I. INTRODUCTION

Pipelining a processor's datapath is a common way to improve program execution time. It works by separating hardware access into stages, like an assembly line. This is effective because a single instruction requires access to multiple hardware elements, but not simultaneously. For example, the addition instruction may need to read from a register, use the ALU, then write to a different register. In this example, two elements idle during each step. Instead, a pipeline allows multiple instructions to execute at the same time by coordinating hardware use. Although pipelining does not reduce the time to complete a single instruction, it increases the number of instructions that complete per cycle. As a result, programs execute faster.

Unfortunately, pipelines come with additional complexity. They also create new risks, or hazards, that the pipeline must detect and prevent. To demonstrate these hazards, as well as solutions, this report demonstrates how to pipeline an ISA with twelve instructions. To keep it simple, the datapath will be accumulator-based. Additionally, program memory will not be separate from data memory.

The rest of this report has five sections. The first section demonstrates the design of a pipelined datapath. Second, this report explains the design of a simulator. The third section presents simulation results. The fourth section evaluates those results. Finally, the last section discusses the simulation results and suggests further optimization outside the scope of this project.

II. DESIGN OF DATAPATH AND CONTROL LOGIC

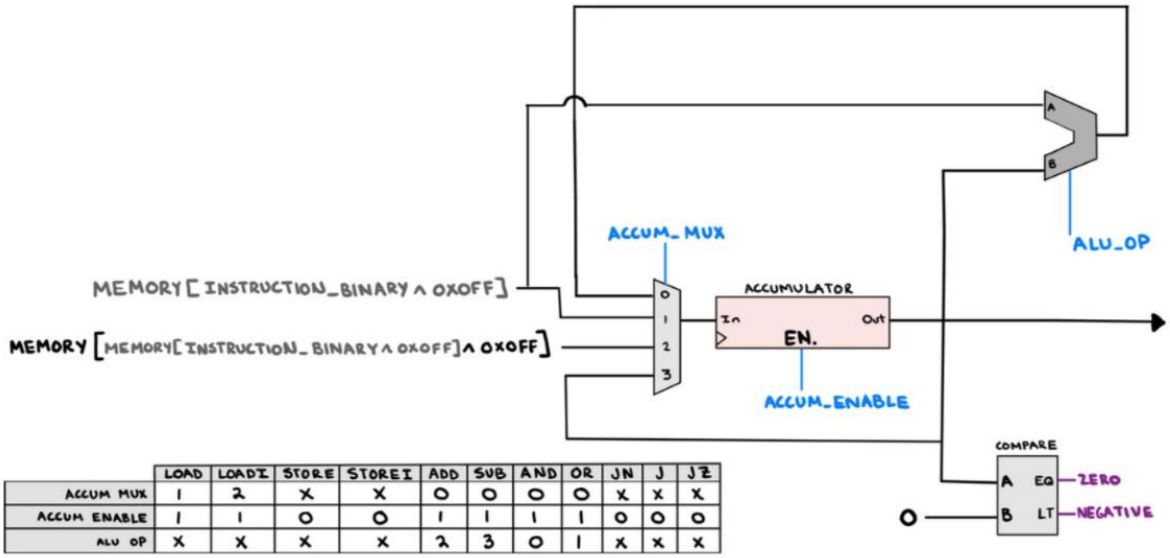
A. General Datapath Design

The heart of this processor is the accumulator. Consequently, this design revolves around it. Analyzing the instructions in the appendix, there are four ways to change the accumulator. They are as follows:

- *ALU Operations:* The four ALU operations use the bottom 8-bits of their instruction as an address for an ALU operand. The other ALU operand is the accumulator itself. The result of the ALU should load into the accumulator.
- *LOAD Operation:* The load operation uses the bottom 8-bits of its instruction as an address of a value that should load.
- *LOADI Operation:* The load indirect operation uses the bottom 8-bits of the instruction as an address. However, the bottom 8-bits of the value at the address is also an address. The accumulator should load the value at the second address.
- *Other Operations:* The other operations should not change the accumulator.

