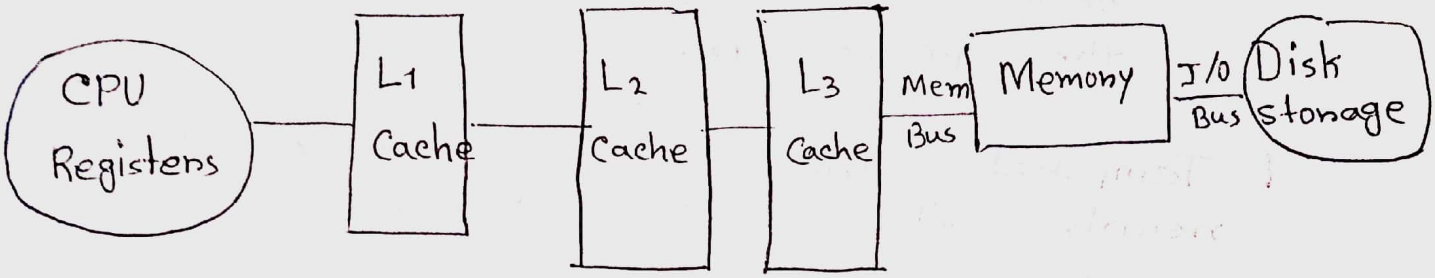
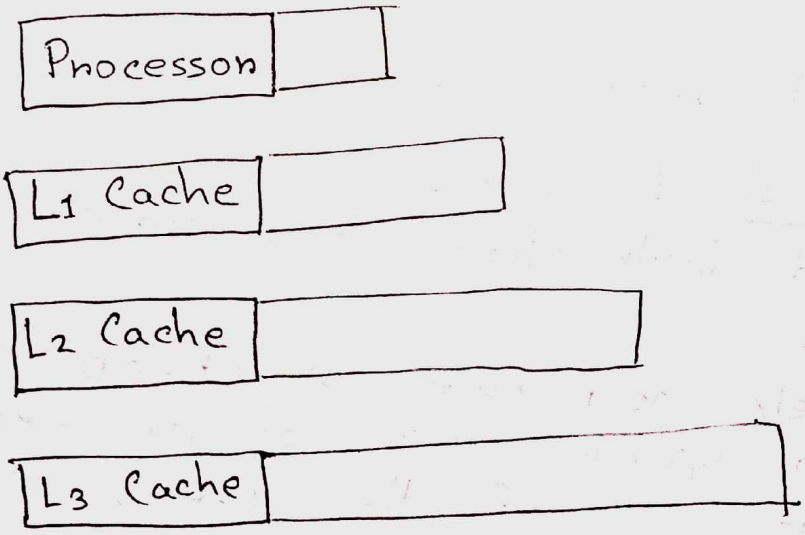
Computer Architecture

Memory Hierarchy



1000 bytes	64 KB	256 KB	2-4 MB	4-16 GB	4-16 TB
300 ps	1 ns	3-10 ns	10-20 ns	50-100 ns	5-10 ms



Why Cache Memory Hierarchy ?

- **Goal**: (Programmers want) unlimited amount of memory with low latency.
- Fast memory technology is more expensive per bit than slower memory.
- **Solution**: Use principle of locality and ~~use~~ organize memory system into a Hierarchy.
  - Entire addressable memory space available in largest, slowest, cheapest memory.

- Incrementally smaller and faster memories each containing a subset of the memory below it, proceed in steps up toward the processor.

▷ Temporal and spatial locality insures that nearly all memory references can be found in smaller memories.
- Gives the illusion of a large, fast memory being presented to the processor.

Principle of Locality / Locality of Reference

▷ Principle of Locality: Programs tend to reuse data and instruction they have used recently.
- A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code.

▷ Temporal Locality: Recently accessed items are likely to be accessed in near future.
- Examples: Loops, variable reuse, etc.

▷ Spetial Locality: Items whose address are closed together tend to be referenced closed together in time.
- Examples: sequential execution of instructions, arrays, etc.

- If a consecutive sequence of content of ~~memo~~ main memory is transferred to cache at a time, system can be benefited by both of localities.

* Large gap in performance between processor and Main memory.

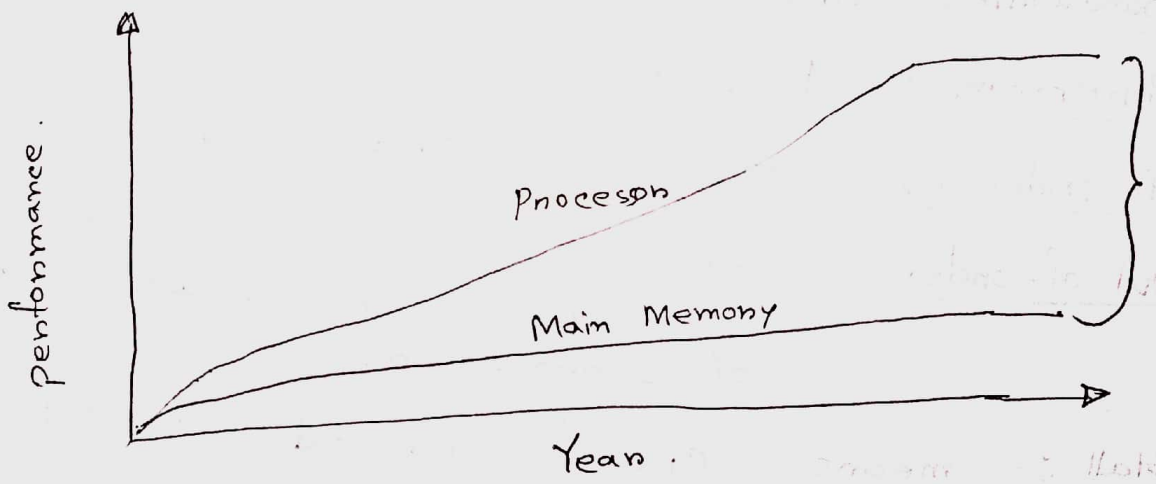


Figure: The processor line shows the increase in memory requests per second on average. (i.e, the inverse of the latency between memory references), while the memory line shows the increase in DRAM accesses per second (i.e, the inverse of the DRAM access latency).

