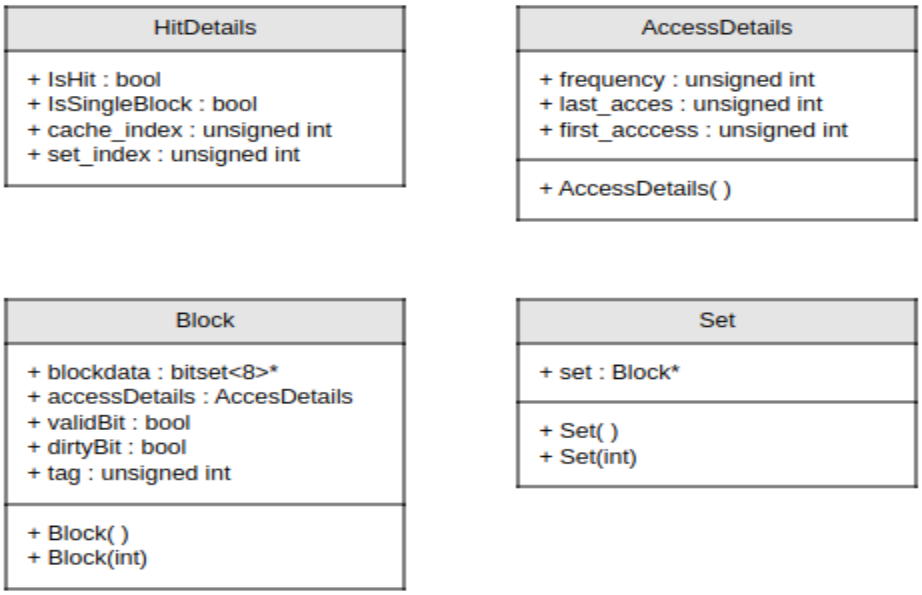
Report

CS23BTECH11018 - Dondeti Samhitha
CS23BTECH11049 - Ramanolla Shravani

Design Section:

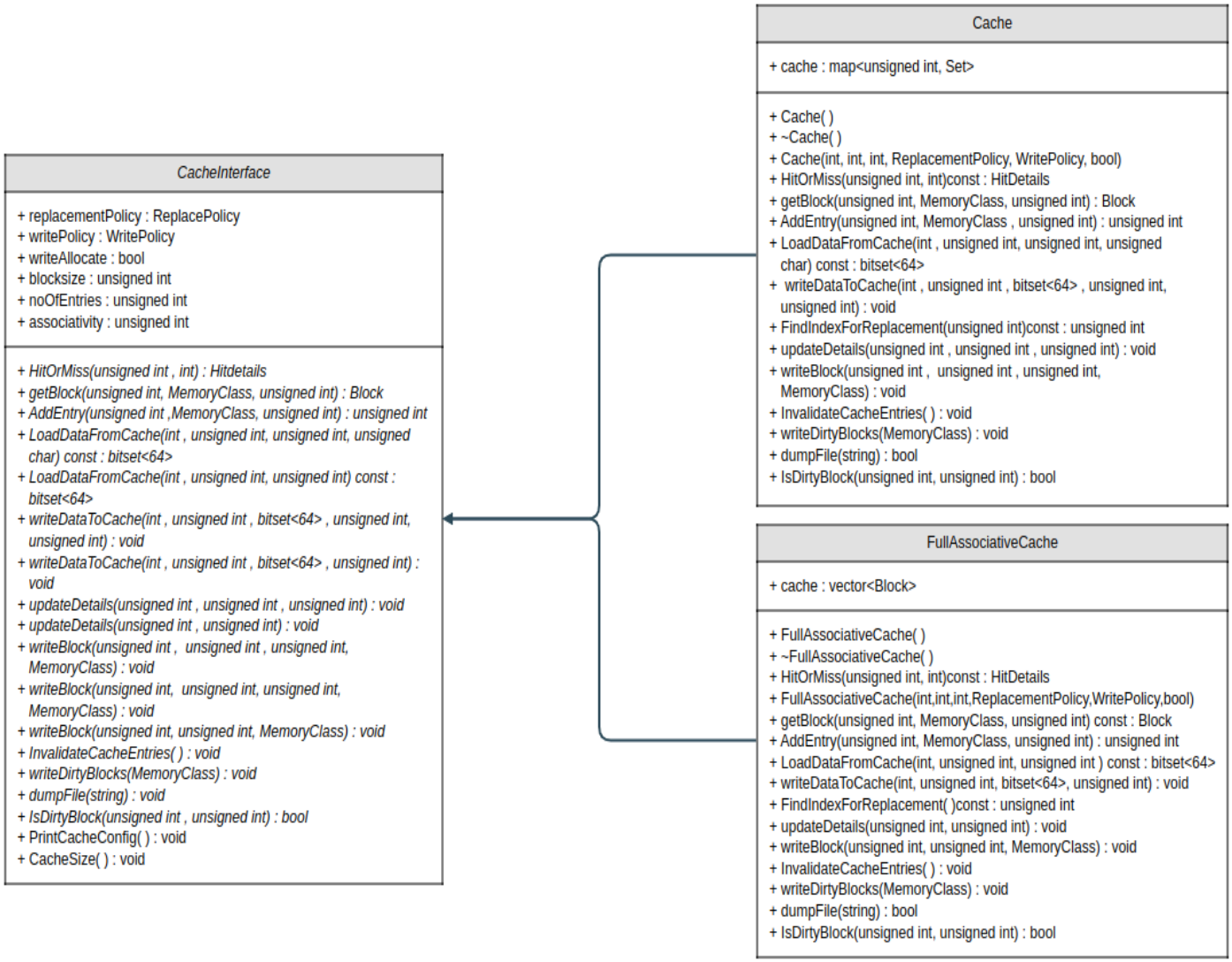
Data Storage:

