LAB 4 REPORT: MULTICORE PROCESSOR

Luke Puchalla [lcp77]
Eldor Bekpulatov [eb654]
Shivangi Gambhir [sg2439]

Luke Puchalla (RTL Design & Verification):

- Found and fixed bugs in lab 2 and lab 3 code, which helped complete the final lab
- Contributed to writing the parallel sorting algorithm for the alternate (multi-core) design
- Wrote the testing strategy section and contributed to the evaluation section of this report
- Wrote about our parallel sorting algorithm in the alternative design section

Eldor Bekpulatov (RTL Design & Verification):

- Assisted with debugging of lab 2 and lab 3 code
- Contributed to writing the parallel sorting algorithm for the alternate (multi-core) design
- Developed python scripts for efficient testing of our microbenchmarks
- Developed python scripts for graphing results of our microbenchmarks
- Wrote the evaluation section of this report

Shivangi Gambhir (RTL Architect, Design & Verification):

- Implemented the software implementation for the Baseline Design
- Implemented the hardware implementation for the Alternative Design
- Wrote the Introduction, Baseline and Alternative Sections of this report
- Debugged and solved problems for the statistics printing in the Alternative Design

INTRODUCTION

The purpose of this lab is to understand the principles of single core and multi core systems and to design and evaluate the two systems. In the past labs, we have learned about pipelined processors and caches. In this lab, we are tasked to integrate these two together and form a working processor. In order to evaluate the performance of the two processors, we wrote software benchmarks for both the designs.

The baseline design, the framework for connecting the single core processor made up of a bypassing pipelined processor and instruction and data set-associative caches, was provided to us. The multi core processor, which connected networks and the four single cores, was successfully designed by us and thoroughly tested. The alternative design basically helped us in understanding how a multi core processor is designed and how multithreading is implemented by dividing the work among the four cores.

After successful implementation and evaluation of both designs, we discover that the quad-core processor expectedly has nearly a 4-times improvement in performance. At the cost of 4-5 times as much hardware space needed, as well as significantly more energy to implement the networks within the processor, the multi-core processor yields nearly 4-times faster performance.

BASELINE DESIGN

The baseline design for this lab consists of hardware and software implementations. The hardware implementation consists of designing a single core system composed of a processor, an instruction cache, and a data cache.

A single core processor is a microprocessor with a single core capable of running only a single thread at any given time. As seen in **Figure 2.1**, the single core system contains a single processor which we designed in Lab 2 and two caches which we designed in Lab 3. The cache request and response of the data and instruction caches connect to the dmem and imem request and response signals of the processor. The memory request and response signals are connected to the Test Memory.

The caches and the processor together form the core or the “brain” of the system. It receives instructions, performs operations or calculations based on the instruction type and provides the output.

The working of the single core is pretty straightforward like we learnt from the discussion in our classes. The processor sends a request to the icache to fetch an instruction. If the instruction is valid and present there, we get a hit and the cache responds back in affirmative. But if it is a mistake, the request is forwarded to the memory to fetch the instruction at the same address. The dcache works in a similar fashion where the data request is sent to the dcache and we receive a miss incase the data is not present in it and then we request the memory for data at the same address.

The data width from the processor to the cache is 32 bits since the request between the processor and caches are 4 bytes or 1 word and the data width from the cache to the memory is 128 bits since they access the entire cache line.

There are few parameters which define the type of implementation being used. For example, we set the number of banks as 0 for both the caches and number of cores as 1 for the processor for a single core implementation.

One of the vital goals in this lab was to understand the calculation of the statistics via the number of cache misses/accesses and the number of committed instructions.

The number of committed instructions are calculated using the processor. It is essentially the number of instructions that are not stalled and have written back completely.

The number of cache accesses depends on the request signals from the processor to the cache, i.e. the number of times we try to access the cache. Hence, it depends on the *req_rdy* and *req_val* signals.

The number of cache misses depends on the response signals from the cache to the processor. Hence, it depends on the *resp_rdy* and *resp_val* signals. In addition to this, we check the test field of the response message of the cache for a miss, i.e. it should be 0.

This qualifies as a good baseline since we’ve chosen the processor with bypassing logic which helps us reduce the cost of squashes and stalls, and the cache chosen is the set associative one which helps us reduce the miss rate and the average memory access latency.

For the software implementation of the design, we write a quicksort benchmark.

The quicksort algorithm used is depicted in **Figure 2.2**.

We have implemented the quicksort program using modular programming. We use a divide and conquer technique where a pivot element is chosen and the array is partitioned around that point. For our implementation, we have chosen the first element as the pivot point.

The first element is chosen as the pivot point and the array is divided into two i.e. one set greater than and one set less than the pivot element. Further, the first elements are chosen as the pivot elements again in the two new obtained sets. The same process is repeated until we get a fully sorted array.
Though this doesn't qualify as the best sorting algorithm because of the pivot element chosen as the first element of the array, it works for our baseline implementation and yields reasonable results.

