Processor Prototyping

Final Report

Yue Yin, William Wong

December 8, 2024

1 Executive Overview

This report explores performance comparisons among various processor designs: a single-cycle processor, a pipelined processor without caches, a pipelined processor with caches, a multicore processor designed for single-threaded programs, and a multicore processor designed to handle dual-threaded programs. All tests are conducted using the mergesort.asm assembly file for the single-threaded designs, while the dual-threaded multicore processor uses the dual.mergesort.asm file, ensuring consistent functionality and computational objectives across experiments. The metrics compared include estimated synthesis frequency, average clocks per instruction (CPI) with and without caches, average latency per instruction, total execution time with and without caches, the FPGA resources required for each design, and the speedup.

The single-cycle processor, known for its simplicity, avoids hazards and forwarding, making it resource-efficient but slower due to its inherently lower frequency and higher CPI. Pipelined designs, while more complex, address hazards and implement forwarding to improve performance, with the inclusion of caching further reducing latency and execution times. The dual-threaded multicore design leverages parallelism with dual.mergesort.asm, achieving even greater efficiency by processing multiple threads simultaneously. Data analysis, derived from synthesis and sweep reports generated via the Makefile and sweep script, highlights the trade-offs between hardware complexity, resource usage, and performance. While multicore processors require more resources, they deliver significant execution time reductions, particularly for parallel workloads.

