memory for instructions (via the program counter), and for data (via load/store instructions, and other instructions that use memory operands). It should be noted that some architectures also allow direct access to the cache from the ISA by the processor (e.g. for flushing the contents of the cache). The secondary storage serves as the home for the entire memory footprint of the program, a part of which is resident in the main memory consistent with the working set principle that we discussed in Chapter 8. In other words, the secondary storage serves as the home for the virtual memory. The processor accesses the virtual memory implicitly upon a page fault to bring in the faulting virtual page into the main (i.e., physical) memory.

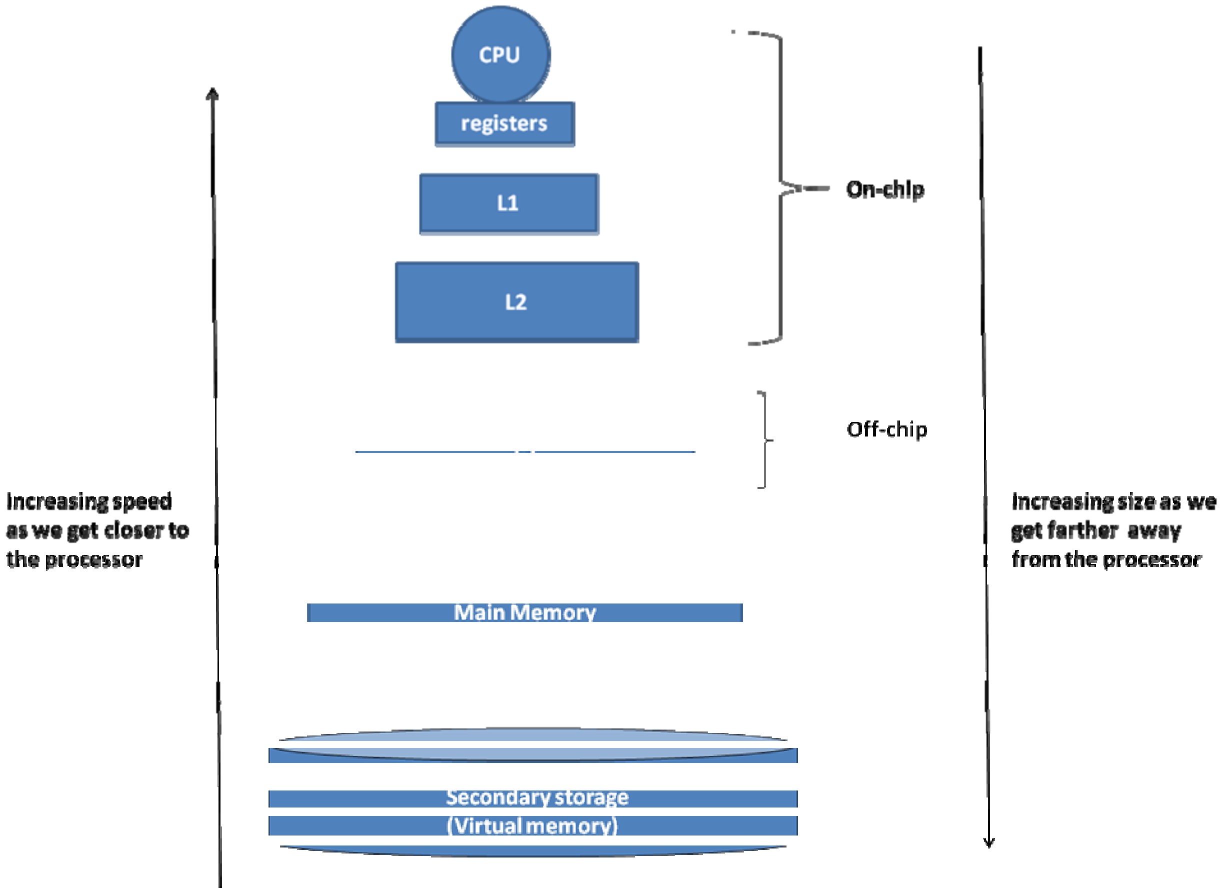

CPU
registers
L1
L2
On-chip
Off-chip
Increasing speed as we get closer to the processor
Increasing size as we get farther away from the processor
Main Memory
Secondary storage
(Virtual memory)

Figure 9.2: The entire memory hierarchy stretching from processor registers to the virtual memory.

This chapter focuses only on the portion of the memory hierarchy that includes the caches and main memory. We already have a pretty good understanding of the registers from earlier chapters (Chapters 2, 3, and 5). Similarly, we have a good understanding of the virtual memory from Chapters 7 and 8. In this chapter, we focus only on the caches and main memory. Consequently, we use the term *cache hierarchy* and *memory hierarchy* interchangeably to mean the same thing in the rest of the chapter.

Example 1:

Consider a three-level memory hierarchy as shown in Figure 9.3. Compute the effective memory access time.

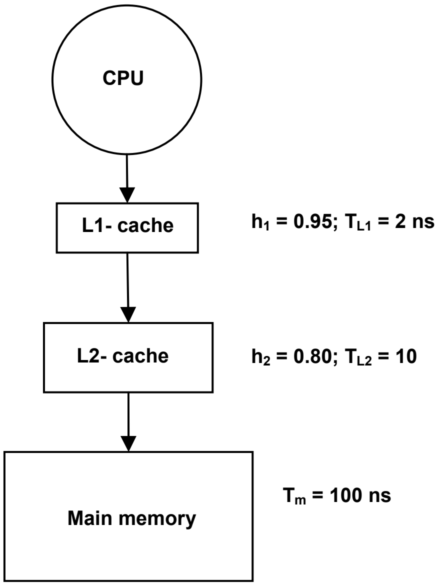


Figure 9.3: Three-level memory hierarchy

Answer:

$EMAT_{L2} = T_{L2} + (1 - h_2) * T_m$
$= 10 + (1 - 0.8) * 100$
$= 30$ ns
$EMAT_{L1} = T_{L1} + (1 - h_1) * EMAT_{L2}$
$= 2 + (1 - 0.95) * 30$
$= 2 + 1,5$
$= 3.5$ ns
EMAT = $EMAT_{L1}$ = **3.5 ns**

9.5 Cache organization

There are three facets to the organization of the cache: *placement, algorithm for lookup,* and *validity*.
These three facets deal with the following questions, respectively:

1. Where do we place in the cache the data read from the memory?
2. How do we find something that we have placed in the cache?
3. How do we know if the data in the cache is valid?

A mapping function that takes a given memory address to a cache index is the key to answering the first question. In addition to the mapping function, the second question concerns with the additional meta-data in each cache entry that helps identify the cache contents unambiguously. The third question brings out the necessity of providing a valid bit in each cache entry to assist the lookup algorithm.

We will look at these three facets specifically in the context of a simple cache organization, namely, a *direct-mapped* cache. In Section 9.11, we will look at other cache organizations, namely, *fully associative* and *set-associative*.

9.6 Direct-mapped cache organization

A direct-mapped cache has a one-to-one correspondence between a memory location and a cache location[2]. That is, given a memory address there is exactly one place to put its contents in the cache. To understand how direct-mapping works, let us consider a very simple example, a memory with sixteen words and a cache with eight words (Figure 9.4). The shading in the figure shows the mapping of memory locations 0 to 7 to locations 0 to 7 of the cache, respectively; and similarly, locations 8 to 16 of the memory map to locations 0 to 7 of the cache, respectively.

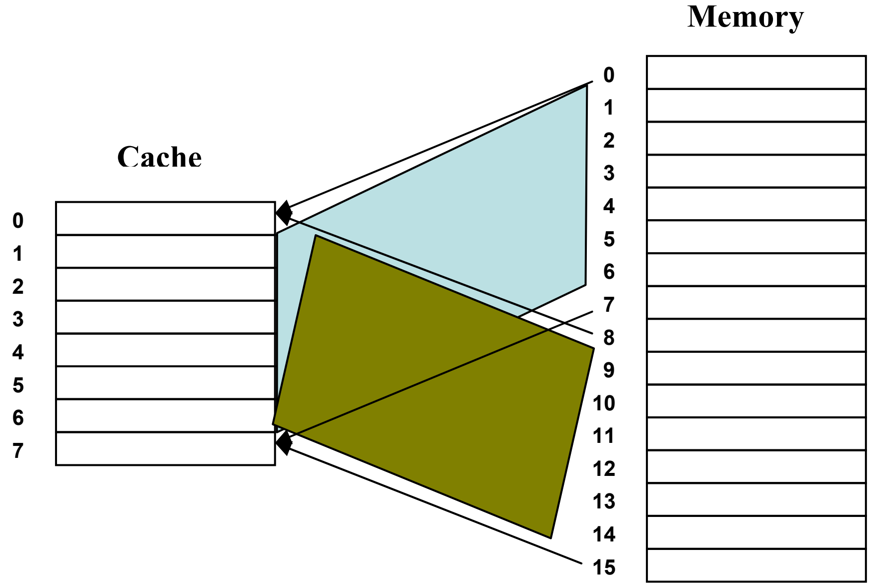


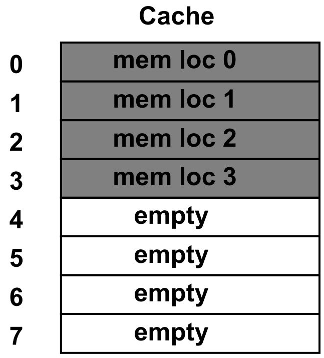
Figure 9.4: Direct-mapped cache

Before we explore the questions of lookup and validity, let us first understand the placement of memory locations in the cache with this direct mapping. To make the discussion concrete, let us assume the cache is empty and consider the following sequence of memory references:

0, 1, 2, 3, 1, 3, 0, 8, 0, 9, 10 (all decimal addresses)

Since the cache is initially empty, the first four references (addresses 0, 1, 2, 3) *miss* in the cache and the CPU retrieves the data from the memory and stashes them in the cache. Figure 9.5 shows the cache after servicing the first four memory references. These are inevitable misses, referred to as *compulsory misses*, since the cache is initially empty.

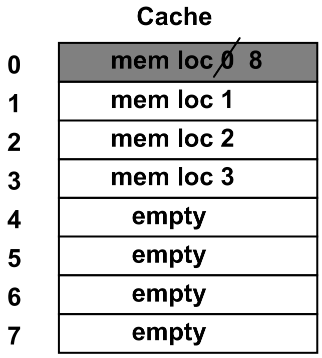
[2] Of course, since cache is necessarily smaller than memory, there is a many-to-one relationship between a set of memory locations and a given cache location.



	Cache
0	mem loc 0
1	mem loc 1
2	mem loc 2
3	mem loc 3
4	empty
5	empty
6	empty
7	empty

Figure 9.5: Content of Cache after the first 4 references

The next three CPU references (addresses 1, 3, 0) *hit* in the cache thus avoiding trips to the memory. Let us see what happens on the next CPU reference (address 8). This reference will *miss* in the cache and the CPU retrieves the data from the memory. Now we have to figure out where the system will stash the data in the cache from this memory location. Cache has space in locations 4-7. However, with direct-mapping cache location 0 is the only spot for storing memory location 8. Therefore, the cache has to evict memory location 0 to make room for memory location 8. Figure 9.6 shows this state of the cache. This is also a *compulsory miss* since memory location 8 is not in the cache to start with.



	Cache
0	mem loc ~~0~~ 8
1	mem loc 1
2	mem loc 2
3	mem loc 3
4	empty
5	empty
6	empty
7	empty

Figure 9.6: Memory location 0 replaced by 8

Consider the next reference (address 0). This reference also *misses* in the cache and the CPU has to retrieve it from the memory and stash it in cache location 0, which is the only spot for memory location 0. Figure 9.7 shows the new contents of the cache. This miss occurs due to the *conflict* between memory location 0 and 8 for a spot in the cache, and hence referred to as a *conflict miss*. A conflict miss occurs due to direct mapping despite the fact that there are unused locations available in the cache. Note that the previous miss (location 8 in Figure 9.6) also caused a conflict since location 0 was already present in that cache entry. Regardless of this situation, first access to a memory location will always result in a mss which is why we categorized the miss in Figure 9.6 as a compulsory miss. We will revisit the different types of misses in some more detail in a later section (Section 9.12).

	Cache
0	mem loc ~~0~~ ~~8~~ 0
1	mem loc 1
2	mem loc 2
3	mem loc 3
4	empty
5	empty
6	empty
7	empty

Figure 9.7: Memory location 8 replaced by 0 (conflict miss)

9.6.1 Cache Lookup

By now, we know the information exchange or the *handshake* between the CPU and the memory: the CPU supplies the address and the command (e.g. read) and gets back the data from the memory. The introduction of the cache (Figure 9.1) changes this simple set up. The CPU looks up the cache first to see if the data it is looking for is in the cache. Only on a miss does the CPU resort to its normal CPU-memory handshake.
Let us understand how the CPU *looks up* the cache for the memory location of interest. Specifically, we need to figure out what *index* the CPU has to present to the cache with a direct-mapped organization. Looking back at Figure 9.4, one can see how the CPU addresses map to their corresponding cache *indices*.

One can compute the numerical value of the cache index as:

Memory-address *mod* cache-size

For example, given the memory address 15 the cache index is
15 mod 8 = 7,
Similarly the cache index for memory address 7 is
7 mod 8 = 7.

Essentially, to construct the cache index we simply take the least significant bits of the memory address commensurate with the cache size. In our previous example, since the cache has 8 entries, we would need 3 bits for the cache index (the least significant three bits of the memory address).

Suppose the CPU needs to get data from memory address 8; 1000 is the memory address in binary; the cache index is 000. The CPU looks up the cache location at index 000. We need some way of knowing if the contents of this cache entry are from memory location 0 or 8. Therefore, in addition to the data itself, we need information in each entry of the cache to distinguish between multiple memory addresses that may map to the same cache entry. The bits of the memory address that were "dropped" to generate the cache index are exactly the information needed for this purpose. We refer to this additional information (which will be stored in the cache) as the *tag*. For example, the most significant bit of the memory address 0000 and 1000 are 0 and 1 respectively. Therefore, our simple cache needs a 1-bit tag with each cache entry (Figure 9.8). If the CPU wants to access memory location 11, it looks up location 11 mod 8 in the cache, i.e., location 3 in the cache. This location contains a tag value of 0. Therefore, the data contained in this cache entry corresponds to memory location 3 (binary address 0011) and not that of 11 (binary address 1011).

	tag	data
0	1	mem loc ~~0~~ 8
1	0	mem loc 1
2	0	mem loc 2
3	0	mem loc 3
4		empty
5		empty
6		empty
7		empty

Figure 9.8: Direct-mapped cache with a tag field and a data field in each entry

Suppose the CPU generates memory address 0110 (memory address 6). Let us assume this is the first time memory address 6 is being referenced by the CPU. So, we know it cannot be in the cache. Let us see the sequence of events here as the CPU tries to read memory location 6. The CPU will first look up location 6 (6 mod 8) in the cache. If the tag happens to be 0, then the CPU will assume that the data corresponds to memory location 6. Well, the CPU did not ever fetch that location from memory so far in our example; it is by chance that the tag is 0. Therefore, the data contained in this cache entry does not correspond to the actual memory location 6. One can see that this is erroneous, and points to the need for additional information in each cache entry to avoid this error. The tag is useful for disambiguation of the memory location currently in the cache, but it does not help in knowing if the entry is *valid*. To fix this problem, we add a *valid* field to each entry in the cache (Figure 9.9).

	valid	tag	data
0	1	1	loc 8
1	1	0	loc 1
2	1	0	loc 2
3	1	0	loc 3
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Figure 9.9: Direct-mapped cache with valid field, tag field, and data field in each entry. A value of "X" in the tag field indicates a "don't care" condition.

9.6.2 Fields of a Cache Entry

To summarize, each cache entry contains three fields (Figure 9.10)

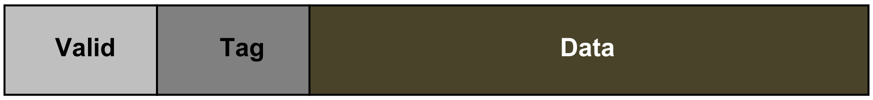


Figure 9.10: Fields of each cache entry

Thus, the memory address generated by the CPU has two parts from the point of view of looking up the cache: *tag* and *index*. Index is the specific cache location that could contain the memory address generated by the CPU; tag is the portion of the address to disambiguate contents of the specific cache entry (Figure 9.11).

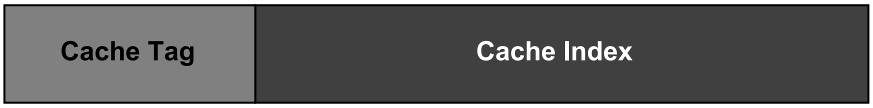
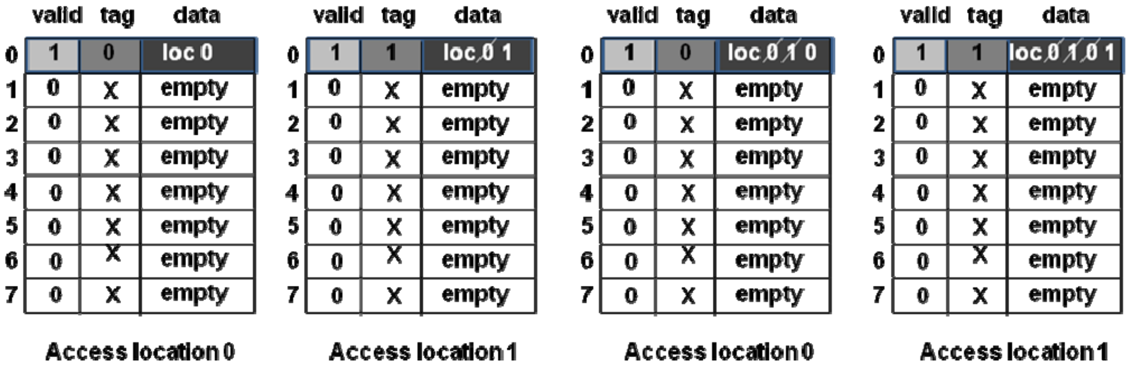


Figure 9.11: Interpreting the memory address generated by the CPU for Cache Lookup

We use the least significant (i.e., the right most) bits of the memory address as cache index to take advantage of the principle of spatial locality. For example, in our simple cache, if we use the most significant 3 bits of the memory address to index the cache, then memory locations 0 and 1 will be competing for the same spot in the cache.

For example, consider the access sequence for memory addresses 0, 1, 0, 1, with the 8-entry direct-mapped cache as shown in Figure 9.4. Assume that the most significant three bits of the memory address are used as the cache index and the least significant bit (since the memory address is only 4-bits for this example) is used as the tag. Figure 9.12 shows the contents of the cache after each access. Notice how the same cache entry (first row of the cache) is reused for the sequence of memory accesses though the rest of the cache is empty. Every access results in a miss, replacing what was in the cache previously due to the access pattern.



	valid	tag	data
0	1	0	loc 0
1	0	X	empty
2	0	X	empty
3	0	X	empty
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Access location 0

	valid	tag	data
0	1	1	loc ~~0~~ 1
1	0	X	empty
2	0	X	empty
3	0	X	empty
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Access location 1

	valid	tag	data
0	1	0	loc ~~0~~ ~~1~~ 0
1	0	X	empty
2	0	X	empty
3	0	X	empty
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Access location 0

	valid	tag	data
0	1	1	loc ~~0~~ ~~1~~ ~~0~~ 1
1	0	X	empty
2	0	X	empty
3	0	X	empty
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Access location 1

Figure 9.12: Sequence of memory accesses if the cache index is chosen as the most significant bits of the memory address

The situation shown in Figure 9.12 is undesirable. Recall the locality properties (spatial and temporal), which we mentioned in Section 9.2. It is imperative that sequential access to memory fall into different cache locations. This is the reason for choosing to interpret the memory address as shown in Figure 9.11.

9.6.3 Hardware for direct mapped cache

Let us put together the ideas we have discussed thus far. Figure 9.13 shows the hardware organization for a direct-mapped cache.