- Each Block in a cache is represented by a struct named ***Block.*** It consists of details like ***tag, validBit , dirtyBit , accessDetails*** and ***actual blockdata***.
- ***tag*** is represented by ***unsigned int*** (since our address is only 20 bit) and ***validBit and dirtyBit*** are represented by ***bool.***
- ***accessDetails*** is the struct that stores details about that particular block access like ***first_access, last_access, and frequency***. These details are useful when we are choosing a block for replacement depending upon replacement policy.
- ***blockdata*** is a pointer to ***bitset<8>***, since each block consists of an array of bytes (one byte is represented by bitset<8>), the number of bitset's that block data points to depends upon block size. The Constructor of Block takes ***blocksize*** as a parameter and allocates that many number of bytes to ***blockdata.***
- In cache each index corresponds to a set.Each set contains an array of ***Blocks.*** Number of blocks in each set depends on ***associativity.*** Each set in cache is represented by struct ***Set.*** Which contains a pointer to ***Block*** as its member which points to an array of Blocks in that set.



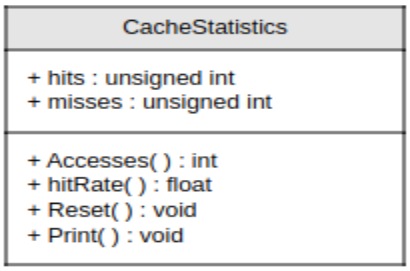
Class Hierarchy:

- We defined a Cache Interface with its members as ***blocksize, associativity, noOfEntries, replacementPolicy, writePolicy, writeAllocate***.
- This Interface also has some virtual functions like ***HitOrMiss , getBlock, AddEntry, writeBlock, LoadDataFromCache*** etc…
- Now we define 2 classes named ***Cache and FullAssociativeCache*** which are inherited from the CacheInteface.
- **Cache** class contains its member ***cache*** as a map from ***unsigned int to Set struct.***
- ***FullyAssociativeCache*** class contains ***cache*** as a vector of ***Blocks.*** Since a fully associative cache has only one set, there is no need for a map from index to set.
- All the virtual functions in the interface are redefined in these two derived classes according to their data structure.



Replacement Policy :

- To Implement Replacement policies like FIFO, LRU we need to keep track of time when each block is accessed.
- So we defined a global variable named ***Timer***, whose value is zero when we load a file, and then we increment it by 1, whenever we execute an instruction. So this variable acts as a clock for that particular program.
- Whenever there is an access to cache, we update the access details of the block which is accessed by incrementing ***frequency*** by 1 and updating ***last_access*** to current ***Timer***.
- If a particular access to cache is **miss**, then we will bring a new block to cache, so we need to equate both ***first_access*** and last_access to current ***Timer*** and make ***frequency*** to 1.
- So when we are replacing blocks, if the replace policy is ***FIFO*** we will replace the block that has minimum value for the first_access, if replacement policy is ***LRU*** we will replace the block that has having minimum value for variable last_access, if the replacement Policy is ***LFU*** we will replace the block having minimum value for frequency.



