Computer Architecture — Lab 4 Multi-Core System
ECE 4750 Fall 2022 — Shihao Cao (sfc72) Brandon Wood (bpw42)

1 Introduction

This lab serves as a conclusion of the course, it exercises less on the actual verilog code itself, and more on higher level concepts such as systems integration, testing, and even crossing into the software domain. The goal was to implement a fully integrated processor using our own variable latency multiplier, pipelined processor, and cache. For the alternative design, we add additional complexity to have a multicore processor connected to the caches and memories using ring networks. By concluding the course with a final composition, we not only learn about the final component that modern computing relies upon, that is networks, but we also gain even more experience in the art of composition. It puts continued focus on the merits of incremental design and testing. In this lab report, we dive into how we built the best processors we could, as well as evaluate the trade space of multi-core designs, and explore the work required (both software and hardware) to take advantage of available parallelism in software workloads.

2 Alternate Design

There are two aspects to our alternate design for this lab: a quad-core processor (that leverages the cache, processor and multiplier code we've written before) and a multi-threaded implementation of merge sort in c to run on this processor. Our quad-core processor uses two compositions of our ring network to compose communication between the cores and the instruction caches and the data caches, respectively. Our software uses the course's implementations threading in tinyrv2, as well as some basic implementations of malloc to distribute a sort across four cores!

As we approach the limits of Moore's law and it becomes less and less beneficial to increase the complexity of processors, we've begun to look for new ways to increase processing time. One way of doing this is, through coordinating with our software colleagues, splitting the work over many cores. Implementing this composition is not easy. We have to manage communication between the processors and the same bank of memory, manage spawning tasks (in software), and manage communication between all the processors and the instruction memory. This may be worth while when our other option for speed up is adding a ton of complexity, which may be even worse or may be untenable in terms of area or energy.

The quad-core processor composes four of the processors from lab 2 (and by extension, four of our multipliers from lab1), and eight of our caches from lab three. Each processor has a private instruction cache, and all four instruction caches communicate with the single main memory port via the instruction cache network. In order to avoid issues with memory coherency, the four data caches are formed into a bank and the communication between the four processors and the bank is managed by the data cache network (see lab handout). Both the data cache network and the instruction cache network are compositions of the ring network unit that we designed (see lab handout).

We implement the ring networks used for this via a composition of routers, which are in turn a composition of route units and switch units. Each route unit implements the simplest algorithm we could get away with: passing all packets to the left if they do not match the router id, and passing it out if it does. Our switching algorithms implement a similarly rudimentary algorithm whereby they give highest priority to the left-most incoming packet, then the right-most and finally new packets. This, while simple, is elegant because the static priorities of incoming traffic are selected so that a steady throughput is maintained.

As an example, consider router 1 in a ring network. Assume the clock strikes and three packets are received, one from the left going to terminal 1, one from the right going to terminal 3, and one from terminal 1 going to terminal 0. Each route unit now has a choice to make: where to send these packets. Two packets will go to the left, the ones going to terminals 3 and 0. Even though terminal 3 is to the right of our router, it still will get pushed to the left in service of our simple algorithm. On the other hand, we get lucky when sending packet 0 to the left, because it takes the shortest path it its destination. The route units make their decision and signal that their outputs are valid to the appropriate switch units (another example of the val/ready uprotocol!). Two switch units now have a decision to make. The switch unit that controls the left exit will now have a choice between the packet coming from the right into terminal 3 and the packet coming from terminal 1 to terminal 0. Since priority is given to those packets already in circulation, it switches the packet from the right into the left exit. The switch gating the input to the terminal of the router receives the packet from the right and puts it into terminal 1's input, since there is nothing else trying to leave that way.

By controlling packet flow between multiple terminals in this way, we can ensure that multiple processors, say, can communicate with multiple data caches without both sending a message at the same time. The network allows the processors and the caches to not have to manage their own communication. The downside of this is some added complexity (measured in time, blood, sweat, tears, etc.) and some additional area, though the complexity and time-wasting of implementing the bus inside the processor and having each processor stall while other processors use the same bus is more painful than implementing this network.

