Introduction to Computer Organization (Cache Memory)

KR Chowdhary
Professor & Head
Email: kr.chowdhary@gmail.com
webpage: krchowdhary.com

Department of Computer Science and Engineering
MBM Engineering College, Jodhpur

November 14, 2013

Introduction to cache

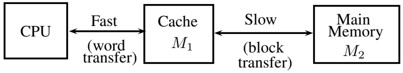


Figure 1: Memory Hierarchy.

*Speed of CPU v/s Main memory is typically 10:1
*A high speed Cache memory (M_1) of relatively small size is provided between main memory (M_2) and CPU forming (M_1, M_2) hierarchy. Some mechanism is provided in cache so that $t_A \approx t_{A_1}$($\Rightarrow$ cache hit ratio ≈ 1) .
* Data and instructions are pre-fetched in cache. Due to locality of reference, other instructions/data are likely to be found cache.
* Cache is usually not visible to programmers.
* Approach: One can search a desired address's word in a cache for match; and does this for all locations in cache simultaneously.
*Cache size should be small: average cost is that of RAM. Cache should be large enough: average access time is close to that of cache.
*VAX 11/780: 8-KB general purpose cache, Motorola 68020: 256 byte on-chip instruction cache, 8086: 4 / 6 byte instruction cache, Dual core: L_1 : 512KB-2MB, L_2 : 1-2MB. Instru. cache v/s GP cache ?

Cache Principle

- Principle: When processor attempts to read a word, its presence is checked in cache first. If found, it is delivered to processor, else a block of fixed no. of words is read into cache which contains this word also. Simultaneously, the word also is delivered to cpu.
- If the word was found in cache, it is **Cache hit** else it is **Cache miss.**

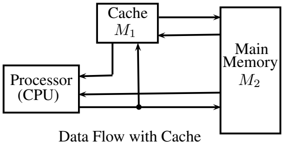


Figure 2: Cache position.

- Cache Design

  Let no. of **words** in RAM $= 2^n$, each addressable by n-bits, cache size m, RAM of M blocks. Word size may be 32, 64, ... bits.

- Memory consists fixed length blocks, each of size K words($= 2^w$), total blocks are $M = 2^n \div K$. Cache has $m = 2^r$ blocks or lines. Each line has 2^w words, each having a **tag field** and **ctrl-field**(to show if line has been modified).

- **tag** identifies which particular blocks are presently in cache.

Cache Principle

Mapping Function: Since lines in cache, $m \ll M$ (blocks in RAM), an algorithm is needed to map M to m. That determine what memory block occupies what cache line.

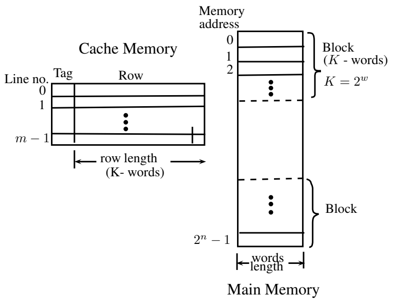
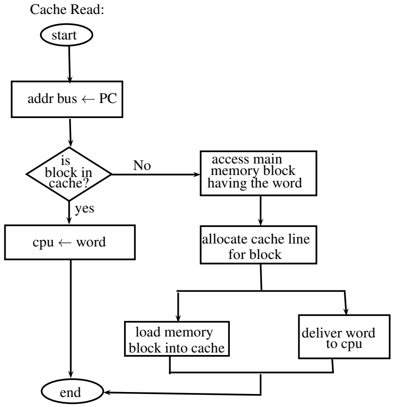


Figure 3: Address mapping.



Cache Read:
start
addr bus ← PC
is block in cache?
No
access main memory block having the word
yes
cpu ← word
allocate cache line for block
load memory block into cache
deliver word to cpu
end

Figure 4: Accessing a word.

Three mapping techniques: *direct, associative, set associative*.

