Hardware Acceleration for Sorting Algorithms

By: Jonya Chen '16
Advisor: Chris Batten, ECE
Summer 2013

Cornell ECE Early Career Research Scholars Grant
Engineering Learning Initiatives Undergraduate Research Report

ABSTRACT

Over the past 30 years, the performance of microprocessors has improved exponentially, but practical energy/power constraints are posing challenges to sustaining this trend. One solution to continue development is through specialization. I wanted to find a way to quantify how much this idea could improve performance and energy efficiency. I decided to implement a specific sorting algorithm in both software and hardware and compare the results. I created my own hardware coprocessor for the sorting algorithm and mapped it onto an FPGA, which served as a proof of concept model. I focused on utilizing a designed research process throughout the project and applied various testing techniques. Through the summer, I was able to study how performance efficiency can be accomplished in the long run by taking small steps towards specialization.

INTRODUCTION

Over the past 30 years, the performance of microprocessors has improved exponentially, led by technological drivers such as Moore's law, transistor-speed improvements, and new core microarchitecture techniques. However, diminishing transistor scaling benefits and practical energy/power constraints are posing significant challenges to sustaining this remarkable trend. Continuing the trajectory of microprocessor performance improvement would require at least a 30x performance increase by the year 2020 [1]. These scaling challenges and ongoing industry expectations introduce difficulty in achieving performance and energy efficiency. The research community is now concerned with inventing new solutions to help sustain exponential improvement with technological capabilities, so that performance will start to track Moore's law again.

The Batten Research Group is tackling this problem through specialization, which could create a not only energy-efficient but also high performance system. By making a system that can take care of one specific task very efficiently, there is a higher chance of getting improved performance at lower energy. The group does a lot of simulation, but I wanted to get a chance to test the designs on an FPGA development board.

My role was to determine whether specialization could improve performance and energy efficiency or not. I explored this idea by comparing a specific sorting algorithm implemented in software versus hardware through analysis of factors such as performance, area, and cycle time. In order to test my hypothesis, I created

my own hardware coprocessor for the sorting algorithm and mapped it onto an FPGA board, which complements the group's simulation based approach. As a result, I created a prototype that could be used as a real proof of concept in the lab. I explored the variations in performance by evaluating the tradeoffs and iteratively refining my implementation through a designed research process. The project not only helped teach and benefit the group, but also gave me the opportunity to discover how performance efficiency can be accomplished in the long run by taking small steps towards specialization. By the end of the summer semester, I hoped to be able to show through a quantification process the advantages involved in specialization for sorting.

Through this research report, I plan to cover the steps taken in my designed research process. I will explain the implementation behind both the hardware and software sides of my sorting algorithm, as well as the incremental design approach that was taken to complete the project. Testing strategies and an evaluation of the results will be discussed and presented.

BASELINE DESIGN

In order to find a way to quantify how much specialization can improve performance and energy efficiency, I created the project flowchart shown below:

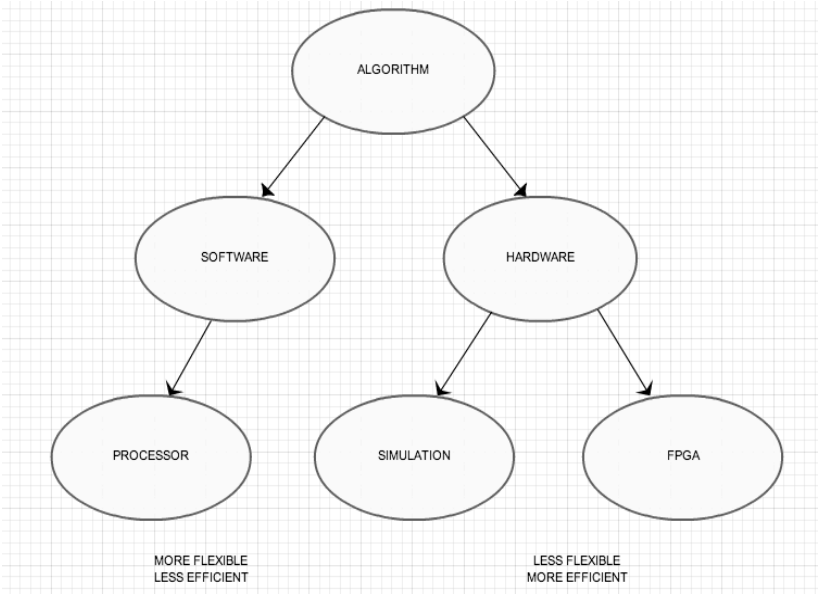


Figure 1

Starting off with a simple sorting algorithm, I would implement it in both software (left side of flowchart) and hardware (right side of flowchart) and then compare results. The key idea is that software implementations are flexible (parameters can be modified to create more variability), but their efficiency isn't as ideal. Hardware implementations are the opposite – less flexible but much more efficient. Therefore, these tradeoffs were analyzed through the comparison of two implementations of the same sorting algorithm.

The initial step taken was to decide on a sorting algorithm that will eventually be implemented in both software and hardware. I started out with a simpler approach by choosing to use bubble sort. Bubble sort is a sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. The pass through the list is repeated until the entire list is traversed with no swaps needed. This indicates that the list has been sorted. Bubble sort has an average complexity of $O(n^2)$, where n is the number of items being sorted.

