PROJECT 4 - OVERVIEW DOCUMENT

Project Overview:
Contained in this document is an overview of the simulator.c file for the Project 4 package. This project is a behavioral simulator of a single-cycle processor with a cache. The cache uses a write-back write-through policy, only writing cache contents back to memory when necessary to free up space by evicting them from the cache. The overview details include instructions on command line arguments used (the block size, number of sets and associativity of the cache are specified on runtime as command line arguments), major functions necessary for the software's simulation, and a general run-through of the behavioral simulator's functionality.

main():

The cache simulator begins by first reading in an assembly code text file with the following parameters: block size, number of sets, associativity. The command line argument follows this format:

./simulator test.m <blockSize> <numSets> <associativity>

Given these input parameters, the cache simulator verifies that both the block size, number of sets, and associativity are powers of two (per the assignment instructions). Directly following that check, the simulator checks to verify that the number of cache blocks is not greater than 256. If these conditions are not met, the simulator will exit with an error.

If the conditions are met, the simulator initializes our cache struct to be used for the remainder of the program's execution. Contained in the cache are the block size, number of sets, associativity, an array representing the lru, valid bit, dirty bit, tag, and the actual data being stored for each cache block.

Following the initialization of the cache, the lru array is initialized from [associativity – 1] to 0 for each set, and every cache tag is set to -1 (the placeholder for tags that are not currently in use). The assembly code file is then read into the simulator and the normal checks are accomplished to verify we have a valid file. The instructions are then read into instruction memory.

Once the file contents have been transferred to our simulated memory, we begin by first checking the cache to see if the current instruction is located there. To check to see if the cache contains the instruction, we call the "accessCache" function (this function is also used to get the data for LW/SW throughout the entirety of the program).

> **accessCache**: The accessCache function acts as the check used on the cache to verify if the data we need is located within the cache. The function begins by determining the block offset (calculated by using the number of block offset bits and the address). Once the block offset has been determined, accessCache determines the index using the same function as just described. Once the block offset and index have been determined, we then determine the tag by bit shifting the address right by [the block offset bits] + [the index bits]. accessCache then reads in the associativity, block size, and determines the lru block using the **findLRU** function (findLRU uses a simple loop to determine which block was the least recently used). With this information, the function goes through the required set and looks for the tag. If the tag is not found, we use

the LRU block and determine what to do with the data in the given block (if there is even any data there at all) by checking both the dirty bit and the valid bit. If necessary, data will be evicted to the appropriate location given the value of the dirty bit. The cache block will then be filled with the appropriate data and the data will be returned. On the flip side, if there was a hit, the data will be received from the cache and returned. Once this operation has occurred, the lru will be updated.

findLRU: This function uses a simple for loop to return the maximum element in the LRU array. Each of the elements of this array are incremented after each instruction except for the element indicating the block that was just accessed, which is set to zero.

After the instruction has been loaded into cache, the normal single cycle simulator operations occur per the performance requirements of Project 1. The changes that have been made to the original single cycle simulator appear when the LW and SW checks happen. Instead of reaching out to memory for the data we need, both LW and SW use the accessCache function (described above) to check the cache for the data we need.

This operation in main, described above, continues for all instructions in the assembly code file.

Difficulties in implementation:

The majority of the difficulties encountered in this project came from the “accessCache” function. Initially, our main difficulty was in figuring out how the addressing would work in the cache. At first, it was a learning curve figuring out how the addressing concept worked and then finally implementing that into our cache simulator. Further on in the project, determining in what location to use “printAction” in our “accessCache” function became a challenge. Since we were working with both LW, SW, and IFs, all in possibly different scenarios, we had to account for hits, misses (including the different types of misses), evictions (and the types of evictions, either to memory or to nowhere based on the dirty bit), using the LRU when necessary, etc. All of these different scenarios made it challenging to determine when to log the different cache actions, but after reorganizing the logic of the “accessCache” function, it better reflected the flow of operations that an actual cache would execute.