ECE 437L Final Report

Brian Rieder (mg267)

Pooja Kale (mg273)

TA: Nick Pfister

9 December 2016

1 Executive Summary

The purpose of this report is to provide a comparison of the pipeline processor with and without a cache hierarchy and the pipeline processor with cache and a multicore processor. To compare the program execution speed of the three different CPU designs, we will look at the estimated synthesis frequency of each design, the number of instructions executed per clock cycle, and the latency of one instruction. The comparisons will be further made by looking at the FPGA resources for required for our design using Total Utilization Combinational Functions on each design and the utilized Dedicated Logic Registers. Finally, an ultimate comparison will be completed through the computation of the speedup from the original pipeline.

The pipeline without caches is designed with five stages: instruction fetch, instruction decode, execute, memory, and write-back. This allows the design to process five instructions simultaneously within different stages of their execution. The original pipeline design does not have a cache system, forcing the design to always access RAM memory. The pipeline design with caches has a level 1 cache hierarchy consisting of a direct-mapped instruction cache and a two-way associative data cache. This allows fast access to addresses that are stored within caches one level above the memory. The final multicore design includes a coherence control unit that is utilized to maintain a coherent address space for all processors and to ensure that for any address all processors see the same data. This allows multiple instructions to be processed at the same time without concern of caches convey conflicting data.

The three designs will be tested using a merge-sort Assembly algorithm with a parallelized implementation used for the dual-core design. The benefit of utilizing the merge-sort algorithm for testing is that it allows for a computationally intensive test that utilizes caches to a large degree due to its extended runtime and memory accesses as well as its inclusion of multiple branches to test branching logic and resolution in all three designs. Due to its complexity and ability to be parallelized, testing with merge-sort is an adequate representation of a "common case" algorithm and is indicative of generalized performance of the processors.