The image below summarizes these four behaviors. A mux determines which behavior feeds into the accumulator, controlled by the signal 'accum_mux'. The ALU also has a select line 'alu_op', which determines how the ALU should combine the two operands. Additionally, the image shows an extra signal 'accum_enable', which enables or disables the accumulator latching. Finally, the table shows the value of each blue control signal for each operation. Notably, the table does not include HALT. This stage of the design ignores it. Also, the table does not use

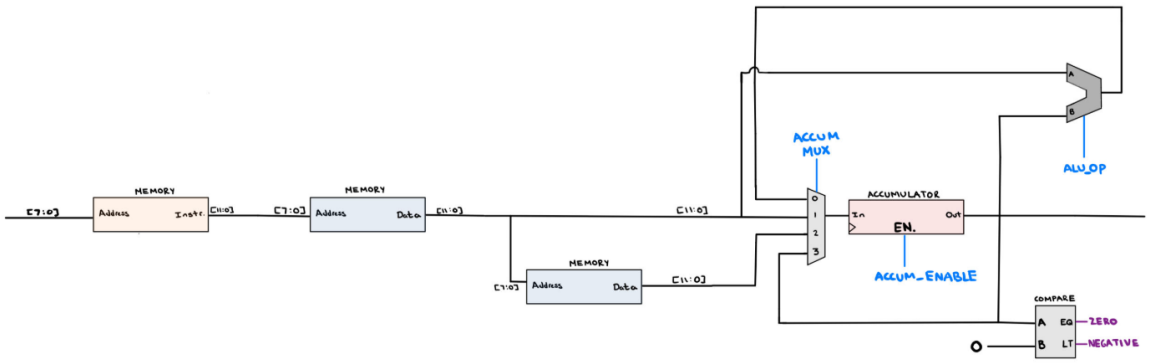
3 for ‘accum_mux’. This is because the final implementation will disable the register instead of feeding itself when the accumulator should not update.



	LOAD	LOADI	STORE	STOREI	ADD	SUB	AND	OR	JN	J	JZ
ACCUM MUX	1	2	X	X	0	0	0	0	X	X	X
ACCUM ENABLE	1	1	0	0	1	1	1	1	0	0	0
ALU OP	X	X	X	X	2	3	0	1	X	X	X

The image above is an incomplete datapath. Each of the accumulator mux inputs needs a source. The image below adds those sources. Notably, there are multiple memory blocks drawn separately. They have the same content, but changing the contents of one would also change the others.

The simplicity of this datapath is convenient. By applying an instruction address to the first memory block, and the instruction’s control signals to the blue lines, the accumulator will respond appropriately to all instructions. However, this design is inconvenient because the logic feeding the accumulator is combinational. This means that only one instruction can execute per clock. Therefore, this datapath in not a pipeline.



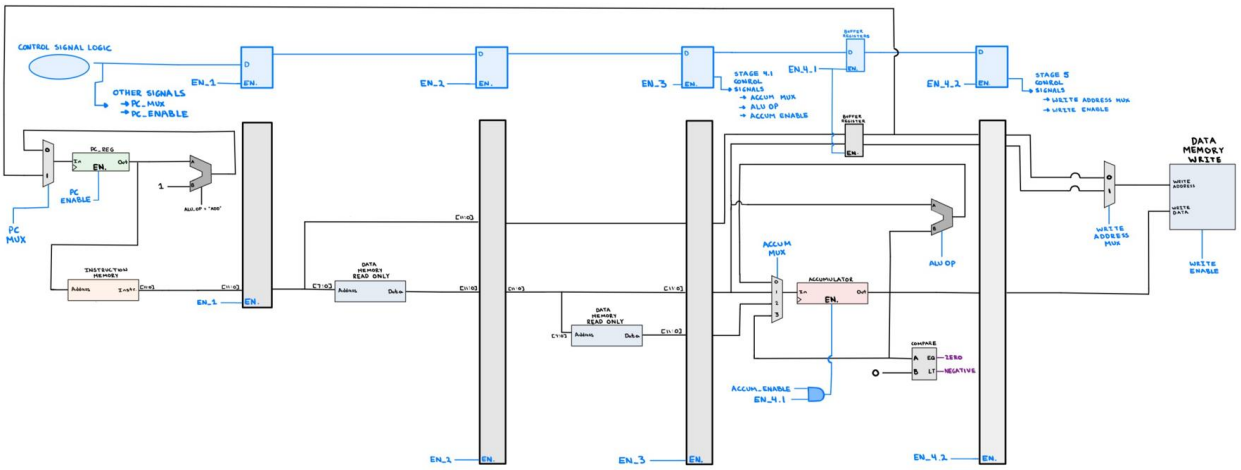
To create a pipeline, different hardware elements should be working on different instructions at the same time. To achieve this, the entire datapath becomes a shift register. At each clock, each stage latches the data and control signals of the previous stage. The image below demonstrates this idea. Notably, the narrow grey blocks latch data signals, such as memory reads. The blue registers are for latching control signals.

