**Documentation**

**Requirements:**

* Python 3.5+
* Built-in modules used: sys, math, collections, copy

**To load cache with a mapping, rum the file with corresponding mapping’s name.**

**Inputs:**

Main memory size, block size, cache size and set size are taken as inputs (Set size only in set-associative).

**Assumptions:**

* Two-level cache
* Input address is in binary
* No restriction on size or type of data
* Main memory size, block size, cache size and set size are powers of two

**Working of code**

**Main memory is not maintained** i.e. if a block is replaced by others in the second level, its data is lost. At each step, the mapping provides options to write(W), read(R), print(P) or exit(E). On switching between mappings, the cache **does not** remain the same. Addresses and data stored in a cache are lost once the user switches from one mapping to the other.

Loading:

L2(level 2) is **exclusive** of L1(level 1).

* If the address is present in the L1, data is stored in the appropriate position.
* Else if the address is present in the second level, the corresponding block is loaded in the first level and the block which is removed from the first level is loaded in the second level.
* Else if there’s space (in L1) for the block corresponding to the address, the block is loaded in L1.
* Else a block is removed from L1 (following appropriate replacement procedure) and the block corresponding to the address is loaded in L1. The removed block is loaded in L2.

Reading:

* If the address is present in L1, there are no changes in cache.
* Else if the address is present in L2 (there’s a MISS in L1 and HIT in L2), the block is loaded in L1 following the loading procedure explained in the second point of loading.
* If the block is neither in L1 nor in L2, there’s no change in the cache.

On replacement in L1, the program prints the data and address deleted. The program also prints the blocks replaced and blocks loaded at different instances. Associative and set-associative mapping uses a **First-In-First-Out** replacement procedure.

**The program terminates if invalid memory size or invalid memory address is provided.**

**Functions and their working:**

**decimalToBinary(n)**

Converts a given decimal number into its binary equivalent. It is used to find line offset, block offset and set value.

**initialize\_cache()**

*(same for L1 and L2)*

* Directive: Stores line offset as keys in a dictionary. Tag values are initialised as an empty string and block offsets are stored in another dictionary. Initially, data stored is an empty string.
* Associative: Cache is initialized as an OrderedDict (a dictionary subclass that remembers the order entries were added)
* Set-associative: Cache is initialized as a dictionary with set offset as its keys and an OrderedDict as their value.

**input\_new\_block()**

* Takes input on write(W) instruction.
* Program is terminated if the number of bits in the address does not match the word length of the machine.

**create\_block()**

*(only in associative and set-associative)*

* Creates a dictionary with a length equal to that of a block.
* The dictionary has block offset as keys

**l1\_cache\_loading()** / **l2\_cache\_loading()**

* Loads data following methods explained above.
* If required, *l1\_replace\_block()* / *l2\_replace\_block()* is called.

**l1\_replace\_block()** / **l2\_replace\_block()**

* In the case of L1, replaces block and calls l2\_cache\_loading if address not present in L2. Else, calls *l2\_to\_l1\_loading()*
* In the case of L2, replaces block
* Prints the tag replaced.
* Uses a First-In-First-Out replacement method in associative and set-associative.

**l1\_cache\_searching()**

* Searches the given address in L1
* If data is present in L1, prints HIT Data: <data>
  + If the given address does not contain any data, <data> is empty
* Else, prints MISS and calls *l2\_cache\_searching()* to search in L2.

**l2\_cache\_searching()**

* Searches the given address in L2.
* If data is not present, prints MISS
* Else, prints HIT and calls *l2\_to\_l1\_loading()* to load address from L2 to L1 as explained in loading.

**l2\_to\_l1\_loading()**

* Loads the block corresponding to an address from L2 to L1
* Calls *l2\_cache\_loading()* to load the block removed from L1 to L2

**print\_cache():**

* Prints all the addresses and data stored at that instance in the cache.
  + In the case of set-associative mapping, ‘----’ is printed after every set for the convenience of the user.