ECE 4750 - Lab 4 Report

Sabrina Herman (sh997), Yoon Jae Oh (yo82), Benjamin Xing (byx2)

Section 1: Introduction

The purpose of this lab was to gain experience with topics taught in class such as software/hardware co-design, programming single- and multi-threaded C programs, and computer architecture evaluation methodologies based on architecture-level statistics through designing single and multicore processors. The lab incorporates many important themes in computer architecture like memory networks and incremental design. Unique features of this lab included its emphasis on structural composition to incrementally create a relatively complex system based on thoroughly unit-tested subsystems, software/hardware co-design such that students must understand the software application, hardware/software interface, and hardware microarchitecture. These computer architecture themes are relevant not only to this class, but to how computer architecture, design, and verification are implemented in industry.

In the baseline design, we were given a single-core processor with its own instruction and data cache. In the alternative design, we designed a multi-core processor with private instruction caches and a shared, banked data cache. We also wrote both a single-threaded and multithreaded sorting microbenchmark in C, explored the compiled and assembled binary, and ran these programs on the two designs. The evaluation of the two designs (discussed more thoroughly in section 5) revealed that the alternative design saves significantly on CPI (cycles per instruction) for the multi-threaded programs. For each of the evaluation tests, the alternative had a smaller CPI than the baseline, for example the multicore had a CPI of 1.51 for the multi-threaded vvadd test whereas the single core had a CPI of 5.32 for the same test. This result is mirrored by all the other multi-threaded evaluations that were conducted, demonstrating how the alternative design can exploit thread-level parallelism to increase throughput and lower the CPI.

The alternative design is more compelling than the baseline design due to its ability to handle more than one thread simultaneously. Multiple cores means that each core can handle a separate stream of data, which leads to significant performance increases for a system running multiple concurrent applications. Multicore processors exploit thread-level parallelism by allocating different threads within a single program to the different cores within the chip. This ideally will fully parallelize an application on P processors and will result in about P× speedup. However, multicore designs also have their drawbacks. First of all, it is not possible to fully parallelize most applications, and serial portions of the code can quickly dominate the execution time. Implementing multiple cores also leads to increases in hardware area and energy requirements. However, the performance increases that they provide make them well-worth the tradeoff, as previously discussed with the CPI. With regards to energy use, although the multicore processor requires more energy overall due to its running of multiple threads and additional hardware, it could be argued that it actually saves energy because it allocates less energy for an individual, single thread in the end than a single-core processor does for that same thread.

In general, increasing the number of cores will significantly increase performance in systems running multiple concurrent threads. However, this is not always the case and is often unnecessary. For the most part, a dual or quad-core processor should be enough to support a user's needs for a basic computer. This is due to software limitations. With increases in cores, software adaptations are necessary to support the addition of more cores. Therefore, specialized software is usually needed for very high-core processors which is why high-core processors are usually machines that perform very complex tasks.

Section 2: Baseline Design

The baseline design, shown in Figure 1, is a single core, pipelined processor with full bypassing composed with an instruction cache and an unbanked data cache. The design does not use any networks because there are only a few modules to connect together, and all memory data must flow through a single processor. It is also relatively simple to handle the val/rdy interfaces between the instruction cache and processor and data cache and processor. The baseline design uses the alternative design from lab 2 in order to reduce stalls through bypassing. Bypassing significantly increases the throughput of the processor while requiring only a small increase in hardware area and power requirements. It also uses the alternative design from lab 3 for the instruction and data caches, which has two-way set-associativity to better exploit temporal locality. The number of banks is set to 0 because the cache does not have to decide which core to send data to - there is only one core in the baseline. The cache miss/access statistics ports and processor statistics ports are connected to outputs of the lab 5 module, and the performance statistics are dumped during the program execution. This design is a good baseline design because all the instructions must execute on only a single core, meaning that it will be very easy to compare and quantify the performance improvements of parallelization after the addition of more cores.

We implemented a scalar quicksort algorithm for sorting an array of integers. Quicksort chooses any element in the array to serve as the pivot, and then partitions the remaining elements of the array on either side of the pivot based on whether they are smaller or larger than the pivot value (smaller on left, larger on right). It then recursively executes quicksort on the arrays to either side of the pivot until all elements in the array have been sorted. While any element in the array can be the pivot, usually it is either the first element, the last element, or the middle element. For simplicity, we choose the last element in the array as the pivot position every time we call the quicksort function. We use the quicksort algorithm because it is very efficient on average with O(n log n)

performance, although in the worst case it can take O(n^2) comparisons. Moreover, it is space efficient as it can be sorted in-place. Quicksort is a good baseline because it has good cache spatial locality compared to other sorting algorithms, namely merge sort which divides the array to sort it. We reuse quicksort in our merge sort algorithm in alternative design to sort the array portions. Therefore, quicksort makes a good baseline design because it can be reused in the merge sort, and we can more easily evaluate the benefits and drawbacks that come from dividing the array among multiple cores.

This design (and the alternative as well) demonstrates hierarchy due to its memory structure. There exists a memory hierarchy from the caches, main memory, and other memory levels that might exist. A hierarchy means there are different levels of memory in which going up levels means increased speed and bandwidth and going down levels means increased capacity. The cache is at a higher level than the main memory and it takes less time to access data within it while having a lower capacity then memory levels below it. The main memory is at a lower level than the caches and it has the ability to hold much more data than a cache but it takes much longer to access its data than the levels above it. The design also demonstrates modularity because the different parts of the single core processor were put together using the different processor and cache modules. Modularity refers to a system composed of separate components that can be connected together, which is exactly how this system was designed. Since we did not have to worry about the implementation of the processor or caches (for this lab at least), the design is also an example of encapsulation, meaning the modules are self-governing. We could easily swap our implementation out for a different implementation if so desired because only the input and output connections must remain the same.

Section 3: Alternative Design

The alternative design is a quad-core processor with four instruction caches and a banked datacache system, shown in detail in Figure 7 and at a higher level in Figure 2. Networks handle the interface between the caches and main memory. The multicore datacache contains four data cache banks, a cachenet, and a memnet where the cachenet connects the caches and handles the banking of the caches and the memnet connects the banked caches to the main memory. We were responsible for connecting this multicore datacache, the instruction caches, the memnet which connects to the instruction caches, and the four processors.