After defining the algorithm, the baseline design used was a software implementation of bubble sort. The code was written in C:

```
//  Bubble sort algorithm in c
//  Created by Jonya Chen on 8/13/13.

#include <stdio.h>

int main()
{
        int length = 5;
        int mem[] = { 5, 1, 3, 2, 4};
        int temp;
        int i, j, c;
        for (i = 0; i < length; i++ )
        {
                for (j = 0; j < length - 1; j++ )
                {
                        if ( mem[j] > mem[j + 1] )
                        {
                                temp = mem[j];
                                mem[j] = mem[j+1];
                                mem[j+1] = temp;
                        }
                }
        }
}
```

Taking this code implemented in C, I compiled it and ran it natively. Next, I needed to cross-compile the code so I could get some performance estimates through a simulator. I ran the C code through a compiler to produce MIPS code, which is a type of assembly language.

Taking the MIPS code, I was able to run bubble sort on a simulator known as the gem5 processor. The gem5 simulator is a module platform for computer system architecture research and encompasses system-level architecture as well as processor microarchitecture [2]. This served as my general-purpose hardware component. Using the simulator, I timed how long it took to do the sorting, so I could get some quantified data to eventually compare with my hardware results. I used various data sets, including ones that were already sorted, ones that were sorted backwards, and ones with random data. The datasets ranged in all types of sizes, including larger ones so it could help amortize any fixed overheads.

Statistics gathered from the gem5 simulator can be found later in the report, in Table 1 of the Results section.

ALTERNATIVE DESIGN

An alternative design was created to use as a comparison with the software implementation. Using a hardware description language, Verilog, I implemented the same sorting algorithm, bubble sort, in hardware. There were several steps involved with this implementation: design, simulation, and prototyping.

In order to get a better sense of what the overall hardware-sorting accelerator would look like, I created a top-level module, which can be seen here:

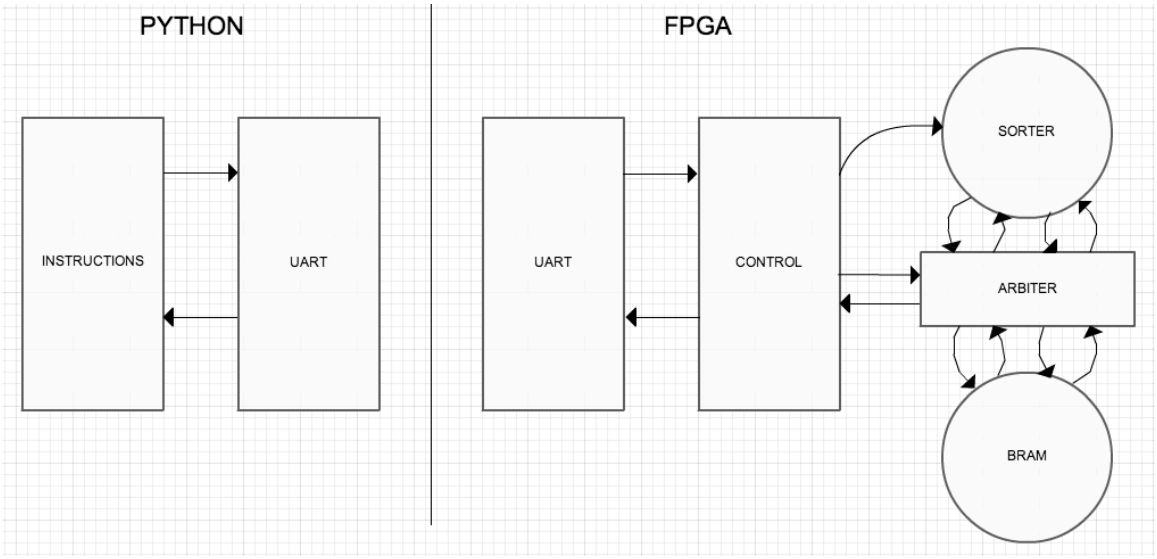


Figure 2

The UART interface noted above allows for the serial communication of messages to the FPGA board. Responses would be generated and transmitted via the UART for all sink events containing all the parameters of the event (sink address, expected value, received value, etc). The corresponding message formats were implemented in PyMTL, a hardware-modeling framework, as wrappers around the original test vectors. The messages will then be used as UART messages to configure the test harness. Once the system is initialized, the PyMTL test harness would send a reset request message over the UART. The control unit is responsible for triggering the reset or beginning the execution of specific modules. A message for transmission can be sent at any time, as there is valid-ready handshaking implemented which will make modules stall if necessary. This is the overall design and plan for interfacing designs onto the FPGA for testing.

The right side of the Figure 2 indicates the sorter that I would be implementing in Verilog. The idea of a sorter in hardware is fairly similar to that of a sorter in software. However, the elements are to be stored in a BRAM (Block Random Access Memory) instead. The only inputs to the hardware sorter would be a start address (to let the sorter know where the elements to be sorted are locating in the BRAM), the number of elements, and the number of times to traverse the

elements until the array is all sorted. In the diagram, the Sorter circle is responsible for receiving data as inputs and swapping the elements when necessary. It would then output a value and send it to the Arbiter, which would decide which memory port to use to store the data. The BRAM circle is the memory that would be used for memory read and write requests. It would also send back memory read and write responses.

