4. *Normalize and round off.* Shift the result mantissa right by one bit if there was a carry-out in the preceding step, and decrease the result exponent by 1. Otherwise, shift left until the most significant bit is a 1, increasing the result exponent by 1 for each bit shifted. Suppress the leftmost 1 for the result mantissa. Round off.

Step 3 (add mantissas) is a little more complex than it appears. Not only does a carry-out need to be detected when both operands are of the same sign, but also, when $D = 0$ and the two operands are of different signs, the resulting mantissa may be of the wrong sign. Its 2's complement must then be taken, and the sign of the result must be flipped.

This algorithm lends itself quite well to pipelining in three (or possibly four) stages: compare exponents, shift and add mantissas (in either one or two stages), and normalize and round off.

The algorithm for multiplication is straightforward:

1. *Add exponents.* The resulting exponent is $E = E_1 + E_2 - 127$ (we need to subtract one of the two biases). The resulting sign is positive if $S_1 = S_2$ and negative otherwise.
2. *Prepare and multiply mantissas.* The preparation simply means inserting 1's at the left of F_1 and F_2. We don't have to worry about signs.
3. *Normalize and round off.* As in addition, but without having to worry about a potential right shift.

Because the multiplication takes longer than the other two steps, we must find a way to break it into stages. The usual technique is to use a Wallace tree of carry–save adders (CSAs).[4] If the number of CSAs is deemed too large, as for example for double-precision f-p arithmetic, a feedback loop must be inserted in the "tree." In that case, consecutive multiplications cannot occur on every cycle, and the delay depends on the depth of the feedback.

With the increase in clock frequencies, the tendency will be to have deeper pipelines for these operations, because less gate logic can be activated in a single cycle. The optimal depth of a pipeline depends on many microarchitectural factors besides the requirement that stages must be separated by stable state components, that is, the pipeline registers. We shall return briefly to this topic in Chapter 9.

2.2 Caches

In our pipeline design we have assumed that the IF and Mem stages were always taking a single cycle. In this section, we take a first look at the design of the memory hierarchy of microprocessor systems and, in particular, at the various organizations

[4] CSAs take three inputs and generate two outputs in two levels of AND–OR logic. By using a log–sum process we can reduce the number of operands by a factor 2/3 at each level of the tree. For a multiplier and a multiplicand of n bits each, we need $n - 2$ CSAs before proceeding to the last regular addition with a carry–lookahead adder. Such a structure is called a Wallace tree.

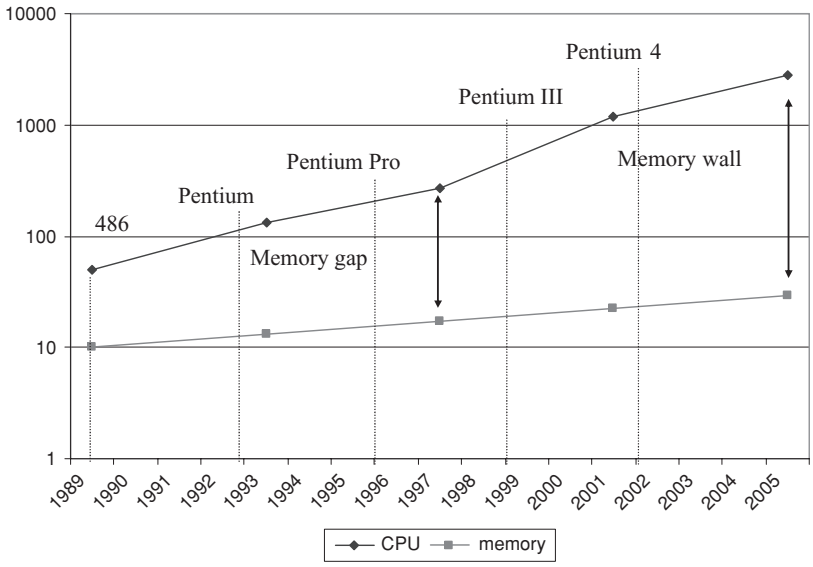

Figure 2.10. Processor–memory performance gap (note the logarithmic scale).

and performance of caches, the highest levels in the hierarchy. Our assumption was that caches were always perfect. Naturally, this is not the reality.