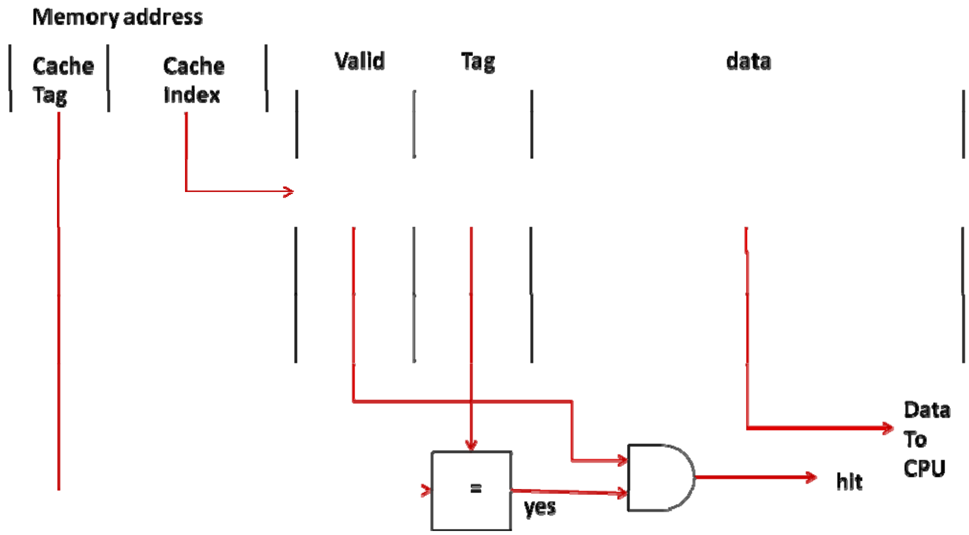


Figure 9.13: Hardware for direct-mapped cache. The shaded entry is the cache location picked out by the cache index.

The index part of the memory address picks out a unique entry in the cache (the textured cache entry in Figure 9.13). The comparator shown in Figure 9.13 compares the tag field of this entry against the tag part of the memory address. If there is a *match* and *if* the entry is *valid* then it signals a *hit*. The cache supplies the data field of the selected entry (also referred to as *cache line)* to the CPU upon a hit. *Cache block* is another term used synonymously with cache line. We have used three terms synonymously thus far: cache entry, cache line, and cache block. While this is unfortunate, it is important that the reader develop this vocabulary since computer architecture textbooks tend to use these terms interchangeably.

Note that the actual amount of storage space needed in the cache is more than that needed simply for the *data* part of the cache. The *valid* bits, and *tag* fields, called *meta-data*, are for managing the actual data contained in the cache, and represent a *space overhead*.

Thus far, we have treated the data field of the cache to hold one memory location. The size of the memory location depends on the granularity of memory access allowed by the instruction-set. For example, if the architecture is byte-addressable then a byte is the smallest possible size of a memory operand. Usually in such an architecture, the word-width is some integral number of bytes. We could place each byte in a separate cache line, but from Section 9.2, we know that the principle of spatial locality suggests that if we access a byte of a word then there is a good chance that we would access other bytes of the same word. Therefore, it would make sense to design the cache to contain a complete word in each cache line even if the architecture is byte addressable. Thus, the memory address generated by the CPU would be interpreted as consisting of three fields as shown below: cache tag, cache index, and byte offset.

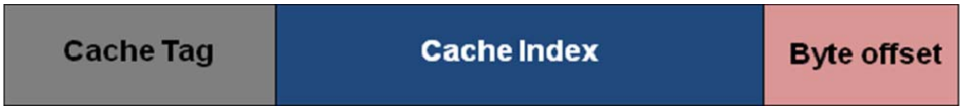


Figure 9.14: Interpreting the memory address generated by the CPU when a single cache block contains multiple bytes

Byte offset is defined as the bits of the address that specify the byte within the word. For example, if the word-width is 32-bits and the architecture is byte addressable then the bottom 2-bits of the address form the byte offset.

Example 2:

Let us consider the design of a direct-mapped cache for a realistic memory system. Assume that the CPU generates a 32-bit byte-addressable memory address. Each memory word contains 4 bytes. A memory access brings in a full word into the cache. The direct-mapped cache is 64K Bytes in size (this is the amount of data that can be stored in the cache), with each cache entry containing one word of data. Compute the additional storage space needed for the valid bits and the tag fields of the cache.

Answer:

Assuming little-endian notation, 0 is the least significant bit of the address. With this notation, the least significant two bits of the address, namely, bits 1 and 0 specify the byte within a word address. A cache entry holds a full word of 4 bytes. Therefore, the least significant two bits of the address, while necessary for uniquely identifying a byte within a word, are not needed to uniquely identify the particular cache entry. Therefore, these bits do not form part of the index for cache lookup.

The ratio of the cache size to the data stored in each entry gives the number of cache entries:

64K Bytes/ (4 Bytes/word) = 16 K entries

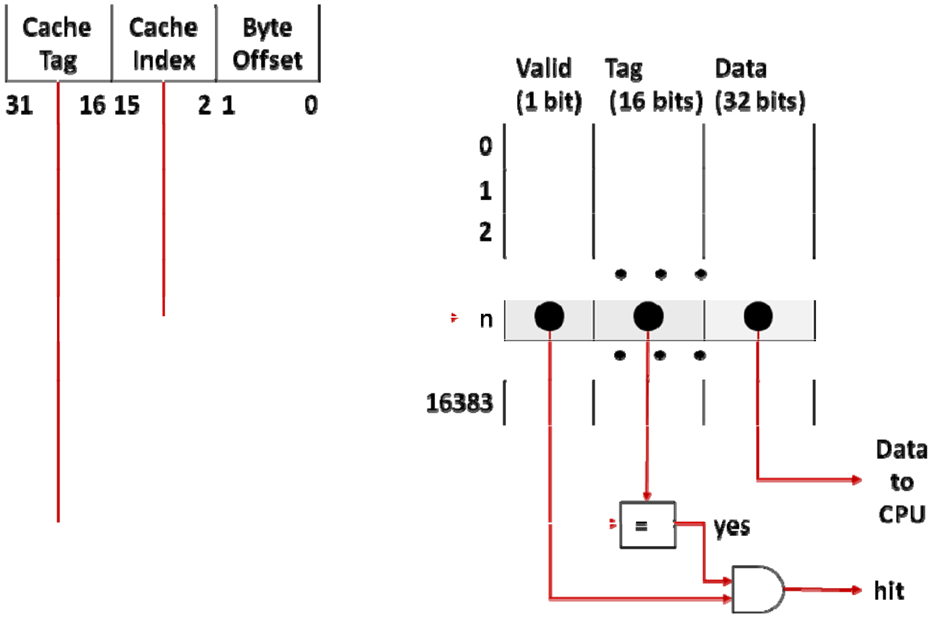
16 K entries require 14 bits to enumerate thus bits 2-15 form the cache index, leaving bits 16-31 to form the tag. Thus, each cache entry has a 16-bit tag.

The meta-data per entry totals 16 bits for the tag + 1 bit for valid bit = 17 bits

Thus, the additional storage space needed for the meta-data:

17 bits x 16K entries = 17 x 16,384 = **278,528 bits.**

The following figure shows the layout of the cache for this problem.



Total space needed for the cache (actual data + meta-data)
= 64K bytes + 278,528
= 524,288 + 278, 528
= 802,816

The space overhead = meta-data/total space = 278,528/802,816 = **35%**

Let us see how we can reduce the space overhead. In Example 2, each cache line holds one memory word. One way of reducing the space overhead is to modify the design such that each cache line holds multiple contiguous memory words. For example, consider each cache line holding four contiguous memory words in Example 2. This would reduce the number of cache lines to 4K. *Block size* is the term used to refer to the amount of contiguous data in one cache line. Block size in Example 2 is 4 bytes. If each cache line were to contain 4 words, then the block size would be 16 bytes. Why would we want to have a larger block size? Moreover, how does it help in reducing the space overhead? Will it help in improving the performance of the memory system as a whole? We will let the reader ponder these questions for a while. We will revisit them in much more detail in Section 9.10 where we discuss the impact of block size on cache design.

9.7 Repercussion on pipelined processor design

With the cache memory introduced between the processor and the memory, we can return to our pipelined processor design and re-examine instruction execution in the presence of caches. Figure 9.15 is a reproduction of the pipelined processor from Chapter 6 (Figure 6.6)

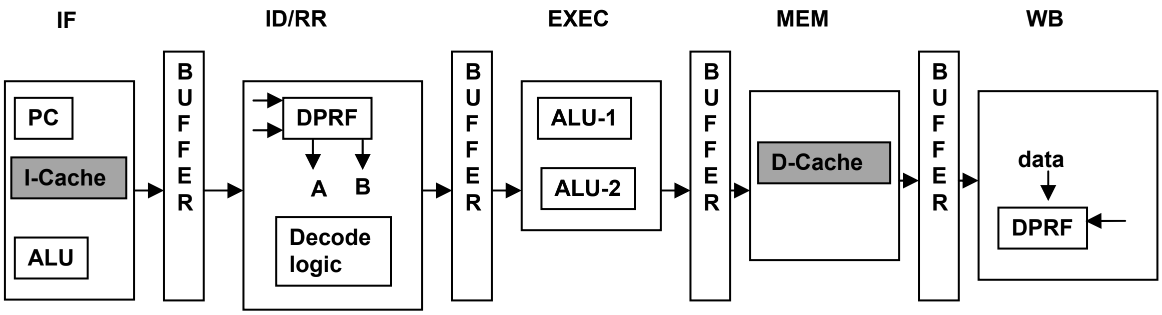


Figure 9.15: Pipelined processor with caches

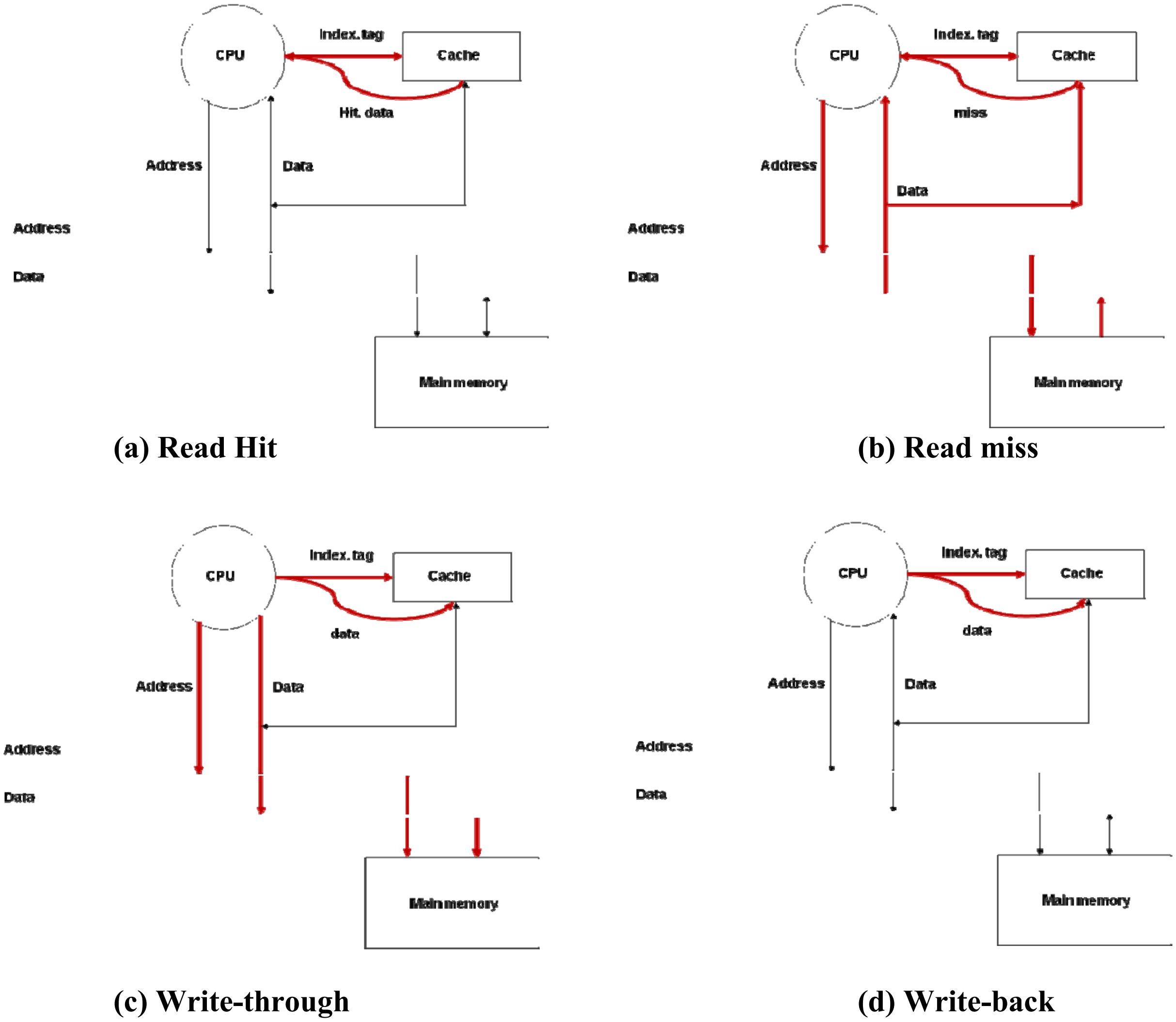
Notice that we have replaced the memories, I-MEM and D-MEM in the IF and MEM stages, by caches I-Cache and D-Cache, respectively. The caches make it possible for the IF and MEM stages to have comparable cycle times to the other stages of the pipeline, assuming the references result in hits in the respective caches. Let us see what would happen if the references miss in the caches.

- **Miss in the IF stage:** Upon a miss in the I-Cache, the IF stage sends the reference to the memory to retrieve the instruction. As we know, the memory access time may be several 10's of CPU cycles. Until the instruction arrives from the memory, the IF stage sends NOPs (bubbles) to the next stage.
- **Miss in the MEM stage:** Of course, misses in the D-Cache are relevant only for memory reference instructions (load/store). Similar to the IF stage, a miss in the MEM stage results in NOPs to the WB stage until the memory reference completes. It also *freezes* the preceding stages from advancing past the instructions they are currently working on.

We define *memory stall* as the processor cycles wasted due to waiting for a memory operation to complete. Memory stalls come in two flavors: *read stall* may be incurred due to read access from the processor to the cache; *write stall* may be incurred during write access from the processor to the cache. We will define and elaborate on these stalls in the next section together with mechanisms to avoid them since they are detrimental to processor pipeline performance.

9.8 Cache read/write algorithms

In this section, we discuss policies and mechanisms for reading and writing the caches. Different levels of the cache hierarchy may choose different policies.



(a) Read Hit **(b) Read miss**

(c) Write-through **(d) Write-back**

Figure 9.16: CPU, Cache, Memory interactions for reads and writes

9.8.1 Read access to the cache from the CPU

A processor needs to access the cache to read a memory location for either instructions or data. In our 5-stage pipeline for implementing the LC-2200 ISA, this could happen either for instruction fetch in the IF stage, or for operand fetch in response to a load instruction in the MEM stage. The basic actions taken by the processor and the cache are as follows:

- **Step 1:** CPU sends the index part of the memory address (Figure 9.16) to the cache. The cache does a lookup, and if successful (a cache *hit*), it supplies the data to the CPU. If the cache signals a miss, then the CPU sends the address on the memory bus to the main memory. In principle, all of these actions happen in the same cycle (either the IF or the MEM stage of the pipeline).
- **Step 2:** Upon sending the address to the memory, the CPU sends NOPs down to the subsequent stage until it receives the data from the memory. *Read stall* is defined as the number of processor clock cycles wasted to service a read-miss. As we observed earlier, this could take several CPU cycles depending on the memory speed. The

cache allocates a cache block to receive the memory block. Eventually, the main memory delivers the data to the CPU and simultaneously updates the allocated cache block with the data. The cache modifies the tag field of this cache entry appropriately and sets the valid bit.

9.8.2 Write access to the cache from the CPU

Write requests to the cache from the processor are the result of an instruction that wishes to write to a memory location. In our 5-stage pipeline for implementing the LC-2200 ISA, this could happen for storing a memory operand in the MEM stage. There are a couple of choices for handling processor write access to the cache: *write through* and *write back*.

9.8.2.1 Write through policy

The idea is to update the cache and the main memory on each CPU write operation. The basic actions taken by the processor and the cache are as follows:

- **Step 1:** On every write (store instruction in LC-2200), the CPU simply writes to the cache. There is no need to check the valid bit or the cache tag. The cache updates the tag field of the corresponding entry and sets the valid bit. These actions happen in the MEM stage of the pipeline.
- **Step 2:** Simultaneously, the CPU sends the address and data to the main memory. This of course is problematic in terms of performance since memory access takes several CPU cycles to complete. To alleviate this performance bottleneck, it is customary to include a *write-buffer* in the datapath between the CPU and the memory bus (shown in Figure 9.17). This is a small hardware store (similar to a register file) to smoothen out the speed disparity between the CPU and memory. As far as the CPU is concerned, the write operation is complete as soon as it places the address and data in the write buffer. Thus, this action also happens in the MEM stage of the pipeline without stalling the pipeline.
- **Step 3:** The write-buffer completes the write to the main memory independent of the CPU. Note that, if the write buffer is full at the time the processor attempts to write to it, then the pipeline will stall until one of the entries from the write buffer has been sent to the memory. *Write stall* is defined as the number of processor clock cycles wasted due to a write operation (regardless of hit or a miss in the cache).

With the write-through policy, write stall is a function of the cache block allocation strategy upon a write miss. There are two choices:

- **Write allocate:** This is the usual way to handle write misses. The intuition is that the data being written to will be needed by the program in the near future. Therefore, it is judicious to put it in the cache as well. However, since the block is missing from the cache, we have to allocate a cache block and bring in the missing memory block into it. In this sense, a write-miss is treated exactly similar to a read-miss. With a direct-mapped cache, the cache block to receive the memory block is pre-determined. However, as we will see later with flexible placement (please see Section 9.11), the allocation depends on the other design considerations.

- **No-write allocate:** This is an unusual way of handling a write miss. The argument for this strategy is that the write access can complete quickly since the processor does not need the data. The processor simply places the write in the write buffer to complete the operation needed in the MEM stage of the pipeline. Thus, there are write stalls incurred since the missing memory block need not be brought from the memory.

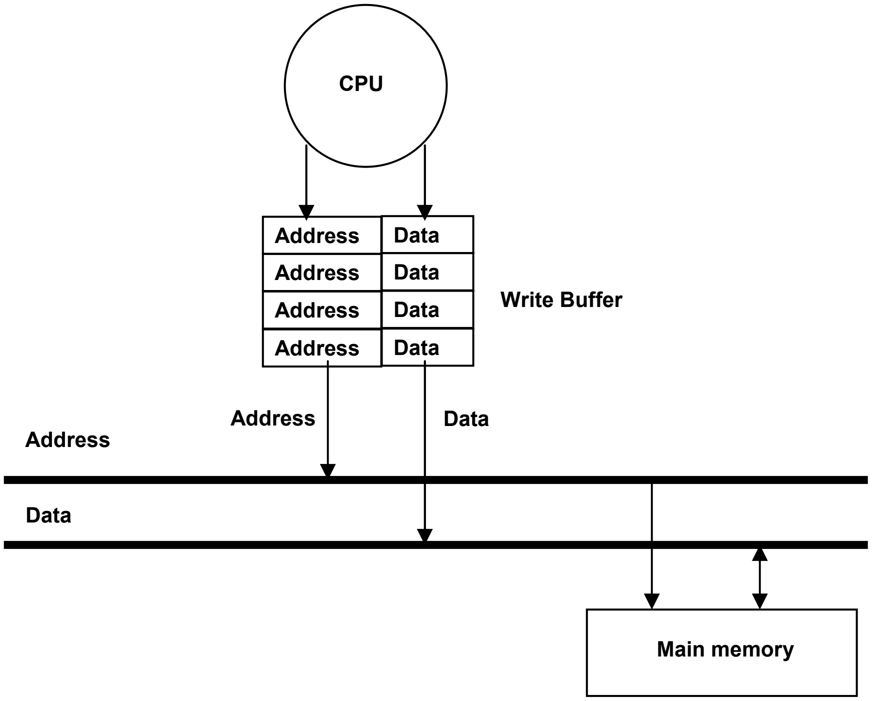


Figure 9.17: A 4-element write-buffer to bridge the CPU and main memory speeds for a write through policy. The processor write is complete and the pipeline can resume as soon as the write has been placed in the write buffer. It will be sent out to the memory in the background in parallel with the pipeline operation. The pipeline will freeze if the write buffer is full.