For this alternative design, we chose to use our alternative processor design from lab 2 for the processors and also our alternative cache design from lab 3 for the instruction caches. We chose to do this because the alternative design for the processor over the baseline design because the baseline only has the capability of utilizing stalling to deal with data hazards whereas the alternative has the ability to use bypassing on top of stalling. We also chose to use the alternative design for the instruction caches over the baseline. This is because the baseline is a direct-mapped cache whereas the alternative is a two-way set-associative cache. Compared to the direct-mapped cache, the set-associative cache results in a larger hit rate since for any given index there are two different ways where the tag can be stored. A larger hit rate means that memory is accessed less, which saves on clock cycles.

The alternative design's CacheNet, shown in Figure 3, handles the interface between the four processors and the four data cache banks. This network has an Upstream (processor) and Downstream (main memory) message adapter which convert processor and main memory messages respectively to network messages. The Upstream adapter extracts the bank bits, seen in Figure 4, from the processor and includes them in the destination header. The MemNet, shown in Figure 5, works in more or less the same way as the CacheNet, but the Upstream adapter always inserts a 0 into the destination field. The MemNet is used to refill the instruction caches and data cache banks. Finally, the McoreDataCache module, seen in Figure 6, combines the CacheNet, MemNet, and four cache banks to create the complete shared, 4-banked data cache system. We instantiate the McoreDataCache along with four processors, four instruction caches, and an additional MemNet for the instruction caches in the MultiCore system, simulating a realistic processor and cache system. This incremental development approach is a good example of modularity and hierarchy because the components are grouped into self-governing modules which build upon each other to form the final MultiCore system at the top of the hierarchy structure. The design also lends itself to the principle of extensibility since it would be easy to improve a subcomponent and simply swap it into the existing design. For instance, if we were to change the network topology, we could simply develop new CacheNet or MemNet modules and replace the current modules.

Additionally, we wrote a hybrid quicksort/mergesort (parallel sort) algorithm for sorting an array of integers. Our mergesort essentially takes two sorted arrays and incrementally combines individual elements from the two arrays into a larger, sorted array. The mergesort function took in an argument vector pointer. With this argument, we could determine the destination and source array pointers as well as the "begin" and "end" indexes of the array needing to be sorted. The size of the array could be determined by subtracting the "begin" index from the "end" index. Our mergesort algorithm breaks up the array into two, one starting at the "begin" index and ending at the middle index and the other starting at the middle index and ending at the "end" index. The index of the middle was calculated by taking half the size of the array and adding it to the begin index. The two sub-arrays have been sorted by the quicksort algorithm. Our mergesort algorithm will then execute a for-loop the size of the final destination array and fill each element. This is done by comparing the elements in the two sorted sub-arrays. At each iteration of the for-loop, the algorithm checks to see which sub-array has the smallest number at the "head" index and that element is placed into the next index of the destination array.

The “head” index for a sub-array starts at its first element and is incremented each time an element from that subarray is put into the final destination array. Mergesort is a good alternative because it utilizes a divide-and-conquer method and its worst complexity is O(n*log(n)). In fact, it is most optimized for combining already-sorted arrays, which is why we use it in tandem with our scalar quicksort algorithm.

A multi-core (4-core) processor is a good alternative design because it has the ability to run multiple threads simultaneously, but we can begin to see the drawbacks of adding more cores. As discussed later in our evaluation, there are costs to adding more cores, hence why we don’t use 100 cores. This comparison is similar to why we did not use 100 pipeline stages in our pipeline processor in lab 2. Though more work can be done in parallel, it becomes harder to divide the work evenly and the overhead becomes higher as well. There is also often no need for so much work to be done in parallel, as not many programs have such characteristics.

All in all, a multicore processor is an optimization over the single core processor in the baseline design because it provides a larger bandwidth and an increased thread-level parallelism that increases processor performance significantly. Though there is more complexity in the multithreaded mergesort than the single threaded quicksort, resulting in more instructions to be processed, the parallelization of the multi-core processor optimizes the performance of sorting the array over the single-core single threaded sort.

Section 4: Testing

In order to test our baseline and alternative designs, we used directed and random testing. Directed testing was important so that we could set up certain situations, predict the behavior, and observe the results in order to make sure that the functioning was correct. Random testing was important because it could produce situations that we did not even think to try in order to test the processor designs. For the random testing, we utilized both random value and random delay testing to cover all the aspects of the processors. We go more into detail about these different testing strategies in the following paragraphs.

Directed testing was essential to ensuring that the processors would respond the way we expected them to respond. Tests for the CacheNet, MemNet, and McoreDataCache are provided to us. These tests use the test sources to load messages into the network and verify that the network properly processes these messages. For the CacheNet and McoreDataCache modules, tests from Lab 3 are used since these modules accept cache requests. Instead of a single source and sink, however, there are now four source and sink pairs. For the entire MultiCore composition, we are provided with an incremental testing strategy. First, the directed assembly sequences are used to verify that arithmetic operations, multiplication operations, memory operations, jumps, branches, etc. are working properly. These tests choose registers to load test values into, perform operations on them, and then check for correctness in the test sink. They also allow us to force data dependencies to occur in order to test proper hazard management. These tests are finally combined into more comprehensive unit test suites that combine several instruction types into a sequence that simulates more realistic computation. At this point, random delay and random value testing are also performed, as explained in the next section.

Random delay tests were used in lab in order to ensure that the ready and valid signals between all the different modules were working correctly. The ready and valid signals are essential in the functioning of these processors because they control what data flows through the system and when that data flows through the system. If these signals are incorrect, the wrong data will be delivered to the module waiting on a specific piece of data. These random delays specify arbitrary stalls into the system to make sure that data is passed around as soon as it is available, it should wait until the valid/ready bits are high between the modules that it is passing through. Random value testing was also implemented to ensure not only that the design is correct across general use cases but also to simulate what the processor might see in a real computing scenario. For instance, random testing is often useful for covering edge cases or other special cases we may not have thoroughly checked in our directed testing. Moreover, random testing sometimes exposes errors in our microarchitectural implementation, such as when we forgot to ensure the reliability of our multiplier from lab 2. Random value testing was implemented using Python’s built-in pseudorandom number generation functions for the operand values and checking the output of the processor against the expected value that we determine as a function of the inputs.

