Processor Prototyping Lab
Midterm Report

Stephen Fetterman, Joshua Hom
Zhaoyu Jin

December 10th, 2023

1 Executive Overview

In this report, different processor designs will be analyzed and compared to each other. They are the single-cycle, pipelined (with and without caches), and multicore designs. The single-cycle design is the simplest form since it only processes one instruction at a time. This form should have the least CPI but is also the slowest. The pipelined design contains latches that allow the processor to take up to five instructions at a time. This allows it to be faster than the singlecycle design with slightly increased CPI. Caches are an addition to the processor which introduces data locality. It allows the processor to access recently used data without getting it from RAM which saves time. The multicore design contains two instances of pipelined datapath and caches. With this, two separate threads can run in parallel, processing more instructions at a time. The performance of each design will be analyzed which includes CPI, instruction latency, and program runtime. This report also includes schematics of the multicore design with caches and a contributions section.

2 Processor Design

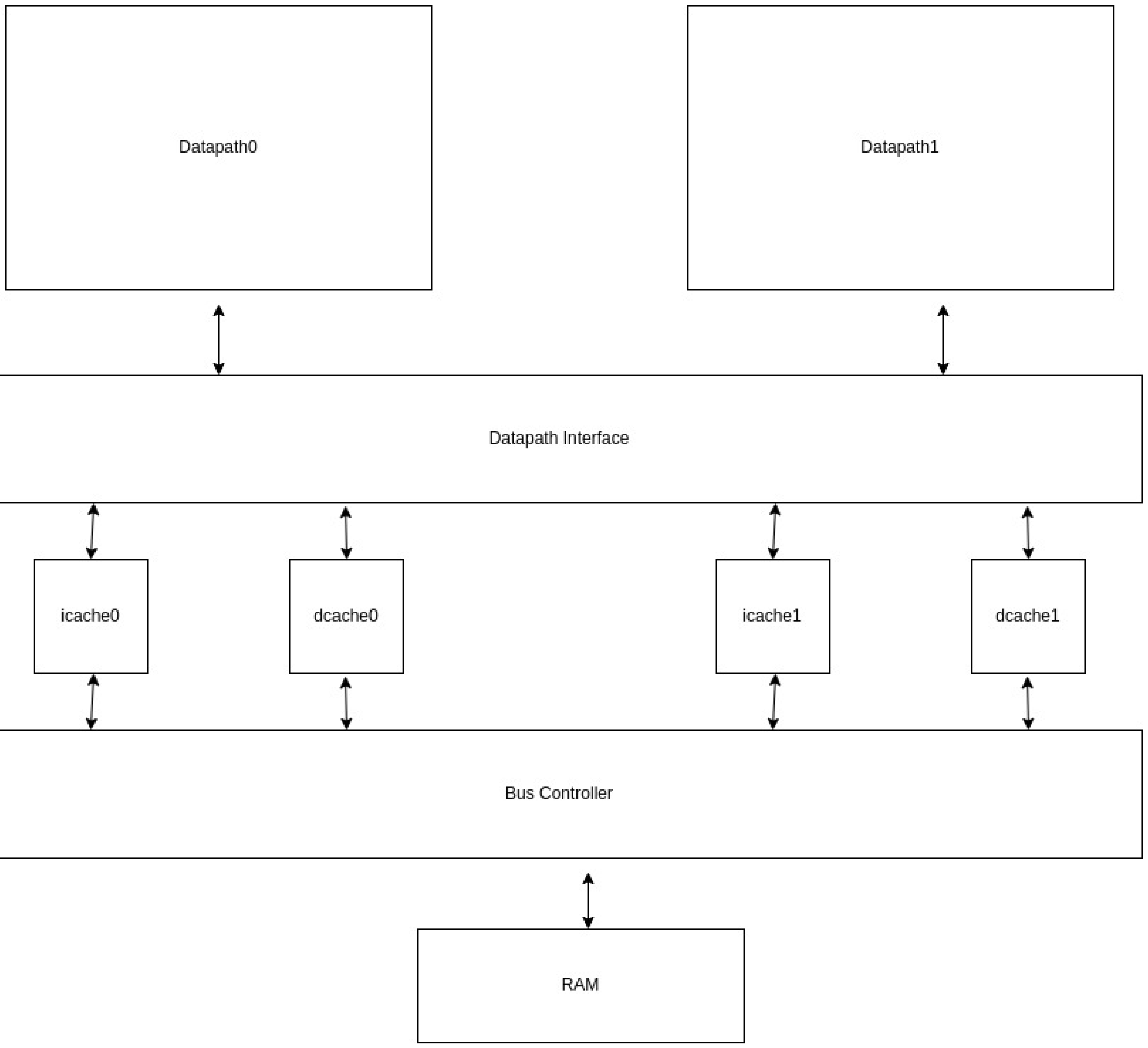

Figure 1: Final Overall Design

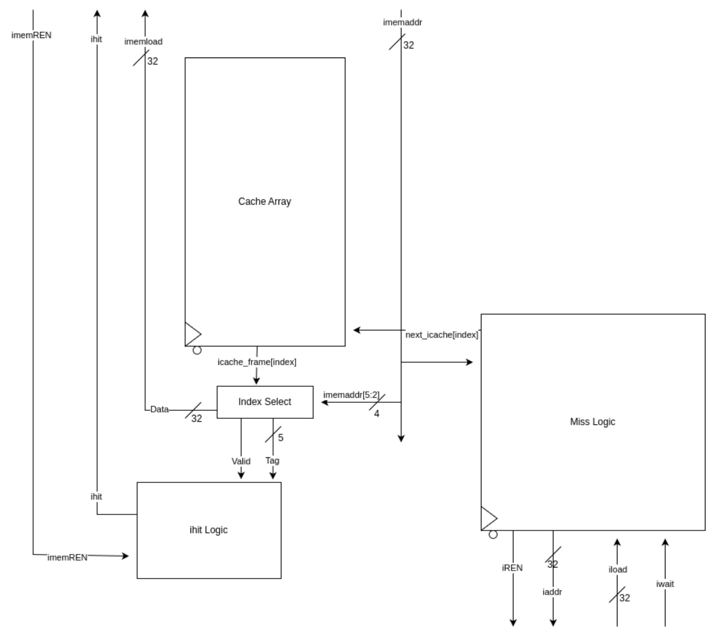

Figure 2: Final I-Cache Design

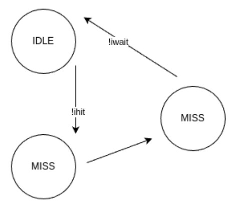

Figure 3: Final I-Cache State Diagram

