Grayson Colwell and Jessica Cashman

Design Document

**Overview:**

We made a few changes to the data structures we were given. Block’s last\_used value is now initialized to 0 in the Block constructor. We also made a method for resetting the cache (resetCache) in the Memory class. The test files all call the resetCache method after resetting the clock and swithing between one of the cache organizations. We made 6 additional methods for the Cache class to separate the process of getting and putting data in the cache by cache organization. The method putData will call either putDirect, putFully, or putTwoWay and the method getData will call either getDirect. getFully, or putTwoWay depending on the organization used.

**Direct Mapped:**Direct mapped allows a given address to be stored in a specific block in the cache based on the address’ value. The word index (wordIndx) was calculated by bit masking the address with 0x03 to isolate the last to bits. The block index (blockIndx) was calculated by bit masking the address with 0x01C and right shifting out the last two bits. The remaining bits were assigned to curTag by right shifting out the five least significant bits. The programs then check the cache to see if the addresses at the appropriate block matched the input address. In getDirect if the addresses match, the method returns the data stored at the location. In putDirect if the addresses match, the method will place the input data in that location. The main memory location is also updated after this.

If the address does not match it is a miss. In this case both getDirect and putDirect will access MainMem and place the block at the address in the cache. getDirect will then return the data in the cache and putDirect will place the input data in the cached location.

**Two-way Associative:**

//how to calculate cache index, tag, word offset, and how to implement the LRU algorithm

Two-way Associative organization allows the address to be stored at two potential blocks in the cache. The blocks are divided into 4 sets instead of 8 blocks. The wordIndx and blockIndx are calculated the same as before, but now there is also a setInx which is equal to blockIndx % 4. The curTag is also one bit larger so it is right shifted 4 bits instead of 5. The input address is compared to the addresses stored at the two blocks it could be stored, either the cblock[setIndx] or cblock[setIndex + 4]. In getTwoWay if the addresses match, the method returns the data stored at the location and increment the last\_used value. In putTwoWay if the addresses match, the method will place the input data in that location and increment the least\_used value. The main memory location is also updated after this.

If the address is not at either location it is a miss. In this case both getTwoWay and putTwoWay will access MainMem and place the block at one of the blocks in the cache. It will decide between the two potential blocks by comparing the least\_used values and placing it as the block with the lowest value. (If they are equal is just uses the one at cblock[setIndx )getTwoWay will then return the data in the cache and putTwoWay will place the input data in the cached location.

**Fully Associative:**

When looking at the implementation of the word offset in the fully associative implementation, the same implementation for the previous two methods was used. The address passed into the function was “anded” with 0x00000003 to get the lower two bits of the address as the word offset. Because this method was fully associative, it meant that any cache block could hold the address passed into the function. Each of the 8 cache addresses were defined as being the tag at each cache address shifted left two bits and added with the word offset. Based off these addresses, each cache location was checked to see if the input address was found in any of the locations. The tag was the remaining bits after calculating the word offset, so the address passed in shifted right two bits would contain the tag for each word in the cache.

When implementing the LRU algorithm, the first thing done was to create two temporary variables. These two variables used were tempLast and tempLastIndex. The cache was then parsed through to find which of the locations was used the least and based off this, this is where the address block from mainMemory would be placed in the cache.

**Performance Analysis:**

The smart tests generally have lower miss rates than the naïve tests. This seems to be because the smart tests increment through the words before the blocks/sets. This makes it faster because a whole block is placed in at once when a single address is not present, so its neighbors will already be in the cache on the next iteration. The fully associative organization has the lowest miss rates and clock cycles. The Two-way associative organization was generally more prone to misses than the direct organization, but it did much better at the 25 smart test than direct.