# ECE 437: Processor Prototyping Lab

Final Report

Nikitha Suraj, Saandiya KPS Mohan

Section 5

Zhaoyu Jin

May 4, 2025

**Overview:**

This report presents a comparative performance and resource analysis of five RISC-V processor architectures developed incrementally throughout the semester which are single-cycle, pipelined without caches, pipelined with caches, multicore with caches running a single-threaded program, and multicore with caches running a dual-threaded program. Each design reflects a progressive architectural enhancement, targeting higher instruction throughput, better memory utilization, and improved scalability. The goal of this comparison is to examine how architectural complexity, through pipelining, cache hierarchy, and multithreading impacts key performance metrics under memory latency constraints. All designs were tested using the mergesort.asm program, with the dual-threaded version using dual.mergesort.asm, allowing consistent benchmarking across designs. Memory latency was standardized at 6 for core analysis, with additional testing across latency values 0, 2, and 10 to evaluate cache effectiveness.

The single-cycle processor, while simple and compact, executes all instruction stages in one clock cycle, resulting in high latency and limited throughput. The pipelined processor without caches introduces instruction-level parallelism across five stages, improving throughput but exposing memory latency bottlenecks. Adding instruction and data caches in the pipelined with caches version mitigates these memory stalls and improves execution time, especially under higher RAM latency. The multicore processors build on the cached pipeline design by duplicating cores and introducing a shared memory bus; the single-threaded variant runs one instance of the benchmark, while the dual-threaded version runs a parallelized version across two cores, demonstrating thread-level parallelism and higher speedup. To evaluate performance, we analyzed synthesis frequency (Fmax), average CPI, average instruction latency, total execution time, and FPGA resource utilization. Data was collected using the mapped netlist with gate timing and simulated with our testbench using sweep tables and simulation logs. Additionally, we calculated speedup from sequential to parallel execution and identified the memory latency at which caches begin to offer measurable performance benefit. The rest of the report includes design diagrams, detailed experimental results, performance tables for each memory latency setting, a final analysis to contextualize results, and contributions of each team member.

