CACHE MEMORY

Cache (French) n. - hidden storage space, v. to save up as for future use

Today's high performance microprocessors operate at speeds that far outpace even the fastest of the memory bus architectures that are commonly available. One of the biggest limitations of main memory is the wait state: period of time between operations. This means that during the wait states the processor wait for the memory to be ready for the next operation. The most common technique used to match the speed of the memory system to that of the processor is caching.

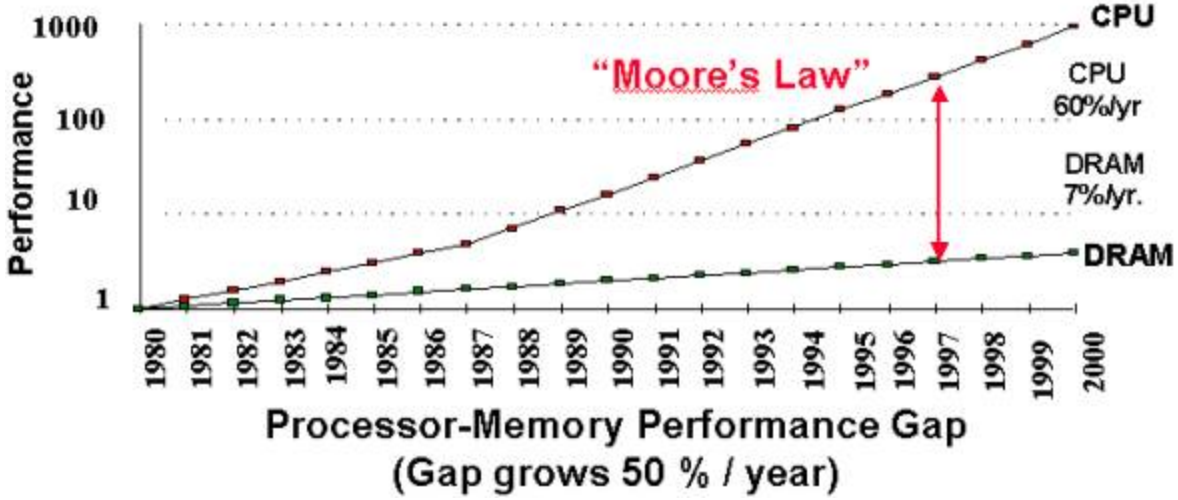

Figure 1: Processor – Memory Performance Gap [1]

Cache Memory is the level of computer memory hierarchy situated between the processor and main memory. It is a very fast memory the processor can access much more quickly than main memory or RAM. Cache is relatively small and expensive. Its function is to keep a copy of the data and code (instructions) currently used by the CPU. By using cache memory, waiting states are significantly reduced and the work of the processor becomes more effective. [2]

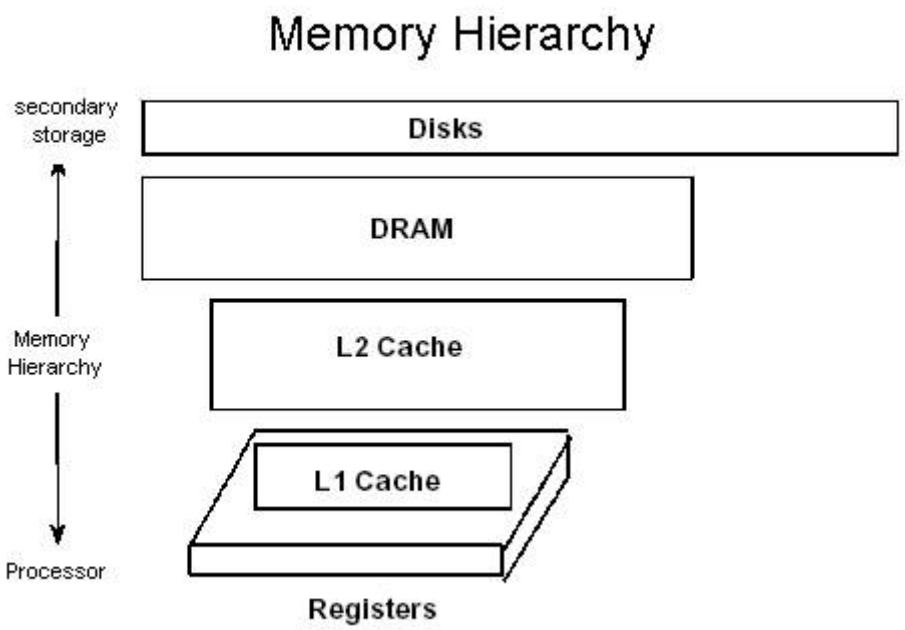

Figure 2: Memory Hierarchy [3]

Cache is much faster than main memory because it is implemented using SRAM (Static Random Access Memory). The problem with DRAM, which comprises main memory, is that it is composed entirely of capacitors, which have to be constantly refreshed in order to preserve the stored information (leakage current). Whenever data is read from the cell, the cell is refreshed. The DRAM cells need to be refreshed very frequently, i.e. typically every 4 to 16 ms. This slows down the entire process. SRAM on the other hand consists of flip-flops, which stay in its state as long as the power supply is on. (A flip-flop is an electrical circuit composed of transistors and resistors. See picture) Because of this SRAM need not be refreshed and is over 10 times faster than DRAM. Flip-flops, however, are implemented using complex circuitry which makes SRAM much larger and more expensive, limiting its use. [4]