Direct-Mapped Cache

- Each block maps to one and only one line in cache always. The mapping is expressed as:

$$i = j \mod m$$

  where, j is main memory block no., i is cache line no., m is no. of lines in cache.

- Words address space = n-bits. RAM is divided into fixed length blocks of $K = 2^w$ words each. A word may be equal a byte. $\therefore$ number of blocks in RAM $= 2^n/K = M = 2^s$ (s is number of bits required to represent block no.) Let $m = 2^r$ is number of rows in cache, each of size one block $= 2^w$ words. r is no. of bits required for row count. The address space in RAM is:

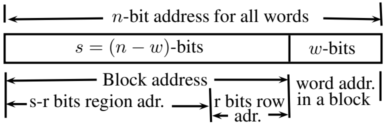


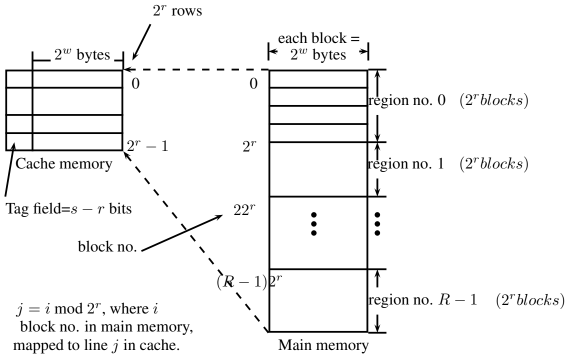
Figure 5: Address Space in RAM.

- There is need of some kind of mapping between 2^r rows in cache v/s 2^s no. of blocks in RAM, as only 2^r out of 2^s blocks can be resident in the cache (a block fits in a row).

Direct-Mapped Cache

- Direct mapping: If $2^S \bmod 2^r = 0$, then RAM is divided into $2^s \div 2^r = R$ regions as follows:
  1. region 0's blocks nos. : $0, \ldots, 2^r - 1$, mapped to $0, \ldots, 2^r - 1$ lines of cache,
  2. region 1's blocks nos.: $2^r, \ldots, 2.2^r - 1$, mapped to $0, \ldots, 2^r - 1$ lines of cache,
  3. $\ldots$
  4. region $R - 1$'s block nos. : $R - 1 \times 2^r, \ldots, R.2^r - 1$, mapped to $0, \ldots, 2^r - 1$ lines of the cache.
- if 2^s is total no. of blocks in RAM, and 2^r is no. of rows in cache, with row size == block size=2^w words, then i^{th} block of RAM is mapped to j^{th} row of cache as follows, so that $j = i \bmod 2^r$.
- E.g., row 5 in cache will hold one block (5th block) from only one of the regions out of $0, \ldots, R - 1$.
- Thus, a row address corresponding to r bits from address field is a direct index to the block position in a region.

Direct-Mapped Cache

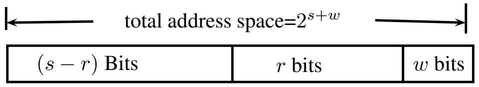


2^r rows
2^w bytes
each block = 2^w bytes
0
0
$2^r - 1$
2^r
22^r
$(R-1)2^r$
region no. 0 ($2^r blocks$)
region no. 1 ($2^r blocks$)
region no. $R-1$ ($2^r blocks$)
Cache memory
Tag field=$s-r$ bits
block no.
$j = i \bmod 2^r$, where i block no. in main memory, mapped to line j in cache.
Main memory

Figure 6: Address mapping.

- For a match, r bits (i.e., index, having value, say, 1011) is taken from address field, and the *tag* field in row no. 2^{1011} of cache is compared with the $s - r$ bits of address field. If matched (equal), then it is *cache hit*, else *cache-miss*.

Direct Mapped Cache



The r bits acts only as index to a row in cache.
If the tag field of cache macthes with $s-r$ bits of addr space for an index, it is hit.
It requires only one comparision per address in the direct mapped cache.