9.8.2.2 Write back policy

The idea here is to update only the cache upon a CPU write operation. The basic actions taken by the processor and the cache are as follows:

- **Step 1:** The CPU-cache interaction is exactly similar to the write-through policy. Let us assume that the memory location is already present in the cache (i. e., a write hit). The CPU writes to the cache. The cache updates the tag field of the corresponding entry and sets the valid bit. These actions happen in the MEM stage of the pipeline.
- **Step 2:** The contents of this chosen cache entry and the corresponding memory location are inconsistent with each other. As far as the CPU is concerned this is not a problem since it first checks the cache before going to memory on a read operation. Thus, the CPU always gets the latest value from the cache.

- **Step 3:** Let us see when we have to update the main memory. By design, the cache is much smaller than the main memory. Thus, at some point it may become necessary to replace an existing cache entry to make room for a memory location not currently present in the cache. We will discuss cache replacement policies in a later section of this chapter (Section 9.14). At the time of replacement, if the CPU had written into the cache entry chosen for replacement, then the corresponding memory location has to be updated with the latest data for this cache entry.

The processor treats a write-miss exactly like a read-miss. After taking the steps necessary to deal with a read-miss, it completes the write action as detailed above.

We need some mechanism by which the cache can determine that the data in a cache entry is more up to date than the corresponding memory location. The *meta-data* in each cache entry (valid bit and tag field) do not provide the necessary information for the cache to make this decision. Therefore, we add a new *meta-data* to each cache entry, a *dirty bit* (see Figure 9.18).

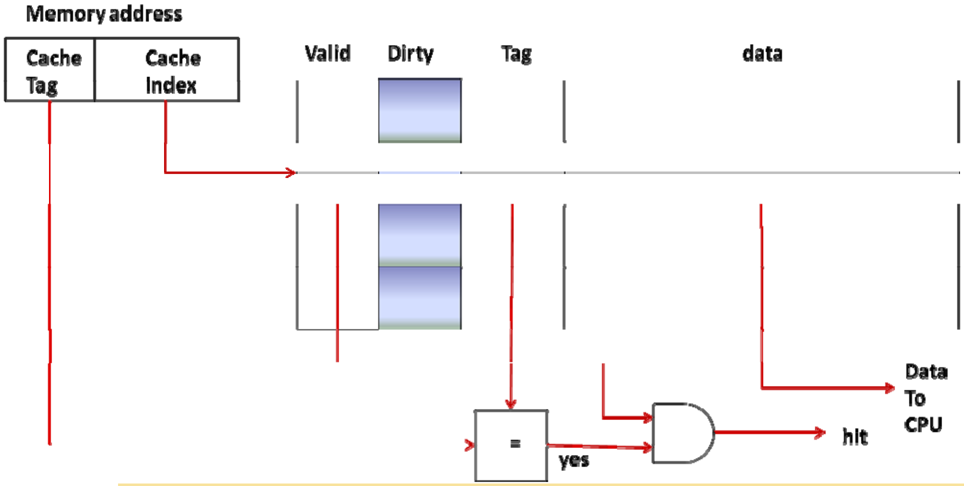


Figure 9.18: A direct-mapped cache organization with write-back policy. Each entry has an additional field (dirty bit) to indicate whether the block is dirty or clean. Horizontally shaded block is the one selected by the cache index.

The cache uses the dirty bit in the following manner:

- The cache *clears* the bit upon processing a miss that brings in a memory location into this cache entry.
- The cache *sets* the bit upon a CPU write operation.
- The cache *writes back* the data in the cache into the corresponding memory location upon replacement. Note that this action is similar to writing back a physical page frame to the disk in the virtual memory system (see Chapter 8).

We introduced the concept of a write-buffer in the context of write-through policy. It turns out that the write-buffer is useful for the write-back policy as well. Note that the focus is on keeping the processor happy. This means that the missing memory block should be brought into the cache as quickly as possible. In other words, servicing the miss is more important than writing back the dirty block that is being replaced. The write-buffer comes in handy to give preferential treatment to reading from the memory as compared to writing to memory. The block that has to be replaced (if dirty) is placed in the write buffer. It will eventually be written back. But the immediate request that is sent to the memory is to service the current miss (be it a read or a write miss). The write requests from the write-buffer can be sent to the memory when there are no read or write misses to be serviced from the CPU. This should be reminiscent of the optimization that the memory manager practices to give preferential treatment for reading in missing pages from the disk over writing out dirty victim pages out to the disk (see Section 8.4.1.1).

9.8.2.3 Comparison of the write policies

Which write policy should a cache use? The answer to this depends on several factors. On the one hand, write through ensures that the main memory is always up to date. However, this comes at the price of sending every write to memory. Some optimizations are possible. For example, if it is a repeated write to the same memory location that is already in the write buffer (i. e., it has not yet been sent to the memory), then it can be replaced by the new write. However, the chance of this occurring in real programs is quite small. Another optimization, referred to as *write merging*, merges independent writes to different parts of the same memory block. Write back has the advantage that it is faster, and more importantly, does not use the memory bus for every write access from the CPU. Thus, repeated writes to the same memory location does not result in creating undue traffic on the memory bus. Only upon a replacement, does the cache update the memory with the latest content for that memory location. In this context, note that the write-buffer is useful for the write-back policy as well, to hold the replacement candidate needing a write-back to the main memory.

Another advantage enjoyed by the write through policy is the fact that the cache is always clean. In other words, upon a cache block replacement to make room for a missing block, the cache never has to write the replaced block to the lower levels of the memory hierarchy. This leads to a simpler design for a write through cache and hence higher speed than a write back cache. Consequently, if a processor has multilevel cache hierarchy, then it is customary to use write through for the closer level to the processor. For example, most modern processors use a write through policy for the L1 D-cache, and a write back policy for the L2 and L3 levels.

As we will see in later chapters (see Chapter 10 and 12), the choice of write policy has an impact on the design of I/O and multiprocessor systems. The design of these systems could benefit from the reduced memory bus traffic of the write back policy; at the same time, they would also benefit from the write through policy keeping the memory always up to date.

9.9 Dealing with cache misses in the processor pipeline

Memory accesses disrupt the smooth operation of the processor pipeline. Therefore, we need to mitigate the ill effect of misses in the processor pipeline. It is not possible to mask a miss in the IF stage of the pipeline. However, it may be possible to *hide* a miss in the MEM stage.

- **Read miss in the MEM stage:** Consider a sequence of instructions shown below:

```
I1: ld      r1,  a       ;    r1 <- memory location a
I2: add     r3,  r4, r5  ;    r3 <- r4 + r5
I3: and     r6,  r7, r8  ;    r6 <- r7 AND r8
I4: add     r2,  r4, r5  ;    r2 <- r4 + r5
I5: add     r2,  r1, r2  ;    r2 <- r1 + r2
```

  Suppose the `ld` instruction results in a miss in the D-Cache; the CPU has to stall this load instruction in the MEM stage (freezing the preceding stages and sending NOPs to the WB stage) until the memory responds with the data. However, I5 is the instruction that uses the value loaded into `r1`. Let us understand how we can use this knowledge to prevent stalling the instructions I2, I3, and I4.

  In Chapter 5, we introduced the idea of a *busy* bit with each register in the register file for dealing with hazards. For instructions that modify a register value, the bit is set in the ID/RR stage (see Figure 9.15) and cleared once the write is complete. This idea is extensible to dealing with memory loads as well.

- **Write miss in the MEM stage:** This could be problematic depending on the write policy as well as the cache allocation policy. First, let us consider the situation with write-through policy. If the cache block allocation policy is no-write allocate, then the pipeline will not incur any stalls thanks to the write buffer. The processor simply places the write in the write-buffer to complete the actions needed in the MEM stage. However, if the cache block allocation policy is write-allocate then it has to be handled exactly similar to a read-miss. Therefore, the processor pipeline will incur write stalls in the MEM stage. The missing data block has to be brought into the cache, before the write operation can complete despite the presence of the write buffer. For write-back policy, write stalls in the MEM stage are inevitable since a write-miss has to be treated exactly similar to a read-miss.

9.9.1 Effect of memory stalls due to cache misses on pipeline performance

Let us re-visit the program execution time. In Chapter 5, we defined it to be:

$$\boxed{\text{Execution time} = \text{Number of instructions executed} * \text{CPI}_{\text{Avg}} * \text{clock cycle time}}$$

A pipelined processor attempts to make the CPI_{Avg} equal 1 since it tries to complete an instruction in every cycle. However, structural, data, and control hazards induce bubbles in the pipeline thus inflating the CPI_{Avg} to be higher than 1.

Memory hierarchy exacerbates this problem even more. Every instruction has at least one memory access, namely, to fetch the instruction. In addition, there may be additional accesses for memory reference instructions. If these references result in *misses* then they force bubbles in the pipeline. We refer to these additional bubbles due to the memory hierarchy as the *memory stall cycles*.

Thus, a more accurate expression for the execution time is:

$$\boxed{\text{Execution time} = (\text{Number of instructions executed} * (\text{CPI}_{\text{Avg}} + \text{Memory-stalls}_{\text{Avg}})) * \text{clock cycle time}} \quad (3)$$

We can define an effective CPI of the processor as:

$$\boxed{\text{Effective CPI} = \text{CPI}_{\text{Avg}} + \text{Memory-stalls}_{\text{Avg}}} \quad (4)$$

The total memory stalls experienced by a program is:

$$\boxed{\text{Total memory stalls} = \text{Number of instructions} * \text{Memory-stalls}_{\text{Avg}}} \quad (5)$$

The average number of memory stalls per instruction is:

$$\boxed{\text{Memory-stalls}_{\text{Avg}} = \text{misses per instruction}_{\text{Avg}} * \text{miss-penalty}_{\text{Avg}}} \quad (6)$$

Of course, if reads and writes incur different miss-penalties, we may have to account for them differently (see Example 3).

Example 3:

Consider a pipelined processor that has an average CPI of 1.8 without accounting for memory stalls. I-Cache has a hit rate of 95% and the D-Cache has a hit rate of 98%. Assume that memory reference instructions account for 30% of all the instructions executed. Out of these 80% are loads and 20% are stores. On average, the read-miss penalty is 20 cycles and the write-miss penalty is 5 cycles. Compute the effective CPI of the processor accounting for the memory stalls.

Answer:

The solution to this problem uses the equations (4) and (6) above.

Cost of instruction misses = I-cache miss rate * read miss penalty
= (1 - 0.95) * 20
= 1 cycle per instruction

Cost of data read misses = fraction of memory reference instructions in the program * fraction of memory reference instructions that are loads * D-cache miss rate * read miss penalty
= 0.3 * 0.8 * (1 – 0.98) * 20
= 0.096 cycles per instruction

Cost of data write misses = fraction of memory reference instructions in the program * fraction of memory reference instructions that are stores * D-cache miss rate * write miss penalty
= 0.3 * 0.2 * (1 – 0.98) * 5
= 0.006 cycles per instruction

Effective CPI = base CPI + Effect of I-Cache on CPI + Effect of D-Cache on CPI
= 1.8 + 1 + 0.096 + 0.006 = **2.902**

Reducing the *miss rate* and reducing the *miss penalty* are the keys to reducing the memory stalls, and thereby increase the efficiency of a pipelined processor.

There are two avenues available for reducing the miss rate and we will cover them in Sections 9.10 and 9.11. In Section 9.12, we will discuss ways to decrease the miss penalty.

9.10 Exploiting spatial locality to improve cache performance

The first avenue to reducing the miss rate exploits the principle of spatial locality. The basic idea is to bring adjacent memory locations into the cache upon a miss for a memory location *i*. Thus far, each entry in the cache corresponds to the unit of memory access architected in the instruction set. To exploit spatial locality in the cache design, we decouple the *unit of memory access* by an instruction from the *unit of memory transfer* between the memory and the cache. The instruction-set architecture of the processor decides the unit of memory access. For example, if an architecture has instructions that manipulate individual bytes, then the unit of memory access for such a processor would be a byte. On the other hand, the unit of memory transfer is a design parameter of the memory hierarchy. In particular, this parameter is always some integral multiple of the unit of memory access to exploit spatial locality. We refer to the unit of transfer between the cache and the memory as the *block size*. Upon a miss, the cache brings an entire *block* of *block size* bytes that contains the missing memory reference.

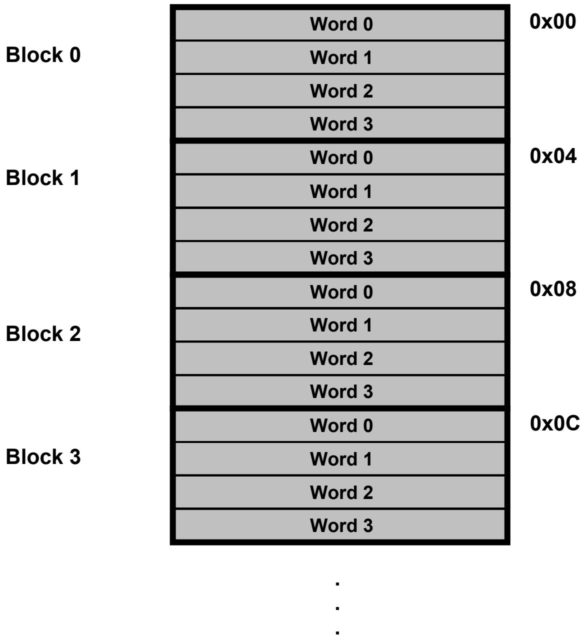


Figure 9.19: Main memory viewed as cache blocks

Figure 9.19 shows an example of such an organization and view of the memory. In this example, the block size is 4 words, where a word is the unit of memory access by a CPU instruction. The memory addresses for a block starts on a block boundary as shown in the Figure. For example, if the CPU misses on a reference to a memory location `0x01`, then the cache brings in 4 memory words that comprises *block 0* (starting at address `0x00`), and contains the location `0x01`.

With each entry in the cache organized as blocks of contiguous words, the address generated by the CPU has *three* parts: *tag*, *index*, *block-offset* as shown in Figure 9.20.

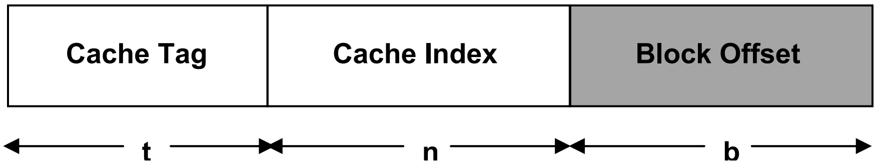


Figure 9.20: Interpreting the memory address generated by the CPU for a cache block containing multiple words

Note that the block offset is the number of bits necessary to enumerate the set of contiguous memory locations contained in a cache block. For example, if the block size is 64 bytes, then the block offset is 6-bits. All the data of a given block are contained in one cache entry. The tag and index fields have the same meaning as before. We will present general expressions for computing the number of bits needed for the different fields shown in Figure 9.20.

Let *a* be the number of bits in the memory address, *S* be the total size of the cache in bytes, and *B* the block size.

We have the following expressions for the fields shown in Figure 9.20:

$$b = log_2B \tag{7}$$
$$L = S/B \tag{8}$$
$$n = log_2L \tag{9}$$
$$t = a - (b+n) \tag{10}$$

L is the number of lines in a direct-mapped cache of size *S* and *B* bytes per block. Field *b* represents the least significant bits of the memory address; field *t* represents the most significant bits of the memory address; and field *n* represents the middle bits as shown in Figure 9.20. Please see Example 4 for a numerical calculation of these fields.

Now let us re-visit the basic cache algorithms (lookup, read, write) in the context of a multi-word block size. Figure 9.21 shows the organization of such a direct-mapped cache.

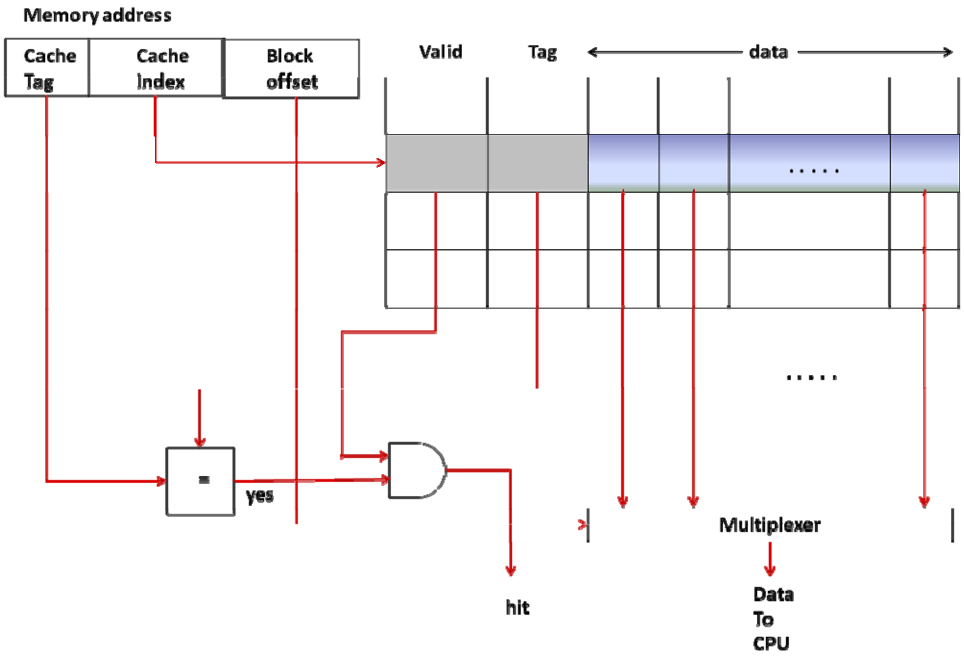


Figure 9.21: A multi-word direct-mapped cache organization. The block offset chooses the particular word to send to the CPU from the selected block using the multiplexer. Horizontally shaded block is the one selected by the cache index.

1. **Lookup:** The index for cache lookup comes from the middle part of the memory address as shown in Figure 9.20. The entry contains an entire block (if it is a hit as determined by the cache tag in the address and the tag stored in the specific entry). The least significant *b* bits of the address specify the specific word (or byte) within that block requested by the processor. A multiplexer selects the specific word (or byte) from the block using these *b* bits and sends it to the CPU (see Figure 9.21).

2. **Read:** Upon a read, the cache brings out the entire block corresponding to the cache index. If the tag comparison results in a hit, then the multiplexer selects the specific word (or byte) within the block and sends it to the CPU. If it is a miss then the CPU initiates a block transfer from the memory.

3. **Write:** We have to modify the write algorithm since there is only 1 valid bit for the entire cache line. Similar to the read-miss, the CPU initiates a block transfer from the memory upon a write-miss (see Figure 9.22).

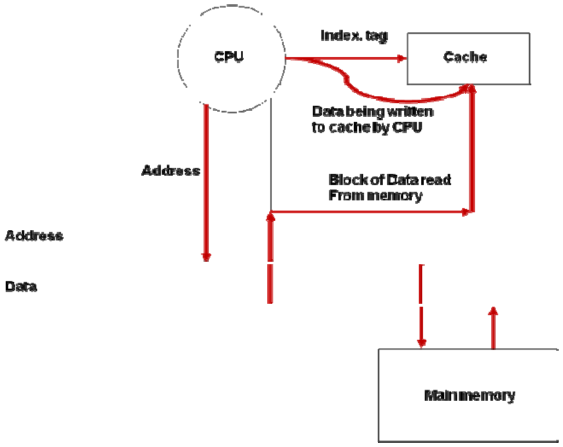


Figure 9.22: CPU, cache, and memory interactions for handling a write-miss

Upon a hit, the CPU writes the specific word (or byte) into the cache. Depending on the write policy, the implementation may require additional meta-data (in the form of dirty bits) and may take additional actions (such as writing the modified word or byte to memory) to complete a write operation (see Example 4).

Example 4:

Consider a multi-word direct-mapped cache with a data size of 64 Kbytes. The CPU generates a 32-bit byte-addressable memory address. Each memory word contains 4 bytes. The block size is 16 bytes. The cache uses a write-back policy with 1 dirty bit per word. The cache has one valid bit per data block.

