ECE 4750 Lab 4: Multicore Processor

Carly Swetz (cms464), Raghav Kumar (rk524), Kyle Betts (kcb82)

1. Introduction

In this final lab we completed the design of our processors. As a milestone of our semester's work we benefited from the modularity principles that we have emphasized in each lab, enabling us to quickly and easily construct the two processors required. A majority of this lab was simply connecting the modules we had created through labs one to three to produce our final designs. In the baseline processor, we created a single-core processor with its own instruction and data cache. For the alternative design we created a multicore processor with private instruction caches and a shared, banked data cache. The largest part of this lab was evaluating the performance of both the baseline and the alternative design. Due to the slowing of Moore's law there has been a transition to multicore processors as a way to continue performance improvements. This lab gave us a first hand perspective of the power of multicore processors. We evaluated both designs on five different baseline algorithms -- binary search, quick sort, vector vector add, complex multiplication, and multirate filter. The multicore processor had a lower CPI in comparison to the single-core processor for every algorithm, with an average CPI speedup of 4. This is a massive improvement and shows a key reason for the adaptation of multicore systems. Another metric we tracked in the algorithms was the instruction cache miss rate. The single-core processor performed better in this metric for each algorithm except the sorting. On average the single-core processor had 1.1% less instruction cache misses than the multicore processor. While the multicore processor does improve CPI significantly, we do recognize the tradeoffs that come with it. Most importantly, by adding three more cores the required area is increased dramatically as well as the required power. In our multicore processor we only run a single algorithm at a time so our cores are always running until which point they finish and are stopped. In practice, when running continuous programs, finding ways to put cores into sleep when not being utilized, and then awaking cores when needed would be crucial for reducing power consumption. Finally the additional cost in hardware must be recognized by quadrupling the logic required to implement the cores.

2. Baseline Design

In this lab, the baseline design is a single-core system which includes the fully bypassing processor from Lab 2 and two instances of the set-associative cache from Lab 3 (one for the instruction cache and the other for the data cache). The composition for the single-core system can be seen in **Figure 1**. As mentioned, the single core consists of one processor which is our five stage pipelined, fully bypassing processor. This design choice was made because the bypassing processor proved to have better overall performance in our Lab 2 evaluation. Since there is only one core is this design, when instantiating the processor module, the core ID is set to 0. Next, the single core system also uses our two-way, set-associative cache for both the instruction cache and data cache. Once again, the set-associative cache was used instead of the direct-mapped cache because it showed to have better performance on the majority of benchmarks in Lab 3. For both caches, we set the number of banks to 0 since the single core does not need a banked configuration. Finally, the baseline design includes code that calculates several statistics used for evaluation including the number of committed instructions and the number of misses and accesses in each of the caches. The system uses the val/rdy signals to know when each cache is accessed and whether or not there was a miss. Overall, because the baseline design is primarily made up of previously constructed modules, it exhibits modularity and hierarchy design principles. The single core system is a good baseline for comparison because it is what we have studied in class and is more straightforward than the multicore design.

For the software part of the baseline design, we were tasked with writing a quicksort algorithm in C that we would then execute on the single core implementation that had been designed by us but put together by the course staff. The main objective of the baseline task in this lab was not to implement verilog designs but rather work on the high level software, motivating the importance of hardware-software co-design. The quicksort algorithm is a famous sorting algorithm and is unique amongst our benchmarks because it is the only recursive algorithm in our suite. The main idea of the algorithm is to choose a seemingly random pivot in the array and then sort the array

around the pivot such that all the elements less than the pivot are on the left side of the pivot while all the elements more than the pivot are on the right side. This is done via 'partition' function and as a result the pivot element is at its final spot in the array. The algorithm then recursively applies the partition function on the two sub-arrays on either side of the pivot since the sub-arrays are not sorted in their own right. The base case for this recursive function is the case when there is either only a single element or no element in the array at which point the partition function just returns. **Figure 2** does a good job of helping visualize the serial benchmark.

3. Alternative Design

The alternative design is a multicore system that consists of four processors, four private instruction caches, four data caches that are in a banked configuration, and four-port networks to connect the subsystems. To try to achieve the highest performance possible, we choose to use the fully bypassing processor and the two-way, set associative cache for each of the four cores. The diagram for the multicore design can be seen in **Figure 3**. As seen in the diagram, the design includes the MemNet and McoreDataCache subcompositions. Therefore, we take an incremental design approach to build the multicore system. The first subcomposition included is CacheNet, which is inside of the McoreDataCache module. The diagram for the CacheNet module can be seen in **Figure 4**. CacheNet is responsible for connecting the four processors to the four-banked cache configuration. To do this, it instantiates two bus networks from last year's Lab 4, one for the request network and the other for the response network. The module also includes upstream and downstream message adapters to convert between memory and network messages. Next, **Figure 5** shows the composition of the MemNet module, which is used in both the McoreDataCache module as well as the culminating MultiCore file. MemNet is constructed similarly to CacheNet except it only has one output terminal. This is because the four caches need to send requests to only one main memory port. Then, CacheNet and MemNet are combined in McoreDataCache, which can be seen in **Figure 6**. The McoreDataCache module completes the shared, banked data-cache subsystem.