In early computers, there was only one level of memory, the so-called primary memory. The secondary memory, used for permanent storage, consisted of rotating devices, such as disks, and some aspects of it are discussed in the next section. Nowadays primary memory is a hierarchy of components of various speeds and costs. This hierarchy came about because of the rising discrepancy between processor speeds and memory latencies. As early as the late 1960s, the need for a memory buffer between high-performance CPUs and main memory was recognized. Thus, caches were introduced. As the gap between processor and memory performance increased, multilevel caches became the norm. Figure 2.10 gives an idea of the processor and memory performance trends during the 1990s. While processor speeds increased by 60% per year, memory latencies decreased by only 7% per year. The ratio of memory latency to cycle time, which was about 5:1 in 1990, became more than one order of magnitude in the middle of the decade (the *memory gap*) and more than two orders of magnitude by 2000 (the *memory wall*). Couched in terms of latencies, the access time for a DRAM (main memory) is of the order of 40 ns, that is, 100 processor cycles for a 2.5 GHz processor.

The goal of a memory hierarchy is to keep close to the ALU the information that is needed presently and in the near future. Ideally, this information should be reachable in a single cycle, and this necessitates that part of the memory hierarchy, a cache, be on chip with the ALU. However, the capacity of this on-chip SRAM memory must not be too large, because the time to access it is, in a first approximation, proportional to its size. Therefore, modern computer systems have multiple levels of caches. A typical hierarchy is shown in Figure 2.11 with two levels of

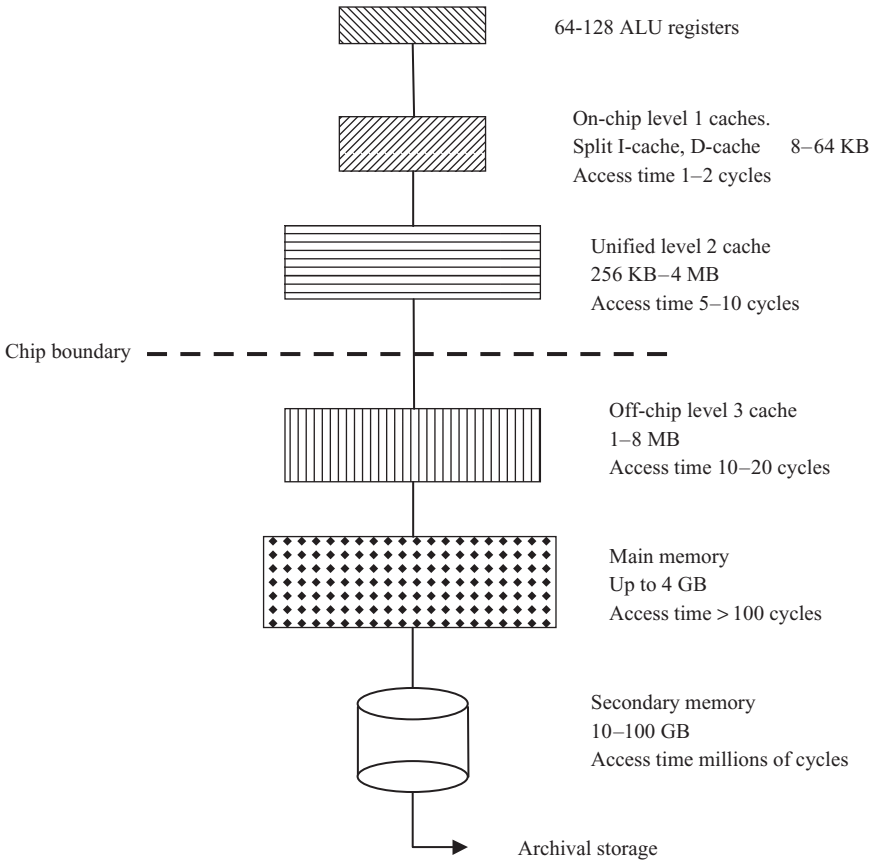

Figure 2.11. Levels in the memory hierarchy.

on-chip caches (SRAM) and one level of off-chip cache (often a combination of SRAM and DRAM), and main memory (DRAM).

Note that the concept of caching is used in many other aspects of computer systems: disk caches, network server caches, Web caches, and so on. Caching works because information at all these levels is not accessed randomly. Specifically, computer programs exhibit the *principle of locality*, consisting of *temporal locality*, whereby data and code used in the past are likely to be reused in the near future (e.g., data in stack, code in loops), and *spatial locality*, whereby data and code close (in terms of memory addresses) to the data and code currently referenced will be referenced again in the near future (e.g., traversing a data array, straight-line code sequences).

