**Design Document - Phase3**

**Appending a cache like memory module to Phase 2**

**Input/Output**

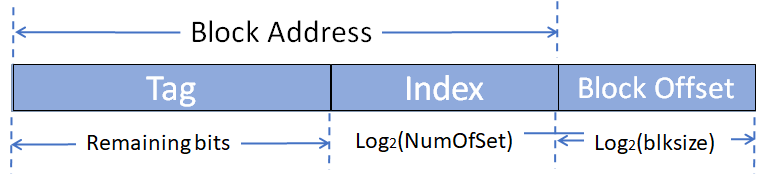
* In this phase, we have created the instruction and data cache modules as **loadFromInstructionCache()** and **loadFromDataCache()** respectively.
* Instead of directly reading from a .mc file like the previous phases, we have read the instructions, read/write data from these cache modules.
* We are effectively modeling the functionality of a Cache, that will not have any impact on the pipeline proceedings. Our cache model is more like an accountant tracking the **accesses, hits, misses**.
* We have instantiated two caches, one will work as **Instruction cache** (instructionCache) and another will work as **Data cache** (dataCache). Therefore, all the requests from the Fetch stage of our pipeline will be handled by Instruction cache and all the requests from Memory stage will be handled by Data Cache.
* Firstly, the user is asked to select the instruction cache input configuration. Subsequently, the Cache Size, Cache Block Size, Number of ways of set associativity is taken as input and stored in **insCacheSize**, **insCacheBlockSize** and **insAssociativity** respectively.
* On the similar lines, input configuration of Data Cache is asked and corresponding parameters - **dataCacheSize, dataCacheBlockSize** and **dataAssociativity** are taken as input.
* **Output Parameters**

The following parameters are displayed on the terminal

* Number of accesses
* Number of misses
* Number of hits

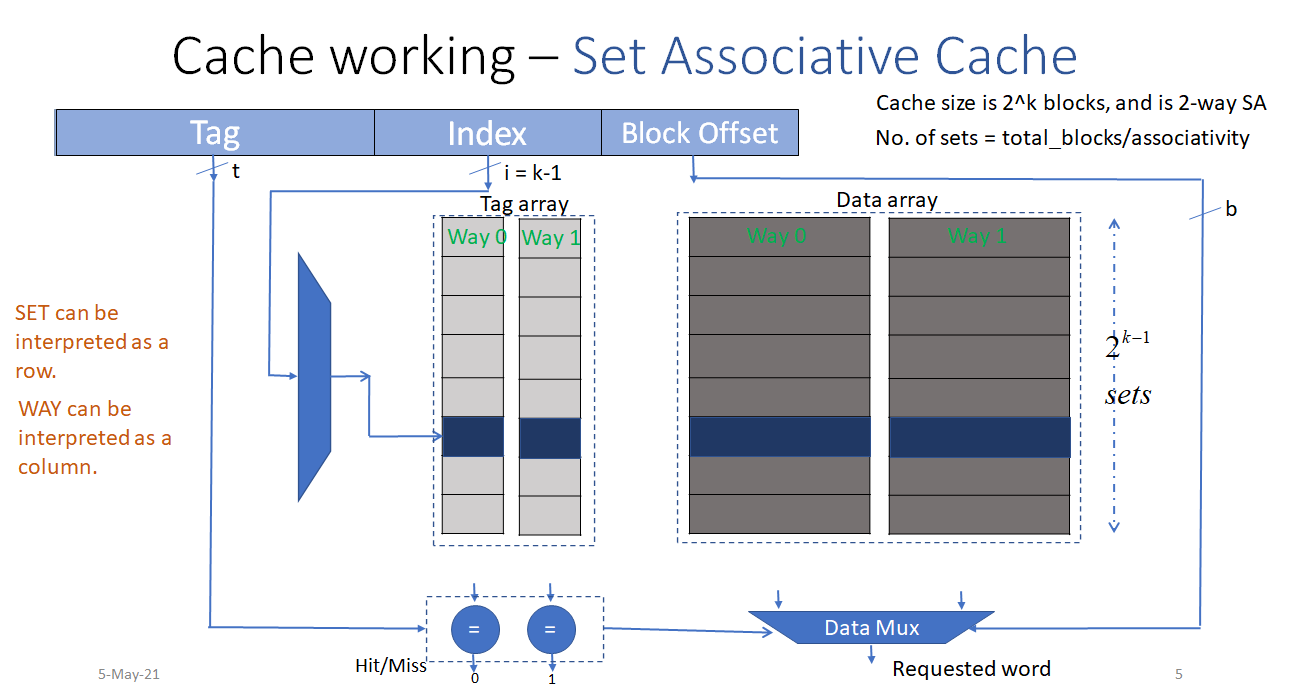
At the end of our simulation, two sets of stats are printed - one for Instruction Cache and another for Data Cache.

* **Set Associative Cache Working**



* **Extracting Tag, Index and Block Offset**

1. **Block Offset(insBlockOffSetSize/dataBlockOffSetSize):** The block offset is calculated by taking the log with base 2 of the cache block size, i.e. Log2(blksize)
2. **Index(insIndexSize/dataIndexSize):** The index is calculated by taking the log with base 2 of the Number of sets, i.e. Log2(NumOfSet)
3. **Tag(insTagSize/dataTagSize):** The tag is calculated by subtracting the index and the block offset from the total number of address bits.