- Becomes more crucial with multi core processors.
- Power consumption is also an issue.

Cache Memory :

- **Cache Memory** :- first level of memory receiving address from processor.
- **Cache hit** :- found in top level cache.
- **Cache miss** :- not found in top level cache.
- **Latency** :- Time to get the first word.
- **Bandwidth** :- Time to get the rest of the block.
- **Performance** :- 1 / Latency.
- **In-order execution** :- maintains program order.
- **Out-of-order execution** :- Independent instruction need not maintain order. It is a capability of a processor.
- **Stall** :- means CPU is paused (on miss)
- **Memory stall cycles** :- No of cycles processor is stalled waiting for memory access.

Miss penalty :- cost / stall cycles per miss.

Miss rate :- No. of missed accesses per number of accesses. One of the most important (but not only) measures of cache design.

Formula :

$$\text{CPU execution time} = \left(\text{CPU clock cycles} + \text{Memory stall cycles}\right) \times \text{clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of Misses} \times \text{Miss penalty}$$

$$= IC \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$= IC \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}$$

Problem :

Assume we have a computer where the cycles per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?

Answer: First compute the performance for the computer that always hits:

$$\text{CPU execution time} = \left(\text{CPU clock cycles} + \text{Memory stall cycles}\right) \times \text{clock cycles}$$

$$= (IC \times CPI + 0) \times \text{clock cycles}$$

$$= IC \times 1.0 \times \text{clock cycles}$$

Now for the computer with the real cache, first we compute memory stall cycles:

$$\text{Memory stall cycles} = IC \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss Rate} \times \text{Miss Penalty}$$

$$= IC \times (1+0.5) \times 0.02 \times 25$$

$$= IC \times 0.75.$$

where the middle term $(1+0.5)$ represents one instruction access and 0.5 data accesses per instruction. The total performance is thus

$$\text{CPU execution time}_{\text{cache}} = (IC \times 1.0 + IC \times 0.75) \times \text{clock cycle}$$

$$= IC \times 1.75 \times \text{clock cycle}.$$

The performance ratio is the inverse of the execution times:

$$\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} = \frac{1.75 \times IC \times \text{clock cycle}}{1.0 \times IC \times \text{clock cycle}}$$

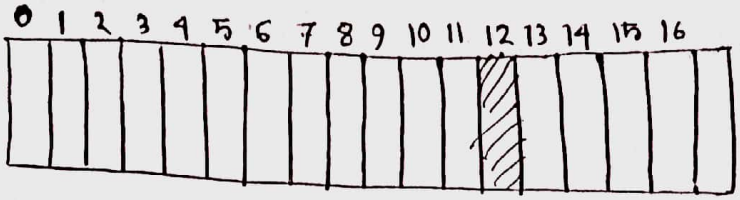
$$= 1.75$$

The computer with no cache misses is 1.75 times faster.

Where is a block placed in the upper level?

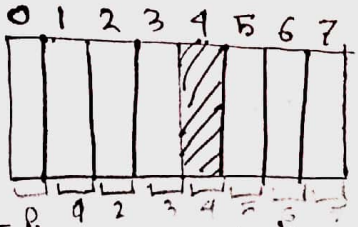
Block placement:

Block frame address:



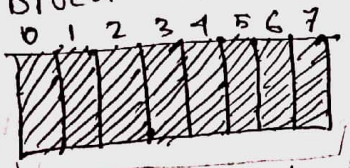
Direct mapping:

Placement → block MOD no of blocks
8 - way set associative
block 12 can go only into block 4 (12 MOD 8)



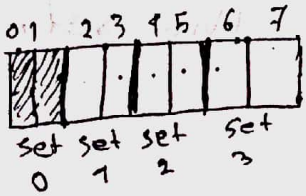
Fully associative mapping:

Placement → Anywhere
8-way set associative.
block 12 can go anywhere.



Set associative mapping:

Set placement → block MOD no of sets (12 MOD 4)
Block placement within set → Anywhere.
block 12 can go anywhere in set 0 (12 MOD 4)



n way set associative:
— n blocks in a set.

How is block found the upper level?

▶ Block Identification:

▷ Physical address coming out of the CPU (after converting virtual address) and entering the cache has three parts ... Tag, Index and Block offset.

▭ Block address:

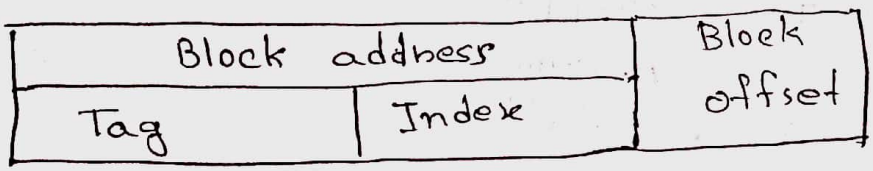
– Tag: Identifies block within a set.

□ validity bit in every entry to signal whether content is valid.

– Index: select the set.

▭ Block offset:

– Select Data within block.



The three portion of an address in a set associative or direct-mapped cache.