The image below also integrates features that were previously missing. The datapath needs a program counter, which holds the address of the current instruction. It feeds into an ALU to calculate the next instruction address. However, the program counter should also be loadable with an address from a branch. To accomplish this, a mux feeds

the input of the program counter register. On a clock edge, the program counter register either increases by one or becomes the address of a branch instruction. The control signal ‘pc_mux’ selects which address to load. Also, the ‘pc_enable’ control signal enables the pipeline to stop loading new instructions. This will be critical later in the design.

Since two of the three branch instructions depend on the accumulator value, the decision to jump for a conditional branch cannot be determined until the instruction before the branch has updated the accumulator. This explains the comparators in the second half of the fourth stage. Each produces a signal relevant to jump conditions. It also explains why the alternate program counter feeds from the fourth stage to the first stage’s program counter mux.

Finally, the image below integrates a final stage for writing to memory. Only STORE and STOREI instructions should cause stage 5 to modify memory. The new ‘write_enable’ control flag enforces this. Additionally, the STORE and STOREI instructions interpret their bottom 8-bits differently. A STORE instruction should use its bottom 8-bits as an address where it stores the accumulator. However, A STOREI instruction first looks up the data at the 8-bits, then use the bottom 8-bits of that data as an address to store the accumulator. Consequently, a mux needs to determine which address to use in stage 5. The new ‘write_address_mux’ signal controls this behavior.



	LOAD	LOADI	STORE	STOREI	ADD	SUB	AND	OR	JN	J	JZ
PC ENABLE	?	?	?	?	?	?	?	?	?	?	?
PC MUX	?	?	?	?	?	?	?	?	?	?	?
ACCUM MUX	1	2	X	X	0	0	0	0	X	X	X
ACCUM ENABLE	1	1	0	0	1	1	1	1	0	0	0
ALU OP	X	X	X	X	2	3	0	1	X	X	X
WRITE ADDRESS MUX	X	X	0	1	X	X	X	X	X	X	X
WRITE ENABLE	0	0	1	1	0	0	0	0	0	0	0
BRANCH ENABLE	0	0	0	0	0	0	0	0	1	1	1

As a new instruction begins in the first stage, combinational logic determines and sets the values of the control signals for the instruction, according to the table above. With each new clock, those signals propagate down the pipe. This appears that it would work well. However, this design has the following issues.

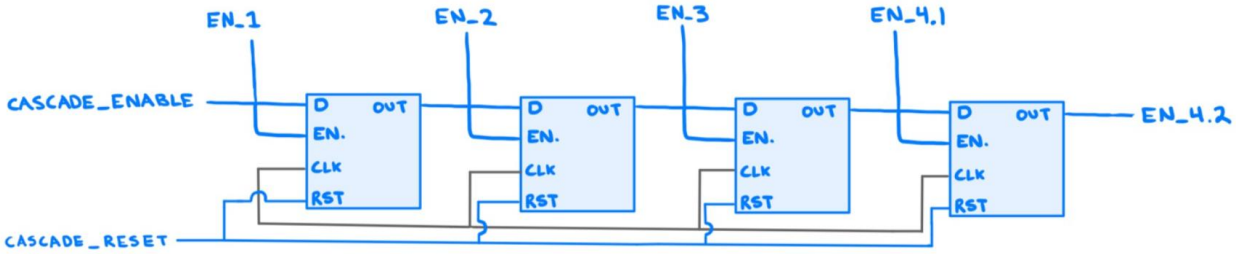
- *Read After Write Hazard:* The first stage looks up an instruction, sets the control signals, then propagates both down the line. However, the memory read by stage one might be incorrect if the pipeline contains a STORE or STOREI instruction that has not finished the fifth stage. The second and third stage have the same issue. In other words, if a STORE or STOREI instruction is in the pipeline and has not finished, then memory is incorrect.

- *Missing Branch Logic:* Above, the table of control signals leaves ‘pc_mux’ undefined. This is because the decision to branch cannot be determined in stage 1 like the other control signals. The decision to branch can only happen after the instructions before the branch have finished.
- *Missing Control Logic:* Above, the datapath’s stage registers have undriven enable control signals. The control signal table leaves these signals undefined. Neither is the control signal ‘pc_enable’. This is because the logic of these signals is more complicated.
- *Missing Halt Logic:* The HALT logic is undefined.

