ECE 4750 Lab 4: Multicore Processor Report

By Morgan Cupp (mmc274), Maria Martucci (mlm423), and Stefen Pegels (sgp62)

Section 1: Introduction

The purpose of this lab was to design and implement both a single core and a multicore microarchitecture, complete with processors and their respective instruction and data caches. These microarchitectures were then used to run two sorting algorithms that we wrote, one with a single thread to be run on one core, and one multi-threaded version to run on our four cores. This lab represents the culmination of our processor and cache designs from previous labs, as we connected how software programs written in a higher level language, C, can be compiled to assembly that our TinyRV2 pipelined, fully bypassed processor can then run with its support for all the instructions required. The introduction of both data and instruction caches increases the microarchitecture's performance, as we can take advantage of spatial and temporal locality for repeated instructions or data from loops by storing them in the cache and limiting walks to the main test memory. This lab connects to lecture material as it is the realization of the incremental design process we have emphasized in class throughout the semester. When learning about new microarchitectures, whether for processors or memory systems, we were introduced to their design by incrementality building up functionality. Through the use of the provided networks and leveraging our cache's support for banking, we were able to finally connect together the pieces of the processor and caches that we have been incrementally designing in each of the previous labs.

We successfully completed both the baseline and alternative designs. The baseline design was largely provided to us, and it consists of a single core as shown in **Figure 1**. The single core module instantiates our fully bypassed processor design, as well as two of our set-associative caches. It served as a model for our alternative design, namely providing an example of the connection of the data cache, instruction memory, and processor's val and rdy signals. We also successfully wrote the single core quicksort benchmark with knowledge from previous programming coursework. The alternative design is shown in **Figure 2**. We followed the provided diagram by instantiating the MemNet and McoreDataCache (four data cache banks shared between all four cores), and using a for loop to generate four processors and their four respective instruction caches. The challenge came in connecting the various msg/val/rdy bundles represented by each arrow in the diagram. We also track the stats of each cache and its hits and misses by leveraging the test (hit/miss) field of the cache response as created in Lab 3. We wrote the parallel sorting benchmark, spawning each core to initially quicksort ¼ of the data, and then having one core combine the results.

The alternative design performed better than the baseline design in terms of total program time, as for the 5 program benchmarks it led to -63.0%, -63.8%, -62.5%, -25.9%, and -47.4% total cycles reductions as compared to the single core. The multicore was able to take advantage of parallel computing to finish these programs faster and reduce the number of data cache misses by taking advantage of spatial locality. The baseline and alternative designs have the same clock frequency, but the multicore design leads to a significant ~x4 area increase. The power consumption would also increase, as the alternative design critical path is longer with need to interface with the MemNet instead of directly interfacing with the test memory. The significant performance increase of 60% on benchmarks with parallel instructions warrants the increase in area and power, as modern processors today are often tasked with running a multitude of programs. This lab placed a new emphasis on the connection between software and hardware and their co-design. We see how software designers with knowledge of microarchitecture can purposefully create threads that each do a fraction of the work required, leveraging the ability of the software to run in parallel on multiple cores and complete faster.

Section 2: Baseline Design

The baseline design was provided to us in this lab and implements the block diagram shown in **Figure 1**. The design is composed into our processor and cache modules with well-defined interfaces, and this is an example of modularity. Each module also contains separate control and datapath modules and even more within for the smallest components. This recursive modularity is an example of hierarchy. The processor-cache interface hides each module's implementation details; only the request and response values matter. This is an example of encapsulation. These modules also make use of the val/ready interface design pattern to aid in encapsulation by making a latency-insensitive interface, further reducing the need for implementation-specific details. Lastly, the baseline takes advantage of regularity since it instantiates the same cache module twice to create one instruction cache and one data cache. This common cache structure made it simple to produce two caches with different purposes, but the same internal functionality.

The baseline design works by simply connecting modules we made in previous labs. Our fully-bypassed, single-core processor is connected to two of our set-associative caches: one for instructions, and one for data. As the processor executes instructions, it sends its memory requests to the appropriate caches. The caches bring requested cache lines in from memory to exploit temporal and spatial locality and send responses to the processor. The caches are in turn connected to the test memory which acts as a main memory. Like before, the processor is also connected to the test source and sink. The source initializes values in registers, and the sink verifies that specific registers contain their expected values. These modules work together to create a single-core system.

The baseline was provided to us and we did not change it. It is good for comparison due to its simplicity and reliable functionality. It is single-core which significantly reduces the hardware complexity compared to multicore, yet it has the hardware necessary to execute most programs. Thus, it sets a standard for accuracy and critical path length that subsequent improvements should try to preserve. However, the baseline design leaves room for improvement and provides a foundation that can be built upon.

