Assignment Name: Project 2: Pipelined CPU Simulation with gem5 and HP Cacti

Class Name and Section: ECE451-001 **Date Submitted:** 12/17/2025

Group Members: Raghav Bharath (rsb49), Arnav Revankar (avr33), and Nicholas Rawle (nar9)

Introduction:

For this project, we got to work with pipelined CPU simulations using gem5, as well as a tool called HP Cacti. We began by running a simple DAXPY loop (Double precision AX Plus Y) utilizing the TimingSimpleCPU from Gem5. This CPU (as the name suggests) has a simpler design and has *no pipelined execution*. This means that instructions that would otherwise benefit from pipelined execution will not benefit at all. We ran this simulation both with and without *pseudo-instructions*, which allow Gem5 to capture simulation statistics. In part 2, we used these pseudo instructions to capture the various operation classes (load, store, branch, etc) for the initialization, computation, and post-processing blocks. Then, we ran the same simulation using the MinorCPU, which has *pipelined execution* (4 stages, F1, F2, DEC, EX) [1], and measured the resulting changes in latency, throughput, and total simulation time. Finally, we used HP Cacti to experiment with various cache line sizes and associativity to optimize access time, cycle time, leakage energy, and more.

Part 1:

For Part 1, we wrote the DAXPY program (Y[i] = aX[i] + Y[i]), compiled it with -O2 to avoid unsupported x87 floating-point instructions, and simulated it using the TimingSimpleCPU in gem5 v25. The TimingSimpleCPU is a non-pipelined processor that completes arithmetic instructions in one cycle and memory operations in multiple cycles. The program sets up the RNG array using initialization code, then performs another loop for the DAXPY computation, and then performs post-processing for accumulation and output. To understand the results, we collected the instruction counts from stats.txt (commit-stage), since the op_class is no longer present in gem5 v25. Here's what we observed:

- numIntInsts (# of integer instructions): 163,460,398
- numFpInsts (# of float instructions): 63,414,017
- numLoadInsts (# of load instructions): 22,773,153
- numStoreInsts (# of store instructions): 12,185,808
- numBranches (# of branches fetched): 14,815,700

As we can see from the instruction counts, the high number of integer and branch instructions tells us that much of the execution is dominated by setup, loop control, and teardown code, most likely for control flow and address calculations. Also, the large number of floating-point instructions and memory loads and stores shows that the DAXPY loop is compute and memory-intensive, performing repeated floating-point arithmetic with regular accesses to arrays X and Y.

Part 2:

In Part 2, we used pseudo-instructions to isolate the region of interest within the program. We did this by inserting the m5_dump_stats(0,0) and m5_reset_stats(0,0) functions immediately before and after the DAXPY loop. This allowed us to reset the statistic counters and divide the execution into three blocks: Block A (initialization), Block B (DAXPY loop), and Block C (post-processing).

From Table 1, we can see that Block A is the heaviest block, as it's dominated by control flow and RNG. It also has a high number of floating-point instructions because initializing X and Y stores float values. Block B evidently isolates the DAXPY kernel, showing the expected mix of FP arithmetic with regular memory traffic (approximately 2 loads, 1 store, and 1 branch per iteration). Block C represents post-processing, and it's mainly integer and branch-heavy, with almost one load per iteration and nearly no stores, as the summation accumulates into a register.

Block	Int Insts	FP Insts	Load Insts	Store Insts	# Branches
A	146,422,773	51,412,441	18,767,019	11,182,395	12,811,847
B	9,000,014	7,000,003	2,000,002	1,000,001	1,000,002
C	7,037,574	5,001,527	1,006,116	3,405	1,003,850

Table 1

Below is the assembly generated by the g++ compiler, specifically for the DAXPY loop. The core instructions, `mulsd`, `addsd`, and `movsd`, are standard x86 instructions for multiplying, adding, and moving *scalar double-precision floating-point* numbers. The `addq` instruction then increments the rax register by 8 (size of a double), and `cmpq` compares the value of N (in the `r12` register) to `rax`. If they're the same, the Z-flag register is set to 1, allowing `jne` to pass. Otherwise, the branch is taken, and it loops back to `.L40`.

```
.L40:
      movq  %rbx, %xmm0
      mulsd 0(%r13,%rax), %xmm0
      addsd 0(%rbp,%rax), %xmm0
      movsd %xmm0, 0(%rbp,%rax)
      addq  $8, %rax
      cmpq  %r12, %rax
      jne   .L40
```