Part of our incremental testing strategy was to thoroughly test the hardware and software portions of the lab separately before combining them. We tested the correctness of our sorting algorithms by creating different arrays that test various edge cases. Since there are four cores, it is important to test the behavior when the number of array elements is not a multiple of four. We simply created arrays of size 127, 126, and 125 to test all the possible remainders when dividing the size by 4. These tests ensure that we handle the array indexing correctly when dividing up the work among the four cores. We also added arrays with all elements of equal value, as well as arrays with fewer than 4 elements in order to test corner cases. The non-multiples-of-four sorting tests were implemented to allow us to confirm the correctness of our multi-core processor’s allocation of work for the different cores even when the work is not able to be split up evenly by the four cores.

After all these tests, we can safely conclude that our design is functionally correct because we carefully tested each aspect of the system with both random and controlled tests. The random value and delay tests ensure that our system functions correctly for cases we might not have considered and the directed tests allowed us to confirm that the designs behave the way we expected.

Section 5: Evaluation

All of the multi-threaded benchmarks performed better on the multicore system than the corresponding single-threaded benchmarks on the single-core system. However, they do not result in the theoretical 4X speedup over the single core; in most cases, they resulted in a speedup of just over 3X (see CPI in Table 0). This fact results from a number of reasons. First, the code that runs on the multicore system tends to require significantly more assembly instructions than the code for the single-core system due to the extra overhead of dividing up arrays and spawning work for the worker cores. Second, the multicore system tends to have a greater number of instruction cache misses than the single core, possibly due to a less efficient use of spatial locality. As a result, the four individual cores have more than a fourth of the instructions of the single core and have a higher CPI than the single core (compare core0 instructions and CPI to single core in Table 0). However, when these cores are combined, the overall CPI becomes lower due to executing the four cores in parallel. Also, the theoretical 4X speedup is not generally attainable because full parallelization is impossible for most programs and serial portions of code can dominate execution time.

The single-threaded benchmarks perform very poorly on the multicore system due to the fact that there was no thread-level parallelism to take advantage of. For example, the multicore system performed worse on the quicksort benchmark than the single core system. This is because the work was not divided up between the cores, meaning the optimization that the multicore system provides was not useful for the single-threaded benchmarks. Though our multicore system seems to have a high CPI, that is only because the calculations are done on the basis that the work is divided among the four cores, not repeated on them. It might also be the case that the multicore system must stall or squash more instructions than the single core due to the data hazard of having multiple cores executing the same instructions. Not only is it unnecessary to execute an instruction four separate times, it is also detrimental to the performance.

Sorting was chosen as an evaluation method because it is work that can be split up between the different cores and re-combined to come to the final answer. The mergesort algorithm essentially breaks up the array into the smallest sub-arrays and starts sorting from there, meaning these sub-arrays can be sorted by the different cores and then re-combined by one of them to produce the final sorted array. This allows us to easily evaluate the difference between running a sorting algorithm on a single core as opposed to multiple cores and compare the results, which is why we used sorting as an evaluation method.

The different benchmarks provided, which all have improved performance in the multithreaded versions, demonstrate that breaking up the single thread into multiple threads on multiple cores will have generally similar effects. The benchmarks perform operations on arrays, which have high spatial locality. The spatial locality of the benchmarks takes advantage of our banked cache system such that the different processor cores all have access to the same data. This fact, along with the benchmarks' ability to be divided among multiple cores, make them good benchmarks for evaluation. The benchmarks did have some small differences. For example, bsearch did not require as many extra instructions to be divided up among the cores, but that is due to the nature of this being a search algorithm which lends itself well to multithreading. Bsearch also had a higher datacache hit rate as compared to the single core version. The sort benchmark on the other hand, required much more extra instructions than the single core to run. This is because the sorting benchmark requires more communication between the cores. However, the result is still similar in that the multicore did much better in terms of CPI than the single core.

The multicore system has a significant increase in area and energy requirements due to the quadrupling of processors and caches in addition to the added networks. The overall reduction in computational time offsets this increase somewhat, but not completely. The multicore system is ideal if performance is the main concern and energy and area requirements are secondary, such as in desktop computers and industrial applications. On the other hand, in certain systems like embedded devices and mobile devices where energy and size are critical requirements, it may not be worth it to use a multicore system.

All in all, we can conclude that the multi-core processor significantly increases the performance of programs with a multi-threaded nature whereas it provides little to no optimization for single-threaded programs. This is obvious because a multi-core processor allocates different threads into the different cores so that they can be executed simultaneously and increase the throughput of the program. Additionally, an important conclusion is that performance does not scale linearly with number of cores, as demonstrated by the less than 4X speedup. In realistic systems, various overheads that come with increasing thread-level parallelism result in diminishing returns as the number of cores increases, which explains why modern systems do not use hundreds or thousands of cores. Besides the obvious energy and area increases that this would entail, realistic multicore systems must manage coherence, consistency, and synchronization. The marginal performance benefits are not worth the astronomical costs of blindly adding more cores.

Section 6: Additional Diagrams

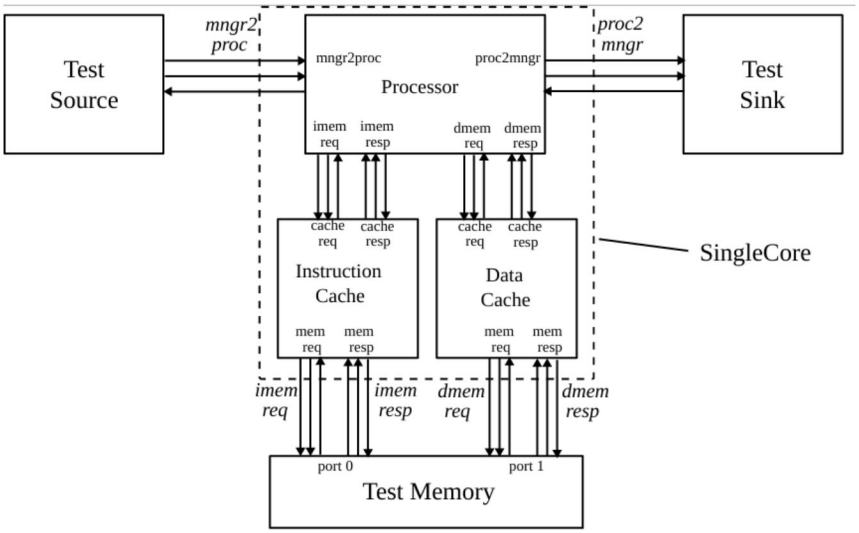

Figure 1: SingleCore – The `SingleCore` module as a whole, is hooked up to the test source, test sink, and test memory for testing and evaluation. Each bundle of the three arrows is a msg/val/rdy port bundle. The ports with inclined names are the top-level ports that `SingleCore` expose.