This lab also contains an important software component. For the baseline design, we implement a scalar quicksort algorithm in C. The goal of quicksort is to sort an array of numbers from lowest to highest. Quicksort has two steps. First, as shown in **Figure 7**, a pivot value is selected from the input array, and the values in the array are rearranged such that values less than the pivot are at lower indices, and greater values are at higher indices. These lesser and greater values are not sorted. Next, quicksort is called recursively on the lesser and greater values. These subarrays will then be partitioned, and the pattern continues. The resulting algorithm is shown in **Figure 8**. The recursion continues until there are no subarrays remaining that are large enough to be partitioned.

The baseline quicksort implementation was fairly straightforward and did not require much creativity since it is a fundamental, common algorithm. This is especially true since the baseline design runs quicksort on a single core. This means the core has to do the entirety of the sorting and there is no notion of parallelizing any work. This was not the case for the alternative design.

Section 3: Alternative Design

The alternative design is a multicore system consisting of four processors, four private instruction caches, four shared data cache banks, and four-port networks that connect the caches to a dual-port test memory. Each processor is also integrated with its own test source and sink for verification purposes. The multicore system's high-level block diagram is shown in **Figure 9**. The multicore system exhibits the same four design principles as the baseline for the same reasons and more. First, the multicore system has more modularity due to more processor and cache instantiations. The MemNet and McoreDataCache are also both their own modules, and the McoreDataCache consists of four caches in a four-bank organization. This recursive modularity in McoreDataCache demonstrates the hierarchy principle. We were able to integrate the provided MemNet and McoreDataCache by just knowing the interface and not the implementation; this is a clear instance of encapsulation. Lastly, the multicore system consists of four cores configured in a very similar way to our single-core system. This repetition of a common structure is an example of regularity.

Figure 2 provides a more detailed block diagram of the multicore system. Our lab 2 processor and lab 3 cache were each instantiated four times and connected in a manner similar to the baseline design. The instruction caches for each processor are private, because we did not want one core to have access to another core's instructions. **Figure 2** shows that the instruction caches achieve privacy by not being banked and connecting to exactly one core. Each core is assigned a unique core ID that provides software with a way to divide work across the cores. Each cache and core functions the same way, so we could easily distribute work across cores, since they all have equal capabilities. We used the fully-bypassed processors since bypassing drastically reduces stalling and improves performance. We chose to use the set-associative caches to reduce conflict misses. We thought reducing conflict misses was more important than the speedup of a direct-mapped cache, because our programs have temporal locality and cache misses are very costly. With associativity, it is more likely that previously accessed values will not get overwritten before their next access, causing a hit.

The McoreDataCache and MemNet were provided to us, and each one was instantiated once. The McoreDataCache, as seen in **Figure 10**, consists of a CacheNet, a four-banked cache organization, and a MemNet. The CacheNet is a network for handling requests and responses between the four processors and four data cache banks. The MemNet is a network for handling requests and responses between the four data cache banks and single test memory port. It is key that the MemNet correctly handles the four different memory request/response pairs from different caches while allowing them to occur one at a time due to the single test memory port. Lastly, we instantiated four of our set-associative caches, this time in a banked cache organization. We used a banked cache organization so that multiple cache requests may occur at once, improving performance by reducing stalling. Since the processors must be able to access all of the data when doing parallel computation, they do not use private data caches. McoreDataCache again uses the set-associative cache to increase performance by reducing conflict misses. A separate MemNet was instantiated to connect the instruction caches to the test memory, and its functionality is essentially the same as the McoreDataCache.

The software portion of this section required implementing a parallel sorting program. Each core has its unique core ID, and the program knows there are four cores. We implemented the hybrid mergesort/quicksort algorithm suggested in the lab handout. **Figure 11** shows this algorithm. First, the unsorted input array is divided into four blocks. Each block is assigned to one core. Each core uses the quicksort algorithm from the baseline design to sort its section. There are then four sorted sections, and core 0 merges them into one sorted array. This is done by calling a merge function three times that can merge two sorted arrays into a single sorted array (see **Figure 11**). The merge is done using the merge algorithm of mergesort. It copies one sorted array into a temporary array, and then copies its values back into the original input array from smallest to largest while also considering the second sorted array.

This algorithm allows the input data to easily be divided evenly, meaning each core does a similar amount of work. This minimizes time that any one core waits for others to finish, and maximizes parallel computation time. The overhead of dividing the data is also very small, since all we have to do is calculate the indices within which each core will do a quicksort. One drawback is that core 0 does all of the merging at the end. This could hinder performance for very large input arrays. A future improvement would thus be to find a way to parallelize the merging as well. Additionally, cores one through three do nothing while core zero sets up the parallel computation. Perhaps during these cycles the cores could be made to do something productive, such as pre-fetching input data values from memory. Nevertheless, the relative simplicity of this first implementation gives a reliable foundation for future changes.

