LAB Paper

Cache Memory

4.1. General aspects

Cache in the memory hierarchy

The literary meaning of 'cache' is 'a hiding place' but in the context of computer architecture, this word means '**a place of temporary storage**'. In a computer system, there is a hierarchy in the storage of information: even inside the processor, registers store data. Then, there is the main memory (or simpler, RAM memory), outside the processor but very close to it, followed by the secondary memory, way down in the memory hierarchy. In Fig. 1, the cache memory is even closer to the processor, directly connected to it through the system bus, just as the main memory is. This means that data from the processor can be transported to/from the cache just as in the case of the main memory. However, cache is considered to be higher in the hierarchy. The cache is where information (data and programs) required for the execution of any program is expected to be found in, for speedup the execution of programs. When the processor is about to execute a task, first it looks in the cache. If the item is found there, a cache 'hit' is said to have occurred; otherwise, it is a 'cache miss'. If a miss occurs, the processor goes to the main memory and brings that item to the cache, because when the required information is in cache, speed of execution is very high.

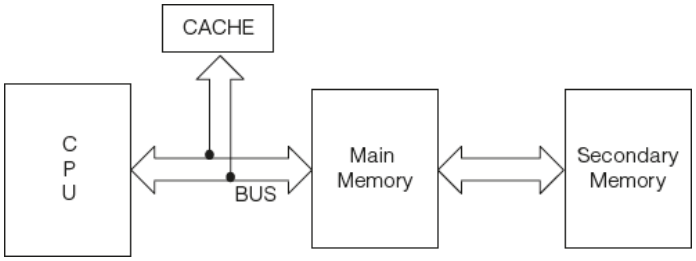


Figure 1. Cache in the memory hierarchy

In a computer system, the fastest component is the processor. It has very little storage because only general-purpose registers are available therein. For a program to have fast execution, there should be a storage space as fast as the processor. The fastest memory available is SRAM whose technology is similar to the technology of the processor, but is very expensive and having sufficiently large quantities of it is prohibitively expensive; it also dissipates a lot of power. So, using SRAM as main memory is ruled out. The next best alternative is DRAM, which is not as good, but sufficient: it has high packing density, low power and low cost. Thus, DRAM has been chosen as the technology of choice for the main memory.

Cache in the x86 family

The early PCs had no cache. The first cache was used for 80386 PCs and it was placed on the motherboard, but the processor offered support for its operation, a cache controller being on the board. The amount of available cache varied depending on the motherboard model and typical values at that time were 64 KB and 128 KB.

With the 80486 processor, Intel added a small amount (8 KB) of cache inside the CPU. This internal cache was called L1 (level 1) or internal, while the external memory cache was called L2 (level 2) or 'external'. The amount and existence of the external memory cache depended also on the motherboard model. Typical amounts for that time were 128 KB and 256 KB.

Then, there was the first Pentium processor with two internal L1 caches– one for data and one for instructions, each with 8 KB in size. Later, the amount of this cache was increased. However, the L2 cache was still external, and its capacity depended on the motherboard manufacturer. Gradually, capacities like 256 KB for L1 cache and 512 KB for L2 became standard. However, having an external cache amounted to less speed, as the CPU clock was high speed (>200 MHz) while motherboard speeds were less (around 66 MHz).

Even at that time, there was the Celeron processor which had no cache and hence slower (but cheaper); these days, also Celeron processors have internal caches.

With P6, things have changed: both L1 and L2 caches were internal ("on chip"). This same architecture is used until today – both L1 and L2 caches are located inside the CPU running at the CPU internal clock rate.

As more and more processors began to include L2 cache into their architectures, Level 3 cache became the name for the extra cache built into motherboards between the microprocessor and the main memory. Quite simply, what was once L2 cache on motherboards now became L3 cache when used with microprocessors containing built-in L2 caches. The latest is that with more CPUs becoming 'multicore', a tertiary level cache was added on to the CPU die, called the L3. It also became common to have the three levels larger in capacity than the next lower level, so that it became not uncommon to find Level 3 cache sizes of 8MB.

For a relatively recent Intel Core i3 generation 5 system, CPU-Z application returned the following specifications for cache memory:

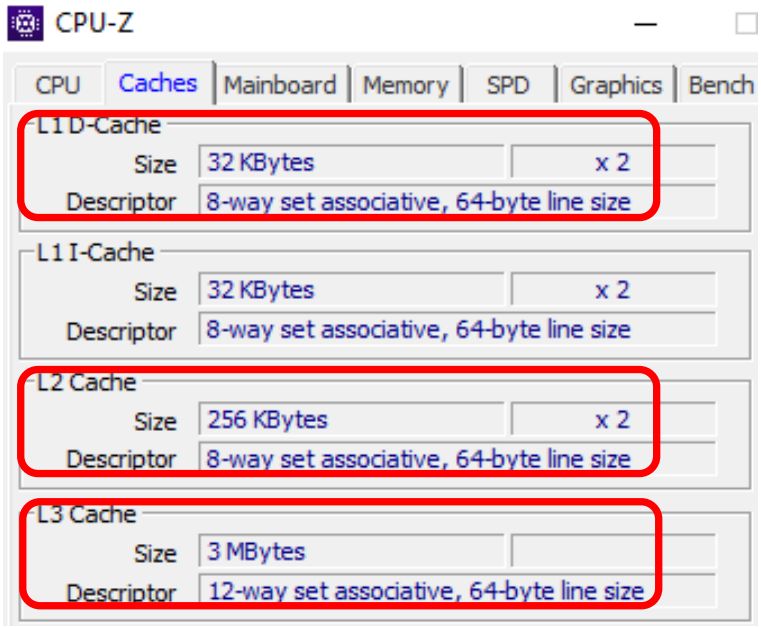


Figure 2. Specifications returned by the CPU-Z application about cache memory

Parameters of the cache memory

The cache memory contains copies of more portions of main memory. When the processor attempts to read a "word of memory"[1], a check is made to determine if that respective word is in the cache memory. In the affirmative case (***if YES***), the word is delivered to the processor and a "cache hit" occurs, so we say that the transaction ended successfully. In the negative case (***if NO***), then the respective word is located inside a block of main memory and the entire block (consisting of a fixed number of words, established in the design stage) will be copied in cache; from this block, only the word required by the processor will be delivered to it (Figure 3). Copying the entire block from main memory to cache memory is done in the idea that those words which are closed to the currently accessed reference could also be accessed (required) soon by the processor (locality principle). Because of the reference locality phenomenon, when a data block is retrieved in cache to satisfy access to the current memory reference, it is highly likely that there will be future references to the same memory location or to other words (from the block containing it), adjacent to it.

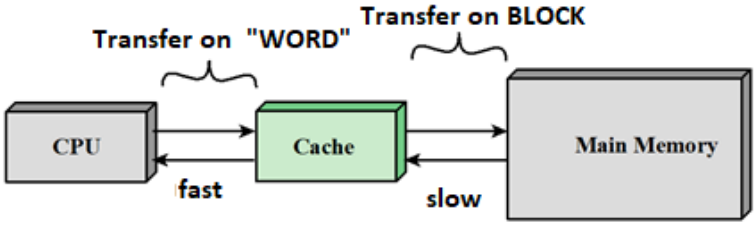


Figure 3. The transfer of data between processor and cache memory, respectively between cache memory and main memory

As one can notice also from Figure 3, the size of cache memory is much smaller than that of main memory (in the real cases, this ratio is around 1:1000), and from this reason a lot of problems appeared. This is because when the processor runs a program, that program may be comprised in different "blocks" of memory and not just one. Thus, the first 3 questions cache designers have faced were:

1) How big the size of a block (that will be copied from RAM in cache) should be?
2) How many such blocks could be "stored" (so their copy to be in the cache) at one moment of time?
3) After what logic should these blocks of data be stored in the cache?

The answer to these questions may be established if the information about cache design is established. The parameters of cache are:

- M = the size of main or RAM memory (in bytes or B)
- N = the size of the cache memory (in bytes or B)
- K = the size of one block (which will be copied from RAM to cache) (in bytes or B)
- The mapping scheme: 1) direct-mapped, 2) fully associative or 3) set-associative (or simpler said: „with sets" or „with ways").

[1] Here, this term, „word of memory" refers to the ***content of one location of memory*** and it does not refer to 16 bits. At x86 family, the content of one location of memory is always a BYTE (octet), so by the generic term "WORD" we refer to 8 bits.

Example 1: The cache system design parameters can be deducted from the information returned by the **CPU-Z application** (Fig. 2): „***Cache L3 of 3MB, 12-way set associative, 64-byte line size***", and suppose that about RAM we know we have a 4GB capacity.

Thus, the first parameter, *the size of the RAM memory* is given: M=4GB=2^{32} B (bytes).

The second parameter, *the size of the cache memory*[2] we infer from the given specifications: N=3MB=3*2^{20} B. We observe that the ratio of the sizes of the two types of memory is $2^{32}/3*2^{20}=2^{12}/3=1365.3$, i.e. 1366 times the main memory is bigger than the cache memory.

The size of one block (which will be copied from RAM to cache) can also be deducted from the returned specifications: „*64-byte line size*" is referring to the line from cache on which the respective block will be stored; these two, i.e. *the block from the RAM* and *the slot from the cache* (where the block will be copied) must always fit together. We will express the size of one block as: K=64=2^6 B.

The mapping scheme is type *3) set-associative*: "*with 12 ways"* is mentioned here; having the number of ways, one can find the number of sets or reverse, because the number of slots from cache can be expressed like: ***nb_of_slots = nb_of_ways * nb_of_sets***. Thus, for our cache memory of 3MB, we will have: 3MB/64 = 12*nb_of_sets => nb_of_sets = 4096. At what these sets refer, we will see after the next section (which explains each mapping scheme in detail).

Most of the time, in practice, one may find the number of blocks from main memory and the number of slots from cache immediately after the system specifications are given. In our case: nb_of_blocks=M/K, and nb_of_slots=N/K, so we have 2^{26} blocks and 3*2^{14} slots.

To conclude, the parameters of the cache system presented in Figure 2 are:

- the size[3] of the RAM memory: M=4GB=2^{32} B (bytes);
- the size of the cache memory: N=3MB=3*2^{20} B;
- the size of one block from the RAM (which is also the size of one slot from the cache): K=64=2^6 B;
    - nb_of_blocks=M/K: 2^{26} blocks, numbered as: Bl 0, Bl 1, …, Bl 2^{26}-1
    - nb_of_slots=N/K: 3*2^{14} slots, numbered as: Sl 0, Sl 1, …, Sl 3*2^{14}-1
- the mapping scheme: type *3)* set-associative: with 12 ways
    - nb of ways: 12 (so, there are 12 slots in one set)
    - nb of sets: 3MB/64 = 12*nb_of_sets => nb_of_sets = 4096=2^{12}, numbered as: Set 0, Set 1, …, Set 2^{12}-1
    - nb of links to one set: 2^{26} / 2^{12} = 2^{14}, so 16384 blocks may attempt to one specific set of 12 slots in cache.

4.2. Cache Mapping Schemes

For cache to be functional (useful to the processor), it must store useful data, those data that the CPU needs to execute the current program. However, those data become useless if the CPU can't find them or it takes too much time to locate them.