Caches are much smaller than main memory, so, at a given time, they cannot contain all the code and data of the executing program. When a memory reference is generated, there is a lookup in the cache corresponding to that reference: the I-cache for instructions, and the D-cache for data. If the memory location is mapped in the cache, we have a *cache hit*; otherwise, we have a *cache miss.* In the case of a hit, the content of the memory location is loaded either in the IF/EX pipeline register in the case of an instruction, or in the Mem/WB register in the case of a load, or else the

content of the cache is modified in the case of a store. In the case of a miss, we have to recursively probe the next levels of the memory hierarchy until the missing item is found. For ease of explanation, in the remainder of this section we only consider a single-level cache unless noted otherwise. In addition, we first restrict ourselves to data cache reads (i.e., loads). Reads for I-caches are similar to those for D-caches, and we shall look at writes (i.e., stores) separately.

2.2.1 Cache Organizations

Caches serve as high-speed buffers between main memory and the CPU. Because their storage capacity is much less than that of primary memory, there are four basic questions on cache design that need to be answered, namely:

1. When do we bring the content of a memory location into the cache?
2. Where do we put it?
3. How do we know it's there?
4. What happens if the cache is full and we want to bring the content of a location that is not cached? As we shall see in answering question 2, a better formulation is: "What happens if we want to bring the content of a location that is not cached and the place where it should go is already occupied?"

Some top-level answers follow. These answers may be slightly modified because of optimizations, as we shall see in Chapter 6. For example, our answer to the first question will be modified when we introduce prefetching. With this caveat in mind, the answers are:

1. The contents of a memory location are brought into the cache *on demand*, that is, when the request for the data results in a cache miss.
2. Basically the cache is divided into a number of cache entries. The mapping of a memory location to a specific cache entry depends on the *cache organization* (to be described).
3. Each cache entry contains its name, or *tag*, in addition to the data contents. Whether a memory reference results in a hit or a miss involves checking the appropriate tag(s).
4. Upon a cache miss, if the new entry conflicts with an entry already there, one entry in the cache will be replaced by the one causing the miss. The choice, if any, will be resolved by a replacement algorithm.

A generic cache organization is shown in Figure 2.12. A cache entry consists of an address, or tag, and of data. The usual terminology is to call the data content of an entry a *cache line* or a *cache block*.[5] These lines are in general several words (bytes) long; hence, a parameter of the cache is its *line size*. The overall cache capacity, or *cache size*, given in kilobytes (KB) or megabytes (MB), is the product of the number of lines and the line size, that is, the tags are not counted in the overall

[5] We will use *line* for the part of a cache entry and *block* for the corresponding memory image.

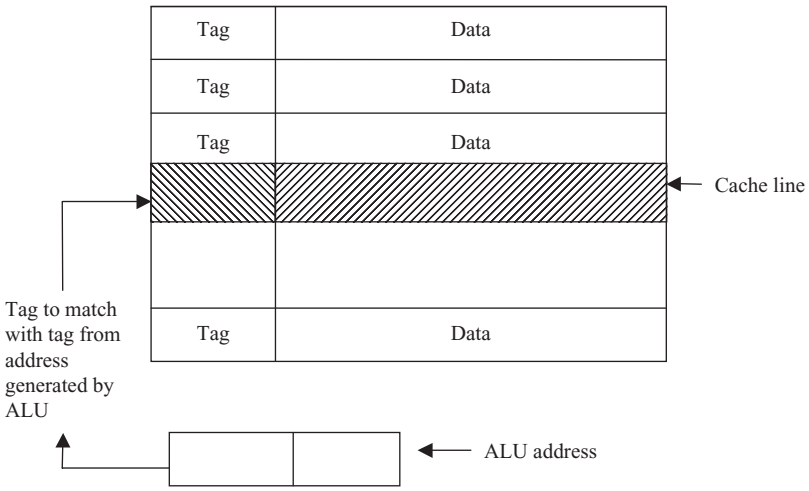

Figure 2.12. Generic cache organization. If (part of) the address generated by the ALU matches a tag, we have a hit; otherwise, we have a miss.

cache size. A good reason for discarding the tags in the terminology (but not in the actual hardware) is that from a performance viewpoint they are overhead, that is, they do not contribute to the buffering effect.

Mapping of a memory location, or rather of a sequence of memory words of line-size width that we will call a *memory block*, to a cache entry can range from full generality to very restrictive. If a memory block can be mapped to any cache entry, we have a *fully associative* cache. If the mapping is restricted to a single entry, we have a *direct-mapped* cache. If the mapping is to one of several cache entries, we have a *set-associative* cache. The number of possible entries is the set-associative *way*. Figure 2.13 illustrates these definitions.