(a) How does the CPU interpret the memory address?

Answer:

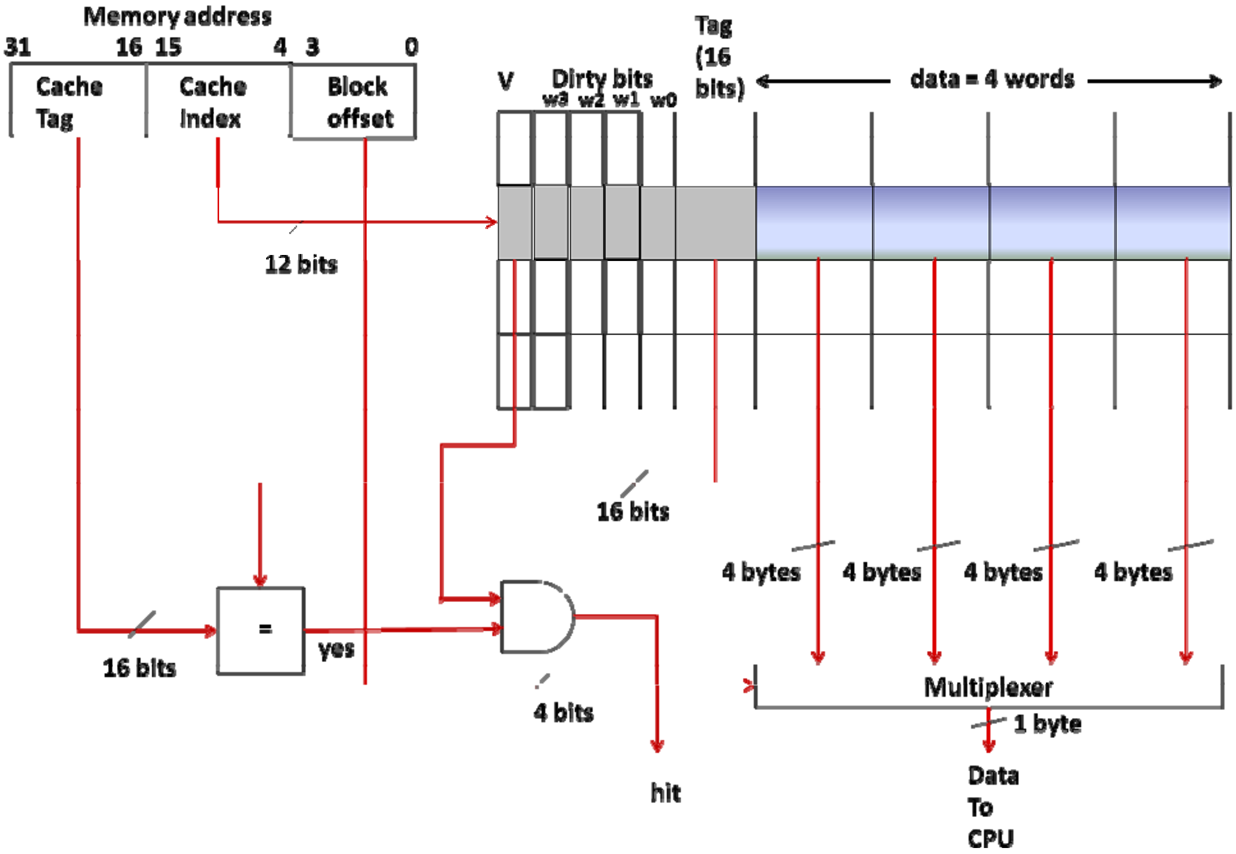
Referring to formula (7),
Block size
B = 16 bytes,
therefore
$b = log_2 16$ = 4 bits

We need 4 bits (bits 3-0 of the memory address) for the block offset.

Referring to formula (8), number of cache lines
$L =$ *64 Kbytes/16 bytes = 4096*
Referring to formula (9), number of index bits
$n = log_2 L = log_2 4096 = 12$

Referring to formula (10), number of tag bits
$t = a - (n+b) = 32 - (12+4) = 16$

Therefore, we need 12 bits (bits 15-4 of the memory address) for the index. The remaining 16 bits (bits 31-16 of the memory address) constitute the tag. The following figure shows how the CPU interprets the memory address.



(b) Compute the total amount of storage for implementing the cache (i.e. actual data plus the meta-data).

Answer:

Each cache line will have:

Data	16 bytes x 8 bits/byte = 128 bits	
Valid bit	1 bit	
Dirty bits	4 bits	(1 for each word)
Tag	16 bits	
	--------	
	149 bits	

Total amount of space for the cache = 149 x 4096 cache lines = **610,304 bits**

The space requirement for the meta-data = total space – actual data = 610,304 – 64Kbytes
= 610,304 – 524,288
= 86,016

The space overhead = meta-data/total space = 86,016/610,304 = **14%**

Recall from Example 2 that the space overhead for the same cache size but with a block size of 4 bytes was 35%. In other words, increased block size decreases the meta-data requirement and hence reduces the space overhead.

9.10.1 Performance implications of increased blocksize

It is instructive to understand the impact of increasing the block size on cache performance. Exploiting spatial locality is the main intent in having an increased block size. Therefore, for a *given total size of the cache,* we would expect the miss-rate to decrease with increasing block size. In the limit, we can simply have one cache block whose size is equal to the total size of the cache. Thus, it is interesting to ask two questions related to the block size:

1. Will the miss-rate keep decreasing forever?
2. Will the overall performance of the processor go up as we increase the block size?

The answer is "No" for the first question. In fact, the miss-rate may decrease up to an inflection point and start increasing thereafter. The *working set* notion that we introduced in Chapter 8 is the reason for this behavior. Recall that the working set of a program changes over time. A cache block contains a contiguous set of memory locations. However, if the working set of the program changes then the large block size results in increasing the miss-rate (see Figure 9.23).

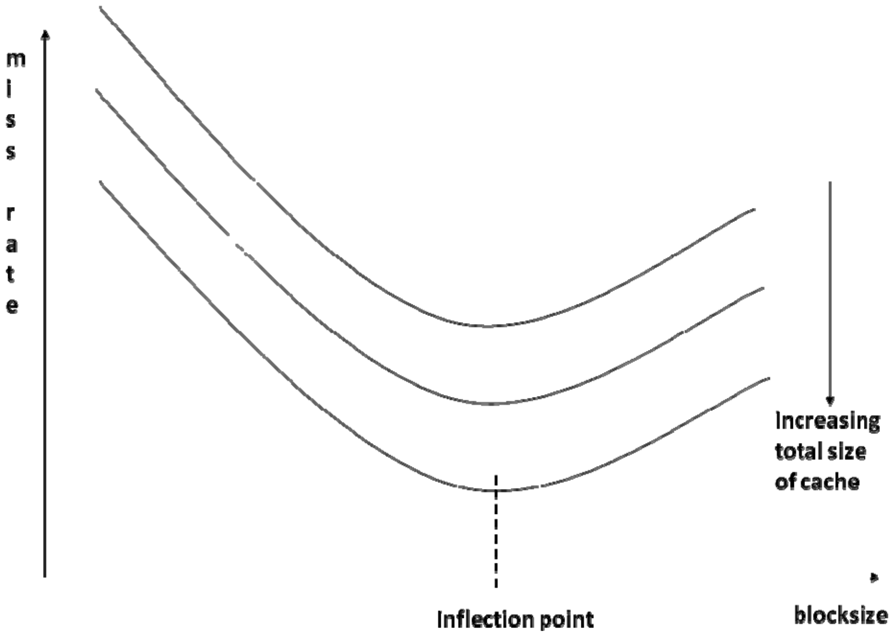


Figure 9.23: Expected behavior of miss-rate as a function of blocksize

The answer to the second question is a little more complicated. As is seen in Figure 9.23, the miss-rate does decrease with increasing block size up to an inflection point for a given cache size. This would translate to the processor incurring fewer memory stalls and thus improve the performance. However, beyond the inflection point, the processor starts incurring more memory stalls thus degrading the performance. However, as it turns out, the downturn in processor performance may start happening much sooner than the inflection point in Figure 9.23. While the miss-rate decreases with increasing block size up to an inflection point for a given cache size, the increased block size could negatively interact with the miss penalty. Recall that reducing the execution time of a program is the primary objective. This objective translates to reducing the number of stalls introduced in the pipeline by the memory hierarchy. The larger the block size, the larger the time penalty for the transfer from the memory to the cache on misses, thus increasing the memory stalls. We will discuss techniques to reduce the miss penalty shortly. The main point to note is that since the design parameters (block size and miss penalty) are inter-related, optimizing for one may not always result in the best overall performance. Put in another way, just focusing on the miss-rate as the output metric to optimize may lead to erroneous cache design decisions.

In pipelined processor design (Chapter 5), we understood the difference between latency of individual instructions and the throughput of the processor as a whole. Analogously, in cache design, choice of block size affects the balance between latency for a single instruction (that incurred the miss) and throughput for the program as a whole, by reducing the potential misses for other later instructions that might benefit from the exploitation of spatial locality. A real life analogy is income tax. An individual incurs a personal loss of revenue by the taxes (akin to latency), but helps the society as a whole to become better (akin to throughput). Of course, beyond a point the balance tips. Depending on your political orientation the tipping point may be sooner than later.

Modern day processors have a lot more going on that compound these issues. The micro-architecture of the processor, i.e., the implementation details of an ISA is both complex and fascinating. Instructions do not necessarily execute in program order so long as the semantics of the program is preserved. A cache miss does not necessarily block the processor; such caches are referred to as lock-up free caches. These and other micro-level optimizations interact with one another in unpredictable ways, and are of course sensitive to the workload executing on the processor as well. The simple fact is that a doubling of the cache block size requires doubling the amount of data that needs to be moved into the cache. The memory system cannot usually keep up with this requirement. The upshot is that we see a dip in performance well before the inflection point in Figure 9.23.

9.11 Flexible placement

In a direct-mapped cache, there is a one-to-one mapping from a memory address to a cache index. Because of this rigid mapping, the cache is unable to place a new memory location in a currently unoccupied slot in the cache. Due to the nature of program behavior, this rigidity hurts performance. Figure 9.24 illustrates the inefficient use of a direct-mapped cache as a program changes its working set.

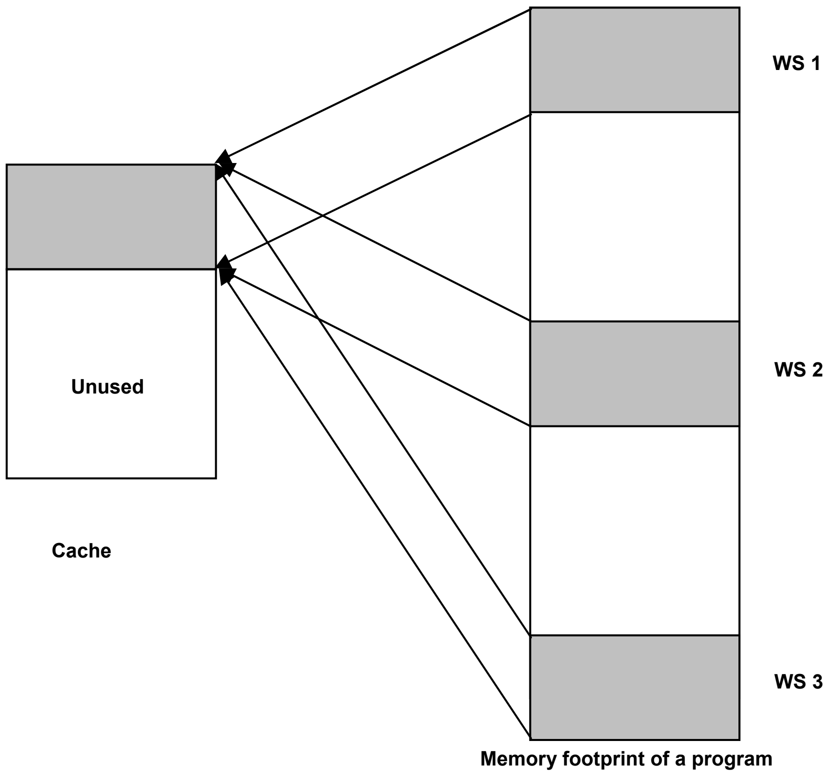


Figure 9.24: Different working sets of a program occupying the same portion of a direct-mapped cache

The program frequently flips among the three working sets (WS1, WS2, and WS3) in the course of its execution that happen to map exactly to the same region of the cache as shown in the Figure. Assume that each working set is only a third of the total cache size. Therefore, in principle there is sufficient space in the cache to hold all three working sets. Yet, due to the rigid mapping, the working sets displace one another from the cache resulting in poor performance. Ideally, we would want all three working sets of the program to reside in the cache so that there will be no more misses beyond the compulsory ones. Let us investigate how we can work towards achieving this ideal. Essentially, the design of the cache should take into account that the locality of a program may change over time. We will first discuss an extremely flexible placement that avoids this problem altogether.

9.11.1 Fully associative cache

In this set up, there is no unique mapping from a memory block to a cache block. Thus, a cache block can host any memory block. Therefore, by design, *compulsory* and *capacity misses* are the only kind of misses encountered with this organization. The cache interprets a memory address presented by the CPU as shown in Figure 9.25. Note that there is no cache index in this interpretation.

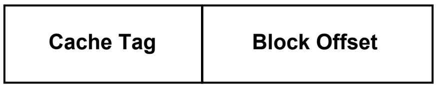


Figure 9.25: Memory address interpretation for a fully associative cache

This is because, in the absence of a unique mapping, a memory block could reside in any of the cache blocks. Thus with this organization, to perform a look up, the cache has to search through all the entries to see if there is a match between the cache tag in the memory address and the tags in any of the valid entries. One possibility is to search through each of the cache entries sequentially. This is untenable from the point of view of processor performance. Therefore, the hardware comprises replicated comparators, one for each entry, so that all the tag comparisons happen in parallel to determine a hit (Figure 9.26).

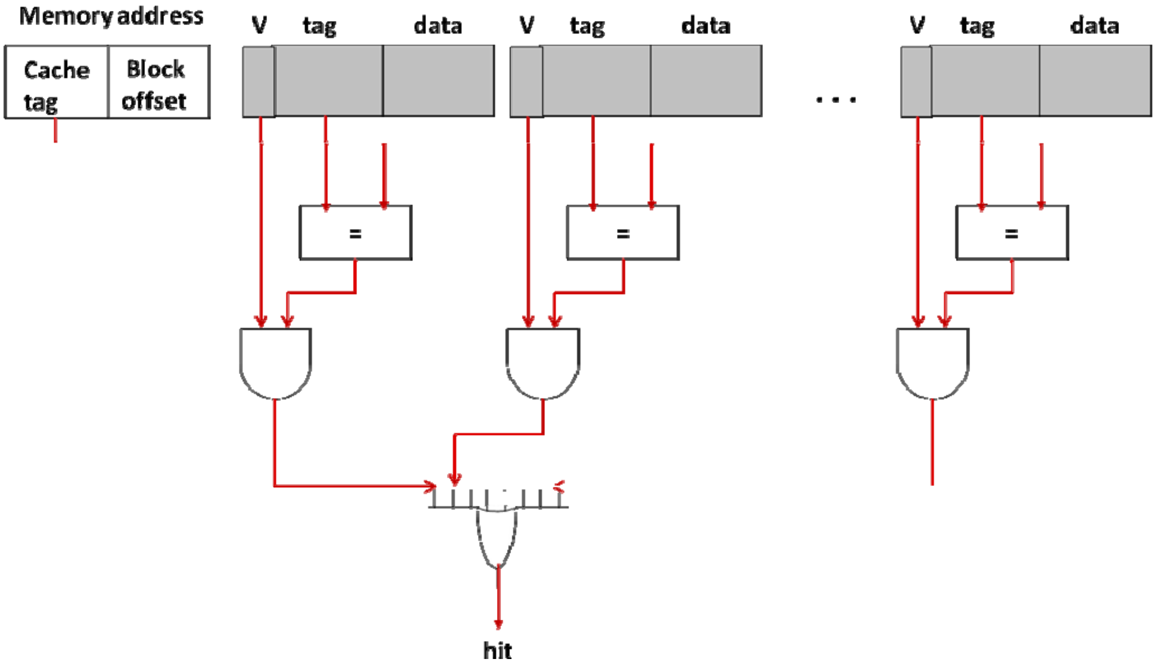
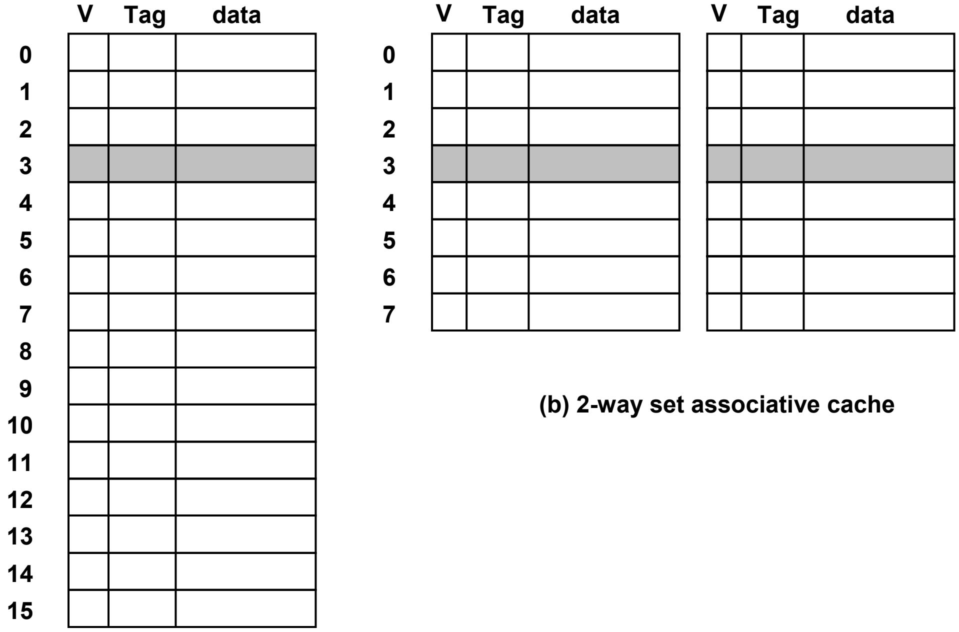


Figure 9.26: Parallel tag matching hardware for a fully associative cache. Each block of the cache represents an independent cache. Tag matching has to happen in parallel for all the caches to determine hit or a miss. Thus, a given memory block could be present in any of the cache blocks (shown by the shading). Upon a hit, the data from the block that successfully matched the memory address is sent to the CPU.

The complexity of the parallel tag matching hardware makes a fully associative cache infeasible for any reasonable sized cache. At first glance, due to its flexibility, it might appear that a fully associative cache could serve as a gold standard for the best possible miss-rate for a given workload and a given cache size. Unfortunately, such is not the case. A cache being a high-speed precious resource will be mostly operating at near full utilization. Thus, misses in the cache will inevitably result in replacement of something that is already present in the cache. The choice of a replacement candidate has a huge impact on the miss-rates experienced in a cache (since we don't know what memory accesses will be made in the future), and may overshadow the flexibility advantage of a fully associative cache for placement of the missing cache line. We will shortly see more details on cache replacement (see Section 9.14). Suffice it to say at this point that for all of these reasons, a fully associative cache is seldom used in practice except in very special circumstances. Translation look-aside buffer (TLB), introduced in Chapter 8 lends itself to an implementation as a fully associative organization due to its small size. Fully associative derives its name from the fact that a memory block can be *associated* with *any* cache block.

