I. INTRODUCTION

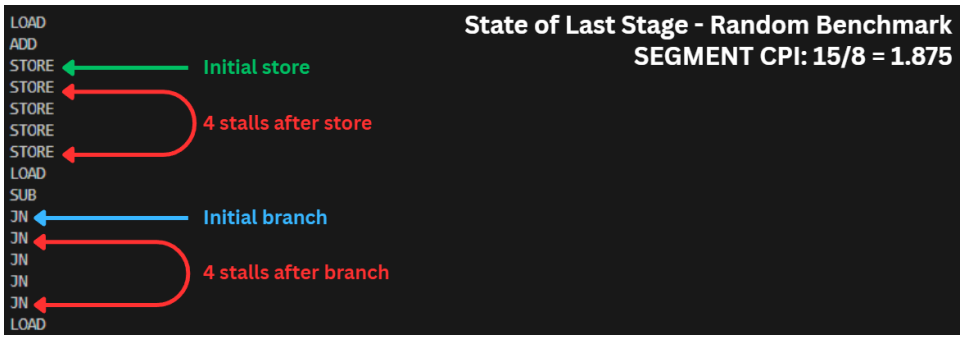
Throughout the semester, this class has built simulators for a toy instruction set, called simple twelve (S12). The previous submission demonstrated how to pipeline it. To prevent data hazards, that pipeline stalled if a fetched instruction was STORE or STOREI. That new instruction had to propagate to the final stage before the pipeline could release the stall. Consequently, each time the program counter fetched STORE or STOREI, the pipeline stalled four cycles. This caused the pipeline to stall at least sixty percent of the time, regardless of the benchmark. Therefore, this report will demonstrate how to improve the previously submitted design, significantly reducing the percentage of time lost to stalls. This report starts by quantifying the stall from the previous design. Second, it provides alternate control logic, attempting to decrease stalls and increase performance. Third, this report will evaluate simulation results of the new design, focusing on the CPI improvement percentage of each benchmark. Finally, this document summarizes the results of this optimization attempt and suggests further testing of the new design.

II. QUANTIFYING THE STALL ISSUE

Before jumping into a solution, it is helpful to identify the scale of the issue this report aims to fix. Below, the table summarizes the percentage of time the previous pipeline spent stalling for each benchmark. Although each benchmarks percentage follows a trend, the variation is small and the averages between benchmarks are similar. When inputs fall between zero and five, the smallest average time spent stalling is 58.52%, and the largest average time spent stalling is 62.31%. This is only a difference of 3.79%. It is important to note that that the ISA's twelve-bit memory limits factorial operations to N = 6. Alternatively, the max number of multiplication operands is 12.

	Percentage of Time Spend Stalling By Benchmark (Old Pipeline)			
N	Speed Multiply	Space Multiply	Speed Factorial	Space Factorial
0	57.14	72.73	65.57	65.57
1	60.00	57.14	65.57	65.57
2	58.73	58.54	60.78	60.79
3	58.71	59.11	60.72	60.68
4	58.36	59.32	60.70	60.63
5	58.18	59.41	60.69	60.59
Average	58.52	61.04	62.34	62.31

While it is no surprise that these benchmarks had extreme amounts of STORE and STOREI operations, these high stall percentages shocked the author. Below, the snapshot of a random benchmark shows the state of the last stage. This snapshot shows that the pipeline stalled four clocks after a store. Then another four after a conditional. Notably, these two stalls nearly doubled the snapshots CPI, as each stall causes the pipeline to lose 4 clock cycles.



While the image above is only a snapshot of a single benchmark. Its behavior is consistent across the entire length of each benchmark. Each benchmark is primarily composed of stalls, explaining why the benchmarks' stall averages are near sixty percent. It also indicates that stalls are common after memory modification operations and branch instructions.

While the image above demonstrates the storage and conditional instructions cause stalls, the table below associates each instruction type with the percentage of the stalls that it initiated. Notably, STORE instructions caused 60% to 70% of the stalls in each benchmark. However, the multiplication benchmarks had zero stalls caused by STOREI instructions while 17% to 20% of the factorial benchmark stalls came from STOREI instructions. Across all benchmarks, JMP instructions caused 16% to 20% of stalls. However, the stalls caused by conditional jumps were negligible.