The Sorter module design may be fairly complex so I decided to take several preliminary steps to help create a clearer picture of what it would look like. Below is a diagram of my main design for the Sorter, which is responsible for getting data from memory, swapping them if needed, and then storing it back to memory:

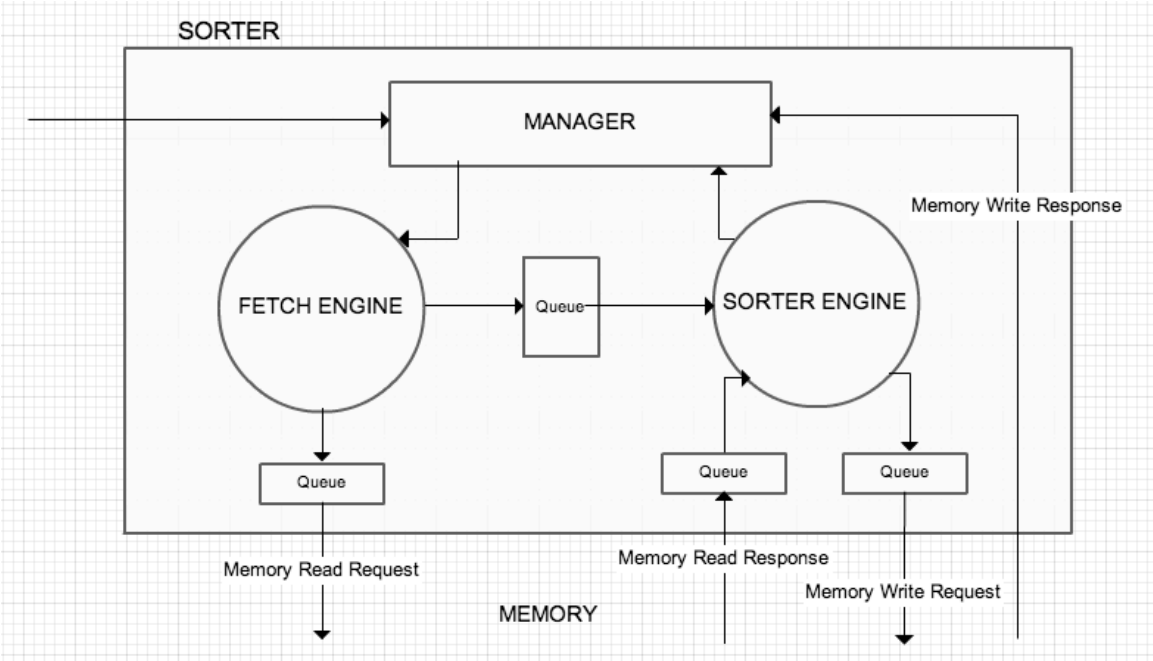


Figure 3

In Figure 3, there are several components. The Manager module is responsible for taking the source messages that consists of several inputs: start address, number of elements, and how many times to traverse the elements. It will then give this information as an input to the Fetch Engine (FE). The FE will take that and generate read requests of elements stored at specific addresses in the memory. Once the memory processes this, it will return the data that was stored at each specific address as a memory read response. This generated data will then move into the Sorter Engine (SE), where the elements will be swapped if needed. The SE will then output a write request to store the data back into memory. The SE will know where in memory to store the data because of an address sent from the Queue going from FE to SE, which is filled with corresponding addresses by the FE. Once the memory has written this new data into the proper address, it will send back a memory write response message. This way, the manager can keep track of the memory write response and determine whether or not it needs to repeat the entire process again.

Following this overall design of the sorter, I then needed to focus on the design of the SE (Sorter Engine). I decided to start out with my software implementation of bubble sort and write the code out in Python first. By compiling and testing that Python code, I could make sure that it worked properly before moving on. This also helped give me an idea of how the overall SE should function.

Afterwards, I moved into creating a hardware design for the SE. I separated the design into 2 main parts: the data path and the control logic. The data path is a collection of functional units, such as multiplexers, registers, or adders, which perform data processing operations. The control logic essentially controls how the data path works. It can have signals such as write enables, multiplexer select lines, and decoder inputs. Below, the datapath is shown in Figure 4, which the control logic is shown in Figure 5.