Although we were given the CacheNet, MemNet, and McoreDataCache modules, we needed to compose them together in the final MultiCore system, which once again can be seen in **Figure 3**. We first instantiated the MemNet module seen at the top, which takes memory requests from the four private instruction caches and outputs one main memory request to port 0 of the Test Memory. This MemNet module also takes a main memory response form the Test Memory and directs it to the correct instruction cache. Next, we created an instance of the McoreDataCache for the data cache network. This McoreDataCache takes memory requests from the processors and sends it to the Test Memory through port 1. It also takes the memory response through the cache network and sends it back to the associated processor. Lastly, we instantiated the four private instruction caches and four processors using a for loop. We make sure to set the number of cores to 4 and assign the corresponding ID to each core. After constructing the MultiCore system, the final step was to gather statistics for each core and the system overall. We keep track of the number of committed instructions for each processor, note when there is an access and sometimes a miss in each instruction cache, and get the data cache statistics from the McoreDataCache module. As one can see, the MultiCore system clearly exhibits modularity given it mostly consists of existing modules.

For the software part of the alternate design, we decided to implement the parallel bottom-up hybrid mergesort/quicksort algorithm as the parallel benchmark to run on our multicore design. This algorithm essentially divides up the input array into four (or as many number of cores) equal sub-arrays and then hands each subarray to a different core to perform quicksort on. The *bthread_spawn* function allows us to invoke the other threads. The data that the core needs to do its sorting is passed via a struct while the sort function is passed essentially as a function pointer. The key here is that each core can perform sorting on its own subsection of the array without having any dependencies with the data of the other cores. This data level independence allows us to easily write and execute correctly a parallel algorithm without having to worry about cache coherence and other conflict resolution techniques. Once each core completes its sorting, it waits until core 0 completes its own sorting after which core 0 sends out a 'join' message via the *bthread_join* function. If any one of the cores hasn't completed its work, core 0 waits at the join call for that thread and continues further only after the thread has merged back. It is important to note that each core individually works on a subsection of memory, which these subsections together make the final

array. So when the threads join back in core 0, the memory essentially has the original array with 4 different subsections sorted. So then, in the last step core 0 needs to cleverly merge these individually sorted subarrays into the final sorted array which it does using the *merge* function. The merge function takes two sorted arrays and returns a single sorted array which is the combination of the elements of the two arrays.

4. Testing Strategy

In testing our two designs we benefited greatly from the elaborate testing we implemented for each module we created during the semester. By using unit testing in this manner, in combining each module together to form our entire processors we had faith in the correctness of each module composing the processor. This meant if any incorrect behavior was found, we knew it would be in our connection of modules, greatly reducing the amount of time needed to identify the bug. In a large complicated design like a processor this is almost necessary, as the alternative would be a nightmare trying to test and debug the module as one single unit. The testing for the two processors was kindly provided to us for this lab. First, there was provided directed tests on the cache network, memory network, and multicore data cache network. In implementing these three structures first and then testing them for correctness we were utilizing the test-driven development process. Next, after assembling the two designs, we used the provided directed, black box tests on each type of instruction. In running these tests we were able to confirm proper implementation of our two designs. Since we implemented the sort algorithms ourselves we created tests for each. For both the single-core algorithm and the multicore algorithm we used black box unit testing. We chose this type of testing since it was quick to implement and also the underlying functionality of how the algorithm was implemented was not important to us, as long as the output array was sorted. In providing numerous tests to corner cases such as duplicate values, only negative, only positive, empty arrays as well as arrays of varying lengths, we were more than confident our algorithms were working correctly.

5. Evaluation

The most exciting part of this lab was the ability to run high level C benchmarks on the processors and caches we had designed earlier in the semester. By combining the multiple instances of the processor and leveraging the ‘banked’ nature of the caches, we were able to implement a multicore processor that was able to run parallel versions of the various serial benchmarks. This gave us a much needed exposure to parallel processing which is of great importance in an age where Moore’s law of transistor scaling has flattened out and higher performance is derived out of clever techniques such as superscalar execution and multicore processing.

The first statistic that jumped out when we compared the single core to the multicore was the improvement in the CPI across the range of benchmarks as seen in **Figure 7**. For all algorithms, the multicore improved performance (throughput) since it was able to process almost 4 times more instructions than the single core. It should be noticed that for each benchmark, each core in the multicore has a CPI number close to that of the one core in the single core processor. Thus, it was only due to the fact that we could operate 4 cores in parallel and have each core work on independent instructions that we achieved this almost 4X improvement in CPI. If the instructions were tightly coupled and there were a lot of dependencies in the assembly, then each core would have to synchronize the data with other cores reducing the potential speedup. However, in our case, all benchmarks are able to divide up the work amongst the different cores without any major dependencies allowing the speedup seen. We notice that cmult shows the greatest improvement in CPI going from the single core to the multicore while mfilt shows the least. This is due to the unique structure of the programs and the difference in the relative efficiencies obtained when parallelizing the serial code.

We also noticed that barring the exception of the sorting benchmark, the number of committed instructions per core in the multicore design is considerably less as compared to the one core in the single core design as seen in **Figure 8**. This makes sense because when running on the multicore, each benchmark is able to divide up the work amongst the cores. However, it is very interesting to note the increasing instruction cache miss rate for the multicore design for these benchmarks as seen in **Figure 9**. This is probably due to the fact that in a single core, the cache is given adequate instructions to first ‘warm up’ after which the processor is able to capitalize on the cached

instructions as the benchmarks are very ‘loop’ heavy and the same instructions keep on repeating. When the total instructions get divided up among the cores, each core finishes operation before the cache is fully warmed up and hence the high instruction cache miss rate is due to this warm up period. Now had there been more instructions in each core, we would have observed a better miss rate since the warmed up cache would have reduced the number of misses but that isn’t the case with such few instructions.

