Lab 4 Report: Single-Core and Multi-Core Systems

1 Introduction

This lab is meant to serve as a culminating design experiment that brings together all the components built and used in previous labs in order to assemble an ultimate completed project - a functional n-core system. Through the implementations of both the single core (n=1) and multi-core systems (n=2+), we were able to examine and evaluate how different architectural choices affected both area and performance through several benchmark tests, including a self-made sorting algorithm benchmark test written to test the functionality and robustness of the hardware, giving us additional experience in seeing the interactions between hardware and software. In doing so, we once again were able to reinforce the idea that hardware optimizations often involve balancing competing factors, and from that came to know the benefits of using one design versus the other. At a high level, the base design combines a five-staged fully-bypassed processor and separate instruction and data caches to formulate a functional single core system capable of supporting single-threaded workloads, while the alternative design extends the architecture used in the base design to four pipelined processors, each with its own individual instruction cache, a shared four-bank data cache, and a ring network to enable inter-core communication as well as parallel execution. Thus, through implementing, testing, and evaluating both systems, we were able to develop a deeper understanding of how each n-core system is composed and operates, as well as the tradeoffs that come with each design in terms of scalability versus simplicity.

2 Alternative Design

The main change made for the alt design was adding in a ring network to enable communication between all four cores, allowing them communication to a shared banked data cache as well as providing a path for instruction cache refills. The base design and alt design use the same processors and cache, but the network keeps all memory traffic requests organized so the cores don't come into conflict with each other. The routing algorithm is kept in the routing unit for each router, and for every message it receives, takes the destination ID from the header of the message and compares it to its own router ID. If the IDs match, the message is sent out from the local port (port 0), but if not, then the route unit forwards the message. The router is set up to support both clockwise and counterclockwise directional sending of messages, though our implementation only uses clockwise due to its simplicity to set up (with the unused counterclockwise port remaining in the file for a future implementation). However, the tradeoff of this is the increased cycle count it takes to move some messages (like a message moving from router 0 to router 3). After the routing logic decides where these messages should go, the switch units, each with three inputs (one from each route unit), selects one message to send in a given cycle and thus determine which messages are sent to their respective destinations. The arbitration algorithm used for the switch unit is that of fixed priority arbitration, with priority given to messages already moving through the network (ports 1 and 2) over ones coming in from the local port, which prevents one

active core from stalling messages that were already moving, keeping the traffic flowing and reducing build up in the queue. From these lower level modules, a router is then created, using an input queue, route unit, and switch unit. The route units and switch units house our routing and arbitration implementations discussed earlier while the input queues temporarily hold onto messages before sending them to their respective route units. When each route unit receives a message, it sends the messages to the switch units, which go through and dissolve any conflicts that arise from the messages wanting to access the same destination port. Every router instantiated in our ring network holds this system.

At the top level for our ring network, we take our four instantiated routers and connect their clockwise and counterclockwise ports in a loop (0→1→2→3→0), with the structure set up so each router behaves the same way (the only difference is the ID). The benefit of this design, which is also used in CacheNet and MemNet, is its reusability, reducing the design complexity. However, the tradeoff of this system is the latency of the message traveling, as due to our routing logic, messages can take much longer to reach a router on the other side of our ring (e.g. a workload that with extensive amounts of long clockwise paths would perform poorly). Despite this, for most datasets, the parallelism of the extra cores is not a detriment and can significantly decrease cycle count.

The main software change was introducing multi-threaded sorting logic that can use the four cores to its advantage. The base design works strictly with a single core so there is no parallelism, and thus was implemented using a simple quick-sort algorithm for single-threaded testing (ubmark-sort). On the other hand, mtbmark-sort takes an input array and splits it between all the active worker cores so each core can sort independently, allowing for parallel work to be run on the hardware. At the top of the file we define a struct (mtbmark_sort_arg_t) to hold the pointer of the array as well as the first and last indices assigned to a worker. Using this struct allows us to pass all the arguments a worker core needs in one go.