When accessing data or instructions, the CPU first generates ***a main memory address*** (in this way the CPU requests data: it tries to access a specific location or reference from main memory). If "the data" resides in cache (so it has been previously copied from main memory to cache), *the address of the data in cache* will not be the same with *the address from the main memory* from where *it* was copied. For example, data located at main memory address 12A8h could be located in the very first location in cache. And thus, one question naturally arises: ***How will the CPU be able to take the data from cache, by knowing its address from main memory?*** (How the CPU knows if the data exists in cache and most important Where it is this located in cache? (If the CPU only knows the main memory localization information?)). The answer to these questions is in the fact that the CPU uses a specific ***mapping scheme***, through which it "converts" (or "map") the *main memory address* into *a cache location*, so that to permanently preserve a trace of what content from main memory it has inside the cache (The CPU always knows *exactly which* areas of the main memory resides at one point in the cache and knows *exactly where*; all these, based on the mapping scheme and the address partitioning operation).

The mapping (conversion) of the addresses from the main memory to cache is done by giving *a specific meaning to the bits that compose the main memory address*. Depending on the used mapping scheme, the bits that compose the main memory address are divided into 2 or 3 distinct groups that we call ***fields***. The processor will use these fields to determine the exact location in cache where that data is located once it has been copied from the main memory; thus, when the processor searches the information in cache, it actually performs the search considering the address from the main memory.

One very important aspect is that when the processor accesses a certain reference from the main memory, (in case there is no such data already in cache) ***an entire block that include that reference will be copied in cache*** and not just that reference. For this, both main memory and cache will be divided into blocks of the same size: in Example 1, this size is 64 ***bytes*** or ***locations***. Most common systems are said to be "byte-addressable", meaning their memory is addressable at the byte level; however, there are also "word-addressable" systems, where memory is organized in words and not in bytes. In the case of our usual x86 systems, when we refer to the usual size of the "word" (the generic term used in the context of cache) we refer to the byte (8 bits) and not to the word (16 bits).

---

[2] For simplicity, we compressed the information returned by the CPU-Z application: we do not refer to cache L1, either to cache L2, but only to the level before RAM (so cache L3); we either specify that cache L1 is divided in cache L1 of data and cache L1 of instructions and we either specified how is this appearing as related to the number of CPU cores.

[3] When referring to "memory size", practically we mean its capacity

The 64 memory entities in Example 1 are 64 bytes. Thus, the 4GB main memory will be (logically, not physically) divided as follows: locations 0 ... 63 will form Block 0, locations 64 ... 127 will form Block 1, locations 128 ... 191 will form Block 2 and so on, the last block having the ID 2^{26}-1 (there are 2^{26} blocks in total).

The cache slots where these blocks will be stored will, of course, have the same size of 64 bytes. Thus, suppose that the processor wants to access the data from the main memory found at location 130: first, it will look for this data in cache; if it does not find it in the cache (where it should be inside the cache - depending on the used mapping scheme), then it will go and look for it in the main memory; once it locates the data in the main memory within block 2, it will copy the entire block 2 in cache for later access.

The size of these blocks may differ from one system to another, the usual dimensions used in the design process being: 8, 16, 32, 64, 128, ... so in general its 2 powers.

The reason for copying an entire block of data in cache is the so-called principle of the locality of references: ***if a sequence was recently accessed, the sequences adjacent to it tend to be accessed soon (the spatial locality)*** and ***if a sequence was recently accessed, it tends to be accessed again soon (the temporal locality)***. This way, even if at the first request of the processor, a miss appeared (the cache did not contained that information), it is possible that at the following requests (at least some of them) to appear hits (the cache may now contain the information required by the CPU). For example, when accessing the information from location 130 (the 3rd in block 2) a miss was reached, but the following 61 accesses (from location 131 to the end of the block, i.e. location 191) will result in 61 hits - if the processor sequentially (in an increasing way) accessed those locations.

To solve the first problem, that is ***to check if the information required by the CPU is in cache*** (it will basically check if the block containing that information from the main memory is in cache), ***the first field*** (called ***tag***) is compared (from the 2 or 3 fields obtained at the address partitioning): it will compare the tag of the reference required by the CPU with the tag of the reference (in Scheme 1) or with the tags of the references (in Schemes 2 and 3) which exists in the cache. It will conclude that: if the two values are equal, then the block containing the data requested by the CPU is located in cache (and a cache hit results).

The problem that remains is ***to locate exactly where that information is in cache*** (that data required by the CPU - because the processor requests the content of a single location, not of an entire block). For this, also the "word" or "offset" field resulting from the partitioning of the address will be compared and the exact position of the respective data will be identified within a specific cache slot.

The mapping schemes are: 1) ***direct mapping***, 2) ***fully associative mapping*** and 3) ***set-associative mapping with sets or ways*** .

Analogy for explaining mapping schemes:

Students (96 students from a specialization) represent the blocks from the main memory (there are 16x6 = 96 blocks). The number 16, which represents the number of chairs from the laboratory room that they must occupy in the laboratory, corresponds to the number of slots in the cache; with this, I pointed out that the cache is much smaller (6 times smaller) than the main memory: not all students of the year may stay in the laboratory room, at one point.

1) Direct-mapping scheme

The direct mapping scheme is the most restrictive of the 3 schemes, because it draws clear ("direct") connections or links between the blocks from the main memory and the place (so the slot) that they could occupy in the cache: a block from the main memory it can only reach a specific cache slot.

From the analogy with the students who arrive in the laboratory room, it is as each student (of the 16x6 = 96 as many attending the lecture course) is allowed to occupy a certain chair in the laboratory room (he or she was told directly at the beginning of the semester, which is that chair and is not allowed to occupy another chair from the lab room).