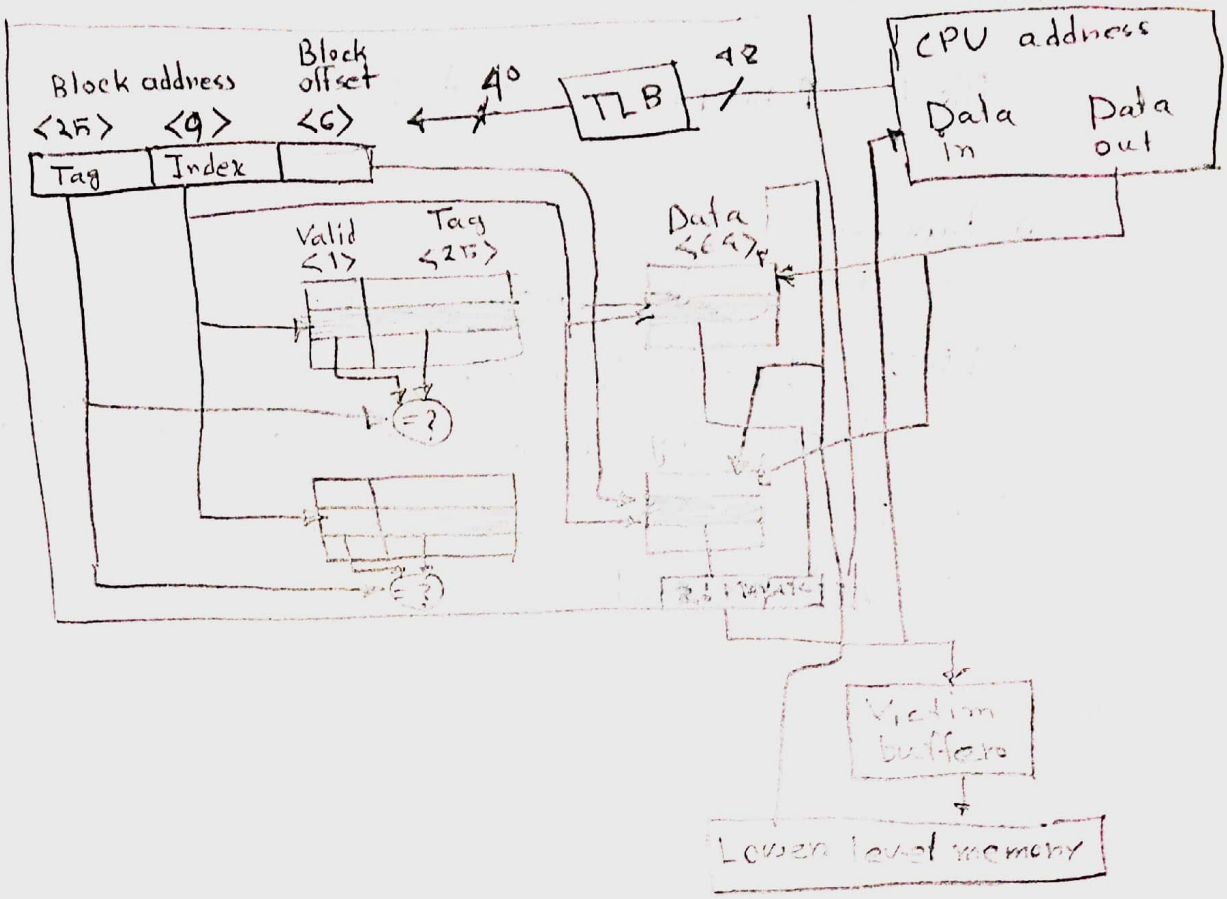
The tag is used to check all the blocks in the set, and the index is used to select the set. The block offset is the address of the desired data within the block.

Fully-associative caches have no index field. because – no need to select the set.

▷ Higher associativity means:

– Less index bits.

– More tag bits



The organization of the data cache in the Opteron microprocessor. The 64 KB cache is two-way set associative with 64 byte blocks. The 9 bit index select among 512 sets.

Which block must be replaced on a miss?

Block replacement:

- Random
  - Easy to implement.
- LRU : Least Recently used
  - Increasing complexity as associative increases.
- FIFO
  - Approximates LRU with a lower complexity.

▷ Write strategy :

▷ Write through :

- All writes send to lower level memory.
- Easy to implement
- Simplifies data coherency (all levels has same copy).
- Performace issues.

▷ Write Back :

- Write hits do not go to the lower level memory.
- Writes at a speed of cache memory.
- Uses less memory bandwidth, making write-back attractive in multiprocessors
- Power efficient, making it attractive for embedded applications.
- Propagation and serialization problems.
- More complex.

Where is write done ?

- Write through : In cache block and next level.
- Write back : Only in cache block

* What happens when a block is evicted from cache?
- Write through : Nothing else
- Write back : Next level in memory is updated.

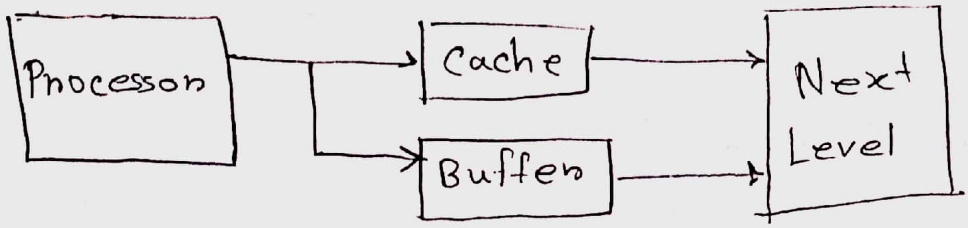
* Debugging :
- Write through :- Easy.
- Write back :- Difficult.