2 Processor Design

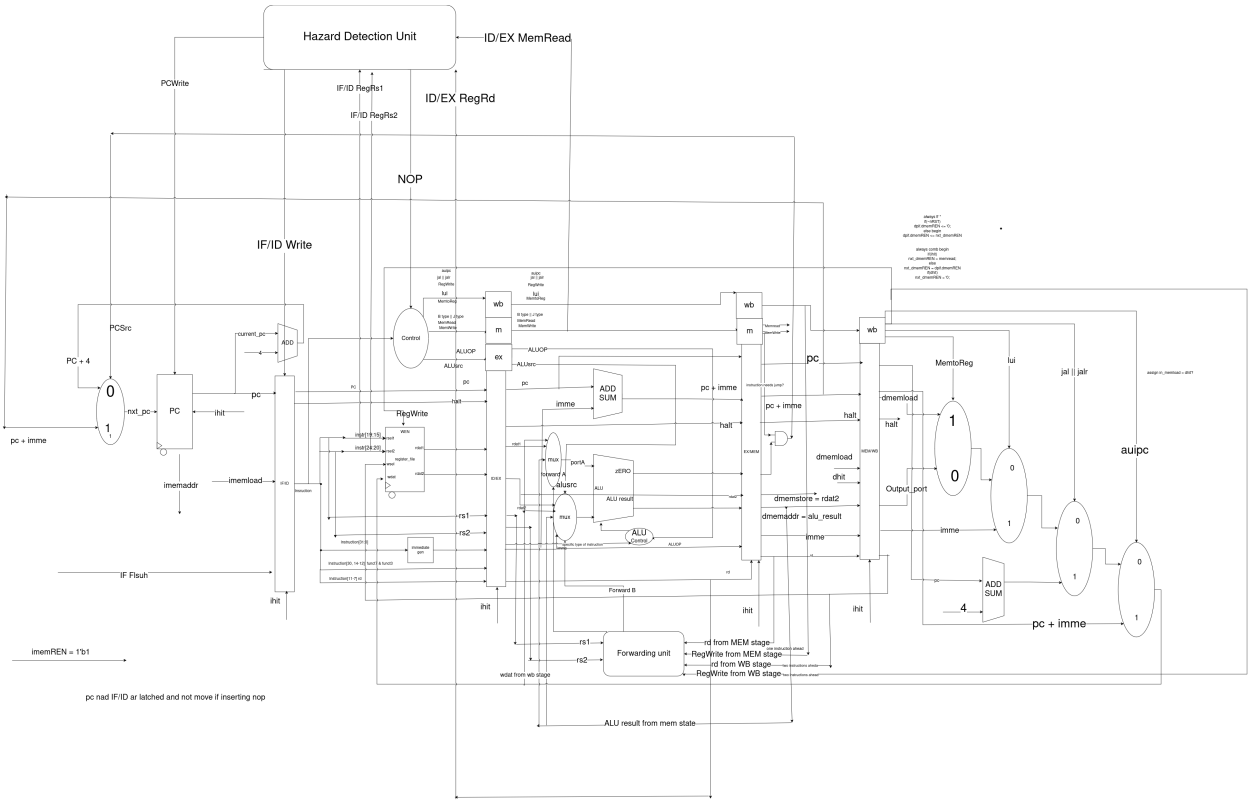


Figure 1: Pipeline Block Diagram

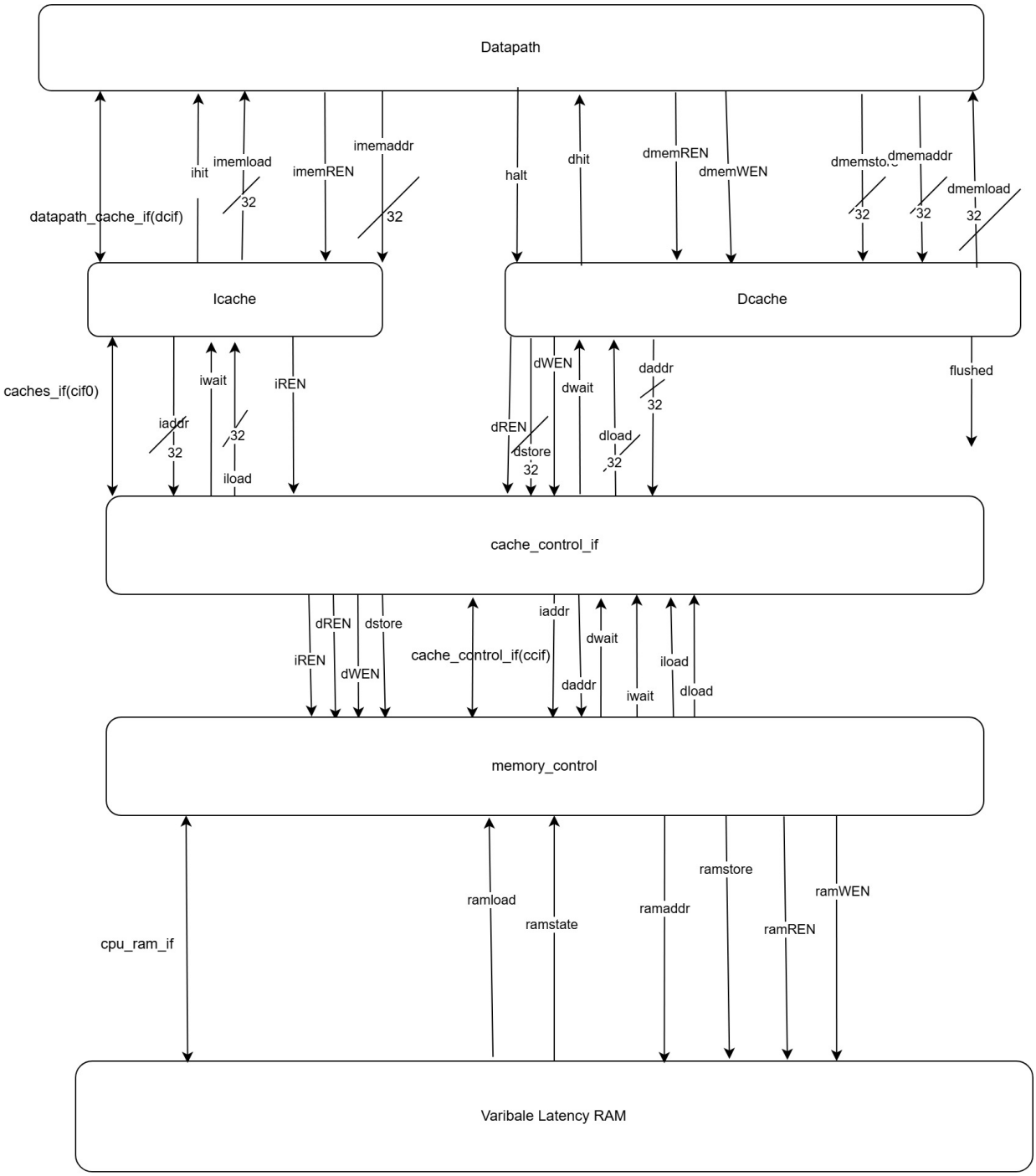


Figure 2: Pipeline Cache Block Diagram

The pipeline cache block diagram illustrates the integration of L1 instruction and data caches with the pipelined CPU architecture. By using the dcif, cif and ccif interface, the datapth, cache and main memory can communicate each other.