**Design:**

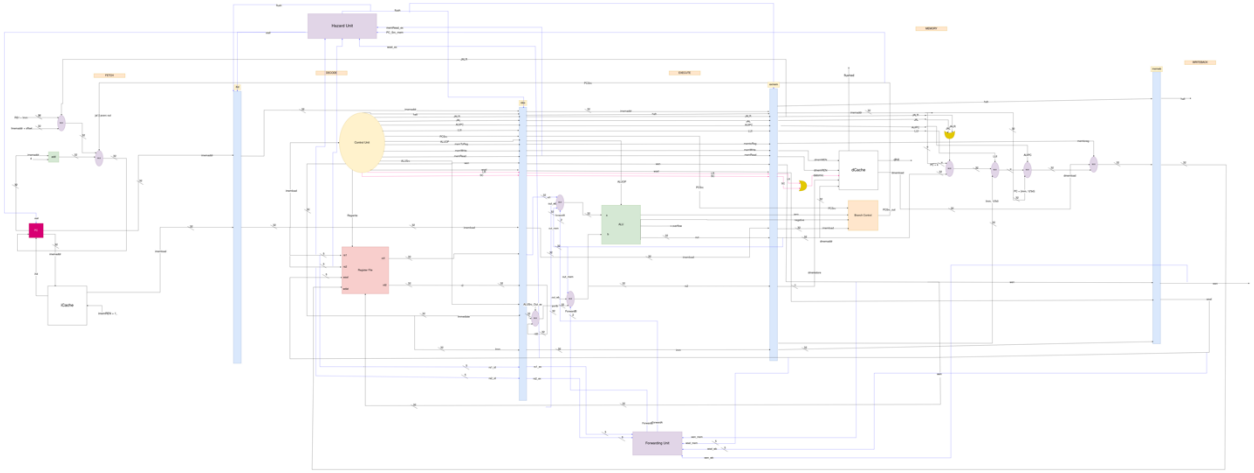


*Figure (a): Pipeline Processor with LR/SC*

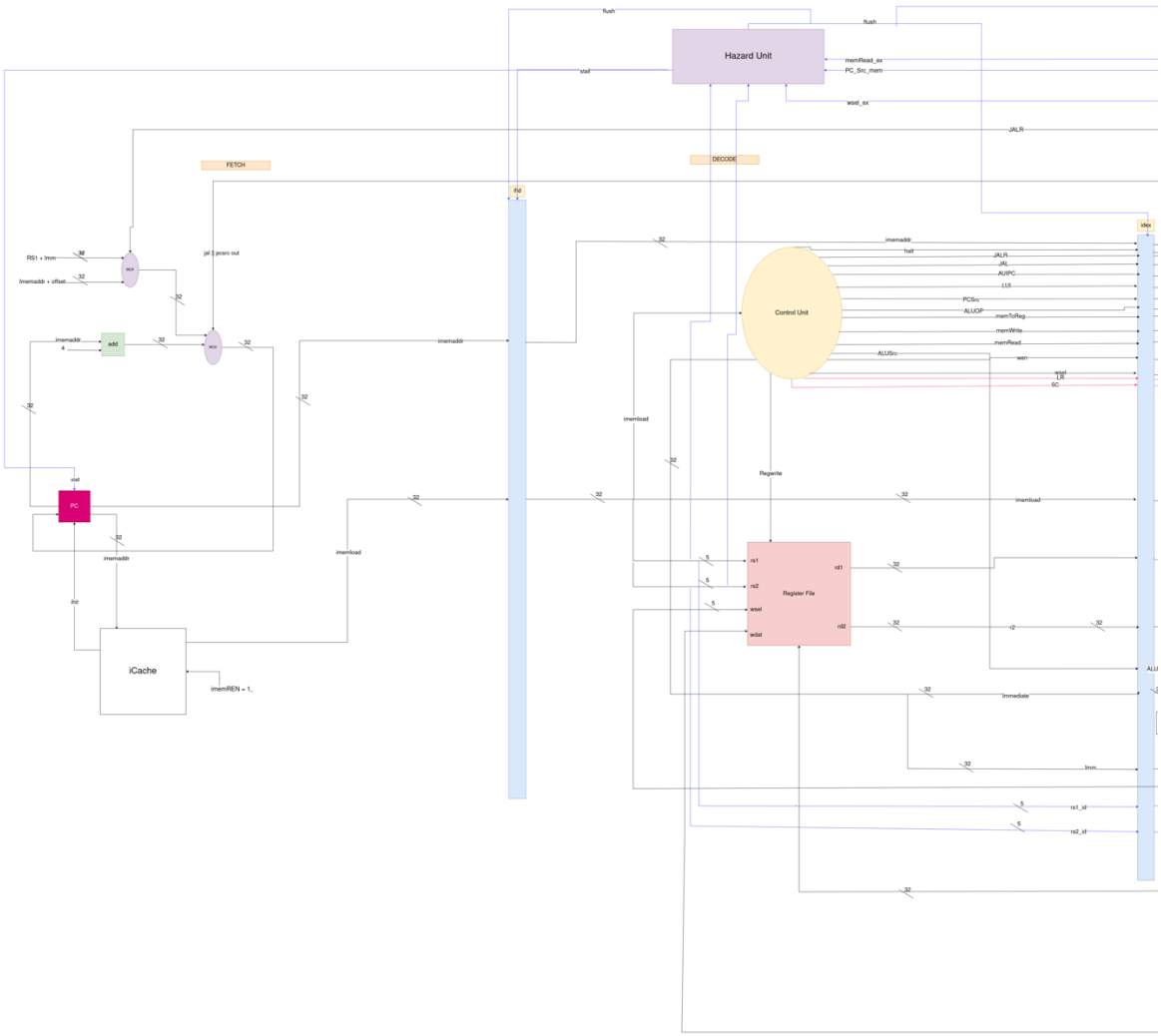


*Figure (b): Pipeline Processor