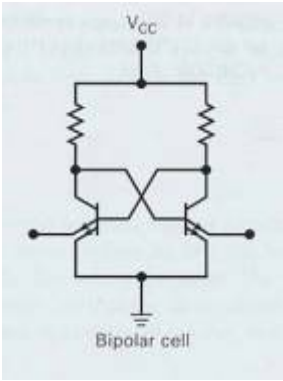

Figure 3: Flip – flop [5]

Principles of cache memory

In general cache memory works by attempting to predict which memory the processor is going to need next, and loading that memory before the processor needs it, and saving the results after the processor is done with it. Whenever the byte at a given memory address is needed to be read, the processor attempts to get the data from the cache memory. If the cache doesn't have that data, the processor is halted while it is loaded from main memory into the cache. At that time memory around the required data is also loaded into the cache. [6]

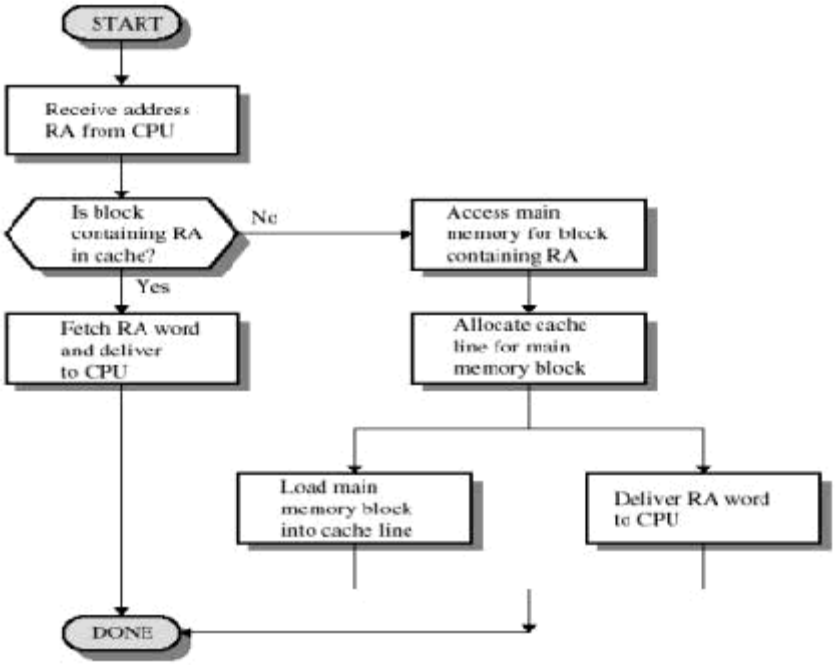

Figure 4: Cache Read Operation [1]

The basic principle that cache technology is based upon is **locality of reference –** programs tend to access only a small part of the address space at a given point in time. This notion has two underlying

assumptions– **temporal locality and spatial locality.**

Temporal locality means that referenced memory is likely to be referenced again soon. In other words, if the program has referred to an address it is very likely that it will refer to it again.

Spatial locality means that memory close to the referenced memory is likely to be referenced soon. This means that if a program has referred to an address, it is very likely that an address in close proximity will be referred to soon.

How does cache memory work?

Main memory consists of up to 2^n addressable words, with each word having a unique *n*-bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of *K* words each. Thus, there are $M = 2^n / K$ blocks. The cache is split into *C blocks* of *K* words each, Figure with number of blocks considerably smaller than the number of main memory blocks (C << M). At any time, only a subset of the blocks of main memory resides in the blocks in the cache. If a word in a block of memory is read, that block is transferred to one of the blocks of the cache. Since there are more blocks in main memory than blocks in cache memory, an individual block of main memory cannot be uniquely and permanently dedicated to a particular block in cache memory. Therefore, each block includes a tag that identifies which particular block of main memory is currently occupying that blocks of cache. The tag is usually a portion (number of bits) of the main memory address [1]