Despite these issues, the current design has is redeemable. The path will work correctly by simply adding extra logic.

B. Cascading Enables

Before addressing the issues at the end of the previous sub-section, this sub-section will explain how a ‘cascading-enable-shift register’ works and relate it to functionality that the current design is missing.



The image above features four flip flops that chain together like a shift register. Unlike a normal shift register, the output of each flip flop is the enable signal of the next. Also, the input data of the first register also enables it. This is interesting for the following reasons:

- If the registers are not enabled, then driving ‘cascade_enable’ high only causes the first register to latch on the next clock. The first register’s latched output would enable the second register for the second clock. When the third clock happens, the fourth register will become enabled. Each clock will enable an extra register until they are all enabled (while ‘cascade_enable’ is high).
- The opposite behavior is also true. If all the registers are enabled, then clocking while ‘cascade_enable’ is low will only cause the first register to become disabled. Each consecutive clock will disable an extra register until all registers are not enabled.
- A mix of the two previous behaviors is also true.

The next three subsections will describe how to implement store, branch, and halt logic with this cascade-enable tool.

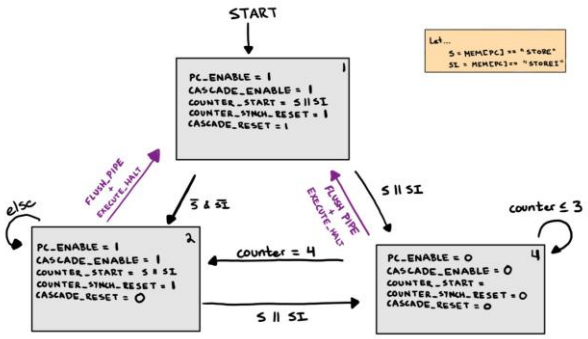
C. Handling Store and StoreI Data Hazards

While it may not be obvious, the cascading-enable behavior makes it easy to solve the STORE/STOREI memory issue. When a STORE/STOREI instruction starts in the first stage, the pipeline needs to stop sending new instructions down the pipe. It must let the STORE/STOREI instruction finish before latching another into stage 2.

To accomplish this, simply drive ‘cascade_enable’ and ‘pc_enable’ low after a STORE/STOREI latches into stage two. This will allow the first stage to load the next instruction but not propagate it down the pipe, and allow the pipeline to finish what it already holds (including the STORE/STOREI). Each clock will cause the STORE/STOREI to propagate down the pipe a single stage but leave duplicates of itself behind.

When the store instruction latches into stage 5, the next clock should latch the instruction at the program counter into stage 2. Then, the first two stages should latch. Then, the first three stages should latch. This pattern should continue until all stages are latching new instructions. Accomplish this by leaving ‘cascade_enable’ and ‘pc_enable’ high for each clock. It is important to note that ‘cascade_enable’ and ‘pc_enable’ should drive low after a STORE/STOREI latches into stage 2, regardless of which stages are active. Below, a finite state machine formalizes this logic.

The logic added in this section is illustrated in the next datapath schematic.



D. Handling Branches

As mentioned earlier, branching must wait for previous instructions to finish updating the accumulator. That is why the alternate pc address feeds into the program counter mux from the fourth stage. However, delaying the branch decision also loads the pipe with the instructions immediately after the branch instruction. This is convenient when the pipeline does not take the branch. However, it is incorrect when the pipeline does take the branch. If the pipeline takes the branch, then the instructions that loaded after the branch are incorrect`.

To solve this issue, the pipeline needs to generate a ‘flush_pipe’ signal. If this signal goes high, the program counter will load the alternate address on the next latch instead of incrementing the program counter. The cascade-enable logic also needs to reset, only propagating the instruction at the jumped address on the next clock. Above, the finite state machine illustrates this condition.