ALTERNATIVE DESIGN

The alternative design for this lab consists of two implementations again:. hardware and software. The hardware implementation consists of designing a multi core system. Multi-core enables the system to perform more tasks at the same time, yielding a greater overall system performance. Multi-core is a type of architecture with two or more CPUs working together on the same chip. A single physical processor contains the core logic of two or more processors. Multithreading is the ability of a central processing unit to provide multiple threads of execution concurrently, supported by the operating system. We are designing such a system to harness the system performance by running multiple threads a time to see a comparison of the processor performance.
The design we are building is composed of four processors, four private instruction caches and four banked shared data cache and a couple of request and response networks to route requests and responses from the processors and caches.
It consists of three major modules as shown in **Figure 3.1**:

- **CacheNet**: Network connecting four processors and four data cache banks
- **MemNet**: Network connecting four instruction caches and one main memory port
- **McoreDataCache**: CacheNet + Four cache banks + MemNet

The above three modules have been designed for us. Each of the networks consist of a request/response pair. We are responsible for using these implementations in our multicore processor system.
As shown in **Figure 3.2,** we need to connect all the request/response signals from different units and tieing them all together at appropriate places. We followed an incremental design approach by starting with the MemNet all the way down to MCoreDateCache. This helps us keep a track of the signals and their connections with each other.
Here, we change the number of cores in the processor to 4 and connect the appropriate processor to the respective core_id which will be discussed later in this section. The number of banks still remains 0 for instruction caches as they are private and not banked. For the data cache, we use the McoreDataCache provided to us which takes the number of banks as 4 since the Dcache is shared among the four processors.
To instantiate the caches and connect them to the cores, we use a **generate** statement in verilog that runs the loop from 0 to the number of cores, i.e. 4 in our case.
Since the data cache is directly used from the McoreDataCache, we don't explicitly design a logic for connecting all the four cores/banks.
We once again look at the way the number of committed instructions and cache miss/access is calculated as that forms a good comparison of our design.
For the number of committed instructions, we just connect the number of committed instructions of each core to their respective core_id. Hence, the number for each core is calculated individually and then outputted individually.
The number of icache misses/accesses is also calculated in the same way.
For the number of dcache miss/access , we directly collect the statistics from the McoreDataCache module.

Our parallel sorting algorithm is relatively simple. First of all, if the source array has less than four elements, we simply use a single core to run quicksort (the single-core sorting algorithm described in the baseline section) on the array. Otherwise (if the array is greater than four elements), we divide the array into four roughly-even sections, and core 0 spawns three more threads to run quicksort on separate quarters of the source array. I.e. each of the three cores, and the parent core 0, are given a separate portion of the source array to run quicksort on. Finally, when all of the quicksort threads are finished, core 0 spawns two more threads, on cores 1 and 2 respectively. These threads execute a merge function, which we developed. One merge merges the first two "quarters" of the source array, and the second merge merges the second two "quarters." Finally, the parent core merges the two halves of the source array. Thus, we have designed a hybrid quicksort-merge function.

TESTING STRATEGY

The entirety of this project, including previous labs, was developed incrementally. We incrementally build compositions, unit test them, and reuse them to build larger components. In the first lab, with the iterative multiplier, we developed many directed and random tests to assert the functionality of our multiplier with all kinds of inputs. In the second lab, we integrated our multiplier as a single unit, which has already been thoroughly tested. We also designed and tested several other units, such as the immediate generator or the ALU, to include in the processor. After individually testing these discrete units within the processor, we combined them and tested instructions on the entire processor as a whole unit. Additionally, we designed and tested the processor in incremental steps: the first (baseline) step did not support stalling and squashing; the second (alternative) step did.

After finishing the processor, we designed the cache in lab 3. Since the state machine that controlled our cache had several different state paths it could take, we used a Design-for-Test design pattern, where we would implement and then explicitly test that path. We wrote many directed and random test cases, which constitute the unit tests for our cache "unit."

On the eve of this lab, we have two thoroughly tested units: the processor and the cache. We were provided with the entire testing framework for this lab. To test the single-core system, we reused the assembly tests for a representative subset of the instructions in Lab 2, shown in **Figure 4.1**. To reiterate: the units in the single-core system (the processor and the cache) were each tested thoroughly before combining into the single system.

As explained earlier, we incrementally build units and test them. We reuse the test cases from Lab 3 to test the CacheNet and the McoreDataCache since the input messages are basically cache requests. In this lab, since there will be four sources and four sinks, we have to use TestNetCacheSink to allow out-of-order delivery. Since we've thoroughly tested our cache in lab 3, we incorporate it into the CacheNet. Once the CacheNet is thoroughly tested, we incorporated the provided MemNet unit (which has also been thoroughly tested) into building the McoreDataCache subsystem. Finally, after testing the McoreDataCache, we can develop and then test the MultiCore unit, which includes four processors, four instruction caches, a shared banked data cache (McoreDataCache), and the MemNet memory network. All of the aforementioned units were discretely tested before combining into the final multi-core system.

