Cache Memory – Part I

Dr. B. R. Bhowmik
Dept. of CSE
NIT Karnataka

Cache Basics

- Cache memory is intended to give memory speed approaching that of the fastest memories available, and at the same time provide a large memory size at the price of less expensive types of semiconductor memories. Thus, cache
  - Is a small amount of fast memory
  - Sits between normal main memory and CPU
  - May be located on CPU chip or module
- The cache contains a copy of portions of main memory.
- When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache.
- If so, the word is delivered to the processor.
- If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor.
- Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.

Overview of a Basic Cache

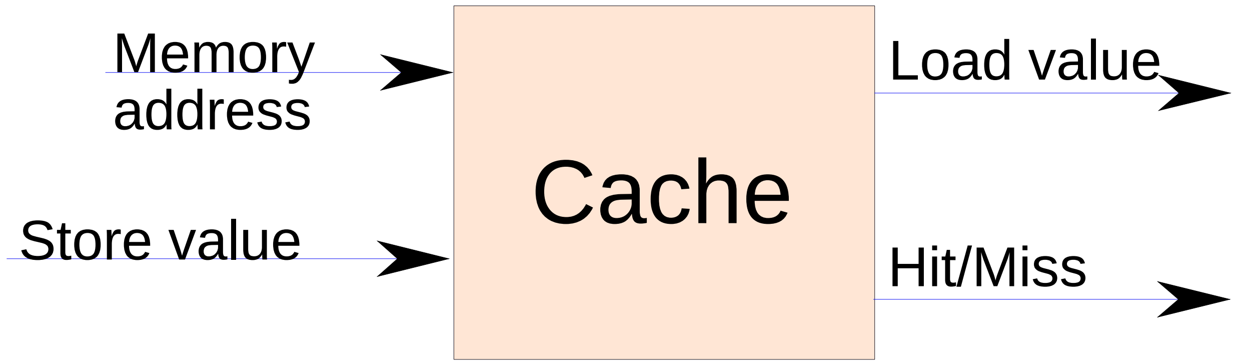


Figure 1: Cache overview

- Saves a subset of memory values
  - We can either have hit or miss
  - The load/store is successful if we have a hit

Cache Hierarchy

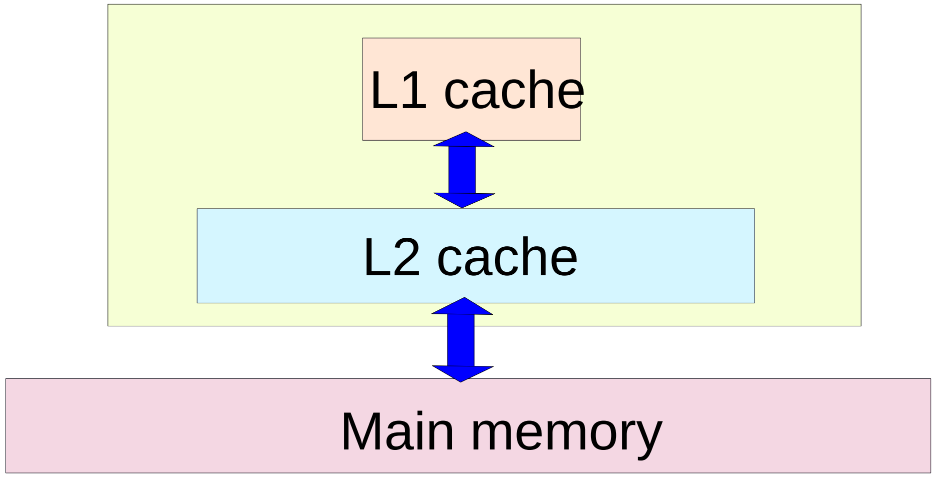


Figure 2: Cache hierarchy

- Cache uses a hierarchical memory system
- L1 (SRAM cells), L2 (SRAM cells), Main Memory (DRAM cells)

Cache Hierarchy Contd…

- Cache Hierarchy
  - The main memory contains all the memory locations
  - The caches contain a subset of memory locations
- Single and multi-level cache organization with main memory are shown in Figure 3.
- Fig. 3(a) depicts the use of single level of cache.
- Fig. 3(b) depicts the use of multiple levels of cache.
- The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.
- The L1 cache has size 8-64 KB and is composed of SRAM cells.
- The L2 cache has size 128 KB – 4 MB and is also made of SRAM cells.
- The main memory is much larger with size 1 – 64 GB and made of DRAM cells.