In the light of the above arguments, it is surprising to see the sort algorithm committing almost an equal number of instructions to each core in the multicore design as compared to the one core in the single core design. It is odd because we *are* dividing up the work among the 4 cores in the code and thus each core should presumably have to handle less instructions. This is also surprising given that the multicore processor still improved CPI by ~4X in-spite of the heavy load on each core. For all the other benchmarks, it makes sense why multicore improves CPI. Because each of the 4 cores handle less work, the overall CPI drops despite the fact that each core still operates at the high CPI of an individual core.. Thus, it is strange to see the CPI fall but the committed instructions per core not change for the sorting benchmark. **Most likely the reason was this outlier is that for all other benchmarks, the ‘actual work’ done by the spawned thread is inlined in the function that the thread starts executing immediately.** With the sort, there are two more helpers that are called per thread and moreover, the *partition* function is a recursive function. Thus, the overhead attached with the function calls increases the overall committed instruction count per core. This also suggests that function calls are costly in terms of the instruction overhead that goes with them and hence inlining functions is an important aspect of optimizing code that is often practiced by compilers and good programmers.

In **Figure 10**, we can see a detailed breakup of the instruction cache and data cache miss rates for the sorting benchmark across the single and multicore processors. It is interesting to note that while the instruction cache miss rate falls when going from the single to the multicore design (contrary to the other benchmarks), the data cache miss rate increases. Given that each core has almost an identical number of committed instructions in-spite of sorting only one-fourth of the number of elements as compared to the one core in the single core processor, it is highly likely that the extra cores are either doing empty nops as they wait for the master core (core 0) to merge the threads, or there is the extra overhead of spawning threads or both. Further, it seems like the spawning overhead (and most definitely the nops) don’t have any cache misses.The instruction cache miss rate is also pretty low for the multicore because performing a recursive function means that we’re accessing the same instructions and data over and over again and as one reduces the array size, the benefit from this recursive instruction access could increase. The data cache miss rate seems to be marginally increasing going from the single core to the multicore and this is probably due to the fact that the data cache is also having to handle all the data requests for the parallelizing overhead. Since the data cache is unified and only two way set associative, it is possible that similar parallelizing overhead across the cores is evicting the same lines in the unified data cache.

Finally, we also tried to analyze the performance of the serial benchmarks on the multicore processor to tease out the overhead of the multicore processor when just a single core is used. When a serial algorithm runs on a single core, the other cores don’t sit idle as one would expect. Rather, the other cores are running nops as per the documentation provided. Unsurprisingly, apart from core 0, all other cores have an instruction cache miss rate of 0. It is a little reassuring to see that in spite of no work given to cores 1, 2 and 3, there are data cache accesses and misses for these cores. For example, running quicksort on the multicore and limiting the useful work done to core 0, we observe ~1000 data cache accesses reinstating our belief that indeed, there is some amount of work being done by the cores behind the curtains that we aren't able to account for. We believe that the little discrepancies we observe in our data comparing the single and multicore processors (for eg. data cache miss rate increasing for sort on multicore) could be due to such ‘under the hood’ operations. Lastly, it is highly likely that the parallel sorting benchmark can be further optimized to make better use of the multicore processor (for eg, the helper functions can be inlined) and it is possible that some of the trends observed with that benchmark go the other way if that be done.