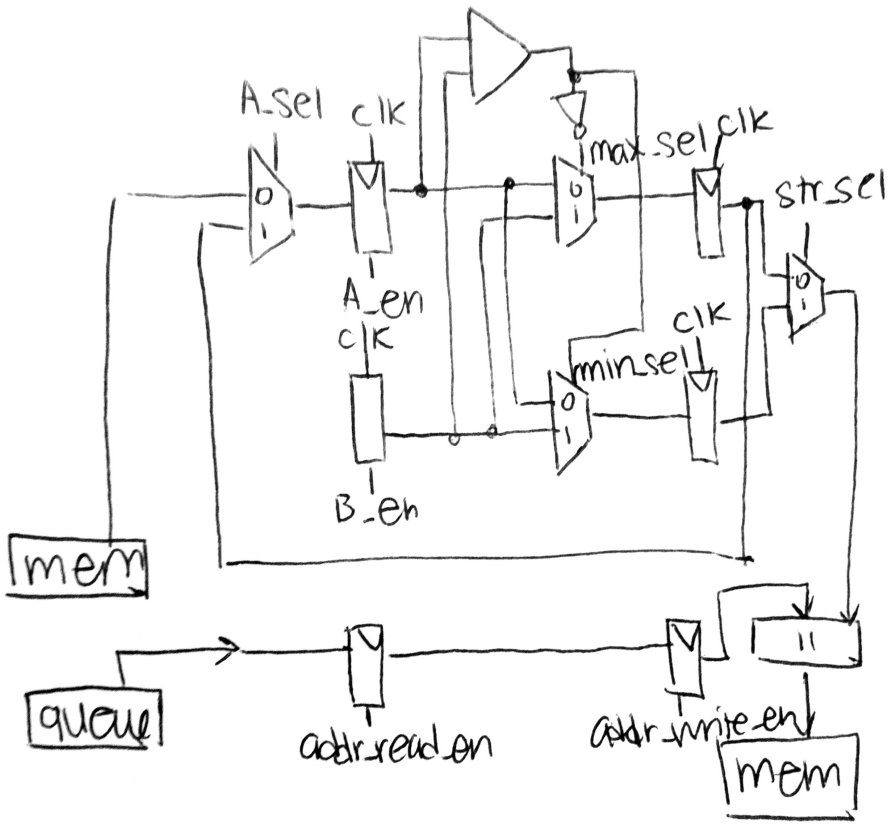


Figure 4

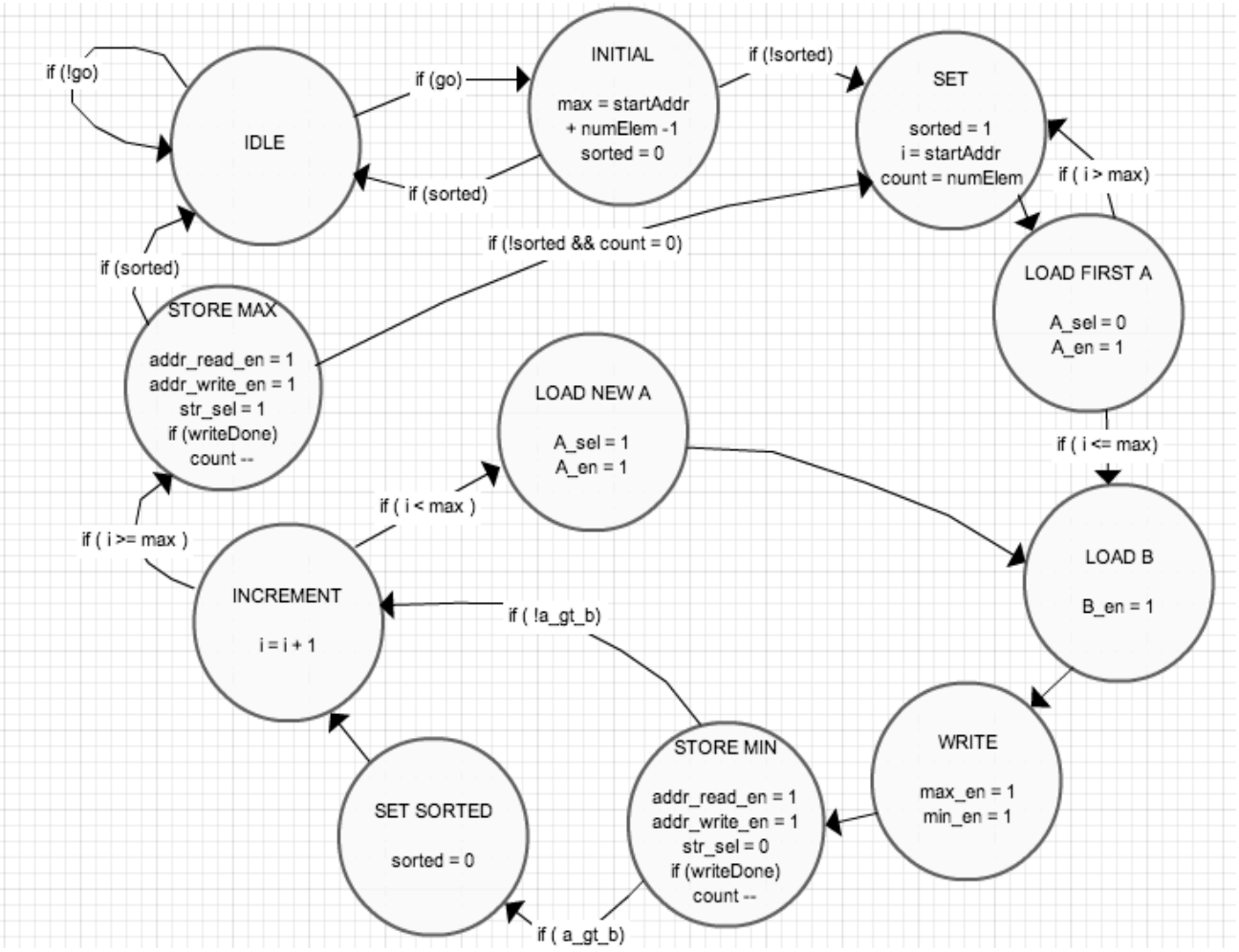


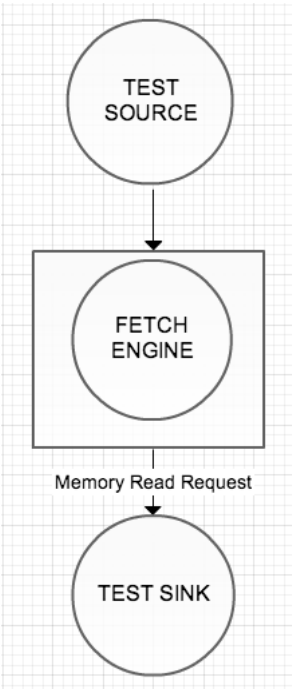
Figure 5

Once I had the entire design planned out for my overall Sorter module, the next step was implementing it using Verilog. I could have gone straight to typing away the code for the entire module; however, I quickly realized that method was not the best approach. One of the main ideas I gathered over this summer was the concept of an incremental design approach. This technique could be applied to all sorts of applications and I chose to utilize it throughout my project.

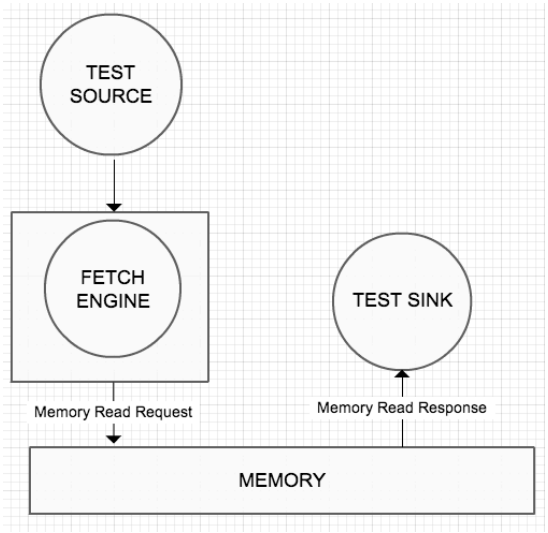
Before I went ahead with work on the sorter, I spent several weeks implementing a complex multiplier. The complex multiplier essentially served as a simpler version of the sorter. By taking the time to write code for this complex multiplier and testing it to