Cache Organization Diagram

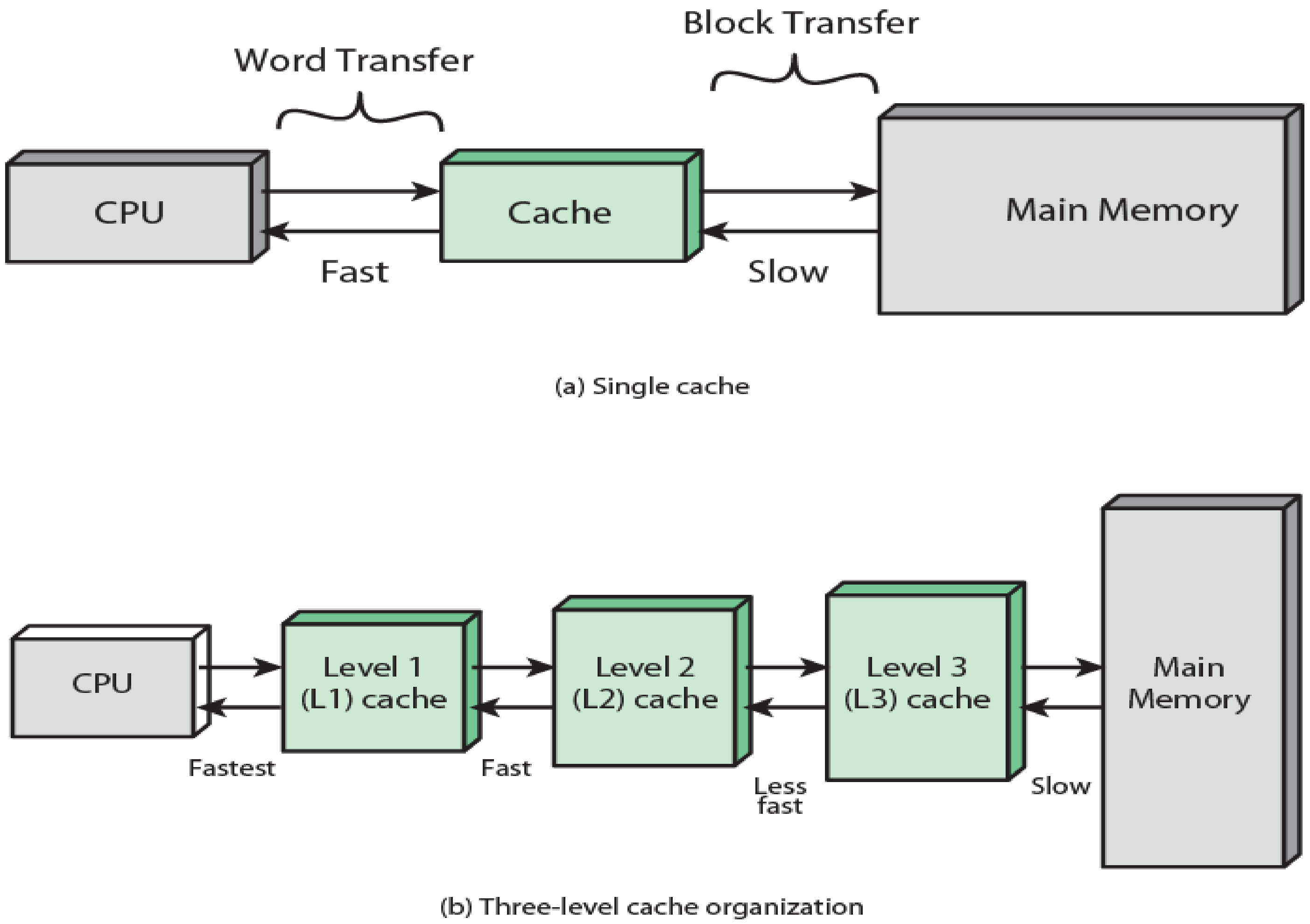
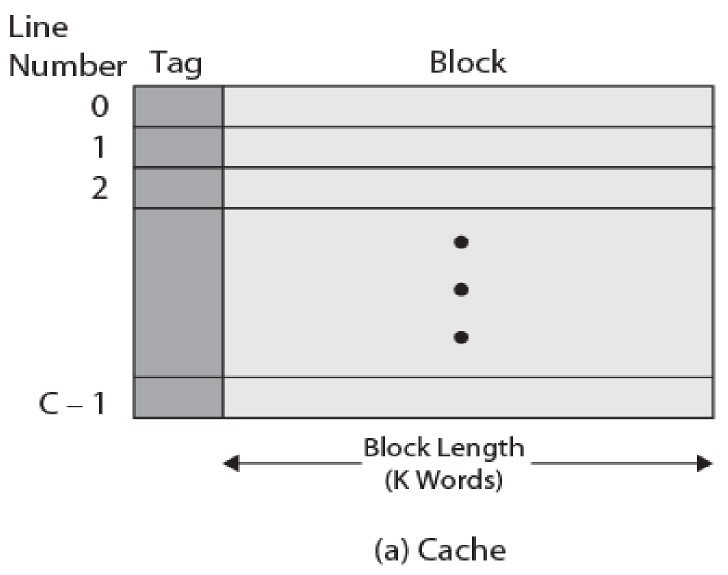


Figure 3: Cache and main memory relation.