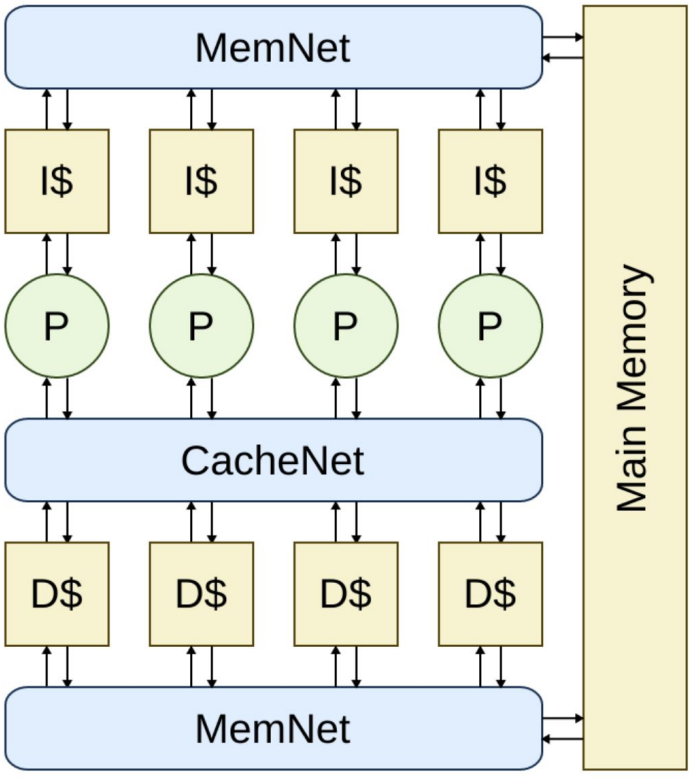

Figure 2: Alternative design block diagram – The alternative design consists of four processors, four private I-caches, a four-banked shared D-cache, and several networks to route dmem request/response from the processors and the D-cache, and refilling both the I-cache and the D-cache. Note that each network shown in the diagram is actually two networks: one for request and the other for response.

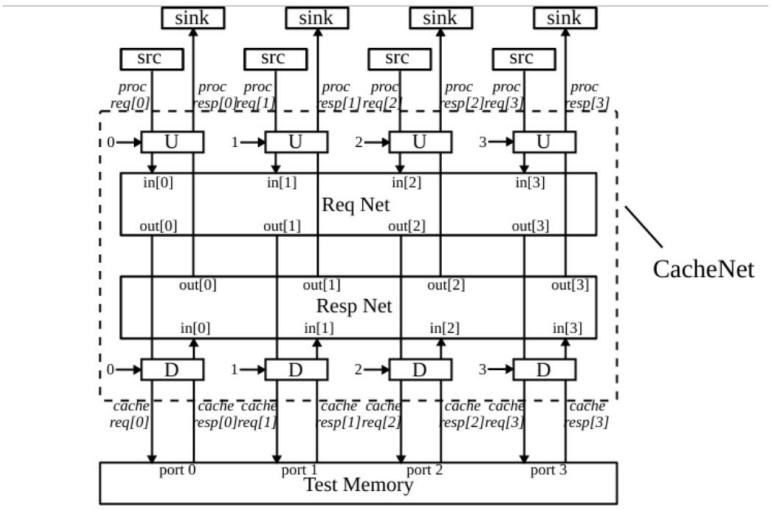

Figure 3: CacheNet – The `CacheNet` module as a whole, is hooked up to the test sources, test sinks, and four-port test memory for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. Each "U" and "D" is an adapter which takes care of both the request port and the response port, and has its own adapter id. The ports with inclined names are the top-level ports that `CacheNet` exposes.

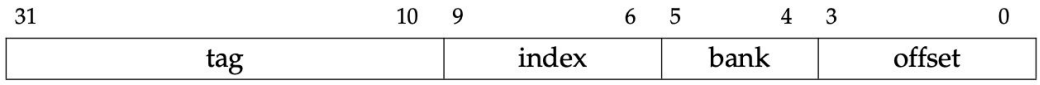

Figure 4: Memory Address Formats With Banking – Addresses for the data cache include two bank bits which are used to choose which bank to send the memory request; in other words, the bank bits are used as the destination field when converting a memory request message into a network message.

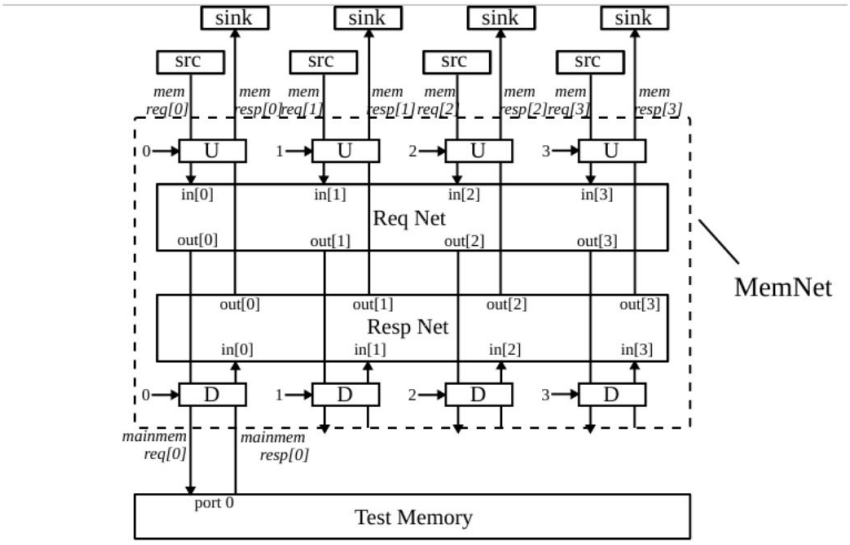

Figure 5: MemNet – The `MemNet` module as a whole, is hooked up to the test sources, test sinks, and single-port test memory for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. Each "U" and "D" is an adapter which takes care of both the request port and the response port, and has its own adapter id. The ports with inclined names are the top-level ports that `MemNet` exposes.