make sure it works, I was able to familiarize myself with the entire design process. Once I had finished this preliminary step of implementing a complex multiplier completely in simulation and on the FPGA, I felt much more knowledgeable about which tools to use and steps to take, and could move forward with the sorter.

I learned that an incremental design approach makes it much easier to tackle challenging problems with multiple components. Not only does it make unit testing much easier, but it also allows for much more efficient debugging with code. Using this method, I was able to iteratively refine my implementation, which is key to the overall research process.

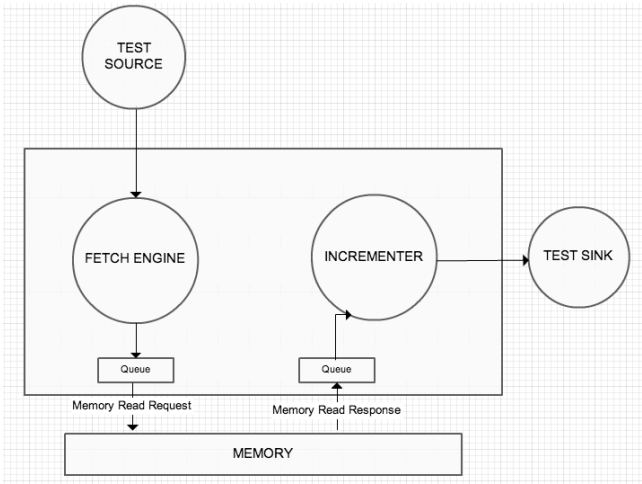
Below are diagrams of the incremental designs I decided to implement:



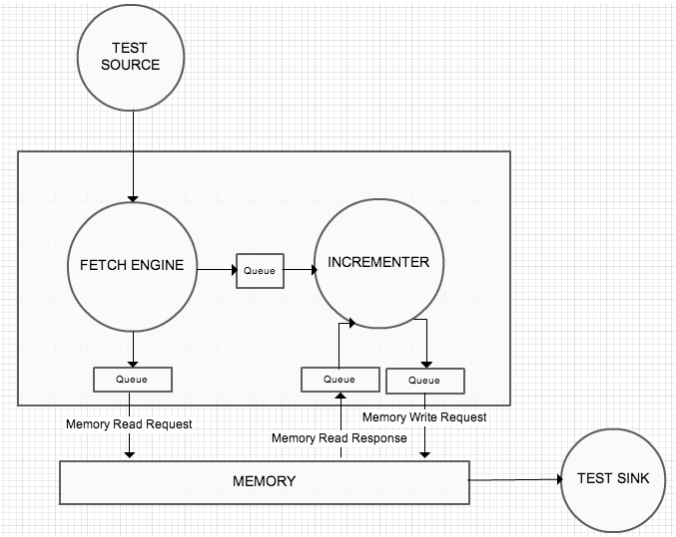
DESIGN 1
Fetch Engine:
Increments addresses and sends out respective memory requests