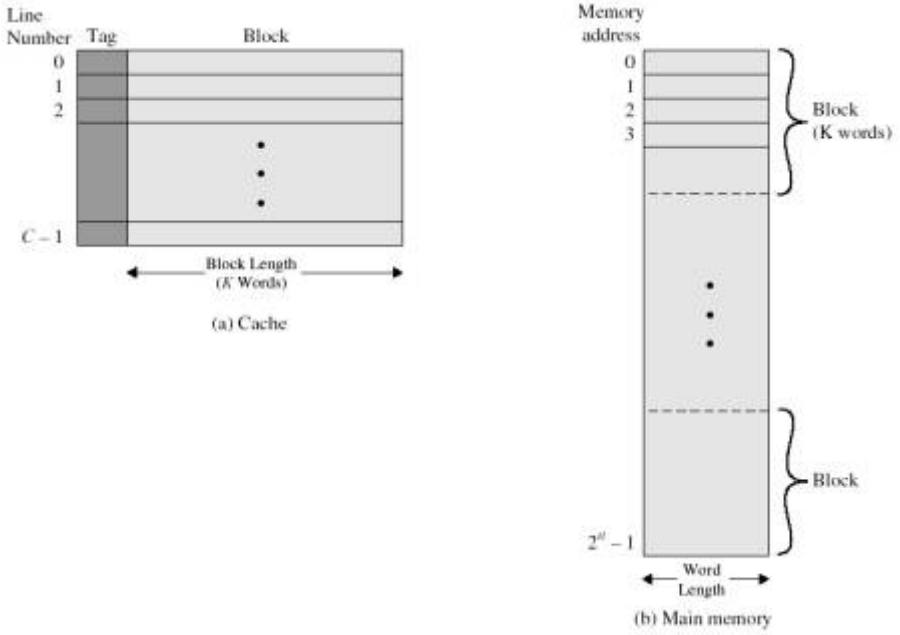

Figure 5: Cache / Main – Memory Structure [1]

Cache's effectiveness:

The effectiveness of the cache is determined by the **number of times the cache successfully provides the required data.** This is the place to introduce some terminology. A **hit** is the term used when the data, required by the processor is found in the cache. If data is not found, we have a cache **miss**. There are three types of misses:

Compulsory misses – the first access to a block is not in the cache, so the block must be brought into the cache. These are also called cold start misses or first reference misses.

C**apacity misses** – if the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. These are misses due to cache size.

Conflict misses – if the block placement strategy is set associative or direct mapped, conflict misses will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called collision misses or interference misses.

The time it takes to access cache on a hit is the **hit time**. If however, the CPU does not find the required data in cache, in other words, when there is a miss, then the data is fetched from main memory and replicated into the cache**. The time needed to fetch the block from the lower levels, replicate it and then access it, is called miss penalty. Hit ratio** is the percentage of hits from total memory accesses. **Miss ratio** is equal to (1-hit ratio). One way to improve cache effectiveness is to reduce the number of misses and increase the number of hits; in other words we strive for high hit rate and low **miss rate** (*1-hit rate*). [11] [8]

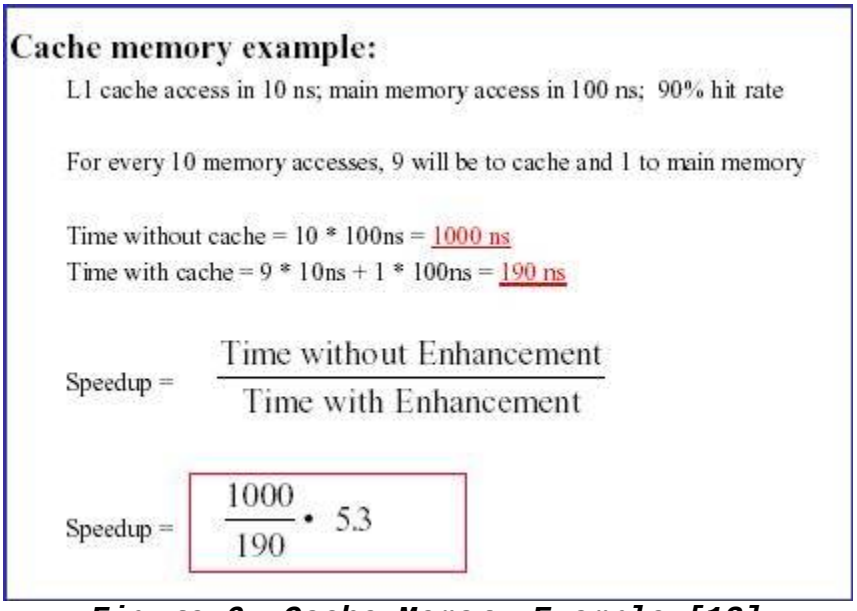

Figure 6: Cache Memory Example [12]

In order to improve the cache performance we need to increase hit ratio. Many different techniques have been developed and the following are the most commonly used ones.