Because there are far more blocks in the main memory than places in the cache where they can be copied (the number of "slots" is much smaller compared to the number of blocks), these direct links have been drawn so that the contents of the cache be more accurately tracked. For example, at some point, the processor will need to know if a particular block from the main memory is in cache, and this type of mapping is one that can answer this question quickly and accurately, because it knows exactly which slot to be checked for (because of that direct link being drawn).

Using the student's analogy, it's as if someone was asking the lab teacher: is the student Pop Ion in the lab now? Then, the lab teacher looks exactly at a certain chair (the one that Pop Ion should have been sitting on, as he was directed at the beginning of the semester) and can almost instantly answer that question (he just looks at that chair, not searching for the rest, in 15 other possible locations).

This mapping scheme was modularly designed, following a certain rule: ***the X block*** from the main memory can reach ***the cache slot Y***, where **Y = X mod u**, the value ***u being the total number of cache slots***, where ***u = N / K***.

Using the analogy with students, student number 5 can sit only on chair 5, and student number 21 can also occupy only chair 5 (but he or she is in another semigroup), because: 5 modulo 16 = 5 and also 21 modulo 16 = 5.

Thus, it is observed that all students with number order: 5, 21, 37, 53, 69, 85 will occupy the same chair in the laboratory, the one with number 5 (only that they participate in different days and hours at the laboratory activity in the lab room; so there are 6 links: the number of students who " attempt" to occupy the same chair in the lab room.

Example 2: If the cache memory has 4 slots, then the main memory will map block 0 into slot 0, block 1 into slot 1, block 2 into slot 2, block 3 into slot 3 and then again: block 4 into slot 0, block 5 into slot 1 and so on.

This example is illustrated in Figure 4: on the left, you can see a main memory with only 8 blocks (numbered from 0 to 7 - notice the 3-bit binary number: from 000b to 111b) which is mapped into 4 cache slots (numbered with the help of 2 bits, from 00b to 11b); it is noticed that in total, there are 2 possible links to the same cache slot.

However, in the figure on the right, the number of links is 4: there are 4 possible blocks from the main memory that could point to the same cache slot.

Thus, towards slot 2, it is observed that blocks which may attempt are: 00***10***b, 01***10***b, 10***10***b, 11***10***b

The encoding used is: any combination of bits, followed by ***the number of the slot encoded in binary***.

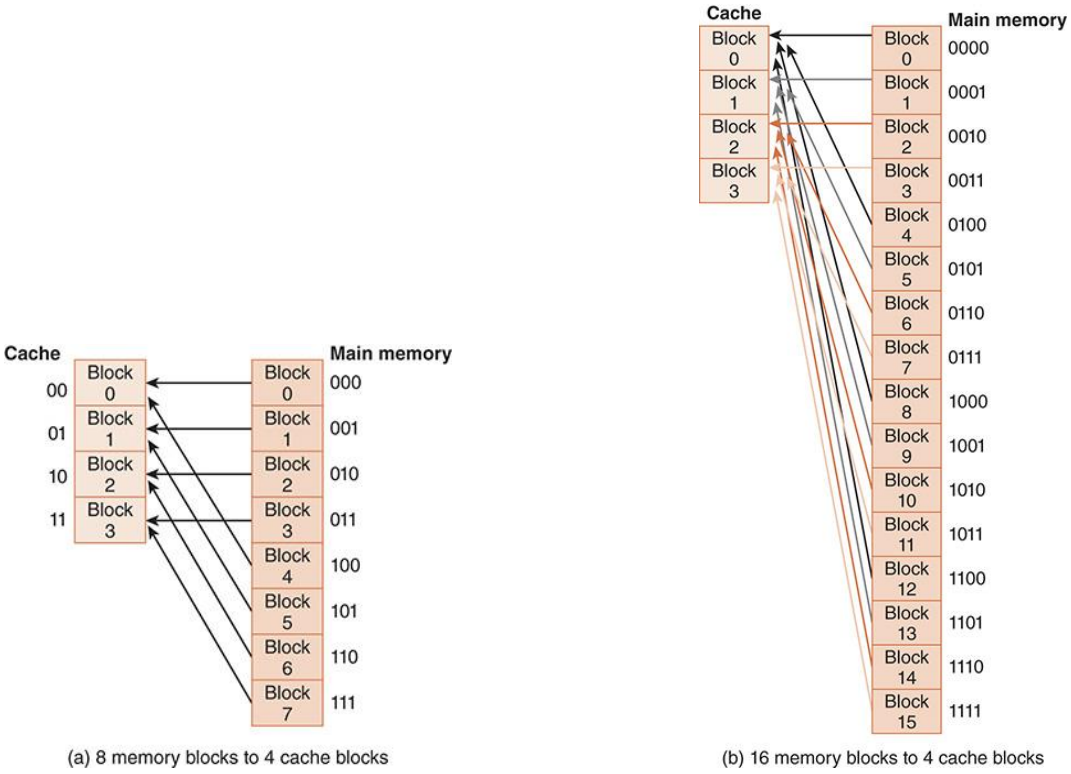


Figure 4. Mapping of blocks from main memory to cache for ***direct-mapped scheme***

Address partitioning:

In order that the CPU to know exactly which block from the main memory resides at one point in a particular cache slot, the so-called "address partitioning" is performed. This is nothing else than a division of an address from main memory into 3 fields, each aiming to identify the position of the information required by the CPU, both in cache and in main memory.

At the direct-mapped scheme, **the address from main memory** is partitioned in 3 fields, as Figure 5 shows (address is written in binary):

tag | **bloc** or **slot**[4] | **word** or **offset**[5] .

Main memory address		
Tag field	**Cache Slot field**	**Word field**

Figure 5. The format of the address from main memory at ***direct-mapped scheme***

The size of each field depends on the size of the main memory, cache, and one block, as follows:

- The size in bits of the whole address is given by the size of the main memory: if we have a main memory of 2^n B, then ***the address will be written on n bits***.