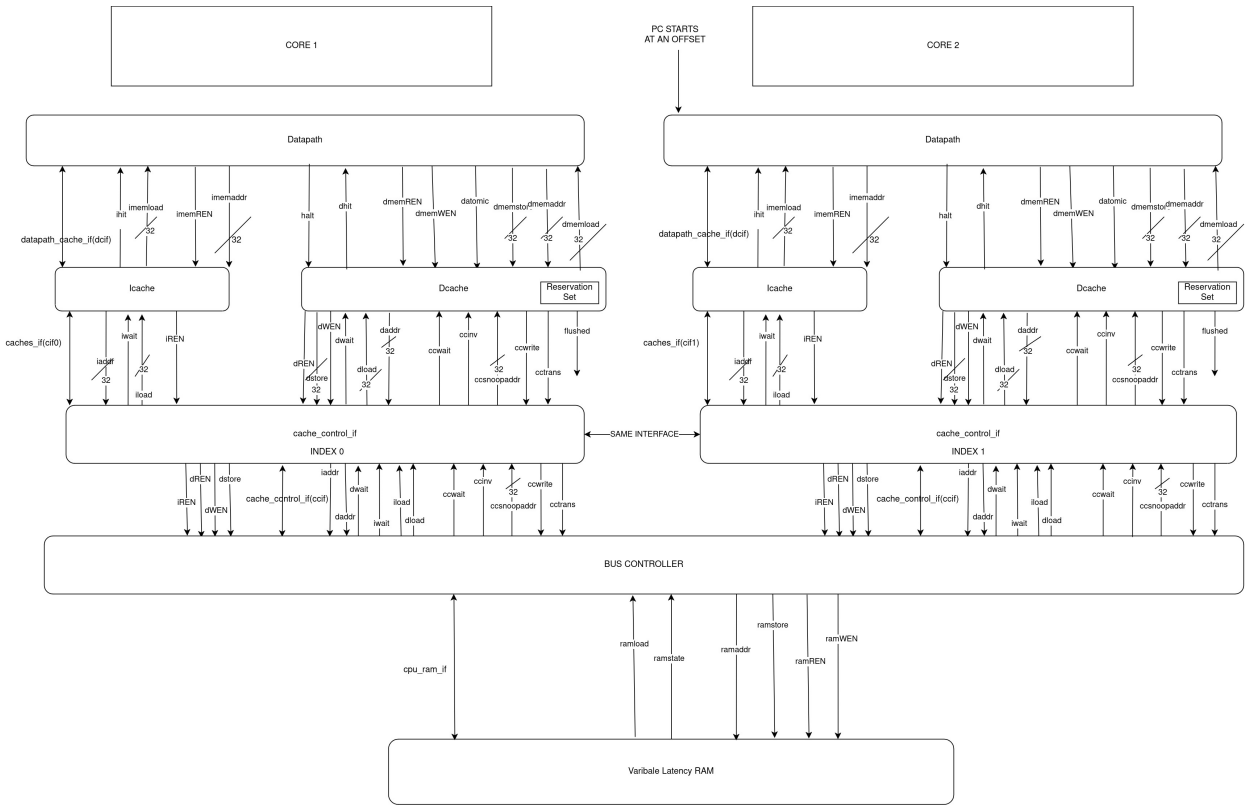


Figure 3: Multicore Block Diagram

In this diagram, we can see that we have a bus controller that is a combination of memory controller and memory arbiter. We add the datomic siganl as well for the lr/sc to signal the reservation register near the cache what it needs to do. the ccif interface will choose what signal to choose from between two cores beased on the memory arbiter. Two cores will take turns to give instruction to icache of two cores. As in normal memory controller, the data signal will take priority than the instruction signal. To cut the critical path, we need to latch the long siganls between the bus controller and the caches including dwait and dload. By adding these signals, we need some wait states to wait for the part to get the real signal to make everything correct.