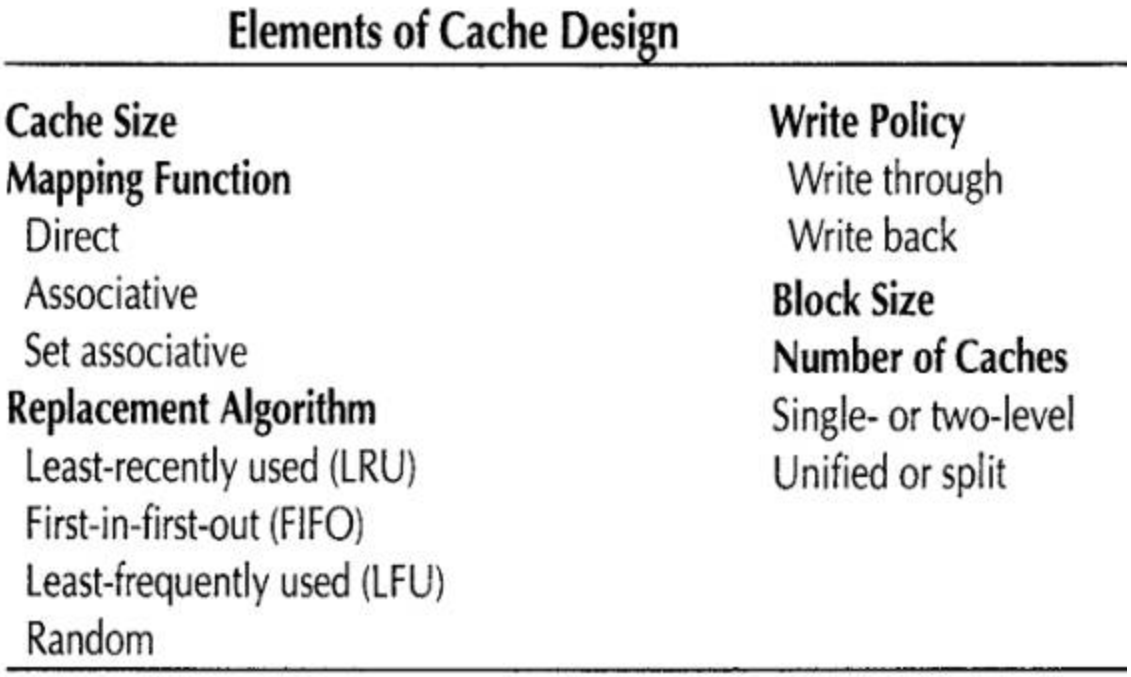

Figure 7: Elements of Cache Design [1]

Cache Size:

One way to decrease miss rate is to increase the cache size. The larger the cache, the more information it will hold, and the more likely a processor request will produce a cache hit, because fewer memory locations are forced to share the same cache blocks. However, applications benefit from increasing the cache size up to a point, at which the

performance will stop improving as the cache size increases. When this point is reached, one of two things have happened: either the cache is large enough that the application almost never have to retrieve information from disk, or, the application is doing truly random accesses, and therefore increasing size of the cache doesn't significantly increase the odds of finding the next requested information in the cache. The latter is fairly rare - almost all applications show some form of locality of reference.

In general the cache has to be small enough so that the overall average “cost per bit” is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other reasons to minimize the size of the cache. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones – even when using the same integrated circuit technology and put into the same place on the chip. Cache size is also limited by the available space on the chip.

Typical cache sizes today range from 1K words to several mega-words. Multitasking systems tend to favor 256 K words as nearly optimal size for main memory cache. The performance of the cache is very sensitive to the nature of the workload, making it impossible to arrive at a single "optimum" cache size.

Mapping function:_

The mapping function gives the correspondence between main memory blocks and cache blocks. Since each cache blocks is shared between several blocks of main memory (the number of memory blocks >> the number of cache blocks), when one block is read in, another one should be moved out. Mapping functions minimize the probability that a moved-out block will be referenced again in the near future. There are three types of mapping functions: direct, fully associative, and n-way set associative.

Direct Mapping: In direct mapped cache each main memory block is assigned to a specific blocks (or lines) in the cache according to the formula i = j modulo C, where i is the cache blocks number assigned to main memory block j, and C is the number of blocks in the cache.
We can say that block j of main memory is assigned to block i of cache memory.
Direct mapping cache interprets a main memory address (comprising of s + w bits) as 3 distinct fields.

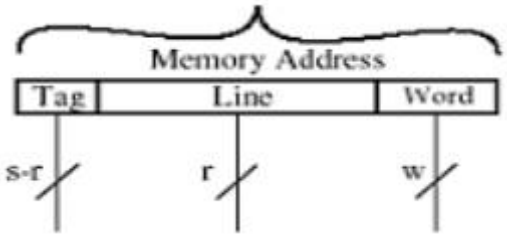

Figure 8: Direct Mapped Cache Address [1]

» Tag (the most significant s - r bits) - a unique identifier for each memory block. It is stored in the cache along with the data words of the blocks specify one of the 2^s blocks.

» Blocks number (middle r bits) – specifies which blocks will hold the referenced address.Blocks number identify one of the C = 2^r blocks of cache.

» Word (least significant w bits) - specifies the offset of a byte within a blocks or block. Word specifies the specific byte in a cache blocks that is to be accessed.[1]