Section 4: Testing Strategy

Our testing strategy leveraged the thorough testing of our subsystems and focused instead on testing the combination of all our incremental designs together. Black box testing was performed in the testing of the processor and caches in previous labs by running a multitude of instructions with different data values, and we performed random testing to remove our bias from the operands selected. The provided test cases for the single and multicore system exhibited white box testing, where representative instructions were chosen to verify the connection of the modules. We wrote two sorting benchmarks which tested the compilation of real high level software programs down to TinyRV2 instruction tests for our single and multi-core systems. We tested the benchmarks themselves through directed testing of different data sets aimed at exposing corner cases of the algorithms. A quantitative summary of our test results can be found in **Figure 3.**

We have been incrementally testing parts of our final design throughout the labs this semester. In each of the previous labs, we first wrote test cases on the Functional Level Model to verify their expected behavior. The iterative multiplier was tested with directed black box testing and random operands to verify normal and corner case functionality such as multiplication by zero. We performed white box testing by masking off bits to zero, to expose the performance improvement of the alternative design in skipping cycles. The Lab 2 processor received test data through the source and updated register values to then be verified by the sink. Each TinyRV2 instruction had its own test file containing both directed and random value unit tests. We also tested with random source and sink delays to verify the processor's val and rdy interface, used to stall the pipeline when the test memory is not ready to accept the processor's request or when the memory takes a variable amount of cycles to return a value. We also performed basic and mixed instruction testing with assembly programs, to ensure that a sequence of instructions can properly fill the pipeline. The instruction and data caches were tested in Lab 3. We performed white box testing by writing tests for the write hit/miss and read hit/miss paths to ensure that all paths of the FSM were functional. We wrote a design-for-test init instruction that filled the cache with test data so we could properly test the cache hit path without the need for a functional write miss path, which involves a walk to memory. We performed all of our directed tests with different numbers of banks, to ensure correct functionality of the banked cache organization required in the multicore system.

By our confidence in the above testing strategy for each element that composed our single core design, we were able to perform tests specifically aimed at the connection of these elements together without worrying about their individual functionality. The provided single core test chose a representative test from each instruction category (e.g. add for register-register). The multitude of directed tests for these instructions, for example with dependencies between the registers, as well as the random value testing and random source and sink delays, are all organized by the Test/SimHarness. The harness instantiates the TestMemory models for both the instruction memory and the data memory, so that the interaction between the processor, instruction, and data caches is simulated along with source and sink delays from the memory to verify all the val/rdy interfaces seen in **Figure 1** as arrows.

The alternative design MultiCore system was similarly tested by the above set of representative instructions to ensure the proper connections between processors and caches. The harness takes as input the number of cores, so it correctly connects four different test sources to each processor and from each processor to its own test sink. The sw and csr instructions were not tested, because if each core writes to the same address, then we would not be able to tell which core finished their instruction first and thus what value is correct to expect. For csr tests with the four different sources and sinks, the tests had to differentiate between which core's source to initialize using the core ID. Continuing in the incremental testing strategy, the CacheNet, McoreDataCache, and MemNet also had their own test files. The CacheNet and McoreDataCache reused the tested cache requests from Lab 3, along with performing banked and random message testing to try to expose corner cases in the designs. The MemNet used memory request messages from the TestMemory tests, as well as random messages to read and write to the main test memory.

We also developed a testing strategy for the single and multithreaded sorting microbenchmarks. We tried to expose corner cases in the single thread quick sort algorithm by modifying the test data file to include arrays that were for example already sorted or had all zeros and a single one at the front, that would require a lot of swaps through the recursive calls to bring it to the front. We similarly made new data sets for the parallel sorting benchmark and paid particular attention to the size of the array, verifying that if the array size was less than the 4 cores that the work was supposed to be split on, the indexing still allowed for proper sorting.

We are confident in the correctness of our single and multicore systems. We incrementally built compositions in each of the previous labs, united tested them extensively, and then reused them to build the larger components of the entire system. This testing included a multitude of directed tests for each instruction type as well as each memory interaction with the cache such as hit or miss. The composition of the processors and caches for both the single and the multicore was verified to run the instructions through directed tests and random value tests, as well as random source and sink delays which highlight the critical interactions between the msg/val/rdy bundle arrows between the banked data cache, processor, instruction memories, memory network, and test memory. The benchmarks for the single and multicore processors all passed, using a mixed set of instructions over thousands of cycles to perform various programs like vector-vector add and binary search. These programs required the utmost exactness of each of our subsystems to achieve the correct final result, solidifying our confidence of correct functionality.

