Cache Simulation Overview

Elena Ryan and Jenna Olson

Our cache takes the form of a 3-D array created using the user inputs. It is initialized after user input as:

Cache[number of sets][associativity][block size in words +3]

The third dimension is initialized +3 in order to store valid bit in position 0, dirty bit in position 1, and tag in position 2, followed by each word in the associated block.

In main, within instruction fetch, load word, and store word, the cache operates extremely similarly. Based on the address that needs to be read from (lw, fetch) or written to (sw), the cache determines what the block tag holding that address would be by using integer division to calculate floor(address/blocksize). Using that calculated tag, the cache then determines which set the block should be in by setting set = block tag % (num sets). Once the set is determined, the cache searches through the associativity for a block that matches the tag, if one exists, it finds the entry associated with the original address by indexing into the block at addr % block size + 3 (offset from initialization), and loading that value from cache into the processor, or writing to that entry and updating the dirty bit, depending on the instruction.

If, in any of these cases, the appropriate block is not loaded into cache, the cache loads the appropriate block from memory into an invalid cache block. If none exists, the cache evicts the last recently used block (writing back to memory if dirty), and writes the block into that entry.

LRU is implemented by initializing an array at the beginning of the program:

LRU[number of sets][associativity]

Whenever a cache block is accessed by the processor, its corresponding entry in the LRU array is updated with the program counter. When identifying the LRU block in a set, the cache finds the minimum LRU in the set and implements the eviction and write back.

When the machine receives a “Halt” instruction, the cache writes all dirty blocks back to memory before ending the machine.

The biggest difficulty in implementation was determining how to best create the cache and ultimately deciding on the 3D array. Another challenge was mapping out exactly how the blocks in cache relate to memory and deriving block number from address and vice-versa. This ended up being fairly simple, but required a little more thought. Implementing the LRU policy was a little challenging because it requires keeping track of how recently each cache was used. Deciding on a 2D array for LRU ended up being a really nice solution for this problem.