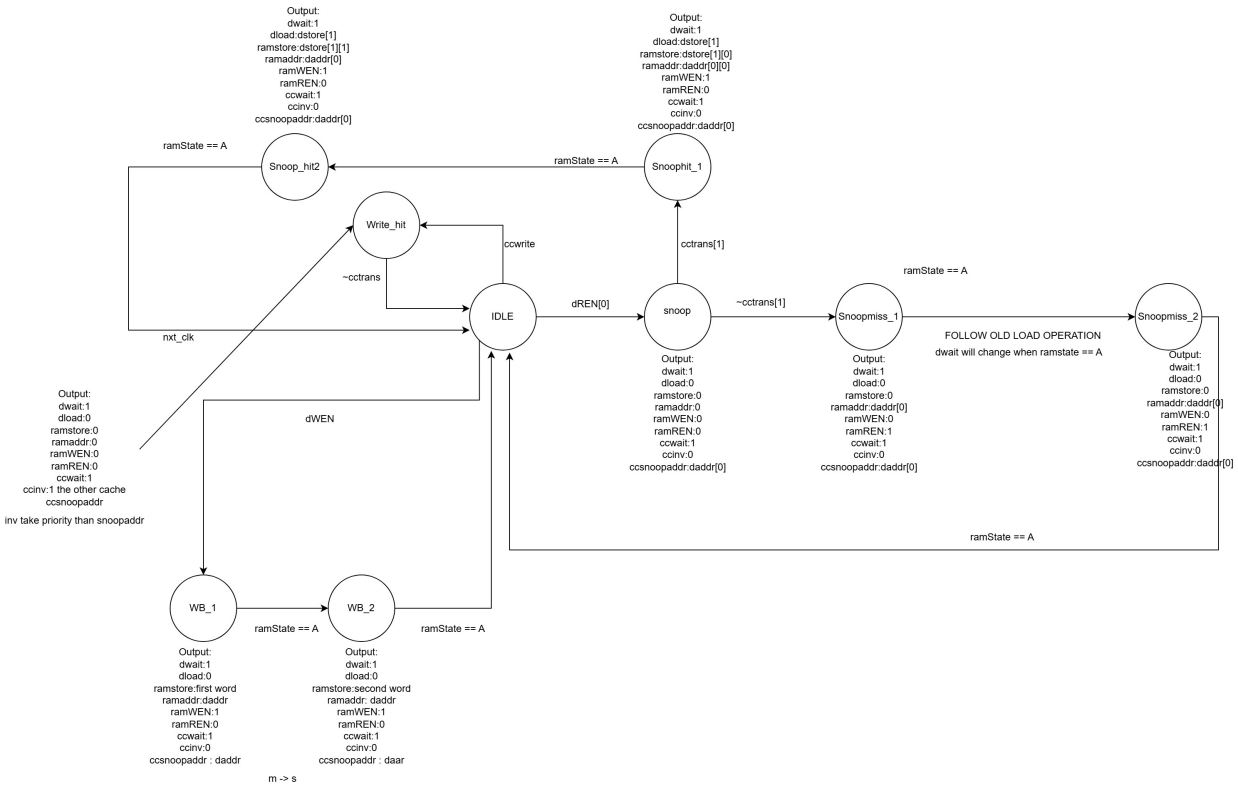
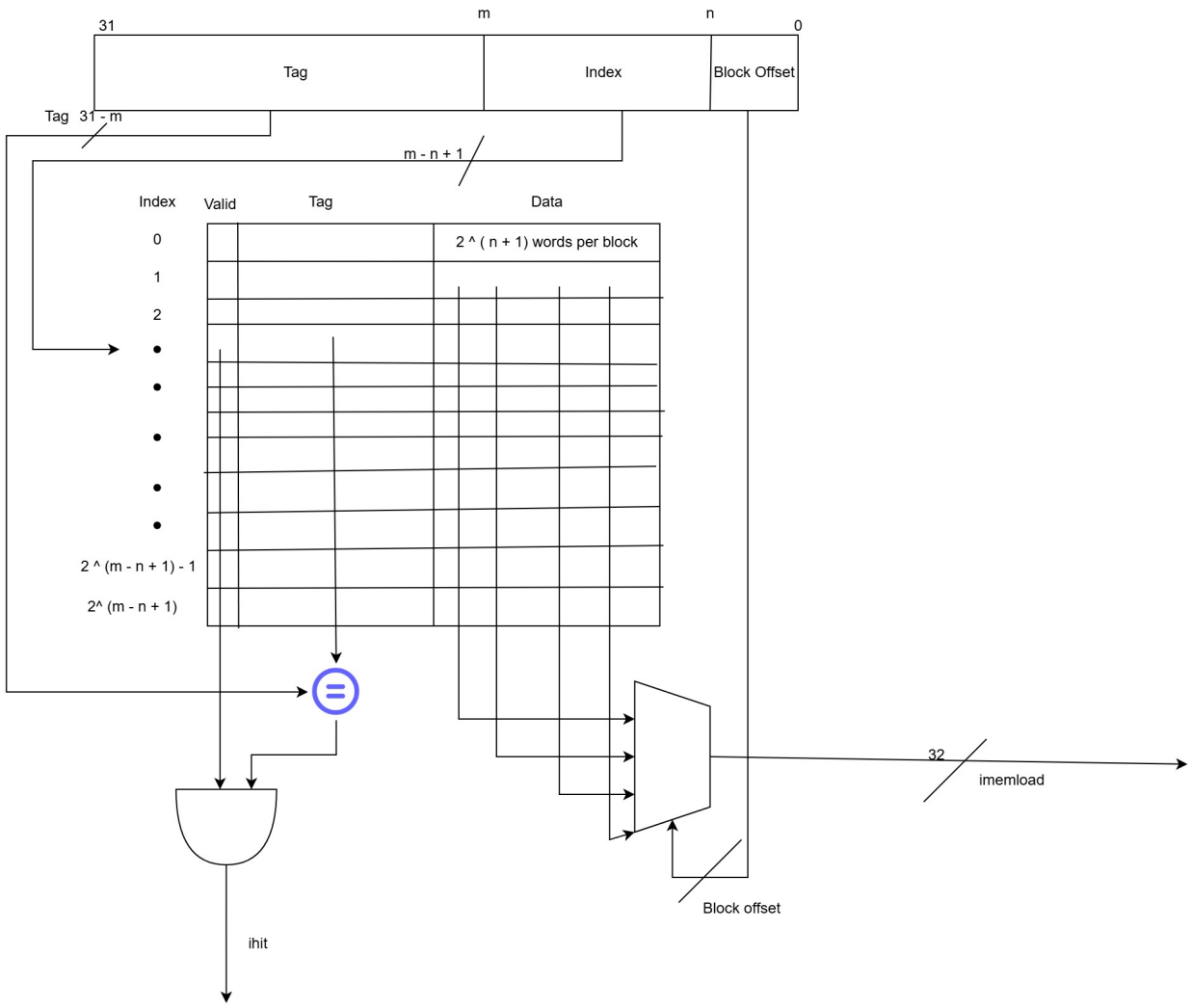