2 Processor Design

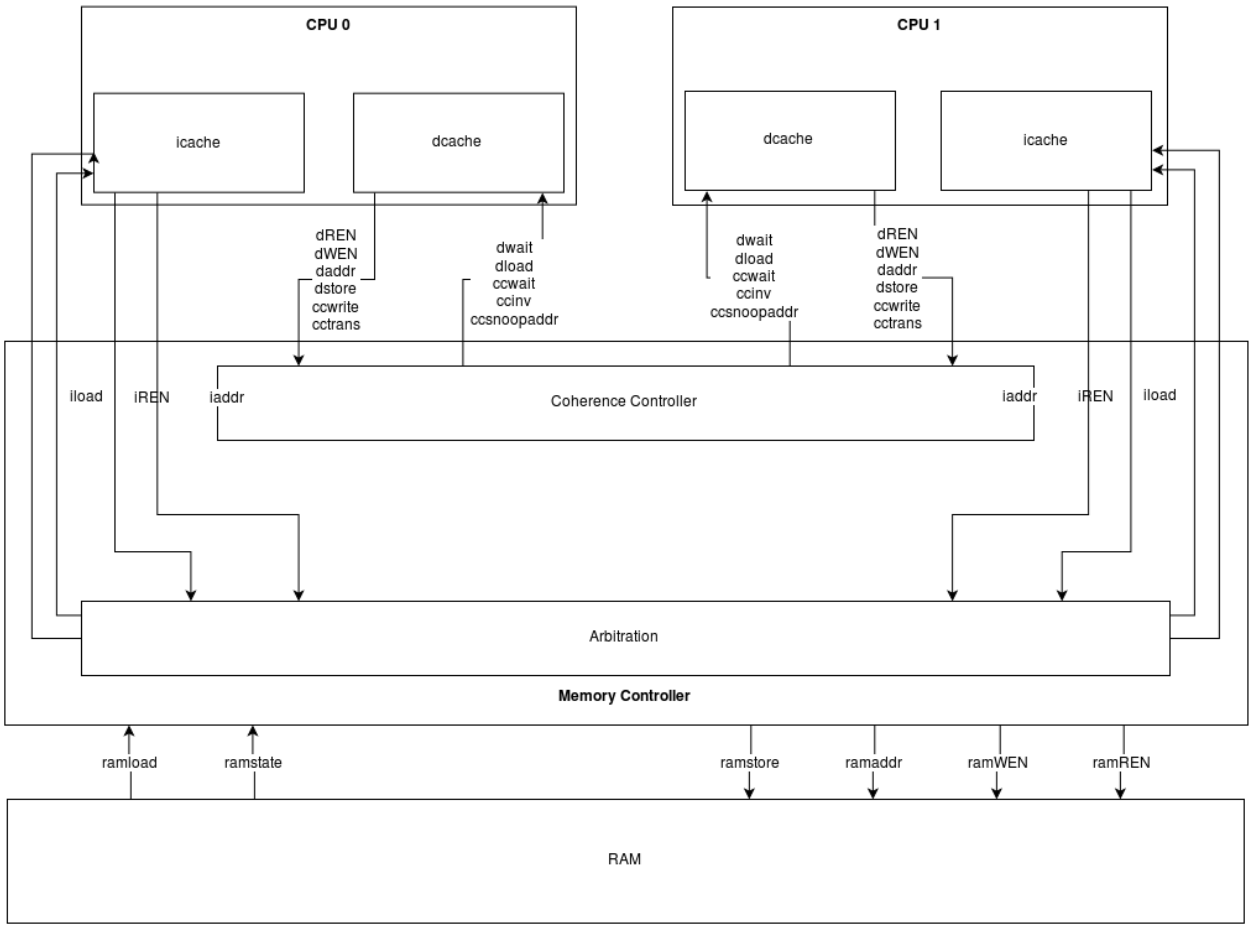

Figure 1: Multicore Block Diagram

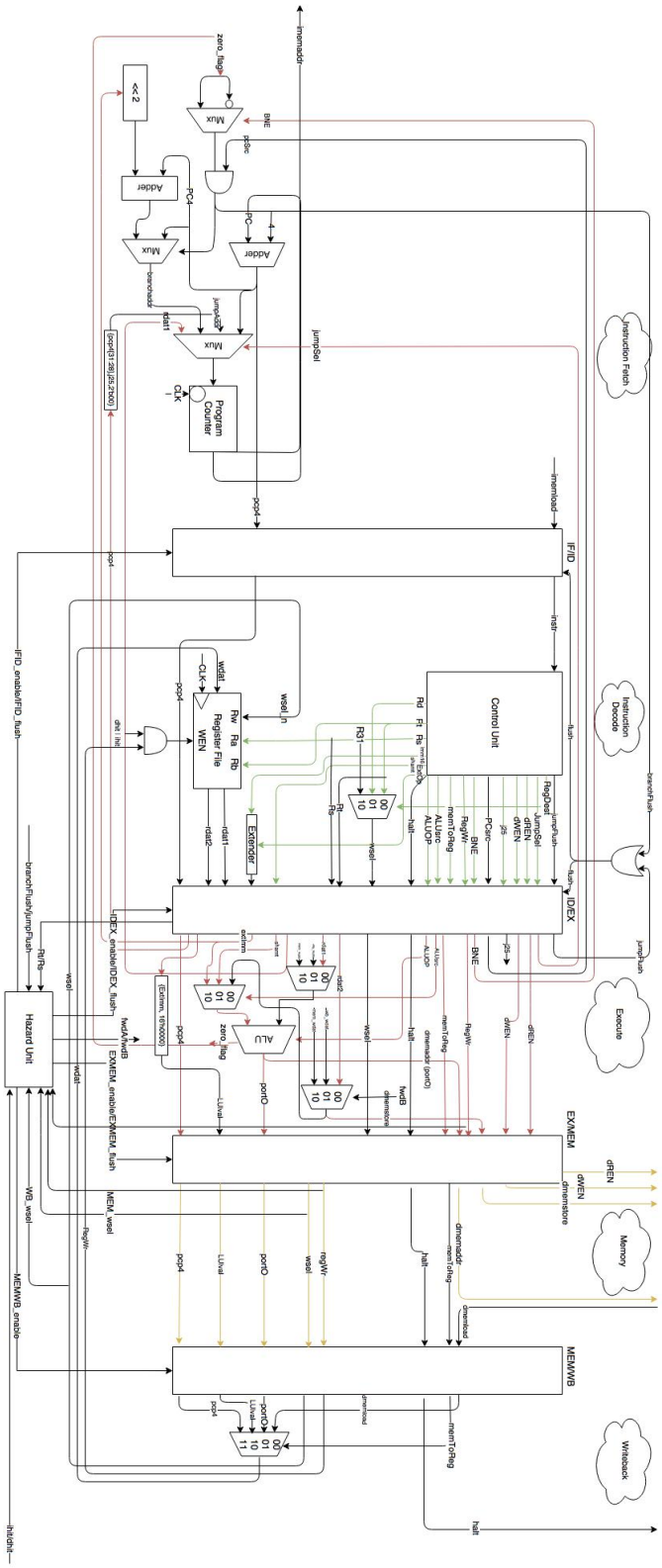

Figure 2: Pipelined Processor Block Diagram

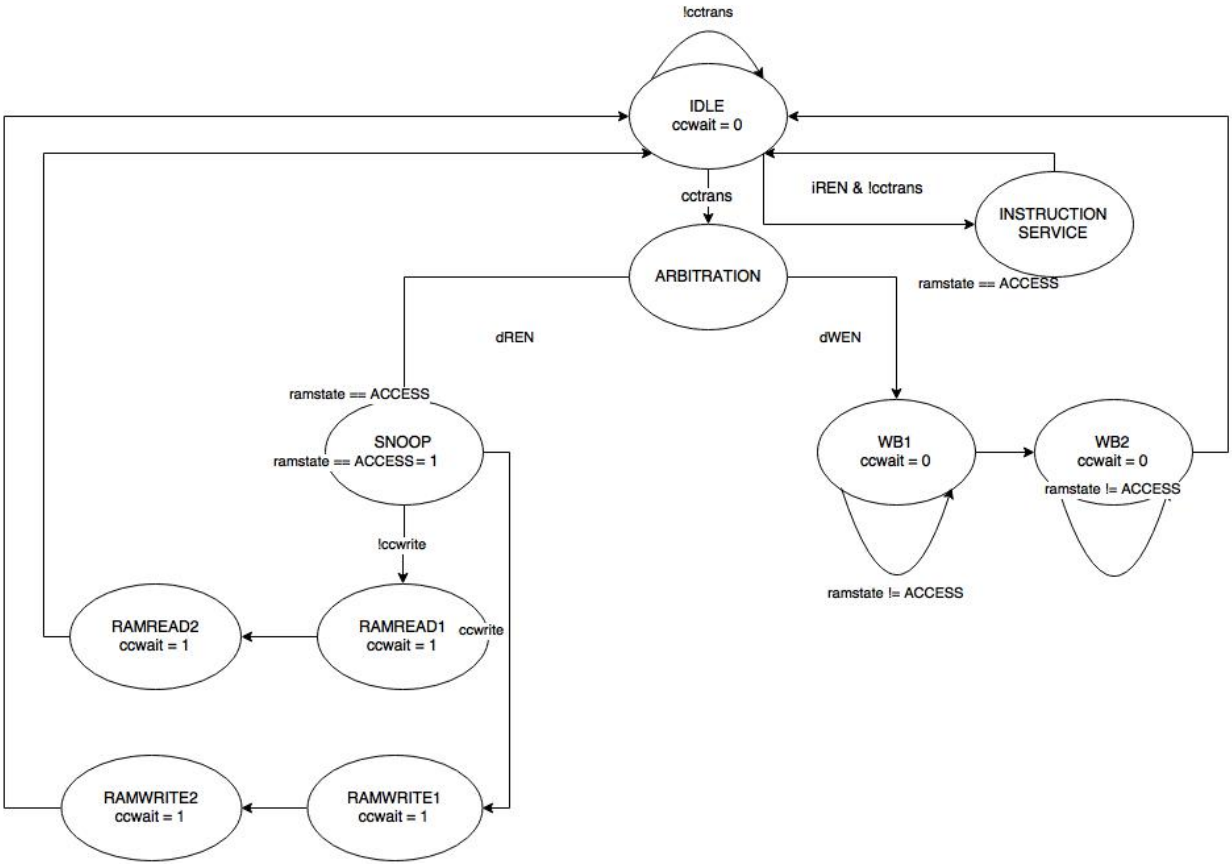

Figure 3: Cache Coherency Unit Block Diagram