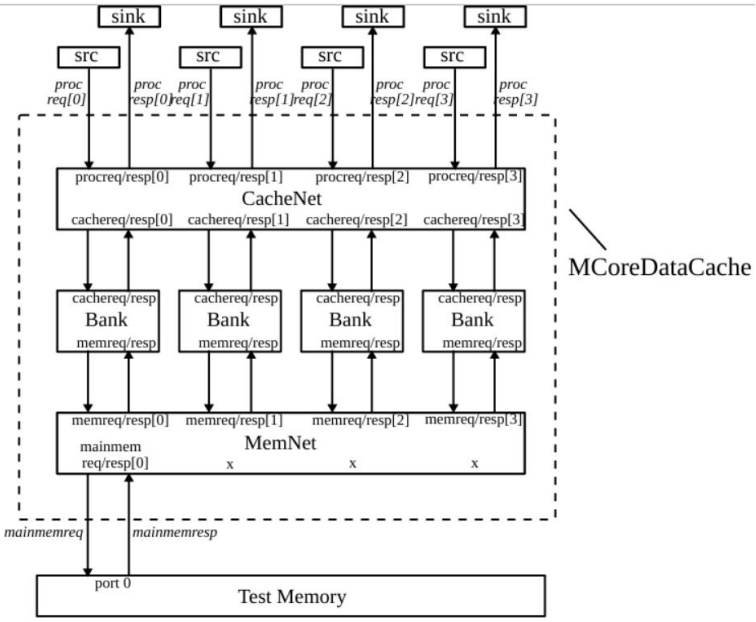

Figure 6: MCoreDataCache – The `McoreDataCache` module as a whole, is hooked up to the test sources, test sinks, and single-port test memory for unit testing. Each arrow is actually a msg/-val/rdy port bundle whose direction is the msg/val direction. We omit the details inside `MemNet` and `CacheNet` but show the port names to match the previous diagrams. The ports with inclined names are the top-level ports that this module exposes.

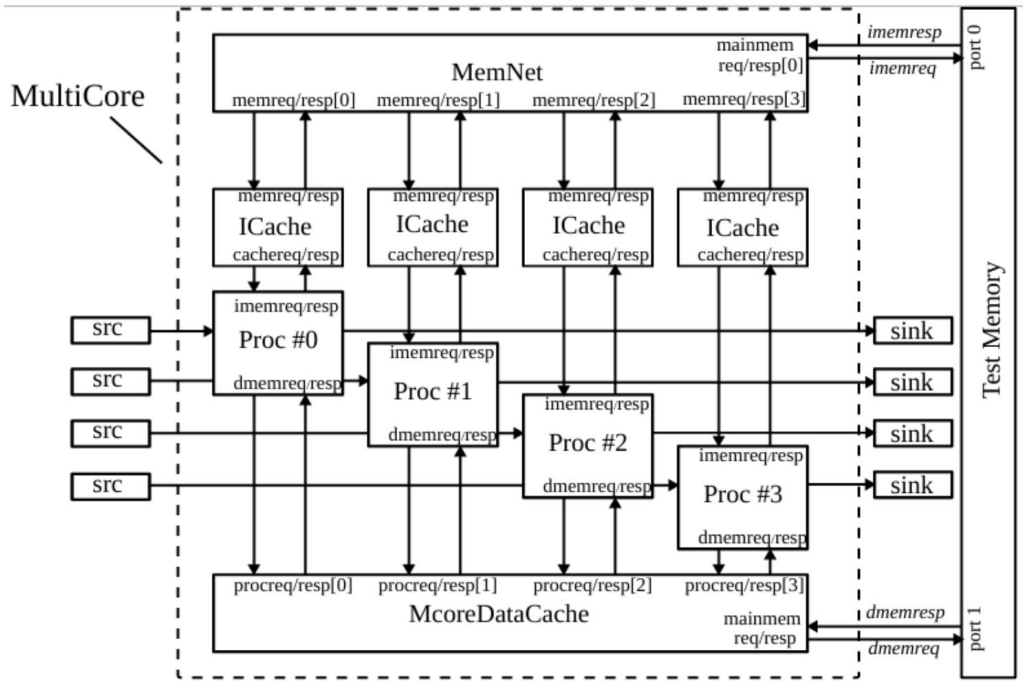

Figure 7: MultiCore – The `MultiCore` module as a whole, is hooked up to a dual-port test memory, four test sources and four test sinks. Note that to not to make the diagram too crowded, we omit the *proc2mngr* and *mngr2proc* port names both at the boundary of `Multicore` as well as in the processor. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. The ports with inclined names are the top-level ports that `MultiCore` exposes.

Table 0: Comparison of single-threaded code on single-core processor to multi-threaded code on multi-core processor

	Sort		VVadd		Bsearch		Cmult		Mfilt	

Metric	Score	Mcore	Score	Mcore	Score	Mcore	Score	Mcore	Score	Mcore
num cycles	56019	38153	4311	2260	10341	3831	12223	4915	25989	9856
total cpi	5.02	1.37	5.32	1.51	4.91	1.34	6.08	1.81	4.73	1.41
total instructions	11158	27790	811	1492	2106	2855	2011	2711	5493	7006
total icache miss rate	0.0013	0.0302	0.0055	0.0334	0.0063	0.0292	0.0033	0.0223	0.0039	0.0176
total dcache miss rate	0.0618	0.1025	0.2533	0.2181	0.6234	0.3369	0.151	0.1559	0.2454	0.2364
core0 commit inst		6282		341		673		641		1673
cor0 cpi		6.07		6.63		5.69		7.67		5.89

Table 1: Ubmark Quicksort

Ubmark-quicksort		
ISA	**Score**	**Mcore**
score	In stats_en region:	In stats_en region:
In stats_en region:		
total_committed_inst = 11158	num_cycles = 56019	num_cycles = 80025
	total_committed_inst = 11158	total_committed_inst = 44489
mcore	total_cpi = 5.02	total_cpi = 1.80
In stats_en region:		
total_committed_inst = 44632	total_icache_miss = 16	total_icache_miss = 61
	total_icache_access = 12780	total_icache_access = 50982
core0_committed_inst = 11158	total_icache_miss_rate = 0.0013	total_icache_miss_rate = 0.0012
core1_committed_inst = 11158		
core2_committed_inst = 11158	total_dcache_miss = 225	total_dcache_miss = 3882
core3_committed_inst = 11158	total_dcache_access = 3642	total_dcache_access = 14484
	total_dcache_miss_rate = 0.0618	total_dcache_miss_rate = 0.2680
		core0_committed_inst = 11158
		core0_cpi = 7.17
		core1_committed_inst = 11168
		core1_cpi = 7.17
		core2_committed_inst = 11084
		core2_cpi = 7.22
		core3_committed_inst = 11079
		core3_cpi = 7.22
		icache0_miss = 16
		icache0_access = 12780
		icache0_miss_rate = 0.0013
		icache1_miss = 15