with LR/SC – Fetch, Decode, Hazard Unit*

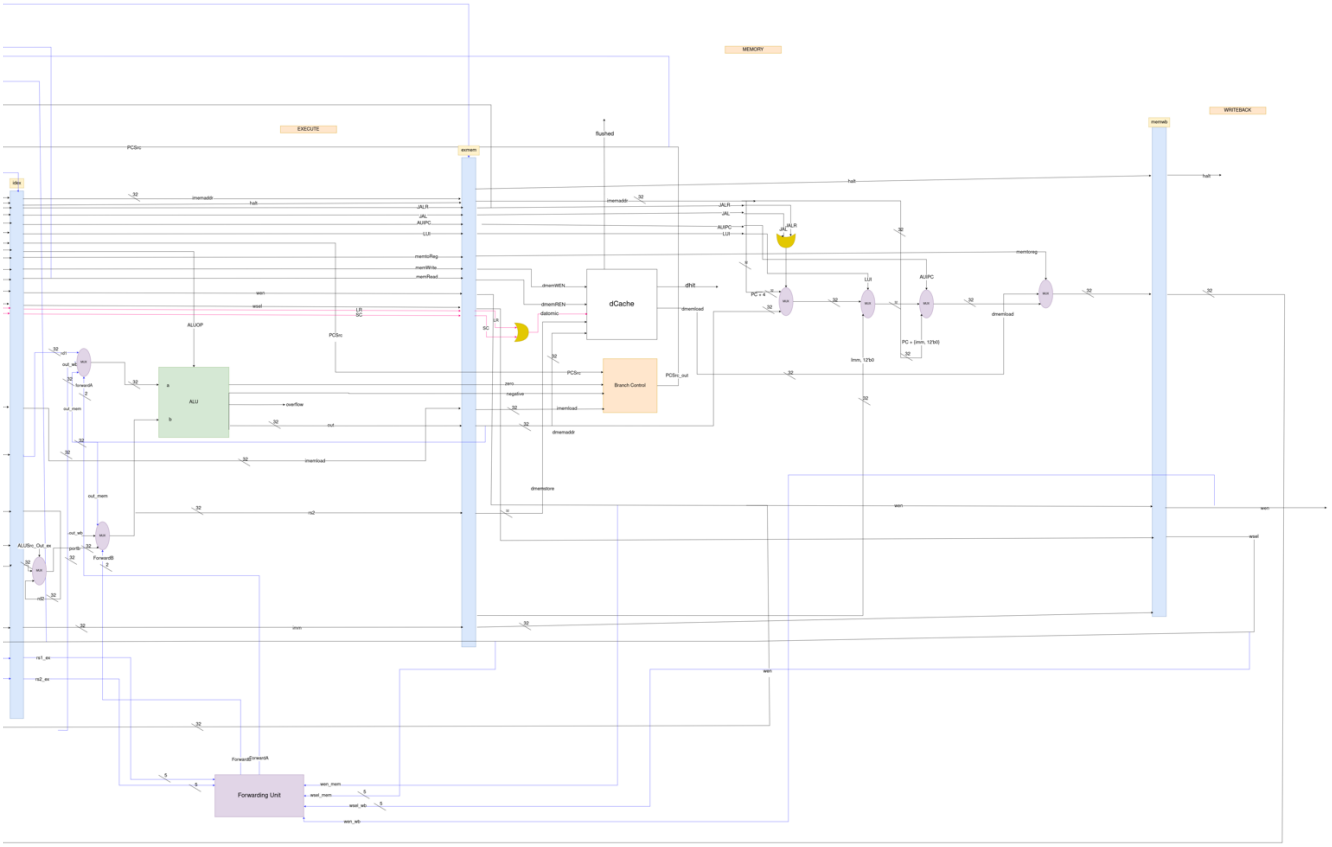


*Figure (c): Pipeline Processor with LR/SC – Execute, Memory, Writeback, Forwarding Unit*