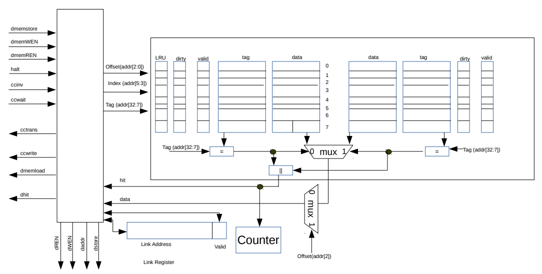

Figure 4: Final D-cache Design

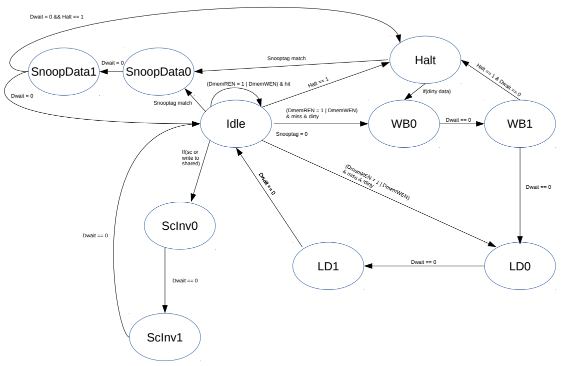

Figure 5: Final D-cache State Diagram

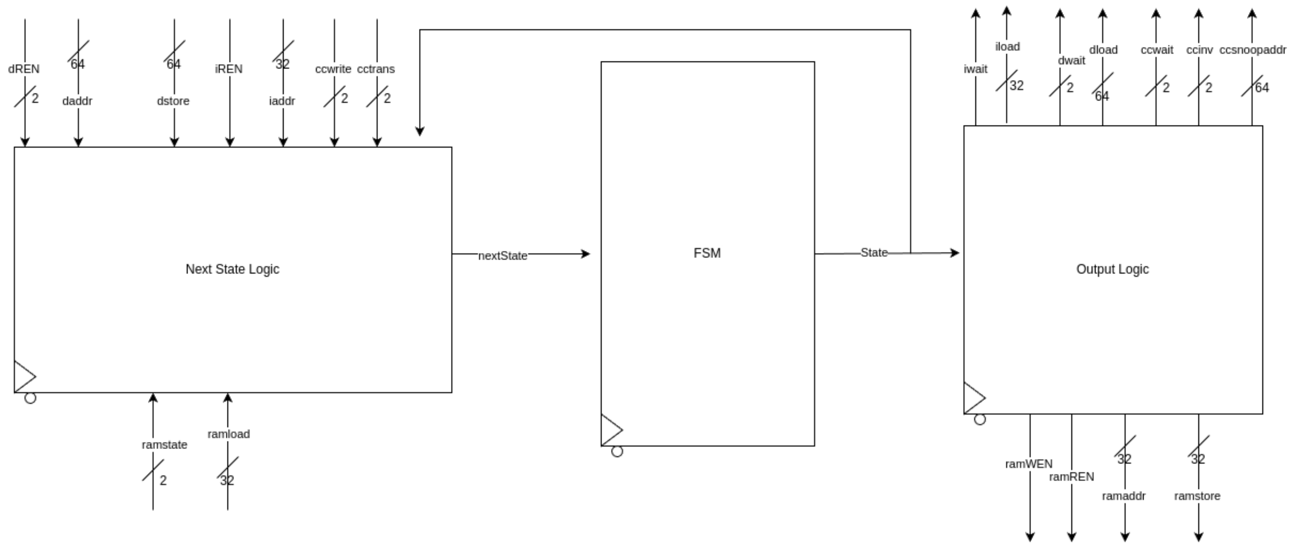

Figure 6: Final Bus Design

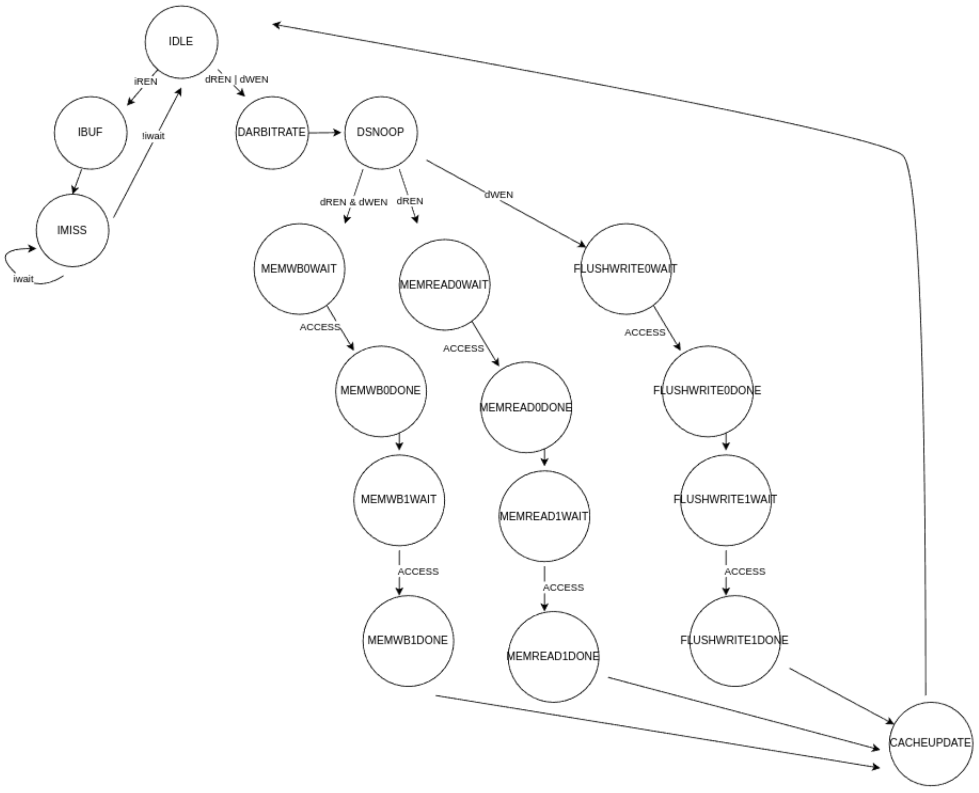

Figure 7: Final Bus State Diagram

For the remainder of the class, three new components were designed and integrated into the existing pipelined processor design. An instruction cache (icache), data cache (dcache) were added to the pipelined processor for a large speedup. The datapath connected to the memory controller which connected to the ram. For the final iteration, the datapath with an icache and dcache were duplicated and connected to the bus controller which replaced the memory controller. The bus controller arbitrates between each dcache and icache, and interfaces with memory.

The icache is a simple FSM with three states used to interface with the memory/bus controller. A struct was provided to standardize and parameterize icache instruction storage and make writing and debugging easier. The icache stays in the idle state by default, only moving to a buffer state when a miss is detected to set up the memory controller signals. Immediately after it enters a wait state until the memory controller notifies the icache that the instruction is ready. After receiving the instruction it is written to the appropriate register in the struct and a hit is