Speed Optimized Multiplication	Percentage of Stalls Caused by Operation Type						
	N	JMP	JN	JZ	STORE	STOREI	Total Stalls
	1	0.00	0.00	33.33	66.67	0.00	12
	2	16.98	0.00	15.09	67.92	0.00	212
	3	17.65	0.00	14.71	67.65	0.00	408
	5	18.00	0.00	12.50	69.50	0.00	800
	15	18.26	0.00	10.87	70.87	0.00	2760

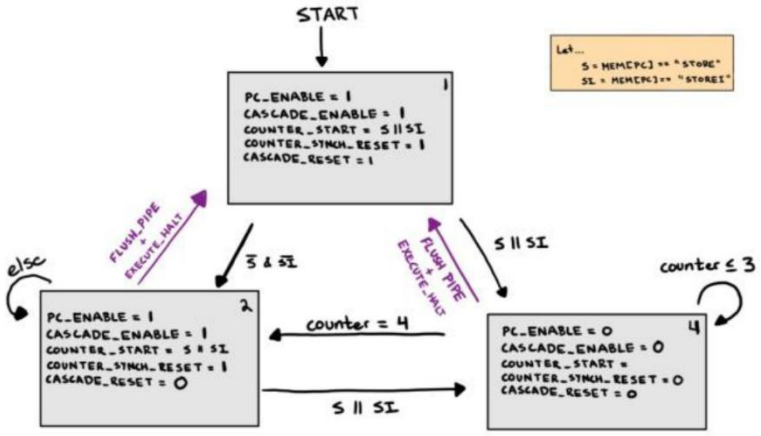
Memory Optimized Multiplication	Percentage of Stalls Caused by Operation Type						
	N	JMP	JN	JZ	STORE	STOREI	Total Stalls
	1	0.00	0.00	33.33	66.67	0.00	12
	2	22.22	0.00	11.11	66.67	0.00	72
	3	23.33	0.00	10.00	66.67	0.00	120
	5	22.22	0.00	11.11	66.67	0.00	180
	15	19.05	0.00	14.29	66.67	0.00	420

Space Optimized Factorial	Percentage of Stalls Caused by Operation Type						
	N	JMP	JN	JZ	STORE	STOREI	Total Stalls
	0	0.00	10.00	0.00	70.00	20.00	40
	2	18.90	0.82	0.55	60.55	19.18	1460
	4	19.45	1.15	0.53	60.30	18.56	4504
	5	19.58	1.37	0.52	60.25	18.28	6128
	6	19.69	1.59	0.51	60.20	18.01	7820

Speed Optimized Factorial	Percentage of Stalls Caused by Operation Type						
	N	JMP	JN	JZ	STORE	STOREI	Total Stalls
	0	0.00	10.00	0.00	70.00	20.00	40
	2	18.22	1.12	0.74	60.97	18.96	1076
	4	18.68	1.56	0.72	60.84	18.20	3340
	5	18.74	1.84	0.70	60.86	17.86	4568
	6	18.77	2.12	0.68	60.89	17.54	5860

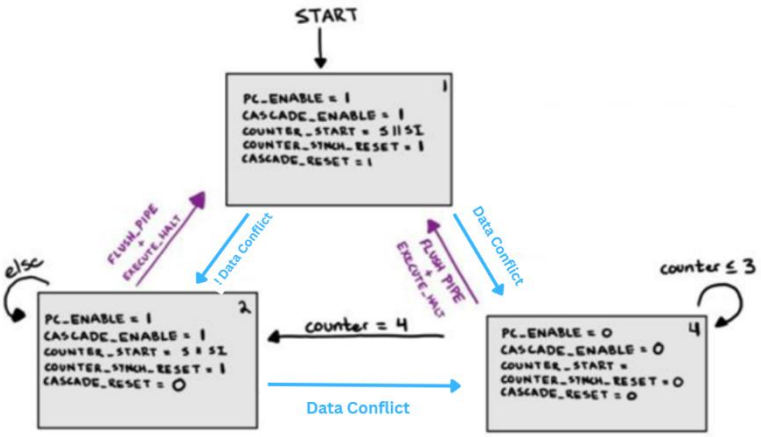
III. SMARTER STALL LOGIC

The previous section revealed that across all benchmarks, approximately sixty percent of clocks are stalls. It also revealed that STORE instructions are responsible for at least sixty percent of stalls for each benchmark. Consequently, improving the stall logic appears to be the highest impact improvement. This section will demonstrate a simple solution to improve the pipeline stall logic, drastically improving performance.



Above, an image reveals the old pipeline's finite state machine. Notable, there are three states. The first state resets the entire pipeline by disabling the registers between stages. The pipeline is in this state when starting, or after a flush. State two, in the lower left, is for normal operation. The third state, in the lower right, is for stalling. If the pipeline encounters a STORE or STOREI instruction while in stage one or stage two, the next state will be stage three. Primarily, state three is different because it disables the '*cascade_enable'* control signal. Then, after four clocks, the fsm will return to the normal operation in state two.

This design is good because it eliminates the risk of data hazards. However, as seen in the previous section, it has bad performance. Fortunately, it is not necessary to stall for each storage instruction, as data hazards only occur in specific conditions. Therefore, the finite state machine can be abstractly represented with the following image.