This network uses the design pattern of modularity, because it is composed of many smaller sub units (which also helps with our unit testing and incremental design approach). It also facilitates modularity by allowing the processors and caches to communicate without taking into account the overall structure of the system. We also leverage encapsulation by encapsulating the routing algorithm and switching algorithm in separate sub-units inside the routers, then connecting up as black-boxes and making them adhere to an interface. We establish a hierarchy within these algorithms, by giving static priority to those packets already in the network. We use hierarchy when we designate one of the regularly repeating cores to be the main processor that spawns work for the others (this is an example of a design principle that spans both software and hardware since both need to be in on this decision). We also use regularity when we repeat each router within the ring networks, and each ring network in the networks for the data network and the instruction network.

2.1 MT Sort

Compared to our baseline single threaded merge sort, the single-threaded design for multithreading is not so different. The first main difference is to split the working array into 4 chunks to work across. Each core will indepenedently sort its own chunk in its own thread.

Then once all cores are done with its chunk, we then core 0 will merge sort across the left 2 chunks, then the right 2 chunks, then lastly perform one more merge. This means we are fully parallel for all but the last two merges.

The other difference is that the other cores cannot dynamically allocate memory. To handle this, we dynamically allocate once on core0, and then pass pointers to that allocated memory within the `arg_t` structs passed to the worker core. Despite merge sort being recursive, we leverage the fact that we know that the merging buffer will be used by at most one `mt_merge_sort_helper` function at a time.

Just like the

In the below diagram, you can see how we first sort individual chunks, and then have core 0 perform the remaining single-threaded work.

3 Optimizations

In our processor, as we discussed in class, the biggest bottleneck was not having single cycle hit latency on our indstruction cache. Every kind of instruction/transaction must be loaded from program memory so this is a bottlenock that is pervasive throughout our machine.

Since our alternative cache was an FSM based design, every transaction would need to pass through the `INIT`, `TAG_CHECK`, `DATA_ACCESS` and `WAIT` states.

f

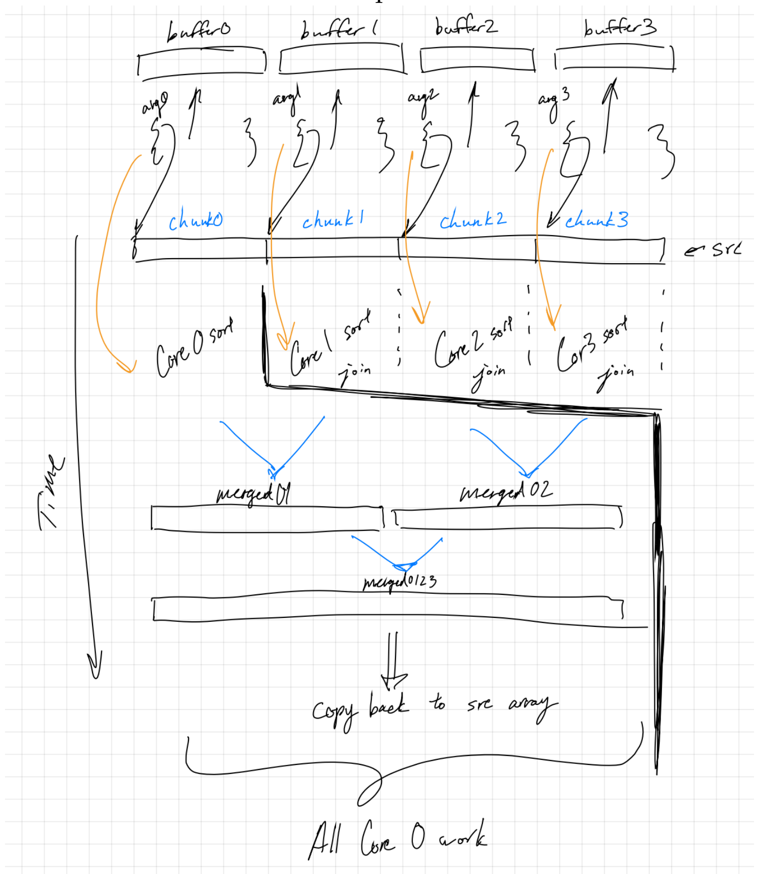

Figure 1: MT Merge Sort breakdown

Per Professor Batten's advice on EdStem, we began by first merging the read data access states into `TAG_CHECK`. We did this by picking our `saved_target_way` to read out of in `TAG_CHECK`, and enabling the data array read early.