Cache and Main Memory

- Figure 4 depicts the structure of a cache/main-memory system.
- Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address.
- For mapping purposes, this memory is considered to consist of a number of fixed length blocks of K words each. That is, there are $M = 2^n/K$ blocks in main memory.
- The cache consists of m blocks, called **lines**.
- Each line contains K words, plus a tag of a few bits. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since being loaded into the cache.
- The length of a line, not including tag and control bits, is the **line size**. The line size may be as small as 32 bits, with each "word" being a single byte; in this case the line size is 4 bytes.

Cache/Main Memory Structure



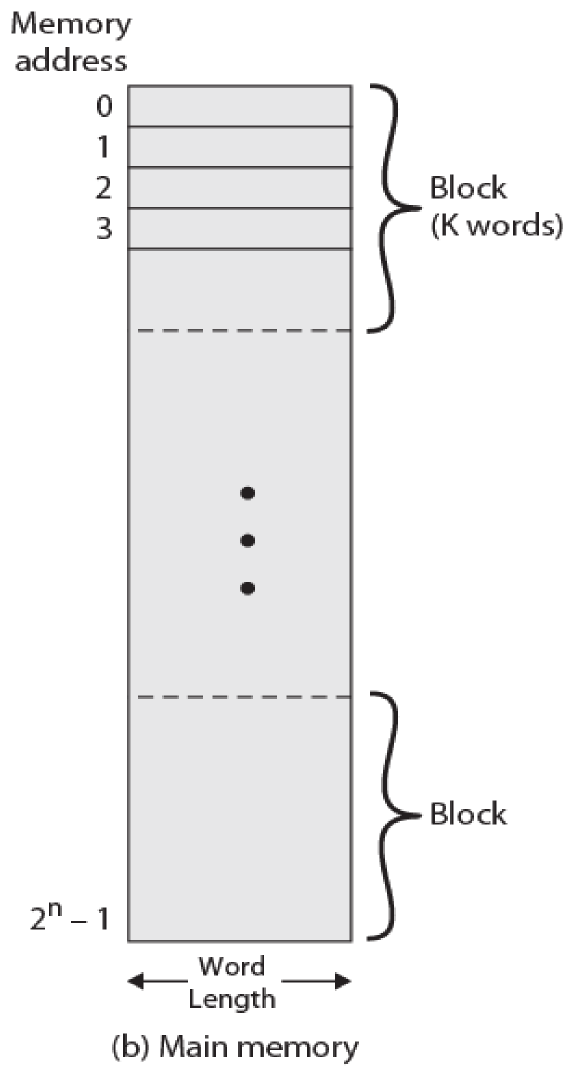


Figure 4. cache and main memory structure

Cache operation – overview

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

Basic Cache Operations

- lookup → Check if the memory location is present
- data read → read data from the cache
- data write → write data to the cache
- insert → insert a block into a cache
- replace → find a candidate for replacement
- evict → throw a block out of the cache

Cache Lookup

A. Let us a cache have two SRAM arrays (Figure 5)
   - tag array → Saves a part of the block address such that the block can be uniquely identified
   - block array → Saves the contents of the block
   - Both the arrays have the same number of entries
   - Efficient cache lookup means to find out a block efficiently in
     - Fully Associative Cache
     - Direct Mapped Cache
     - Set Associative Cache