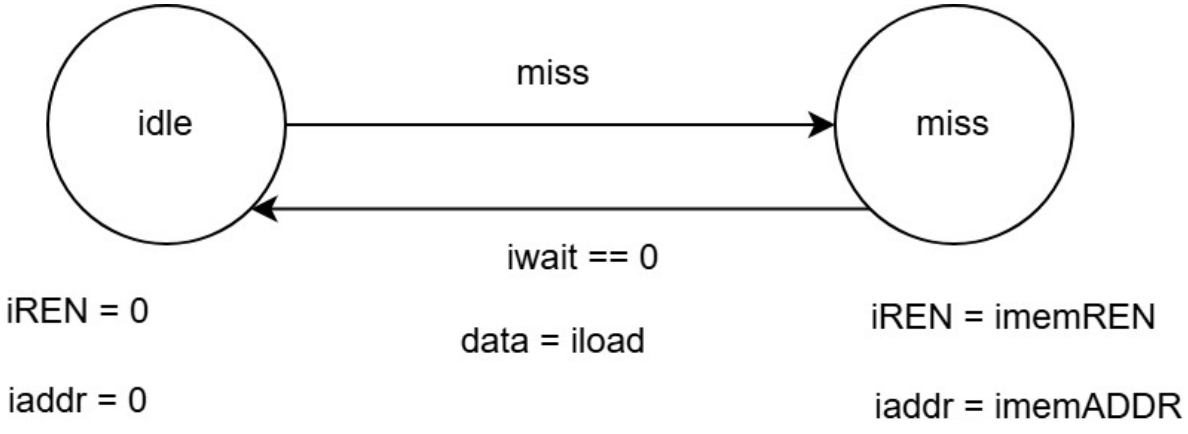