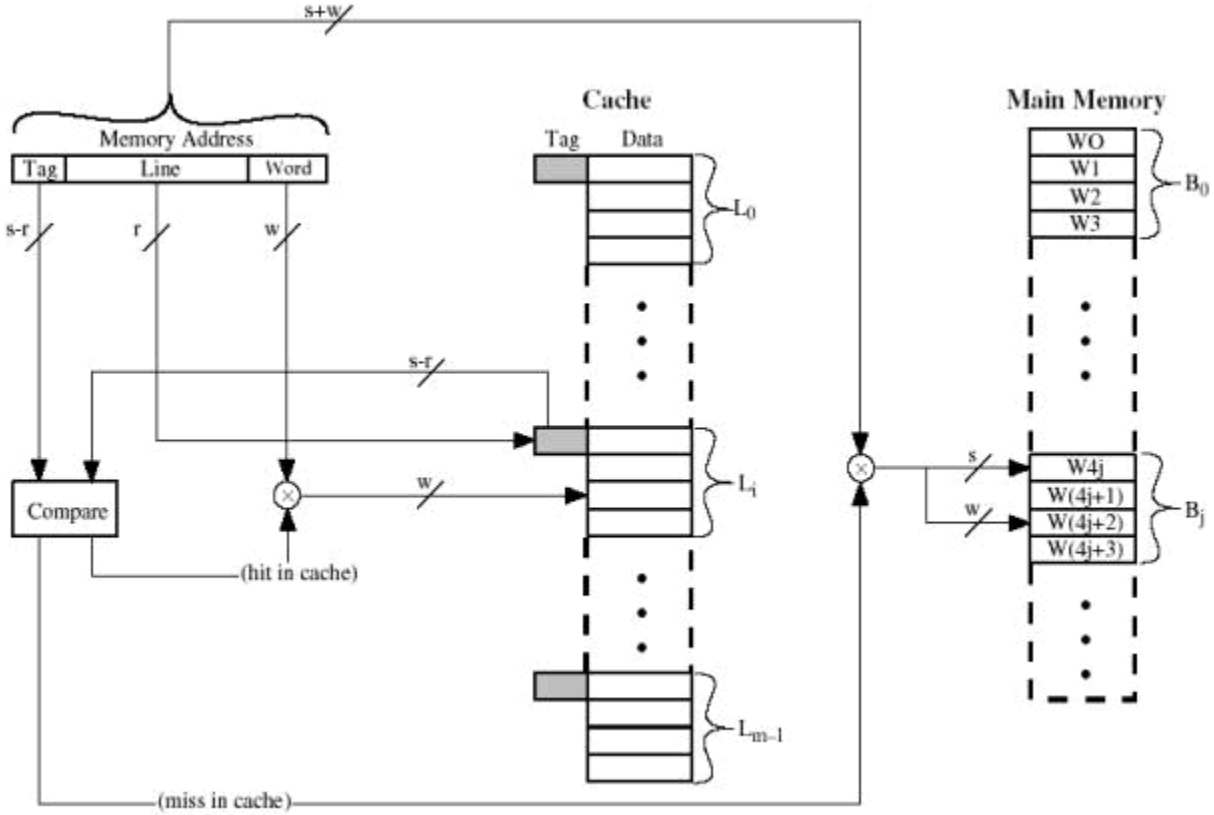

Figure 9: Direct Mapped Cache Access

For every memory reference that the CPU makes, the tag of the cache blocks that is supposed to hold the required block is checked to see if the correct block is in the cache. Since no two blocks that map into the same blocks have the

same tag, the cache controller can determine if we have a cache hit or a cache miss.

Direct mapped cache is the simplest form of cache. It is easy and relatively inexpensive to implement, and determining whether a main memory block can be found in cache is simple and quicker than with other mapping functions. However, it has one significant disadvantage - each main memory block is mapped to a specific cache blocks. Through locality of reference, it is possible to repeatedly reference blocks that map to the same blocks number. These blocks will be constantly swapped in and out of cache, causing the hit ratio to be low. Although such swapping is rare in a single-tasking system, in multi-tasking systems it can occur quite often and thus slow down a direct-mapped cache. In general the performance is worst for this type of mapping.

Fully Associative Mapping: Fully associative cache allows each block of main memory to be stored in any blocks of the cache. Main memory address (s + w bits) is divided into two distinct fields: tag (s bits) and word offset (w bits).

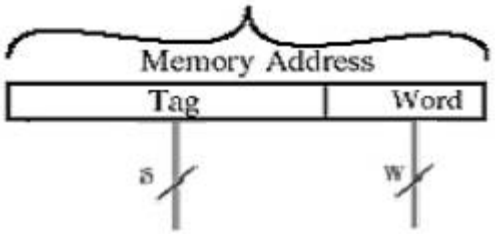

Figure 10: Fully Associative Cache Address

The cache controller must now check the tag of every blocks in the cache to determine if a given memory address is currently resident in the cache. This is not done using common searching techniques but rather using associative memory (also called Content Addressable Memory (CAM)). CAM basically allows the entire tag memory to be searched in parallel. Unlike typical RAM, CAM associates logic with each memory cell in the memory. This allows for the contents of all memory cells in the CAM to be checked in a single clock cycle. Thus, access to the CAM is based upon content not address as with ordinary RAM. CAMs are considerably more expensive in terms of gates than ordinary access by address memories (RAMs) and this limits their use (at least with current technologies) to relatively small memory systems such as cache.