The first helper function we made is an insertion sort meant for a subarray. The idea was to have a simple sort that can be used by each worker for its portion of work, and since each worker should be receiving a smaller segment of the array when all are active, insertion sort vies as the better option, being more efficient than quicksort as it works better on shorter ranges. Using insertion sort as our fallback case and worker case also keeps the implementation consistent. Each worker uses this helper in mtbmark_sort_worker, and the function unpacks the pointer and its indices, then calls insertion sort on that portion of the array. Each worker core does not need to know anything about the other cores as its only focus is to sort the segment it was given (if the size to sort is 1 or less, it just returns the array as there is nothing to sort). Our next function is mtbmark_sort_partition, created with the purpose to take the full array and split it into evenly sized blocks to be given to each active worker core. We start with finding the base block size with division and the remaining elements due to uneven division. The extra elements are assigned to a core until all elements have been distributed for a more even workload between cores, and to create the boundaries of each block we initialize starts[0] = 0 and then compute the next few entries by adding the block size plus one more to deal with remainders. We do this so starts[i] is the start of the block and starts[i+1] is the end, capturing every entry in between while making sure there is no overlap.

The top level function, mtbmark_sort, brings together the helpers described and decides

when to run the sort sequentially or in parallel. In the native environment, it first checks the amount of active workers and uses that to get the appropriate block size. If there is only one active core, we just fallback onto the single thread algorithm. We then partition our array, using our struct to fill in arguments that we'll use to spawn our workers for all blocks but block 0 (which is directly sorted by the master core). Once all the workers finish their sorts (joined with ece4750_bthread_join), they are merged back into one fully sorted array (as handled by mtbmark_sort_merge which combines two adjacent sorted arrays into one by checking and adding the smaller element to a temp array until fully sorted). This keeps the multi-threaded logic simple while producing a fully sorted array. In the RISC-V environment, mtbmark does not use any of our multi-threaded logic and simply calls ubmark_sort. This is to ensure that in running evaluations with ubmark, both the base and alt design run the same software and we can purely test the performance of our hardware. We only use our multi-threaded path in the native environment to show how the software can allow for parallelism in hardware. Some tradeoffs our software has to deal with is the cost of merging (dividing the arrays quickens the sorting portion of the algorithm, but the final merge brings in extra work that grows with the size of a block). The overall speedup depends on the balance of block size and the cost of merge while our use of insertion sort relies on our core count heavily (insertion sort scales poorly on larger ranges - a large workload or not enough active workers can reduce the efficiency), yet this approach still provides a noticeable speedup due to parallelism.

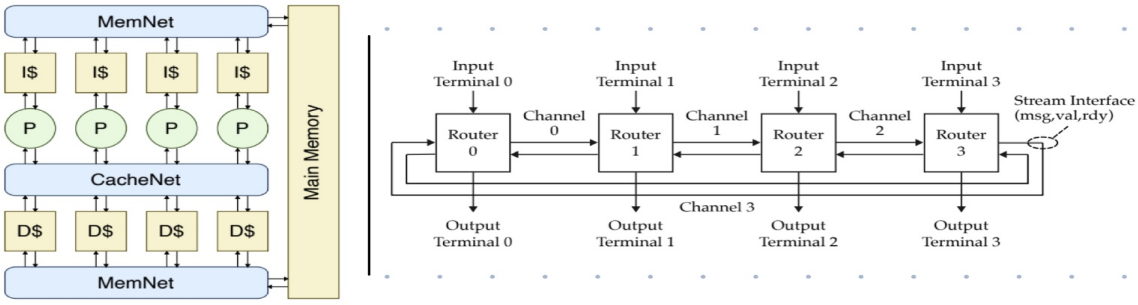

Figure 1: Alternative Design Hardware Datapath (left) and Ring Network (right).

3 Testing Strategy

Across both the hardware and software components used in this lab, our testing strategy largely revolved around using incremental validation, allowing us to start small and recognize issues with our logic early before they progressed into the higher level modules, where it would be harder to detect the bug causing the issue. For each new module instantiated in the hardware, we began with unit-level assertion-based tests while using the FL model as a reference before expanding to directed and randomized system-level instruction testing (with and without delays). In the testing and creation of the sorting algorithm, we started by first testing it only against the provided basic test first to ensure it was working correctly before adding in more tests (using directed and white-box testing) and concluding with using the sorting benchmarks for both single-core and multi-core execution.