Figure 4: Bus Controller FSM

This is the bus controller FSM, it basically has three paths. One is when the cache has a miss and the core will first snoop from the other core. If the other core does have the data then it will do a cache to cache transfer and the other core will store the data to main memory for the memory coherence. If the other core doesn't have the data then this core will directly get data from main memory. The second path is when the cache needs to do a write to the main memory like writing back all the dirty values to the main memory when the program is done. The third path is when one core does a hit write so it needs to tell the buscontroller and invalidate the other core block if it has.



(a) I-Cache Table



(b) I-Cache FSM

Figure 5: I-Cache Table and FSM Combined

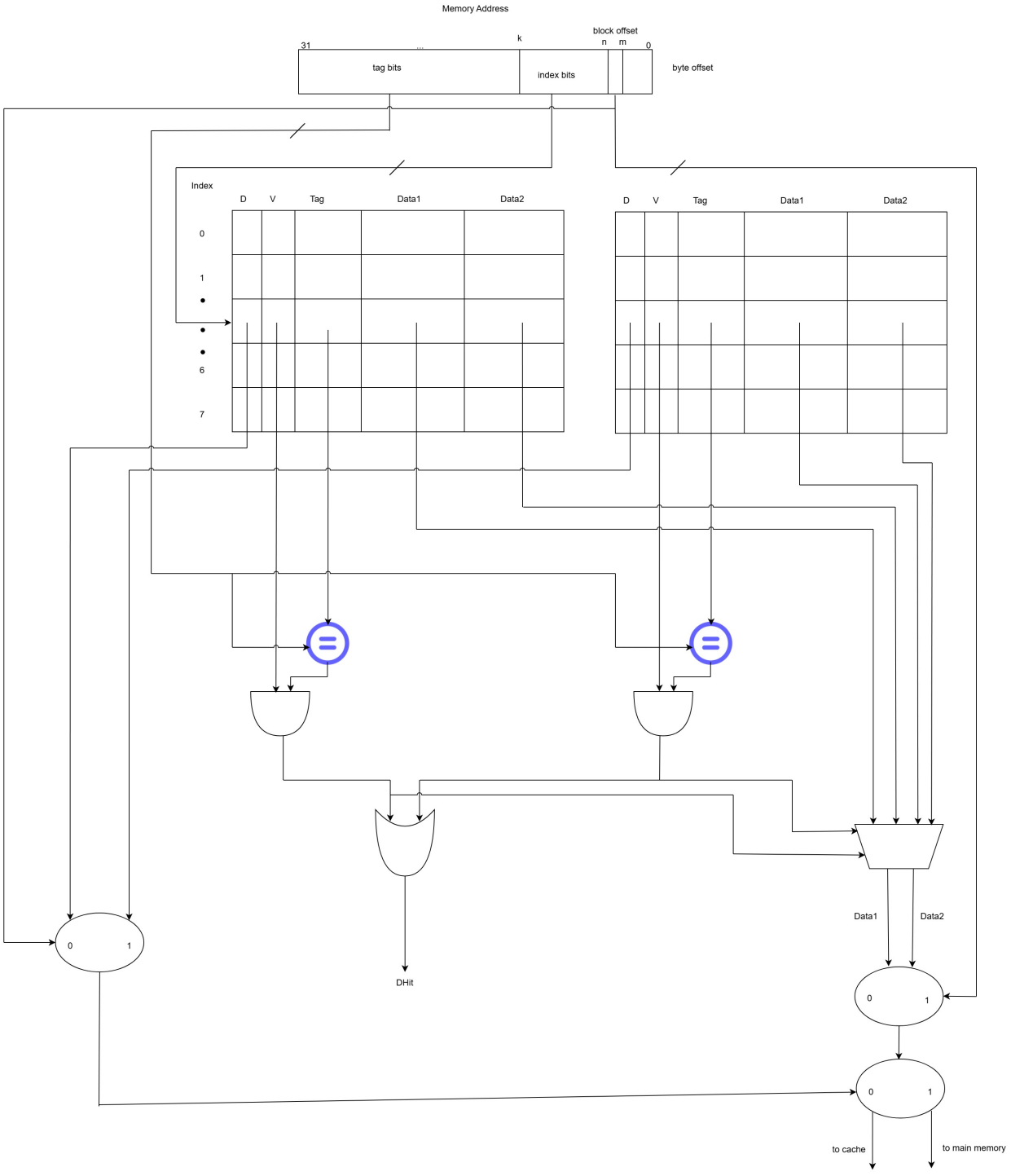


Figure 6: D-Cache Table

This is the micro-architecture for 2-way dcache. The instruction is split into tag bits, index bits, block offset and byte offset. We compare the input tag and the tag inside dcache, it's a hit if same.