Figure 7: Address mapping.

- Only one comparison required to test for hit, hence it requires simple circuitry to implement.
- Thus, if there is frequent reference to blocks from different regions, it will cause **thrashing**, and cache functioning will be inefficient.

Example: Direct Mapped Cache

- Let $n = 32$, $= 4GB$, $w = 4$. One block size, $K = 2^4 = 16$ bytes. No. of total blocks$=2^{32-4} = 2^{28}$=256 million. Size of cache $= 4MB$, $m = 4MB$, No. of rows in cache=4MB/16 bytes$=2^{22}/2^4 = 2^{18}$. Total no. of regions R $=$ (total blocks)/(blocks in cache)$=$ $2^{28}/2^{18} = 2^{10}$.
- $\therefore n = 32, w = 4, r = 18, s = n - w = 32 - 4 = 28.$ Tag.$=s - r = 28 - 18 = 10.$
- For address:

  1 0 0 0.1 1 1 1.1 0 0 1.0 1 1 0.1 1 0 0.1 0 0 1.1 0 1 0.0 0 0 1

  1 0 9 8.7 6 5 4.3 2 1 0.9 8 7 6.5 4 3 2.1 0 9 8.7 6 5 4.3 2 1 0

  block addr$=(A_0 - A_3)$=1, line addr$(A_4 - A_{21}) =$ $2^r - 1 = 010110110010011010_2$, region adr. $=A_{31} \dots A_{22}$ $=$ tag bits.
- This line addr is common for all the 2^{10} regions. Thus, if tag bits in the cache for line no. 010110110010011010_2 matches with the region no. address $A_{31} \dots A_{22}$, then this block $(A_4 - A_{21}) = 010110110010011010_2$, is in cache else not.
- Line in cache is, $i =$ j mod m $= (A_{31} - A_4) mod 2^{18}$ $=$ 1 0 0 0.1 1 1 1.1 0 0 1.0 1 1 0.1 1 0 0.1 0 0 1.1 0 1 0 mod 2^{18} $=$ 0 1.0 1 1 0.1 1 0 0.1 0 0 1.1 0 1 0

Block Replacement Algorithm

- Objective: Every request for instruction/data fetch should be met by the cache. Blocks transfer to cache will improve the hit. *Hit ratio* = (total hits)/(total attempts), which should be close to unity (normally > 0.99).
- The objective is partly achieved by algorithm used for block replacement.
- What happens when request for loading new block in cache does not find space in cache?
- One of the row (block frame or line) is to be moved back to main memory to create space for new block.
- Which block should be removed?:
  1. The one which has remained in the cache for a longest time, called FIFO (first in first out) algorithm.
  2. The one which was used long back: LRU (least recently used) algorithm.
  3. The one which is not frequently used: LFU (least frequently used) algorithm.
  4. Random time: a random no. is generated, and a block is selected correspondingly for removal (random block selected)).

Block Replacement Algorithm

- A block should be written back to main memory only if it is changed, otherwise not.
- For each row in cache, which corresponds to a block, a special bit (sticky bit or dirty bit, as it is called) is allocated. If the block is modified, the bit is set else remain reset. This bit helps to decide whether to write this line back to RAM or not.

How much you have learned in direct mapped cache?

1. If two same block numbers from two regions are required in the cache, what is consequence? That is, can they be present same time in cache?
2. Does the cpu fetch directly from RAM, if cache is full?
3. A block can be resident at how many alternate locations in cache?