For our single-core system hardware, our testing strategy largely revolved around using assertion-based testing. After first ensuring that our multiplier, pipelined processor, and cache designs were working correctly and passing all the staff tests (ensuring we would not encounter any hiccups from previous lab modules we would use in this lab), we moved on to implementing the processor and two caches and wiring everything together. Once the hardware was fully working and succinctly passing all of the course staff test cases, we moved on to a directed-based testing approach by including all of the various register-register, register-immediate, memory, branch, and jump instructions, as well as incorporating the four new instructions added into this lab (XORI, ANDI, SRLI, and BGE) with both random and directed test cases into the SingleCoreSysFL_test.py file to more fully test the robustness of the single core system. No issues arose after adding in these instructions, so we moved on to implementing the software.

The software testing for our single core system went relatively smoothly, and we tackled it using a very incremental approach with the simple C unit testing framework. We started by adding in a simple quick-sort algorithm and then optimizing it using insertion, making the necessary range and partition helper functions in the ubmark-sort.c file before adding a call to helpers in the corresponding ubmark-sort.h header file for unit testing. Then we made sure that the sorting algorithm we used was passing natively before we changed to the build directory and ran only the basic test against the single-core system FL simulator and RTL implementation of the single-core system, only moving on to the next testing environment only after ensuring the basic test had passed for each preceding testing environment. Once the quick-sort algorithm was passing everywhere for the basic test case, we moved on to include more directed test cases using assertion-based testing which were designed to specifically target edge cases. Some examples of the tests used include sorting datasets that contain negative numbers, repeated values, no values, just one value, the large dataset found in the ubmark-sort.dat file, or a set that requires no additional sorting. To fully test the internal behavior of the algorithm in order to fully ensure the robustness of the design, we then implemented two additional tests using white-box testing, targeting behaviors like the partition boundaries and tail-recursion structure in the range helper, which may have otherwise appeared correct at the top level even if subtle off-by-one errors occurred. All test cases were tested in a similar manner to the basic test case (against all three environments) before moving on to tackling the multi-core system and the network.

Our testing strategy for multi-core system hardware similarly revolved around assertion-based testing using an incremental approach. After thorough review of the lab handout to ensure we understood the priority and transition logic, we started by first compiling the lower level route and switch unit modules, testing each unit in isolation before moving on to the next against the staff tests, and repeating the process with the higher level router and network modules, which incorporated clockwise (and unused counterclockwise) transition logic between the lower level modules. Specifically, we used a combination of directed tests, randomized traffic, and made use of source and sink delays to stress our design under back pressure for all of the units made. For our route unit, our added tests focused on testing different routing outcomes - creating cases where all messages were local, forwarded, and a mixture of local plus remote messages as well as a case where we applied back pressure to verify that the unit could still correctly deliver the message under high stress conditions. Our switch unit tests focused on verifying the correctness of our arbitration algorithm, with some

cases only making use of one input type to confirm correct baseline behavior while another has multiple inputs in conflict for the same output (which was our most important test as it demonstrated the intended fixed priority behavior). The router needed traffic patterns that stressed all ports so we created cases where multiple srcs targeted the same destination (and its inverse), and where alternating src and destination pairs to simulate bouncing traffic back and forth, confirming that our routing and arbitration logic are communicating correctly. The final full ring network tests were designed to stress the overall stability, using both a hotspot traffic test to check if the network could correctly handle many routers sending to the same destination and back and forth patterns that confirmed that our design can still make progress through the ring, even though we only implemented clockwise forwarding. Additional assembly was written using directed testing for the mem_mcore instruction by exercising varied access patterns across cores, cache banks, and cache lines, ensuring correct functionality under higher memory pressure and stress to the cores.