We also thoroughly tested our quicksort and parallel-sort microbenchmarks. For directed testing, we developed several small dat files to #include at the tops of the quicksort or parallel sort files, and quickly run them in the build-native directory. We tested our algorithms with data arrays of sizes 1, 2, 3, and 4 specifically. These tests are generally designed to test the integrity of our algorithms in splitting up portions of the array to be worked on. We also utilized python to generate longer arrays of various types, such as an already-sorted array and a descending-value array. These types of arrays tested whether our quicksort algorithm and merging algorithms actually sorted properly.

Additionally, for the evaluation section, we used python to iteratively generate data files with source arrays of increasing length, 'make' the quicksort and parallel sort files to be run in assembly on their respective processors, execute the evaluations, and output their stats to output files. These comparisons will all be explained in depth in the evaluation section.

EVALUATION

We were provided with four benchmarks with which to assess our final designs: vvadd (vector-vector add), cmult (complex multiplication), mfilt (masked filter), and bsearch (binary search). We also designed our own quicksort and parallel sort algorithms and, as aforementioned, used python for in-depth analysis of each sorting algorithm's performance on its respective system (quicksort on the single-core system, and parallel sort on the multi-core system). Refer to **Table 5.1** for the results of the four provided evaluators, as well as the quicksort and parallel sort evaluators with the provided source array data.

Comparison across 4 provided benchmarks:

Generally across these benchmarks, the CPI for the quad-core is improved by about 3.5, nearly a factor of 4, which is certainly expected. Additionally, across these benchmarks, the instruction cache miss rate in the single-core processors is generally around a tenth of the icache miss rate for the quad-core processor. This makes sense, too, because each core has a separate instruction cache, and the program flow depends on the data given, which dictates the miss rate. We noticed that for true parallelizable functions like vvadd, cmult, mfilt where each core gets an independent piece of the original data and processes them in parallel, the performance benefits are realized greatly. Number of cycles and CPI are reduced by 4, while instruction accesses are multiplied by 4. Since each core gets an independent data sector, meaning the processing of the data does not depend on the data processed by the other cores, the data cache accesses are constant for both designs.

In-depth analysis with quicksort/parallel sort

Refer to the graphs in **Figures 5.1-5.5**. These graphs were developed by generating 256 different source arrays with randomly generated values; the arrays varied from size 0 to 255. We sorted each array with both sorting algorithms, and compared their performance statistics in the stated graphs. The single-core performance is graphed in orange, and the multi-core in blue. The green line represents a model for the predicted performance improvements from a quad-core over a single-core processor. This line appears in selected meaningful graphs.

Notice in the first graph (**Figure 5.1)**, with both the single and multi core processors, the number of cycles taken for the sorting increases linearly with the size of the array. Although the multi core processor increases at a smaller rate, it's slope is not 4 times less than that of the single-processor, which is depicted with a green line. Generally we observed reduction in the number of cycles with quad core, however for arrays less than the size of 50, quad-core performs noticeably worse due to its instruction overhead to start up the core. In **Figure 5.2**, notice that the multi-core processor committed many more instructions. We estimated the total number of instructions committed by quad-core should be about 4X of the single core, which is depicted in green. INote that for array lengths fewer than 50, quad core commits more than 4X the number of instructions committed by single core. At around 50 element arrays, quad core starts to commit about exactly 4 times as much as the single core, which is depicted by the intersection of green and blue lines in **Figure 5.2.** This is when the performance benefit of the quad core is realised, and we start seeing reduction in the number of cycles relative to the single core, depicted as the intersection of blue and orange lines in **Figure 5.1**. We had expected the CPI to be reduced, to be exact, approximately about 4 times as much. **Figure 5.3** shows the CPI of the multi-core processor at about 1.4, and the CPI of the single-core at about 5.2. This clearly shows the multi-core design as a 4x increase in instruction throughput, which is quite expected of a quad-core over a single-core.

Quad-core also came with a predictable instructions access pattern. We had expected running quad core to access instructions 4 times as much as depicted with a green line in **Figure 5.4,** and the results reflected to be close to prediction. We did again notice a turning point a where quadcore makes more instruction requests than out prediction prior to array lengths of 50 and intersects our prediction line at about array length of 50. **Figure 5.5** depicts the total data accesses made by each of the designs, and we predicted that data accesses should be approximately the same since the single core is simply processing the overall data, while multi core processes them in pieces. However we noticed the multicore makes twice as much data accesses compared to the single core, and we believe the cause to be the nature of the program. Since we did not implement a full quicksort, but a hybrid one, we definitely lost the true time complexity benefits of quicksort while merging.