sent to the datapath before going back to idle. In the common case where the instruction at the requested address is present in the icache, a hit signal is asserted immediately within the same cycle. When evaluating the waveforms, it is frequent to see the hit signal high for a dozen or more cycles in a row while inside a loop. The icache was tested thoroughly and is functionally correct at all required latencies.

The dcache has invalidate, load, writeback snoop, and halt states. If the dcache has a miss to a dirty block, it will go to the writeback states and make a request to the bus to writeback the dirty data back to RAM or a cache. It will then go to the load states, which can also occur when there is a miss and the block is not dirty. The load states will make a request to read from RAM or a cache. There are 0 and 1 states for each word. "ScInv0" states occurs when a dcache recieves an sc or has shared data that is just modified. It is used to invalidate the other dcache's link register or stale shared data. The snoop states occurs when there is a match between the snooptag from the bus and a dcaches tag. In these states, the dcache will send the data the tag matches for. These states are also where the dcache's invalidation occurs. Finally, there is a halt state, this state brings the dcache out of the idle state and starts to make requests to the bus, writing dirty data back to the ram. During halt, the dcache can also be snooped.

The bus controller is a complex FSM with three main ‘paths’ corresponding with the required actions. The bus controller starts in idle and performs arbitration between both icaches and dcaches. Giving priority to data requests and a least recently used flip-flop, one of the four caches are granted access. A signal is asserted to tell each cache when they have been selected, and for each data transaction the bus ‘snoops’ into the other. After memory signals are set up, one of the ‘paths’ are chosen based on the results of the ‘snoop’ and data transfers begin. Either data is read from memory and sent to a data cache, data is written back from a data cache and transferred from cache to cache (if applicable), or data is only written back after halting. For an instruction request, a cycle opposite to the icache FSM was used.

3 Results

Frequency = max(CPUCLK, CPU/2)

CPI = (Cycles/2) / # of Instructions

Total Execution Time = (# of instructions) * CPI * (1/Frequency * 1E6)

Latency = 1E9 * Total Execution Time / # of instructions

Results of running the provided mergesort on an Altera DE2-115. The terminal command “testasm” script was used to compare the MIPS simulated memory dump with the memory dump from each processor at every step respectively. Variable memory latencies were utilized to ensure each processor implementation is not dependent on memory speed, which is necessary for real world implementations.