		icache1_access = 12793
		icache1_miss_rate = 0.0012
		icache2_miss = 15
		icache2_access = 12707
		icache2_miss_rate = 0.0012
		icache3_miss = 15
		icache3_access = 12702
		icache3_miss_rate = 0.0012
		dcache_bank0_miss = 1001
		dcache_bank0_access = 3749
		dcache_bank0_miss_rate = 0.2670
		dcache_bank1_miss = 898
		dcache_bank1_access = 3429
		dcache_bank1_miss_rate = 0.2619
		dcache_bank2_miss = 1017
		dcache_bank2_access = 3701
		dcache_bank2_miss_rate = 0.2748
		dcache_bank3_miss = 966
		dcache_bank3_access = 3605
		dcache_bank3_miss_rate = 0.2680

Table 2: Mtbmark Sort

Mtbmark-sort	
ISA	**Mcore**
In stats_en region:	In stats_en region:
total_committed_inst = 26280	
	num_cycles = 38153
core0_committed_inst = 6310	total_committed_inst = 27790
core1_committed_inst = 6596	total_cpi = 1.37
core2_committed_inst = 6674	
core3_committed_inst = 6700	total_icache_miss = 931
	total_icache_access = 30831
	total_icache_miss_rate = 0.0302
	total_dcache_miss = 570

	total_dcache_access = 5560
	total_dcache_miss_rate = 0.1025
	core0_committed_inst = 6282
	core0_cpi = 6.07
	core1_committed_inst = 7074
	core1_cpi = 5.39
	core2_committed_inst = 7217
	core2_cpi = 5.29
	core3_committed_inst = 7217
	core3_cpi = 5.29
	icache0_miss = 287
	icache0_access = 7207
	icache0_miss_rate = 0.0398
	icache1_miss = 213
	icache1_access = 7794
	icache1_miss_rate = 0.0273
	icache2_miss = 197
	icache2_access = 7932
	icache2_miss_rate = 0.0248
	icache3_miss = 234
	icache3_access = 7898
	icache3_miss_rate = 0.0296
	dcache_bank0_miss = 144
	dcache_bank0_access = 1882
	dcache_bank0_miss_rate = 0.0765
	dcache_bank1_miss = 160
	dcache_bank1_access = 1097
	dcache_bank1_miss_rate = 0.1459
	dcache_bank2_miss = 115
	dcache_bank2_access = 1105
	dcache_bank2_miss_rate = 0.1041
	dcache_bank3_miss = 151
	dcache_bank3_access = 1476

	dcache_bank3_miss_rate = 0.1023

Table 3: Ubmark VVAdd

Ubmark-vvadd		
ISA	**Score**	**Mcore**
score	In stats_en region:	In stats_en region:
In stats_en region:		
total_committed_inst = 811	num_cycles = 4311	num_cycles = 5934
	total_committed_inst = 811	total_committed_inst = 3272
mcore	total_cpi = 5.32	total_cpi = 1.81
In stats_en region:		
total_committed_inst = 3244	total_icache_miss = 5	total_icache_miss = 20
	total_icache_access = 912	total_icache_access = 3687
core0_committed_inst = 811	total_icache_miss_rate = 0.0055	total_icache_miss_rate = 0.0054
core1_committed_inst = 811		
core2_committed_inst = 811	total_dcache_miss = 76	total_dcache_miss = 236
core3_committed_inst = 811	total_dcache_access = 300	total_dcache_access = 1196
	total_dcache_miss_rate = 0.2533	total_dcache_miss_rate = 0.1973
		core0_committed_inst = 811
		core0_cpi = 7.32
		core1_committed_inst = 828
		core1_cpi = 7.17
		core2_committed_inst = 820
		core2_cpi = 7.24
		core3_committed_inst = 813
		core3_cpi = 7.30
		icache0_miss = 5
		icache0_access = 912
		icache0_miss_rate = 0.0055
		icache1_miss = 5
		icache1_access = 931
		icache1_miss_rate = 0.0054
		icache2_miss = 5
		icache2_access = 926
		icache2_miss_rate = 0.0054
		icache3_miss = 5

		icache3_access = 918
		icache3_miss_rate = 0.0054
		dcache_bank0_miss = 60
		dcache_bank0_access = 303
		dcache_bank0_miss_rate = 0.1980
		dcache_bank1_miss = 55
		dcache_bank1_access = 287
		dcache_bank1_miss_rate = 0.1916
		dcache_bank2_miss = 57
		dcache_bank2_access = 301
		dcache_bank2_miss_rate = 0.1894
		dcache_bank3_miss = 64
		dcache_bank3_access = 305
		dcache_bank3_miss_rate = 0.2098

Table 4: Mtbmark VVAdd

Mtbmark-vvadd	
ISA	**Mcore**
In stats_en region:	In stats_en region:
total_committed_inst = 1400	
	num_cycles = 2260
core0_committed_inst = 341	total_committed_inst = 1492
core1_committed_inst = 353	total_cpi = 1.51
core2_committed_inst = 353	
core3_committed_inst = 353	total_icache_miss = 55
	total_icache_access = 1646
	total_icache_miss_rate = 0.0334
	total_dcache_miss = 89
	total_dcache_access = 408
	total_dcache_miss_rate = 0.2181
	core0_committed_inst = 341
	core0_cpi = 6.63
	core1_committed_inst = 385

	core1_cpi = 5.87
	core2_committed_inst = 385
	core2_cpi = 5.87
	core3_committed_inst = 381
	core3_cpi = 5.93
	icache0_miss = 26
	icache0_access = 379
	icache0_miss_rate = 0.0686
	icache1_miss = 9
	icache1_access = 424
	icache1_miss_rate = 0.0212
	icache2_miss = 10
	icache2_access = 424
	icache2_miss_rate = 0.0236
	icache3_miss = 10
	icache3_access = 419
	icache3_miss_rate = 0.0239
	dcache_bank0_miss = 23
	dcache_bank0_access = 125
	dcache_bank0_miss_rate = 0.1840
	dcache_bank1_miss = 22
	dcache_bank1_access = 96
	dcache_bank1_miss_rate = 0.2292
	dcache_bank2_miss = 21
	dcache_bank2_access = 85
	dcache_bank2_miss_rate = 0.2471
	dcache_bank3_miss = 23
	dcache_bank3_access = 102
	dcache_bank3_miss_rate = 0.2255