* Miss causes write?
- Write through : No
- Write back : Yes.

* Repeated write goes in the next level?
- Write through : Yes
- Write back : No

* Write stall
- Processor must wait for writes to complete during write-through

* Write Buffer / Victim Buffer
- Allows the processor to continue as soon as the data are written to the buffer.
- Thereby overlapping processor execution with memory updating.
- Reduces write stalls



▷ Write Miss:

- The block to be updated is absent in the cache.
- Data is not needed by the CPU, like read miss

* Two alternative policies, shown bellow:

○ Write allocate:

- Block is allocated on a write miss, followed by a write hit.
- natural option, write misses act like read misses.

○⊗ No Write allocate

- the block is modified only in the lower level memory
- do not affect the cache.

▷ Either write-miss policy can be used with write through or write back.

▷ Normally, write back caches use write allocate, hoping that subsequent writes to that block will be captured by the cache.

▷ Write through caches often use no write allocate Because subsequent writes to that block still go to the lower level memory.

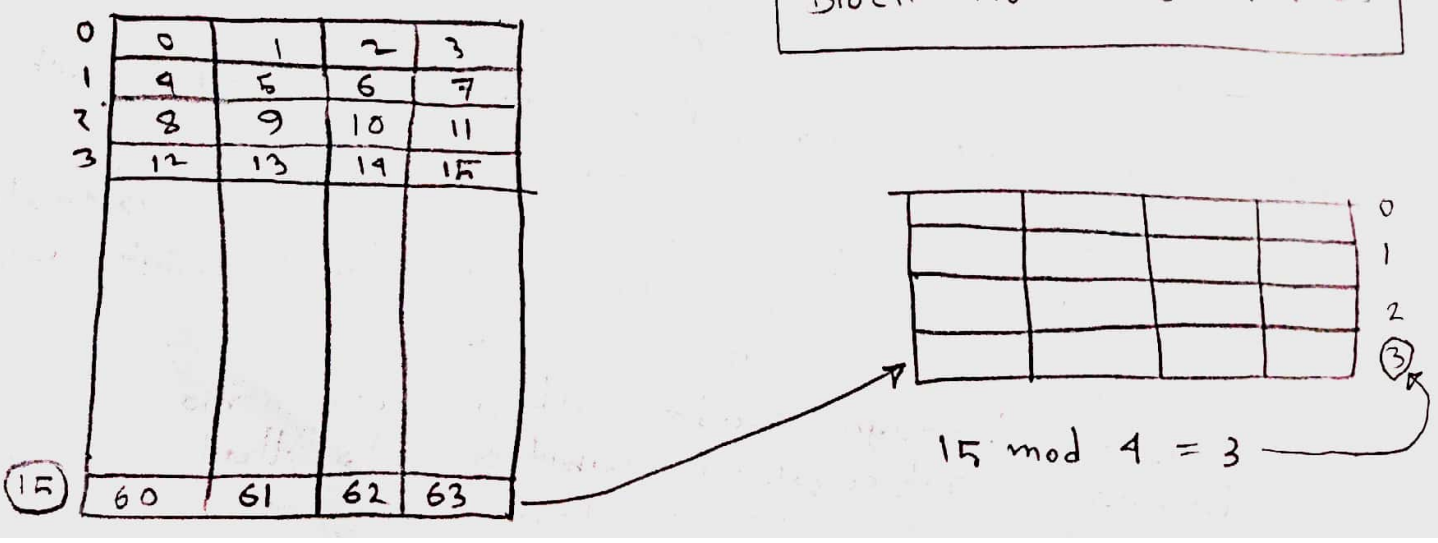
Cache Design

1. Block Placement:
   - Where to place the main memory Block in the Cache?
2. Block Identification:
   - How to find the Main Memory Block in the cache?
3. Block Replacement:
   - During a Cache Miss, how to choose which entry to replace from the cache?
4. Write Strategy:
   - How are the updations propagated?

※ Block Placement

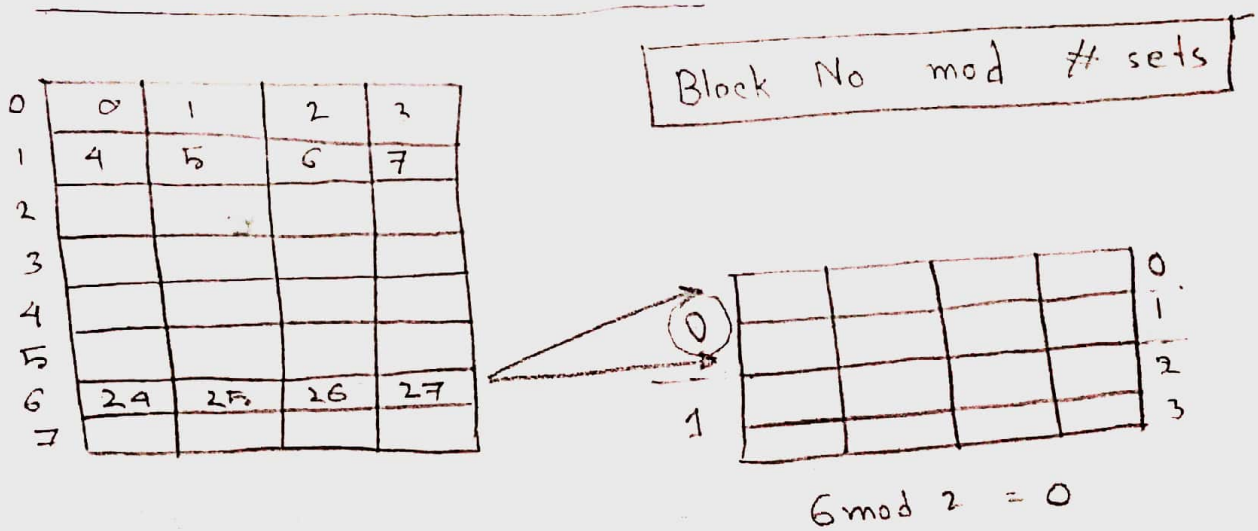
- Direct mapping:

Block No mod #lines



If we are to find out where the contents of the main memory block number 15 is going to be placed inside the cache all we need is to perform 15 mod 4 because that is the number of lines inside the cache which will give us the line number 3. Therefore the contents of the main memory blocks no 15 that is word nos 60, 61 62 and 63 will be placed inside the line no 3.