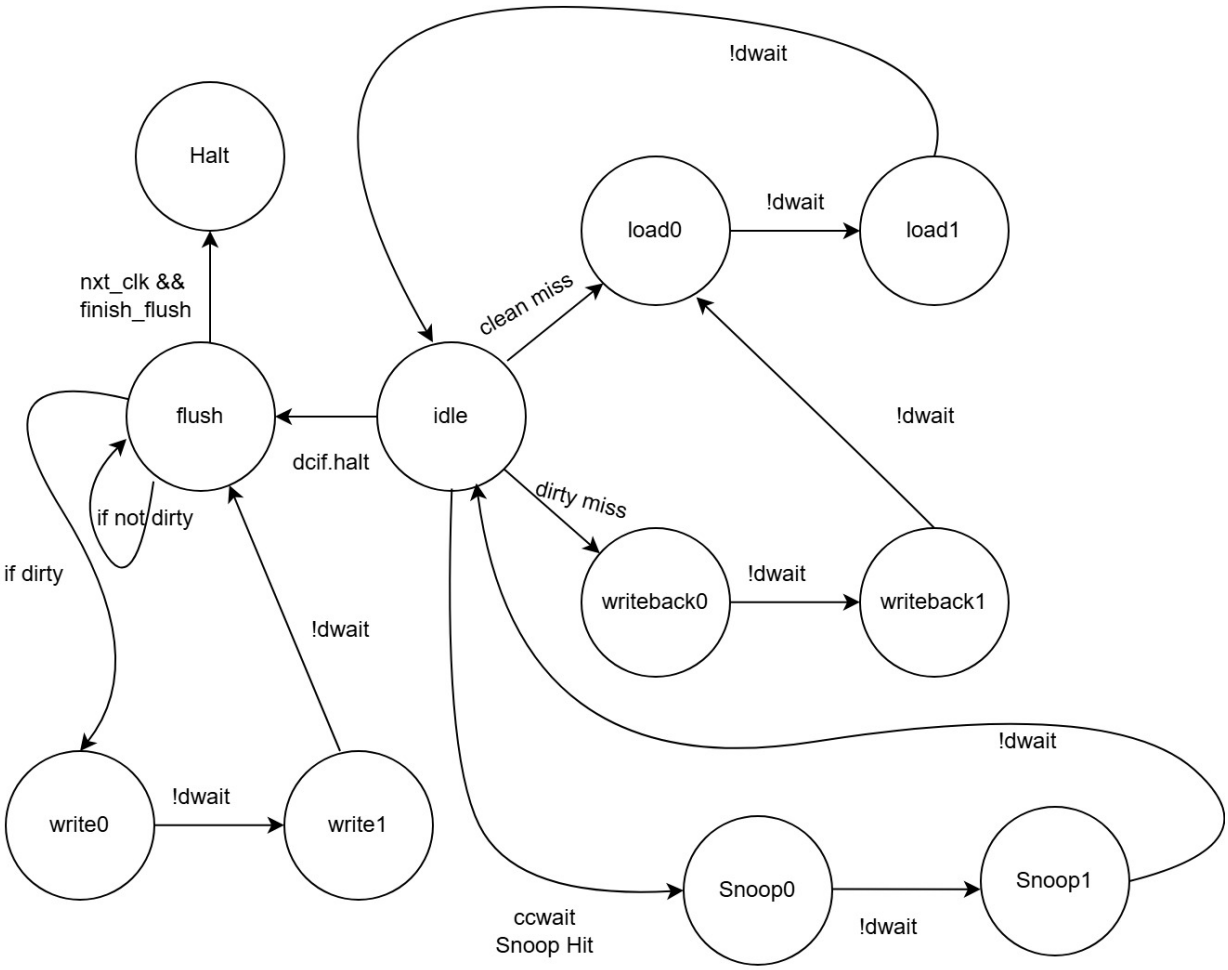


Figure 7: D-Cache FSM

In this D-Cache FSM, we can observe that if the desired block cannot be found or is marked as invalid, the system will either snoop to the other core or load the block from main memory. For normal write operations or write-back to main memory, two steps are taken since each block contains two words. It is important to note that in the multicore configuration, hit counts are not necessary, simplifying the design and improving efficiency. This FSM ensures coherent and efficient handling of memory operations within the D-Cache system.

3 Results

Processors	LAT	SYN FREQ(MHz)	CPI	LATENCY(s/instr)	TOTAL EXEC TIME(s)	REGS	LOGIC ELEM	SPEEDUP
Single-Cycle	6	32.42	5.11	1.58e-7	8.52e-4	1293	3145	N/A
Pipeline w/o Caches	6	56.24	7.48	6.65e-7	7.19e-4	1803	3730	18%
Pipeline w/ Caches	6	47.66	1.68	1.77e-7	1.91e-4	4277	10,110	346%
Multicore w/ Single-Thread	6	52.64	1.85	1.76e-7	1.90e-4	8704	21,781	348%
Multicore w/ Multi-Thread	6	52.64	1.14	1.08e-7	1.17e-4	8704	21,781	628%