Table 5: Ubmark Bsearch

Ubmark-bsearch		
ISA	**Score**	**Mcore**
score	In stats_en region:	In stats_en region:

In stats_en region:		
total_committed_inst = 2106	num_cycles = 10341	num_cycles = 11721
	total_committed_inst = 2106	total_committed_inst = 8430
mcore	total_cpi = 4.91	total_cpi = 1.39
In stats_en region:		
total_committed_inst = 8424	total_icache_miss = 15	total_icache_miss = 57
	total_icache_access = 2393	total_icache_access = 9574
core0_committed_inst = 2106	total_icache_miss_rate = 0.0063	total_icache_miss_rate = 0.0060
core1_committed_inst = 2106		
core2_committed_inst = 2106	total_dcache_miss = 149	total_dcache_miss = 262
core3_committed_inst = 2106	total_dcache_access = 239	total_dcache_access = 958
	total_dcache_miss_rate = 0.6234	total_dcache_miss_rate = 0.2735
		core0_committed_inst = 2106
		core0_cpi = 5.57
		core1_committed_inst = 2108
		core1_cpi = 5.56
		core2_committed_inst = 2108
		core2_cpi = 5.56
		core3_committed_inst = 2108
		core3_cpi = 5.56
		icache0_miss = 15
		icache0_access = 2393
		icache0_miss_rate = 0.0063
		icache1_miss = 14
		icache1_access = 2394
		icache1_miss_rate = 0.0058
		icache2_miss = 14
		icache2_access = 2393
		icache2_miss_rate = 0.0059
		icache3_miss = 14
		icache3_access = 2394
		icache3_miss_rate = 0.0058
		dcache_bank0_miss = 87
		dcache_bank0_access = 286
		dcache_bank0_miss_rate = 0.3042

		dcache_bank1_miss = 58
		dcache_bank1_access = 252
		dcache_bank1_miss_rate = 0.2302
		dcache_bank2_miss = 60
		dcache_bank2_access = 252
		dcache_bank2_miss_rate = 0.2381
		dcache_bank3_miss = 57
		dcache_bank3_access = 168
		dcache_bank3_miss_rate = 0.3393

Table 6: Mtbmark Bsearch

Mtbmark-bsearch	
ISA	**Mcore**
In stats_en region:	In stats_en region:
total_committed_inst = 2827	
	num_cycles = 3831
core0_committed_inst = 687	total_committed_inst = 2855
core1_committed_inst = 714	total_cpi = 1.34
core2_committed_inst = 713	
core3_committed_inst = 713	total_icache_miss = 94
	total_icache_access = 3216
	total_icache_miss_rate = 0.0292
	total_dcache_miss = 125
	total_dcache_access = 371
	total_dcache_miss_rate = 0.3369
	core0_committed_inst = 673
	core0_cpi = 5.69
	core1_committed_inst = 722
	core1_cpi = 5.31
	core2_committed_inst = 730
	core2_cpi = 5.25
	core3_committed_inst = 730
	core3_cpi = 5.25
	icache0_miss = 36

	icache0_access = 765
	icache0_miss_rate = 0.0471
	icache1_miss = 19
	icache1_access = 811
	icache1_miss_rate = 0.0234
	icache2_miss = 20
	icache2_access = 818
	icache2_miss_rate = 0.0244
	icache3_miss = 19
	icache3_access = 822
	icache3_miss_rate = 0.0231
	dcache_bank0_miss = 34
	dcache_bank0_access = 134
	dcache_bank0_miss_rate = 0.2537
	dcache_bank1_miss = 36
	dcache_bank1_access = 85
	dcache_bank1_miss_rate = 0.4235
	dcache_bank2_miss = 33
	dcache_bank2_access = 81
	dcache_bank2_miss_rate = 0.4074
	dcache_bank3_miss = 22
	dcache_bank3_access = 71
	dcache_bank3_miss_rate = 0.3099

Table 7: Ubmark Cmult

Ubmark-cmult		
ISA	**Score**	**Mcore**
score	In stats_en region:	In stats_en region:
In stats_en region:		
total_committed_inst = 2011	num_cycles = 12223	num_cycles = 23616
	total_committed_inst = 2011	total_committed_inst = 8050
mcore	total_cpi = 6.08	total_cpi = 2.93
In stats_en region:		
total_committed_inst = 8044	total_icache_miss = 7	total_icache_miss = 31
	total_icache_access = 2112	total_icache_access = 8452

core0_committed_inst = 2011	total_icache_miss_rate = 0.0033	total_icache_miss_rate = 0.0037
core1_committed_inst = 2011		
core2_committed_inst = 2011	total_dcache_miss = 151	total_dcache_miss = 900
core3_committed_inst = 2011	total_dcache_access = 1000	total_dcache_access = 4001
	total_dcache_miss_rate = 0.1510	total_dcache_miss_rate = 0.2249
		core0_committed_inst = 2011
		core0_cpi = 11.74
		core1_committed_inst = 2011
		core1_cpi = 11.74
		core2_committed_inst = 2012
		core2_cpi = 11.74
		core3_committed_inst = 2016
		core3_cpi = 11.71
		icache0_miss = 7
		icache0_access = 2112
		icache0_miss_rate = 0.0033
		icache1_miss = 8
		icache1_access = 2111
		icache1_miss_rate = 0.0038
		icache2_miss = 8
		icache2_access = 2112
		icache2_miss_rate = 0.0038
		icache3_miss = 8
		icache3_access = 2117
		icache3_miss_rate = 0.0038
		dcache_bank0_miss = 229
		dcache_bank0_access = 1005
		dcache_bank0_miss_rate = 0.2279
		dcache_bank1_miss = 217
		dcache_bank1_access = 992
		dcache_bank1_miss_rate = 0.2188
		dcache_bank2_miss = 221
		dcache_bank2_access = 996
		dcache_bank2_miss_rate = 0.2219

		dcache_bank3_miss = 233
		dcache_bank3_access = 1008
		dcache_bank3_miss_rate = 0.2312