Large fully associative caches are not practical, because the detection of a cache hit or miss requires that all tags be checked in the worst case. The linear search can be avoided with the use of *content-addressable memories* (CAMs), also called associative memories. In a CAM, instead of addressing the memory structure as an array (i.e., with an index), the addressing is by matching a key with the contents of (part of) all memory locations in the CAM. All entries can be searched in parallel. A matching entry, if any, will raise a flag indicating its presence. In the case of a fully associative cache, the key is a part of the address generated by the ALU, and the locations searched in parallel are the tags of all cache entries. If there is no match, we have a miss; otherwise, the matching entry (there can be only one in this case) indicates the position of the cache hit.

While conceptually quite attractive, CAMs have for main drawbacks that (i) they are much more hardware-expensive than SRAMs, by a factor of 6 to 1, (ii) they consume more power, and (iii) they are difficult to modify. Although fully associative hardware structures do exist in some cases, e.g., for write buffers as we shall see in this section, for TLBs as we shall see in the next section, and for other hardware data structures as we shall see in forthcoming chapters, they are in general very small, up to 128 entries, and therefore cannot be use for general-purpose caches.

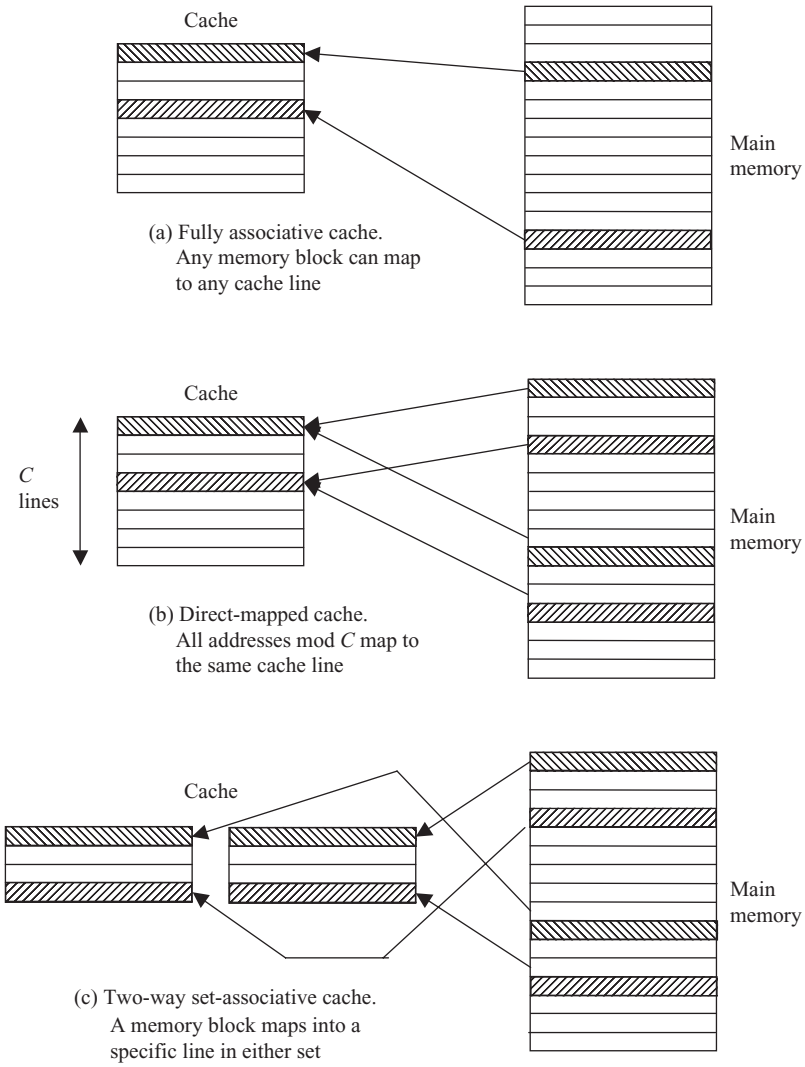

Figure 2.13. Cache organizations.

In the case of direct-mapped caches, the memory block will be mapped to a single entry in the cache. This entry is *memory block address mod C*, where *C* is the number of cache entries. Since in general-purpose microprocessors *C* is a power of 2, the entry whose tag must be checked is very easy to determine.