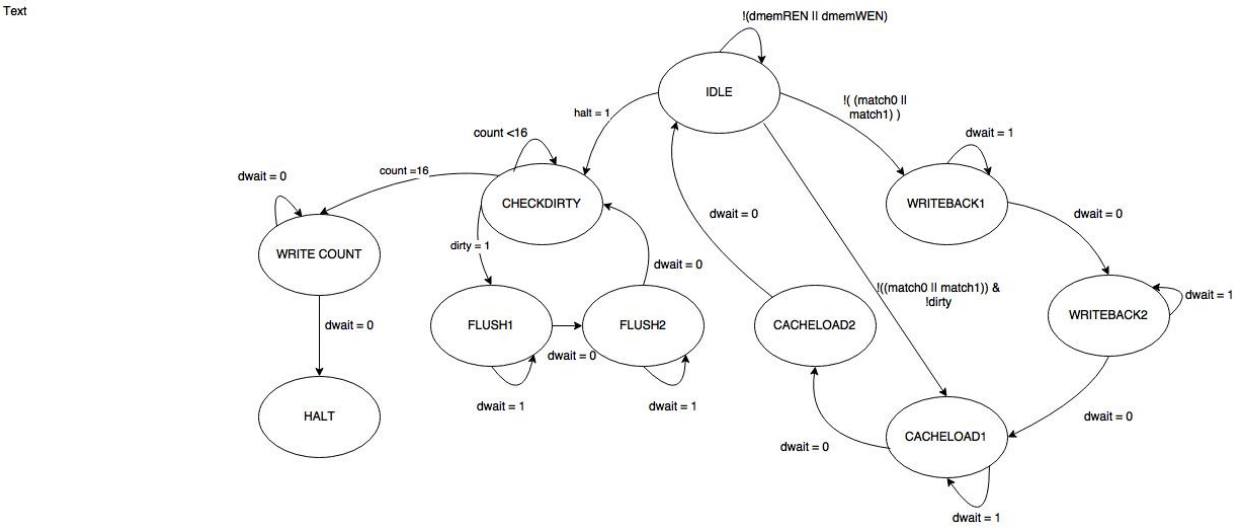

Figure 4: Cache State Diagram

3 Results

	Pipeline w/o Cache	**Pipeline w/ Cache**	**Multicore**
Estimated Synthesis Frequency	45.09 MHz	38.01 MHz	20.37 MHz
Instructions per Cycle (IPC)	0.078	0.230	0.294
Execution Time	1.535 ms	0.6175 ms	0.9040 ms
Single Instruction Latency	110.9 ns	131.5 ns	245.5 ns
Total Utilized Combinational Functions	3,274	6,446	13,800
Utilized Dedicated Logic Registers	1,720	4,175	8,208
Speedup from Sequential to Parallel	1.000	2.486	1.698

