Eric Huang (eh549)
Jonya Chen (jc957)
Nathan Shen (nds64)

I. Introduction

In this lab, our group implemented a single-core and multi-core system with a combination of processors, networks, and caches. By evaluating trade-offs between these two processor designs after running various configuration benchmarks, we could determine if enabling different program threads to run at the same time on different processors improves performance. In class, we discussed how using this idea of parallelism to take advantage of four processors should result in a theoretical speedup of 4x over the single core. Therefore, we could then run various benchmarks on each design to analyze the benefits and drawbacks from exploiting thread-level parallelism.

For our baseline implementation, we implemented a single-core processor based on components that we created in previous labs. Specifically, we had a bypassing processor (with the addition of a few new instructions), an instruction cache, and a data cache. Our alternative design was to make a few more modifications to implements a multi-core system with four processors, four-banked caches, and a four-node ring network that accounts for multi-banked caches. We utilized an incremental design approach when implementing our system by designing, implementing, and testing our single-core design, before continuing on to the multi-core design. Throughout the coding process, we ran self-checking assembly tests and applications on ISA simulators in order to ensure that our designs were implemented correctly. The goal is that eventually, by comparing results from both processor designs once we run both single and multi-threaded microbenchmarks, we can determine whether or not performance can be improved with parallelism. This overall project is a culminating project utilizing concepts and designs from previous labs throughout the course.

For this lab, we assumed the following roles: Nathan was the design lead, Jonya was the architect, and Eric was the verification lead. Referring to the initial project roadmap that we set (Figure 1), it can be seen that the actual roadmap (Figure 2) resulted in much more parallel and tighter workflow. It can be seen how we found it challenging to begin the project earlier due to Thanksgiving break. To account for varying schedules, we had more parallel tasks so that the team collaboratively helped each other with debugging implementations and generally overlapped while working. Despite the shift in days spent working on the lab, the project was a success. The baseline and alternative design were all completed while passing all of the assembly tests. By constantly communicating throughout the entire project, we could properly coordinate our milestone tasks in order to make sure that everything was completed.

Our alternative implementation was similar to that of the baseline, except that instead we had four processors, rather than just one. The main difference was modifying our old code in order to have banked caches and a four-node ring network. For our multi-banked cache design, we had to utilize an additional bank index field in the memory address to modify the cache index and tag bits used. Our network was also modified in order to properly route messages to the correct destination, as well as use the processor id to send responses back. In order to verify the functionality of our alternative design, we then ran multi-threaded assembly tests to see if our execution resulted in any failed tests. In order to account for this new multi-core system, we needed to add additional hardware for each core and the corresponding overhead set up time.

Comparing the two designs, the multi-core processor is able to perform faster, with a lower CPI, than the single-core processor, due to its ability to split tasks across all four cores. However, a trade-off occurs when we consider the amount of overhead involved with setting up these other cores. This issue explains why we observed that running a single-threaded load performs better on our baseline single-core design, rather than the alternative multi-core design. Otherwise, for the multi-thread loads, the total amount of time to run decreases when running on more cores even with resulting overheads, since work is spread more evenly across all cores and there is a lower load latency on misses.

II. Baseline Design

Our baseline design is composed of a bypassing processor from lab2 and two caches from lab3, one as an instruction cache and the other as a data cache. Because our baseline design is only a single-core, we do not need a network for this design. This is a good baseline design because we can use the design principle of modularity and incorporate as many of these cores as we want for our multi-core design. So for example, if our alternative design was that we wanted 8 cores instead of 4 cores, it would not be that much different to implement it because we already have our baseline single core processor. We implemented the single core configuration similar to figure 3. The cache requests and response ports of the caches connect to the respective instruction cache and data cache ports of the processors. In addition, we had to implement more instructions into our processor. The processor supports a special “stats” bit in order to tell our simulation when to record the stats for evaluation purposes. We also needed to implement jalr instruction which jumps to address and places the return address into the GPR. Lastly, we need to be able to store the number of cores and the core ID in order to run compiled programs.

The quicksort algorithm is heralded as one of the quickest sorting algorithms next to mergesort and heapsort, outperforming other sorting algorithms such as selection sort, bubble sort, etc. In fact, while those sorts have an average run time on uniform random data sets of about n^2, quicksort has an average run time on similar data sets of n log n, showing its speed easily when the amount of numbers to sort gets larger and larger. There are other sorts out there with similar run time on uniform random data sets, but when it comes to using a baseline sorting algorithm that effectively uses all of the data components of our single-core processor while also allowing for the quickest run time when it comes to sorting on a single processor, then quicksort is the best sorting algorithm to use. The other sort that has the same speed, Mergesort, can easily be parallelized, making it more useful in our alternative design which we will discuss more in depth later.