In the case of an *m*-way set-associative cache, the cache is divided into *m banks* with *C*/*m* lines per bank. A given memory block can be mapped to any of *m* entries, each of which is at address *memory block address mod C*/*m* in its bank. Detection of a cache hit or miss will require *m* comparators so that all comparisons can be performed concurrently.

A cache is therefore completely defined by three parameters: The number of cache lines, *C*; the line size *L*; and the associativity *m*. Note that a direct-mapped cache can be considered as one-way set-associative, and a fully associative cache as

C-way associative. Both L and C/m are generally powers[6] of 2. The cache size, or capacity, S is $S = C \times L$: thus either (C,L,m) or (S,L,m), a notation that we slightly prefer, can be given to define the *geometry* of the cache.

When the ALU generates a memory address, say a 32-bit byte address, how does the hardware check whether the (memory) *reference* will result into a cache hit or a cache miss? The memory address generated by the ALU will be decomposed into three fields: *tag, index*, and *displacement*. The *displacement* d, or *line offset*, indicates the low-order byte within a line that is addressed. Since there are L bytes per line, the number of bits needed for d is $d = \log_2 L$, and of course these bits are the least significant bits of the address. Because there are C/m lines per bank, the *index* i of the cache entry at which the m memory banks must be probed requires $i = \log_2(C/m)$ bits. The remaining t bits are for the tag. In the first-level caches that we are considering here, the t bits are the most significant bits.

EXAMPLE 3: Consider the cache (S,m,L) with $S = 32KB$, $m = 1$(direct-mapped), and $L = 16B$. The number of lines is $32 \times 1024/16 = 2048$. The memory reference can be seen as the triplet (t,i,d), where:

The displacement $d = \log L = \log 16 = 4$.
The index $i = \log(C/m) = \log 2048 = 11$.
The tag $t = 32 - 11 - 4 = 17$.

The hit–miss detection process for Example 3 is shown in Figure 2.14.

Let us now vary the line size and associativity while keeping the cache capacity constant and see the impact on the tag size and hence on the overall hardware requirement for the implementation of the cache. If we double the line size and keep the direct-mapped associativity, then d requires one more bit; the number of lines is halved, so i is decreased by 1; and therefore t is left unchanged. If we keep the same line size but look at two-way set associativity ($m = 2$), then d is left unchanged, i decreases by 1 because the number of lines per set is half of the original, and therefore t increases by 1. In addition, we now need two comparators for checking the hit or miss, or more generally one per way for an m-way cache, as well as a multiplexer to drive the data out of the right bank. Note that if we were to make the cache fully associative, the index bits would disappear, because $i = \log(C/C) = \log 1 = 0$.

From our original four questions, one remains, namely: What happens if on a cache miss all cache entries to which the missing memory block maps are already occupied? For example, assume a direct-mapped cache. Memory blocks a and b have addresses that are a multiple of C blocks apart, and a is already cached. Upon a reference to b, we have a cache miss. Block b should therefore be brought into the cache, but the cache entry where it should go is already occupied by block a. Following the principle of locality, which can be interpreted as favoring the most recent references over older ones, block b becomes cached and block a is evicted.

[6] m is not necessarily a power of 2. In that case, if m' is the smallest power of 2 larger than m, then there are m banks of C/m' lines.