Table 1: Mapped Design Results w/ Mergesort Test Program

‘

LAT	TOTAL EXECUTION TIME(s)	Caches Help?
0	1.84e-4	N/A
2	4.26e-4	N/A
6	8.52e-4	N/A
10	1.23e-3	N/A

Table 2: Single-Cycle Processor Specs

LAT	TOTAL EXECUTION TIME(s)	Caches Help?
0	1.79e-4	N/A
2	3.60e-4	N/A
6	7.19e-4	N/A
10	1.08e-3	N/A

Table 3: Pipeline Processor Specs Without Caches

LAT	TOTAL EXECUTION TIME(s)	Time Difference(s)	Caches Help?
0	1.41e-4	N/A	N/A
2	1.76e-4	3.5e-5	N/A
6	1.91e-4	1.5e-5	YES
10	2.06e-4	1.5e-5	YES

Table 4: Pipeline Processor Specs With Caches

LAT	TOTAL EXECUTION TIME(s)	Time Difference(s)	Caches Help?
0	1.55e-4	N/A	N/A
2	1.72e-4	1.7e-5	N/A
6	1.90e-4	1.8e-5	NO
10	2.09e-4	1.9e-5	NO

Table 5: Multicore Processor Specs for Single-Thread Programs

LAT	TOTAL EXECUTION TIME(s)	Time Difference(s)	Caches Help?
0	7.15e-5	N/A	N/A
2	8.85e-5	1.7e-5	N/A
6	1.17e-4	2.9e-5	NO
10	1.44e-4	2.7e-5	NO

Table 6: Multicore Processor Specs for Multi-Thread Programs

We compared 5 processors based off of 7 parameters, at a constant memory latency of 6: synthesis frequency, clocks per instruction(CPI), average latency of one instruction, total execution time, total number of registers, total logical elements, and speedup. LAT is just the variance of latency values for the ram to extend how long each instruction takes. Max frequency is determined by the sweep script that will synthesize our processors against the mergesort and dual mergesort assembly files. To calculate CPI we will use the command "make system.sim" to get the testbench clock, however, to get the CPU clock we would need to divide by 2. Additionally, we will use "sim -t" to get how many total instructions are in the mergesort assembly file. Cpi is calculated by dividing the CPU clock by the total instructions in mergesort. For multicore, we would additionally need to divide by 2 to account for the two cores. To calculate total execution time we take the inverse of our max frequency times cpi times the total number of instructions per program which is Iron Law:

$$TotalExecutionTime = \frac{sec}{cycle} * \frac{cycles}{instr} * \frac{instr}{prog}$$

We can calculate latency in turn by dividing our total execution time by total instructions times the total number of stages/pipes:

$$Latency = \frac{TotalExecutionTime}{TotalInstructionsInMergesort} * NumberofStages$$

We are able to pull the FPGA values like total registers and total logical elements from the system.summary file that is generated after we synthesize a processor. Finally, we are able to calculate speedup by taking the old excecution time over the new execution time. We will base all of our speedup for the other processors against the single-cycle execution time.

The bottom four tables each show every processor's memory latency, total execution time, time difference between consecutive latencies, and if caches helped at all. We are only looking for the first latency the caches help with performance.

4 Conclusion

Finally, it is evident by comparing a pipelined processor with and without caches the major benefit of utilizing caches. By lowering access latency, a cache helps to minimize the time needed for memory-related activities including load and store instructions. Under higher latency settings, such a latency of six cycles, where the CPU with caches has a rather superior CPI, this benefit is very clear. By means of caches, the processor can rapidly access instructions or data without stalling for many clock cycles to reach main memory, therefore greatly enhancing execution efficiency. The average memory access time (AMAT) equation states that, with stable clock frequency, a lower CPI directly results in improved performance. Including caches thus not only helps to alleviate memory congestion but also increases general system performance, particularly in high-latency situations.

Given its capacity to distribute the task between two cores, a multicore CPU clearly provides superior performance than a pipelined processor with caches. Tasks are spread among the cores in a multicore system so they may run instructions in parallel, hence lowering the overall running time for a given workload. Given that the effective workload per core is halved under ideal conditions, this parallelism greatly increases the average CPI relative

to a single-core pipeline with caches. Furthermore, the inclusion of caches in the multicore architecture lowers memory access latency, thereby allowing each core to process instructions more effectively and so improves speed. The multicore processor is therefore better than a single-core pipelined processor with caches since it achieves a compounded performance benefit from both parallelism and enhanced memory access.