Section 5: Evaluation

In evaluating the performance of our base and alternative designs, total program time, represented by cycles, was the primary consideration. In **Figure 4**, the multicore design was shown to be effective at reducing cycles on all benchmarks, with diminishing returns on benchmarks that involved more operations that utilized shared resources/communication between cores. The multicore system also took advantage of spatial locality, where present, to reduce data cache misses (**Figure 5**). The largest improvements came in the bsearch, cmult, and mfilt benchmarks (-63.0, -63.8, -62.5% total cycles, respectively) due to parallelization of instructions allowing for speedup with less work for an individual core. CPI decrease was also significant for the multicore system(4.92 vs 1.34 for bsearch), as the multicore system was able to provide such a drastic reduction in cycles while also handling more instructions (33% more for bsearch). The percent change in cycles from the base model for the other benchmarks are given in **Figure 4**.

Conversely, the multicore system resulted in a large increase in instruction cache miss rate, even among benchmarks that had a significant improvement in data cache miss rate. For bsearch, icache miss rate shot from 0.0063 to 0.0292, a 363% increase (dcache miss rate decreased from 0.6318 to 0.3369). This icache miss rate increased due to the multicore system beginning execution on four cores instead of one, meaning initial icache misses. This increase was not significant to total execution time, because of the small icache miss rate and the very substantial dcache miss improvement, due to spatial locality used by the individual cores. These cache comparisons can be found in **Figure 5**. The other drawback to consider for the multicore system was its performance when running single core applications. In **Figure 4,** for single core benchmarks run on the multicore system, total cycles increased by an average of 12.8%. This increase is due to the increased hardware present in the multicore system, and the more complicated memory transactions that are moving between the multi banked data cache systems present in the multicore system compared to the simpler single-core approach. Also, when the other 3 cores are not in use, they are sent repeated lines of NOPs, which account for the increase in instructions compared to the single core. The choice to implement the multicore system will then depend on the specific software; if the code can be parallelized, the multicore benefit is immense, but if not, extra overhead instruction costs will hurt performance. This again reinforces the lesson that software handling is important in making hardware decision choices; in other words, the way memory is accessed in the software will dictate whether single core or multicore is more efficient.

An important evaluation of the multicore system is the individual behavior of the constituent cores, and how they contribute to the overall performance of the system. In **Figure 6**, a comparison of the full multicore system versus one individual core is shown(the other 3 cores had similar statistics, and were thus omitted for clear data readability). The icache and dcache miss rates were nearly identical to that of the overall system, and the core for bsearch ran 730 instructions compared to the overall 2855(25.5% of total). For the bsearch benchmark, Core 2 had a CPI of 5.25, compared to the 1.34 CPI of the entire system. From this we can evaluate that Core 2 behaved similarly in its execution to the single core system (4.94 CPI), and the strength of the multicore system comes from when it can cut the data into quarters and run four single core processors at once. This supports our argument of the strength of parallelism in computing on large datasets; it allows the hardware to speed up by dividing data and using all its resources.

Compared to the single core, the multicore system adds three more processors, icaches, and dcaches organized in the McoreDataCache bank system, as well as the MemNet entirely(**Figure 1 and Figure 2)**. This results in a significant increase in the processor area. The alternative design did not remove components from the base design; so the chip size would only increase. The manufacturing size of the chip would most likely have to be more than 4x larger, indicating increased cost, but the performance increase of about 60% for most benchmarks warrants this increase in area.

The energy consumption of a processor is related to the amount of heat generated moving through the circuit in a given clock cycle, as well as proportional to the clock frequency. Since both the base and alternative designs have the same clock frequency, the energy difference can be attributed to the change in the critical path of the alternative design, which would produce the same amount of heat given an equivalent critical path. The alternative design has a longer critical path, so its energy consumption would be greater than that of the baseline design. In the alternative design, the processors have to interface with the MemNet rather than having the icache directly connect to test memory, lengthening the critical path. The overall energy consumption can also be attested to the number of cycles for each implementation, which sways heavily favoring the multicore system, given the parallelism in play. Thus, single cycle energy is higher in the multicore, but total program energy use is considerably less.

Cycle time can be evaluated by looking at the critical path for both designs(see critical path discussion above). The associative cache has a longer critical path, and thus a greater cycle time. Thus for instructions interacting with the test memory, the multicore system will have slightly longer cycles, but not enough to negate the CPI increase.

While exploring parallel computing can affect speed, learning about the organization of software iterations and loop patterns that highlight specific parallel optimizations showed how software decision-making can impact hardware performance. Implementing the multicore system from scratch showed us how the modules we have been building come together and interact to create machine code out of C programs.This provided valuable insight into real life processing units are constructed, and how they take what is on the screen and decipher it into a series of messages that the computer's components can interpret.