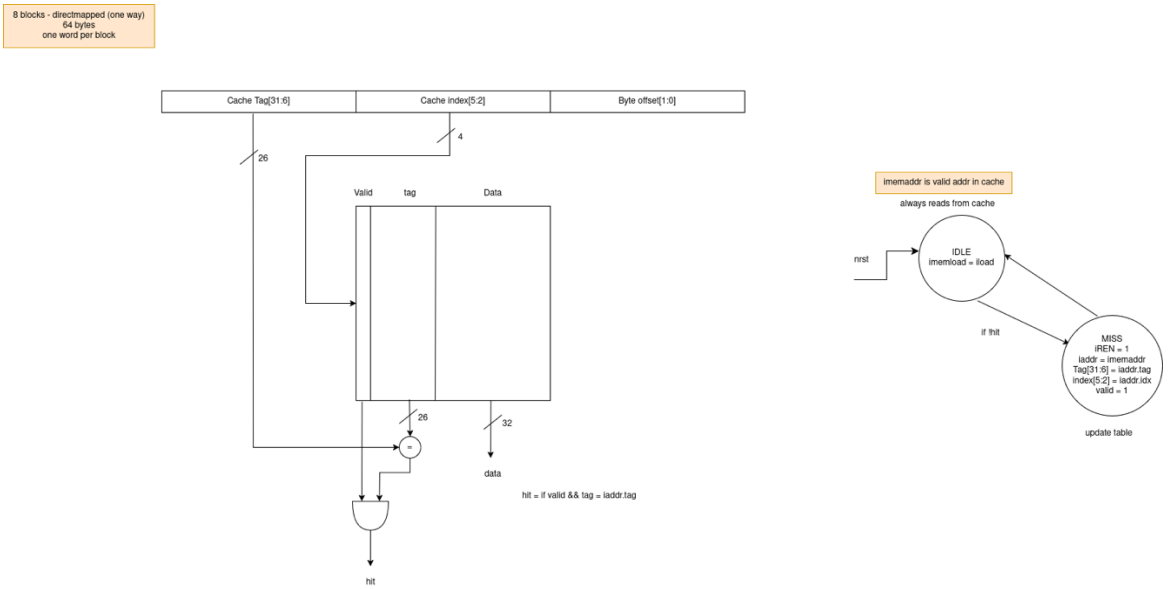


*Figure (d): icache RTL and FSM*

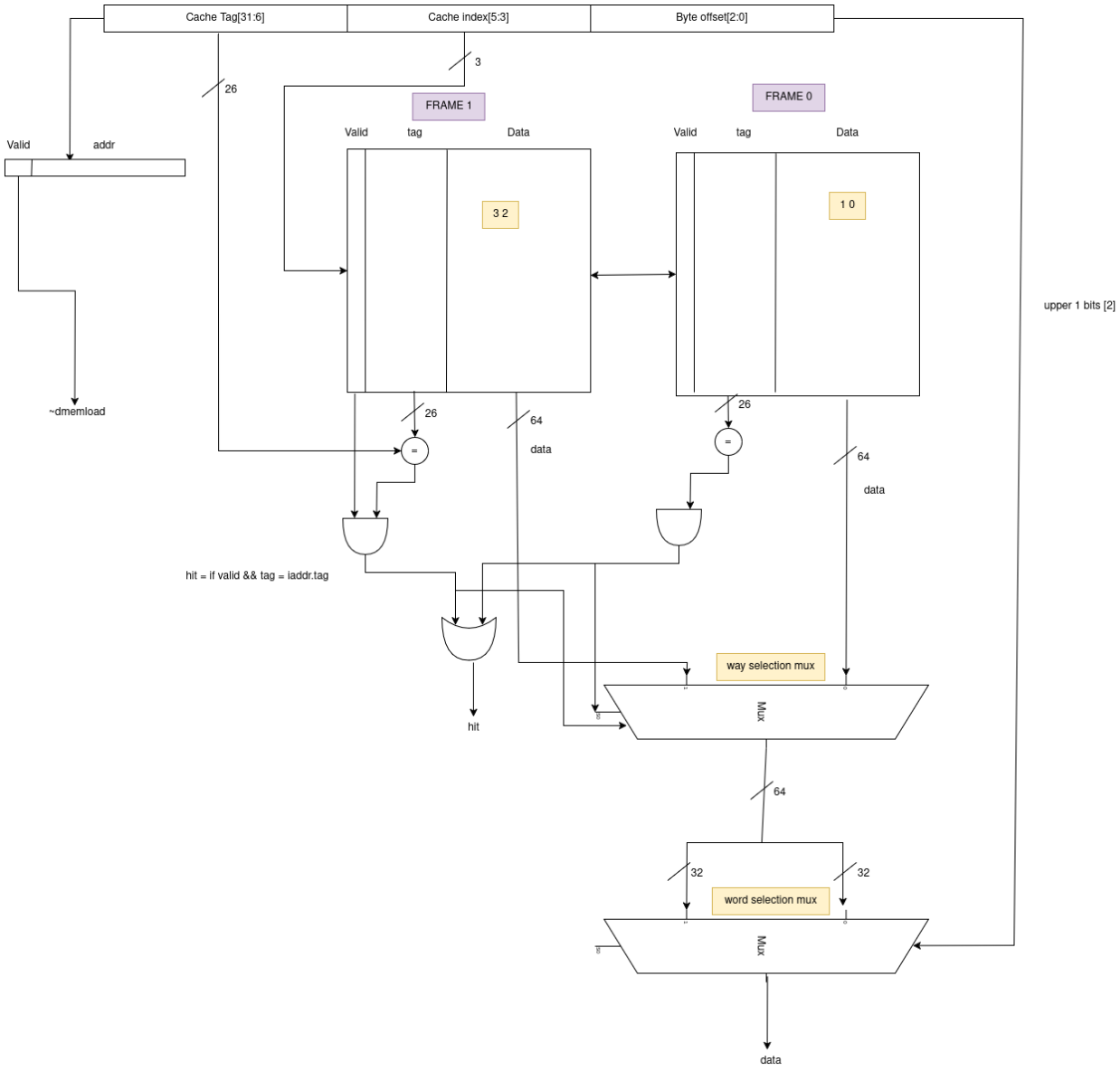


*Figure (e): dcache RTL with LR/SC*

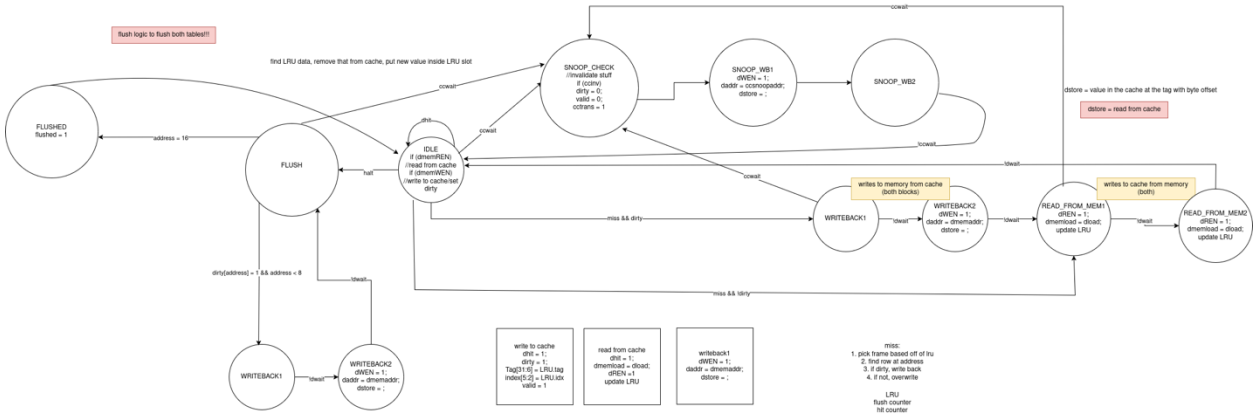


*Figure (f): dcache FSM without LR/SC*

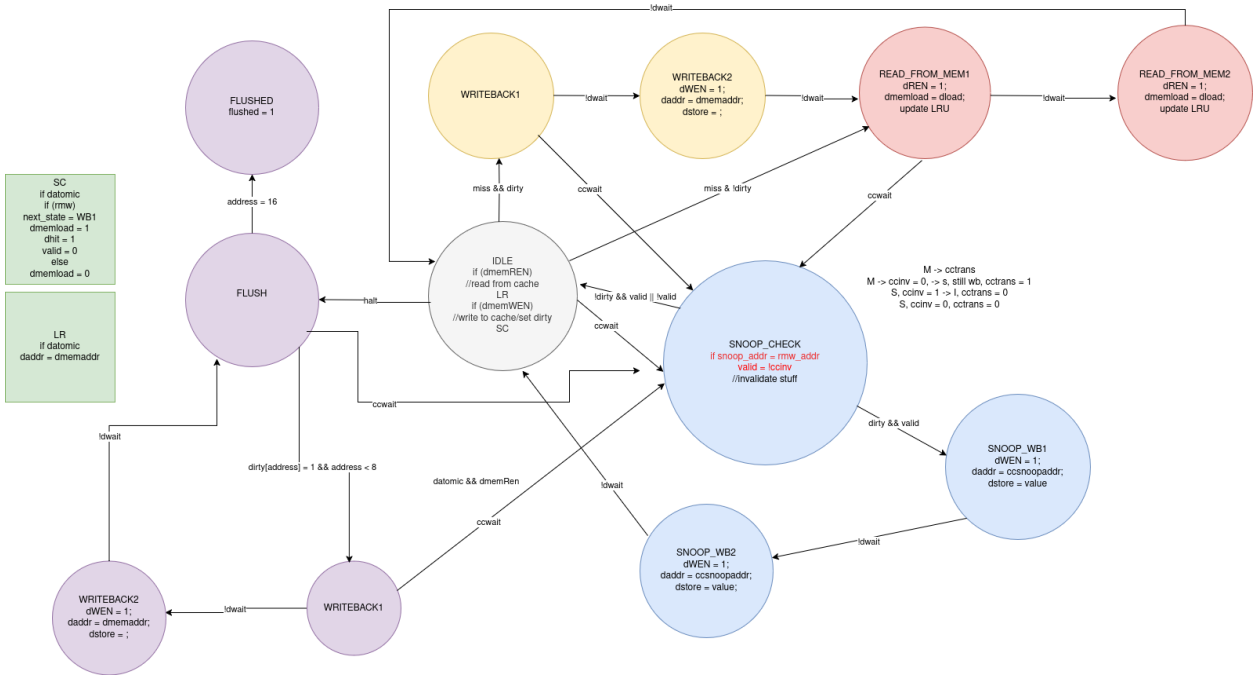


*Figure (g): dcache FSM with LR/SC*