Performance

The cycle time is not expected to vary between the processors. The execution time for a certain program on each processor, however, certainly varies. Across the board, we notice nearly 4X improvement in performance on the quad-core processor over the single-core processor. This is obviously expected if we include four cores. The type of program, however, can influence the degree of performance improvement. On simple tasks like vvadd or cmult, the improvement is obvious. In sorting tasks like quicksort, however, the performance can depend on the organization of data in the array.

Area

With this added performance improvement comes an area increase of 4-5 times more required hardware space. Certainly, we need 4 cores instead of one, however we also need space for the network in the processor to connect the cores and caches together. With more cores, such as eight or 12-core processors, this extra networking hardware space grows.

Energy

Though the memory network implementations were provided for us, they certainly require a significant amount of energy to correctly implement. However, since the same processor and cache designs were used in the quad-core as in the single-core system, no extra energy was required in the design of those smaller units. The bulk of the energy required to implement the quad-core system is dedicated to the network implementation and the overall connection of the components.

After successful implementation and evaluation of both designs, we discover that the quad-core processor expectedly has nearly a 4-times improvement in performance. At the cost of 4-5 times as much hardware space needed, as well as significantly more energy to implement the networks within the processor, the multi-core processor yields nearly 4-times faster performance across various application benchmarks. In some instances, however, the nature of the application can influence the processor performance; hence, this overall observation of 4x increase in performance is not totally and absolutely conclusive.