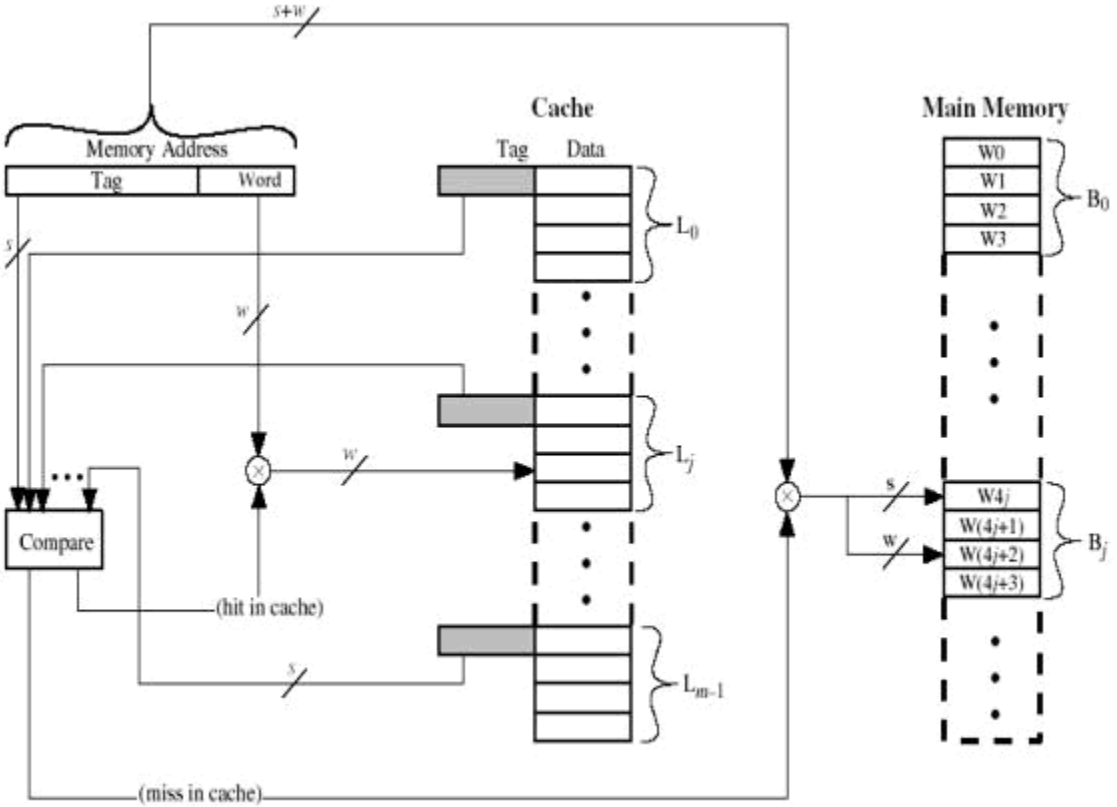

Figure 11: Fully Associative Cache Access

The cache controller will need to uniquely identify every main memory block, which will require s bits since there are 2^s blocks in the main memory. As before, within the block, w bits will be required to uniquely identify a particular byte within a specific block. Since any cache blocks can hold any main memory block at any time, the cache controller needs to perform a simultaneous search over all tags to find the desired blocks (to see if it is in the cache). This is where the CAM comes into play. The entire contents of the cache can be compared simultaneously with the CAM. A cache hit is indicated when one of the CAM cells contents matches the search address. If none of the CAM cells contents matches the search address then a cache miss has occurred and the proper block from main memory will need to be loaded into one of the cache blocks. At this point the cache controller will invoke the blocks replacement algorithm, which will select which cache blocks is to be replaced to make room for the new incoming block.

Fully associative mapping is very flexible and overcomes direct mapping's main weakness. The fully associative cache has the best hit ratio because any blocks in the cache can hold any address that needs to be cached. However this cache suffers from problems involving searching the cache. Even with specialized hardware to do the searching, a performance penalty is incurred. And this penalty occurs for *all* accesses to memory, whether a cache hit occurs or not. In

addition, more logic must be added to determine which of the various blocks to use when a new entry must be added (usually some form of a "least recently used" algorithm is employed to decide which cache blocks to use next). All this overhead adds cost, complexity and execution time. That is why, fully associative cache is very rarely used.

Set Associative Cache is basically a good compromise between direct mapping and fully associative mapping. It builds on the strengths of both: namely, the easy control of the direct mapped cache and the more flexible mapping of the fully associative cache.

In set associative mapping, cache is divided into a number of sets (*v*), with each set holding a number of blocks (*k*).The cache is then described by the number of blocks each set contains. If a set can hold *X* blocks, the cache is referred to as an X-way set associative cache. A main memory block can be stored in any one of the *k* blocks in a set such that *cache set number = j modulo v* (where *j* is the main memory block number).

Direct mapping cache interprets a main memory address (comprising of s + w bits) as 3 distinct fields: Tag (s - d bits), Set (d bits), and Word (w bits).

