RISC-V ISA SIMULATOR : PHASE 3

(Pipeline with Caches)

This Phase of RISC-V ISA Simulator is a continuation of Phase-1 and Phase-2, with addition of cache memory, along with pipeline design. In this phase, instruction and data cache modules are created and instructions are read from these modules indirectly.

Input

A *.mc* file similar to previous phases is read which is used to create instruction and data segments of main memory.

The GUI interface will also ask the user to input cache associativity, cache size and block size of cache.

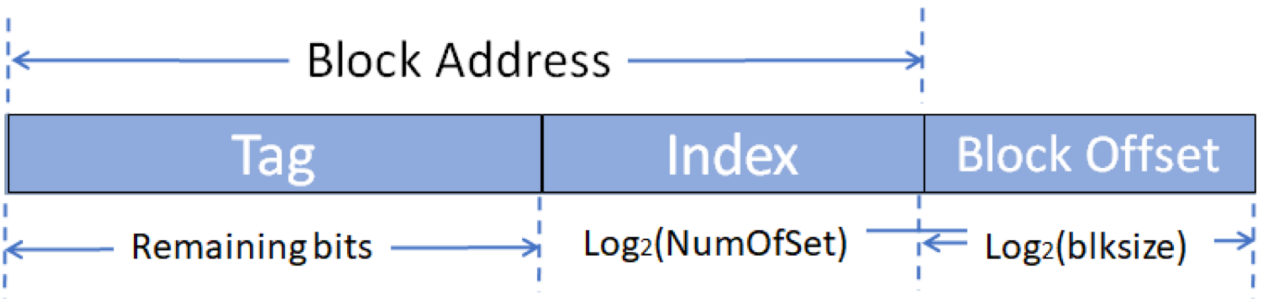
Cache memory will get continuously updated during the simulation as data and instructions are fetched or accessed from the main memory. This simulator will read from the created cache modules.

Output

Along with previous phase output, following parameters will be displayed on the GUI interface:-

- Data cache and Instruction cache.
- Recency list associated with cache sets.
- Number of cache misses, cache hits, and cache accesses.

Address fields of Set Associative Cache



The address is divided into three parts: *Tag, Index, and Block Offset.*

Following functions are utilised in our simulator program to implement set associative cache:-

- **addToTagAdd()**

This function creates the address consisting of *tag, index, and block offset* fields.

- **getBlockOffset()**

This function extracts *block offset* from address.

- **getIndex()**

This function extracts *index* from address.

- **getTag()**

This function extracts *tag* from address.

Reading from Cache

dataReadCache()

- *Tag, index and block offset* fields are extracted from address.

- Index is used to identify cache set, then tag is compared with blocks of that set and if it is a cache hit, block offset is used to extract required word or byte from that block. If no tag is matched in that set, then the simulator has to deal with cache miss.

- In case of cache miss, a block needs to be brought from main memory to the cache memory. If the corresponding set contains an empty space to accommodate a new block, then this space is used to do the same. But, if all the block spaces of the set are already occupied, then a victim needs to be chosen from that set which would be replaced by the new incoming block from the main memory. This replacement is done by following **LRU replacement policy.**

LRU Replacement Policy: According to the replacement policy, some extra bits of information is maintained with each *Way* of each set of the cache, which will store the value according to the recency of the block which is stored in that *Way.* The block which is most recent will be associated with the highest value and the least recent block will be associated with the smallest value. According to this value, the least recent block is selected as victim, and is replaced by newly requested block from the main memory.

This is implemented using a dictionary *lru_data.*

Writing to Cache

dataWriteCache()

- The blocks are written only after the tag gets matched.

- **Write Through** and **No Write Allocate** policies are followed.

Write Through Policy: Upon a write, data will be written both to the cache block and its next level memory(main memory).

No Write Allocate Policy: This policy is implemented when there is a right miss. In this policy, space will not be allocated for the block and write will be forwarded to next level memory(main memory). The contents of the cache on which the simulator is working do not change.

Test Plan

The simulator is tested by assembly programs of *fibonacci series, bubble sort, and recursive factorial.*