Frequency	35.335 MHz
Average CPI	4.44 clocks per instruction
Average Latency	129.2 ns
Average Total Execution Time	0.689 ms
Total Registers	1,287
Total Pins	102

Table 1: Processor Specs for Pipeline with Caches

Frequency	63.01 MHz
Average CPI	6.27 clocks per instruction
Average Latency	103.5 ns
Average Total Execution Time	0.559 ms
Total Registers	1,766
Total Pins	102

Table 2: Processor Specs for Pipeline without Caches

Frequency	56.49 MHz
Average CPI	2.53 clocks per instruction
Average Latency	46.91 ns
Average Total Execution Time	0.253 ms
Total Registers	8,803
Total Pins	102

Table 3: Processor Specs for Pipeline with Caches

Frequency	57.8 MHz
Average CPI	2.68 clocks per instruction
Average Latency	47.34 ns
Average Total Execution Time	0.256 ms
Total Registers	44,555
Total Pins	102

Table 4: Processor Specs for Multicore with caches and single-threaded program version

Frequency	58.47 MHz
Average CPI	3.44 clocks per instruction
Average Latency	60.29 ns
Average Total Execution Time	0.327 ms
Total Registers	44,555
Total Pins	102

Table 5: Processor Specs for Multicore with caches and dual-threaded program version

4 Conclusion

Our dual-core multithreaded pipelined processor with caches worked as intended and fulfilled the goals we set for ourselves. Adding caches to the design offered a massive speedup from previous iterations, which was more noticeable at higher latencies. The Lab 9 average CPI of 2.53 was significantly lower than the Lab 7 average CPI of 6.27. Data for each latency was omitted for brevity, but the latency 10 CPI lowered from a slow 11.12 all the way to 3.03, a massive improvement at similar frequencies. Although the multicore implementation was more conceptually difficult, the numerical improvements weren't as visible and didn't provide a CPI decrease or total execution time decrease. This is likely due to program selection, where a routine with less memory accesses would be faster and take better advantage of the caches.

Overall, the SystemVerilog implementation was a success and we delivered upon the key features required for the last half of the course. Caching was implemented to benefit from a memory hierarchy, where the common case of a data/instruction hit is fast. Cache coherence ensures writes are atomic using our MSI cache protocol. Synchronization and consistency are guaranteed with our load linked and store conditional implementation. Despite the functionality, there are some ways to improve. First, the bus controller's interactions with memory were not optimal, and there were many cycles where the bus is looping through states while RAM is free. Also, the bus protocol was unoptimized due to the complexity and the required time debugging. These changes were referred to as “weirdnesses” in the lecture, and involve unnecessary reads and writes to memory that simplify the protocol. Finally the improvements from the midterm report were addressed and successfully implemented. A forwarding unit and sophisticated branch predictor were designed and interfaced correctly with the final design. These contributed to the large performance improvement between labs 7 and 9.

5 Contributions

Joshua designed the D-Cache and created the testbench for the I-Cache. Stephen designed the I-Cache and created the testbench for the D-Cache. Joshua and Stephen created the unit assembly files to test the caches and their associativity. Stephen designed the bus controller and made the datapath changes while Joshua created the respective testbench and unit assembly files to test the MSI protocol. Joshua also updated the D-Cache to implement LL-SC based locks. Stephen wrote and debugged the parallel algorithm assembly file. Additionally, Stephen connected Joshua’s forwarding unit to the datapath. Stephen also designed and implemented a 2-bit saturating branch predictor. Joshua performed most of the top level design debugging, and Stephen assisted whenever available. For this report, Joshua made the rough draft of the overview and contributions while also supplying the figures related to his lab contributions. Stephen wrote the icache and bus controller designs, the conclusion, and edited the report.