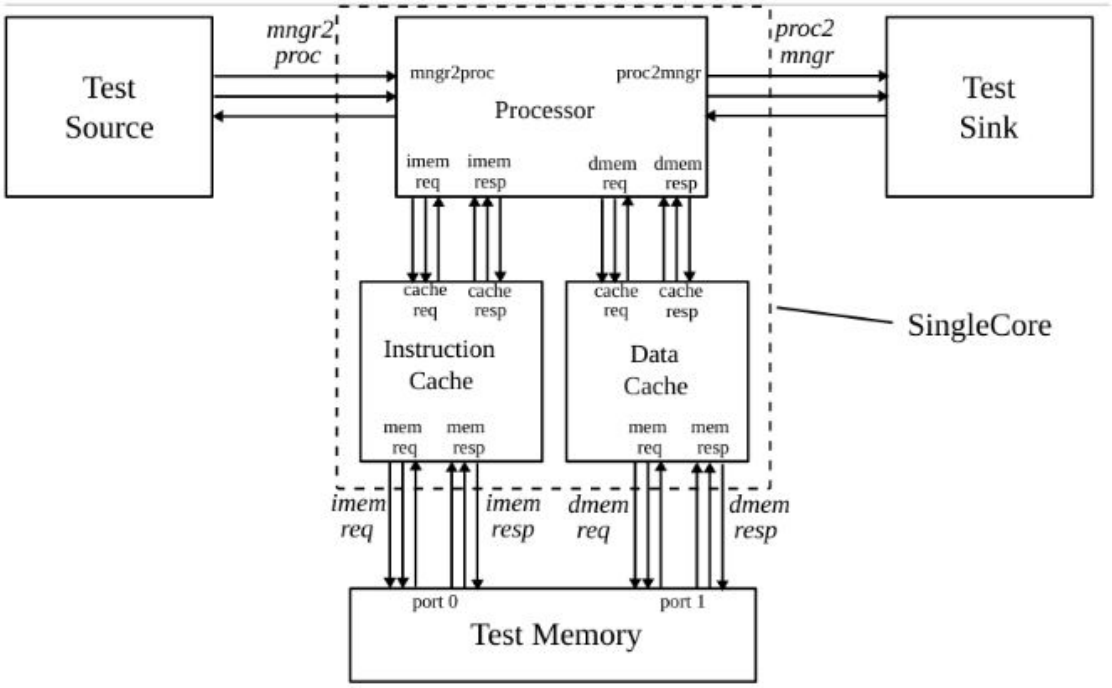


Figure 2.1

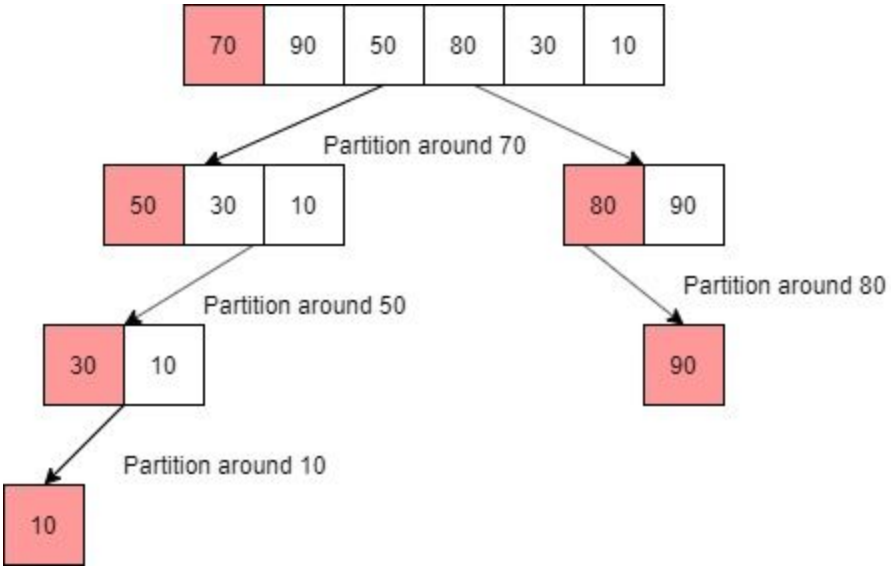
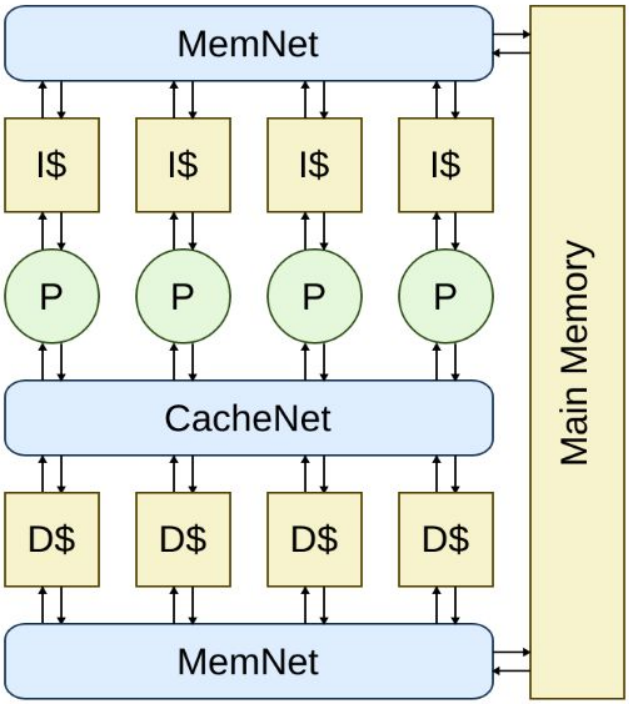