Once the hardware was working and succinctly passing all of the course staff test cases (for each isolated unit we implemented), we added in all of the instructions from Lab 2 (and the ones excluded initially in Lab 2), with both random and directed test cases into the MultiCoreSysFL_test.py file to more fully test the robustness of the multi-core system, just as we did for the base design. Similar to the single core, no issues arose after adding in these instructions, but we did find a number of errors when running all of the course staff tests that were still present, as indicated by seeing failing cases for both the DCache and Sys tests. The Sys tests were resolved by simply parameterizing the processor back in Lab 2 while the DCache bugs were found using the line traces and –tb=short, which helped us identify a common characteristic between the two failing test cases that relied on the "init" transaction. Once we identified this issue, we went back to the control unit for Lab 3, and identified that we were never previously enabling the read data register in only the init transaction, which led to the issues of seeing an unexpected miss when a hit should have occurred. After identifying this issue and making the appropriate change to the cs table, all the tests passed, and we moved on to software.

The software testing for our multi-core system and network went relatively smoothly in comparison to the headaches of finding bugs in the hardware. Software testing was done similarly to how it was done when testing the single-core: incrementally and by first making an insertion-sort sorting algorithm (because with alt design's multiple cores, arrays are already broken up in a way similar to how insertion works, making it more desirable) that passed natively, against the FL simulator, and RTL implementation with only the basic test case before moving onto the incorporating more test cases before adding in our own directed tests. Once all tests were passing, we were confident that our design was robust and working exactly as expected.

4 Evaluation

We evaluated the performance of the base (single-core) and alternative (multi-core) system designs using the four software benchmark tests provided, as well as an additional sorting benchmark that we created to highlight the differences between the two designs in terms of scalability and parallel efficiency, among other tradeoffs, for both single-threaded and multi-

	VVADD					CMULT					Sorting Algorithms				
	ubmark		mtbmark			ubmark		mtbmark			ubmark		mtbmark		
	base	alt	base	alt 1	alt	base	alt	base	alt 1	alt	base	alt	base	alt 1	alt
num_cycles	4267	6171	5284	7497	2895	12461	17383	13458	18824	5812	29192	39127	29279	38799	38815
num_inst	812	812	973	973	376	1812	1812	1973	1973	626	5652	5652	5656	5656	5656
CPI	5.25	7.6	5.43	7.71	7.7	6.88	9.59	6.82	9.54	9.28	5.16	6.92	5.18	6.86	6.86
num_icache_access	913	913	1100	1100	419	1913	1913	2100	2100	669	6600	6600	6605	6605	6605
num_icache_miss	6	6	45	45	34	9	9	49	49	37	197	197	198	198	198
icache_miss_rate	0.01	0.01	0.04	0.04	0.08	0	0	0.02	0.02	0.06	0.03	0.03	0.03	0.03	0.03
num_dcache_access	301	301	355	355	124	801	801	855	855	249	1874	1874	1874	1874	1874
num_dcache_miss	75	75	98	85	26	150	150	172	166	46	73	34	80	34	34
dcache_miss_rate	0.25	0.25	0.28	0.24	0.21	0.19	0.19	0.2	0.19	0.18	0.04	0.02	0.04	0.02	0.2

	MFILT					BSEARCH				
	ubmark		mtbmark			ubmark		mtbmark		
	base	alt	base	alt 1	alt	base	alt	base	alt 1	alt
num_cycles	29093	35988	28504	35797	13460	10705	12468	11858	14008	4671
num_inst	6241	6241	6009	6009	2210	2124	2124	2309	2309	770
CPI	4.66	5.77	4.74	5.96	6.09	5.04	5.87	5.14	6.07	6.07
num_icache_access	6800	6800	6495	6495	2320	2497	2497	2708	2708	872
num_icache_miss	24	24	138	138	56	12	12	69	69	45
icache_miss_rate	0	0	0.02	0.02	0.02	0	0	0.03	0.03	0.05
num_dcache_access	1083	1083	1165	1165	448	240	240	302	302	119
num_dcache_miss	184	198	215	221	97	136	98	159	120	35
dcache_miss_rate	0.17	0.18	0.18	0.19	0.22	0.57	0.41	0.53	0.4	0.29

Figure 2: Base vs. Alternative Designs for Single (ubmark) and Multi-Threaded (mtbmark) Software Benchmark Tests with Testing Cycles, Cache Accesses, and Miss Rates.

threaded inputs. At a high level, the alternative design consistently incurred higher access latency due to the added complexity of the ring network and the shared four-bank data cache, making it slower for all single-threaded (ubmark) workloads compared to the simpler base design, which features significantly less hardware and digital logic. Yet, in cases where the workload exposes thread-level parallelism that can be exploited or features memory accesses that cleanly distribute across the cache banks, the multi-core system provided substantial performance benefits.