Associative Cache Memories

- Many storage and retrieval problems require accessing certain subfields within a set of records, *what is John's ID no. and age?*

```
Name    ID number Age
---------------------
J. Smith 124      24
J. Bond  007      40
A. John  106      50
R. Roe   002      19
J. Doe   009      28
--------------------
```

- The row 3 has no logical relationship to Jones, hence to search entire table using name sub-field as an address. But, that is slow.
- The associative memory simultaneously examines all the entries. They are also known as content addressable memories. The subfield chosen is called *tag* or *key*.
- Items stored in Associative memories can be viewed as:

  KEY, DATA

Other cache types

Associative mapping Cache Memories:

- A memory block $(0, \ldots, 2^s - 1)$ of main memory can go (mapped to) anywhere (any line) in cache. This removes the drawback of Direct mapped cache. Cache of size 2^r lines (r bits) may hold any 2^r blocks out of total 2^s blocks of RAM. Larger is size of cache, the larger no. of RAM blocks it can hold.
- Since any of 2^r blocks out of 2^s blocks may be there in cache, we need a tag field of s bits along with each line of cache.
- Steps to find out if a block no. $b = 2^s$ is in cache:
  1. Total address space $=$ $s + w$ bits, there are 2^s blocks in RAM and each blocks has 2^w words.
  2. s bits from address space are compared (**in parallel**) with all the tags of 2^r lines of cache. The line in which match occurs, holds the block having required word corresponding to the address field. If no match occurs with any tag, it shows that it is cache-miss, and the required block is fetched from RAM.

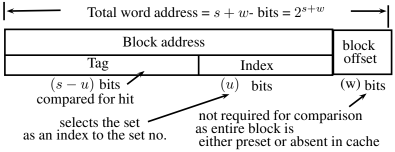
Set Associative Cache

Set Associative cache memories:

- If there are ten Cache lines to which a RAM location are mapped, ten cache entries must be searched. It takes power, chip area, and time. Caches with more associativity has fewer misses. The rule of thumb: Doubling the associativity, from direct mapped to 2-way, or from 2-way to 4-way, has same effect on hit ratio as doubling cache.
- A set-associative scheme is a hybrid between fully associative and direct mapped, and a reasonable compromise between complex hardware (of fully associative) v/s simple of direct-mapped.
- solution is set associative cache
- For a cache of size **8 lines**, and RAM of **32 blocks**. If the cache is fully associative, the block (say 12 no.) can go in any of the 8 rows of cache.
- In a set associative cache, the block 12 will be mapped to a set in the cache. Set can be of size 1, 2, 4 rows here. For two sets (each of 4-rows), block 12 will be mapped to a set, i.e., $\{1,2\}$. This is called 4-way set associative cache.
- If set is represented by u-bits in address field, then set no. can be found by index of u bits. The tag field of each row $= s - u$ bits.

Set Associative Cache

- Compromises: Requires complex comparison hardware (to find the correct slot) out of a small set of slots, instead of all the slots. Such hardware is linear in the number of slots(row in a set).
- Flexibility of allowing up to N cache lines per slot for an N-way set associative.
- Algorithm to find cache hit:
  1. Pick up the u bits out of total $(s-u)+u$ of block address; use the u bits as index to reach to 2^u th set in the cache.
  2. compare(in parallel) the $s-u$ bits from address field with tag fields of all the 2^{s-u} lines in that set.
  3. If any match occurs, it is hit, and line whose tag is matched, has the required block. So the byte from that word is transferred to CPU. Else, it is miss, and the block is replaced from RAM.



Total word address = $s+w$- bits = 2^{s+w}
Block address
block offset
Tag
Index
$(s-u)$ bits compared for hit
(u) bits
(w) bits
selects the set as an index to the set no.
not required for comparison as entire block is either preset or absent in cache

Analysis

- In fully associative, u length is zero. For a given size of cache, we observe following:
- tag-index boundary moves to right

  $\Rightarrow$ decrease index size u

  $\Rightarrow$ increase no. of blocks per set

  $\Rightarrow$ increase size of tag field

  $\Rightarrow$ Increase associativity
- Direct mapped cache is set associative with set size $=1$.
- tag-index boundary moves to left

  $\Rightarrow$ increases index size u

  $\Rightarrow$ decreases no. of blocks per set

  $\Rightarrow$ decreases in size of tag field

  $\Rightarrow$ Decrease in associativity
- What are the effect of:

1. Index boundary max left?
2. Index boundary max right?
3. How far it can go left/right?
4. What is best position of index boundary?