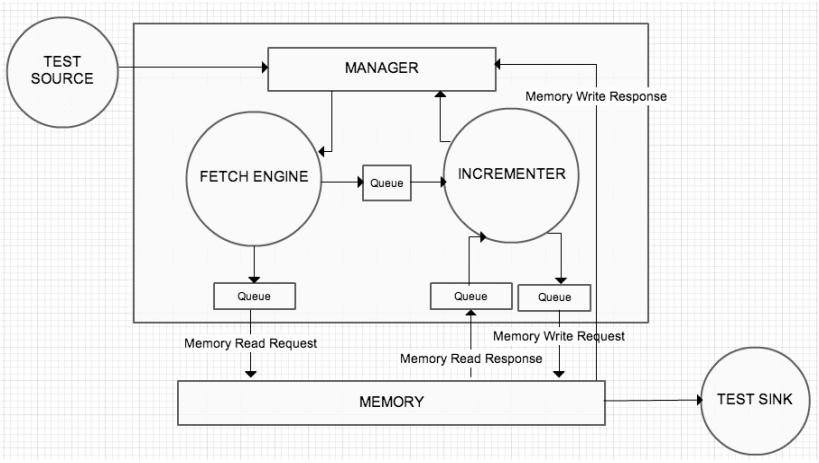
DESIGN 2
Fetch Engine + Memory:
Sends out data from Memory at each respective address



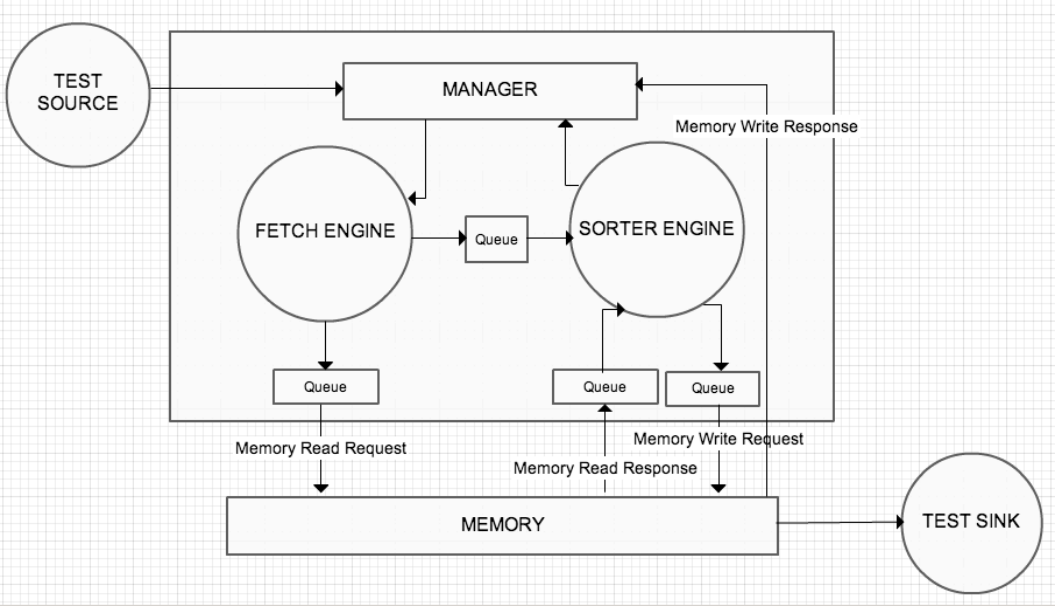
DESIGN 3
Single Incrementer:
Goes through all the elements once and increments each element by 1



DESIGN 4
Single Incrementer #2:
Goes through all the elements once and increments each element by 1 with stores back into memory



DESIGN 5
Double Incrementer:
Has a manager that goes through all the elements twice and increments each element by 1 each time



DESIGN 6 and 7
Sorter:
Rather than incrementing, the Sorter Engine will do the swaps for each adjacent pair of elements.
Design 6 will only go through all the elements once, while Design 7 will swap elements until completed sorted.

Working my way through the list of designs, I would implement each in Verilog and then use a source/sink test so that I could be sure that it worked properly. Throughout the implementations of these designs, I slowly built upon each so that eventually, I would have a completed sorter. As a result, I had a much more systematic and orderly way of knowing my sorter would work in simulation.

Once I had finished the above steps in order to have a sorter implemented, I needed to map the design onto an FPGA. This stage is known as prototyping. With an FPGA, I could get a more detailed analysis of the sorting process. This is where I would use the UART mentioned in Figure 2 in order to send messages over to the FPGA to begin the sorting process.

TESTING STRATEGY

In terms of methods of testing my modules within the sorter, there were various strategies used in the process. First off, one of the main reasons as to why I decided to use the incremental design strategy mentioned before is because it not only mitigates design complexity, but also creates a much more efficient testing procedure. It allowed me to be able to test each design and verify its functionality.

One of the key testing methods I used was unit testing. There are various types of unit testing and all came in handy when I wanted to see if a module could work properly in isolation. One of these types was cycle-by-cycle testing, which is a lower level testing method for modules. I can specify what happens and verify given inputs and outputs. Another type of testing I used throughout the project was source/sink testing. I utilized this method with each of the incremental designs

mentioned above. I could put a bunch of messages as inputs to a test source and also put a bunch of messages as outputs in a test sink. The test source would feed its messages to whichever module I wanted to test. Then, at the end of the function, the output from that module would be compared to the expected outputs in the test sink. If they matched up properly, then the test succeeds. If not, the test would fail and I would know to go back and fix the code. Overall, unit testing was extremely helpful with finding problems early in the development cycle.