B. Running example : 8 KB Cache, block size of 64 bytes, 32 bit memory system.

Structure of a Cache

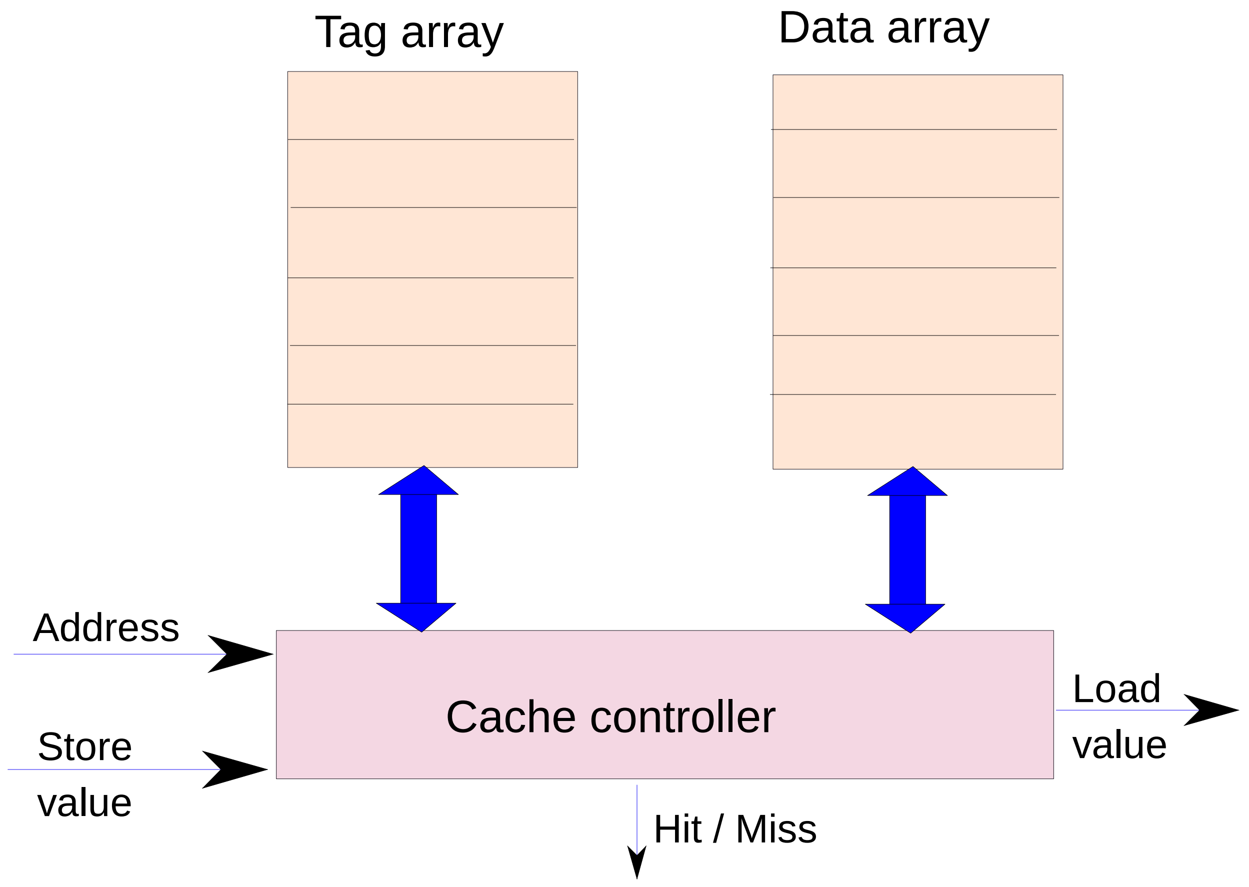
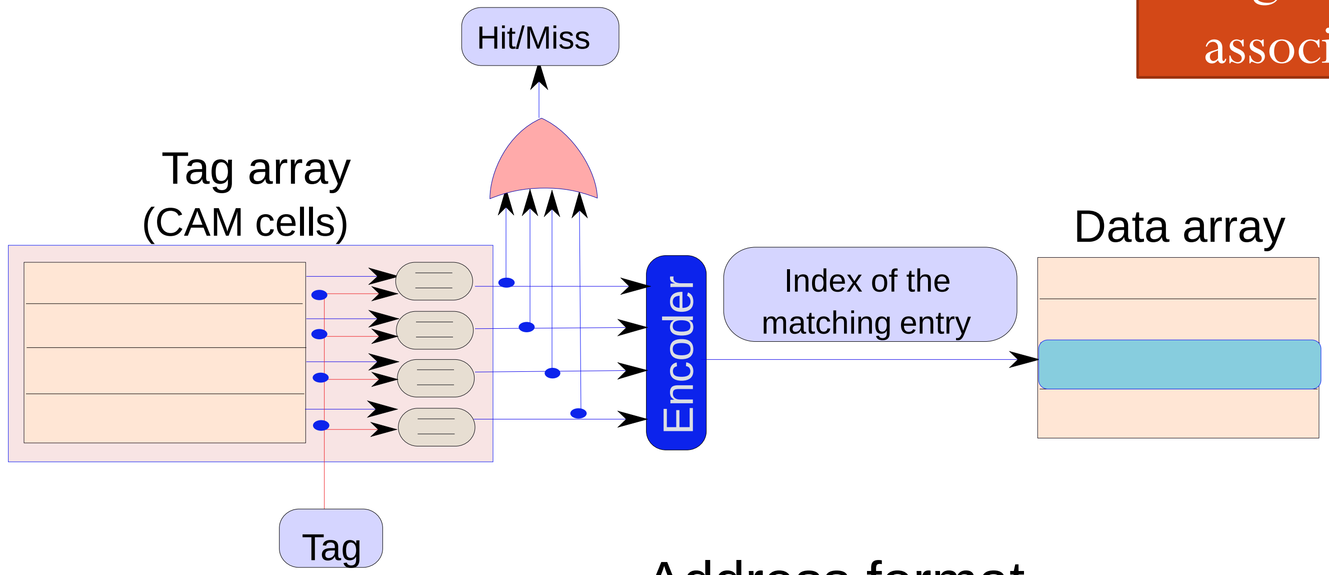


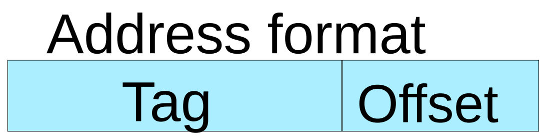
Figure 5. Structure of Cache with Tag and data Array.