The quicksort algorithm widely used online uses two components, a partitioning component and a distributing component. The partitioning component looks at either end of the array it's given and makes that its pivot. It then starts at both ends of the array and works toward the middle, where one half will be the elements greater than the pivot while the other half is less than the pivot. It achieves this by going to the middle of the array from both ends, and if one number doesn't belong in one half, it swaps it with a number in the other half that also doesn't belong. It then returns the index that separates the array into the two halves. The distributing component gets this index and recursively calls on itself on both halves of the array. We decided to instead merge these two components together into one function to make the code cleaner and easier to read. Because this is a single-core processor, we didn't have to worry about prioritizing the right cores for running the proper sections of our array, and so we simply had the master-core call on the quicksort function on our input array.

III. Alternative Design

When we implemented our alternative design, we utilized modularity and composed 4 of our single core processors that we built for our baseline design for our alternative design. We arranged our multi-core configuration similar to figure 4. The only difference is that we have four instruction caches that are private to each processor and four data cache banks that are shared among all four cores. In order to accommodate for this design, we needed to change our ring network from lab 4 into a four-node ring which was easily done because we used modularity in our lab 4 router node design. In addition, we needed to change the cache index addressing fields because of the inclusion of a bank index for our multi-banked data caches.

As described earlier in the baseline design, mergesort performs as quickly as quicksort, but mergesort can be easily split up to take advantage of our multi-core processor. After seeing how the mergesort essentially breaks everything into halves until it's atomic elements, it's easy to see how we can parallelize this algorithm to make the breaking down of the array faster. For example, if we want to sort an array of size 16 with a single processor, we would need 1 call to break it into two 8 size arrays, two calls to break those two arrays into four 4 size arrays, and so on and so forth. Because it's only one processor making all of these calls, the total cycle time to sort this array will be quite large. However, if we break the input array into quadrants where each quad-core will mergesort one quadrant, then the amount of cycles to sort this array will be significantly smaller. In our previous 16-entry array, each processor will only need to sort a 4-entry array, and so each processor will only have to make 3 calls to break down their section, and since all of the processors are running in parallel, then the multi-core will sort this much faster than the single-core.

We achieve parallelizing mergesort by getting the size of the input array and splitting it into quartiles. We simply spawn the next three cores and designate each of those cores to work with the array's quartile after the previous core's quartile. This makes sure that each of the cores sort a different section of the array, maximizing parallelism. At the end, we merge each quartile together using the same method (compare each element of both arrays and add them into a separate array in order), making sure that only the master core is doing the merging to prevent any redundancies or extra cycles spent merging the array when the master core is already merging the array.

It's clear that a multi-core system is able to perform faster than a single-core system by being able to split a task across its cores. However, when it comes to tasks that cannot be split, such as single threaded loads, it's clear that it will not perform better than a single-core processor; the reason is because the alternative design has a lot of overhead to set up the other cores. In the end, because it's a single threaded load, the multi-core processor can't split the work across the other cores, thus having only one core perform the load, just like the baseline design.

However, it is quite opposite for multi-threaded loads. Because loads may take many cycles to perform during a miss, being able to split multi-threaded loads across multiple cores running in parallel would decrease the total time to perform these loads. If this same multi-threaded load was instead done on a single-core, there would be a higher total number of cycles to perform this load since this single core has to do all of the work, including going into memory on cache misses to get the data.

Even with the overhead to set up a multi-core system, running a multi-threaded load on a multi-core system easily decreases the total time to run since the work is spread out more evenly with each processor finishing their load in parallel.

Although the multi-core system can perform better than a single-core system on parallelizable instructions, there is a limit to the increase in performance with the increase in cores. The more we add in cores, then the more we have to split up the work to utilize each core. Also, the more cores we add, then the more hardware we have to add along with the extra overhead time to set up that many cores and spreading the work to maximize all of the cores. There will be a saturation point for how much work we can split up among the cores, and after that tipping point, adding in more cores for the extra cost of hardware and overhead setup versus the extra benefit in performance will be marginally indifferent. As described in earlier lectures though, having fewer cores would cause a decrease in performance efficiency since more cores will be under higher stress to perform the instructions. This means that the relationship between performance and cores will be logarithmic; there will be a saturation point of performance versus the number of cores.