[4] Instead of the name "Slot" (to designate a cache line) you can often find the term "Block". We will adopt "SLOT" whenever possible.

[5] Sometimes, instead of the word "word" (to designate the column within a cache line) the term "offset" can be found. We will adopt "WORD" whenever possible.

Example Camera[6]: in Camera, there is a main memory of 256 bytes, so the address will be written using **8** bits;

- The size in bits of the ***slot*** field is given by the number of slots within the cache.

  Example: in Camera, there are 16 slots, so the size of the slot field is **4** bits (from the **8** bits, **4** bits will encode "the slot" in which there exists the information required by the CPU in cache);
- The size in bits of the ***word*** *or* ***offset*** field is given by the number of bytes within a cache slot; through this field, one can identify the exact position of the data within that slot (it represents the column from one cache line);

  Example: in Camera, we have 8 bytes in a block or slot, so we will use **3** bits to encode the information through which we know exactly the location or "the word" on which is the data requested by the CPU within that slot;
- The size in bits of the ***tag*** field is given by the number of links between the blocks from main memory and a specific slot of cache, or in other words: from how many areas from the main memory can come to a certain cache slot;
- In Camera, there are 2 links for the direct-mapped scheme, so it is enough one single bit to encode this information; the size of the tag field may also be computed by subtracting from the total number of bits, the number of bits obtained for the other 2 fields:

8-4-3=1

When a block from the main memory is copied in cache, the value of the tag field is also copied to a small separate memory, called "tag memory" (so an accurate record of all tag values of existing blocks in the cache is kept), and so, the CPU will know exactly from which part of the memory that block came from (1 bit -> from the first or second half?; 2 bits -> from which quarter did it come?, etc.).

Example Camera: if the CPU asks the information from the **address 6Ah**, then it will be in cache inside **slot 13**; within that slot it will be known that the byte required by the CPU is in **word 2,** and the tag field will say that the block came from **the first half of memory**:

6Ah=01101010b = **0** | **1101** | **010**b ; the Id of the block from the main memory can also be found, by concatenating the first 2 fields:

0 | **1101** => **block 13**.

If it was the **reference EEh,** for example, then the block to which it belongs would attempt exactly at **the same cache slot** and would overwrite the block that existed there (block 13), because:

EEh= 11101110b = **1** | **1101** | **110**b, where we observe the attempt to the same slot, but the respective block is arriving from **the second half of the memory**, being the block **1** | **1101** => **block 29**; the **offset** where the information required by the CPU is located is **6** in this case and not **2**, so we get it from another column (but this information does not count here when discussing slot overwriting).

Let **M** = number of bytes from main memory, **N** = number of bytes from cache, and **K** = number of bytes within a block or slot.

The interpretation of the address (by the MMU-Memory Management Unit) is done by dividing it into 3 fields:

- The length[7] of the field ***word***=$\log_2$K, where K is the size of the block (expressed in number of "words", here bytes)
- The length of the field ***slot***= $\log_2$N/K, where ***N/K*** is the size of the cache memory (expressed in number of slots), u=N/K.
- The length of the field ***tag***= $\log_2$((M/K)/(N/K)) = $\log_2$(M/N), where ***M/K*** is the size of the main memory (expressed in ***number of blocks***), and ***N/K*** is the size of the cache memory (expressed in ***number of slots***).
- The number of bits of the address from the main memory = $\log_2$(M).

The direct mapping scheme is the most economical as concerns the implementation costs, as it does not require any search algorithm (the processor knows exactly which slot to look for, it will only compare bits in the tag field, once the slot has been identified by comparing bits in the slot field), but its major disadvantage is the high restrictivity. Due to the existence of tight links between the blocks from the main memory and only a certain cache slot, different scenarios are possible where although there are a lot of other free slots in the cache, however the information in a specific cache slot is overwritten.

For example, a program could run pieces of code that are in 2 different blocks of the main memory pointing to the same slot: suppose the diagram in Fig. 4 on the right, in which the program repeatedly switches: it uses information (suppose in the form of a single reference) from block 2, then information from block 6 and so on (the 2 blocks point to the same slot numbered 2). In this way, when in the cache in slot 2 is written the content of block 2, the CPU requests information from block 6, so the slot will be overwritten with the information from block 6, the information from block 2 being erased from the cache. Then the CPU requests information from block 2, so block 6 from slot 2 of the cache is overwritten with the information from block 2 and so on. In the end, it will result only many misses and no hit.

Considering the analogy with the students in the lab room, it is as if a colleague of Pop Ion, for example Rusu Mihai (who occupies the same chair as Ion in the lab but from another hour), would like to come with the semi-group of which Ion belongs. Because there is a new student who attempt to occupy that chair, Ion will be kicked out from the lab and Mihai will occupy his place, although there may be other vacant chairs in the lab. Even other free seats exist in the lab room, neither Mihai nor Ion are allowed to occupy them. Sounds absurd, doesn't it? This is the major disadvantage of direct-mapped cache and because of this, it is not the mapping method that has been implemented in real systems for caching operation.

[6] „Camera" is the name of the simulator we will use in the laboratory.

[7] All these "lengths" are expressed in number of bits.

2) Fully associative mapping scheme

The fully associative mapping scheme tries to remedy this problem, being the most permissive scheme. For this type of mapping, there is no rule that deals with the slots (if they are empty) from the cache. Any block from the main memory can reach any slot in the cache, but generally deals in the following order: first free slot, second free slot, third free slot, etc. So in this scheme, there is maximum permissiveness: the blocks can be copied anyway in cache, in any order (we have no way to know the order in which the information from the main memory is requested by the CPU): so at a certain point, in the cache, there can be any block, in any cache slot.