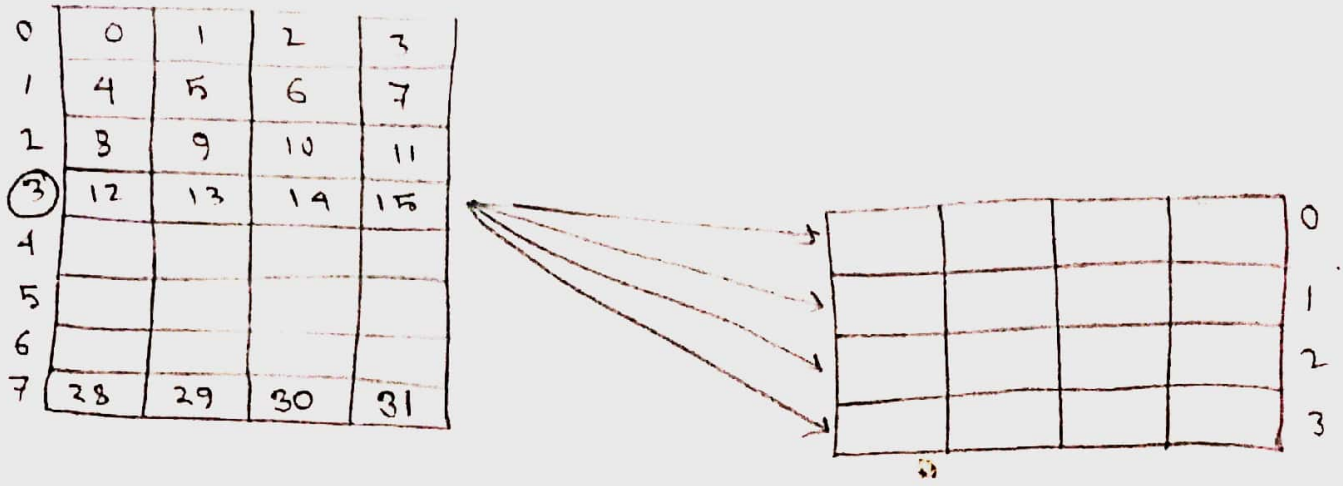
- Set Associative Mapping :



If we are to find out where the contents of the main memory block number 6 will be placed inside this 2-way set associative cache with two different sets all we need to do is perform 6 mod 2 because that is the no of sets inside the cache which will give us the set no 0. Therefore the contents of the main memory block

number 6 can be mapped onto. Now if the set no 0 is empty previously in that case for the main memory block no 6, we have two different mapping options that is the line no- 0 and the line no. 1.

- Fully Associative Mapping:



If we are to find out where the contents of the main memory block number 3 will be placed inside the cache in that case we don't really have any restrictions it can be placed onto any of the cache lines. therefore in associative mapping we can place a main memory block any where inside the cache. And that's how block placement are done.

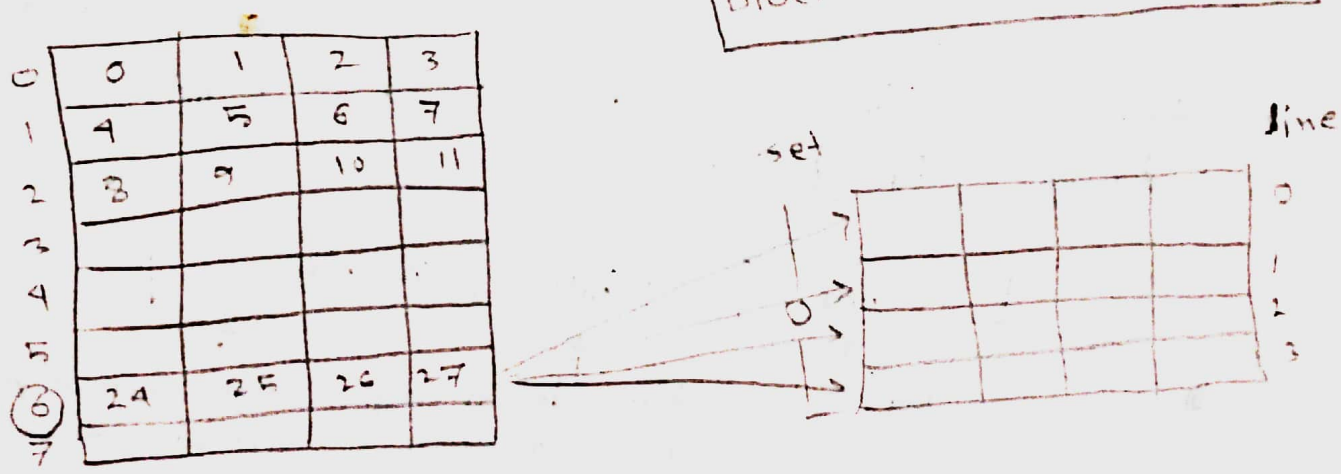
☐ (# lines in cache) 4-way set associative is direct mapping.