Despite some of the nuances necessary to utilize all of the cores in a multi-core system, it is still worth investing in its complexity. There is a caveat though; as described above, there comes a point when the returns in adding in cores begin to diminish. That means that we should invest in multi-core systems until we get to the point where adding in extra cores will not have that much of an impact on the performance. Even though we will have extra overhead to utilize the extra cores, it is still worth the investment, especially when our system needs to run thousands and thousands of instructions. Being able to run a large amount of instructions in parallel will decrease the total amount of time to compute all of the instructions.

IV. Testing Strategy

One of the main testing techniques that we've learned throughout the course of this semester is incremental test-driven design. By determining the simplest component to implement and thoroughly testing it before adding other complexities, we can ensure that each design is working and carry out a more efficient debugging method. For example, when we started working on our baseline single-core system, we had to modify our lab 2 processor to support more instructions. For each additional instruction, we made sure to create test cases to unit test our processor. Using self-checking assembly test, we ensured that our single-core system worked as expected. First, we compiled our program natively to model it similarly as if we were to compile on our processor. Once we made sure that our application worked natively, we then ran it on an ISA simulator, which is similar to the functional-level models we used in previous labs, to do any further debugging. Finally, we could then run on RTL model. By taking this incremental testing approach, we can gradually make sure our design passes all tests before moving on. In situations where we had to debug any failing tests, we then used other testing techniques that we learned in class like line traces and waveforms. By viewing waveforms, we could check each signal to make sure the values were what we expected in each cycle. By viewing line traces, we could ensure that instructions were being sent to the correct cores for execution and check to see which state processors were in. Once we used the above techniques mentioned to test our single-core baseline system, we then did the same method for testing the functionality our multi-core processor. Therefore, we could ensure that we have sophisticated and working design to simulate a real cache and network system.

In order to test the single-core and multi-core processor, we ran different types of assembly tests. We began by running parc-v1 tests, followed by parc-v2, and then finally mt tests. By making sure each of these instructions work for each of our designs, we can then test our implementation on our apps, ie. vv-add, quicksort, sort. In order to ensure that all four cores, eight caches, and three networks work together, we analyzed our line traces as well as the register file traces. As mentioned earlier in the baseline design, because we made our single-core processor modular, we are able to exploit our design and not need to test it in isolation. Tests that successfully work on a single-core processor will also work on a multi-core processor because the multi-core processor can simply use just one core to run all of the instructions, but that defeats the purpose of using a multi-core processor over a single-core processor. We want to take advantage of the parallelism, and so that's where the mergesort comes in. By splitting the arrays that the four cores should sort and using line traces and also printing out the results of the array before merging would show if the four cores are all being fully utilized properly. When we print out our input array right before we merge all of the arrays together, we should see four quadrants of the array, all sorted, but only within each quadrant. This would show that each processor were utilized and sorted within their area in parallel. Printing out our input array as a processor sorts it showed that each processor was sorting within the quadrant of the array assigned, thus ensuring that our four cores were being utilized properly, and when we saw that the numbers were also sorted within each quadrant, then the computation within each core was also being used properly too.

When we wish to check if our tests work, we use self-checking assembly tests for this specific lab. As mentioned in the lab handout, the benefit of using self-checking assembly tests is that we do not have to tinker with test sources and test sinks like in previous labs. Not only that, but it does not convolute the output with results for each and every single assembly test and instead outputs a fail or a success if our code passed all of the tests or failed one or more tests. This means that we can test mass amount of instructions and not have to code as many sources and sinks which could be a tedious task. Instead, the self-checking assembly tests will verify all of the basic arithmetic operations for us and also even check the address of the PC for jumping and branching instructions too.

However, there are also cons to self-checking assembly tests. Because self-checking assembly tests can verify our results without us having to code the individual test sources and sinks, that means that self-checking assembly tests require a lot more instructions to be implemented, making incremental design slightly harder. We cannot unit test if we implemented one instruction correctly at a time since there's no way of testing if that instruction is implemented correctly without implementing other ones to take advantage of self-checking assembly tests. That means when we wish to test our instructions and one of them fail, it will be harder to determine which instruction we implemented fail. Another con of self-assembly test is trying to figure out which part of the processor fails. As mentioned in the lab handout, it says that self-checking assembly tests will simply tell if we either pass or failed instead of sending individual results. What if our processor is passing some addiu tests but failing others? It is hard to tell which specific tests we are failing, thus making it harder which part of our hardware or software needs polishing.