When it came to running the single-threaded (ubmark) tests, the base design consistently outperformed the alternative design across every single benchmark. For example, in the heavy arithmetic benchmark of vector-vector add (VVADD), the base design only required 4,267 cycles with a CPI of 5.25, while the alternative multi-core system required 6,171 cycles and thus a higher CPI of 7.60. These increases in both the number of cycles and CPI in the alt design are due to the higher miss penalty (and thus a higher AMAL) with the added logic of going through the ring network, despite both designs experiencing only 6 instruction-cache misses and nearly similar low miss rates. Other more memory-based benchmarks also denoted a similar trend with the increased number of cycles due to additional stalls needed as irregular accesses produced more contention in the banked data-cache. The MFILT benchmark in particular makes a note of this, with the base design requiring nearly 7,000 less cycles to complete (29,093 vs. 35,988 cycles) than the alternative design. Therefore, it is obvious that a clear trend emerges from the comparison of the two designs against the single-threaded tests, with the multi-core system's deeper memory hierarchy dominating the behavior of the system, resulting in a greater amount of cycles needed to fully complete and thus slower performances.

In contrast, running the multi-threaded (mtbmark) tests allowed the alternative design's advantages to flourish when true parallelism was exploited. This became apparent in the cases when the alt design was forced to use 1 worker (and as a result performed in a similar way to the base design, though expectedly a little worse due to the added logic and com-

plexity of the ring network and shared multi-bank data-cache) versus the standard 4 workers the alternative design is equipped to use - highlighting that the multi-core system alone does not improve performance, as the parallel work distribution is required to offset the higher per-access latency. When running with all four workers, VVADD sees its cycle count drop to 2,895 cycles in comparison to the base design's 5,284 - nearly half as many cycles and running twice as fast as either single-threaded version. The CMULT benchmark similarly improves for the alternative design as its regular per-thread access pattern is able to be exploited by the multi-threaded tests (13,458 cycles (base) vs. 5,812 cycles (alt)). The other two provided benchmarks also experienced improvements in the number of cycles used and the CPI, albeit more modestly, with BSEARCH in particular only seeing a slight improvement since its irregular access pattern caused multiple threads to probe nearby addresses, which increased bank conflicts and yielded minimal speedup. Still though, for nearly all the provided benchmarks, it is worth noting that there was an obvious trend of a decreased number of cycles used, total misses, and miss rates (due to fewer accesses) for the data cache when the alternative design was used in multi-thread testing (with the exception of MFILT, the only benchmark that experiences an increased d-cache miss rate due to heavy competition with threads interfering with each other's sliding windows and boundaries between the thread partitions, causing increased eviction conflicts). It is additionally worth noting that the sorting benchmark made does not follow the trend of the other benchmarks, though this can be explained via the mtbmark_sort always falling back to single-threaded ubmark_sort, so in essence there is no parallel sorting happening in the hardware tests, coupled with more irregular memory accesses and a higher miss penalty caused by the ring network.

Beyond cycle counts and miss rates, the alternative design introduces tradeoffs in area, energy, and cycle time. As opposed to the single-core system, the multi-core system uses significantly more (roughly four times as much) hardware with four processors, four private instruction caches, four data-cache banks, and multiple routers, resulting in a much larger area being used. Similar to the area, the energy per cycle also increases with the alternative design because there is more switching going on between routers and each cache miss involving multi-hop network traversal. While the alternative design does benefit with highly parallel workloads that may reduce total execution time, memory-bound workloads may consume more total energy due to prolonged stalls. The critical path used would also likely be lengthened from that of the base design with the additional routing, arbitration, and banking logic, reducing the maximum clock frequency. Thus, it is clear that the multi-core design prioritizes scalability over efficiency, with benefits only realized when multi-thread parallelism is both abundant and well-structured.