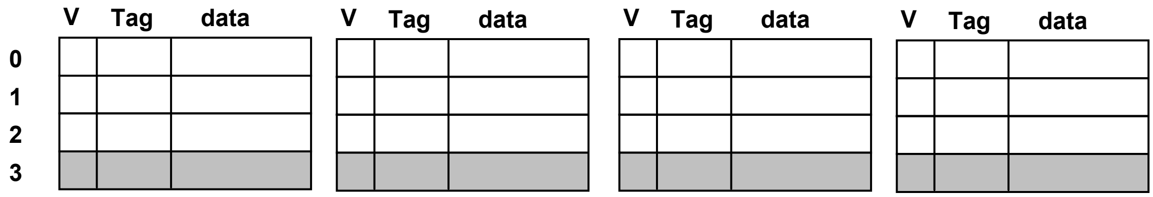
9.11.2 Set associative cache

An intermediate organization between direct-mapped and fully associative is a *set associative cache*. This organization derives its name from the fact that a memory block can be *associated* with a *set* of cache blocks. For example, a 2-way set associative cache, gives a memory block two possible homes in the cache. Similarly, a 4-way set associative cache gives a memory block one of four possible homes in the cache. The *degree of associativity* is defined as the number of homes that a given memory block has in the cache. It is 2 for a 2-way set associative cache; and 4 for a 4-way set associative cache; and so on.



(b) 2-way set associative cache

(a) Direct-mapped cache



(c) 4-way set associative cache

Figure 9.27: Three different organizations of 16 cache blocks. A given memory block could be present in one of several cache blocks depending on the degree of associativity. The shaded blocks in each of the three figures represent the possible homes for a given memory block.

One simple way to visualize a set associative cache is as multiple direct-mapped caches. To make this discussion concrete, consider a cache with a total size of 16 data blocks. We can organize these 16 blocks as a direct-mapped cache (Figure 9.27-(a)), or as a 2-way set associative cache (Figure 9.27-(b)), or as a 4-way set associative cache (Figure 9.27-(c)). Of course, with each data block there will be associated meta-data.
In the direct-mapped cache, given an index (say 3) there is exactly one spot in the cache that corresponds to this index value. The 2-way set associative cache has two spots in the cache (shaded spots in Figure 9.27-(b)) that correspond to the same index. The 4-way has four spots that correspond to the same index (shaded spots in Figure 9.27-(c)). With 16 data blocks total, the first organization requires a 4-bit index to lookup the cache, while the second organization requires a 3-bit index and the third a 2-bit index. Given a memory address, the cache simultaneously looks up all possible spots, referred to as a *set*, corresponding to that index value in the address for a match. The amount of tag matching hardware required for a set associative cache equals the degree of associativity.

Figure 9.28 shows the complete organization of a 4-way set associative cache, with a block size of 4 bytes.

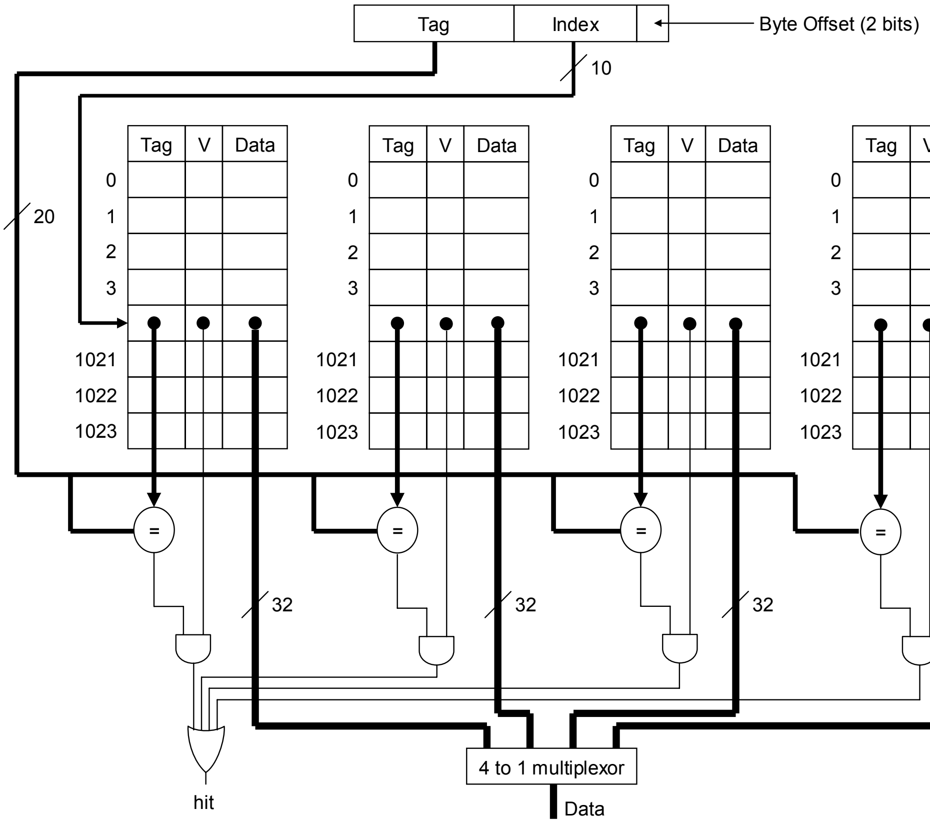


Figure 9.28: 4-way set associative cache organization

The cache interprets the memory address from the CPU as consisting of *tag*, *index*, and *block offset*, similar to a direct-mapped organization.

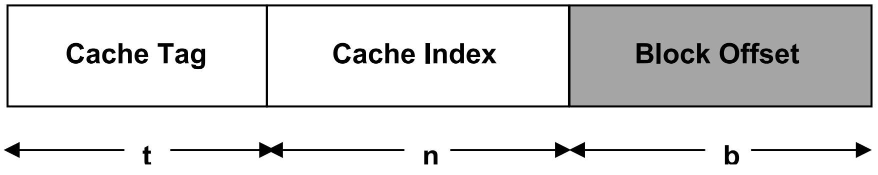


Figure 9.29: Interpreting the memory address generated by the CPU for a set-associative cache is no different from that for a direct-mapped cache

Let us understand how to break up the memory address into index and tag for cache lookup. The total size of the cache determines the number of index bits for a direct-mapped organization (*$log_2(S/B)$*, where *S* is the cache size and *B* the block size in bytes, respectively). For a set associative cache with the same total cache size, *$log_2(S/pB)$*

determines the number of index bits, where p is the degree of associativity. For example, a 4-way set associative cache with a total cache size of 16 data blocks requires $log_2(16/4)= 2$ *bits* (Figure 9.27-(c)).

9.11.3 Extremes of set associativity

Both direct-mapped and fully associative caches are special cases of the set-associative cache. Consider a cache with total size S bytes and block size B bytes. N, the total number of data blocks in the cache is given by S/B. Let us organize the data blocks into p parallel caches. We call this an *p-way set associative* cache. The cache has N/p cache lines (or sets[3]), each cache line having p data blocks. The cache requires p parallel hardware for tag comparisons.

What if $p = 1$? In this case, the organization becomes a direct-mapped cache. The cache has N cache lines with each set having 1 block.

What if $p = N$? In this case, the organization becomes a fully associative cache. The cache has 1 cache line of N blocks.

Given this insight, we will re-visit the formulae (7) through (10).

Total cache size = S bytes
Block size = B bytes
Memory address = a bits
Degree of associativity = p

Total number of data blocks in the cache:

$$\boxed{N = S/B} \qquad (11)$$

We replace equation (8) for the number or cache lines by the following general equation:

$$\boxed{L = S/pB = N/p} \qquad (12)$$

Equation (8) is a special case of (12) when $p = 1$. The number of index bits n is computed as log_2L (given by equation (9)).

As we said earlier, a fully associative cache is primarily used for TLBs. The degree of associativity usually depends on the level of the cache in the memory hierarchy. Either a direct-mapped cache or a 2-way set-associative cache is typically preferred for L1 (close to the CPU). Higher degree of associativity is the norm for deeper levels of the memory hierarchy. We will revisit these design consideration later in this chapter (please see Section 9.19).

[3] So now, we have four terminologies to refer to the same thing: cache line, cache block, cache entry, set.

Example 5:

Consider a 4-way set associative cache with a data size of 64 Kbytes. The CPU generates a 32-bit byte-addressable memory address. Each memory word contains 4 bytes. The block size is 16 bytes. The cache uses a write-through policy. The cache has one valid bit per data block.

a) How does the CPU interpret the memory address?

b) Compute the total amount of storage for implementing the cache (i.e. actual data plus the meta-data).

Answer:

a)

Number of bits in memory address:

$a = 32 \text{ bits}$

Since the block size is 16 bytes, using equation (7), the block offset

b = 4 bits (bits 0-3 of the memory address)

Since the cache is 4-way set associative (Figure 9.27-(c)),

$p = 4$

Number of cache lines (equation (12)):

$L = S/pB = 64 \text{ K bytes} / (4*16) \text{ bytes} = 1 \text{ K}$

Number of index bits (equation (9)):

$n = log_2L = log_21024 = 10 \text{ bits}$

Number of tag bits (equation (10)):

$t = a – (n+b) = 32 – (10+4) = 18 \text{ bits}$

Thus the memory address with 31 as msb, and 0 as the lsb is interpreted as:

Tag	**18 bits (31 to 114)**
Index	**10 bits (13 to 4)**
Block offset	**4 bits (3 to 0)**

b)

A block in each of the 4 parallel caches contains (data plus meta-data):

Data: 16 x 8 bits	= 128 bits (16 bytes in one cache block)
Valid bit	= 1 bit (1 valid bit per block)
Tag	= 18 bits
Total	= 147 bits

Each line of the cache contains 4 of these blocks = 147 * 4 = 588 bits

With 1K such cache lines in the entire cache, **the total size of the cache** =
588 bits/cache line * 1024 cache lines = **602,112 bits**

Example 6:

Consider a 4-way set-associative cache.

- Total data size of cache = 256KB.
- CPU generates 32-bit byte-addressable memory addresses.
- Each memory word consists of 4 bytes.
- The cache block size is 32 bytes.
- The cache has one valid bit per cache line.
- The cache uses write-back policy with one dirty bit per word.

a) Show how the CPU interprets the memory address (i.e., which bits are used as the cache index, which bits are used as the tag, and which bits are used as the offset into the block?).

b) Compute the total size of the cache (including data and metadata).

Answer:

a) Using reasoning similar to example 5,

Tag	**16 bits (31 to 16)**
Index	**11 bits (15 to 5)**
Block offset	**5 bits (0 to 4)**

b)

A block in each of the 4 parallel caches contains:

Data: 32 x 8 bits	=	256 bits (32 bytes in one cache block)
Valid bit	=	1 bit (1 valid bit per block)
Dirty bits: 8 x 1 bit	=	8 bits (1 dirty bit per word of 4 bytes)
Tag	=	16 bits
Total	=	281 bits

Each cache line contains 4 of these blocks = 281 * 4 = 1124 bits

With 2K such cache lines in the entire cache, **the total size of the cache** =
1124 bits/cache line * 2048 cache lines = **281 KBytes**

9.12 Instruction and Data caches

In the processor pipeline (please see Figure 9.15), we show two caches, one in the IF stage and one in the MEM stage. Ostensibly, the former is for instructions and the latter is for data. Some programs may benefit from a larger instruction cache, while some other programs may benefit from a larger data cache.

It is tempting to combine the two caches and make one single large *unified* cache. Certainly, that will increase the hit rate for most programs for a given cache size irrespective of their access patterns for instructions and data.

However, there is a down side to combining. We know that the IF stage accesses the I-cache in every cycle. The D-cache comes into play only for memory reference instructions (loads/stores). The contention for a unified cache could result in a structural hazard and reduce the pipeline efficiency. Empirical studies have shown that the detrimental effect of the structural hazard posed by a unified cache reduces the overall pipeline efficiency despite the increase in hit rate.

There are hardware techniques (such as multiple read ports to the caches) to get around the structural hazard posed by a unified I- and D-cache. However, these techniques increase the complexity of the processor, which in turn would affect the clock cycle time of the pipeline. Recall that caches cater to the twin requirements of speed of access and reducing miss rate. As we mentioned right at the outset (see Section 9.4), the primary design consideration for the L1 cache is matching the hit time to the processor clock cycle time. This suggests avoiding unnecessary design complexity for the L1 cache to keep the hit time low. In addition, due to the differences in access patterns for instructions and data, the design considerations such as associativity may be different for I- and D-caches. Further, thanks to the increased chip density, it is now possible to have large enough separate I- and D-caches that the increase in miss rate due to splitting the caches is negligible. Lastly, I-cache does not have to support writes, which makes them simpler and faster. For these reasons, it is usual to have a split I- and D-caches on-chip. However, since the primary design consideration for L2 cache is reducing the miss rate, it is usual to have a unified L2 cache.

9.13 Reducing miss penalty

The miss penalty is the service time for the data transfer from memory to cache on misses. As we observed earlier, the penalty may be different for reads and writes, and the penalty depends on other hardware in the datapath such as *write buffers* that allow overlapping computation with the memory transfer.

Usually, the main memory system design accounts for the cache organization. In particular, the design supports block transfer to and from the CPU to populate the cache. The memory bus that connects the main memory to the CPU plays a crucial role in determining the miss penalty. We refer to *bus cycle time* as the time taken for each data transfer between the processor and memory. The amount of data transfer in each cycle between the processor and memory is referred to as the *memory bandwidth*. Memory bandwidth is a measure of the throughput available for information transfer between processor and memory. The bandwidth depends on the number of data wires connecting the processor and memory. Depending on the width of bus, the memory system may need multiple bus cycles to transfer a cache block. For example, if the block size is four words and the memory bus width is only one word, then it takes four bus cycles to complete the block transfer. As a first order, we may define the miss penalty as the total time (measured in CPU clock cycles) for transferring a cache block from the memory to the cache. However, the actual latency experienced by the processor for an individual miss may be less than this block transfer time. This is because the memory system may respond to a miss by providing the specific data that the processor missed on before sending the rest of the memory block that contains the missed reference.

Despite, such block transfer support in the memory system, increasing the block size beyond a certain point has other adverse effects. For example, if the processor incurs a read-miss on a location x, the cache subsystem initiates bringing in the whole block containing x, perhaps ensuring that the processor is served with x first. Depending on the bandwidth available between the processor and memory, the memory system may be busy transferring the rest of the block for several subsequent bus cycles. In the meanwhile, the processor may incur a second miss for another memory location y in a different cache block. Now, the memory system cannot service the miss immediately since it is busy completing the block transfer for the earlier miss on x. This is the reason we observed earlier in Section 9.10 that it is not sufficient to focus just on the miss-rate as the output metric to decide on cache block size. This is the classic tension between latency and throughput that surfaces in the design of every computer subsystem. We have seen it in the context of processor in Chapter 5; we see it now in the context of memory system; later (Chapter 13), we will see it in the context of networks.

9.14 Cache replacement policy

In a direct-mapped cache, the placement policy pre-determines the candidate for replacement. Hence, there is no choice.

There is a possibility to exercise a choice in the case of a set associative or a fully associative cache. Recall that temporal locality consideration suggests using an LRU policy. For a fully associative cache, the cache chooses the replacement candidate applying the LRU policy across all the blocks. For a set associative cache, the cache's choice for a replacement candidate is limited to the set that will house the currently missing memory reference.

To keep track of LRU information, the cache needs additional meta-data. Figure 9.30 shows the hardware for keeping track of the LRU information for a 2-way set associative cache. Each set (or cache line) has one LRU bit associated with it.

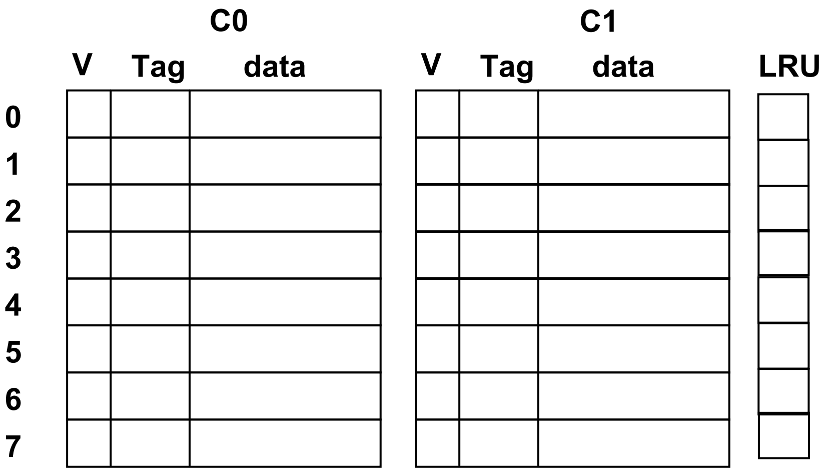


Figure 9.30: 1-bit LRU per set in a 2-way set associative cache

On each reference, the hardware sets the LRU bit for the set containing the currently accessed memory block. Assuming valid entries in both the blocks of a given set, the hardware sets the LRU bit to 0 or 1 depending on whether a reference hits in cache C0 or C1, respectively. The candidate for replacement is the block in C0 if the LRU bit is 1; it is the block in C1 if the LRU bit is 0.

The hardware needed for a 2-way set associative cache is pretty minimal but there is a time penalty since the hardware updates the LRU bit on every memory reference (affects the IF and MEM stages of the pipelined design).

The LRU hardware for higher degrees of associativity quickly becomes more complex. Suppose we label the 4 parallel caches C0, C1, C2, and C3 as shown in Figure 9.31. Building on the 2-way set associative cache, we could have a 2-bit field with each set. The encoding of the 2-bit field gives the block accessed most recently. Unfortunately, while this tells us the most recently used block, it does not tell us which block in a set is the least recently used. What we really need is an ordering vector as shown in Figure 9.31 for each set. For example, the ordering for set S0 shows that C2 was the least recently used and C1 the most recently used. That is, the decreasing time order of access to the blocks in S0 is: C1, followed by C3, followed by C0, followed by C2. Thus, at this time if a block has to be replaced from S0 then the LRU victim is the memory block in C2. On each reference, the CPU updates the ordering for the currently accessed set. Figure 9.32 shows the change in the LRU ordering vector to set S0 on a sequence of references that map to this set. Each row shows how the candidate for replacement changes based on the currently accessed block.

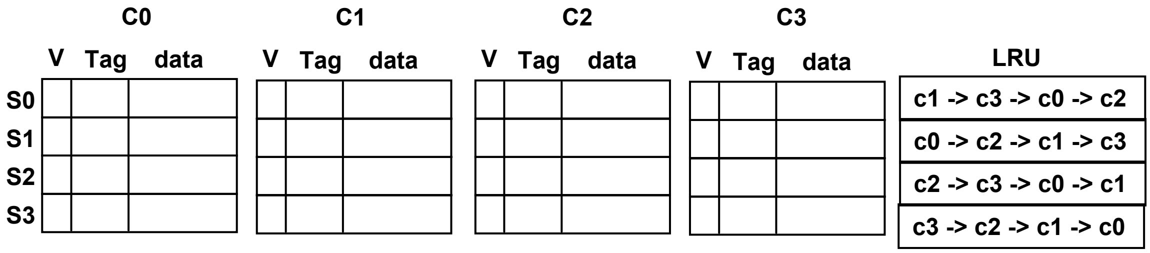


Figure 9.31: LRU information for a 4-way set associative cache

	LRU for set S0	
Access to C1	c1 -> c3 -> c0 -> c2	Replacement candidate: current block in C2
Access to C2	c2 -> c1 -> c3 -> c0	Replacement candidate: current block in C0
Access to C2	c2 -> c1 -> c3 -> c0	Replacement candidate: current block in C0
Access to C3	c3 -> c2 -> c1 -> c0	Replacement candidate: current block in C0
Access to C0	c0 -> c3 -> c2 -> c1	Replacement candidate: current block in C1

Figure 9.32: Change in the LRU vector for set S0 on a sequence of references that map to that set

Let us understand what it takes to implement this scheme in hardware. The number of possibilities for the ordering vector with 4 parallel caches is 4! = 24. Therefore, we need a 5-bit counter to encode the 24 possible states of the ordering vector. An 8-way set associative cache requires a counter big enough to encode 8 factorial states. Each set needs to maintain such a counter with the associated fine state machine that implements the state changes as illustrated in the above example. One can see that the hardware complexity of implementing such a scheme increases rapidly with the degree of associativity and the number of lines in the cache. There are other simpler encodings that are approximations to this true LRU scheme. There is ample empirical evidence that less complex replacement policies may in fact perform better (i.e., result in better miss-rates) than true LRU for many real code sequences.

9.15 Recapping Types of Misses

We identified three categories of misses in the cache: *compulsory*, *capacity*, and *conflict*. As the name suggests, compulsory misses result because the program is accessing a given memory location for the first time during execution. Naturally, the location is not in the cache and a miss is inevitable. Using the analogy of an automobile engine being cold or warm (at the time of starting), it is customary to refer to such misses as *cold* misses.