Block no mod # set

6 mod 4 = 0

☐ 1-way set associative is Fully associative mapping.

Block no mod # set



6 mod 1 = 0

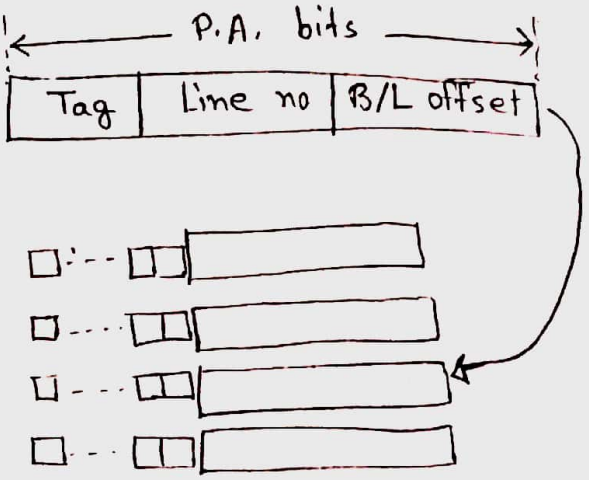
* Block Identification

Here we actually try to identify whether the main memory block is present on not inside the cache.

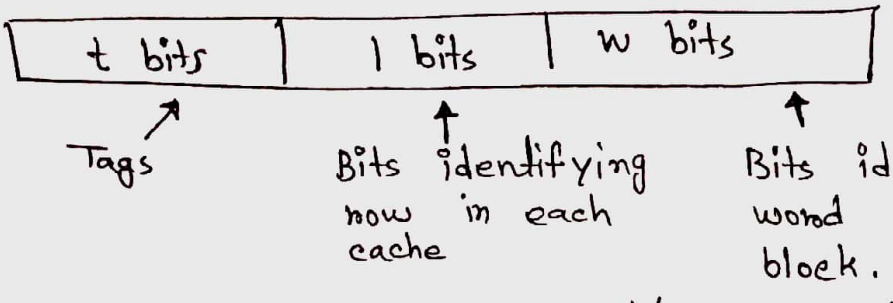
• Direct Mapping:

we know the physical address in that case is divided into tag fills, line index and block on line offset.

- ✓ Find the potential match using Line No & offset bits
- ✓ Compare the tag bits..



Example: Consider a cache with 512 lines, then a line would need 9 bits to be uniquely identified.



Direct mapping divides an address into three parts: t tag bits, l line bits, and w word bits. The word bits are least significant bits that identify the specific word within a block of memory.

The line bits are the next least significant bits that identify the line of the cache in which the block is stored.

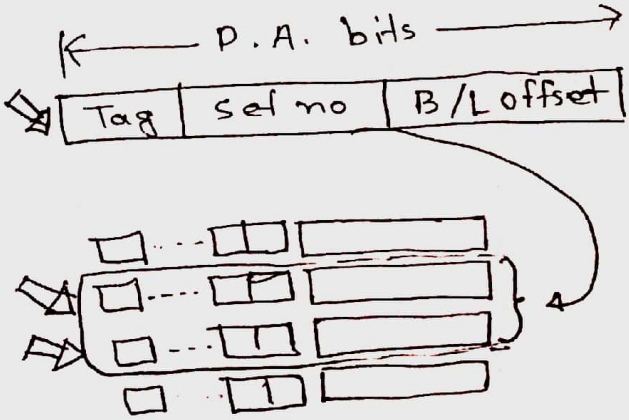
The remaining bits are stored along with the block as the tag which locates the block's position in the main memory.

Direct mapping is higher associative. So, index bits are less.

- Set Associative Mapping:

We know the physical address is split as tag fills, set index and block or line offset.

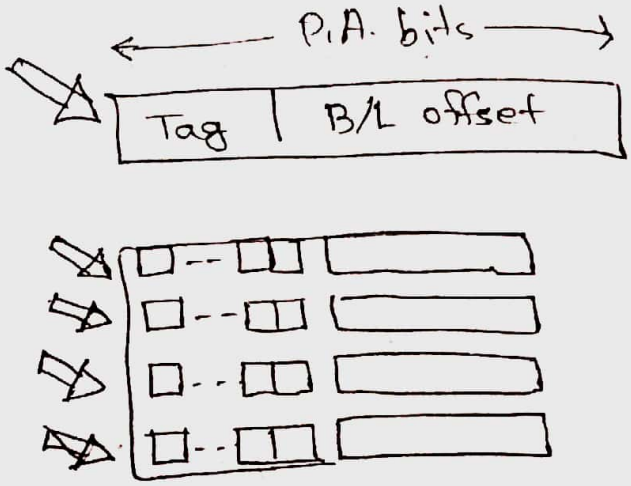
- ✓ find the set using set index.
- ✓ Compare the tag with all the tags associated with the lines belonging to that particular set parallelly



◦ Fully Associative Mapping:

We know the physical address is split as tag bits and block or line ~~offset~~.

✓ Find the potential match comparing all the tag bits associated to every line, simultaneously.



3. Block Replacement:

- Cach is Limited in size.
- What should be done.

(a) When the cache is full?

aka. "Capacity Miss"

— that means we are asking for a new block however the cache is full at that point, that kind of miss is known as "capacity miss".

(b) When the potential match can't be found?

it can happen in two scenarios.