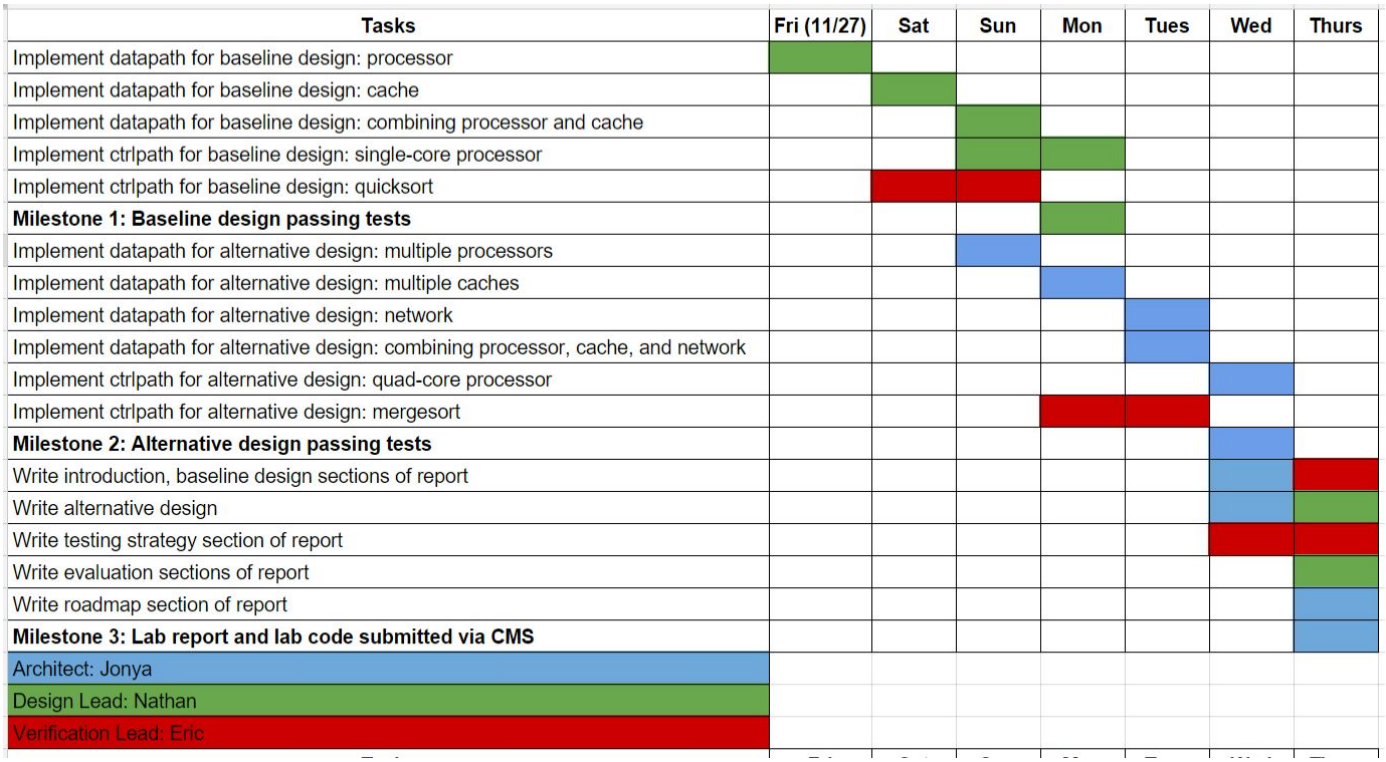
V. Evaluation

Looking at Code Snippet 1 and 2, we see a few different ways to implement an assembly sequence for our C program, vv-add. From first glance, our hand assembled unoptimized vv-add code from lab 2 is more readable. However, when we analyze the code, the hand assembled optimized vv-add from lab 2 gives the best performance. This is because it avoids stalling by using software register renaming. The trade-off from achieving this better performance compared to the other assembly sequences is that it uses more registers, and therefore more hardware and area. The compiler generated assembly sequence attempts to optimize both the performance and number of registers used. As a result, the compiler generated assembly sequence is more optimal in terms of static code size and the number of registers used. The compiler generated assembly sequence contains less register than the hand assembled optimized vv-add, but it has more register than the unoptimized vv-add. Likewise, the static code size for the compiler generated assembly sequence is greater than the optimized vv-add but less than the unoptimized vv-add. Therefore the compiler generated assembly sequence trades performance for a smaller static code size and less register used for the more optimal assembly sequence.