Fully Associative Cache

- We have $2^{13} / 2^{6} = 128$ entries
- A block can be saved in any entry
- 26 bit tag, and 6 bit offset

Figure 6. A fully associative cache





Implementation of the FA Cache

- We use an array of CAM cells for the tag array
- Each entry compares its contents with the tag
- Sets the match line to 1
- The OR gate computes a hit or miss
- The encoder computes the index of the matching entry.
- We then read the contents of the matching entry from the block array

Direct Mapped Cache

- Each block can be mapped to only 1 entry

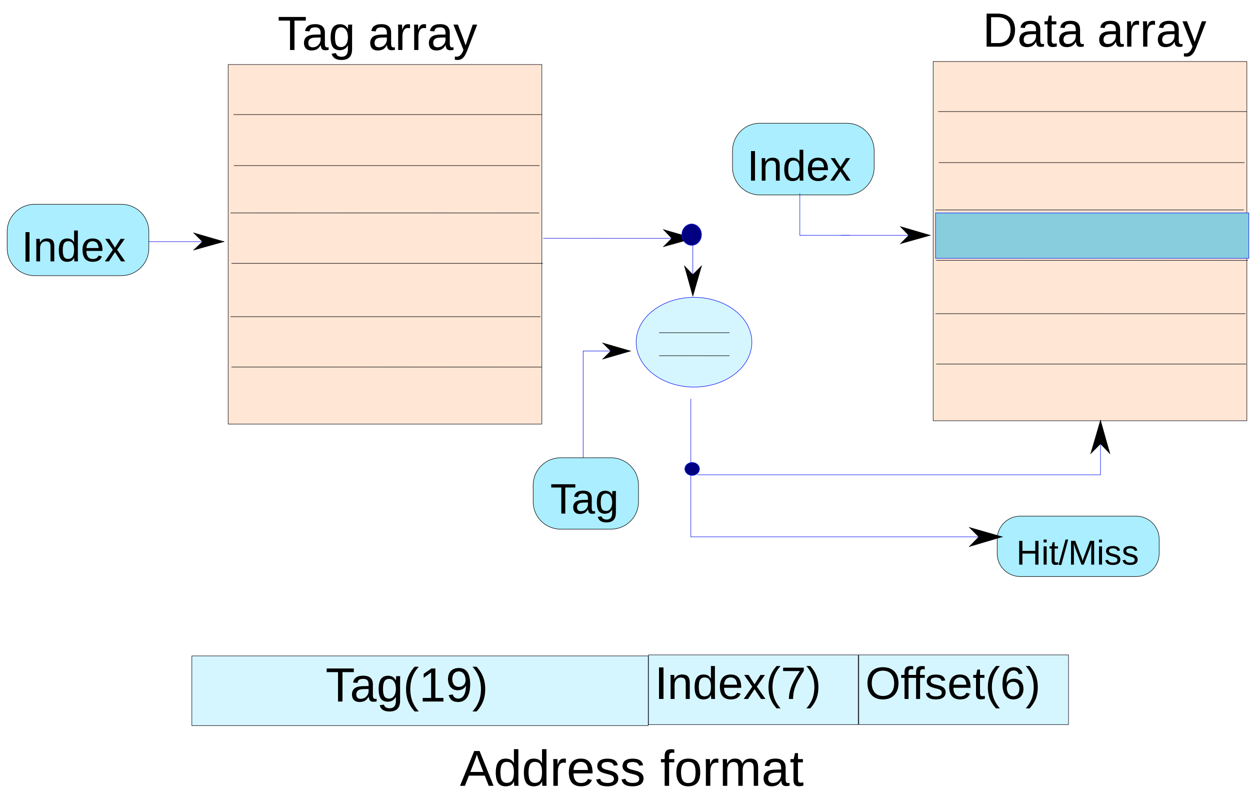
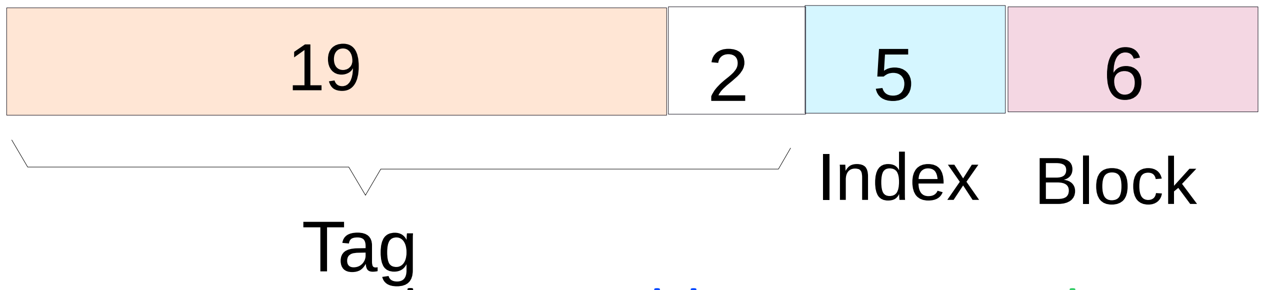


