CACHE MEMORY

WHAT IS CACHE MEMORY?

Cache memory is a smaller and faster type of memory located quite close to the CPU and holds the most recently accessed data or code. A cache is made up of cache lines which in turn are made up of words. Mathematically speaking, Cache size = No of cache lines * size of one cache line

The word length of the machine is 16 bits which imply that any address with an integer value from 0 to 2**16 - 1, i.e., 0 to 65535 can be entered. A multilevel cache with 2 levels- Level1 and Level 2 is implemented. L1 is a subset of L2, i.e, L2 stores all the contents of L1 (however this is not true vice versa).

Cache operations are based on locality of reference-

- Temporal Locality- Data fetched now may be fetched again in future. LRU replacement scheme is based on Temporal Locality.
- Spatial Locality- The address of data near to the address of data being currently used may be fetched soon in future. That is why we keep close addresses together in a block.

READING CACHE MEMORY-

For a valid address, there can be 3 cases while reading a cache-

1. First, Level 1 will be checked. If the **entered address exists in Level 1 of the cache** (HIT), the data on that address will be read off. A 'hit' is printed
2. Then, Level 2 will be checked. If the **entered address exists in Level 2 of the cache** (HIT), the data on that address will be read off and the corresponding block will be added to L1. A 'hit' is printed.
3. If the **address doesn't exist in either of the two levels** (MISS) (implying that the block corresponding to this address got evicted due to addition of some other block or maybe never existed in the cache), the block belonging to that word (whose address is given) is again added back to the cache. 'Address not found' is printed.
4. If **there is no data in the cache at that address**, we print a message "EMPTY!" implying that we are trying to read data at an address that has never been written onto, i.e. it is empty.

WRITING TO CACHE MEMORY

We adopt the Write-through cache policy wherein data is simultaneously written to Level 1 and Level 2 of the cache memory unlike Write-back policy wherein data is written to the Lower Level (L2) at a later stage.

For writing to the cache, the user has to **enter an address** as described above **along with data** that is to be written. There can be 3 cases while writing to a cache-

1. If the **entered address exists in Level 1 cache** (HIT), the data will be written to both L1 and L2 Cache (Write through policy).
2. If the **entered address doesn't exist in the L1 cache but does exist in L2,** the data will be written in the L2 cache and that updated block will be added to L1. A 'hit' is printed.
3. If the **entered address doesn't exist in either of the two caches,** we will update the two caches with the new block (with the data) corresponding to the address entered. In case any

or both of them are full, we will use the LRU replacement policy to replace the least recently used blocks in the caches with the new block.

If an address already has data written and we again write on that address, the data gets overwritten by the new data entered.

EVICTION AND REPLACEMENT-

We replace and evict by using the **Least Recently Used (LRU) Policy.**

LRU Policy: In LRU Cache Implementation, we remove the least recent block and add the most-recently accessed block in it's place (at the time of reading/writing to the cache).

TYPES OF CACHES-

1. **Direct Mapped Cache**

   In a direct-mapped cache, a block of the main memory can come into only 1 specific cache line (irrespective of other cache lines being empty), and that cache line is given by **B mod cl** where B is the block number and cl is the number of cache lines. For example, in a cache with cl = 4, block no 0 will come into cache line number 0 only and not any other cache line(because 0 mod 4 = 0). Due to its specific nature, this cache has **higher rates of cache misses.**

   A Direct Mapped physical address can be split up into Tag, Line-Offset and Word-Offset.

Tag- (16-(w+cl) bits)	Line-Offset(cl bits)	Word-Offset- (w bits)

   where w = number of bits needed for representing block size, cl = number of bits needed for representing number of lines

2. **Fully Associative Cache**

   In a fully associative cache, a block of the main memory **can come into any** of the cache lines whichever is empty. For example, Block no = 0 and cl = 4; Block 0 can come in any of the cache lines- 0, 1, 2 or 3. It significantly **reduces the number of misses** due to its non-specific nature.

   A fully associative physical address can be split up into Tag and Word-Offset.

Tag- (16-w bits)	Word-Offset- (w bits)

   where w = number of bits needed for representing block size.

3. **N-way Set Associative Cache**

   In an N-way Set Associative Cache, a block of the main memory can come into only 1 specific set irrespective of other sets being empty, and that set number is given by **B mod s** where B is the block no and s is the number of sets. For example, in the cache given above with N = 2 (2-way set associative), Block no 0 will come in Set no 0 (because 0 mod 2 = 0) as no of sets are 2 (cl/N =4/2 = 2). The no of cache hits and misses are somewhat **intermediate** when compared to the other two caches.

   The physical address of a N-way set associative cache can be split up into Tag, Set-Offset and Word-Offset.

Tag- (16-(w+s) bits)	Set-Offset(s bits)	Word-Offset- (w bits)

   where w = number of bits needed for representing block size, s = number of bits needed for representing number of sets.