Analyzing the performance of the different benchmarks on the single-core and multi-core, we notice that the CPI is significantly lower for our multi-core design (Chart 1, Chart 2, Table 1). If we assume that both designs have the same cycle time, we can use the number of cycles as a proxy for the performance and compare them between the two designs. For the vv-add benchmark, we see that the performance(num_cycles) is better for the multi-core design than the single-core design, which is as expected! However, looking at the number of cycles for the sorting benchmarks, we notice that the multi-core actually performs worse than the single-core! This seems surprising at first until we realize that there are overheads to make an application multi-threaded. This overhead is translated into the extra number of instructions for the program. As we can see in the table, the multi-core design takes more number of instructions to execute the same C-code. But because it is multi-core, we are able to run multiple instructions in parallel and still minimize our number of cycles and have an improvement in CPI! In order to get a feel for the amount of overhead to make a program multi-threaded, we ran the single threaded vv-add microbenchmark on the multi-core alternative design and recorded the results in table 2. When we compare the CPI of the single threaded vv-add on the single-core design and the multi-core design we see that it takes an average of 3 CPI of overhead. Although we expected to get a 4X speedup, it is inhibited by the overhead, and therefore we only get a speedup of about 3X with regards to the CPI.

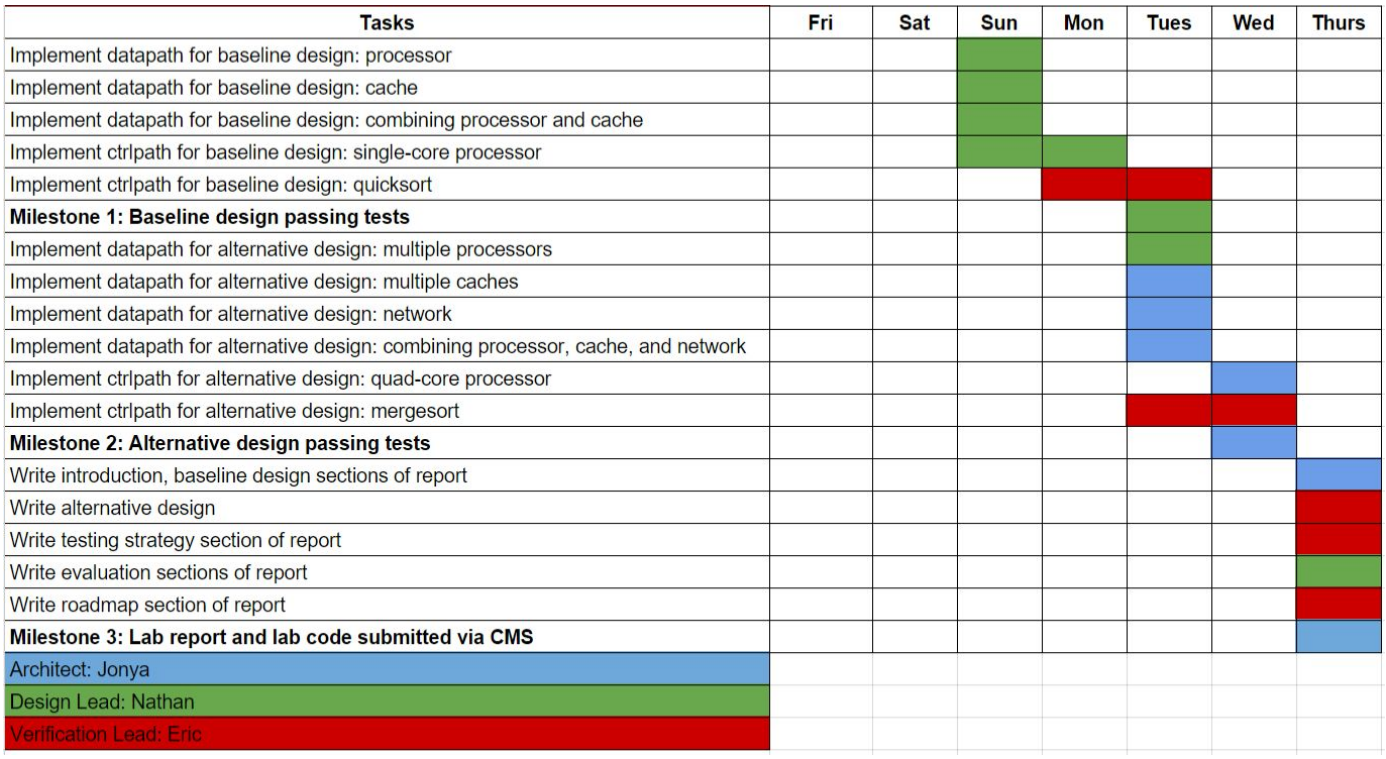
Even though in our test cases, the multi-core sorting performed worse than the single-core sorting, this is not always the case. For example, when we looked at the sort.dat, we noticed that the number of elements we had to sort was only 128 elements. If we increase the number of elements, we should theoretically see better performance for our multi-core. In addition, we implemented two different sorting algorithms and is only a rough estimate of the performance of the implementations. For the single-core, we implemented quicksort and for the multi-core we implemented mergesort. So therefore a better comparison would be if we implemented mergesort for our single-core implementation and compared those results. We should expect that for the same sorting algorithm, if the data set is large enough, our multi-core implementation should have better performance.