Figure 7. Direct mapped cache

Direct Mapped Cache Contd…

- We have 128 entries in our cache.
- We compute the index as idx = *block address* % 128
- We access entry, idx, in the tag array and compare the contents of the tag (19 msb bits of the address)
- If there is a match → **hit** else → **miss**
- Need a solution that is in the middle of the spectrum

Set Associative Cache



- Let us assume that an address can reside in 4 locations
    - Access all 4 locations, and see if there is a hit
- Thus, we have 128/4 = 32 indices
- Each index points to a *set* of 4 entries
- → We now use a 21 bit tag, 5 bit index

Set Associative Cache Contd...

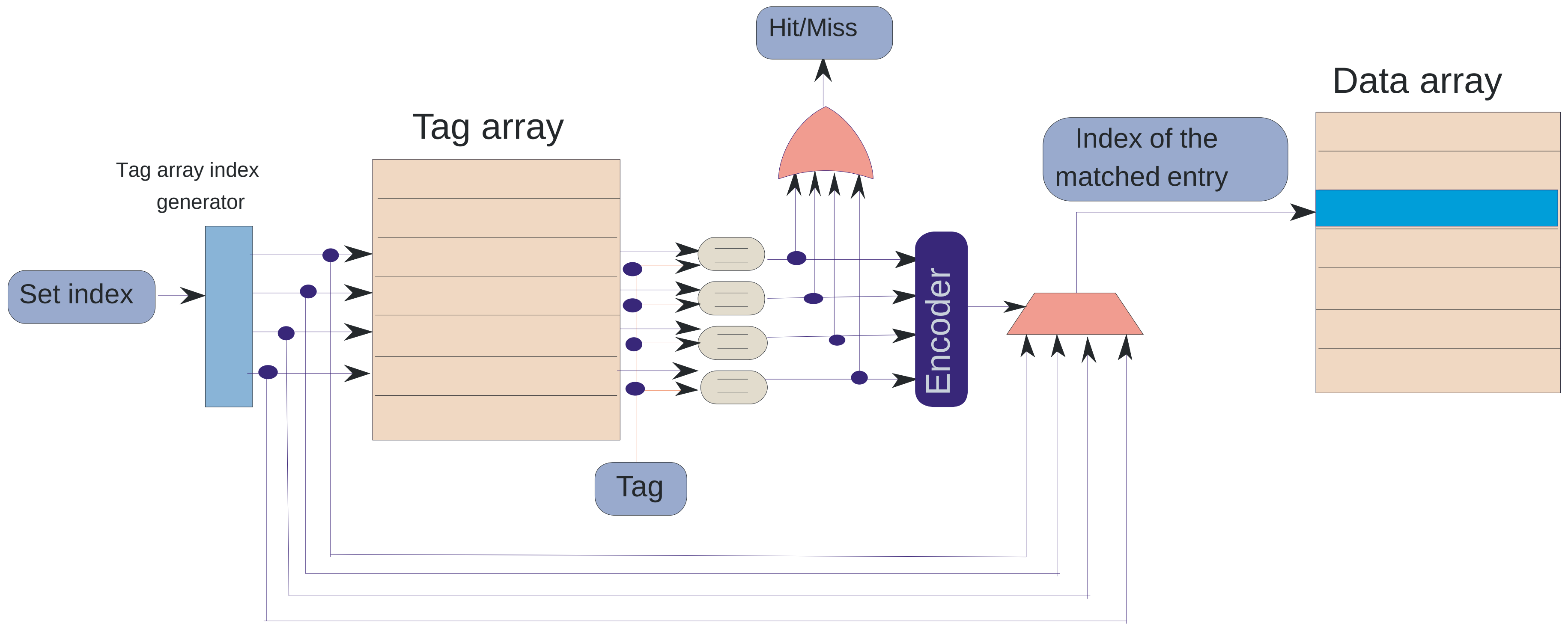