According to the image above, stalling should only happen when there is a data conflict. Data conflicts only happen when a new operation tries to use a memory location before a previous storage instruction finished modifying it. Therefore, our pipeline stalling logic needs to…

1. Keep a list of addresses that are busy. Whenever the pipeline fetches a storage instruction, the pipeline should add the target address to the list. For a store indirect operation, this address is not the eight-bit operand of the instruction.
2. Keep a separate list of counters. When the pipeline adds an address to the busy list, the control logic starts a new counter and adds it to the counter list. It is important for the indices of the address and its counter to match.
3. Increment each counter in the list on each clock.
4. On each clock, remove a timer from the list if it has reached its max value. The max value is the number of clocks that the original storage instruction needs to finish. When removing a counter, its corresponding busy address should also be removed.

On each clock, there is a data conflict if the fetched instruction's operand is in the list of busy addresses. This approach is an excellent way to minimize store operations. However, this logic leads to a common bug. If the pipeline stalls while a storage instruction is in the first stage, that storage instruction will sit in the first stage for four clock cycles. According to the logic above, this will add the target address of the stalled storage instruction to the list of busy addresses four times. When the pipeline resumes operation, it will immediately stall because the store instruction will be in stage one, and its operand will be in the list of busy addresses. This causes a runaway affect that deadlocks the pipeline.

Notably, the '*pc_enable'* control signal will be high only for the first clock that the storage instruction is in the first stage. During a stall, it is low. Consequently, when a storage instruction is in the first stage, its target address should only be added to the list of addresses if the '*pc_enable'* control signal is high.

IV. QUANTIFYING IMPROVEMENT OF BENCHMARKS

The last section provided an alternate control logic that would reduce the number of stalls. Integrating those suggestions into the old pipeline's simulator resulted in the following table. Notably, the smallest factorial benchmark improvement was 40.94%. Similarly, the speed multiplication benchmark saw 35% to 39% improvements. Strangely, the memory optimized multiplication benchmark did not improve, but got worse. It started with a negative improvement of 44%. That percentage magnitude does drop off but remains negative.

Speed Optimized Multiplication	CPI		
N	Old	New	% Improvement
1	2.500	2.500	0.00
2	2.423	1.564	35.45
3	2.422	1.530	36.82
5	2.391	1.492	37.61
15	2.370	1.433	39.53

Memory Optimized Multiplication	CPI		
N	Old	New	% Improvement
1	1.3333	1.923	-44.23
2	1.4118	1.655	-17.23
3	1.4458	1.649	-14.06
5	1.4634	1.661	-13.50
15	1.4841	1.711	-15.29

Space Optimized Factorial	CPI		
N	Old	New	% Improvement
0	2.9048	1.6400	43.54
2	2.5499	1.4877	41.66
4	2.5446	1.4926	41.34
5	2.5440	1.4956	41.21
6	2.5436	1.4968	41.15

Speed Optimized Factorial	CPI		
N	Old	New	% Improvement
0	2.9048	1.6400	43.54
2	2.5504	1.4900	41.58
4	2.5399	1.4926	41.23
5	2.5375	1.4949	41.09
6	2.5356	1.4971	40.96

Attempting to explain the performance decrease, the authors of this paper speculated the original design would only stall 4 clock cycles on a stall, but the new design might stall for more than 4 if the stall is delayed. In other words, the finite state machine of the previous section shows that stalling adds four clock cycles. In the previous design, these four clock cycles happened every time the pipeline fetched a storage operation. However, the new design holds onto the affected address of a storage operation for four clock cycles, only initiating a stall if a fetched instruction tries to use that address during those four clock cycles. Only then, if a fetched instruction tries to use a busy address, does the pipeline stall four clock cycles. However, this could happen when the storage instruction is nearly completed.

For example, imagine the pipeline fetches a storage instruction and adds it to the list of busy addresses. Then, the pipeline latches two new instructions that do not present a data conflict. Then, the pipeline latches an instruction that relies on the busy address. This starts the stall sequence, and the pipeline will stall for an additional four clock cycles.

Initially, the authors speculated that this situation would cause more than 4 stalls for a single storage instruction. However, they now believe that this is not the case, as the pipeline did not lose the initial 3 clock cycles. Although the stall was delayed, the pipeline only stalled four clocks. This is particularly interesting, because the authors verified that each benchmark calculated the correct output for each input N. Since the authors cannot explain the decrease in performance, they believe that the new design is not safe, and needs further testing.

VI. CONCLUSION AND FUTURE WORK

The previous pipeline achieved safety with an unnecessary number of stalls. This report presented alternative control logic, intended to reduce the 60% percent stall rates of the previous design. When this logic was implemented in simulation, three of the four benchmarks saw CPI improvements greater than 40%. However, the cpi of the memory optimized multiplication benchmark increased significantly. The authors of this paper attempted to explain this decrease in performance but have not been able to identify the issue. This leads them to believe that there is a logical error in the new design, that the other benchmarks did not identify. Consequently, the authors cannot confirm that the new design is safe.