Part 3:

For Part 3, we executed the annotated DAXPY program using the MinorCPU, an in-order pipelined model. To study the impact of functional unit timing, we swept the Float/SIMD unit in BaseMinorCPU.py under the condition that opLat + issueLat = 7. We evaluated 6 configurations: (1,6), (2,5), (3,4), (4,3), (5,2), and (6,1).

opLat	issueLat	simSeconds	simTicks	IPC
1	6	0.040575	40574902000	0.172521
2	5	0.039146	39145981000	0.178818
3	4	0.037435	37434522000	0.186993
4	3	0.034787	34786948000	0.201225
5	2	0.034251	34251492000	0.204371
6	1	0.033874	33873723000	0.206650

Table 2

For each configuration, we only analyzed Block B (DAXPY loop) by extracting the stats corresponding to each section. As Table 2 shows, we recorded the simulation seconds, simulation ticks, and IPC for each configuration, with a constant instruction mix.

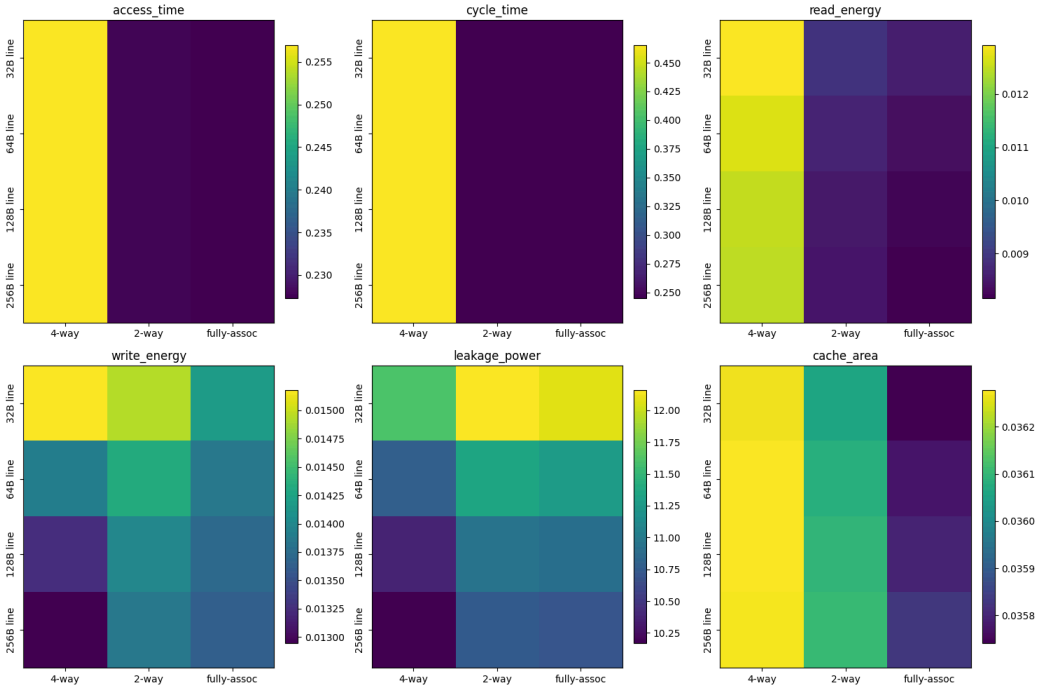
From the table, it's evident that as issueLat decreases, Block B's execution time consistently decreases and its IPC increases, despite opLat increasing. This shows that the throughput (issue rate) has a stronger impact on DAXPY

performance than individual operation latency. Because each DAXPY iteration is independent, the pipeline can keep issuing new FP operations and hide long latencies through overlap, as long as the issue rate remains high.

We can also see that as opLat increases, the improvements begin to stagnate because longer operation latencies eventually limit how much overlap the pipeline can achieve, despite the low issue latency.

Part 4:

In this part, we used HP Cacti to simulate L1 caches for the DAXPY loop. We used cacti/sample_config_files/ddr3_cache.cfg as a starting point, and then created these configuration files. We then tested various caches, ranging from 4-way to 1-way (fully-associative), and from 32B to 256B line sizes, using a 256-bit bus width for each. The resulting parameters have been visualized in the heatmap below.