Index

Figure 1: Baseline Design, Single Core

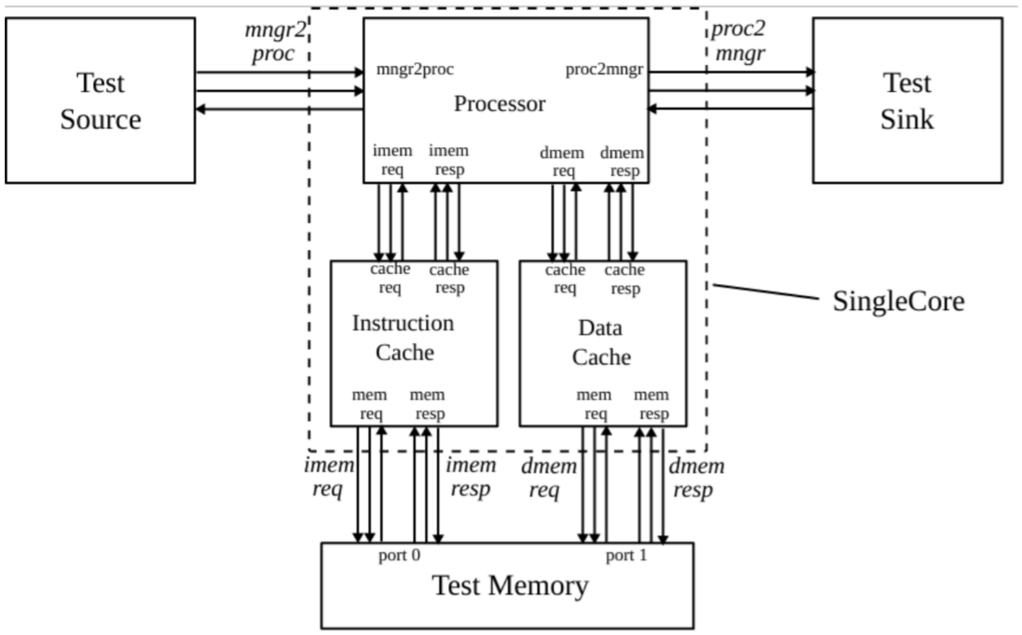


Figure 2: Alternative Design, Multicore

Note: proc2mngr and mngr2proc port names are omitted, and each arrow represents the msg, val, and rdy signal for that connection

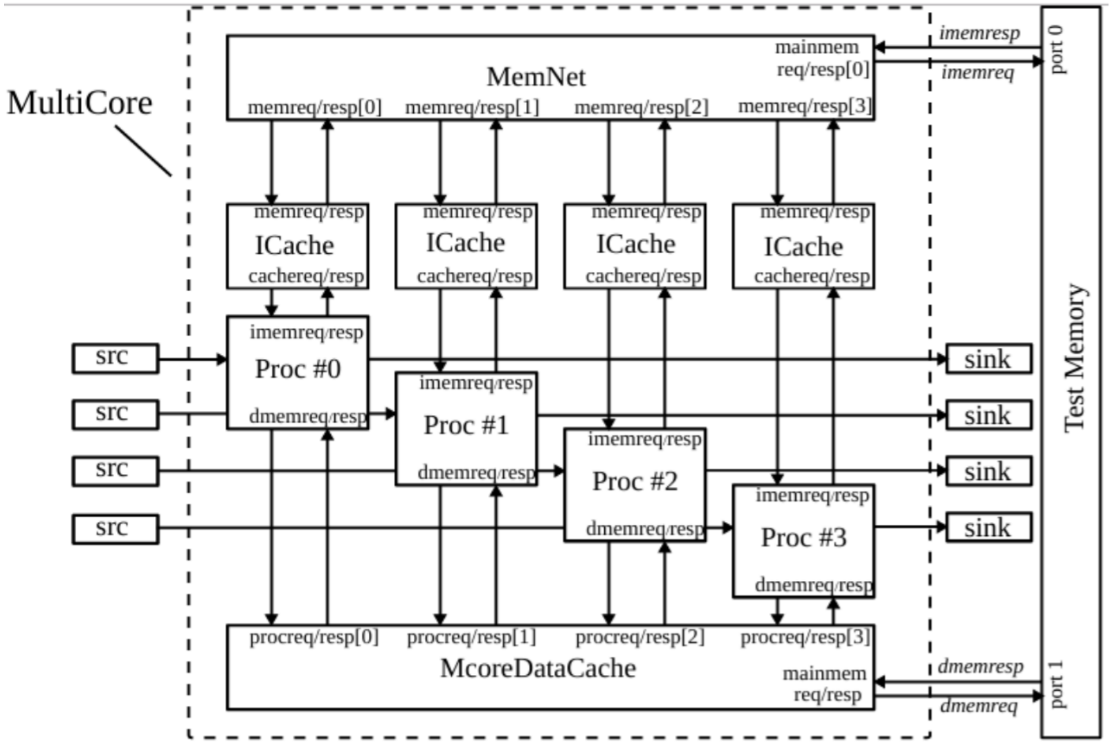


Figure 3: Table summarizing testing strategy

Test Script name	Number of unit test functions passed
CacheNetRTL_test.py	4
McoreDataCacheRTL_test.py	4
MemNetRTL_test.py	4
MultiCoreRTL_test.py	63
MultiCoreRTL_test.py	68