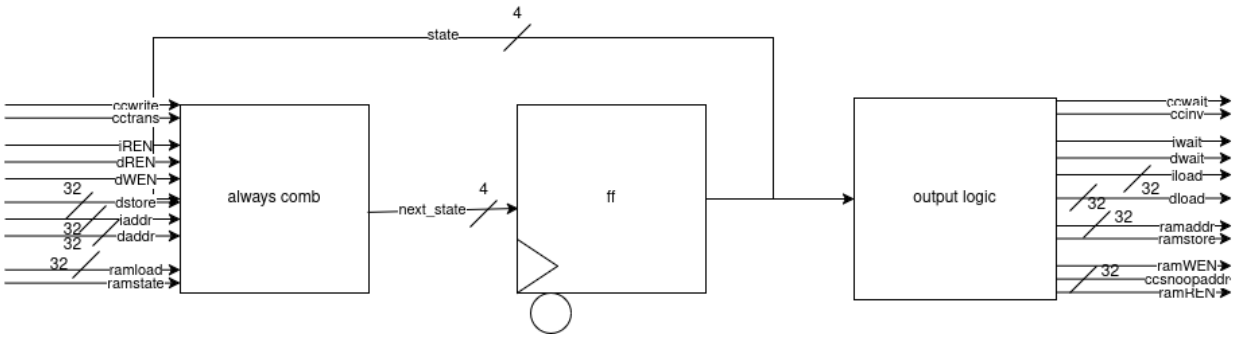


*Figure (h): Bus controller RTL*

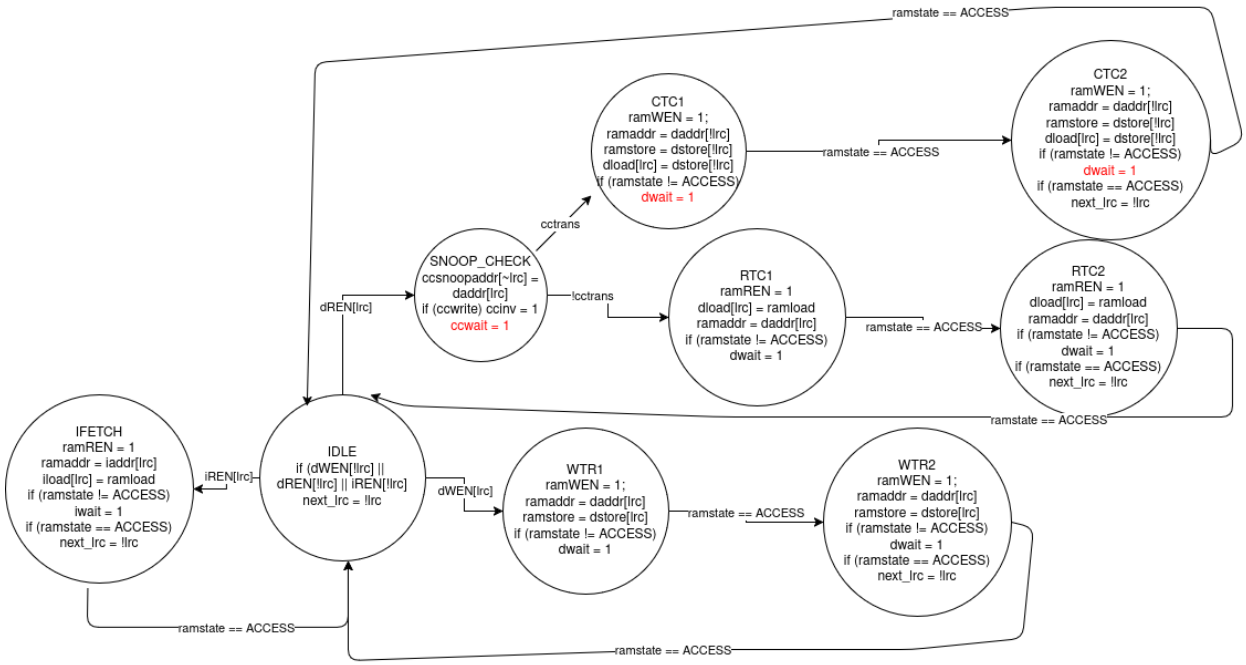


*Figure (i): Bus controller FSM*

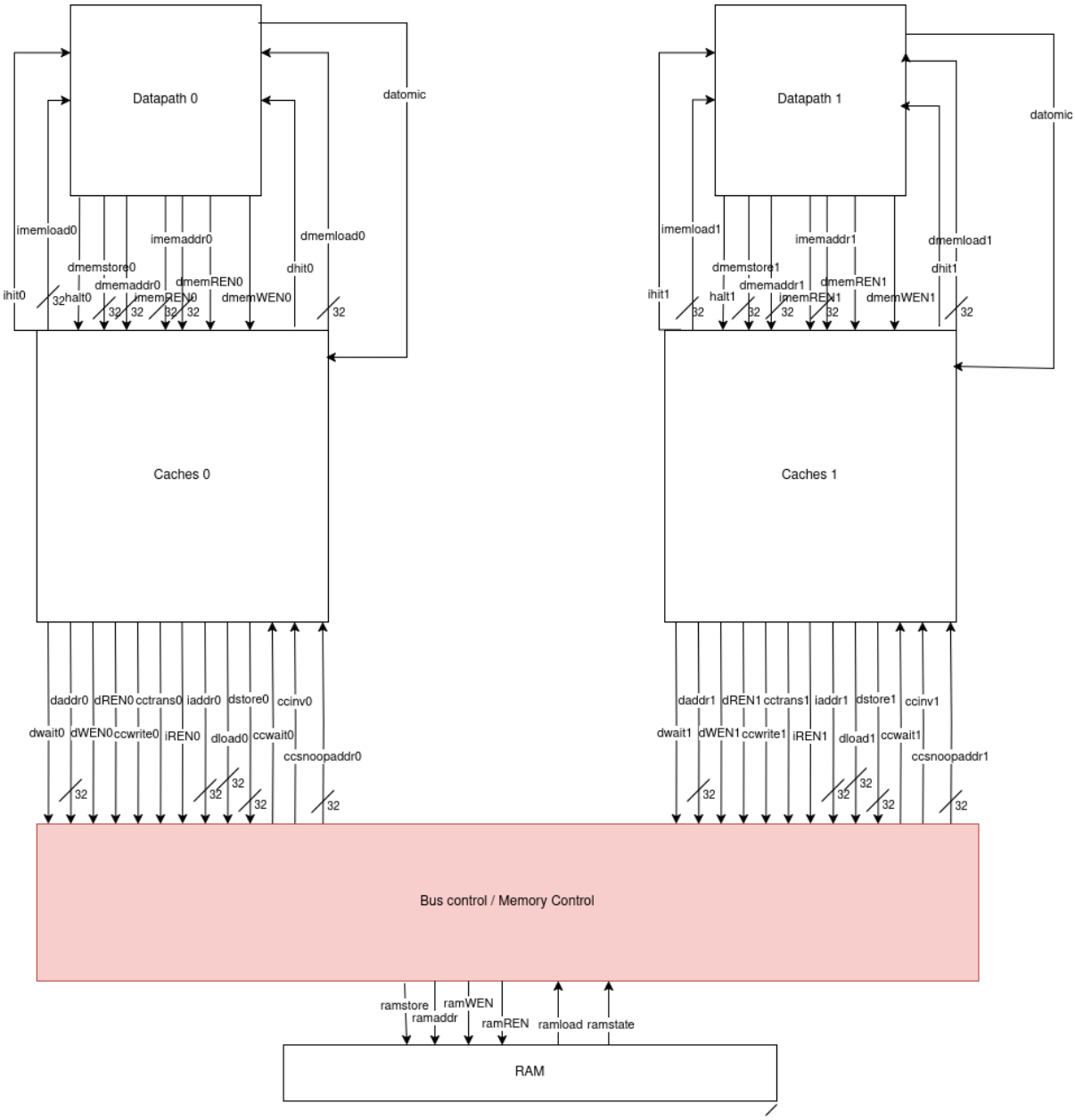


*Figure (j): Multi Core CPU Block Diagram*

**Results:**

| Design | LAT | Fmax (MHz) CPUCLK | Average CPI | Average Latency (ns) | Total execution time (ms) | FPGA Resources |
|---|---|---|---|---|---|---|
| **Single Cycle** | **6** | 38.83 | 5.108 | 131.542 | 0.712 | **Total Logic Elements: 3,217 Total Registers: 1293** |
| **Pipeline without caches** | **6** | 66.55 | 7.478 | 561.84 | 0.608 | **Total Logic Elements: 3,471 Total Registers: 1703** |
| **Pipeline with caches** | **6** | 61.97 | 2.552 | 205.906 | 0.223 | **Total Logic Elements: 7390 Total Registers: 4171** |
| **Multicore (Single Threaded)** | **6** | 60.11 | 2.770 | 230.38 | 0.249 | **Total Logic Elements: 15427 Total Registers: 8298** |
| **Multicore (Dual Threaded)** | **6** | 63.33 | 1.957 | 154.51 | 0.170 | **Total Logic Elements: 15427 Total Registers: 8298** |