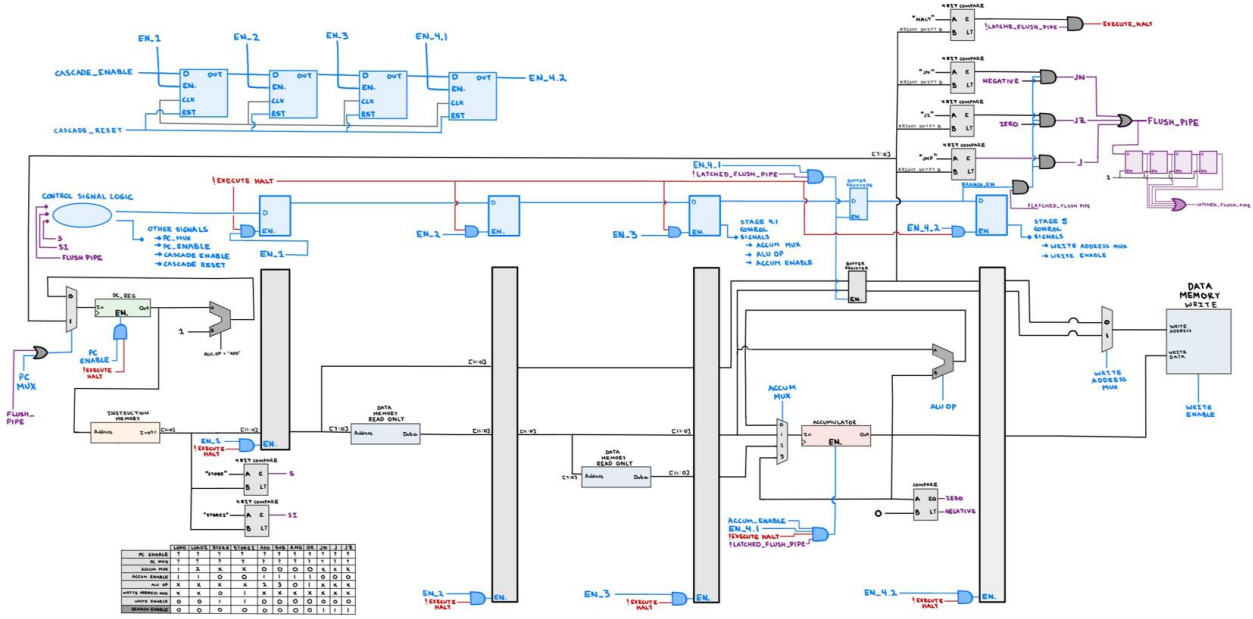
Another branching issue stems from the default register values, and the opcode binary encoding. Assuming the registers hold zeros by default, then stages 2 through 5 will hold a JUMP 0 instruction when the pipeline starts. This is because the JUMP opcode is 0000. A ‘branch_enable’ control signal prevents branching from default register values. This signal must be set high in the first stage, then propagated down the pipe with its instruction.

When the pipe is flushed, new instructions propagate down the line until they replace the flushed instructions. However, the old instructions remain. Even though the finite state machine above keeps them from latching into a new state, they still sit in the later stages with their control signals. Consequently, the signal ‘flush_pipe’ signal is latched into a series of chained flip flops. After a flush pipe, this will take 4 clocks to clear from the flip flops. While it is latched by at least one, the pipe has not finished flushing. This signal is used to interrupt other logic that would incorrectly execute during a flush, such as loading the accumulator. If a flush causes an alu operation to pause on in stage 4, this ‘latched_flush_pipe’ signal will prevent the stuck instruction from modifying the accumulator on each clock that it takes to clear the pipe.

The changes mentioned in this section are illustrated in the next datapath schematic below.

A. Handling Halts

Like a branch instruction, the decision to halt the program must wait until prior instructions have completed. Therefore, there is a comparator in stage 4 that identifies HALT instructions. However, it is possible that a HALT arrives on the clock after a branch was taken. In this case, the HALT should not be executed. To prevent this, a HALT should only be executed if the ‘latched_flush_pipe’ signal is low when it arrives in stage 4. If the HALT should execute, the signal ‘execute_halt’ disables all the registers, as seen below in the completed design.



III. SIMULATOR DESIGN

A Java simulator was coded to ensure correct functionality. The simulator was different from the single stage simulator written for the previous assignment in the following ways.