We are subtracting bits from 16 because we've implemented a 16-bit system.
NOTE- All the three cache types and both the levels L1 and L2 described above follow a write-through policy for writing data and LRU scheme for replacement and eviction.

ASSUMPTIONS

1. The command for reading is "xyz READ" where xyz is a valid address.
2. The command for writing to the cache is "xyz WRITE d" where xyz is a valid integer address and d is the data to be written. Address will be an integer in a range 0 to 65535. Data should be a valid integer between -2147483648 to 2147483647 (range of integer in java)
3. Block size, no of cache lines, N, should be in the power of 2.
4. Line size/ Block size of the two levels are equal.
5. In a N-way set associative cache, the number of sets in L1 and L2 are different, however the number of cache lines per set is fixed and same in both the levels, and is equal to k. (FACT)
6. Size of L2 cache (S) = block size * number of cache lines (FACT)
7. K <= no of cache lines (k and no of cache lines should be a valid integer) (FACT)

INPUT

1. Block size, number of cache lines, the value of n for n-way set associative mapping, number of test cases (number of read-write commands).
2. A valid address for reading the cache.
3. A valid address and data for writing to the cache.

OUTPUT

1. If it's a hit or a miss.
2. Data stored in the address (in case of a hit) when read command is given.
3. Physical address, tag and offset for every integer address for both L1 and L2.
4. Incase of a replacement, address of the block being replaced.
5. L1 and L2 cache structure.

CODE EXPLANATION-

CODE1- DIRECT MAPPED CACHE

File name:AnoushkaNarang_2019235_FinalAssignment_code1

- Valid inputs are taken. The number of test cases is entered.
- For every command, a read/write operation is performed (as **mentioned in the Reading and Writing section above**, depending on the addresses' presence/absence in the caches.) For a read operation, the data on that address (if present) is read off.
- A HashMap Data Structure is used to store the Cache Line number, Block number and Block contents of the two caches. Since a block is replaced every time when a call of the cache line (to which the former belongs) is made, we need not store when which block was called.
- In the end, the two caches are printed using the *printMapString* function.

CODE2- FULLY ASSOCIATIVE CACHE

File name:AnoushkaNarang_2019235_FinalAssignment_code2

- Valid inputs are taken. The number of test cases is entered.
- For every command, a read/write operation is performed (as **mentioned in the Reading and Writing section above**, depending on the addresses' presence/absence in the caches.) For a read operation, the data on that address (if present) is read off.
- A HashMap Data Structure is used to store the Block no and Block contents of the two caches. An ArrayList 'freq1' and 'freq2' keeps a track of occurrence of the blocks in the cache and helps decide which of the blocks is/are least recent and has to be evicted according to the LRU scheme.
- In the end, the two caches are printed using the *printfamap* function.
- A function called *'removal'* is a helper function used to remove the least recently used element from the caches and their respective 'freq' ArrayLists.

CODE3- N-WAY SET ASSOCIATIVE MEMORY

File name:AnoushkaNarang_2019235_FinalAssignment_code3

- Valid inputs are taken. The number of test cases is entered.
- For every command, a read/write operation is performed (as **mentioned in the Reading and Writing section above**, depending on the addresses' presence/absence in the caches.) For a read operation, the data on that address (if present) is read off.
- A HashMap Data Structure is used to store the Set number, Block number and Block contents of the two caches. An ArrayList 'freq1' and 'freq2' keeps a track of occurrence of the blocks in the cache and helps decide which of the blocks is/are least recent and has to be evicted according to the LRU scheme.
- In the end, the two caches are printed using the *PrintMap* function.
- A function called *'removal'* is a helper function used to remove the least recently used element from the caches and their respective 'freq' ArrayLists.

COMMON FUNCTIONS USED IN THE CODES-

1. A function ispowerof2() is used to check whether the entered number is in power of 2.
2. A function getn() is used to return the log of a number to base 2. It is used to get the number of bits required to represent block size, number of sets and cache lines.
3. A function getbinary() is used to print the tag of an integer address in binary.
4. A function get_address() is used to print the address in binary, tags and offset values of both L1 and L2 caches.

ERRORS HANDLED-

- If block size, number of cache lines or n for n-way associative are not in a power of 2.
- If the value of n for n-way associative is greater than the number of cache lines.
- If the entered address is an invalid integer, i.e. it doesn't lie between 0 to 65535.
- If a command other than read/write is given.

OUTPUT FORMAT

```
------------------------------OUTPUT---------------------------------
LEVEL ONE CACHE:
Set number    Block number      Block Address             Block Content
0               0             00000000000000              Empty  Empty  Empty  Empty
3               16363         11111111101011              Empty  Empty  76  98

LEVEL TWO CACHE:
Set number    Block number      Block Address             Block Content
0               0             00000000000000              Empty  Empty  Empty  Empty
3               16363         11111111101011              Empty  Empty  76  98
4               16364         11111111101100              99  Empty  Empty  Empty
```