Next, in order to merge `WRITE_DATA_ACCESS` into `TAG_CHECK`, we conditionally control the `dirty_bit_wen` signal based on whether or not we're doing a write transaction or doing a refill update (to clear the dirty bit. We also bypass the read data result early to the data register so that data can be read immediately.

Since data reads and data writes now occur in `TAG_CHECK`, we added many more state registers such as `is_initial_hit` to save whether or not this is the first or second time we've arrived back at `TAG_CHECK`.

The more complex and nuanced optimization was removing the `WAIT` and `IDLE` states on a hit path.

We first make sure that on a hit, we send our response early in `TAG_CHECK` combinationally and also set the `proc2cache_respstream_val` bit high. Then, the `WAIT` state should be bypassed if the `proc2cache_respstream_rdy` signal is simultaneously high, meaning the response was sent, and we can go straight to `IDLE`.

We can

4 Testing Strategy

For this design, we approached testing with the goal in mind of creating a set of tests that, once passed, provide a compelling argument that our hardware and software function correctly. To do this, we use a combination of directed/random testing, whitebox/blackbox testing, and unit/integration testing for both software and hardware. We do not use adhoc testing for hardware because of the complexity of the system (we wouldn't be able to look at the line traces and discern much meaningful information), but used a few adhoc software tests as we developed our sorting algorithms.

We developed both hardware and software incrementally and sought to develop directed unit tests (both whitebox and blackbox) by spanning the input space (see specifics below for what we mean by this) in as few tests as possible. While we believe that these tests were designed creatively and to hit as many corner cases as we could, we developed random testing cases that spanned the same input space to potentially generate combinations of inputs that we couldn't think of.

For our hardware systems (ie; the single core system and the multi-core system), this model of parameterizing the input space didn't make as much sense. In particular, we found it hard to "parameterize" the use of an ISA. So, we instead sought to first check that each instruction works, then that simple combinations of instructions work, then that complicated combinations of instructions work, then that devious combinations of instructions that stretch the ISA work. We can then do some random testing based on these tests.

For each subsection of hardware or software we finished along this process, we identified all possible input parameters, and tried to develop directed tests that would span each of those parameters. Then, we wrote random tests that would combine these parameters in new ways to potentially generate new cases that we didn't think of (giving us some insurance in case out directed testing turned out to be non-comprehensive). We focused primarily on black box testing, because, first and foremost, we wanted to demonstrate that our "box" (whether it be hardware or software) adheres to whatever specification we set out to meet (whether that be the ISA for the processor system or the sorting correctness of the sorting algorithm). However, we used whitebox testing both for helper functions in our software and to generate directed test cases. For instance, we might want to hit every condition of a case statement in a piece of c or verilog.

We packaged these tests into unit tests (using the PyMTL3 frameowrk and Pytest) so they could be run repeatedly and automatically. Infrastructure provided to us by the course staff provides a continuous integration interface that allows us to receive real-time feedback on all of our tests whenever we push to github. As we progress through development, we're receiving clear feedback automatically and are notified immediately if anything we developed previously in the incremental design process breaks.

4.1 Hardware Testing

When testing our single core system composition we used many of our tests from lab2. Those tests were already heavily comprehensive, so we were satisfied with what we had. In addition, we pulled in cache tests for the data and instruction caches. See lab 2 and lab 3 for more detail on those tests, respectively.

Our alternative design introduced multiple cores. In order to integrate these cores, we developed, incrementally, a ring network with a very simple routing and switching algorithm. We began our testing by unit testing our Route Unit and Switch Unit. This proved to be incredibly valuable as this reminded us to double check to check for the val and rdy bits in the micro protocol between the input and output ports. Without this, our switch unit would have dropped inputs if they were received at the same time.