Figure 8. Set associative cache

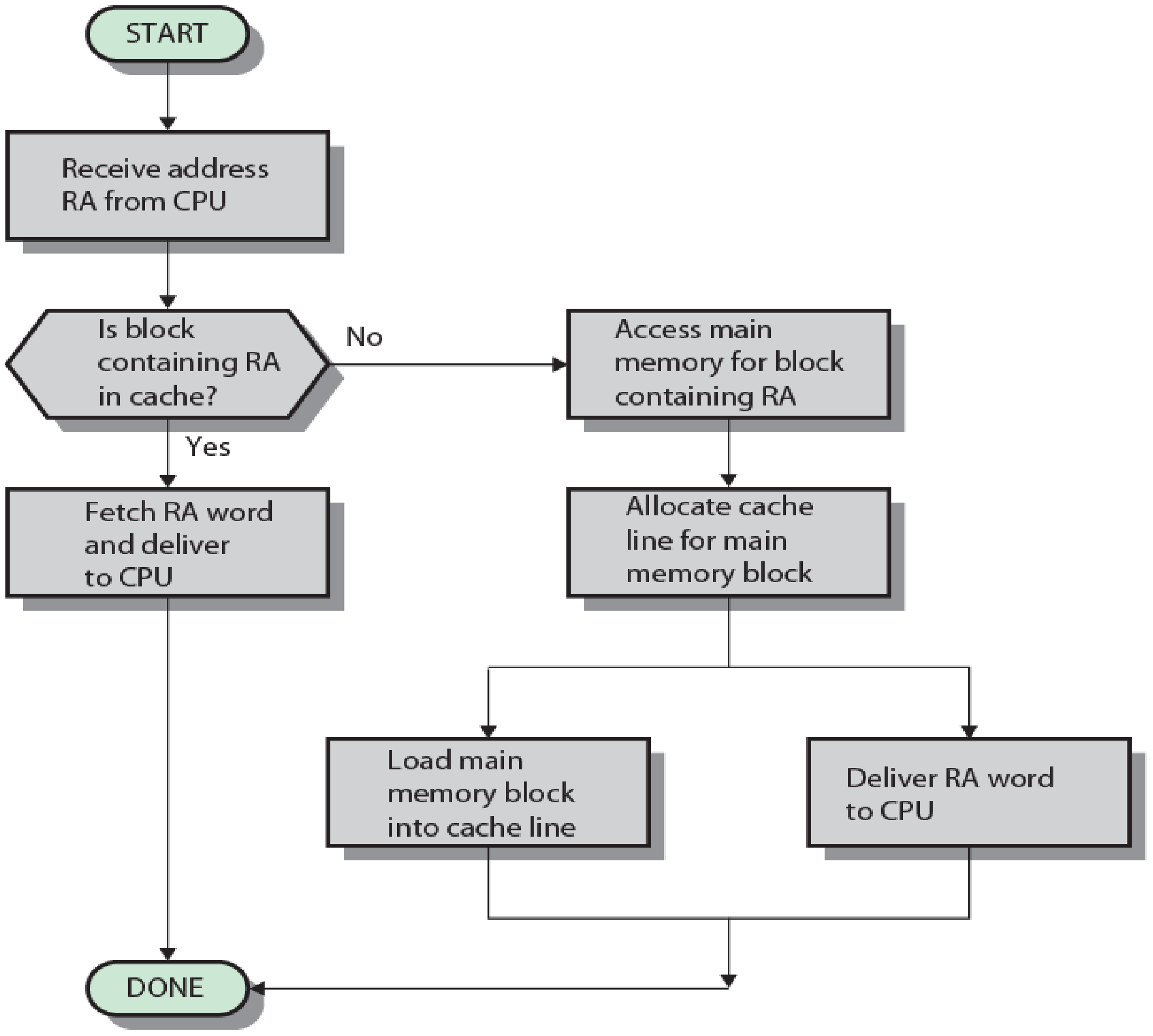
Set Associative Cache Contd…

- Let the index be *i* , and the number of elements in a set be *k*
  - We access indices, i*k, i*k+1 ,.., i*k + (k-1)
  - Read all the tags in the set
  - Compare the tags with the tag obtained from the address
  - Use an OR gate to compute a hit/ miss
  - Use an encoder to find the index of the matched entry

Data read operation

- This is a regular SRAM access.
- Note that the data read and lookup can be overlapped for a load access
  - We can issue a parallel data read to all the ways in the cache
  - Once, we compute the index of the matching tag, we can choose the correct result with a multiplexer.