Write policy

- When new block is to be fetched into cache, and there is no space in cache to load that block, there is need to create space in cache. For this one of the old block is to be lifted off as per the algorithms discussed earlier.
- if a line or block frame is not modified, then it may be over written, as it does not require to update into RAM.
- if the block is modified in RAM (possible when multiple processors with multiple caches exist or it is modified by device) then cache is invalid.
- One of the technique to solve this problem is called write through, which ensures that main memory is always valid. Consequently if any change has taken place, it should be updated into RAM immediately.
  - disadv: Wastage of bandwidth due to frequent updating of RAM.
  - adv: Easier to implement, cache is always clean, read misses (in cache) never lead write to RAM.
  - RAM has most current copy of data, hence simplifies the data coherency

Write policy

- When write through takes place, the *cpu stalls*. To solve this, the data is written to buffer, before it goes to RAM.
- write back: The cache is written (updated) into RAM only when block is to be removed to create space in cache and the dirty bit of the cache line is found set.
  - Consumes lesser BW for writes.
  - Suitable for servers (with multiple processors).
  - Good for embedded applications as it saves power.

- Shared memory Architecture
- Two main problems with shared memory system: performance degradation due to contention, and coherence problems.
  - Multiple processors are trying to access the shared memory simultaneously.
- Having multiple copies of data, spread throughout the caches, might lead to a coherence problem.
- The copies in the caches are coherent if they are all equal to the same value.

Cache coherence

- Coherence(def.): A sticking or cleaving together; union of parts of the same body; cohesion.
  - Connection or dependence, proceeding from the subordination of the parts of a thing to one principle or purpose, as in the parts of a discourse, or of a system of philosophy; a logical and orderly and consistent relation of parts; consecutiveness.

  Coherence of discourse, and a direct tendency of all the parts of it to the argument in hand, are most eminently to be found in him. –Locke.

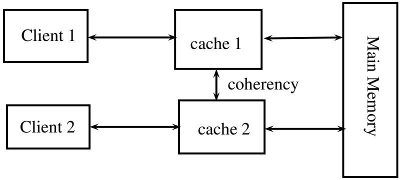


Figure 8: Multiple caches using a shared resource.

- Client 1 modifies the cache 1, but does not write to main memory
- client 2 accesses that memory block which was modified in the cache 1.
- result is invalid data access by client 2.

Achieving coherence

- Principle of coherence: client 1 reads location X, modifies it, and writes location X. In between no other processor accesses X. This results to memory consistency, else not.

Cache coherence mechanisms:

- Directory-based coherence: Data being shared is placed in a common directory, which acts as a filter through which processor must ask permission to load an entry from the RAM to cache. When entry is changed, the directory updates or invalidates other caches with that entry.
- Snoopy-bit: Individual cache monitors address lines of RAM that have been cached. When write operation is observed in a location (X) whose copy is with cache, the cache controller invalidates its own copy of that location (X).
- Snarfing: Cache controller watches both addr and data lines, and if RAM location (X) gets modified, the cache updates its own copy of (X).
- The coherence algorithms above, and others maintains consistency between all caches in a system of distributed shared memory.

Snooping Protocols

- They are based on watching bus activities and carry out the appropriate coherency commands when necessary.
- Each block has a state associated with it, which determines what happens to the entire contents of the block.
- The state of a block might change due to: Read-Miss, Read-Hit, Write-Miss, and Write-Hit.
- Cache miss: Requested block is not in the cache or it is in the cache but has been invalidated.
- Snooping protocols differ in whether they update or invalidate shared copies in remote caches in case of a write operation.
- They also differ as from where to obtain the new data in the case of a cache miss.
- State: Valid, Invalid.
- Event: Read-Hit, Read-Miss, Write-Hit, Write-Miss, Block replacement.
- **Write-Invalidate and Write-Through:** Memory is always consistent with the most recently updated cache copy.
- **Write-Invalidate and Write-Back**

Instruction Cache, data cache, Cache levels

- Harvard Cache: Separate data and instruction caches. Allows accesses to be less random.