Functions related to cache:

- ***HitOrMiss*** function takes an ***unsigned int address and int n*** as inputs , and checks whether the access is cache hit or cache miss and it also checks whether the data access is within the same block of cache or in different blocks in a cache.It returns a struct named ***HitDetails*** containing members like ***IsHit, IsSingleBlock, cache_index and set_index. Cache_index*** and ***set_index*** are used to identify the block in cache.
- ***getBlock*** function takes an unsigned int address , reference to Memory Class and unsigned int Timer and returns the struct Block . It creates a new Block and fill it with the memory starting from the given address and it also fills the details of Block like tag,accessDetails and make validBit to true.
- ***FindIndexForReplacement*** functions help us to decide which block to be replaced . In ***Cache*** Class it takes ***cache_index*** as input and returns ***set_index*** for the block to be replaced inside the set that corresponds to ***cache_index.*** In ***FullyAssociativeCache*** class it does not take any input parameters since we only have one set in fully associativeCache and returns an index to Block that is to be replaced.
- ***AddEntry*** function takes an unsigned int address , reference to Memory Class and unsigned int Timer.as input parameters. This function is called when there is a cache miss. It uses ***getBlock*** and ***FindIndexForReplacement*** functions mentioned above and

places the newly brought Block at the appropriate position in cache.It also checks whether we are replacing dirty Block or not , if yes it will write the replacing block to memory.

- ***updateDetails*** function is used to update access details of a particular block like increasing frequency and updating last_access to current Timer.
- ***writeDataToCache*** function writes make changes to data in a particular block inside cache. It takes ***int n , unsigned int address , bitset<64> data*** and other parameters to identify block (like ***cache_index and set_index*** for ***Cache*** Class and ***block_index*** for ***FullyaAsociativeCache*** Class) and it writes n bytes from ***data*** to block at appropriate position.
- ***writeBlock*** function writes a particular block to Memory at appropriate address. It takes an ***unsigned int start_address , reference to Memory Class*** and other parameters to identify block and writes that complete block to memory starting from address ***start_address.*** This function is used when we want to write back dirty blocks to Memory.
- ***InvalidateCacheEntries*** function invalidates all entries in the cache.
- ***writeDirtyBlocks*** function writes all dirty blocks back to the memory.
- ***dumpFile*** function creates a new file and dumps details of all valid blocks in the cache to that file.
- ***PrintCacheConfig*** function prints the details about the cache like its blocksize , associativity , cachesize , replacement policy , write policy etc..
- ***CacheSize*** function calculates the size of cache using ***block size , associativity and no of entries.***
- ***IsDirtyBlock*** function tells us whether a particular block is dirty or not.

Execution Flow:

- When the cache is enabled the execution flow of all instructions other than load and store instructions will not change. Because only for the load and store instructions we need memory access. When the cache is disabled the execution is the same as in the lab4.
- When the cache is enabled and the config file is opened then the global variable ***IsCacheEnabled*** is set to true. By default the variable ***IsCacheEnabled*** is set to false.
- When an address is accessed then according to the address the block_index, the set_index and tag are calculated. Then we check if the given access corresponds to a single block of cache or not and the value is stored in ***IsSingleblock*** variable(0 if not a single block, 1 if single block). If it doesn't correspond to a single block then we don't consider it as cache access and we just do the access from memory itself. Else we check if the particular block with that block_index, set_index has the same tag as the accessed address tag then in HitDetails it is updated as hit else miss.
- When the cache is enabled the command **cache_sim status** displays the configuration of the cache(Cache Size, Block Size, Associativity, Replacement Policy, Write Back Policy) with the function *PrintConfigCache( )* else it displays "D-cache status : disabled".

- When the cache is enabled the command **cache_sim invalidate** calls the function *writeDirtyBlocks(Memory)* (to write every dirty block of cache to the memory before making the block invalid) and then *InvalidateCacheEntrires( ).*
- When the cache is enabled and the command is **cache_sim dump** filename then the blocks of cache which have their validBit as 1 are displayed(their set number, tag and validBit) in the file named filename with the function *dump(string filename)*.
- When the cache is enabled and the command is **cache_sim stats** then it displays the D-cache Statistics (Accesses, Hits, Misses, Hit Rate) with the function *Print( )*.
- When the cache is enabled the command ***cache_sim disabled*** then all the blocks of the cache are made invalid with the function *InvalidateCacheEntrires( )*, ***IsCacheEnabled*** is made false and the cache object is deleted.
- When the cache is enabled, the command **run** at the end of the execution of the program calls the function *Print( )* to print the D-cache statistics.

Error Handling :

- When the cache is enabled and the memory access doesn't belong to one block then the error message is displayed and the access is nor considered in cache statistics.
- Remaining error handling is the same as lab4.

Limitations:

- The cache size and block size should be in the powers of 2 so as to calculate the tag, index, byte offset which is done by shifting the bits.
- Multiple blocks of cache cannot be accessed at a time which is used when the memory accessed isn't present in a particular block of the cache.

Testing:

- We tested our Cache simulator with programs given in HomeWork4 and verified the cache statistics by changing the config file for different configurations and verified it with Ripes Simulator.
- We also created some small examples to verify write through, write back policies and verified whether the memory behaves accordingly or not.
- We also checked with GCD code and compared results with Ripes Simulator.