Also, for each incremental design step, I created corresponding tests that can still be individually run, regardless of whether or not they are used in later design implementations. For example, I can still go back and run the test of the Fetch Engine by itself to verify that it works properly. This method proved to be very effective, especially since I knew I could always go back to previous working designs in case if I had incorrectly implemented a future step.

Some modules that didn't function in isolation were more difficult to unit test, so instead, I utilized integration testing. For example, this was the case with my Manager unit. As I gradually built upon each incremental design step, I created new test cases in order to make sure that the new implementation would all work together.

RESULTS AND EVALUATION

For my software of implementation of bubble sort, I ran the code through a gem5 processor and got statistics on how many cycles it took to do the overall sorting process. I did this several times, switching up the lengths of the randomly generated array of elements.

Dataset [length]		Number of Cycles
mem[5]	Random	7976
mem[10]	random	10861
mem[20]	Random	22672
mem[50]	Random	106503
mem[100]	Random	405995
mem[500]	Random	9874152
mem[1000]	Random	39583256
mem[1000]	Ascending order	33000262
mem[1000]	Descending order	45987298

Table 1

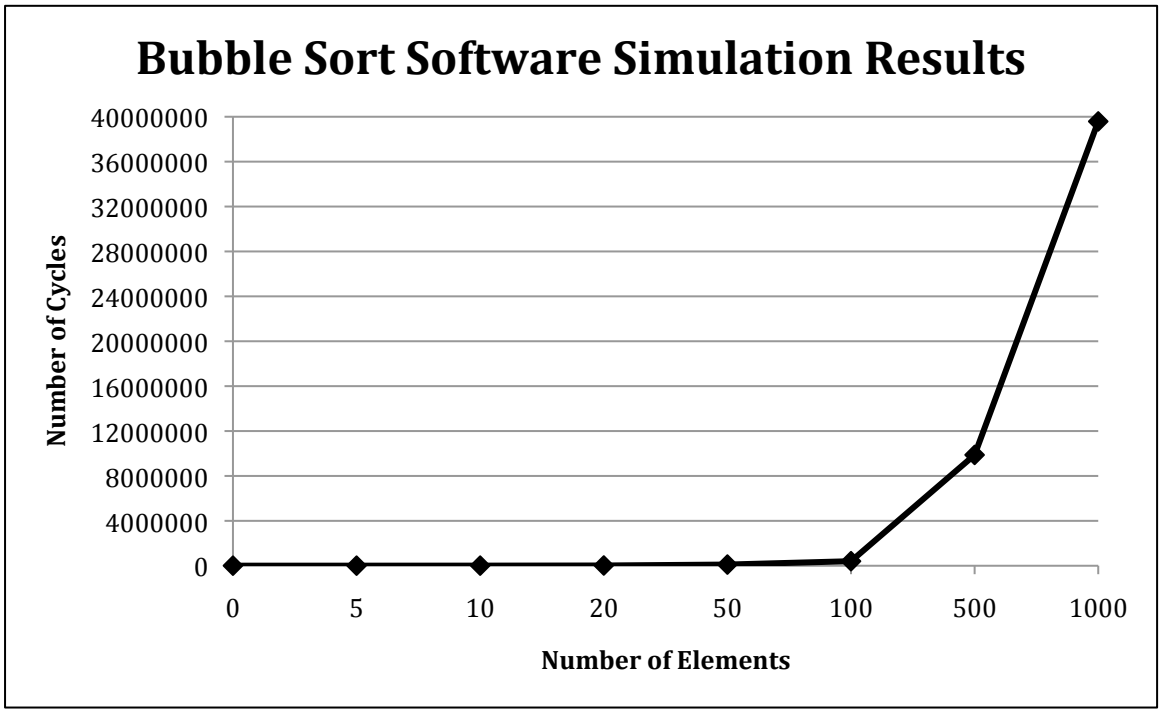
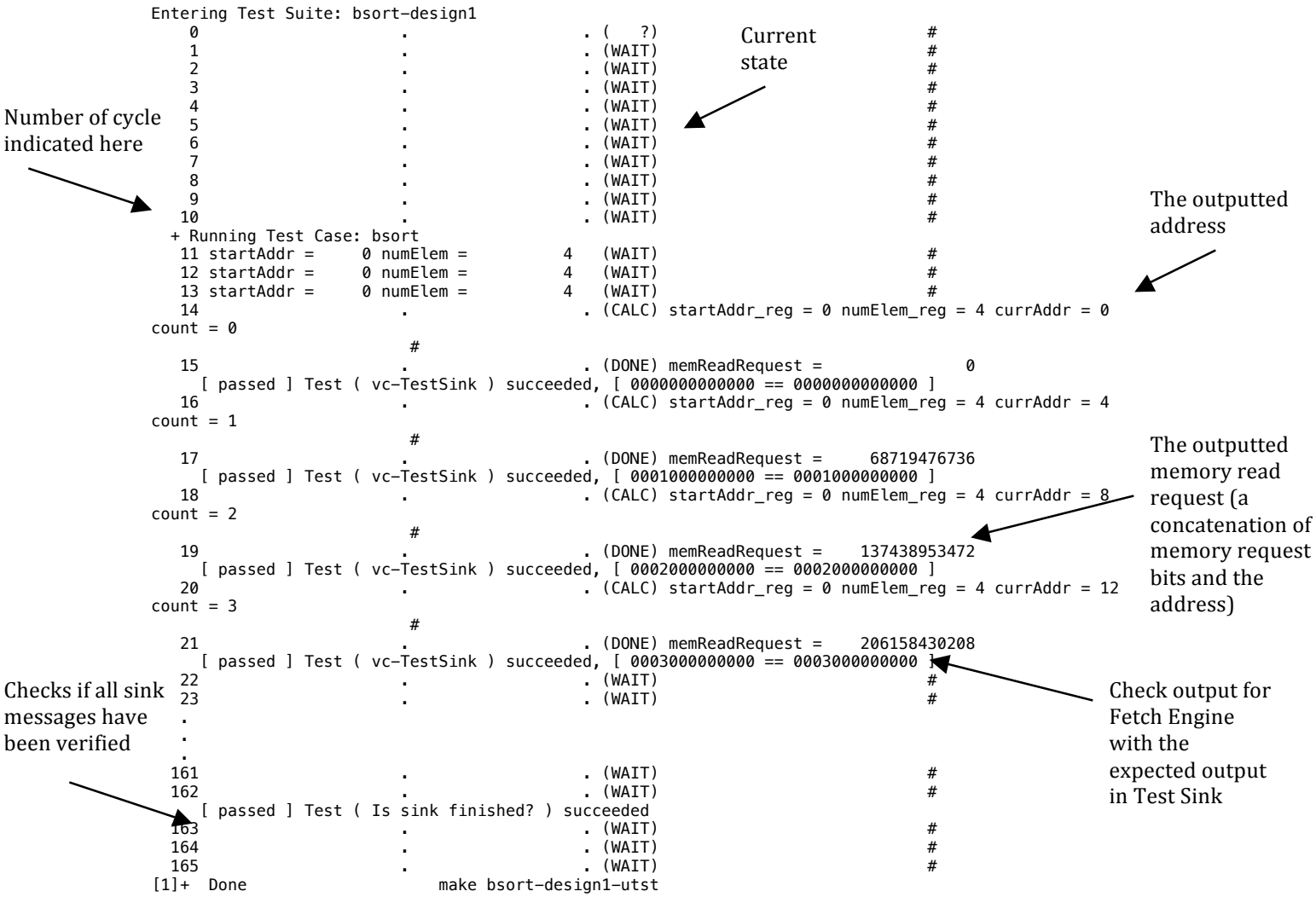