5 Conclusion

In conclusion, our evaluation shows that the base single-core system consistently delivers the best performance for all single-threaded inputs, completing workloads like VVADD and MFILT in 4,267 and 29,093 cycles, respectively, whereas the multicore system incurs additional ring-network and banked-cache latency, slowing the same tests by 20-30 percent. However, when true parallelism is available, the alternative multicore design becomes substantially more effective, as seen in VVADD and CMULT, where using all four workers

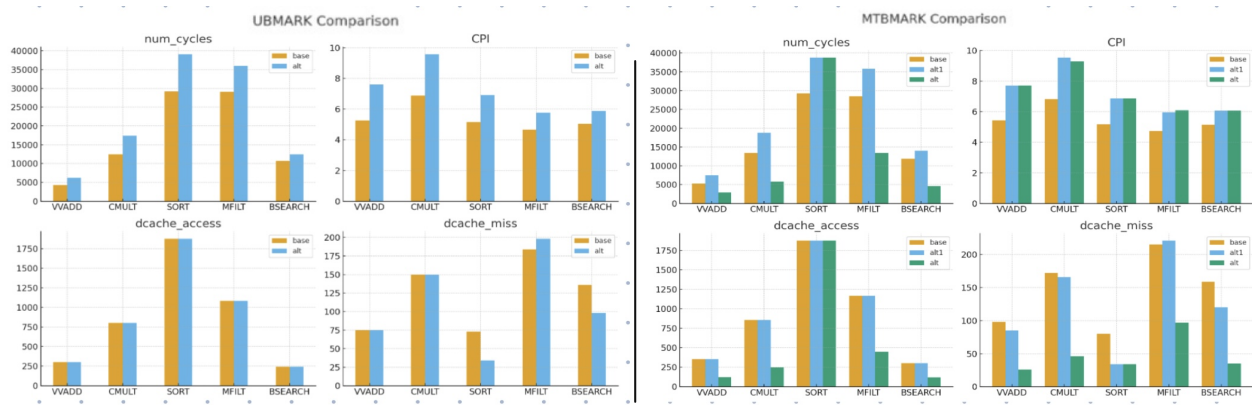

Figure 3: Single-Threaded vs. Multi-Threaded Benchmark Performance Plots for Base (orange) and Alternative (green (1 worker) and blue (4 workers) Designs.

reduces execution time to 2,895 and 5,812 cycles, achieving results in less than half the time over any single-core run, while more memory-bound or irregular workloads (MFILT, BSEARCH, sorting) show only moderate gains due to bank conflicts in the data cache, among other things. Qualitatively, these results highlight that the multicore system excels on regular, data-parallel inputs, but performs worse on single-threaded or irregular inputs where its deeper memory hierarchy introduces unnecessary overhead and a greater miss penalty. Looking forward, the choice between designs depends entirely on the workload, as the base design is preferable for simpler, sequential, or latency-sensitive tasks with its faster, less complex memory path, whereas the multicore design should be used when software can reliably expose parallelism or needs to be scalable. Ultimately, this reinforces the central architectural lesson that performance, area, and energy tradeoffs are workload-dependent, with no single design being universally superior.

6 Work Distribution

Throughout this lab, both members of the team contributed meaningfully to its completion. Rawsen implemented the base design and sorting tests used in the implementation of the single core system while David took on the network logic used in the hardware of the alternative multi core system. Both members of the team worked together in the implementation of the software used in the multi-core system, as well as going through the past three labs to identify and debug bugs that were previously missed by staff tests and only became apparent after the multi-core implementation with increased banks and the number of cores, through the use of line tracing, the waveforms, and help received in Office Hours. Artificial intelligence was used in this lab in the assistance of creating software tests, as well as in the revision of the alternative design of the two-way set associative blocking cache from Lab 3 to fix datapath misconceptions, and a few test cases for the networking used in the alternative design of this lab. Past the code, artificial intelligence was also used in the brainstorming of a few edge test cases used in the testing of the mem_mcore instruction, but not in the preparation of this lab report.