Cache Read Operation - Flowchart

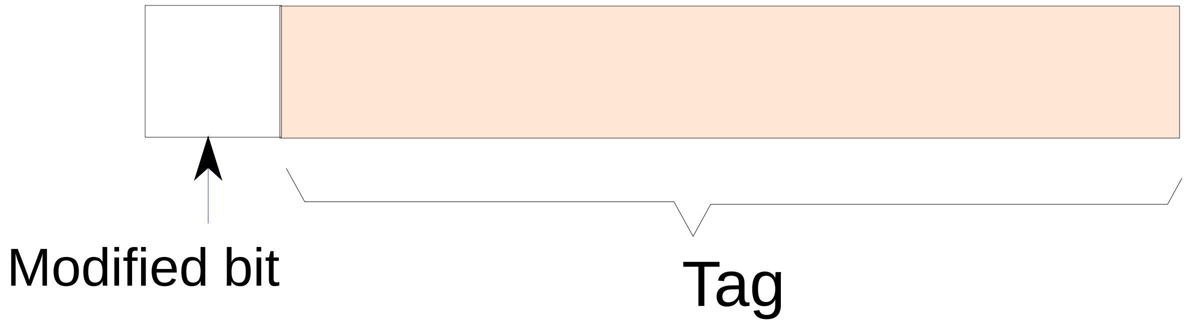

START
Receive address RA from CPU
Is block containing RA in cache?
No
Yes
Access main memory for block containing RA
Fetch RA word and deliver to CPU
Allocate cache line for main memory block
Load main memory block into cache line
Deliver RA word to CPU
DONE

Data write operation

- Before we write a value, we need to ensure that the block is present in the cache.
- Why ?
    - Otherwise, we have to maintain the indices of the bytes that were written to
    - We treat a block as an atomic unit
    - Hence, on a miss, we fetch the entire block first
- Once a block is there in the cache, Go ahead and write to it .
- Maintain a modified bit in the tag array.

Data write operation Contd…

- If a block has been written to, after it was fetched, set it to 1.



- Data write operation follows write policies.
  - Write through
  - Write back

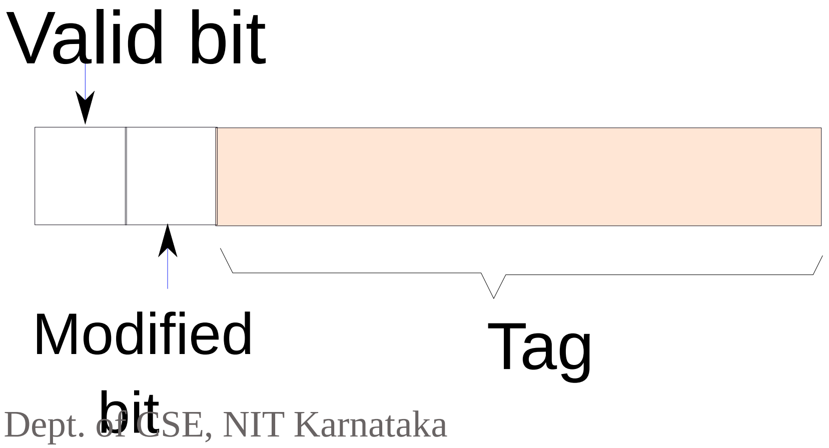
Write Policies

- Write through → Whenever we write to a cache, we also write to its lower level
  - ➢ Advantage : Can seamlessly evict data from the cache
- Write back → We do not write to the lower level. Whenever we write, we set the modified bit.
  - ➢ At the time of eviction of the line, we check the value of the modified bit

Insert Operation

If we don't find a block in a cache. We fetch it from the lower level. Then we insert the block in the cache → insert operation

- Let us add a valid bit to a tag
  - If the line is non-empty, valid bit is 1
  - Else it is 0
- Structure of a tag



Insert Operation Contd…

- Check if any way in a set has an invalid line
  - If there is one, then write the fetched line to that location, set the valid bit to 1.
- Otherwise, find a candidate for replacement.

The replace operation

- A cache replacement scheme or replacement policy is a method to replace an entry in the set by a new entry.
- Replacement Schemes
  - FIFO Replacement Scheme
  - Least Recently Used
  - Psuedo-LRU
  - Random Replacement Scheme

Replacement Schemes: FIFO

- For replacement, choose the way with the highest counter (oldest).
- Problems :
  - Can violate the principle of temporal locality
  - A line fetched early might be accessed very frequently.

LRU (least recently used)

A. Replace the block that has been accessed the least in the recent past and Most likely we will not access it in the near future.
   - Directly follows from the definition of stack distance
   - Sadly, we need to do more **work** per access
   - Proved to be **optimal** in some restrictive scenarios

B. True LRU requires saving a hefty timestamp with every way
   - Let us implement pseudo-LRU

Psuedo-LRU

A. Let us try to mark the most recently used (MRU) elements.

B. Let us associate a 3 bit counter with every way.

- Whenever we access a line, we increment the counter.
- We stop incrementing beyond 7.
- We periodically decrement all the counters in a set by 1.
- Set the counter to 7 for a newly fetched block
- For replacement, choose the block with the smallest counter.

evict Operation

- If the cache is write-through
  - Nothing needs to be done
- If the cache is write-back
  - AND the modified bit is 1
  - Write the line to the lower level

Thank You