Daniel Gapper
11758227

The first thing I did to create my cache simulator was create three structs:
“CacheLine”, which contains the tag, valid bit, and least recently used counter.
“CacheSet”, which contains a pointer to a collection of “CacheLine” s
“Cache”, which contained the number of set index bits, number of lines per set and number of bit blocks and a pointer to a college of “CacheSet” s

Once I had these three structs outlined, I moved on to initializing the cache within “init_cache”. I first created a new cache and made it a pointer. In doing this I allocated the appropriate amount of memory based on the size of “Cache”.
I then went through a similar process as I did for the “Cache” with the “CacheSets” within “Cache”. This time to determine the amount of memory to allocate I took the size of a cache set multiplied by the number of sets. I determined the number of sets by using 2^s where s is the number of set index bits.
Once I had completed this I set the number of set index bits, number of lines per set and number of bit blocks to their appropriate value that was passed into the function.
I then needed to allocate memory for each cache line within each set. I did this with a for loop where the loop went through each set and allocated the memory for the lines within. I then added a nested inner loop went through each line contained within each set and set the least recently used counter, tag and valid bit to 0.
Once I completed all this, I returned the newly initialized cache.

My next step was parsing the appropriate trace file, and I wrote my logic within the “parse_tracefile” function. To do this I first opened the trace file and read through each line contained within it. If the operation was an instruction load, I skipped over it, but for every other operation I called the simulate cache function and passed in the appropriate parameters.
Once I had read each line from the trace file, I then closed the file.

The next, and most complicated function I wrote was “simulate_cache” which was called from “parse_tracefile” for each line contained within the trace file. This function started by extracting the tag and set index, once this was done, I initialized the necessary variables. My next step was to go through each line within the set and determine if it was a hit or not. A hit was when a line was valid, and the tags matched. Once I had done this, I made sure to track the first empty line so I could use it to store data. I then grabbed the least recently used line for eviction if necessary. My next step was handling if it was not a hit. For these cases I first added one to the miss count and then checked if the empty line had been updated since it was initialized. If it had been updated, I then overwrote the empty line with the current data. If it hadn’t been updated, I then added one to the eviction count and overwrote the eviction line which was determined as the least recently used line. The next step was to go through every valid line and add one to the least recently used counter. To handle the modify operation I added a statement that counted an extra hit.

Now that I had these functions defined and paired with the provided “print_summary” and “print_usage” functions I moved on to implement main. The heart of main contains a while loop that uses the “getopt” function to read each flag and corresponding argument. For each iteration it takes the flag that was read in and calls the appropriate switch case which extracts and saves the argument value to the specified variable. Once all operations have been read, I do a check for errors and print out the details if the “-v” flag was called. From here I then call my “init_cache” function followed by “parse_tracefile” and “print_summary”. I complete my project by freeing any allocated memory.

Thanks for reading!