  Both have locality of reference property.
- Multilevel cache hierarchy: Internal cache, external cache, L_1, L_2, L_3, etc.
- Cache Performance: 1. Hit Ratio, 2. Average memory access time.

- *Average memory access time = Hit time + Miss rate * Miss penalty(i.e. no. of clock cycles missed)*
- *CPU time = (CPU execution clock cycles + memory stall clock cycles) * clock cycle time*
- *Miss penalty = (memory stall cycles / instruction) = (Misses/Instruction)*(Total miss latency - Overlapped Miss latency)*
- Overlapped miss latency is due to out-of-order CPUs stretching the hit time.

Cache Positioning

- A cache is always positioned between RAM and CPU. A programmer is unaware of its position, because the most part of cache operation and management is done by hardware and a very small by operating system.

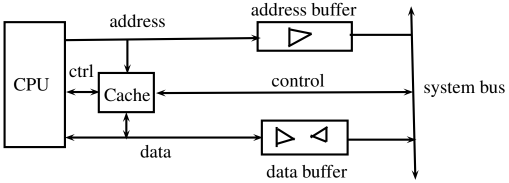
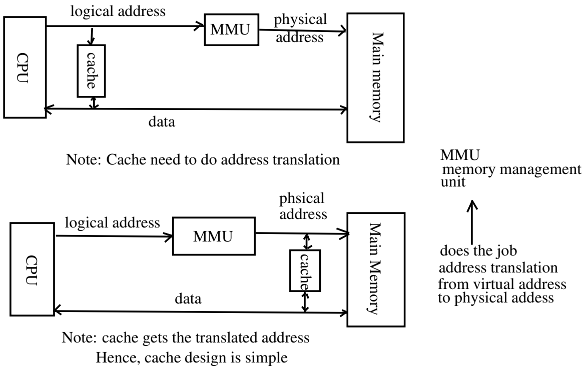


Figure 9: Cache positioning

Virtual to Physical Address Translation

- In virtual memory system, the logical address needs to be translated into physical address, before instruction/data is fetched.
- For cache memory this may be done by cache itself (in that case cache shall be fast) or by MMU (the cache will have to wait till the address translation is done).


logical address
MMU
physical address
CPU
cache
Main memory
data
Note: Cache need to do address translation
MMU
memory management unit
does the job address translation from virtual address to physical addess
phsical address
logical address
MMU
CPU
cache
Main Memory
data
Note: cache gets the translated address
Hence, cache design is simple

- In first, cache itself is doing address translation, in second address translation is done by MMU and cache see only the physical address.

Exercises

1. A block-set-associative cache consists of a total 64-blocks divided into 4-block sets. The main memory consists of 4096 blocks, each consisting of 128 words.
    1.1 How many bits are there in a main memory address?
    1.2 How many bits are there in each of the TAG, SET, and WORD fields?
2. A 2-way set associative cache memory uses blocks of four words. The cache can accommodate a total of 2048 words from main memory. The main memory size if 128K $\times$ 32.
    2.1 Formulate all the required information to construct the cache memory.
    2.2 What is size of the cache memory?
3. Let us consider a memory hierarchy (main memory + cache) given by:
    - Memory size 1 Giga words of 16 bit (word addressed)
    - Cache size 1 Mega words of 16 bit (word addressed)
    - Cache block size 256 words of 16 bit

Exercises

Let us consider the following cache structures:

- direct mapped cache;
- fully associative cache;
- 2-way set-associative cache;
- 4-way set-associative cache;
- 8-way set-associative cache.