VI. Appendix



Tasks	Fri (11/27)	Sat	Sun	Mon	Tues	Wed	Thurs
Implement datapath for baseline design: processor							
Implement datapath for baseline design: cache							
Implement datapath for baseline design: combining processor and cache							
Implement ctrlpath for baseline design: single-core processor							
Implement ctrlpath for baseline design: quicksort							
Milestone 1: Baseline design passing tests							
Implement datapath for alternative design: multiple processors							
Implement datapath for alternative design: multiple caches							
Implement datapath for alternative design: network							
Implement datapath for alternative design: combining processor, cache, and network							
Implement ctrlpath for alternative design: quad-core processor							
Implement ctrlpath for alternative design: mergesort							
Milestone 2: Alternative design passing tests							
Write introduction, baseline design sections of report							
Write alternative design							
Write testing strategy section of report							
Write evaluation sections of report							
Write roadmap section of report							
Milestone 3: Lab report and lab code submitted via CMS							
Architect: Jonya							
Design Lead: Nathan							
Verification Lead: Eric							

Figure 1: Our initial roadmap and projected timeline



Tasks	Fri	Sat	Sun	Mon	Tues	Wed	Thurs
Implement datapath for baseline design: processor							
Implement datapath for baseline design: cache							
Implement datapath for baseline design: combining processor and cache							
Implement ctrlpath for baseline design: single-core processor							
Implement ctrlpath for baseline design: quicksort							
Milestone 1: Baseline design passing tests							
Implement datapath for alternative design: multiple processors							
Implement datapath for alternative design: multiple caches							
Implement datapath for alternative design: network							
Implement datapath for alternative design: combining processor, cache, and network							
Implement ctrlpath for alternative design: quad-core processor							
Implement ctrlpath for alternative design: mergesort							
Milestone 2: Alternative design passing tests							
Write introduction, baseline design sections of report							
Write alternative design							
Write testing strategy section of report							
Write evaluation sections of report							
Write roadmap section of report							
Milestone 3: Lab report and lab code submitted via CMS							
Architect: Jonya							
Design Lead: Nathan							
Verification Lead: Eric							

Figure 2: Our actual roadmap and timeline

Eric Huang (eh549)
Jonya Chen (jc957)
Nathan Shen (nds64)