Appendix 1

Figure 1: SingleCore Diagram

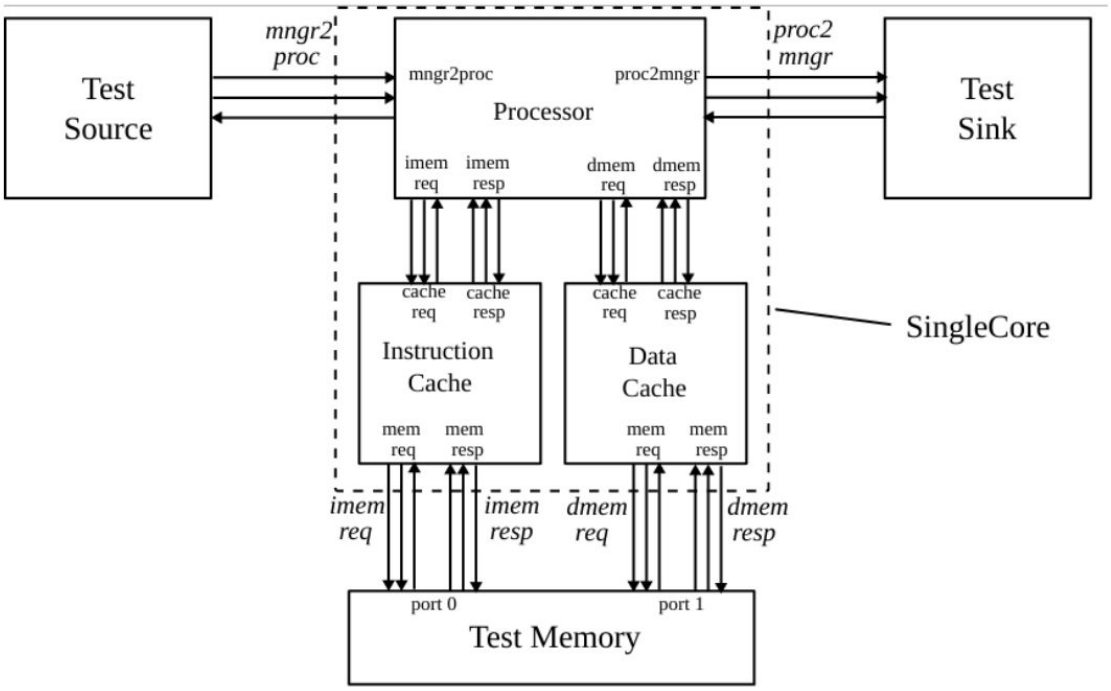
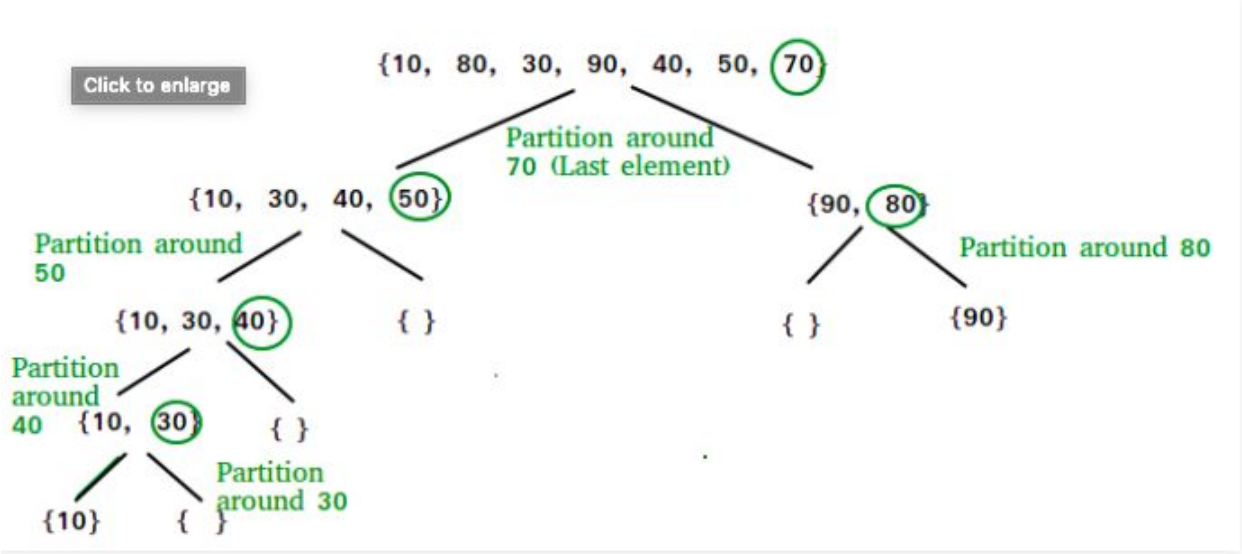