Successful datasets run on both single-thread quicksort and parallel multithreaded sort

Random value array of size 1 Random value array of size 2 Random value array of size 3 Random value array of size 4
Array of all negative numbers
Array of already sorted numbers
Array of many zeros followed by a single 1
Large array of many random values
Random odd sized array
Random even sized array

Figure 4: Comparison of Cycles, Instructions, and CPI Among Benchmarks, Their Multicore Versions, and Their Single Core Versions Run on the Multicore System

Bench mark	Cycles	Instrs	Total CPI	Cycles	Instrs	Total CPI	Cycle Change	Cycles	Instrs	Total CPI	Cycle Change
bsearch	10355	2106	4.92	3831	2855	1.34	**-63.0%**	10883	6188	1.76	**+5.1%**
cmult	13874	2011	6.90	5302	2702	1.96	**-63.8%**	16231	8098	2.00	**+17.0%**
mfilt	26034	5493	4.74	9770	6987	1.40	**-62.5%**	29675	16621	1.79	**+14.0%**
qsort	63336	12678	5.00	46963	36250	1.30	**-25.9%**	71755	39587	1.81	**+13.3%**
vvadd	4296	811	5.30	2260	1492	1.51	**-47.4%**	4923	2656	1.85	**+14.6%**

Key: Single Core, Multi-Core, Single Core Benchmark on Multi Framework

Figure 5: Overall Comparison of ICache and DCache Behavior of Single Core and Multi-Core Benchmarks

Benchmark	Icache miss rate	Dcache miss rate	Icache miss rate	Dcache miss rate	Icache Change	Dcache Change
bsearch	0.0063	0.6318	0.0292	0.3369	**+363.49%**	**-46.68%**
cmult	0.0033	0.1510	0.0224	0.1515	**+578.79%**	**+0.33%**
mfilt	0.0039	0.2561	0.0177	0.2319	**+353.85%**	**-9.45%**
qsort	0.0448	0.0401	0.0182	0.0402	**-59.38%**	**+0.25%**
vvadd	0.0066	0.2533	0.0334	0.2181	**+406.06%**	**-13.90%**

Key: Single Core, Multi-Core

Figure 6: Comparison of Multi-Core Individual Core vs Overall

Benchmk	Total				Core2			
	Instr	CPI	Icache Miss Rate	Dcache Miss Rate	Instr	CPI	Icache Miss Rate	Dcache Miss Rate
bsearch	2855	1.34	0.0292	0.3369	730	5.25	0.0244	0.4074
cmult	2702	1.96	0.0224	0.1515	695	7.63	0.0150	0.1617
mfilt	6987	1.40	0.0177	0.2319	1809	5.40	0.0155	0.2652
qsort	46963	1.30	0.0182	0.0402	9493	4.95	0.0148	0.0226
vvadd	1492	1.51	0.0334	0.2181	385	5.87	0.0236	0.2471