- Calculate the structure of the addresses for the previous cache structures;
- Calculate the number of blocks for the previous cache structures;
- Calculate the number of sets for the previous set associative caches.

4. A cache memory is usually divided into lines. Assume that a computer has memory of 16 MB, and a cache size of 64 KB. A cache block can contain 16 bytes.
   4.1 Determine the length of the tag, index and offset bits of the address for: a. Direct Mapped Cache, b. 2-way set Associative Cache, c. Fully Associative Cache
   4.2 Assuming a memory has 32 blocks and a cache consists of 8 blocks. Determine where the 13th memory block will be found in the cache for: a. Direct Mapped Cache, b. 2-way Set Associative Cache, c. Fully Associative Cache

Exercises

5. Consider a machine with a byte addressable main memory of 2^{16} bytes and block size of 8 bytes. Assume that a direct mapped cache consisting of 32 lines is used with this machine.
   5.1 How is the 16-bit memory address divided into tag, line number, and byte number?
   5.2 Into what line would bytes with each of the following addresses be stored?
   0001 0001 1001 1011
   1100 0011 0010 0100
   1010 1010 0110 1010
   5.3 Suppose the byte with address *0001 0001 1001 1011* is stored in the cache.What are the addresses of the other bytes stored along with this?
   5.4 How many total bytes of memory can be stored in the cache?
   5.5 Why is the tag also stored in the cache?

Exercises

6. In a cache-based memory system using FIFO for cache page replacement, it is found that the cache hit ratio H is unacceptedly low. The following proposals are made for increasing H:
   6.1 Increase the cache page size.
   6.2 Increase the cache storage capacity.
   6.3 Increase main memory capacity.
   6.4 Replace the FIFO policy by LRU.

   Analyse each proposal to determine its probable impact on H.
7. Consider a system containing a 128-byte cache. Suppose that set-associative mapping is used in the cache, and that there are four sets each containing four lines. The physical address size is 32-bits, and the smallest addressable unit is the byte.
   7.1 Draw a diagram showing the organization of the cache and indicating how physical addresses are related to cache addresses.
   7.2 To what lines of the cache can the address $000010AF_{16}$ be assigned?
   7.3 If the addresses $000010A_{16}$ and $FFFF7Axy_{16}$ are simultaneously assigned to the same cache set, what values can the address digits x and y have?

Exercises

8. Discuss briefly the advantages and disadvantages of the following cache designs which have been proposed and in some cases implemented. Identify three nontrivial advantages or disadvantages (one or two of each) for each part of the problem:
   8.1 *An instruction cache*, which only stores program code but not data.
   8.2 A *two-level cache*, where the cache system forms a two level-memory hierarchy by itself. Assume that the entire cache subsystem will be built into the CPU.
9. A set associative cache comprises 64 lines, divided into four-line sets. The main memory contains 8K block of 64 words each. Show the format of main memory addresses.
10. A two-way set associative cache has lines of 16 bytes and a total size of 8 kbytes. The 64-Mbyte main memory is byte addressable. Show the format of main memory addresses.
11. Consider a 32-bit microprocessor that has an on-chip 16-Kbyte four-way set associative cache. Assume that the cache has a line size of four 32-bit words. Draw a block diagram of this cache showing its organization and how the different address field are used too determine a cache hit/miss. Where in the cache is the word from memory location $ABCDE8F8_{16}$ mapped?

Bibliography

- John P. Hayes, “Computer Architecture and Organization”, 2nd Edition, McGraw-Hill, 1988.
- William Stalling, “Computer Organization and Architecture”, 8th Edition, Pearson, 2010.
- M. Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education, 2006.
- Carl Hamacher, Zvono Vranesic, and Safwat Zaky, “Computer Organization ”, , 5th edition, McGrawhill Education, 2011. (chapter 7)