On the other hand, consider the situation when the CPU incurs a miss on a memory location X that used to be in the cache previously[4]. This can be due to one of two reasons: the cache is full at the time of the miss, so there is no choice except to evict some memory location to make room for X. This is what is referred to as a *capacity*

[4] We do not know why X was evicted in the first place but it is immaterial from the point of view of characterizing the current miss.

miss. On the other hand, it is conceivable that the cache is not full but the mapping strategy forces X to be brought into a cache line that is currently occupied by some other memory location. This is what is referred to as a *conflict* miss. By definition, we cannot have a conflict miss in a fully associative cache, since a memory location can be placed anywhere. Therefore, in a fully associative cache the only kinds of misses that are possible are compulsory and capacity.

Sometimes it can be confusing how to categorize the misses. Consider a fully associative cache. Let us say, the CPU accesses a memory location X for the first time. Let the cache be full at this time. We have a cache miss on X. Is this a capacity or a compulsory miss? We can safely say it is both. Therefore, it is fair to categorize this miss as either compulsory or capacity or both.

Note that a capacity miss can happen in any of direct-mapped, set-associative, or fully associative caches. For example, consider a direct-mapped cache with 4 cache lines. Let the cache be initially empty and each line hold exactly 1 memory word. Consider the following sequence of memory accesses by the CPU:

0, 4, 0, 1, 2, 3, 5, 4

We will denote the memory word at the above addresses as m0, m1, …, m5, respectively. The first access (m0) is a compulsory miss. The second access (m4) is also a compulsory miss and it will result in evicting m0 from the cache, due to the direct-mapped organization. The next access is once again to m0, which will be a miss. This is definitely a conflict miss since m0 was evicted earlier by bringing in m4, in spite of the fact that the cache had other empty lines. Continuing with the accesses, m1, m2, and m3 all incur compulsory misses.

Next, we come to memory access for m5. This was never in the cache earlier, so the resulting miss for m5 may be considered a compulsory miss. However, the cache is also full at this time with m0, m1, m2, and m3. Therefore, one could argue that this miss is a capacity miss. Thus, it is fair to categorize this miss as either a compulsory miss or a capacity miss or both.

Finally, we come to the memory access for m4 again. This could be considered a conflict miss since m4 used to be in the cache earlier. However, at this point the cache is full with m0, m5, m2, and m3. Therefore, we could also argue that this miss is a capacity miss. Thus, it is fair to categorize this miss as either a conflict miss or a capacity miss or both.

Compulsory misses are unavoidable. Therefore, this type of a miss dominates the other kinds of misses. In other words, if a miss can be characterized as either compulsory or something else, we will choose compulsory as its characterization. Once the cache is full, independent of the organization, we will incur a miss on a memory location that is not currently in the cache. Therefore, a capacity miss dominates a conflict miss. In other words, if a miss can be characterized as either conflict or capacity, we will choose capacity as its characterization.

Example 7:

Given the following:

- Total number of blocks in a 2-way set associative cache: 8
- Assume LRU replacement policy

		C1	C2
	0		
index	1		
	2		
	3		

The processor makes the following 18 accesses to memory locations in the order shown below:

0, 1, 8, 0, 1, 16, 8, 8, 0, 5, 2, 1, 10, 3, 11, 10, 16, 8

With respect to the 2-way set associative cache shown above, show in a tabular form the cache which will host the memory location and the specific cache index where it will be hosted, and in case of a miss the type of miss (cold/compulsory, capacity, conflict).

Memory location	C1	C2	Hit/miss	Type of miss

Note:

- The caches are initially empty
- Upon a miss, if both the spots (C1 and C2) are free then the missing memory location will be brought into C1
- Capacity miss dominates over conflict miss
- Cold/compulsory miss dominates over capacity miss

Answer:

Memory location	C1	C2	Hit/miss	Type of miss
0	Index = 0		Miss	Cold/compulsory
1	Index = 1		Miss	Cold/compulsory
8		Index = 0	Miss	Cold/compulsory
0	Index = 0		Hit	
1	Index = 1		Hit	
16		Index = 0	Miss	Cold/compulsory
8	Index = 0		Miss	Conflict
8	Index = 0		Hit	
0		Index = 0	Miss	Conflict
5		Index = 1	Miss	Cold/compulsory
2	Index = 2		Miss	Cold/compulsory
1	Index = 1		Hit	
10		Index = 2	Miss	Cold/compulsory
3	Index = 3		Miss	Cold/compulsory
11		Index = 3	Miss	Cold/compulsory
10		Index = 2	Hit	
16	Index = 0		Miss	Capacity
8		Index = 0	Miss	Capacity

9.16 Integrating TLB and Caches

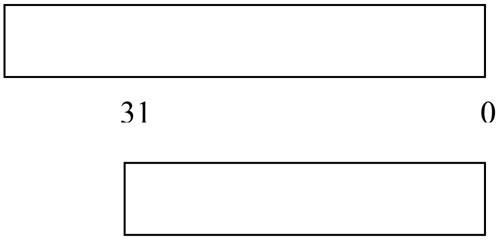
In Chapter 8, we introduced TLB, which is nothing but a cache of addresses. Recall that given a virtual page number (VPN) the TLB returns the physical frame number (PFN) if it exists in the TLB. The TLB is usually quite small for considerations of speed of access, and the space of virtual pages is quite large. Similar to the processor caches, there is the need for a mapping function for looking up the TLB given a VPN. The design considerations for a TLB are quite similar to those of processor caches, i.e., the TLB may be organized as a direct-mapped or set-associative structure. Depending on the specifics of the organization, the VPN is broken up into tag and index fields to enable the TLB look up. The following example brings home this point.

Example :

Given the following:

Virtual address 64 bits

Physical address 32 bits



Page size 4 K Bytes

A direct mapped TLB with 512 entries

(a) How many tag bits per entry are there in the TLB?
(b) How many bits are needed to store the page frame number in the TLB?

Answer:

(a)
With a pagesize of 4 KB the number of bits needed for page offset = 12
Therefore, the number of bits for VPN = 64-12 = 52
Number of index bits for looking up a direct mapped TLB of size 512 = 9
So, the number of tag in the TLB = 52-9 = **43 bits**

(b)
The number of bits needed to store the PFN in the TLB equals the size of the PFN.

With a pagesize of 4 KB, the PFN is 32-12 = **20 bits**

Now we can put the TLB, and the memory hierarchies together to get a total picture. Figure 9.33 shows the path of a memory access from the CPU (in either the IF or the MEM stages) that proceeds as follows:

- The CPU (IF or MEM stage of the pipeline) generates a virtual address (VA)
- The TLB does a virtual to physical address (PA) translation. If it is a hit in the TLB, then the pipeline continues without a hiccup. If it is a miss, then the pipeline freezes until the completion of the miss service.
- The stage uses the PA to look up the cache (I-Cache or the D-Cache, respectively). If it is a hit in the cache, then the pipeline continues without a hiccup. If it is a miss, then the pipeline freezes until the completion of the miss service.

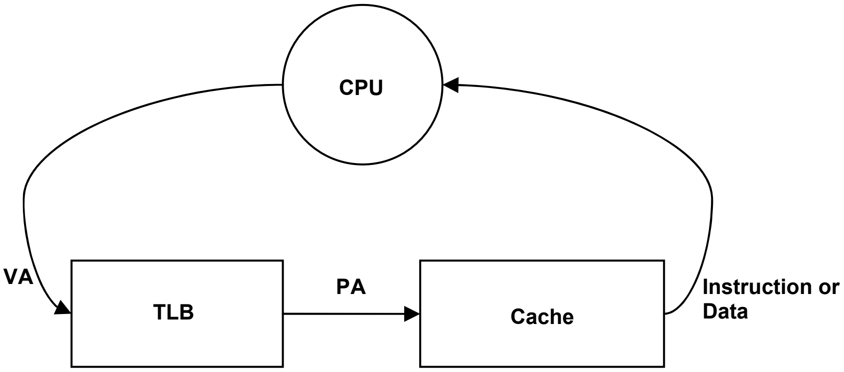


Figure 9.33: Path of a memory access

Note that the IF and MEM stages of the pipeline may access the TLB simultaneously. For this reason, most processors implement split TLB for instructions and data (I-TLB and D-TLB), respectively, so that two address translations can proceed in parallel. As shown in Figure 9.33, the TLB is in the critical path of determining the processor clock cycle time, since every memory access has to go first through the TLB and then the cache. Therefore, the TLB is small in size, typically 64 to 256 entries for each of I- and D-TLBs.

9.17 Cache controller

This hardware interfaces the processor to the cache internals and the rest of the memory system and is responsible for taking care of the following functions:

- Upon a request from the processor, it looks up the cache to determine hit or miss, serving the data up to the processor in case of a hit.
- Upon a miss, it initiates a bus transaction to read the missing block from the deeper levels of the memory hierarchy.
- Depending on the details of the memory bus, the requested data block may arrive asynchronously with respect to the request. In this case, the cache controller receives the block and places it in the appropriate spot in the cache.
- As we will see in the next chapter, the controller provides the ability for the processor to specify certain regions of the memory as "uncachable." The need for this will become apparent when we deal with interfacing I/O devices to the processor (Chapter 10).

Example 8:

Consider the following memory hierarchy:

- A 128 entry fully associative TLB split into 2 halves; one-half for user processes and the other half for the kernel. The TLB has an access time of 1 cycle. The hit rate for the TLB is 95%. A miss results in a main memory access to complete the address translation.
- An L1 cache with a 1 cycle access time, and 99% hit rate.
- An L2 cache with a 4 cycle access time, and a 90% hit rate.
- An L3 cache with a 10 cycle access time, and a 70% hit rate.

- A physical memory with a 100 cycle access time.

Compute the average memory access time for this memory hierarchy. Note that the page table entry may itself be in the cache.

Answer:

Recall from Section 9.4:

$$\text{EMAT}_i = T_i + m_i * \text{EMAT}_{i+1}$$

$\text{EMAT}_{\text{physical memory}}$ = 100 cycles.
$\text{EMAT}_{\text{L3}} = 10 + (1 – 0.7) * 100 = 40$ cycles
$\text{EMAT}_{\text{L2}} = (4) + (1 – 0.9) * (40) = 8$ cycles
$\text{EMAT}_{\text{L1}} = (1) + (1 – 0.99) * (8) = 1.08$ cycles
$\text{EMAT}_{\text{TLB}} = (1) + (1 – 0.95) * (1.08) = 1.054$ cycles
$\text{EMAT}_{\text{Hierarchy}} = \text{EMAT}_{\text{TLB}} + \text{EMAT}_{\text{L1}} = 1.054 + 1.08 = 2.134$ cycles.

9.18 Virtually indexed physically tagged cache

Looking at Figure 9.33, every memory access goes through the TLB and then the cache. The TLB helps in avoiding a trip to the main memory for address translation. However, TLB lookup is in the critical path of the CPU. What this means is that the virtual to physical address translation has to be done first before we can look up the cache to see if the memory location is present in the cache. In other words, due to the sequential nature of the TLB lookup followed by the cache lookup, the CPU incurs a significant delay before it can be determined if a memory access is a hit or a miss in the cache. It would be nice if accessing the TLB for address translation, and looking up the cache for the memory word can be done in parallel. In other words, we would like to get the address translation "out of the way" of the CPU access to the cache. That is, we want to take the CPU address and lookup the cache bypassing the TLB. At first, it seems as though this is impossible since we need the physical address to look up the cache.

Let us re-examine the virtual address (Figure 9.34).

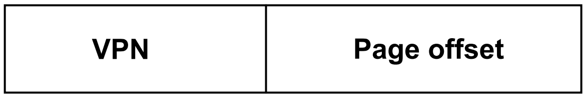


Figure 9.34: Virtual address

The VPN changes to PFN because of the address translation. However, the page offset part of the virtual address remains unchanged.

A direct-mapped or set associative cache derives the index for lookup from the least significant bits of the physical address (see Figure 9.11).
This gives us an idea. If we derive the cache index from the unchanging part of the virtual address (i.e. the page offset part), then we can lookup the cache in parallel with the TLB lookup. We refer to such an organization as a *virtually indexed physically*

tagged cache (Figure 9.35). The cache uses virtual index but derives the tag from the physical address.

With a little bit of thought it is not difficult to figure out the limitation with this scheme. The unchanging part of the virtual address limits the size of the cache. For example, if the page size is 8 Kbytes then the cache index is at most 13 bits, and typically less since part of the least significant bits specify the block offset. Increasing the set associativity allows increasing the size of the cache despite this restriction. However, we know there is a limit to increasing the associativity due to the corresponding increase in hardware complexity.

A partnership between software and hardware helps in eliminating this restriction. Although the hardware does the address translation, the memory manager is the software entity that sets up the VPN to PFN mapping. The memory manager uses a technique called *page coloring* to guarantee that more of the virtual address bits will remain unchanged by the translation process by choosing the VPN to PFN mapping carefully (see Example 7). Page coloring allows the processor to have a larger virtually indexed physically tagged cache independent of the page size.

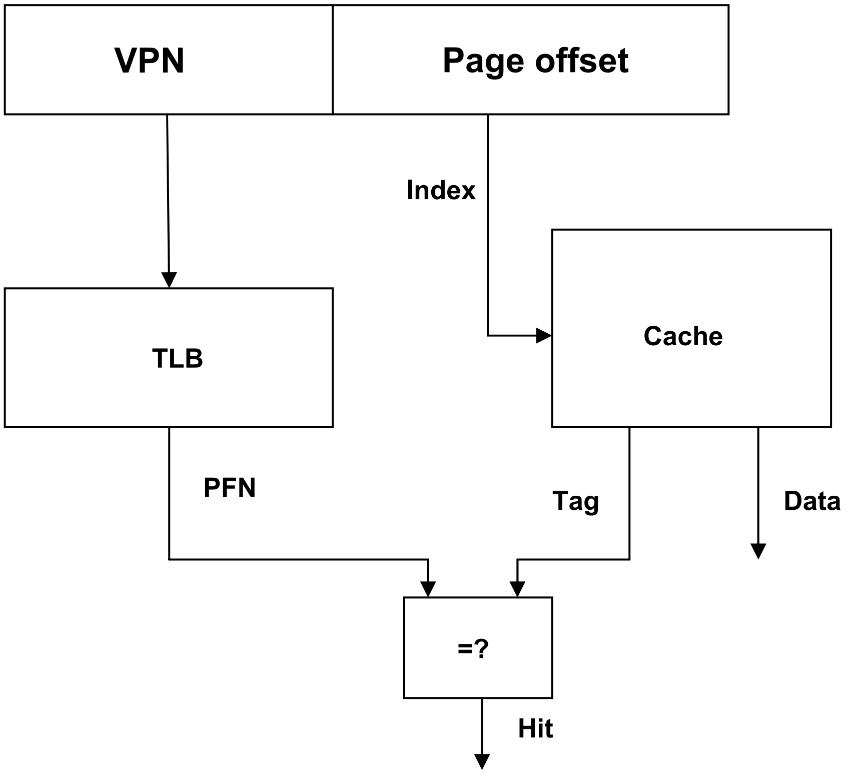


Figure 9.35: A virtually indexed physically tagged cache

Another way to get the translation out of the way is to use *virtually tagged* caches. In this case, the cache uses virtual index and tag. The reader should ponder on the challenges such an organization introduces. Such a discussion is beyond the scope of this book[5].

Example 9:

Consider a virtually indexed physically tagged cache:

1. Virtual address is 32 bits
2. Page size 8 Kbytes
3. Details of the cache
   - 4-way set associative
   - Total Size = 512 Kbytes
   - Block size = 64 bytes

The memory manager uses page coloring to allow the large cache size.

1. How many bits of the virtual address should remain unchanged for this memory hierarchy to work?
2. Pictorially show the address translation and cache lookup, labeling the parts of the virtual and physical addresses used in them.

Answer:

An 8 Kbytes page size means the page offset will be 13 bits leaving 19 bits for the VPN.

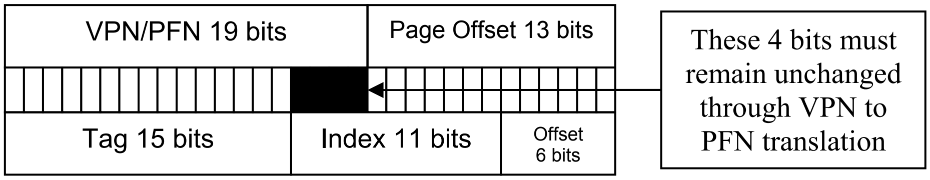
The cache has 4 blocks/set x 64 bytes/block = 256 bytes/set and

512 Kbytes total cache size / 256 bytes/set = 2 K sets which implies 11 bits for the index.

Therefore the cache breakdown of a memory reference is

Tag, 15 bits; Index, 11 bits; Offset 6 bits

Thus, the memory management software must assign frames to pages in such a way that the least significant 4 bits of the VPN must equal to the least significant 4 bits of the PFN.



[5] Please see advanced computer architecture textbooks for a more complete treatment of this topic. Hennessy and Patterson, Computer Architecture: A Quantitative Approach, Morgan-Kauffman publishers.

9.19 Recap of Cache Design Considerations

Thus far, we have introduced several concepts and it will be useful to enumerate them before we look at main memory:

1. Principles of spatial and temporal locality (Section 9.2)
2. Hit, miss, hit rate, miss rate, cycle time, hit time, miss penalty (Section 9.3)
3. Multilevel caches and design considerations thereof (Section 9.4)
4. Direct mapped caches (Section 9.6)
5. Cache read/write algorithms (Section 9.8)
6. Spatial locality and blocksize (Section 9.10)
7. Fully- and set-associative caches (Section 9.11)
8. Considerations for I- and D-caches (Section 9.12)
9. Cache replacement policy (Section 9.14)
10. Types of misses (Section 9.15)
11. TLB and caches (Section 9.16)
12. Cache controller (Section 9.17)
13. Virtually indexed physically tagged caches (Section 9.18)

Modern processors have on-chip TLB, L1, and L2 caches. The primary design consideration for TLB and L1 is reducing the hit time. Both of them use a simple design consistent with this design consideration. TLB is quite often a small fully associative cache of address translations with sizes in the range of 64 to 256 entries. It is usually split into a system portion that survives context switches, and a user portion that is flushed upon a context switch. Some processors provide process tags in the TLB entries to avoid flushing at context switch time. L1 is optimized for speed of access. Usually it is split into I- and D-caches employing a small degree of associativity (usually 2), and the size is relatively small compared to the higher levels of the memory hierarchy (typically less than 64KB for each of I- and D-caches in processors circa 2008). L2 is optimized for reducing the miss rate. Usually it is a unified I-D cache, employing a larger associativity (4-way and 8-way are quite common; some are even 16-way). They may even employ a larger block size than L1 to reduce the miss rate. L2 cache size for processors circa 2008 is in the range of several hundreds of Kilobytes to a few Megabytes. Most modern processors also provide a larger off-chip L3 cache, which may have similar design considerations as L2 but even larger in size (several tens of Megabytes in processors circa 2008).

9.20 Main memory design considerations

The design of the processor-memory bus and the organization of the physical memory play a crucial part in the performance of the memory hierarchy. As we mentioned earlier, the implementation of the physical memory uses DRAM technology that has nearly a 100:1 speed differential compared to the CPU. The design may warrant several trips to the physical memory on a cache miss depending on the width of the processor-memory bus and the cache block size.

In the spirit of the textbook in undertaking joint discovery of interesting concepts, we will start out by presenting very simple ideas for organizing the main memory system. At the outset, we want the reader to understand that these ideas are far removed from the sophisticated memory system that one might see inside a modern box today. We will end this discussion with a look at the state-of-the-art in memory system design.

First, let us consider three different memory bus organizations and the corresponding miss penalties. For the purposes of this discussion, we will assume that the CPU generates 32-bit addresses and data; the cache block size is 4 words of 32 bits each.

9.20.1 Simple main memory

Figure 9.36 shows the organization of a simple memory system. It entertains a block read request to service cache misses. The CPU simply sends the block address to the physical memory. The physical memory internally computes the successive addresses of the block, retrieves the corresponding words from the DRAM, and sends them one after the other back to the CPU.