The number of links to a cache slot in this case is equal to the number of blocks = 32 (in the Camera simulator): any of the 2^5 blocks from the main memory can be mapped to a cache slot.

Considering the analogy with the students, it is as if the students are allowed to occupy any chair in the laboratory room (but generally they will occupy the first free, then the second free and so on) as they enter in the lab room (we do not know which students intend to participate in the lab session). And now, if anyone were to ask the lab teacher again: is the student Pop Ion in the lab? Then, the lab teacher will have to individually check each chair in the room and look for Ion, and his answer may be much delayed for this reason (however, in practice, all 16 possible locations are being sought in parallel).

To find a block in this mapping scheme, all the slots are searched in parallel, which requires that the cache be built from a type of associative memory, which requires special hardware (additional comparators) and therefore is more expensive to implement. This type of memory is expensive to implement also because of the set of multiplexers used to select that data (once it was located in cache).

Thus, the tag of the address from main memory is compared with all the tags from cache to determine whether that block containing the information required by the CPU is in cache or not.

The problem that arises here is when the cache is filled: once the cache is full (no slot is left empty) and a new block should be brought in cache, in which slot will the information be overwritten? The block that will be overwritten in this case is called a "victim block". Of the commonly used schemes, we mention: **LRU** (Least Recently Used), **LFU** (Least Frequently Used), **FIFO** and **Random**. The **LRU** algorithm is implemented in the Camera, whereby the slot containing the block with the oldest information will be overwritten.

Address partitioning:

In the fully associative mapping scheme, there is no slot field (as it was at the direct-mapped scheme) because of the permissiveness used here: any block can reach anywhere in the cache. Thus, when partitioning the address, there will be only 2 fields, as shown in Figure 6: the ***tag*** field and the ***word*** *or* ***offset*** field.

Main memory address	
Tag field	Word field

Figure 6. The format of an address from the main memory at ***fully associative mapping***

Now resuming the **Example Camera** given to the direct mapping cache, we will have:

If the CPU requests the information from **address 6Ah**, then it will map this in **any slot of the cache**, depending on the number of slots previously occupied. Within that slot it will be known that the byte required by the CPU is in **word 2** (the column); the tag field will be able to identify exactly the id of the main memory block: **01101** => **block 13**.

6Ah=01101010b = **01101** | **010**b ;

If next, the CPU will access the **reference EEh**, for example, then the block containing this will attempt to **the next free slot from cache** (nothing will be overwritten as long as there are free slots in the cache).

EEh= 11101110b = **11101** | **110**b, where we notice that the block in question came from memory as **block 29**; the **offset** where the information required by the CPU is located is **6** in this case and not **2** (but this does not count here when discussing the occupied slot).

When looking for the existence of a particular block in this type of cache, all the values of the tag field (stored in the ***tag memory***) will be compared with the value of the tag field of the reference required by the CPU: if 2 identical values are found, then a cache hit results, otherwise a cache miss results and in what follows, the block from main memory will be transferred (copied) in cache.

Because this mapping scheme has to provide a high search speed and is also very complex as concerns the organizational mode (comparators, multiplexers, etc.) its implementation through hardware is very expensive. In addition, the ***tag memory*** is much larger (5 bits vs. 1 bit) compared to the one from the direct mapping, so it occupies more space.

3) Set associative mapping scheme

To overcome the disadvantages of the first 2 mapping schemes, the set associative mapping scheme has been proposed, which is actually a combination of the first 2: it combines the advantages of both direct mapping cache (links are also drawn here: not to a single slot, but to a set of slots), as well as of the fully associative mapping cache (there is freedom of placement within the respective set). This type of cache is also said to be n-ways: the number of ways means nothing else than the number of slots within a set. All sets have the same number of slots. For example, in the Camera simulator, a 2-way scheme can be chosen, which means that out of the 16 slots, 8 sets will be formed, each set having 2 slots inside.

Address partitioning:

At the set associative mapping scheme, **the address from main memory** is partitioned also in 3 fields, like Figure 7 shows (the address is written in binary): **tag** | **set** | **word** or **offset**.

Main memory address		
Tag field	**Set field**	**Word field**

Figure 7. The format of an address from the main memory at ***set associative mapping***

From the analogy with the students who arrive in the laboratory room, it is as if each student (of the 16x6 = 96 as many attending the lecture course) is allowed to occupy a certain set of 2 chairs in the lab room (he or she was told directly at the beginning of the semester, which is the working desk he or she will be working on, and at each working desk there are 2 chairs) and he or she are not allowed to occupy any chair from any other working desk.

Now, if anyone were to ask the lab teacher again: is the student Pop Ion in the lab? Then, the lab teacher will have to individually check each of the 2 seats in the set where Ion should be located. Ion could sit on any of the 2 chairs, and moreover, he could be a "set" colleague with Mihai, but the searching is also fast because Ion could stay at a single (specific) working desk.

Example Camera:

For a 2-way set associative mapping scheme, which means that out of the 16 slots, 8 sets will be formed, each set having 2 slots inside, if the CPU requests the information from **address 6Ah**, then it will be in cache in **any of the 2 slots** (depending on the number of slots previously occupied), of **set 5**. Within that slot occupied from the set, it will be known that the byte required by the CPU is in the **word 2**; the tag field together with the set field will be able to identify exactly the id of the main memory block: **01101** => **block 13**.

6Ah=01101010b = **01** | **101** | **010**b ;

If next, the CPU will access the **reference EEh**, for example, then also the block comprising this will be copied in **the next free slot** of the same **set number 5** (nothing will be overwritten as long as there are free slots in the respective set).

EEh= 11101110b = **11** | **101** | **110**b, where we notice that the respective block arrived from main memory as **block 29** and it attempts also to **set number 5**, and if there is an empty slot within this set, it will be copied there, in the exact order they were requested by the CPU. The **offset** where the information required by the CPU is located is **6** in this case, not **2** (but this does not matter here when discussing the position of the block inside the cache set).