- This version simulated how instructions would proceed through a pipeline, while the previous simulator only imitated how the instructions would change the program counter, accumulator, and memory.
- This version imitated sequential and combinational logic, while the previous did not. To achieve this, every flip flop demands an input and output variable for the single signal passing though it. The output signal of every input/output pair adopted the input value when a 'clock()' method was called. This functionality also included the ability to enable or disable flip flops by setting the value of a third signal for every input/output pair. Combinational logic was imitated by updating the input values of flip flops based on the outputs, logic gates, and hardware that fed them.
- The order of resolving latches is delicate. To avoid issues, a Hardware object stores the state of the processor. Instead of trying to update the output of the latches in a way that does not create conflicts, the sim driver works with two Hardware objects. One holds the previous processor state. For each clock, the simulator makes a copy of the previous Hardware object. Then, the sim driver calls the clock() method on the copy. Then, values latched by the registers in the copy are determined by referencing the unmodified original Hardware object. Finally, the sim driver sets the current object as the previous object and repeats the process.

In summary, this simulator attempted to be as real as possible. Initially, I believed the project required this level of detail. After learning that it was a level of abstraction too deep, I chose to keep it because it was easier to debug. On multiple occasions I found myself watching how the control and data signals propagated through the pipeline, revealing conceptual mistakes.

I admit that the simulator is very bulky. For example, a single control signal needs six separate variables to pass through all stages. To alleviate this, I created a naming scheme and documented well. Additionally, I tried to abstract away the use of numbers with enums.

IV. DESIGN ANALYSIS

The previous sections demonstrated how a pipeline datapath could be implemented for a simple instruction set, and explained the simulator software design. This section will investigate how the following benchmarks run on the simulated pipeline:

1. Speed Optimized Multiplication of N Numbers
2. Memory Optimized Multiplication of N Numbers
3. Space Optimized Factorial of a Number N
4. Speed Optimized Factorial of a Number N

The analysis will primarily look at the instruction mix, number of clocks, and number of stalls of each benchmark. This section will conclude by determining which benchmark is best suited for this pipeline architecture.

A. Characterization of Multiplication Benchmarks

The primary difference between the '*speed*' and '*memory*' multiplication benchmarks is the number of instructions executed. Below, the two tables show the number of instructions executed as a function of N.

		Operation Type					
	N	LOAD/LOADI	STORE/STOREI	ALU	BRANCH	MISC	TOTAL INSTRUCTIONS
SPEED OPTIMIZED MULTIPLICATION	0	0	0	1	1	1	3
	1	2	2	1	2	1	8
	2	45	36	37	30	1	149
	3	87	69	73	57	1	287
	4	131	104	111	84	1	431
	5	175	139	149	111	1	575
	15	615	489	529	381	1	2015
	40	1715	1364	1479	1056	1	5615

		Operation Type					
	N	LOAD/LOADI	STORE/STOREI	ALU	BRANCH	MISC	TOTAL INSTRUCTIONS
MEMORY OPTIMIZED MULTIPLICATION	0	0	1	0	1	1	3
	1	2	2	1	3	1	9
	2	14	12	10	14	1	51
	3	23	20	17	22	1	83
	4	30	26	22	28	1	107
	5	35	30	25	32	1	123
	15	85	70	55	72	1	283
	40	210	170	130	172	1	683

This is surprising, because it shows that the '*memory'* optimized version was almost an order of magnitude faster than the '*speed'* version. These numbers were pulled from the pipeline simulator. However, they were cross referenced with the previous single stage simulator, confirming that this simulator worked the same.

This report elected to look at the instruction mix as a function of N, which is presented in the table below.

		Operation Type				
	N	LOAD/LOADI %	STORE/STOREI %	ALU %	BRANCH %	MISC %
SPEED OPTIMIZED MULTIPLICATION	0	0.00	0.00	33.33	33.33	33.33
	1	25.00	25.00	12.50	25.00	12.50
	2	30.20	24.16	24.83	20.13	0.67
	3	30.31	24.04	25.44	19.86	0.35
	4	30.39	24.13	25.75	19.49	0.23
	5	30.43	24.17	25.91	19.30	0.17
	15	30.52	24.27	26.25	18.91	0.05
	40	30.54	24.29	26.34	18.81	0.02

		Operation Type				
	N	LOAD/LOADI %	STORE/STOREI %	ALU %	BRANCH %	MISC %
MEMORY OPTIMIZED MULTIPLICATION	0	0.00	33.33	0.00	33.33	33.33
	1	22.22	22.22	11.11	33.33	11.11
	2	27.45	23.53	19.61	27.45	1.96
	3	27.71	24.10	20.48	26.51	1.20
	4	28.04	24.30	20.56	26.17	0.93
	5	28.46	24.39	20.33	26.02	0.81
	15	30.04	24.73	19.43	25.44	0.35
	40	30.75	24.89	19.03	25.18	0.15

While the authors of this report did expect a significant instruction mix change as N increased, the table above indicates that the change is small. The chart shows that STORE/STOREI percentage decreases with N. It then stabilizes quickly for both benchmarks around 24.5%. Similarly, the branch percentage decreases for both benchmarks. The branch mix stabilizes around 18% for the '*speed'* benchmark, and 25% for the '*memory'* benchmark.