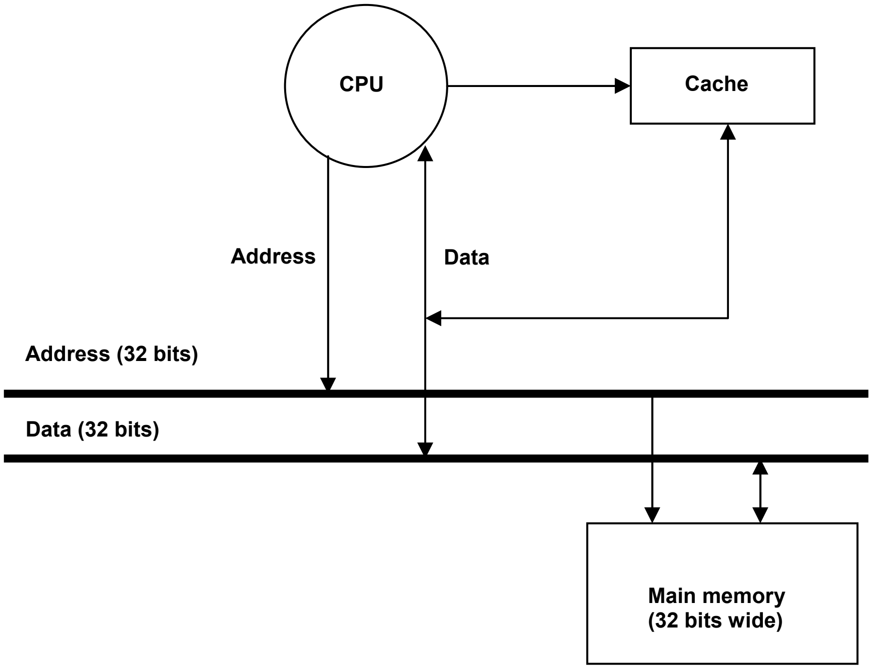


Figure 9.36: A simple memory system

Example 10:

Assume that the DRAM access time is 70 cycles. Assume that the bus cycle time for address or data transfer from/to the CPU/memory is 4 cycles. Compute the block transfer time for a block size of 4 words. Assume all 4 words are first retrieved from the DRAM before the data transfer to the CPU.

Answer:

Address from CPU to memory = 4 cycles
DRAM access time = 70 * 4 = 280 cycles (4 successive words)

Data transfer from memory to CPU = 4 * 4 = 16 cycles (4 words)

Total block transfer time = **300 cycles**

9.20.2 Main memory and bus to match cache block size

To cut down on the miss penalty, we could organize the processor-memory bus and the physical memory to match the block size. Figure 9.37 shows such an organization. This organization transfers the block from the memory to CPU in a single bus cycle and a single access to the DRAM. All 4 words of a block form a single row of the DRAM and thus accessed with a single block address. However, this comes at a significant hardware complexity since we need a 128-bit wide data bus.

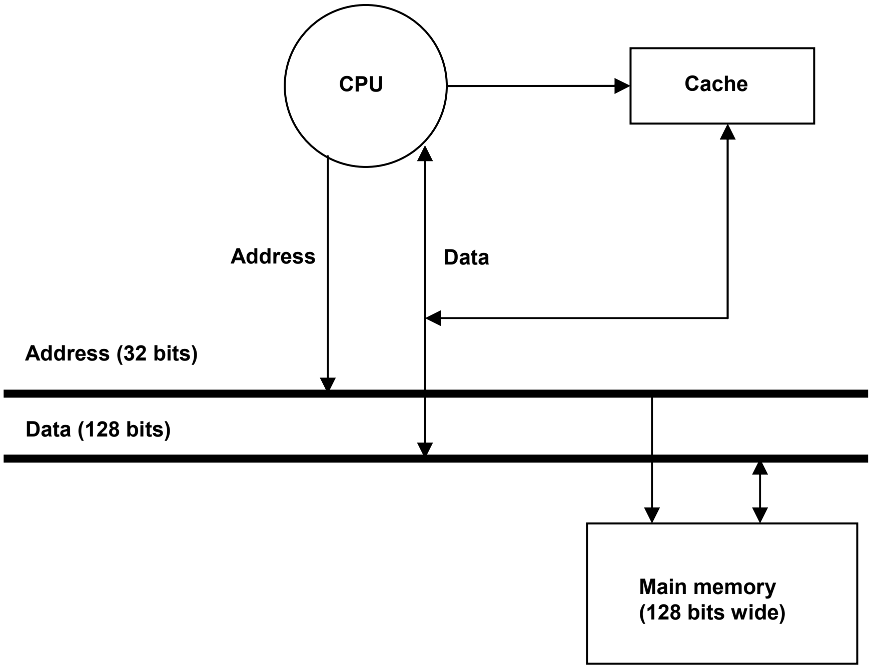


Figure 9.37: Main memory organization matching cache block size

Example 11:

Assume that the DRAM access time is 70 cycles. Assume that the bus cycle time for address or data transfer from/to the CPU/memory is 4 cycles. Compute the block transfer time with the memory system, wherein the bus width and the memory organization match the block size of 4 words.

Answer:

Address from CPU to memory = 4 cycles
DRAM access time = 70 cycles (all 4 words retrieved with a single DRAM access)
Data transfer from memory to CPU = 4 cycles

Total block transfer time = **78 cycles**

9.20.3 Interleaved memory

The increased bus width makes the previous design infeasible from a hardware standpoint. Fortunately, there is a way to achieve the performance advantage of the previous design with a little engineering ingenuity called *memory interleaving*. Figure 9.38 shows the organization of an interleaved memory system. The idea is to have multiple *banks* of memory. Each bank is responsible for providing a specific word of the cache block. For example, with a 4-word cache block size, we will have 4 banks labeled M0, M1, M2, and M3. M0 supplies word 0, M1 supplies word 1, M2 supplies word 3, and M3 supplies word 3. The CPU sends the block address that is simultaneously received by all the 4 banks. The banks work in parallel, each retrieving the word of the block that it is responsible for from their respective DRAM arrays. Once retrieved, the banks take turn sending the data back to the CPU using a standard bus similar to the first simple main memory organization.

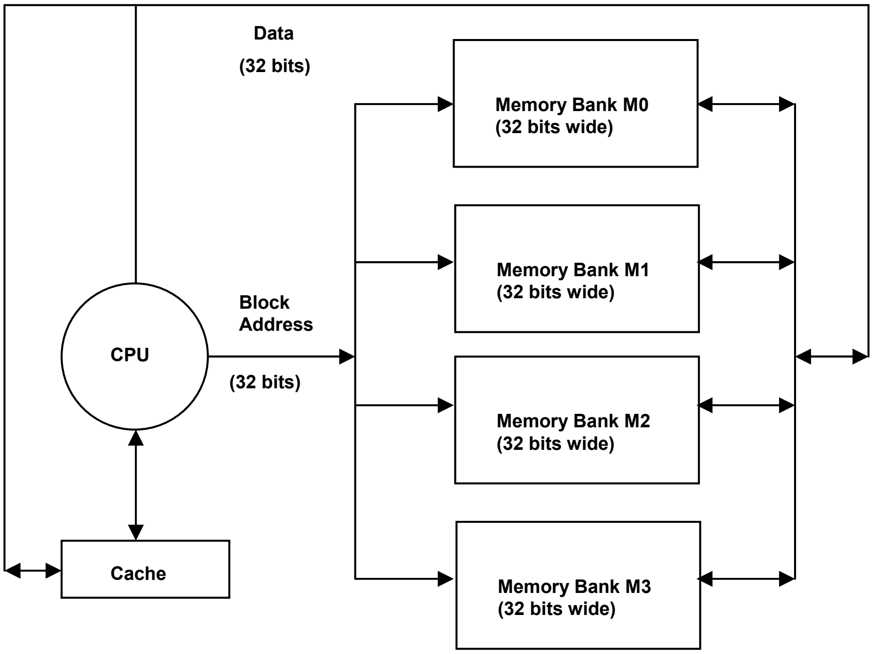


Figure 9.38: An interleaved main memory

The interleaved memory system focuses on the fact that DRAM access is the most time intensive part of the processor-memory interaction. Thus, interleaved memory achieves most of the performance of the wide memory design without its hardware complexity.

Example 12:

Assume that the DRAM access time is 70 cycles. Assume that the bus cycle time for address or data transfer from/to the CPU/memory is 4 cycles. Compute the block transfer time with an interleaved memory system as shown in Figure 9.38.

Answer:

Address from CPU to memory = 4 cycles (all four memory banks receive the address simultaneously).

DRAM access time = 70 cycles (all 4 words retrieved in parallel by the 4 banks)
Data transfer from memory to CPU = 4 * 4 cycles (the memory banks take turns sending their respective data back to the CPU)

Total block transfer time = **90 cycles**

Keeping the processor busy is by far the most important consideration in the design of memory systems. This translates to getting the data from the memory upon a cache read miss to the processor as quickly as possible. Writing to an interleaved memory system need not be any different from a normal memory system. For the most part, the processor is shielded from the latency for writing to memory, with techniques such as write-buffers that we discussed in Section 9.7. However, many processor-memory buses support block write operations that work quite nicely with an interleaved memory system, especially for write-back of an entire cache block.

9.21 Elements of a modern main memory systems

Modern memory systems are a far cry from the simple ideas presented in the previous section. Interleaved memories are a relic of the past. With advances in technology, the conceptual idea of interleaving has made it inside the DRAM chips themselves. Let us explain how this works. Today, circa 2007, DRAM chips pack 2 Gbits in one chip.

However, for illustration purposes let us consider a DRAM chip that has 64 x 1 bit capacity. That is, if we supply this chip with a 6-bit address it will emit 1 bit of data. Each bit of the DRAM storage is called a *cell*. In reality, the DRAM cells are arranged in a rectangular array as shown in Figure 9.39. The 6-bit address is split into a row address ***i*** (3 bits) and a column address ***j*** (3 bits) as shown in the figure. To access one bit out of this DRAM chip, you would first supply the 3-bit row address ***i*** (called a *Row Access Strobe – RAS –* request). The DRAM chip pulls the whole row corresponding to ***i*** as shown in the figure. Then, you would supply the 3-bit column address ***j*** (called a *Column Access Strobe – CAS –* request). This selects the unique bit (the purple bit in the row buffer) given by the 6-bit address and sends it to the memory controller, which in turn sends it to the CPU.

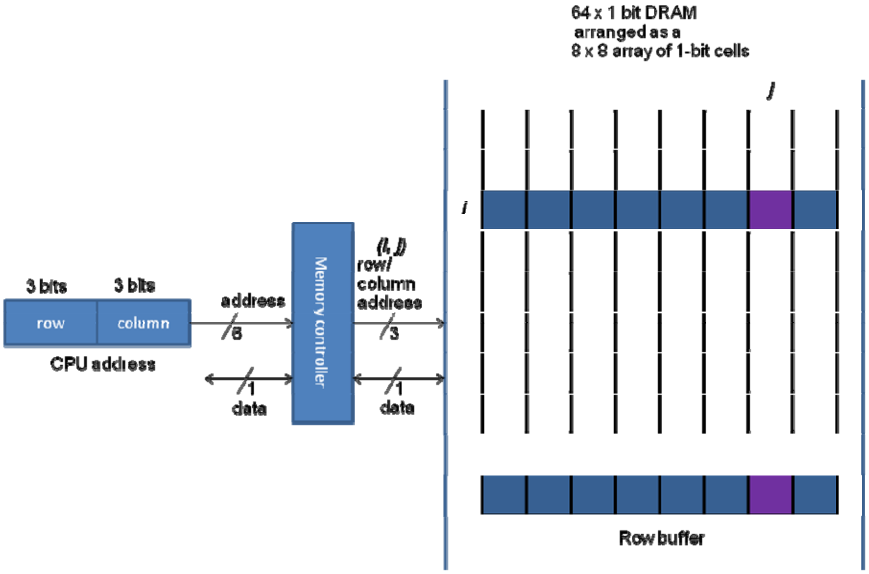


Figure 9.39: Accessing a 64x1 bit DRAM

For a 1 G-bit chip[6], we will need a 32 K x 32 K array of cells. The size of the row buffer in this case would be 32 K-bits. It is instructive to understand what constitutes the *cycle time* of a DRAM. As we have said before, each cell in the DRAM is a capacitive charge. There is circuitry (not shown in the figure) that senses this capacitive charge for every cell in the chosen row and buffers the values as 0's and 1's in the corresponding bit positions of the row buffer. In this sense, reading a DRAM is destructive operation. Naturally, after reading the chosen row, the DRAM circuitry has to re-charge the row so that it is back to what it originally contained. This destructive read followed by the re-charge combined with the row and column address decode times add up to determine the cycle time of a DRAM. Reading a row out of the array into the buffer is the most time consuming component in the whole operation. You can quickly see that after all this work, only 1 bit, corresponding to the column address, is being used and all the other bits are discarded. We will shortly see (Section 9.21.1) how we may be able to use more of the data in the row buffer without discarding all of them.

We can generalize the DRAM structure so that each cell (*i, j*) instead of emitting 1 bit, emits some *k* bits. For example, a 1 M x 8-bit DRAM will have a 1 K x 1 K array of cells, with each cell of the array containing 8 bits. Address and data are communicated to the DRAM chip via *pins* on the chip (see Figure 9.40). One of the key design considerations in chip design is reducing the number of such input/output pins. It turns out the electrical current needed to actuate logic elements within a chip is quite small. However, to communicate logic signals in and out of the chip you need much larger currents. This means having bulky signal driving circuitry at the edges of the chip, which eats into the real estate available on the chip for other uses (such as logic and memory). This is the reason why the cells within a DRAM are arranged as a square array rather than a linear array, so that the same set of pins can be time multiplexed for sending the row and column addresses to the DRAM chip. This is where the RAS (row address

[6] Recall that 1 G-bit is 2^{30} bits.

strobe) and CAS (column address strobe) terminologies come from in the DRAM chip parlance. The downside to sharing the pins for row and column addresses is that they have to be sent to the DRAM chip in sequence, thus increasing the cycle time of the DRAM chip.

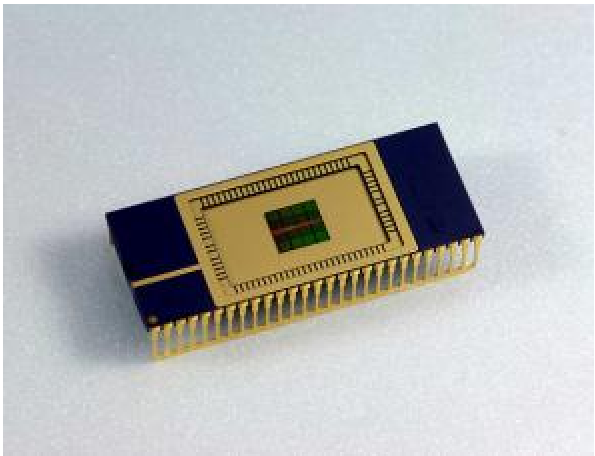


Figure 9.40: Samsung's 1 Gbit DRAM chip[7]

The following example illustrates how to design a main memory system out of these basic DRAM chips.

Example 13:

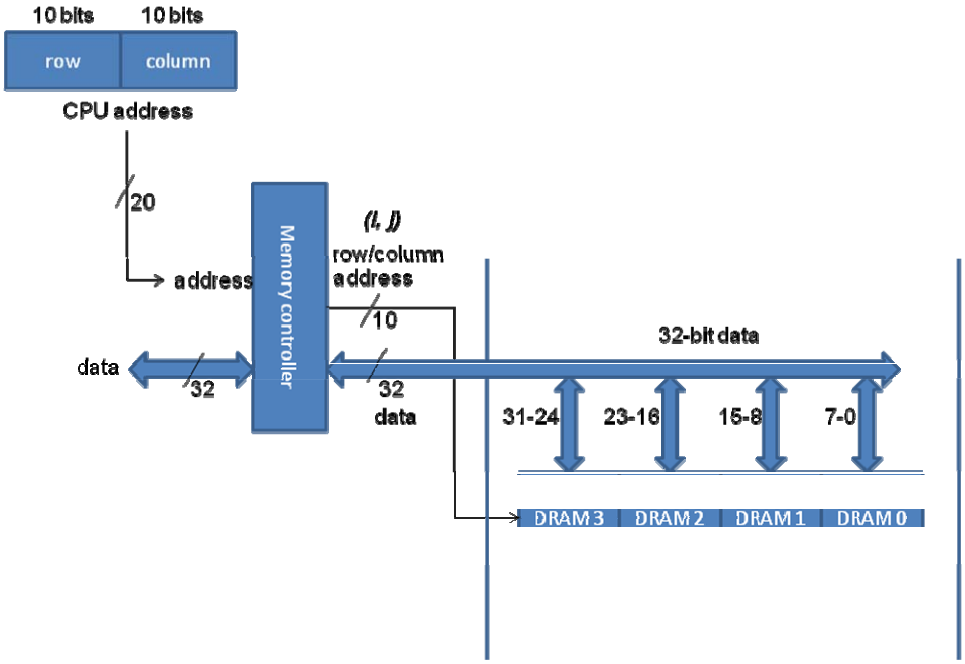


Figure 9.41: 32 Mbits memory system using 1 M x 8-bits DRAM chips

[7] Source: http://www.physorg.com/news80838245.html

Design a main memory system given the following:

- Processor to memory bus
    - Address lines = 20
    - Data lines = 32
- Each processor to memory access returns a 32-bit word specified by the address lines
- DRAM chip details: 1 M x 8-bits

Answer:

Total size of the main memory system = 220 words x 32-bits/word = 1 M words
= 32 Mbits

Therefore, we need 4 DRAM chips each of 1 M x 8-bits, arranged as shown in Figure 9.41.

It is straightforward to extend the design to a byte-addressed memory system. The following example shows how to build a 4 GB memory system using 1 Gbit DRAM chips.

Example 14:

Design a 4 GByte main memory system given the following:

- Processor to memory bus
    - Address lines = 32
    - Data lines = 32
- Each CPU word is 32-bits made up of 4 bytes
- CPU supports byte addressing
- The least significant 2-bits of the address lines specify the byte in the 32-bit word
- Each processor to memory access returns a 32-bit word specified by the word address
- DRAM chip details: 1 Gbits

Answer:

In the 32-bit address, the most significant 30 bits specify the word address.
Total size of the main memory system = 2^{30} words x 4 bytes/word = 4 GBytes = 32 Gbits

Therefore, we need 32 DRAM chips, each of 1 Gbits arranged for convenience in a 2-dimensional array as shown in Figure 9.42. For writing individual byte in a word, the memory controller (not shown in the figure) will select the appropriate row in this 2-dimensional array and send the 15-bit RAS and CAS requests to that row along with other control signals (not shown in the figure). For reading a 32-bit word, it will send the 15-bit RAS and CAS requests to all the DRAMs so that a full 32-bit word will be returned to the controller.