The formula from the direct mapping scheme is also valid here, but adapted to the number of sets instead the number of slots: ***the block numbered X*** from the main memory can reach ***the set numbered Y*** in the cache, where **Y = X mod v**, the value ***v being the total number of sets from the cache***.

Also here, the replacement policy is **LRU**, like in the case of fully associative mapping.

For the LAB:

a) Students will run the CPU-Z application on their own system, will make a capture and will analyse information about cache and main memory as in the Example 1.

b) Students will install Camera simulator on their own system and will try to solve the proposed exercises (pages 10-11).

Study the following solved problems with cache memory:

Ex1. Consider the following case of a system with 2048 cache slots and 8192 main memory blocks. Find out where in the cache the main memory blocks Bl.15 and Bl.2029 will be placed for the mapping policies of: direct mapping, fully associative mapping, 4-way set associative.

Solution

1. Bl.15

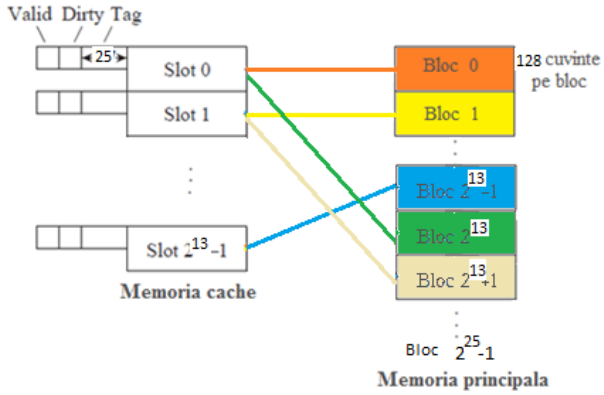
Direct mapping: This block will be copied to Sl.15.

Fully associative: This Bl.15 block can be copied to any cache slot which is empty.

Four-way set associative: In four-way set associative, the cache will be divided into sets of 4 blocks. Thus, there will be 2048/4 = 512 sets in the cache => 15 mod (512) = 15 i.e., Bl.15 can be copied to any of the four cache slots of Set 15.

2. Bl. 5029
Direct mapping: 5029 mode (2048) = 933. This block will be copied to Sl.933.
Fully associative: This block can be copied to any cache slot.
Four-way set associative: 5029 mod (512) = 421. This block can be copied to any of the four cache slots of Set 421.

Ex2. A system with **4Gbytes main memory** has a direct mapped cache memory of capacity **1MB** and each cache slot (or cache line) has **128 words** (locations).
a) find the number of slots from the cache and the number of blocks (or pages) from the main memory.
b) how many blocks (or pages) could be mapped in each cache slot (or cache line) ?
c) draw a figure to show the organization between the main memory and the cache memory
d) Explain below how is the address partitioned in such a system
e) Explain in detail (also make a draw) where the address 12345678h will be located
e_1) in the main memory: _____
but also e_2) in the cache: _____
Solution: a) 1MB=2^{20} => 2^{20}:2^7=2^{13} cache slots or lines, numbered Sl0, Sl1, …Sl 2^{13}-1.
Cache memory contains: 2^{13} slots in which we can place blocks of the main memory
A space of 2^{32} words from main memory will be "divided" in 2^{25} blocks of 2^7 = 128 words per block (25+7=32)
b) there are much more blocks than slots, so $2^{25}/2^{13}$ =2^{12} blocks from main memory which may be mapped in a single slot of cache (or 2^{12} blocks from main memory attempt to the same cache slot)
c)



d) The address is partitioned in 3 fields, called tag | slot | word (or offset).
Since the main memory has a capacity of 4GB, we get the address from main memory expressed with 32 bits. Out of these 32 bits, 7 of them will encode the word or offset field (because there are 128 words in a slot or block), 13 of them will encode the slot field (because there are 2^{13} slots) and the remaining 32 – (13+7) = 12bits will encode the tag field. The size of the tag field can also be found as the number of connections between main memory blocks and a cache slot (how many blocks attempt to the same slot)
tag | slot | word (or offset) → 12 bits | 13 bits | 7 bits.
e) where the address 12345678h will be located (or address partitioning scheme):
we must write in binary the address from main memory and divide it in the corresponding fields:
12345678h=0001 0010 0011 0100 0101 0110 0111 1000b,
so the requested content will be found in cache memory inside slot 0 1000 1010 1100b=08ACh, at the offset 111 1000b=78h, and we can say that it was copied from main memory from that area having the tag value 0001 0010 0011b=123h. Inside main memory, it is located in block number 0001 0010 0011 0100 0101 0110 0b = 0 0010 0100 0110 1000 1010 1100b=02468ACh, at offset 111 1000b=78h.

Ex3. Consider computing the hit ratio (HR) and the effective access time (EAT) for a benchmark program running on a computer that has a ***direct mapped cache*** with **four 16-word slots**. The relation between main memory and cache is shown in Figure 1. Suppose the cache access time is 80 ns, and the time for transferring a main memory block to the cache is 2500 ns. Also, assume that the cache is initially empty and load-through mechanism is used. The test program executes instructions from the main memory in the following order: it begins at memory location 48, then go forward to the next location and so on until reaches location 95, and then loops 10 times the instructions between location 15 to 31 before halting. The occurred events (as the program executes) have been registered and organized in Table 1. Please complete the table and then proceed to obtaining HR and EAT values, knowing that HR(HitRatio) is the number of successful accesses (or attempts) reported to the entire number of accesses (or attempts). **EAT (Effective Access Time)** represents an average time, used to execute an attempt, so it is defined as the sum between the number of successful attempts multiplied with the time required for an successful attempt and the number of unsuccessful attempts multiplied with the time required for an unsuccessful attempt, all divided to the entire number of attempts.
Simpler, these are: **HR=nb.of hits/ nb. of attempts** and **EAT=(nb of hits *$time_{hit}$ + nb of misses *$time_{miss}$)/ nb of attempts**