The results we found were understandable, yet at times surprising. As you can see above, the heatmaps for access time, cycle time, read energy, and cache area showed virtually the same trends: smaller values for fully-associative caches, and larger values for 2- and 4-way caches. This makes sense, since on the architecture level, fully associative caches are more *flexible* (data in memory can go anywhere in cache), yet have more *overhead*, both in the spatial and temporal realms. 2- and 4-way caches, on the other hand, split up the cache into 2 or 4 ways, indicated by one or two bits of the memory address, respectively. This reduces flexibility, but also significantly reduces overhead; a 5x32 decoder will take up less space and energy than two 4x16 decoders. Below are some of the values we got for 2-way associative caches.

Design	**Line Size**	**Associativity**	**Access Time (ns)**	**Read Energy (nJ/access)**	**Total Leakage Power of a bank (mW)**	**Cache Area (mm^2)**
Design Y	32B	2	0.227732	0.0088955	12.1586	0.036063
Sweep	64B	2	0.227732	0.0086537	11.3546	0.036085
Design X	128B	2	0.227732	0.0085103	10.9519	0.036095

As for how these cache designs apply to DAXPY, each DAXPY loop reads 2 input operands and writes 1 operand, which is approximately 12B of memory traffic per iteration, making it bandwidth-intensive. For the fastest looping speed possible, it would be best to have the largest line size possible; however, the approach pays no regard to write energy,

cache area, or leakage power. A more realistic and optimal configuration would be a 128B 2-way associative cache, which minimizes read energy and leakage power, while improving bandwidth and keeping cache area and access times at acceptable levels.

Furthermore, a cache's access time can act as a bottleneck on a CPU's minimum issue latency. As we found in part 3, increasing a processor's operation latency and decreasing its issue latency directly helps the processor's performance (to a limit). The MinorCPU by default has a 64B 2-way set associative cache, which, from the CACTI simulation, puts it at 0.2277 ns access time. Since the MinorCPU runs at 1 GHz (1 cycle per nanosecond), the cache should be able to meet this demand easily, even at an issueLat of 1 with a higher frequency. In this case, the CPU could run at a maximum of 4.39 GHz (1 cycle per 0.2277 ns) for the CPU's pipeline to match the cache's access time. Thus, in this case, it is the CPU's opLat which is the bottleneck.

For the best performing cache, we should thus look to optimize reading and writing energy, leakage power, and cache area, since a slowdown on the cache-side will not impact the CPU's throughput. A 128B 2-way associative cache, as we mentioned before, will be among the optimal choices for such a configuration.

Conclusion:

This project used a DAXPY kernel as a workload to study and build intuition about pipelined CPU execution. DAXPY, a commonly used vector operation, combines floating-point arithmetic with memory accesses. We saw how instruction mix, pipeline structure, functional-unit latency, and throughput all affect performance. Part 1 of this project established a baseline, using a non-pipelined CPU (TimingSimpleCPU), which helps us reason with the instruction mix and instruction count. We were able to see the instruction composition without the complexity of overlapping execution. This allows us to observe the heavy floating-point instructions as well as the memory-heavy instructions, such as load and store, in our kernel. As a result, we can see that setup and teardown instructions are associated with the instruction mix due to the large number of integer instructions. This leads us into part 2 on why we needed to isolate the DAXPY loop, as this is our region of interest when evaluating the performance. As such, after generating the assembly, we can see what instructions are executed in the loop, where we see MOV, MUL, and ADD x86 instructions. After moving to a pipelined CPU, we can see exactly how functional unit timing can affect performance. Our results show that reducing the issue interval increases throughput even when the single-operation latency increases. Thereby showing that the latter has less impact on the performance. We come to realize that this is because the DAXPY iterations are independent, meaning they can be fully overlapped in a pipeline. In the last part, we used CACTI, a memory evaluation software, to examine the memory system design. By tweaking the L1 cache's line sizes and associativities, we can easily see how a workload like DAXPY favors high effective bandwidth. We learn that the 128B 2-way associative cache best balances our needs of latency, energy, bandwidth, and area. We learned from this project how overlapping execution and memory bandwidth dominate performance for data-parallel workloads by using the DAXPY kernel and simulating different throughput tradeoffs.

References:

[1]. "Gem5: Minor CPU Model." Gem5.org, 2025, www.gem5.org/documentation/general_docs/cpu_models/minor_cpu.