Table 8: Mtbmark Cmult

Mtbmark-cmult	
ISA	**Mcore**
In stats_en region:	In stats_en region:
total_committed_inst = 2600	
	num_cycles = 4915
core0_committed_inst = 641	total_committed_inst = 2711
core1_committed_inst = 653	total_cpi = 1.81
core2_committed_inst = 653	
core3_committed_inst = 653	total_icache_miss = 64
	total_icache_access = 2870
	total_icache_miss_rate = 0.0223
	total_dcache_miss = 173
	total_dcache_access = 1110
	total_dcache_miss_rate = 0.1559
	core0_committed_inst = 641
	core0_cpi = 7.67
	core1_committed_inst = 693
	core1_cpi = 7.09
	core2_committed_inst = 697
	core2_cpi = 7.05
	core3_committed_inst = 680
	core3_cpi = 7.23
	icache0_miss = 30
	icache0_access = 679
	icache0_miss_rate = 0.0442
	icache1_miss = 11
	icache1_access = 735
	icache1_miss_rate = 0.0150
	icache2_miss = 11
	icache2_access = 737

	icache2_miss_rate = 0.0149
	icache3_miss = 12
	icache3_access = 719
	icache3_miss_rate = 0.0167
	dcache_bank0_miss = 43
	dcache_bank0_access = 277
	dcache_bank0_miss_rate = 0.1552
	dcache_bank1_miss = 43
	dcache_bank1_access = 304
	dcache_bank1_miss_rate = 0.1414
	dcache_bank2_miss = 44
	dcache_bank2_access = 266
	dcache_bank2_miss_rate = 0.1654
	dcache_bank3_miss = 43
	dcache_bank3_access = 263
	dcache_bank3_miss_rate = 0.1635

Table 9: Ubmark Mfilt

Ubmark-mfilt		
ISA	**Score**	**Mcore**
score	In stats_en region:	In stats_en region:
In stats_en region:		
total_committed_inst = 5493	num_cycles = 25989	num_cycles = 32518
	total_committed_inst = 5493	total_committed_inst = 22065
mcore	total_cpi = 4.73	total_cpi = 1.47
In stats_en region:		
total_committed_inst = 21972	total_icache_miss = 23	total_icache_miss = 87
	total_icache_access = 5836	total_icache_access = 23457
core0_committed_inst = 5493	total_icache_miss_rate = 0.0039	total_icache_miss_rate = 0.0037
core1_committed_inst = 5493		
core2_committed_inst = 5493	total_dcache_miss = 322	total_dcache_miss = 1262
core3_committed_inst = 5493	total_dcache_access = 1312	total_dcache_access = 5269
	total_dcache_miss_rate = 0.2454	total_dcache_miss_rate = 0.2395
		core0_committed_inst = 5493
		core0_cpi = 5.92

		core1_committed_inst = 5506
		core1_cpi = 5.91
		core2_committed_inst = 5534
		core2_cpi = 5.88
		core3_committed_inst = 5532
		core3_cpi = 5.88
		icache0_miss = 23
		icache0_access = 5836
		icache0_miss_rate = 0.0039
		icache1_miss = 22
		icache1_access = 5851
		icache1_miss_rate = 0.0038
		icache2_miss = 21
		icache2_access = 5886
		icache2_miss_rate = 0.0036
		icache3_miss = 21
		icache3_access = 5884
		icache3_miss_rate = 0.0036
		dcache_bank0_miss = 321
		dcache_bank0_access = 1333
		dcache_bank0_miss_rate = 0.2408
		dcache_bank1_miss = 324
		dcache_bank1_access = 1341
		dcache_bank1_miss_rate = 0.2416
		dcache_bank2_miss = 307
		dcache_bank2_access = 1296
		dcache_bank2_miss_rate = 0.2369
		dcache_bank3_miss = 310
		dcache_bank3_access = 1299
		dcache_bank3_miss_rate = 0.2386

Table 10: Mtbmark Mfilt

Mtbmark-mfilt	
ISA	**Mcore**

In stats_en region:	In stats_en region:
total_committed_inst = 6866	
	num_cycles = 9856
core0_committed_inst = 1681	total_committed_inst = 7006
core1_committed_inst = 1715	total_cpi = 1.41
core2_committed_inst = 1759	
core3_committed_inst = 1711	total_icache_miss = 132
	total_icache_access = 7491
	total_icache_miss_rate = 0.0176
	total_dcache_miss = 368
	total_dcache_access = 1557
	total_dcache_miss_rate = 0.2364
	core0_committed_inst = 1673
	core0_cpi = 5.89
	core1_committed_inst = 1771
	core1_cpi = 5.57
	core2_committed_inst = 1821
	core2_cpi = 5.41
	core3_committed_inst = 1741
	core3_cpi = 5.66
	icache0_miss = 48
	icache0_access = 1788
	icache0_miss_rate = 0.0268
	icache1_miss = 29
	icache1_access = 1875
	icache1_miss_rate = 0.0155
	icache2_miss = 29
	icache2_access = 1958
	icache2_miss_rate = 0.0148
	icache3_miss = 26
	icache3_access = 1870
	icache3_miss_rate = 0.0139
	dcache_bank0_miss = 77
	dcache_bank0_access = 373

	dcache_bank0_miss_rate = 0.2064
	dcache_bank1_miss = 104
	dcache_bank1_access = 487
	dcache_bank1_miss_rate = 0.2136
	dcache_bank2_miss = 99
	dcache_bank2_access = 362
	dcache_bank2_miss_rate = 0.2735
	dcache_bank3_miss = 88
	dcache_bank3_access = 335
	dcache_bank3_miss_rate = 0.2627

Table 11. Member Roles

Lab	sh997	byx2	yo82
Lab 1	RTL	Verification	RTL (architect)
Lab 2	Verification	RTL(architect)	RTL
Lab 3	RTL(architect)	RTL	Verification
Lab 4	-	-	-
Lab 5	RTL	Verification	RTL (architect)

Table 12. Role and Task Table

	sh997	byx2	yo82
Tasks	**Baseline Design:** -Helped write Quicksort algorithm **Alternative Design:** -Helped debug Mergesort algorithm, made connections and debugged multi-core hardware **Testing:** Wrote some of the arrays we tested quicksort & mergesort on **Report:** Alternative, evaluation	**Baseline Design:** - **Alternative Design:** -Helped write parallel sorting algorithm -Verifying connections to networks and memory **Testing:** -Wrote software tests for parallel sorting algorithm **Report:** -Testing, baseline design, evaluation	**Baseline Design:** -Helped write Quicksort algorithm **Alternative Design:** -Helped write Mergesort algorithm, connections for multi-core **Testing:** -Ensured passing of tests of designs **Report:** -Introduction, baseline design, alternative design, testing