Table 1: Synthesis Results

4 Conclusion

The performance results of running the merge sort algorithm on the pipelined processor, pipelined processor with caches, and multicore processor with caches indicate decisively that the addition of caches speed up the execution time of the algorithm. Both the cache-added processor and the multicore processor showed increases in instructions per cycle, a decrease of execution time, and a net speedup over the original design. To accompany these positives, however, for both processors the clock speed was required to increase due to longer critical paths, single instruction latency increased in duration (number of latency cycles remained the same at 5), and area increased substantially. While all of these changes are expected due to the increases in area that accompany addition of hardware and intricacies within logical flow, the net gain of performance of the processor was shown to be positive.

Interestingly, as well as somewhat disappointingly, the findings also show that the multicore design actually slowed down in comparison to that of the processor with added caches. Despite the fact that the multicore design has a substantially higher number of instructions

per cycle (IPC) with an increase of 27.8% over that of the cache-added processor, the clock speed loss between the two iterations is sufficient enough to cause the execution time reduction of 46.4%. There are several reasons that this could be the for this increased critical path such the increase of rise times of long wires with high capacitances within the multicore design. The speedup, however, still shows that the execution time has remained low enough to net a gain in speed over that of the pipelined processor without caches - a threshold that, if passed, would have implicated more serious issues with the approach to handling parallelism.

5 Contributions and Collaboration Strategies

The entirety of the processor design in all of its stages was done in tandem between Pooja and Brian. Early on in the process, the design was formulated together with all principles of the design being taken into mutual consideration - all design decisions (for instance, the design of the state machines for DCACHE and the coherence controller) were made together before any development began.

5.1 Individual Contributions

During development, tasks were partitioned between Pooja and Brian in order to ease the development process. As such, there were two primary initiatives in this phase of development, divided up as follows:

5.1.1 Cache Design

For cache development, there were two primary products: the instruction cache and the data cache. Both were constructed as a team with the following contributions:

- **Instruction Cache** - The instruction cache was developed by Brian and a testbench was written and used to ensure functionality by Pooja. The instruction cache, in an entirely anomalous case compared to the rest of the project, required no debugging in order to work properly.

- **Data Cache** - The data cache state machine was developed together in order to ensure understanding between the two teammates. After the cache design was discussed, Brian began to add it to the data cache itself and Pooja constructed the testbench to ensure that the data cache itself would work. At this point, the cache appeared to work properly, but when it was inserted into the greater system new bugs began to appear. The diagnosis process at this point was done together and the changes were discussed and added until the design itself was complete.

5.1.2 Multicore Design

The multicore adaptation of this design, being the most significant time investment of the project, required an immense amount of work. The entirety of the development process consisted of both team members being present for all but a small amount of debugging. Control of cache coherency was added to the design and debugged in tandem - ultimately causing some of the final bugs in the project. Additionally, the data cache was adapted to support a snooping mechanism to support multicore cache coherency. All of these features were added with both teammates present and required a substantial amount of debugging.

5.2 Collaboration Strategies

Account information was shared between team members at the beginning of the collaboration stages and SSH keys were set up between the two accounts for ease of user-switching, but each team member did their development within their respective accounts and contributed to GitHub using their own account. Because of this Git usage and presence on GitHub, all contributions are documented in a quantitative manner within the repository.

5.2.1 Git VCS Usage

Just as in the midterm, in order to compartmentalize development of features and to keep both contributors on the same page, an industry strategy of progressive-stability branching was employed and all development was done within respective branches and "work silos" - a master branch was maintained with the most recent functioning design, "singlecycle",

"pipeline", "caches", and "multicore" branches were maintained, and each feature were developed on an independent branch for that feature (some examples being "branchpredict" and "forwarding" in the earlier stages of development). While this required more overhead in terms of planning and maintenance, the preservation of states and releases within the development process enabled ease of rollbacks, identification of changes, and the ability to develop without worrying about destruction of a prior working state.