Figure 2: Quicksort Partition Helper Function



Source: https://www.geeksforgeeks.org/quick-sort/

Figure 3: MultiCore Diagram

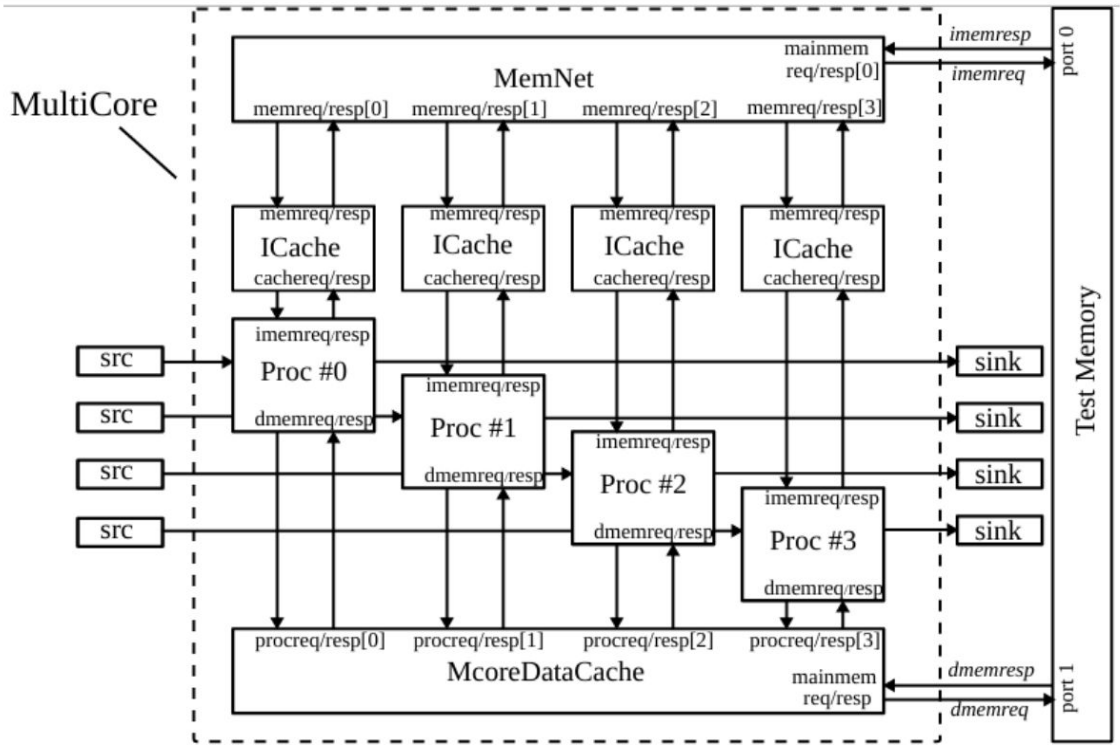


Figure 4: CacheNet Diagram

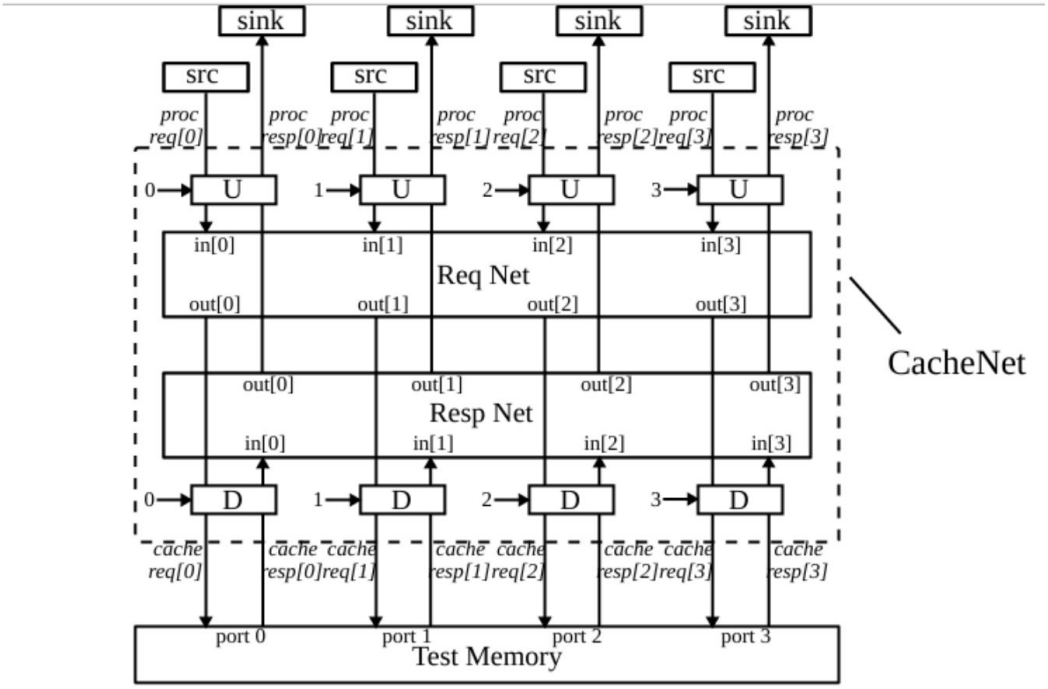


Figure 5: MemNet Diagram

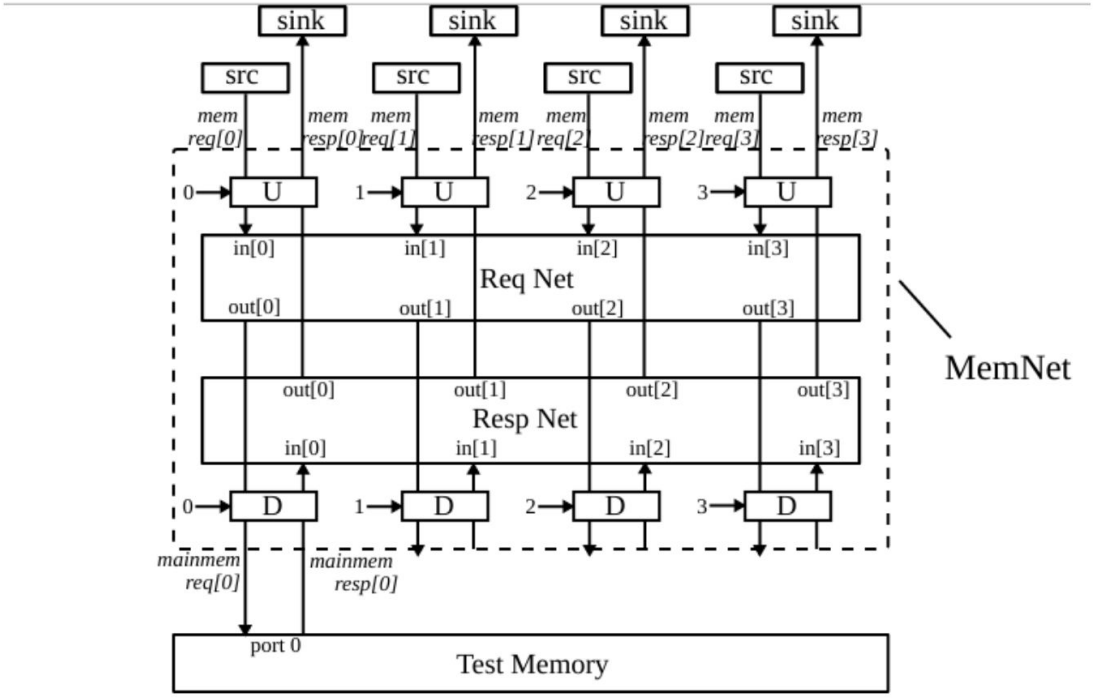


Figure 6: McoreDataCache Diagram

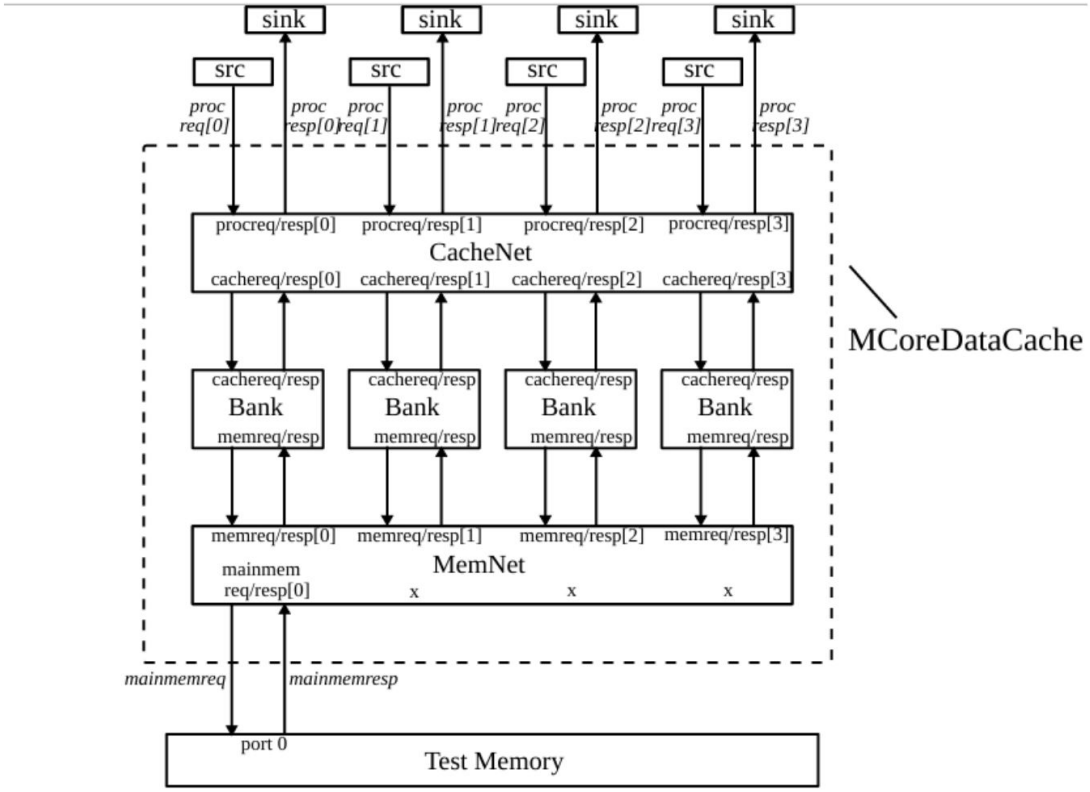


Figure 7: CPI of Different Processors and Speedup for each Algorithm

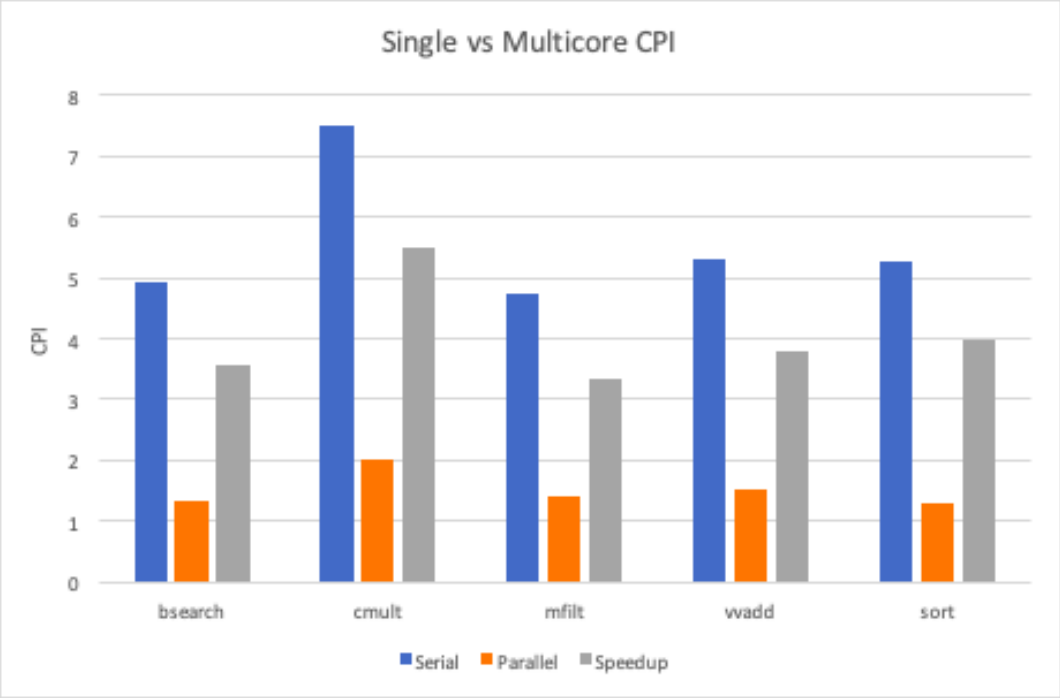


Figure 8: Distribution of Work Amongst Cores

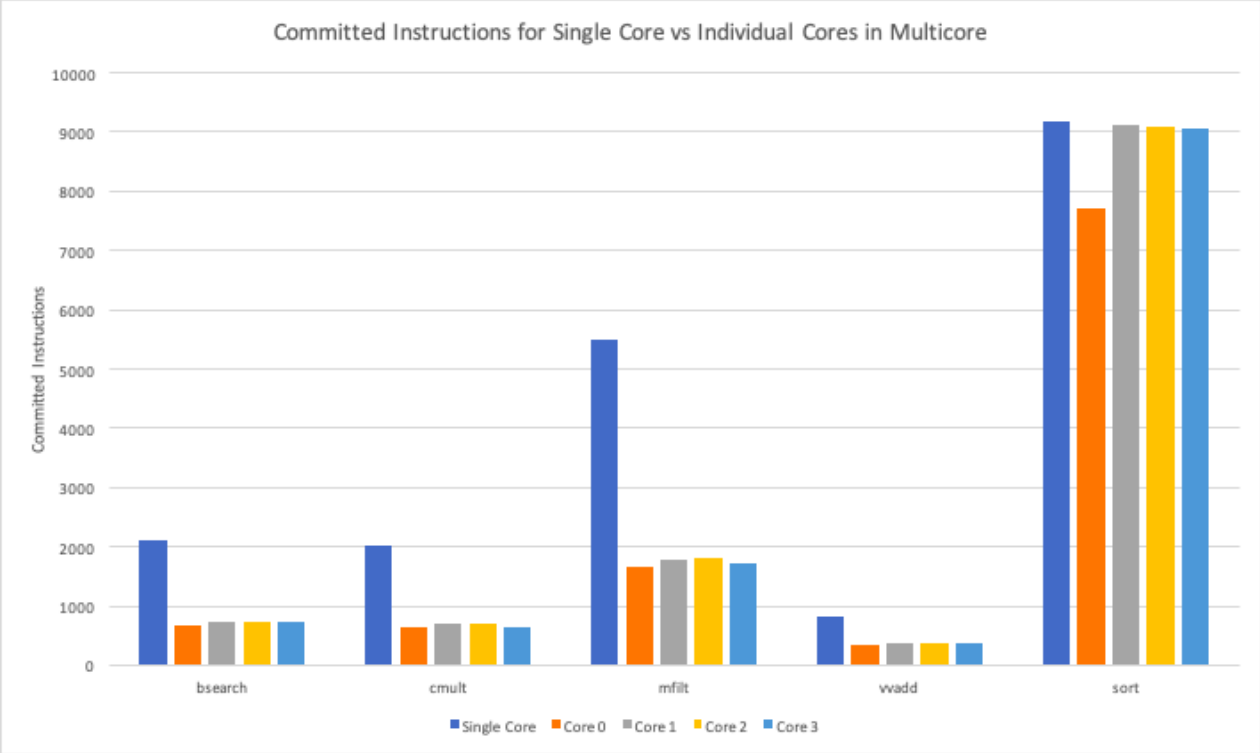


Figure 9: Instruction Cache Miss Rate in Single Core vs. Multicore

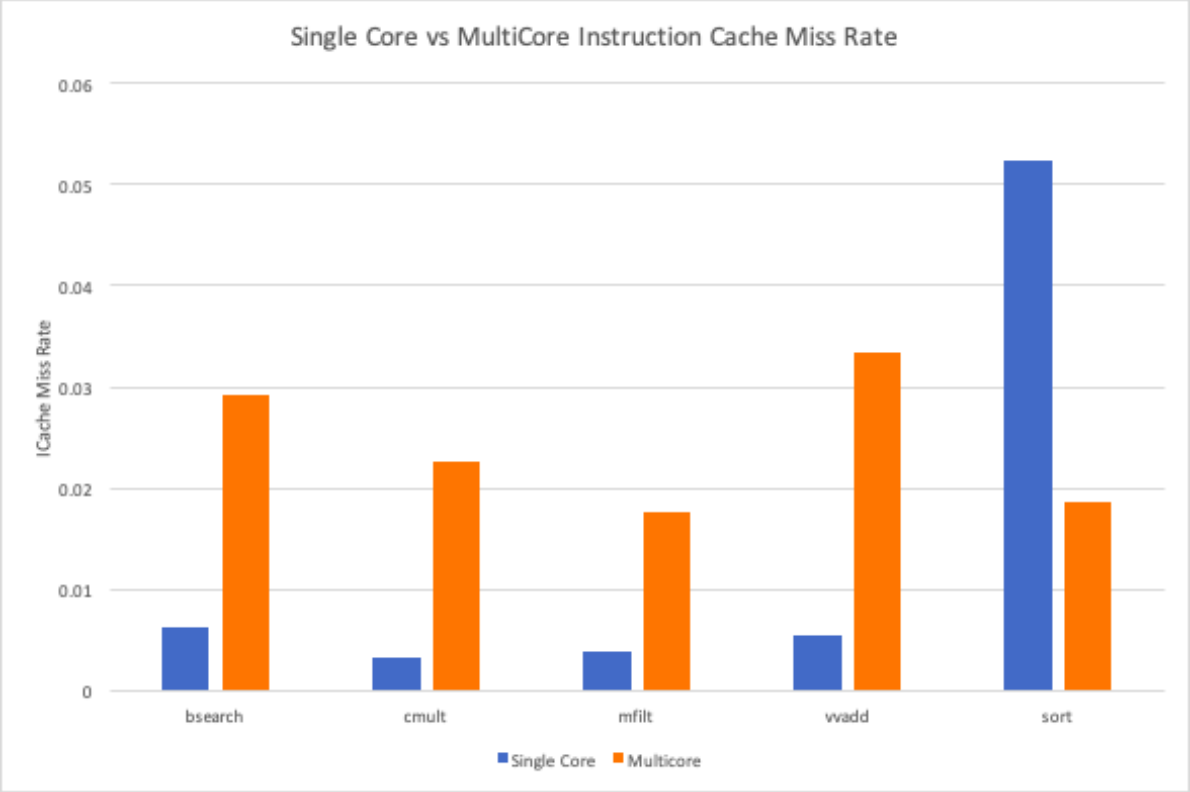
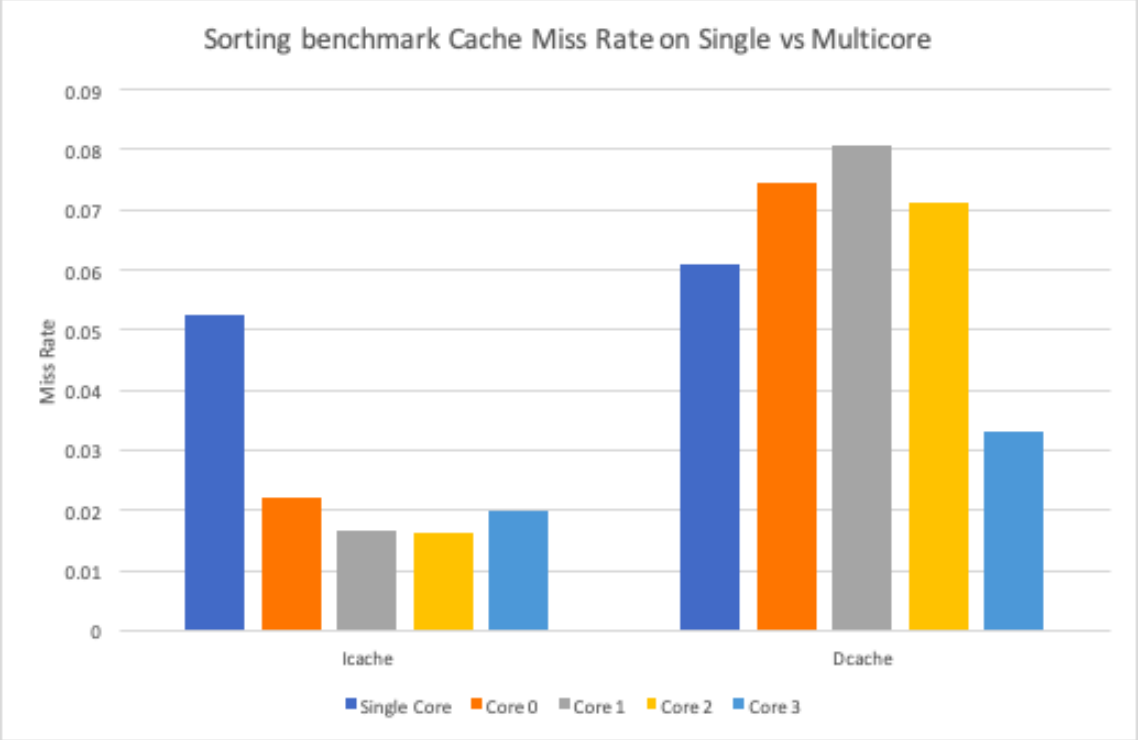


Figure 10: Cache Miss Rates for Sorting Benchmark