We used unit testing for the route units, switch units, routers, and nets, respectively. Each was tested in a fairly similar way, so it isn't worth describing an example for each. However, for each, we check the routing algorithm works as intended for small quantities of packets going into the unit from one terminal. We then increase the number packets. Then, we add more source terminals at the same time. Then we add more source terminals and more packets. Then, we add some random testing on a small number packets to one destination, then many random packets, then many random packets with a lot of delay time. We also made sure to make the delays short and long. For the router in particular, we added tests that touched every input port going to every output port with random delays. We also added a test that "slammed" a single input port with a bunch of packets going to a single destination, and parameterized that on each port. We also pulled off a selection of more difficult tests and ran those against each router id. We think these tests span the possible inputs, spanning quantity, timing, delays, and messages. We still parameterize these tests and provide some "full random" unit tests.

We started to run out of time, and used the tests given to us by the staff for out ring network. We comment that these achieve the goals of our own testing strategy by spanning the input space of possible inputs, delays, timing, and quantify of inputs, and parameterized them into some random tests. The same is true of the data cache tests and instruction tests.

Our quad-core testing was able to leverage much of the testing that we did for the single core system to verify it's basic functionality. Since each of these processors are pulled directly from lab 2, we trust most of their functionality based on the testing and subsequent code revisions we did for that lab. However, there are new interfaces introduced in the multi core system and we sought to test those first by generating common use cases (as in the case of the for loop for the multi-core system memory test). We didn't have a lot of time to do this and we we're super comfortable with the new specifications for memory instructions, so we think our tests for the current system are lack luster compared to what we could have done, given more time.

4.2 Software Testing

Our software testing approach didn't differ much from the hardware testing, aside from doing some additional adhoc testing. This is because the same philosophy of "developing an argument in favor of our functionality" applies to both and both are readily separated into nice functional units that can be tested individually. We do not have anything analogous to our hardware system testing like we did for the software, since our code was designed to evaluate the hardware performance, and those were just functions (an analogous system might be an operating system or something that composes many different functions).

The software testing approach is modular just like the hardware testing approach. Instead of trying to get one giant sort function to work on the first try, we first break it down into helper functions. Then we can write unit tests on each of the helper functions. To test our merge sort we first implemented tests for our helper functions `merge` and `merge_sort_helper` For our recursive helper function, we made sure to test the base cases, then a simple case, then the edge cases. To test more potential edge cases of the sort, we decided to add duplicate values as well as negative and zero values. Lastly we finished off single core testing with randomized testing across all ints, on multiple different sizes of arrays as well.

As a final stage of software testing for the multicore version, we added even more tests hitting the number of elements in the array to stress the sections of code that would partition the work into chunks in a white box testing manner. Then to catch bugs such as memory leaks, we relied heavily on the infrastructure that was already provided in `mtb-sort-eval.c`

5 Evaluation

We evaluate the single core and quad core systems against typical use cases. Namely, we test a vector-vector-add (vvadd) instruction that takes the inner product of two arrays, a complex multiplication (cmult) instruction that multiplies two sets of complex numbers, and a filter (mfilt) instruction that applies a linear filter to an image, a binary search algorithm (bsearch), and a merge sort algorithm. Two versions of each of these are presented, one for the single core processor with no parallel capabilities and one for the quad-core processor that can run on one to four cores. The comparative performance, measured in cycles per instruction is show below for each benchmark. These benchmarks cover a wide range of use cases, and demonstrate a wide range of instruction, data and computational parallelism over time and space.

It appears that the multi core processor struggled with the same workloads that the single core struggled with, and even though we were able to implement the single-cycle hit latency, it still wasn't able to overcome the software overhead needed to synchronize the four cores. Interestingly, it appears that the quad core struggled disproportionately with the sorting algorithm. We think this may have more to do with our specific implementation, where we implemented something that didn't take into account how expensive the memory operations were, with our simple ring network. We're also causing a number of cache misses by instantiating new arrays, then running over four of them in parallel, which our two-way set associative

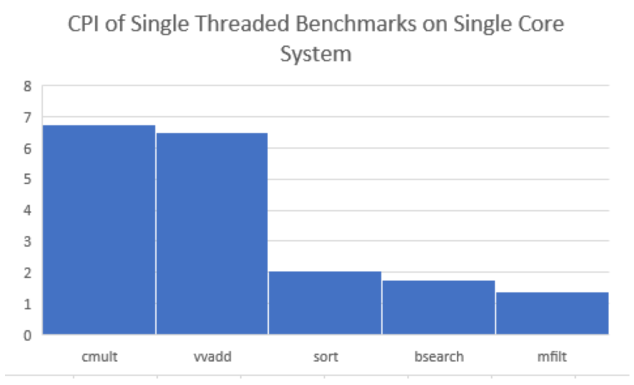

Figure 2: Overview of benchmark performance, measured in terms of CPI.