—(i) "Compulsory Miss"

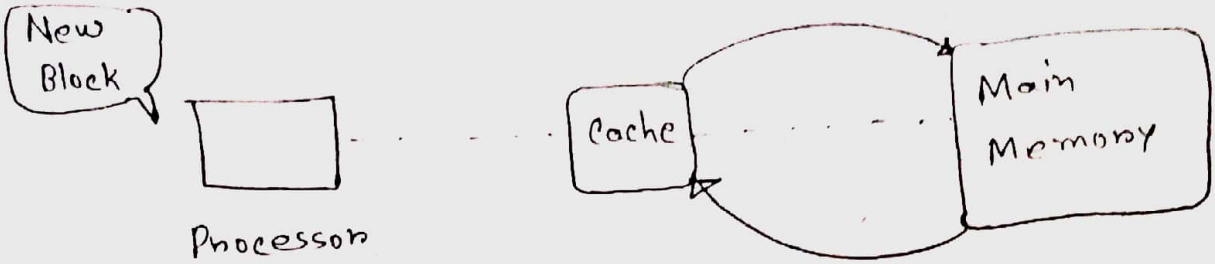
— that means the main memory block that we are looking for inside the cache has not been accessed yet, therefore we can't do anything about it, so it's a "compulsory miss".

(ii) "Conflict Miss"

— that means the main memory block inside the cache where it should be has already been occupied by some other main memory block.

Now for "capacity miss" and "conflict miss" we can do block replacement.

— Replace a 'Block' residing in cache with the new block request and move the replaced 'Block into the next level of memory Hierarchy.



- Which "Block" to replace?
  - that means which cache block will be selected for eviction so that it can make space for the newly requested block.

✓ Cache Replacement Policies:

- Random
  - where we replace any of the cache block at random.
- LRU: Least Recently Used
  - where we keep track of the access orders of the blocks. we replace the block which is not recently used.
- FIFO: First In First Out
  - that means the cache blocks are evicted in their arrival order, whichever block came into the cache first will be selected for eviction

4. Write strategy :

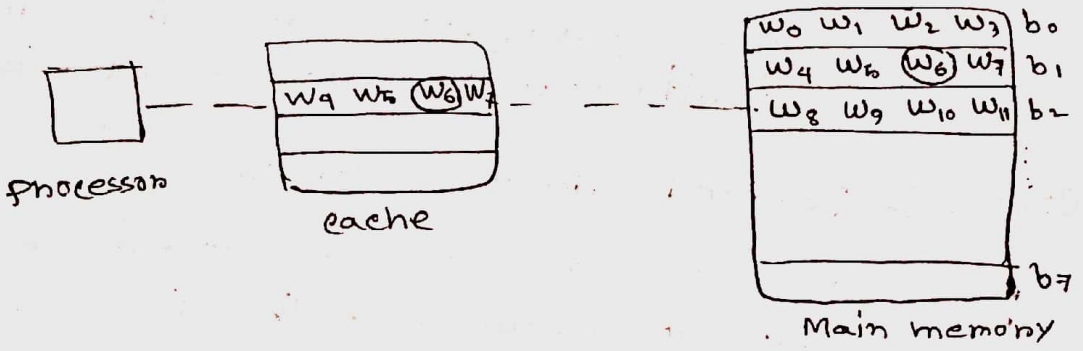
- When it is needed ?
  - Whenever the processor needs to modify data word.

- Situations :

(1) Write Hit : The data word which is to be modified is already present inside the cache.

We have two different kind of strategies

(1.1) Write through :
- Both the cache & Main Memory are updated simultaneously.
- Used during lesser Write operations.



- Now, since it's a ~~right~~ write hit situation ⇒ suppose the content of the main memory block number 1 are already present inside the cache.. Suppose the processor needs to modify this ~~part~~ particular data word (W_6). Now according to the write through strategy when the modification will be made it will be propagated to the main memory as well updating the main memory block.

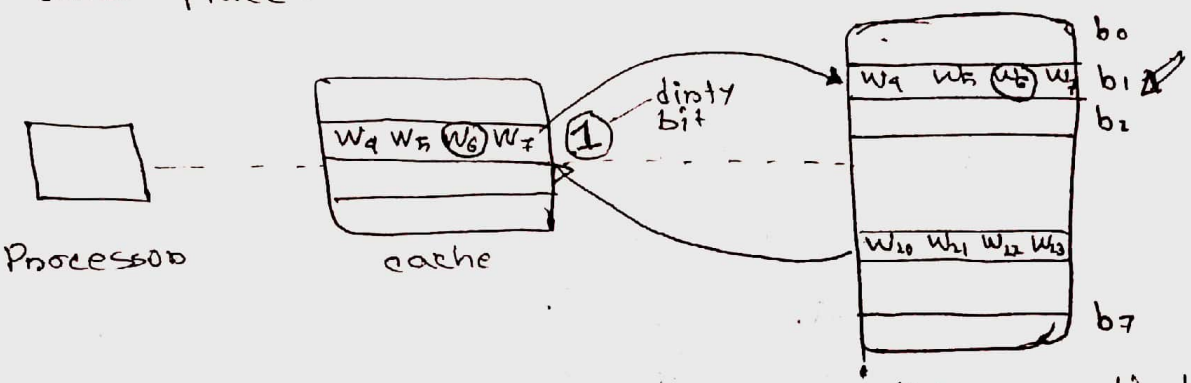
Now this strategy is used during lesser write operations, because cache is embedded into the processors themselves. Therefore accessing the cache requires way less time than accessing the main memory. now since in case of "write through" both the cache and the main memory are updated simultaneously, therefore wherever there is a need for write as the modification should be propagated to the next level as well. so for each write operation the time required will be longer.

pros : ▷ Reliable and helps in data recovery
— if the cache fails at a certain point of time since the updations are propagated to the main memory simultaneously we won't lose any data.

cons : ▷ Delayed data writes.
— because for every updation we actually need to access both the cache and the main memory.