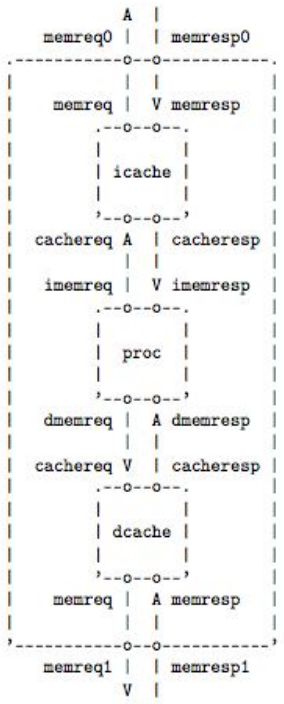


Figure 3: Baseline Single-Core Block Diagram

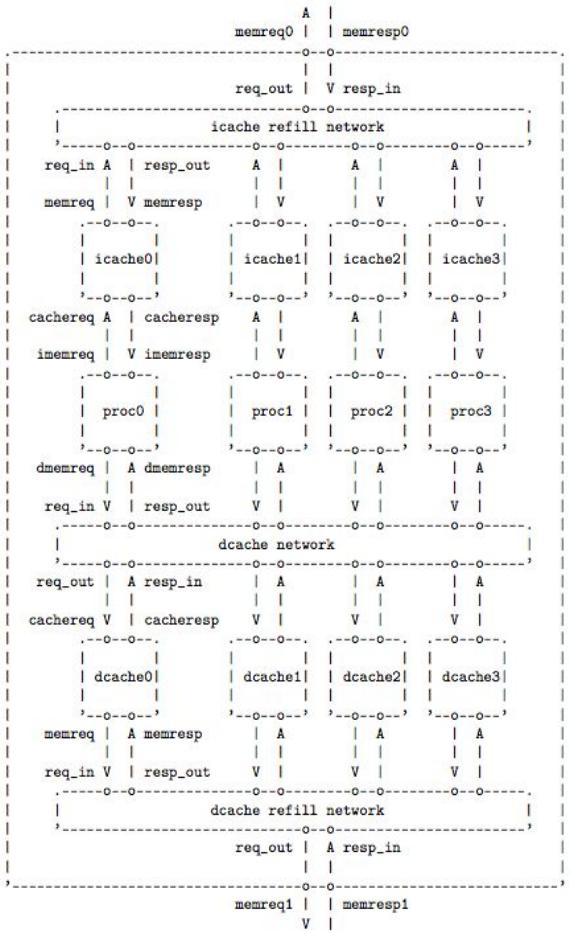


Figure 4: Alternative Multi-Core Block Diagram

Eric Huang (eh549)
Jonya Chen (jc957)
Nathan Shen (nds64)