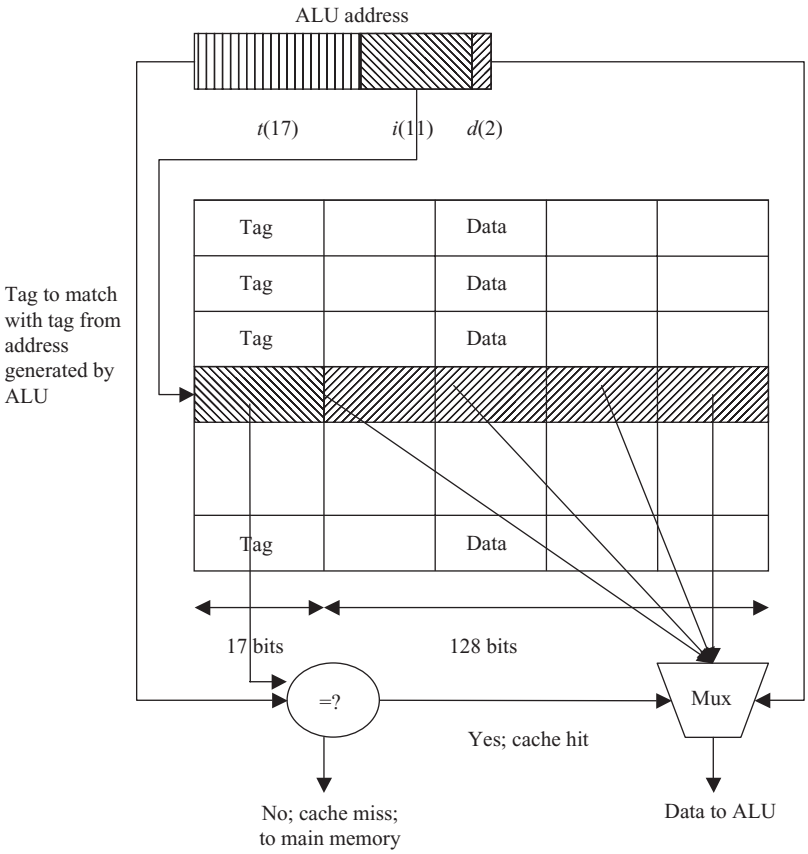

Figure 2.14. Hit and miss detection. The cache geometry is (32 KB,1,16).

In the case of an m-way set-associative cache, if all m cache entries with the same mapping as the block for which there is a miss are occupied, then a *victim* to be evicted must be selected. The victim is chosen according to a *replacement algorithm.* Following again the principle of locality, a good choice is to replace the line that has been not been accessed for the longest time, that is, the *least-recently used* (LRU). LRU replacement is easy to implement when the associativity m is small, for example $m \leq 4$. For the larger associativities that can happen in second- or third-level caches, approximations to LRU can be used (Chapter 6). As long as the (maybe two) most recently used (MRU) lines are not replaced, the influence of the replacement algorithm on cache performance is minimal. This will not be the case for the paging systems that we discuss in the next section.

Write Strategies

In our answers to the top-level questions we have assumed that the memory references corresponded to reads. When the reference is for a write, (i.e., a consequence of a store instruction), we are faced with additional choices. Of course, the steps taken to detect whether there is a cache hit or a cache miss remain the same.

In the case of a cache hit, we must certainly modify the contents of the cache line, because subsequent reads to that line must return the most up-to-date information. A first option is to write only in the cache. Consequently, the information in the cache might no longer be the image of what is stored in the next level of the memory hierarchy. In these *writeback* (or *copyback*) caches we must have a means to indicate that the contents of a line have been modified. We therefore associate a *dirty bit* with each line in the cache and set this bit on a write cache hit. When the line becomes a victim and needs to be replaced, the dirty bit is checked. If it is set, the line is written back to memory; if it is not set, the line is simply evicted. A second option is to write both in the cache and in the next level of the hierarchy. In these *write-through* (or *store-through*) caches there is no need for the dirty bit. The advantage of the writeback caches is that they generate less memory traffic. The advantage of the write-through caches is that they offer a consistent view of memory.

Consider now the case of a cache miss. Again, we have two major options. The first one, *write-allocate*, is to treat the write miss as a read miss followed by a write hit. The second, *write-around*, is to write only in the next level of the memory hierarchy. It makes a lot of sense to have either write-allocate writeback caches or write-around write-through caches. However, there are variations on these two implementations, and we shall see some of them in Chapter 6.

One optimization that is common to all implementations is to use a *write buffer.* In a write-through cache the processor has to wait till the memory has stored the data. Because the memory, or the next level of the memory hierarchy, can be tens to hundreds of cycles away, this wait is wasteful in that the processor does not need the result of the store operation. To circumvent this wait, the data to be written and the location where they should be written are stored in one of a set of temporary registers. The set itself, a few registers, constitutes the write buffer. Once the data are in the write buffer, the processor can continue executing instructions. Writes to memory are scheduled as soon as the memory bus is free. If the processor generates writes at a rate such that the write buffer becomes full, then the processor will have to stall. This is an instance of a structural hazard. The same concept can be used for writeback caches. Instead of generating memory writes as soon as a dirty replacement is mandated, the line to be written and its tag can be placed in the write buffer. The write buffer must be checked on every cache miss, because the required information might be in it. We can take advantage of the fact that each entry in the buffer must carry the address of where it will be written in memory, and transform the buffer into a small fully associative extra cache. In particular, information to be written can be coalesced or overwritten if it belongs to a line already in the buffer.

The Three C's

In order to better understand the effect of the geometry parameters on cache performance it is useful to classify the cache misses into three categories, called the three C's:

1. Compulsory (or cold) misses: This type of misses occurs the first time a memory block is referenced.