Figure 7: Partition step of quicksort

Source: CS 2110 lecture slides on sorting

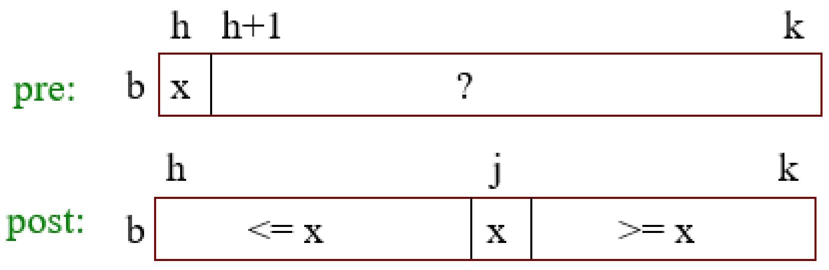


Figure 8: Full quicksort algorithm

Source: https://gaebster.ch/quicksort/

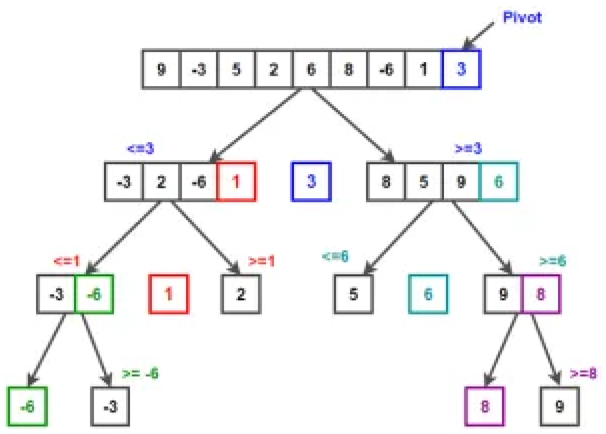


Figure 9: Alternative design block diagram

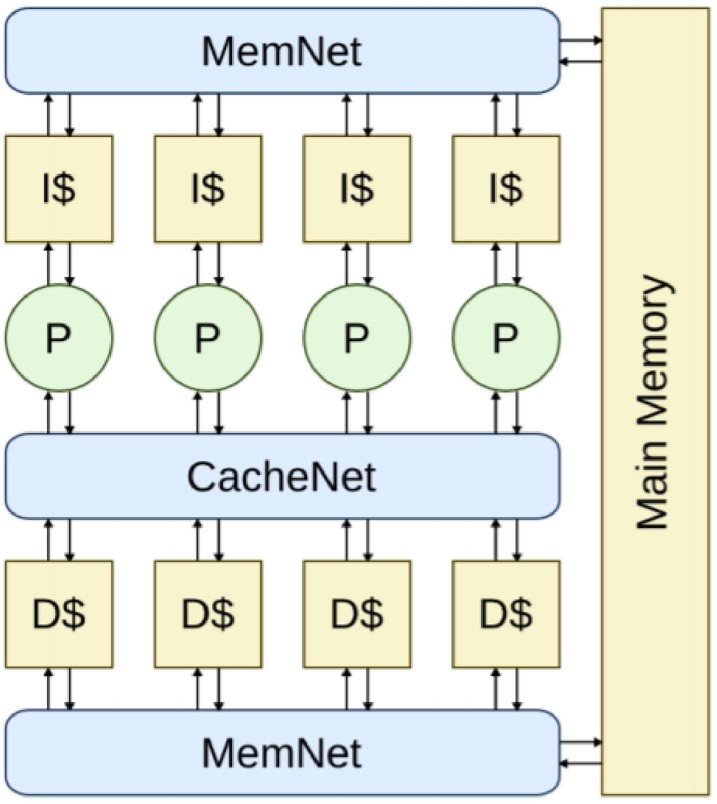


Figure 10: McoreDataCache block diagram

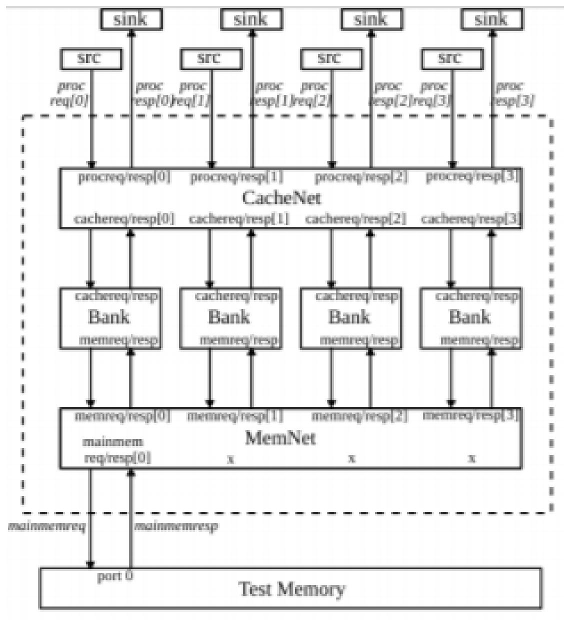
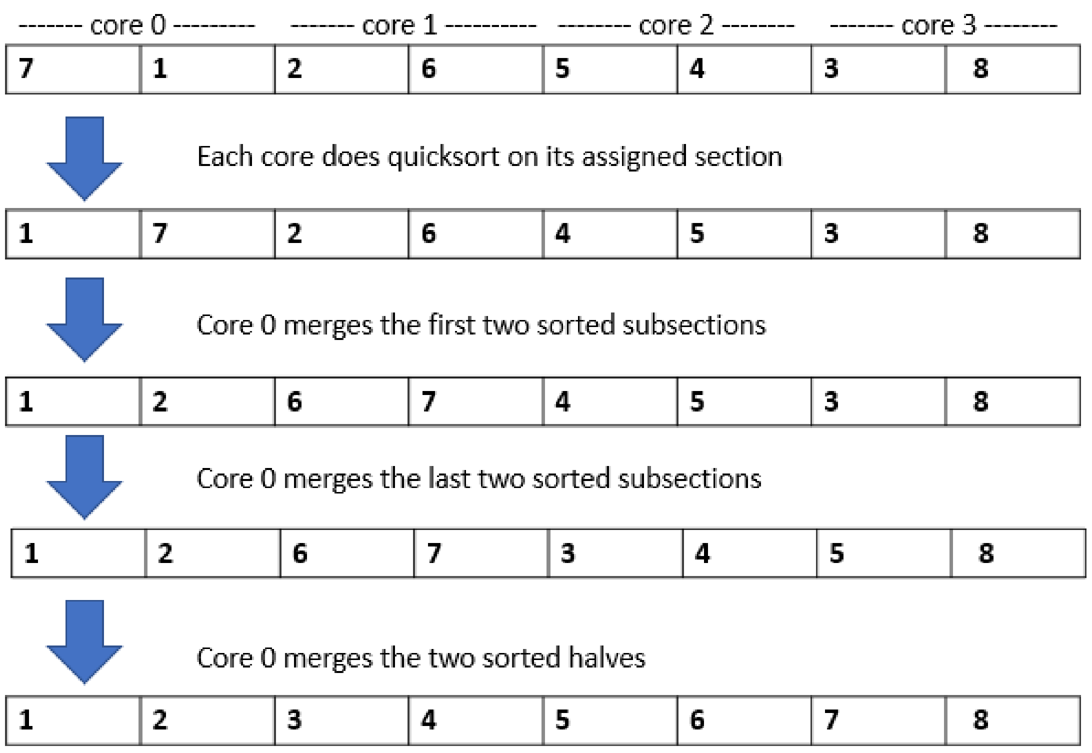