Role and Task Table

Carly Swetz	**Raghav Kumar**	**Kyle Betts**
RTL Design Engineer	RTL Design Engineer (architect)	RTL Verification Engineer
● Helped copy over the provided code from the previous Lab 4 ● Ensured that the given baseline design passed all the tests ● Commented out line tracing from Lab 3 that was causing **McoreDataCache** to abort when tested ● Checked that **CacheNet**, **MemNet**, and **McoreDataCache** were passing all tests ● Implemented the final composition of the Multicore Processor System based off of the diagram in **Figure 3** ● Debugged compilation errors in the Alternative Design ● Explained the Single Core system in the Baseline Design section and MultiCore system in the Alternative Design section of the report ● Helped come up with reasons behind the statistics from each benchmark on the two different designs ● Prepared lab report for submission	● Wrote the C quicksort benchmark. ● Wrote the parallel C sort benchmark using the bthread library. ● Tested the quicksort and parallel sort functions to ensure proper functionality. ● Cross compiled and ran evaluations for all benchmarks on the single core and multicore. ● Ran the serial algorithms on the multicore to evaluate the overhead of multicore processors. ● Analyzed the performance of different benchmarks on different single core and multicore. ● Came up with a list of questions and interesting observations to be discussed in the evaluation section of the report. ● Made figures 7, 8, 9, 10 in the lab report. ● Wrote the evaluation section of the report. ● Contributed in the coding part of the Baseline and Alternate sections of the report.	● Helped ensure that the given baseline design passed all the tests ● Examined the CacheNet, MemNet, and McoreDataCache modules and learned their underlying design and implementation ● Helped implement the final composition of the Multicore Processor System based off of the diagram in **Figure 3** ● Helped determine various bit length and bit fields for input and outputs of modules we did not write such as the U and D modules as well as the Bus Net ● Determined the averages and evaluated the results from the baseline algorithm results ● Helped distribute workload for the lab ● Helped debug compilation errors in the Alternative Design ● Completed the two section introduction and testing for the lab report