Figure 2.2


MemNet
I$
I$
I$
I$
P
P
P
P
CacheNet
D$
D$
D$
D$
MemNet
Main Memory

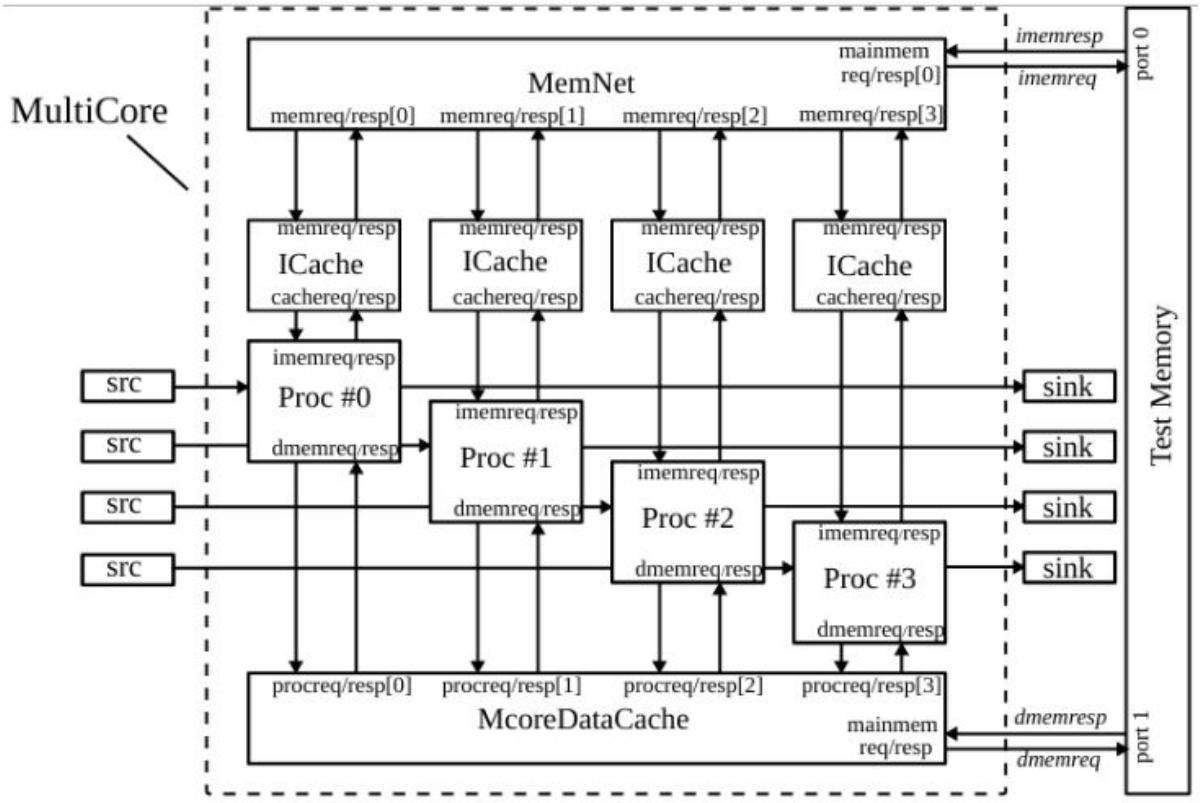

MultiCore
MemNet
memreq/resp[0]
memreq/resp[1]
memreq/resp[2]
memreq/resp[3]
mainmem
req/resp[0]
imemresp
imemreq
port 0
ICache
cachereq/resp
src
sink
Proc #0
Proc #1
Proc #2
Proc #3
imemreq/resp
dmemreq/resp
procreq/resp[0]
procreq/resp[1]
procreq/resp[2]
procreq/resp[3]
McoreDataCache
mainmem
req/resp
dmemresp
dmemreq
port 1
Test Memory

Figure 3.1

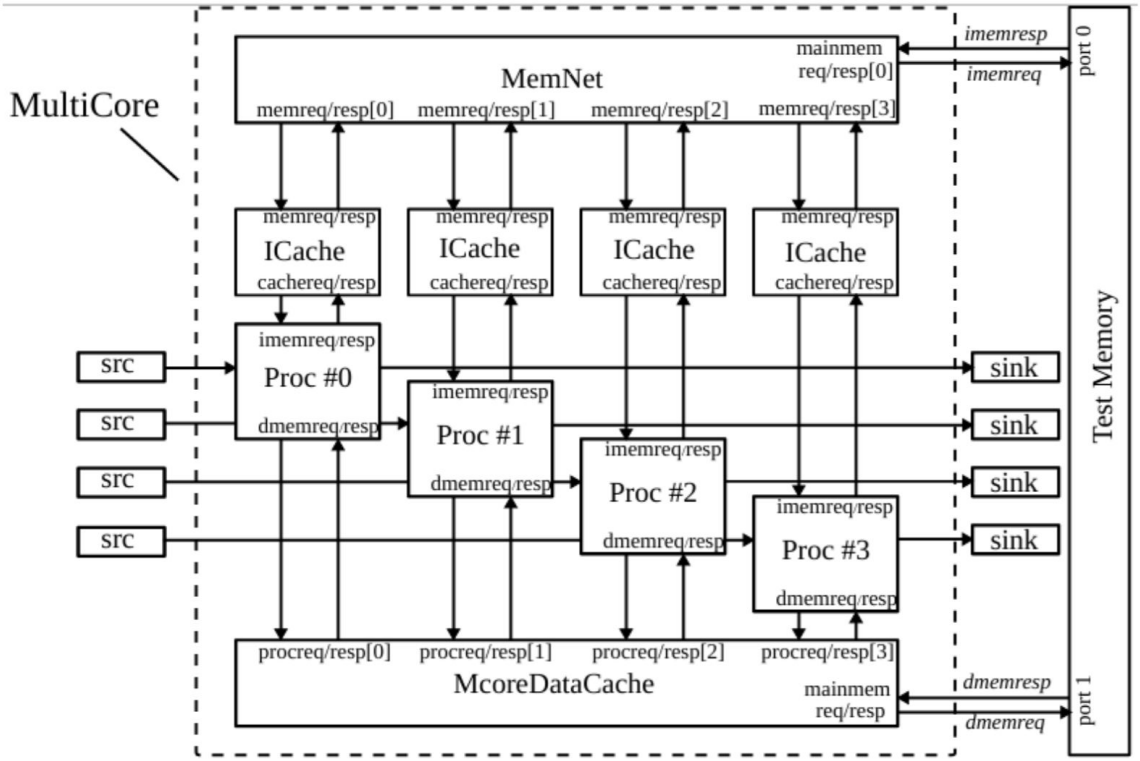


Figure 3.2

Figure 4.1: Single-core instructions-- a representative set

- CSR : `csrr, csrw`
- Reg-Reg : `add, mul`
- Reg-Imm : `addi`
- Memory : `lw, sw`
- Jump : `jal`
- Branch : `bne`