*Table (a): Performance data from test program*

| Design | LAT | Total Execution time (ms) |
|---|---|---|
| **Single cycle** | **0** | 0.193 |
| | **2** | 0.356 |
| | **6** | 0.712 |
| | **10** | 1.067 |
| **Pipeline without caches** | **0** | 0.147 |
| | **2** | 0.304 |
| | **6** | 0.608 |
| | **10** | 0.911 |
| **Pipeline with caches** | **0** | 0.172 |
| | **2** | 0.188 |
| | **6** | 0.223 |
| | **10** | 0.258 |
| **Multicore (Single Threaded)** | **0** | 0.196 |
| | **2** | 0.213 |
| | **6** | 0.249 |
| | **10** | 0.285 |
| **Multicore (Dual Threaded)** | **0** | 0.116 |
| | **2** | 0.137 |
| | **6** | 0.170 |
| | **10** | 0.200 |

*Table (b): Total execution time for each major design*

We used mergesort.asm as our test program for single threaded, 5409 instructions and dual.mergesort.asm for multithreaded, 5429 instructions (core 0 + core 1). The formulas below were used to calculate the CPI, latency, and total execution time for tables a and b. The parameters required are the total instructions per program, Fmax, from 'system.log' and total number of cycles executed, recorded from the "make system.sim" command for each LAT value. The total number of instructions was obtained from running 'sim -t' which reports it at the end. FPGA resources were recorded from the fitter report in 'system.log'.

$$CPI = \frac{cycles}{instructions}$$

$$latency = \frac{time \cdot (stages\ in\ pipeline)}{instructions}$$

$$total\ execution\ time = (\#\ of\ instructions) \cdot \left(\frac{cycles}{instruction}\right) \cdot \left(\frac{sec}{cycle}\right)$$

$$Speedup = \frac{total\ execution\ time\ multicore\ single\ threaded}{total\ execution\ time\ multicore\ dual\ threaded} = \frac{0.249}{0.170}$$

For the latency calculation in single cycle, the number of stages in the pipeline is 1, as all 5 stages of each instruction complete in one clock cycle (no pipelining involved). For the pipelined processor, the number of stages is 5. To calculate the total execution time, the cycles/instruction is effectively the CPI, and the seconds/cycle is the reciprocal of the Fmax value (clock frequency). The RAM latency at which caches started helping is at LAT = 2. The speedup from sequential to parallel program is 1.44 times (31.73%) faster.

**Conclusion:**

The results from the performance evaluations of all 5 processors running mergesort.asm as the benchmark highlight clear improvements with the modifications added to each progressive iteration of the processor, starting with the single cycle. Although the single cycle processor utilized the least FPGA resources, it had the slowest clock speed, and the execution time for merge sort was the slowest of all the processors. While introducing pipelining increased the CPI of the processor compared to the single cycle model, the total execution time of merge sort was decreased significantly. After adding caches to the pipelined processor, while the FPGA resources nearly doubled, the average CPI decreased from 7.478 to 2.552, and the execution time decreased by around 63%, from 0.608ms to 0.223ms. This substantial improvement in performance is due to caching instructions and data; in programs involving a lot of looping and repeated memory accessing, caching can allow the CPU to do fewer memory accesses, which decreases the execution time. At higher latencies, when the time spent during memory accesses are greater, caching becomes very important- the effect of this can be seen in Table b: the difference between the execution time of pipelined CPU without caching and pipelined with caches at higher latencies is very significant.

After caches were implemented, the processor was extended to a multicore design. The FPGA resources greatly increased, but there is no significant change in performance with the single-threaded merge sort program. However, running a dual threaded merge sort program on the multicore processor (with latched dcache signals) showed the best performance of all five implementations of this CPU. The dual-threaded multicore design resulted in the lowest CPI, latency, and the execution time of dual merge sort, and had the fastest clock speed, despite utilizing the greatest FPGA resources. The dual-threaded configuration utilizes parallelism so that there are two cores executing instructions simultaneously, increasing the throughput of instructions and decreasing overall latency.

Overall, the results of the different CPU designs indicate that pipelining, caching, and parallelism improves performance. Although the simpler designs were resource-efficient, there was an apparent bottleneck in performance as the number of instructions and memory operations increase. As the complexity of the CPU design increased, the FPGA resources increased as well, but the total execution time got faster. In applications where speed is critical and hardware resources are not strictly constrained, it is worth utilizing more complex processors at the expense of greater FPGA resource consumption.

**Contributions:**

**Nikitha Suraj:** Modified multicore processor to include LR/SC instructions, wrote dcache source code, icache testbench, memory control source code

**Saandiya KPS Mohan:** Wrote palgorithm.asm, wrote icache source code, dcache testbench, memory control testbench

** all other work (designing the multicore RTL, integration with datapath, etc.) was done collaboratively in lab.