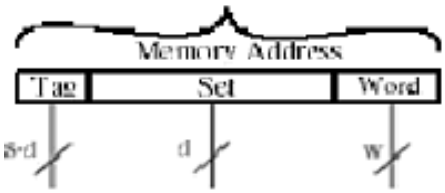

Figure 12: Set-AssociativeCache Address[1]

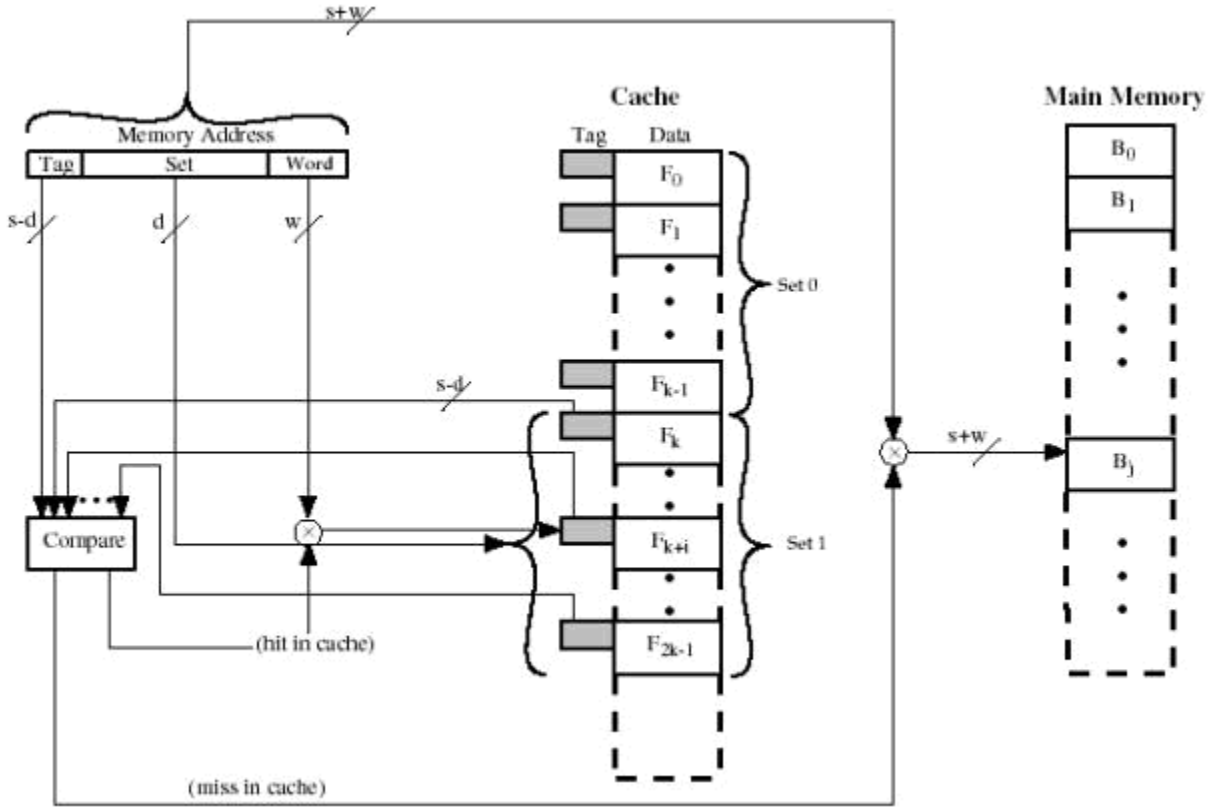

Figure 13:Set-Associative Cache Access[1]

For each main memory group, the cache is capable of holding v different main memory blocks simultaneously. A four-way set associative cache could hold up to four main memory blocks from the same group simultaneously, while an eight-way set associative cache would be capable of holding up to eight main memory blocks from the same group simultaneously, and so on.

Whenever the CPU issues a block request, the cache controller for the k-way set associative cache will need to check the contents of k different cache blocks for the particular group to which the required block belongs. Once again, the use of CAM will allow the contents of all cache blocks to be checked in parallel. If one of the "k" cache blocks contains the requested block, a cache hit occurs, otherwise a cache miss occurs and one of the two cache blocks belonging to that group will need to be selected for replacement. In general, set associative mapping has better hit ratio than direct mapped cache and is faster than fully associative, therefore providing better performance than the other two mapping functions. 2 and 4 way associative caches give the best ratio between hit ratio and search speed.

Summary: In the "real world", the direct mapped and set associative caches are by far the most common. Direct mapping is used more for level 2 caches on motherboards, while the higher-performance set-associative cache is found

more commonly on the smaller primary caches contained within processors.

Cache Type	Hit Ratio	Search Speed
Direct Mapped	Good	**Best**
Fully Associative	**Best**	Moderate
N-Way Set Associative, N>1	Very Good, Better as N Increases	Good, Worse as N Increases

Figure 14: Mapping Function Comparison Table

Cache Blocks Replacement Algorithms :

When a new blocks is loaded into the cache, one of the existing blocks must be replaced. In a direct mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache we have a choice of where to place the requested block, and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set associative cache, we must choose among the blocks in the selected set. Therefore a blocks replacement algorithm is needed which sets up well defined criteria upon which the replacement is made. A large number of algorithms are possible and many have been implemented. Three of the most common cache blocks replacement algorithms are:

