Computer Architecture Brandon Pramann

EECS 645 KU id# 2966435

Final Project

My approach to creating a simulation of cache coherence was to create a virtual world where for every object in real life there was an object in the code. I started by studying the interactions of the processor and the caches and also how they used the bus to share or request data. After that, I did calculations to figure the size of each of the components in my model. I choose to do the 32 KB, 64 byte catch-line with 2 way LRU replacement policy for all processors. This meant that I would have 2 ^ 8 number of lines. My index would therefore be 8 bits and my offset would be 6 bits. This would leave my tag with 18 bits. My next step was to parse through the trace files and divide it into an array of time stamps, an array of Booleans that described whether the processor should write, and an array of memory addresses. I then iterated through every memory address in the array of memory addresses and converted every one into binary before I split it into an array of tags, an array of indexes, and an array of offsets. After this, I created a function in my computer object that would look at the first item of the time stamp array for each of the processors that the computer had. Then it would get the id of the processor that had the lowest time stamp. It would then have the processor with the lowest time stamp run an instruction using the data at the front of the arrays for the trace that was associated with the processor. After the instruction had run, the processor would then pop that data of the front of every trace array it had. This would allow the computer to run the instructions in the order that the time stamps provided. In order to run an instruction based off the data, it would first check to see if the data was read or write. If the data was write, it would send a signal over the bus to invalidate the other caches data before it would write the data to it’s own cache and change its own state. If the data was read, the processor would first check to see if it had the data on it’s own cache before it would check the other processors to see if they had the data. If the processor had the data, nothing in our model would change. But if the processor did not have the data and other processors did, it would get the data from those processors and then it would increment the counter that was keeping track of how many times that the caches transferred data between itself. If the processor had no other choice other than to get the data from the memory, it would do so. After all these cases, it would change the state machine for the cache data appropriately.

For this project, I worked alone and although I was unable to fully complete and test the model, most of the components are there and the model is close to working properly. When the program is run using “Ctrl + F5”, the results will be displayed to the console. Here is a visual copy for your enjoyment. 