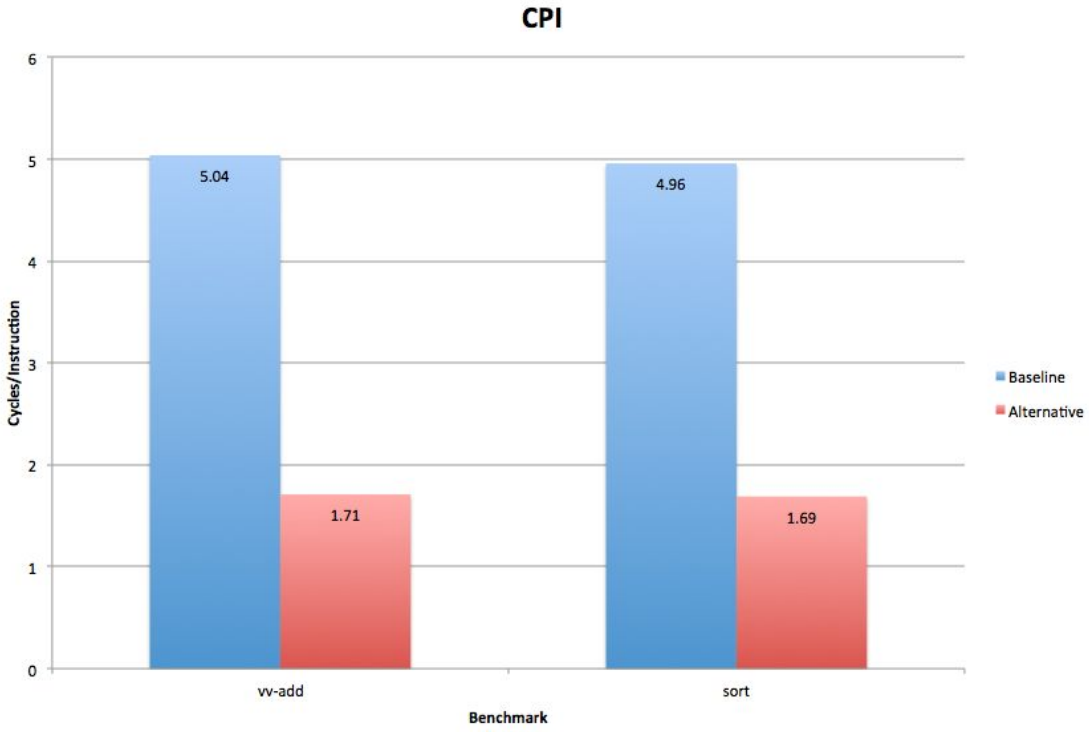


Chart 1: CPI for the two different benchmarks on our two design

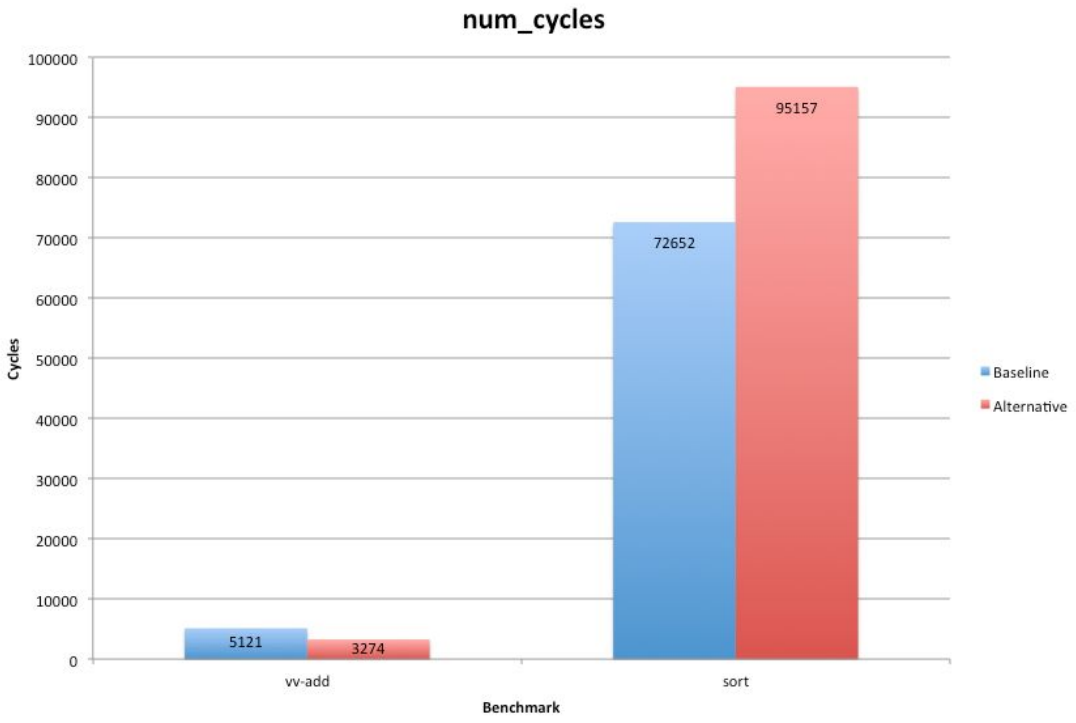


Chart 2: Number of cycles for the two different benchmarks on our two designs

Benchmark: vv-add			
	Baseline	Alternative	Speed Up
num_cycles	5121	3274	1.56
num_instr	1016	1910	
CPI	5.04	1.71	2.94

Benchmark: sort			
	Baseline	Alternative	Speed Up
num_cycles	72652	95157	0.76
num_instr	14646	56376	
CPI	4.96	1.69	2.94

Table 1: Evaluation Results for our single-core and multi-core processor

Single Threaded vv-add on multi-core design	
num_cycles	8865
num_instr	1016
CPI	8.73

Table 2: Overhead in the software to make vv-add multi-threaded

```
1    addiu r5, r0, 0
2  loop:
3    lw    r6, 0(r2)
4    lw    r7, 0(r3)
5    addu  r8, r6, r7
6    sw    r8, 0(r4)
7    addiu r2, r2, 4
8    addiu r3, r3, 4
9    addiu r4, r4, 4
10   addiu r5, r5, 1
11   bne   r5, r1, loop
```

```
1     addiu r5, r0, 0
2   loop:
3     lw    r6,   0(r2)
4     lw    r7,   4(r2)
5     lw    r8,   8(r2)
6     lw    r9,  12(r2)
7     lw    r10,  0(r3)
8     lw    r11,  4(r3)
9     lw    r12,  8(r3)
10    lw    r13, 12(r3)
11    addu  r6, r6, r10
12    addu  r7, r7, r11
13    addu  r8, r8, r12
14    addu  r9, r9, r13
15    sw    r6,   0(r4)
16    sw    r7,   4(r4)
17    sw    r8,   8(r4)
18    sw    r9,  12(r4)
19    addiu r5, r5, 4
20    addiu r2, r2, 16
21    addiu r3, r3, 16
22    addiu r4, r4, 16
23    bne   r5, r1, loop
```

Code Snippet 1: Hand assembled assembly instruction from lab 2. vvadd-unopt (left). vvadd-opt (right)

```
1   1168:   18e0000c    blez    a3,119c <vvadd_scalar(int*, int*, int*, int)+0x34>
2   116c:   00001021    move    v0,zero
3   1170:   00001821    move    v1,zero
4   1174:   00c24821    addu    a5,a2,v0
5   1178:   00a24021    addu    a4,a1,v0
6   117c:   8d2a0000    lw      a6,0(a5)
7   1180:   8d090000    lw      a5,0(a4)
8   1184:   24630001    addiu   v1,v1,1
9   1188:   00824021    addu    a4,a0,v0
10  118c:   01494821    addu    a5,a6,a5
11  1190:   ad090000    sw      a5,0(a4)
12  1194:   24420004    addiu   v0,v0,4
13  1198:   1467fff6    bne v1,a3,1174 <vvadd_scalar(int*, int*, int*, int)+0xc>
```

Code Snippet 2: vv-add assembly instructions compiled from C program