Figure 6

As Table 1 shows, the number of cycles increased as the length of the dataset increase. Furthermore, Figure 6 is a graph created with the data and it can be seen that the number of cycles increases exponentially with respect to the length of the dataset. This trend makes sense with bubble sort, since it has an average complexity of O(n^2) and a best-case complexity of O(n). In the c code for bubble sort, it can be seen that the inner for loop does O(n) work on each iteration and the outer for loop also runs for O(n) iterations. Therefore, the total work should be O(n^2). By graphing with respect to the number of elements, we can expect to observe a parabolic growth curve.

In terms of creating the hardware comparison, due to time limitations this summer, I was not able to gather the overall expected results. I am currently in the middle of the incremental design approach for getting Verilog code down for the bubble sorter. I was able to get several hardware designs working in simulation, with proper testing in isolation. For example, the Fetch Engine is able to output proper memory read requests to memory. I used line traces as a method of understanding performance as well as a debugging tool. Line traces allow me to see what's going on inside the unit by printing lines each cycle. Below is a line trace of my Fetch Engine, which outputs addresses. In this example, the given inputs are startAddr = 0 and numElem = 4:



CONCLUSION

Overall, this summer, I gained so much knowledge about working with a designed research process. One of the main ideas that I gathered throughout my time on this project was creating an iterative design method. I find it much more efficient and extremely helpful when it comes to debugging code. Not only can I apply this idea to my work on this project, but I can also use this on everyday work.

In terms of future work, I hope to finish the hardware implementation of my overall sorter design and be able to map it on to the FPGA. I could then focus on optimizing the design to get better results and be able to gather some quantified data to compare with my software data.

There are many other options for further expanding my research project. For example, rather than just implementing bubble sort, I could try out various other sorting algorithms. It could be interesting to see what results would be given through a sorting algorithm with a time complexity of $O(n \log n)$ or $O(n(\log n)^2)$.

By gathering the concluding data from my hardware accelerator, I would be able to quantify and determine whether or not specialization can improve efficiency.

REFERENCES

[1] S. Bokar and A. Chien, "The Future of Microprocessors," CACM, 2011.
[2] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi4, “The gem5 Simulator”.
[3] Palnitkar, Samir. *Verilog HDL: A Guide to Digital Design and Synthesis*. Mountain View, CA: SunSoft, 1996. Print.

ACKNOWLEDGEMENTS

I would like to give many thanks to Professor Chris Batten for supervising me this entire summer and giving me the incredible opportunity to be able to work in his research group.

Also, thanks to Edgar Munoz for his assistance and advice throughout the project, as well as Chris Torng for his help with setting up the processor to gather my software results.

Finally, thanks for ELI (Engineering Learning Initiatives) and the ECE Early Career Research Scholars program for giving me an opportunity to gain experience with undergraduate research this summer.