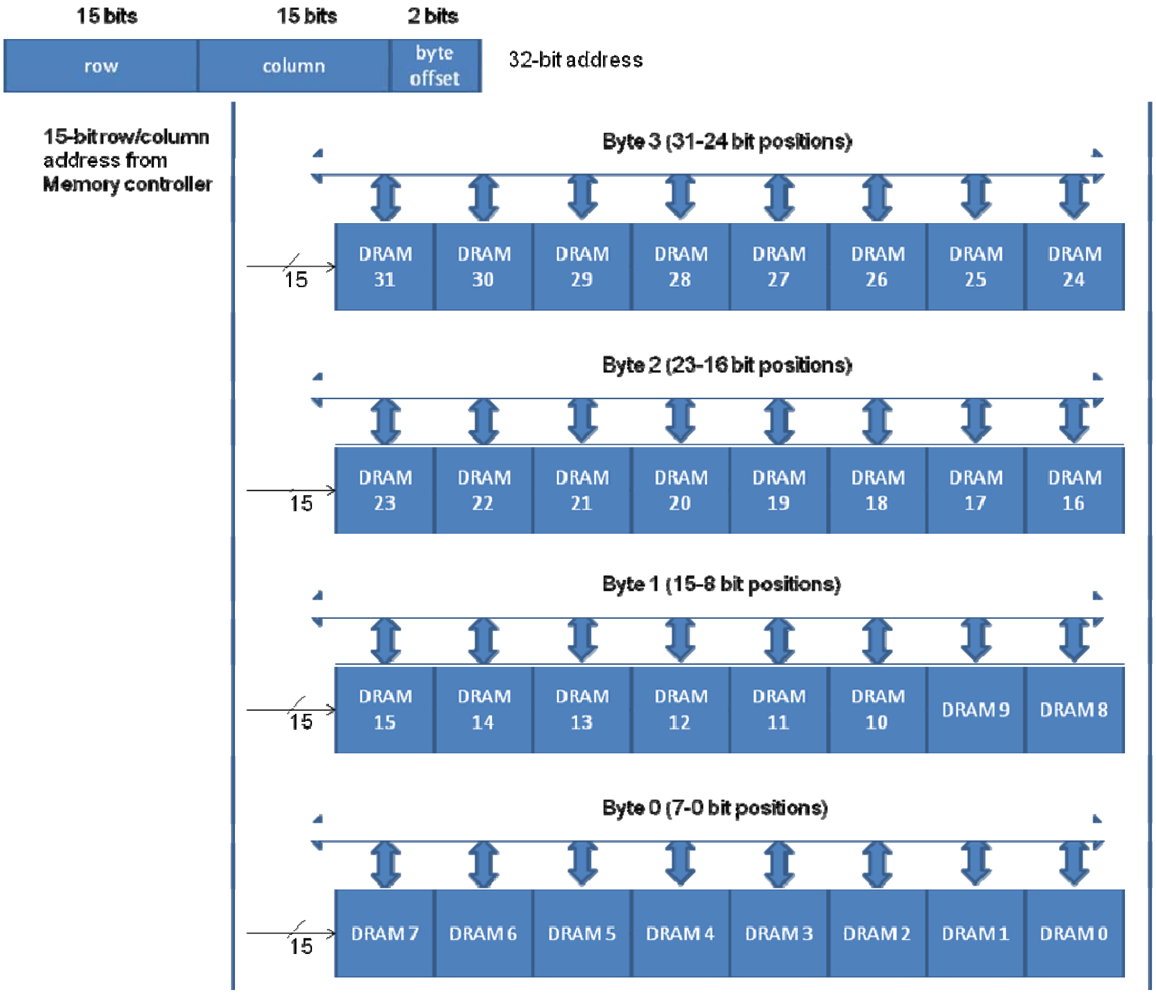


Figure 9.42: A 4 GByte memory system using a 1 Gbit DRAM chip

Manufacturers package DRAM chips in what are called *Dual In Line Memory Modules* (DIMMs). Figure 9.43 shows a picture of a DIMM. Typically, a DIMM is a small printed circuit board and contains 4-16 DRAM chips organized in an 8-byte wide datapath. These days, DIMM is the basic building block for memory systems.

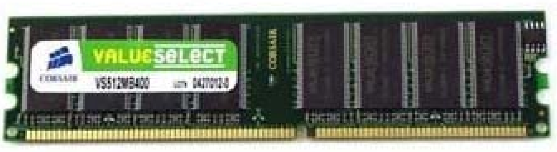


Figure 9.43: Dual in Line Memory Module (DIMM)[8]

[8] Source: http://static.howstuffworks.com/gif/motherboard-dimm.jpg

9.21.1 Page mode DRAM

Recall, what happens within a DRAM to read a cell. The memory controller first supplies it a row address. DRAM reads the entire selected row into a row buffer. Then the memory controller supplies the column address, and the DRAM selects the particular column out of the row buffer and sends the data to the memory controller. These two components constitute the *access time* of the DRAM and represent the bulk of a DRAM cycle time. As we mentioned earlier in this section, the rest of the row buffer is discarded once the selected column data is sent to the controller. As we can see from Figures 9.41 and 9.42, consecutive column addresses within the same row map to contiguous memory addresses generated by the CPU. Therefore, getting a block of data from memory translates to getting successive columns of the *same row* from a DRAM. Recall the trick involved in building interleaved memory (see Section 9.20.3). Basically, each memory bank stored a different word of the same block and sent it on the memory bus to the CPU in successive bus cycles. DRAMs support essentially the same functionality through a technique referred to as *fast page mode (FPM)*. Basically, this technique allows different portions of the row buffer to be accessed in successive CAS cycles without the need for additional RAS requests. This concept is elaborated in the following example.

Example 15:

The memory system in example 14 is augmented with a processor cache that has a block size of 16 bytes. Explain how the memory controller delivers the requested block to the CPU on a cache miss.

Answer:

In Figure 9.44, the top part shows the address generated by the CPU. The memory controller's interpretation of the CPU address is shown in the bottom part of the figure. Note that the bottom two bits of the column address is the top two bits of the block offset in the CPU address. The block size is 16 bytes or 4 words. The successive words of the requested cache block is given by the 4 successive columns of row ***i*** with the column address only changing in the last two bits.

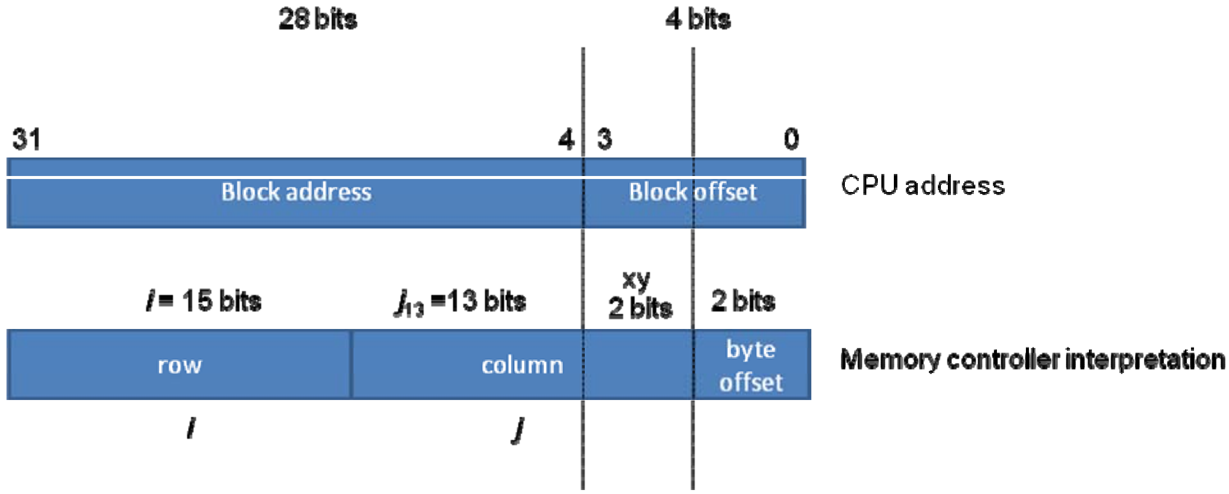


Figure 9.44: Memory controller's interpretation of a 32-bit CPU address

For example, if the CPU address for the block is (***i, j***), where ***j*** = 0100000110001**01**.
Let us denote

$\boldsymbol{j}$ by $\boldsymbol{j_{13}xy}$, where $\boldsymbol{j_{13}}$ represents the top 13 bits of the column address $\boldsymbol{j}$.

To fetch the entire block from the DRAM array, the memory controller does the following:

1. Send RAS/CAS request to DRAM array for address (***i, j***)
2. Send 3 more CAS requests with addresses $\boldsymbol{j_{13}xy}$, where xy = 00, 10, 11

Each of the CAS requests will result in the DRAM array sending 4 words (the first word returned will be the actual address that incurred a miss) successively. The memory controller will pipeline these 4 words in responses back to the CPU in four successive memory bus cycles.

Over the years, there have been numerous enhancements to the DRAM technology. We have given a flavor of this technology in this section. Hopefully, we have stimulated enough interest in the reader to look at more advanced topics in this area.

9.22 Performance implications of memory hierarchy

There is a hierarchy of memory that the CPU interacts with either explicitly or implicitly: processor registers, cache (several levels), main memory (in DRAM), and virtual memory (on the disk). The farther the memory is from the processor the larger the size and slower the speed. The cost per byte decreases as well as we move down the memory hierarchy away from the processor.

Type of Memory	Typical Size	Approximate latency in CPU clock cycles to read one word of 4 bytes
CPU registers	8 to 32	Usually immediate access (0-1 clock cycles)
L1 Cache	32 (Kilobyte) KB to 128 KB	3 clock cycles
L2 Cache	128 KB to 4 Megabyte (MB)	10 clock cycles
Main (Physical) Memory	256 MB to 4 Gigabyte (GB)	100 clock cycles
Virtual Memory (on disk)	1 GB to 1 Terabyte (TB)	1000 to 10,000 clock cycles (not accounting for the software overhead of handling page faults)

Table 9.1: Relative Sizes and Latencies of Memory Hierarchy circa 2006

As we already mentioned the actual sizes and speeds continue to improve yearly, though the relative speeds and sizes remain roughly the same. Just as a concrete example, Table 1 summarizes the relative latencies and sizes of the different levels of the memory hierarchy **circa 2006**. The clock cycle time **circa 2006** for a 2 GHz Pentium processor is 0.5 ns.

Memory hierarchy plays a crucial role in system performance. One can see that the miss penalty affects the pipeline processor performance for the currently executing program. More importantly, the memory system and the CPU scheduler have to be cognizant of the memory hierarchy in their design decisions. For example, page replacement by the memory manager wipes out the contents of the corresponding physical frame from all the levels of the memory hierarchy. Therefore, a process that page faulted may experience a significant loss of performance immediately after the faulting is brought into physical memory from the disk. This performance loss continues until the contents of the page fill up the nearer levels of the memory hierarchy.

CPU scheduling has a similar effect on system performance. There is a *direct* cost of context switch which includes saving and loading the process control blocks (PCBs) of the de-scheduled and newly scheduler processes, respectively. Flushing the TLB of the de-scheduled process forms part of this direct cost. There is an *indirect* cost of context switching due to the memory hierarchy. This costs manifests as misses at the various levels of the memory hierarchy all the way from the caches to the physical memory. Once the working set of the newly scheduled process gets into the nearer levels of the memory hierarchy then the process performance reaches the true potential of the processor. Thus, it is important that the *time quantum* used by the CPU scheduler take into account the true cost of context switching as determined by the effects of the memory hierarchy.

9.23 Summary

The following table provides a glossary of the important terms and concepts introduced in this chapter.

Category	Vocabulary	Details
Principle of locality (Section 9.2)	Spatial	Access to contiguous memory locations
	Temporal	Reuse of memory locations already accessed
Cache organization	Direct-mapped	One-to-one mapping (Section 9.6)
	Fully associative	One-to-any mapping (Section 9.11.1)
	Set associative	One-to-many mapping (Section 9.11.2)
Cache reading/writing (Section 9.8)	Read hit/Write hit	Memory location being accessed by the CPU is present in the cache
	Read miss/Write miss	Memory location being accessed by the CPU is not present in the cache
Cache write policy (Section 9.8)	Write through	CPU writes to cache and memory
	Write back	CPU only writes to cache; memory updated on replacement
Cache parameters	Total cache size (*S*)	Total data size of cache in bytes
	Block Size (*B*)	Size of contiguous data in one data block
	Degree of associativity (*p*)	Number of homes a given memory block can reside in a cache
	Number of cache lines (*L*)	*S/pB*
	Cache access time	Time in CPU clock cycles to check hit/miss in cache
	Unit of CPU access	Size of data exchange between CPU and cache
	Unit of memory transfer	Size of data exchange between cache and memory
	Miss penalty	Time in CPU clock cycles to handle a cache miss
Memory address interpretation	Index (*n*)	log_2L bits, used to look up a particular cache line
	Block offset (*b*)	log_2B bits, used to select a specific byte within a block
	Tag (*t*)	$a – (n+b)$ bits, where *a* is number of bits in memory address; used for matching with tag stored in the cache

Cache entry/cache block/cache line/set	Valid bit	Signifies data block is valid
	Dirty bits	For write-back, signifies if the data block is more up to date than memory
	Tag	Used for tag matching with memory address for hit/miss
	Data	Actual data block
Performance metrics	Hit rate (*h*)	Percentage of CPU accesses served from the cache
	Miss rate (*m*)	$1 - h$
	Avg. Memory stall	Misses-per-instruction$_{\text{Avg}}$ * miss-penalty$_{\text{Avg}}$
	Effective memory access time (EMAT$_{\text{i}}$) at level *i*	EMAT$_i$ = $T_i + m_i *$ EMAT$_{i+1}$
	Effective CPI	CPI$_{\text{Avg}}$ + Memory-stalls$_{\text{Avg}}$
Types of misses	Compulsory miss	Memory location accessed for the first time by CPU
	Conflict miss	Miss incurred due to limited associativity even though the cache is not full
	Capacity miss	Miss incurred when the cache is full
Replacement policy	FIFO	First in first out
	LRU	Least recently used
Memory technologies	SRAM	Static RAM with each bit realized using 6 transistors
	DRAM	Dynamic RAM with each bit realized using a single transistor
Main memory	DRAM access time	DRAM read access time
	DRAM cycle time	DRAM read and refresh time
	Bus cycle time	Data transfer time between CPU and memory
	Simulated interleaving using DRAM	Using page mode bits of DRAM

Table 9.2: Summary of Concepts Relating to Memory Hierarchy

9.24 Memory hierarchy of modern processors – An example

Modern processors employ several levels of caches. It is not unusual for the processor to have on-chip an L1 cache (separate for instructions and data), and a combined L2 cache for instructions and data. Outside the processor there may be an L3 cache followed by the main memory. With multi-core technology, the memory system is becoming even more sophisticated. For example, AMD introduced the Barcelona[9] chip in 2006[10]. This chip which has quad-core, has a per core L1 (split I and D) and L2 cache. This is followed by an L3 cache that is shared by all the cores. Figure 9.45 show the memory hierarchy of the Barcelona chip. The L1 cache is 2-way set-associative (64 KB for instructions and 64 KB for data). The L2 cache is 16-way set-associative (512 KB combined for instructions and data). The L3 cache is 32-way set-associative (2 MB shared among all the cores).

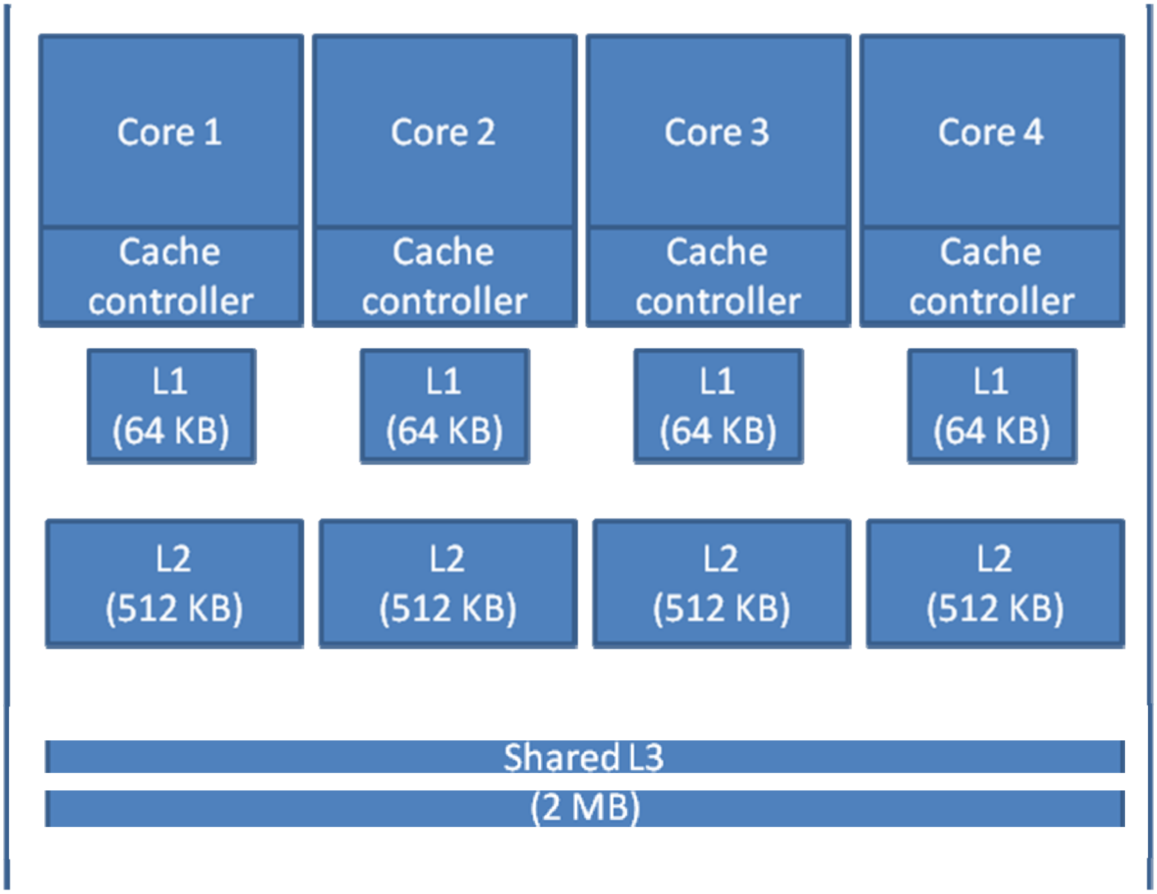


Figure 9.45: AMD's Barcelona chip

9.25 Review Questions

1. Compare and contrast spatial locality with temporal locality.

2. Compare and contrast direct-mapped, set-associative and fully associative cache designs.

[9] Phenom is the brand name under which AMD markets this chip on desktops.

[10] AMD Phenom processor data sheet: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/44109.pdf

3. A fellow student has designed a cache with the most significant bits containing the index and the least significant bits containing the tag. How well do you suppose this cache will perform?

4. In a direct-mapped cache with a t-bit long tag how many tag comparators would you expect to find and how many bits would they compare at a time?

5. Explain the purpose of dirty bits in a cache.

6. Give compelling reasons for a multilevel cache hierarchy.

7. What are the primary design considerations of caches? Discuss how these considerations influence the different levels of the memory hierarchy
8. What is the motivation for increasing the block size of a cache?

9. What is the motivation for a set-associative cache design?

10. Answer True/False with justification: L1 caches usually have higher associativity compared to L2 caches.

11. Give three reasons for having a split I- and D-cache at the L1 level.

12. Give compelling reasons for a unified I- and D-caches at deeper levels of the cache hierarchy.

13. Answer True/False with justification: As long as there is an L2 cache, the L1 cache design can focus exclusively on matching its speed of access to the processor clock cycle time.

14. Your engineering team tells you that you can have 2 MB of on-chip cache total. You have a choice of making it one big L1 cache, two level L1 and L2 caches, split I- and D-caches at several levels, etc. What would be your choice and why? Consider hit time, miss rate, associativity, and pipeline structure in making your design decisions. For this problem, it is sufficient if you give qualitative explanation for your design decisions.

15. How big a counter do you need to implement a True LRU replacement policy for a 4-way set associative cache?

16. Consider the following memory hierarchy:
    - L1 cache: Access time = 2ns; hit rate = 99%
    - L2 cache: Access time = 5ns; hit rate = 95%
    - L3 cache: Access time = 10ns; hit rate = 80%
    - Main memory: Access time = 100ns

Compute the effective memory access time.

17. A memory hierarchy has the following resources:

L1 cache	2 ns access time	98% hit rate
L2 cache	10 ns access time	??
Memory	60 ns access time	

Assume that for each of L1 and L2 caches, a lookup is necessary to determine whether a reference is a hit or miss. What should be the hit rate of L2 cache to ensure that the effective memory access time is no more than 3 ns.

18. You are designing a cache for a 32-bit processor. Memory is organized into words but byte addressing is used. You have been told to make the cache 2-way set associative with 64 words (256 bytes) per block. You are allowed to use a total of 64K words (256K bytes) for the data (excluding tags, status bits, etc.)

Sketch the layout of the cache here (note: no control logic should be shown)

Show how a memory reference will be subdivided: block offset, index, tag.

19. From the following statements regarding cache memory, select the ones that are True.

- It can usually be manipulated just like registers from the instruction-set architecture of a processor

- It cannot usually be directly manipulated from the instruction-set architecture of a processor

- It is usually implemented using the same technology as the main memory

- It is usually much larger than a register file but much smaller than main memory