Table 5.1:

category	Evaluation functions									
	vvadd		cmult		mfilt		bsearch		qsort	
	single	quad	single	quad	single	quad	single	quad	single	quad
num_cycles	4311	2260	13471	5208	25989	9797	10341	3831	57129	47284
total_committed_inst	811	1492	2011	2707	5493	6994	2106	2855	10902	34146
total_cpi	5.32	1.51	6.70	1.92	4.73	1.40	4.91	1.34	5.24	1.38
total_icache_miss	5	55	7	64	23	133	15	94	747	893
total_icache_access	952	1646	2112	2866	5836	7483	2393	3216	12498	38456
total_icache_miss_rate	0.0055	0.0334	0.0033	0.0223	0.0039	0.0178	0.0063	0.0292	0.0598	0.0232
total_dcache_miss	76	89	151	168	322	363	149	125	206	747
total_dcache_access	300	408	1000	1109	1312	1562	239	371	4036	7004
total_dcache_miss_rate	0.2533	0.2181	0.1510	0.1515	0.2454	0.2324	0.6234	0.3369	0.0511	0.1067

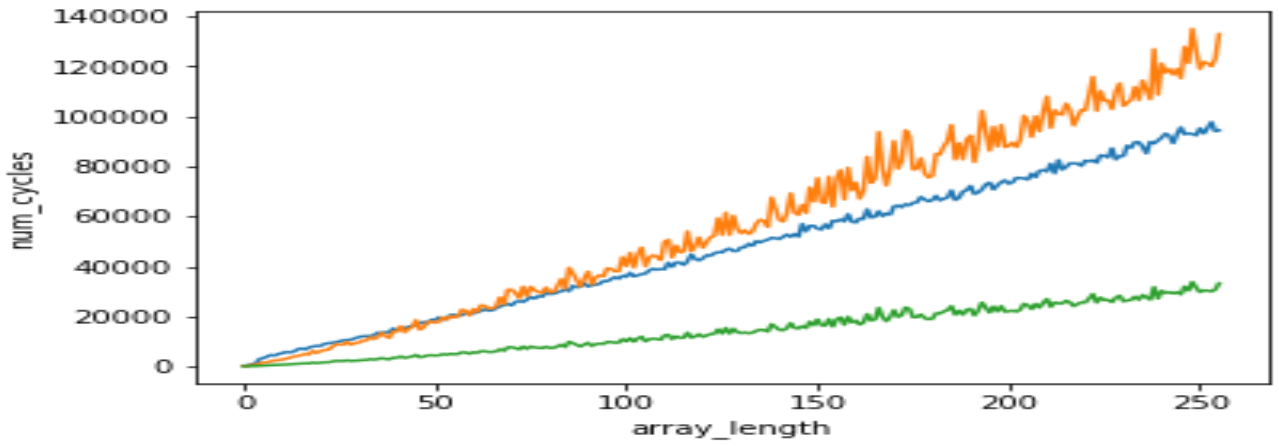


Figure 5.1

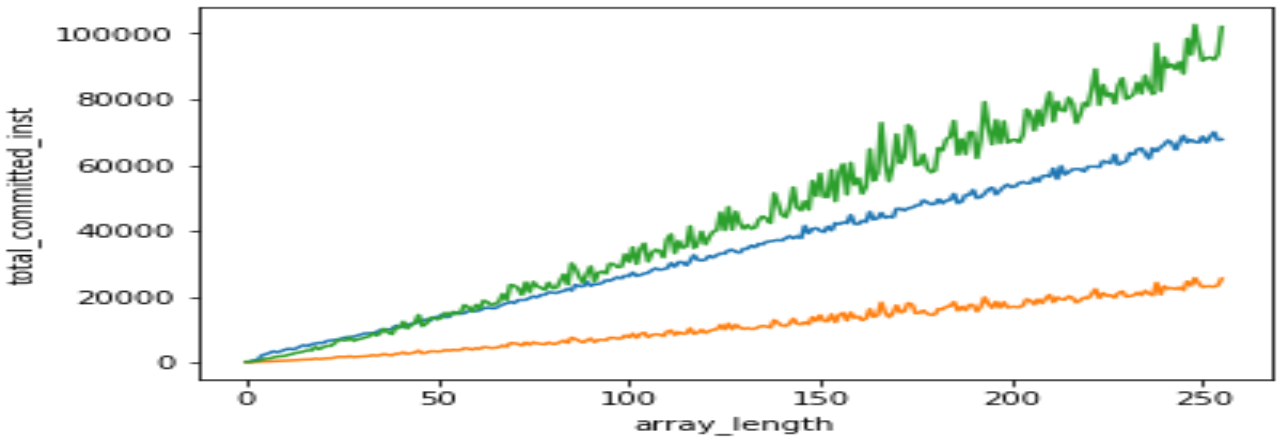


Figure 5.2

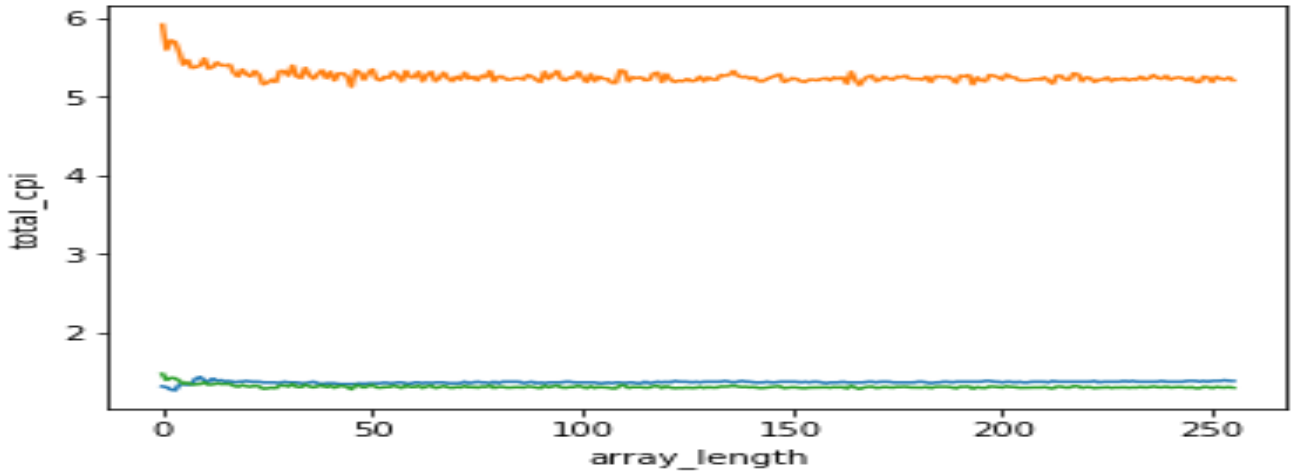


Figure 5.3

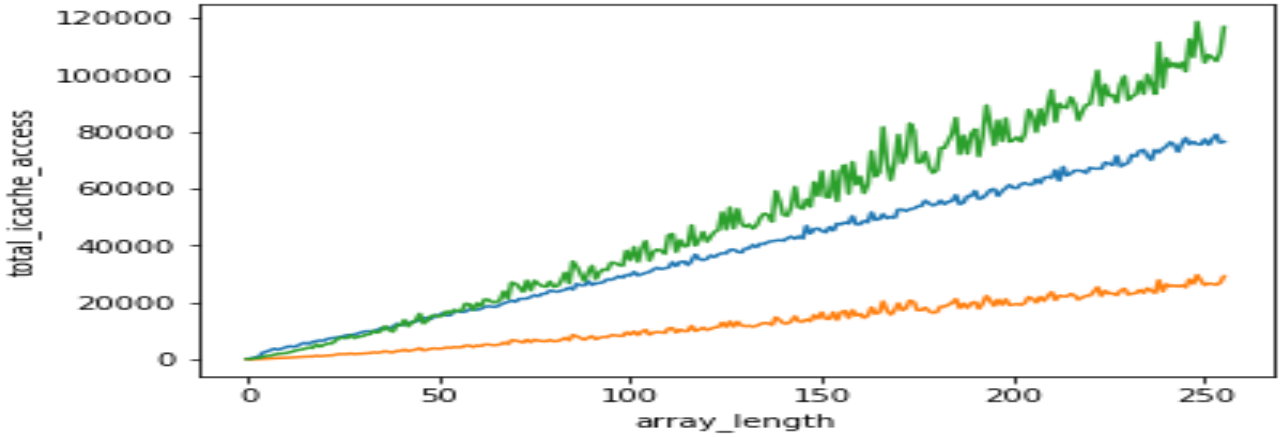


Figure 5.4

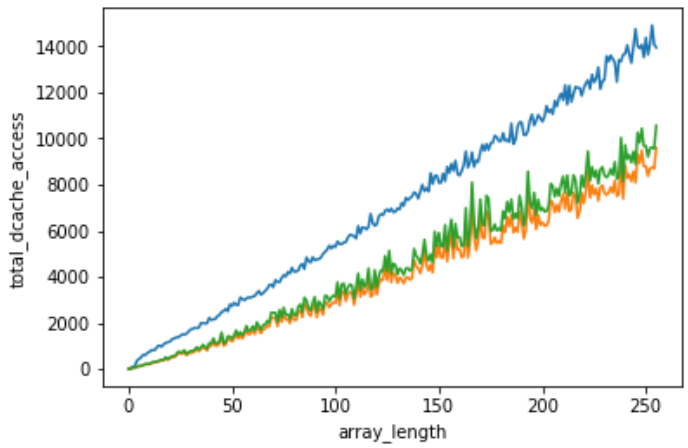

14000
12000
10000
8000
6000
4000
2000
0
total_dcache_access
0
50
100
150
200
250
array_length

Figure 5.5