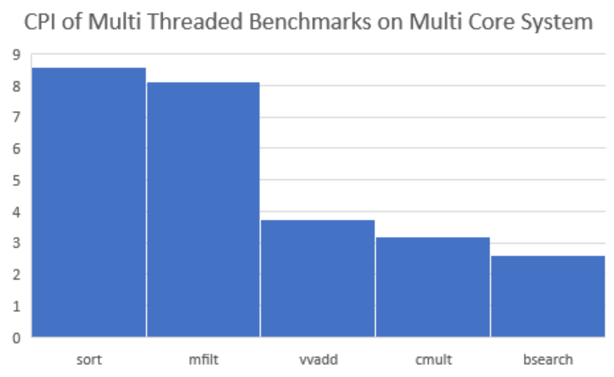

Figure 3: Overview of benchmark performance, measured in terms of CPI.

bank is unable to provide sufficient associativity to handle. We think the implementation of the sorting algorithm could be updated to take this into account and maybe made faster than the single core if given enough time.

We had some issues getting the single core system to run our multi-threaded code, and we're able to get data from it. As a result, the column in the table below is empty. There's still a lot of interesting data in this diagram. This chart shows the CPI performance of each combination of sorting algorithm on system micro architecture. We can see the effect that the software overhead of the multi-threaded algorithm on the performance across both core, and, especially when the single worker on the multi-core processor has to deal with that overhead and is unable to catch up. We do see a speed up due to the parallelism, which is exciting! That's exactly what we would expect. We also expect the single core algorithm to run faster on the single core because of the lower hardware overhead associated with the network latency (both for the instructions and data).

Qualitatively, there's no question that the quad-core will take up more area and gulp up more energy. Any energy wasted in the cache, or processor will be multiplied by four in the quad-core system, and additional energy is needed to push all the packets around the processor. In terms of cycle time, we think it's somewhat of a toss up. The additional hardware in the network is likely not a driver of cycle time, and stands independent of the rest of the system. However, it is the case that the single core processor can be driven harder because of the energy savings. For each cycle, less energy is wasted and the same processor can be given more area, and thereby more thermal relief, so we can drive it faster.

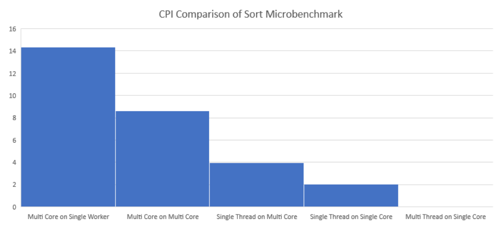

Figure 4: Comparison of software and hardware overheads for the various implementations of software and hardware

6 Conclusion

In conclusion, a fully integrated processor is no easy feat to design, integrate, and test. While out of order processors can take advantage of instruction-level-parallelism, a multicore processor can take advantage of software level parallelism. To implement this though, it requires support obviously from the software level in the form of multi-threaded applications, but also a memory network and cache network to support coherenecy among the cores when working across shared addresses. Most notably, these networks are additional circuitry that consume additional energy, and take up area that both do not contribute to useful work. They also contribute to additional cycle time due to the extra logic, and especially to processor time in general as transaction packets move across networks.

Indeed the tradeoff at hand explored with the multicore design is whether or not these hardware costs are worth the performance that can be exploited through software-level parallelism.

7 Work Distribution

In the beginning, we tried partner programming again, but there was just too much ground to cover for this lab, and it was difficult to find time that overlapped for both of us. Therefore, for this lab, we split up for much of the work, assigning chunks of instructions to take care of. In order to maintain the flow of our work though, we made sure to always test our own interactions that we implemented. Throughout our lab, our work was evenly distributed between us.