**Fig: Shows the implementation ofSet Associative Cache through an example**

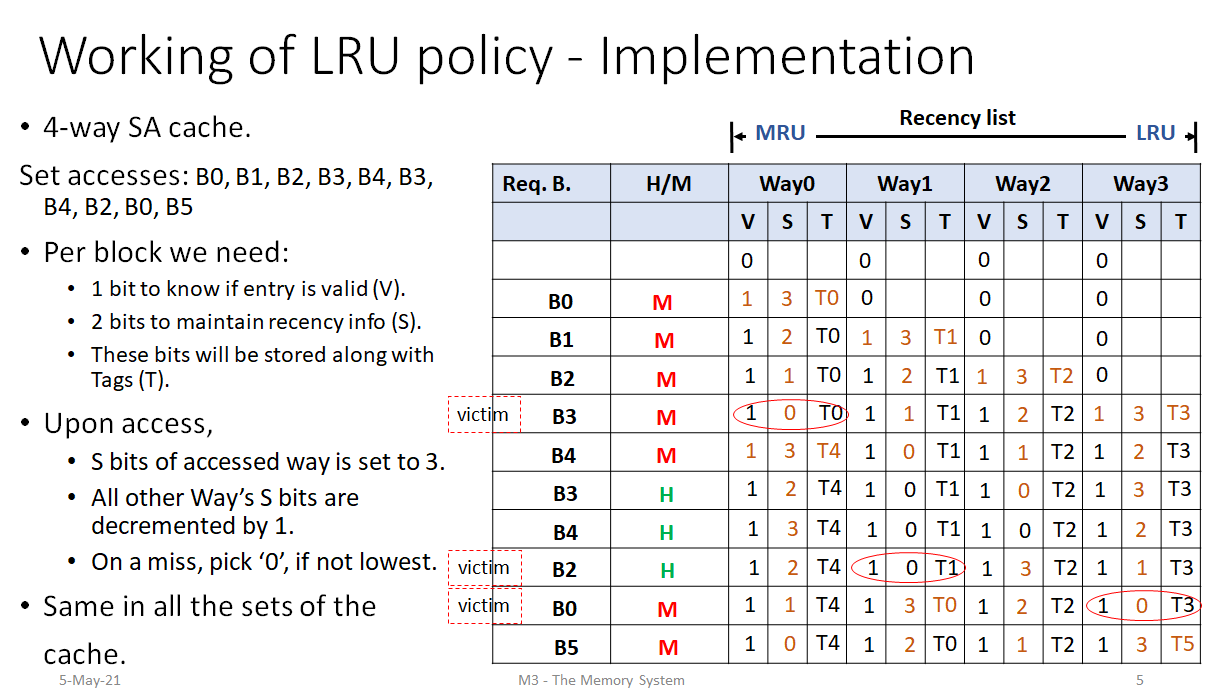
(Source: Lecture Slides)

* **Determining hit/miss**

The Tag from the requested address will be matched with tags of each block in this one set.

1. If all the tag comparators output is 0, then it’s a **miss**.
2. When one of them outputs 1, it’s a **hit**.

* A word (based on block offset) of data of the Hit tag will be sent out.
* **Replacement Policy**
* LRU is implemented as a replacement policy. The tag value is stored in a dictionary dataCache and instructionCache.
* If the tag value is not found in the Cache, i.e. there is a miss, then the block having “0” State is selected as the victim block and evicted to be replaced by the current block.
* The dataCache and instructionCache are updated in accordance with the LRU Policy.

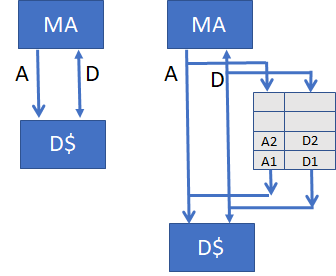


**Fig: Shows the implementation of LRU Policy through an example**

(Source: Lecture Slides)

* **Implementing Writes**

For implementing writes, we modified the block only after the tag is matched.

****

**Fig: Shows the implementation Writes**

(Source: Lecture Slides)

* **Write Policies Followed**

The writes are implemented by following **Write Through** -> “Write Policy” and No **Write Allocate** -> “Write Allocate Policy”.

* **Write Through:**
* Upon a write, data will be written to both the block in the cache and its next-level memory.
* If L1 is Write Through cache, then all writes to L1 will also be written to L2, i.e., Data in L1 and L2 will always be the same.
* **No Write Allocate**

1. Upon a write miss, space will not be allocated for the block.
2. The write will be forwarded to the lower level of memory.
3. The contents of this level do not change.
4. Preferable choice if data written is not reused.

* **Test Plan**

We test the simulator with the following assembly programs:

* Fibonacci Program
* Bubble Sort
* Factorial
* Sum of the array of N elements. Initialize an array in the first loop with each element equal to its index. In the second loop find the sum of this array, and store the result at Arr[N].
* Lab-2, Lab-3 machine codes

**--------------------------------------------------------------END-------------------------------------------------------**