(1.2) Write Back / Write Deferred :

- Only the cache is updated in real-time & uses "Dirty-bit".
- Main memory is updated when replacement takes place.



If the processor wants to update this particular data word inside the cache, it will update it and the updation will be reflected using a bit called "dirty bit". And when the updation is met the dirty bit will set to 1. now the information regarding the dirty bit will kept inside the tag directory for each tag directory entry along with its tag information

The updation will be propagated to the main memory whenever the replacement takes place. whenever the processor asks for a new block which will be placed inside the cache in the same line, the contents of the line will be evicted from the cache. If for that particular line the dirty bit is set to 1 the updation will be propagated to the main memory and afterwards the newly requested block will be placed inside the cache.

Pros : ▷ It is way fasten.

— because the updation is propagated to the main memory only when replacement takes place. So apart from replacement we are not going to access the main memory very often.

Cons : ▷ It makes the data recovery impossible.

— that means at any certain point of time if the cache fails, we won't be able to recover the updated data.

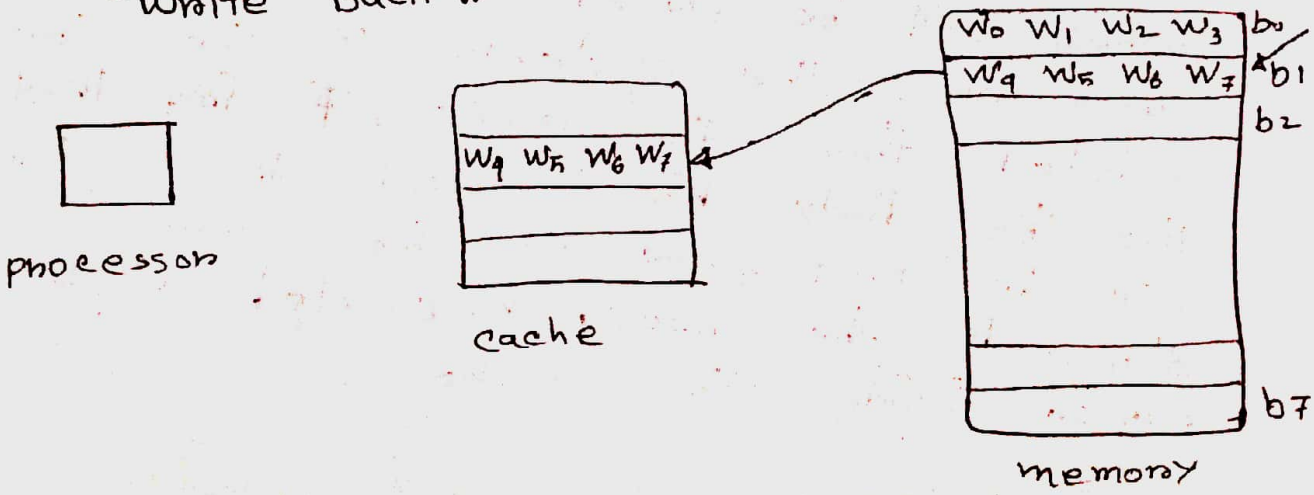
(2) Write Miss : The data the processor needs to modify is actually absent from the cache.

We have two different strategies..

(2.1) Write allocate :

— since the data is absent from the cache, the data is brought into the cache first.

— can work equally with "Write-through" & "Write Back"



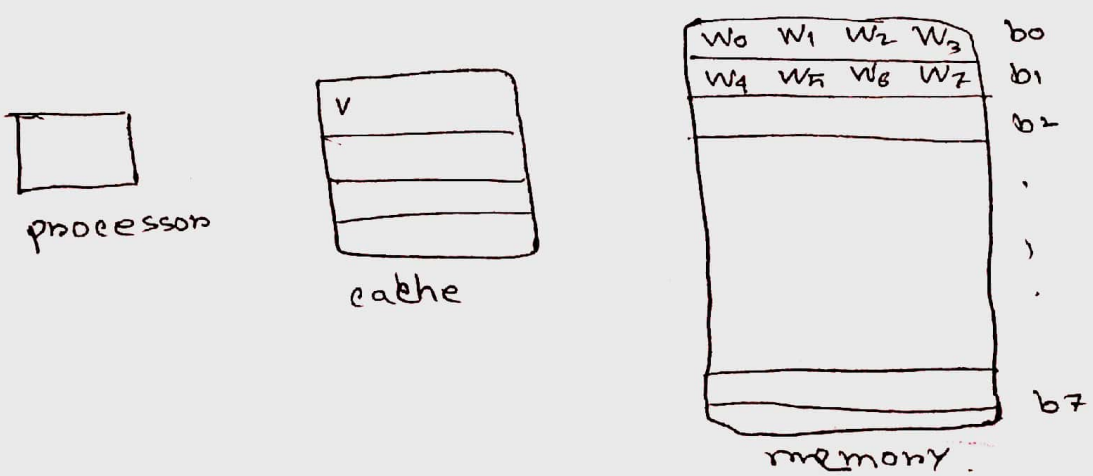
Now, suppose the processor needs to modify this particular data word of the main memory block no 1, now as the name suggests we have to allocate the main memory block inside the cache first and afterwards the updation will be made.

Write allocate can work equally with 'write through' that means whenever the updation is made it will propagated to the main memory at the same time as well as 'write back' that means the updation will propagated when only the cache block is replaced.

However since 'write through' suffers from delayed data writes, that's why 'write allocate' is mostly used with "write back".

(2.2) No - Write Allocate :

— the data is updated directly in the Main Memory, which means we don't really bother the cache regarding the allocation.



Suppose the processon needs to modify this particular data word of block no - 1. In that case the processon will update that particular data word of that particular block inside the main memory itself.