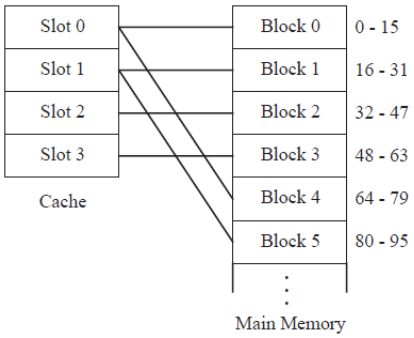


Figure 1. Direct mapped cache

Table 1. A table of events for a program executing on an architecture with a small direct mapped cache memory.

Event	Location from main mem.	Time consumed	Observations
miss	48	2500 ns	BI 3-> Slot 3
15 hits	49-63	15 x 80 ns	
.			
.			
.			

Solution:

Since the memory is initially empty, the first instruction that executes causes a miss.

A miss will occur at location 48, because the CPU requires the information from location 48 in the main memory, but cache is empty (it would have been better that cache contained it); since cache is empty, a miss result; when a miss is registered, the CPU will command the cache controller to update its content, so cache will copy the entire block from the main memory where location 48 is within, in the corresponding cache slot. This miss at location 48 will thus causes main memory block #3 to be read into cache slot #3. This first memory access takes 2500 ns to complete. Load-through is used for this example, and so the word that causes the miss at location 48 is passed directly to the CPU while the rest of the block is loaded into the cache slot. The next event consists of 15 hits (for locations 49 to 63, because now cache contains their content in slot #3). The events that follow are recorded in a similar manner, and the result is a total of 213 hits and 5 misses. The total number of accesses is 213 + 5 = 218. The HR and EAT are computed as shown below: HR=213/218=97.7% and EAT=(213*80ns+5*2500ns)/218=136 ns

Although the hit ratio is 97.6%, the effective access time for this example is almost 75% longer than the cache access time. This is due to the large amount of time spent in accessing a block from main memory.

Mandatory exercises (so I can put you present at the LAB 4 activity):

Each student should provide a pdf file named: TSTe_Poplon_L4_P.pdf or EAe_Poplon_L4_P.pdf (where the name is given as an example).
Please Upload the files in TEAMS.

1. A) Like **Fig.2** from the *LAB Paper* shows, obtain the corresponding ***picture*** on your own system and insert it here in the left side of the paper. On the right side, please provide a similar ***picture*** showing how much RAM memory do you have in your system ("memory" tab in the CPU-Z application).

Your figures go here

B) For the values obtained at point A), please do compute the following[8]:
- the entire size of L1 cache (data and instructions) on your system (in bytes):
- the size of L2 cache on your system (in bytes):
- the size of L3 cache on your system (in bytes):
- the size of RAM memory on your system (in bytes):
- find the ratio between size of L1 and size of L2:
- find the ratio between size of L2 and size of L3:
- find the ratio between size of L3 and size of RAM:

C) Like **Example 1** from the *LAB Paper* shows how to compute the cache system parameters for L3 cache, do the same for your system:
The parameters of your cache L3 system are (please use powers of 2 whenever possible and the notations M, N, K, u, v, … as in the *LAB Paper*):
- the size of the RAM memory: (bytes);
- the size of the cache memory: (bytes);
- the size of one block from the RAM and the size of one slot from the cache: (bytes);
  - nb_of_blocks:
  - nb_of_slots:
- the mapping scheme:
  (if a set-associative cache, then also compute …)
  - nb of ways:
  - nb of sets:

[8] Please use powers of 2 whenever possible

2. Install the **Camera simulator** on your system and then:

A) Use the ***direct-mapped cache*** and press the "**Auto Generate Address Reference String**" **2 times**, such that 20 references will be generated. Write them here as a string of values: _____

B) For each of the first 5 references from point A), do the ***address partitioning***, by completing the table:

Direct-mapped cache:

Reference (hex):	Reference (binary):	Block (binary and hex)	Tag (binary and hex)	Slot (binary and hex)	Word (binary and hex)

Fully associative cache:

Reference (hex):	Reference (binary):	Block (binary and hex)	Tag (binary and hex)	Slot (binary and hex)	Word (binary and hex)

2-ways set associative cache:

Reference (hex):	Reference (binary):	Block (binary and hex)	Tag (binary and hex)	Slot / set (binary and hex)	Word (binary and hex)

4-ways set associative cache:

Reference (hex):	Reference (binary):	Block (binary and hex)	Tag (binary and hex)	Slot /set (binary and hex)	Word (binary and hex)

C) For each of the 3 mapping scheme, use the "**Self Generate Address Reference String**" option in order to introduce the string with 20 references from point A in Camera; complete the table with **Hit** or **Miss** in each cell and then count them on column:

Reference	**Direct-mapped cache**	**Fully associative cache**	**2-ways set associative cache**	**4-ways set associative cache**
1)				
2)				
3)				
20)				
TOTAL nb. of hits:				

D) For each of the 4 cases from point C), propose 5 additional references such that a minimum of 5 hits to result. Highlight these 5 new references (a different color). Write down the entire reference strings here:

- for Direct-mapped: _____ => Hit Ratio = ____/25 = ____
- for Fully Assoc.-mapped: _____ => Hit Ratio = ____/25 = ____
- for 2-ways set Assoc.-mapped: _____ => Hit Ratio = ____/25 = ____
- for 4-ways set Assoc.-mapped: _____ => Hit Ratio = ____/25 = ____