- *Least Recently Used* (LRU) - the cache blocks that was last referenced in the most distant past is replaced.
- *FIFO* (First In- First Out) - the cache blocks from the set that was loaded in the most distant past is replaced.
- *Random* - a randomly selected blocks form cache is replaced

The most commonly used algorithm is LRU. LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a

two-way set associative cache, tracking when the two blocks were used can be easily implemented in hardware by adding a single bit (*use* bit) to each cache blocks. Whenever a cache blocks is referenced its use bit is set to 1 and the use bit of the other cache blocks in the same set is set to 0. The blocks selected for replacement at any specific time is the blocks whose use bit is currently 0. The principle of the locality of reference means that a recently used cache blocks is more likely to be referenced again, LRU tends to give the best performance. In practice, as associativity increases, LRU is too costly to implement, since tracking the information is costly. Even for four-way set associativity, LRU is often approximated – for example, by keeping track of which of a pair of blocks is LRU (which requires one bit), and then tracking which blocks in each pair is LRU (which requires one bit per pair). For large associativity, LRU is either approximated or random replacement is used.

Cache Write Policies:

Before a cache blocks can be replaced, it is necessary to determine if the blocks has been modified. The contents of the main memory block and the cache blocks that corresponds to that block are essentially copies of each other and should therefore hold the same data. If cache blocks X has not been modified since its arrival in the cache, updating the main memory block it corresponds to is not required prior to its replacement. The incoming cache blocks can simply overwrite the existing cache memory. On the other hand, if the cache blocks has been modified, at least one write operation has been performed on the cache blocks and thus the corresponding main memory block must be updated. Basically there are two different policies that can be employed to ensure that the cache and main memory contents are the same: write-through and write-back.

Write-through: Assuming a cache hit (a write hit), the information is written immediately to both the blocks in the cache and to the block in the lower-level memory (with its normal wait-state delays). The advantages of this technique are that the contents of the main memory and the cache are always consistent, it is easy to implement and read miss never results in writes to main memory. On the other hand, the write through policy has a significant drawback. It needs a main memory access, which is slower and results in a more memory bandwidth usage. In spite of this, most Intel microprocessors use a write-through cache design.

Write-back (sometimes called a *posted write or copy back* cache): On a cache hit, the information is written only to

the blocks in the cache. This allows the processor to immediately resume processing. The modified cache blocks is written to main memory only when it is replaced. To reduce the frequency of writing back blocks on replacement, a *dirty bit* is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If it is clean the block is not written on a miss. The advantages of the write-back policy are that writes occur at the speed of the cache memory, multiple writes within a block require only one write to main memory, which results in less memory bandwidth usage.
Write-back is a faster alternative to the write-through policy but it has one major disadvantage - the contents of the cache and the main memory are not consistent. This is the cache coherency problem and it is an active research topic. The cache coherency becomes an issue, for example, when a hard disk is read and information is transferred into the main memory through the DMA (Direct Memory Access) system, which does not involve the processor. The cache controller must constantly monitor the changes made in the main memory and ensure that the contents of the cache properly track these changes to the main memory. There are many techniques, which have been employed to allow the cache controller to "snoop" the memory system - but once again, these add complexity and expense. In the PC environment there are special cache controller chips that can be added which basically handle all the responsibilities for supervising the cache system. The cache coherency problem becomes acute in a bus architecture in which more than one device (typically processors) has a cache and main memory is shared.
Writes introduce yet another complication not present in reads. If a write operation has been requested and a cache miss results, again one of two options for handling the write miss can be employed. The blocks may be brought into the cache, and then updated, which is termed a *write-allocate* policy, or the block may be updated directly in main memory and not brought into the cache which is termed a *write-no allocate* policy. Typically, write-through caches will employ a write-no allocate policy while write-back caches will utilize a write-allocate policy.

Write hit policy	**Write miss policy**
Write Through	Write Allocate
Write Through	No Write Allocate

Write Back	Write Allocate
Write Back	No Write Allocate

Figure 15: Possible Combinations on write [15]

Write Through with Write Allocate: On hits it writes to cache and main memory, *on misses it updates the block in main memory and brings the block to the cache.* Bringing the block to cache on a miss does not make a lot of sense in this combination because the next hit to this block will generate a write to main memory anyway (according to Write Through policy)

Write Through with No Write Allocate: On hits it writes to cache and main memory, *on misses it updates the block in main memory not bringing that block to the cache;* Subsequent writes to the block will update main memory because Write Through policy is employed. So, some time is saved not bringing the block in the cache on a miss because it appears useless anyway.

Write Back with Write Allocate: on hits it writes to cache setting "dirty" bit for the block, main memory is not updated, *on misses it updates the block in main memory and brings the block to the cache.* Subsequent writes to the same block, if the block originally caused a miss, will hit in the cache next time, setting dirty bit for the block. That will eliminate extra memory accesses and result in very efficient execution compared with Write Through with Write Allocate combination.

Write Back with No Write Allocate: on hits it writes to cache setting "dirty" bit for the block, main memory is not updated, *on misses it updates the block in main memory not bringing that block to the cache.* Subsequent writes to the same block, if the block originally caused a miss, will generate misses all the way and result in very inefficient execution. [15]