Figure 11: Hybrid mergesort/quicksort parallel sorting example



7	1	2	6	5	4	3	8

Each core does quicksort on its assigned section

1	7	2	6	4	5	3	8

Core 0 merges the first two sorted subsections

1	2	6	7	4	5	3	8

Core 0 merges the last two sorted subsections

1	2	6	7	3	4	5	8

Core 0 merges the two sorted halves

1	2	3	4	5	6	7	8

Role and Task Table

Morgan Cupp	Maria Martucci	Stefen Pegels
RTL Design Engineer (architect)	RTL Verification Engineer	RTL Verification Engineer
Wrote the Baseline and Alternative Design sections of this lab report. Established some progress goals for when we wanted to have different sections completed. Studied the alternative design and gave overarching explanation to group members. Helped figure out corner cases to test parallel sort. Made one corner case where 1 starts on one end of the array and must be moved to the opposite end during sorting. Suggested key statistics to analyze in the evaluation section to assess each design's performance and compare them to each other. Helped coordinate meeting times. Studied baseline code to make sure everyone understood it. Wrote the code for MultiCoreVRTL.v with some supervision and debugging help from Maria and Stefen. Found a bug where the core_id was not being assigned correctly and fixed several port connections. Discussed how quicksort works and supervised as it was implemented. Helped write the parallel sorting algorithm by making sure our implementation was following the provided vvadd example. Helped debug the parallel sorting algorithm by placing print statements throughout the code to isolate where the error was (turned out to be the --mcore issue). Went to office hours with Maria to figure out the val/rdy issue.	Wrote the Introduction and Testing Sections of this lab report. Went to office hours to increase understanding of the connection of the MemNet and the test/main memory. We were initially very confused why only the mainmemreq[0] was sent instead of all 4 positions. Went to office hours to talk about the val ready interface not working. Through running the provided test cases we saw we were in an infinite loop, signally that we weren't advancing correctly with the val/rdy. Realized we had them flipped with respect to mainmemreq and imemerq. Implemented the software testing strategy for the parallel sorting benchmark. Utilized a series of print statements to step through the code, and see where the references to the source and destination array were not being passed corrected. Also discovered that some of the array was being overwritten to zeros through these statements, and realized we were copying a blank array over to the destination in some places. Figured out a bug in the parallel sorting algorithm - we were stuck for a very long time because it seemed as if the threads we spawned were not doing their ¼ of the work. Realized that we were running the wrong command on the lab handout and didn't add the --mcore! Worked together to write the parallel sorting benchmark, following the threading set up of the other multicore benchmarks like cmult. Used the same quicksort as for the single core and then merged them together through reviewing CS2110 slides. Worked together to write the quicksort algorithm for the single core, using knowledge from previous CS2110 classwork.	Wrote the Evaluation section of this lab report. Contributed to MultiCore datapath design by creating wires and connecting submodules to form the multicore system. Debugged improper assignments of bit width on test memory access signals. Figured out the connections of val/rdy signals between memnet, mcoredatacache, and each of the processors and icaches. Debugged initial problems finding correct directories to compile and test code. Figured out how to correctly use the makefile to run each of the simulations to load onto the multicore system. Worked to debug issue where initial multicore design was reaching its maximum cycle limit and not finishing execution. Included investigation of processor states and communication between cache and processors for refill requests and cache responses. Worked to debug multicore quicksort that seemed to be only affecting one core instead of all four, including multiple code refactors to make use of pointers in different ways. Added functionality to single core test cases to enable that they be tested on the multicore system, to compare drawback of the extra hardware for the same operation. Ran/tested all the benchmarks to obtain evaluation data and draw performance conclusions.