Considering the flaws of the pipelined architecture designed in this report, the largest flaw is stalling. Stalls are exclusively caused by store and branch instructions in this design. While the '*speed'* benchmark branches less often, its number of instructions are almost an order of magnitude larger. Therefore, the advantage is negated. If every instruction of the '*memory'* benchmark stalled, it would still be faster than the '*speed'* benchmark on the pipelined design.

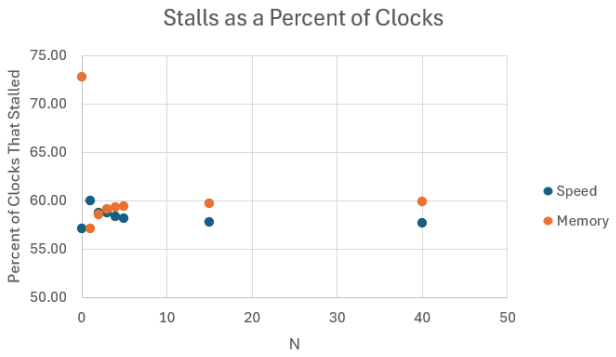
B. Multiplication Benchmark Pipeline Performance

Below, the table reveals how the '*speed'* and '*memory'* multiplication benchmarks performed on the pipelined architecture. The table distinguishes actual clock from total clocks, where total clocks include the initial clocks required to fill the pipeline at startup.

	N	TOTAL INSTRUCTIONS	TOTAL CLOCKS	ACTUAL CLOCKS	TOTAL STALLS	% OF CLOCKS THAT STALL
SPEED OPTIMIZED MULTIPLICATION	0	3.00	11	7	4	57.14
	1	8.00	24	20	12	60.00
	2	149.00	365	361	212	58.73
	3	287.00	699	695	408	58.71
	4	431.00	1039	1035	604	58.36
	5	575.00	1379	1375	800	58.18
	15	2015.00	4779	4775	2760	57.80
	40	5615.00	13279	13275	7660	57.70

	N	TOTAL INSTRUCTIONS	TOTAL CLOCKS	ACTUAL CLOCKS	TOTAL STALLS	% OF CLOCKS THAT STALL
MEMORY OPTIMIZED MULTIPLICATION	0	3.00	15	11	8	72.73
	1	9.00	25	21	12	57.14
	2	51.00	127	123	72	58.54
	3	83.00	207	203	120	59.11
	4	107.00	267	263	156	59.32
	5	123.00	307	303	180	59.41
	15	283.00	707	703	420	59.74
	40	683.00	1707	1703	1020	59.89

The most striking part of this table is the percentage of clocks that stalled. It is surprisingly high. However, this should not be a surprise. The previous section identified that half of the instruction mix is branching and stores. Both instruction types cause many stalls, so a high clock stall rate should not be surprising. This percentage can also be visualized with the chart below. The percentage of clock stalls evens out as N gets large. Again, this is not surprising because the percentage of store and branch instruction levels out with these two benchmarks, and the number of stalls is a linear function of the number of branches and stores.



C. Characterization of Factorial Benchmarks

The pipeline simulation provides the data in the image below, for the '*speed'* and '*memory'* optimized factorial benchmarks. Notable, the only significant difference is the number of instructions executed. The '*memory'* benchmark uses approximately 20% more instructions than the '*speed'* benchmark.

		Operation Type					
	N	LOAD/LOADI	STORE/STOREI	ALU	BRANCH	MISC	TOTAL INSTRUCTIONS
SPACE OPTIMIZED FACTORIAL	0	8	9	2	1	1	21
	1	8	9	2	1	1	21
	2	294	291	213	143	1	942
	3	594	584	433	295	1	1907
	4	908	888	662	457	1	2916
	5	1236	1203	900	629	1	3969
	6	1578	1529	1147	811	1	5066

		Operation Type					
	N	LOAD/LOADI	STORE/STOREI	ALU	BRANCH	MISC	TOTAL INSTRUCTIONS
SPEED OPTIMIZED FACTORIAL	0	8	9	2	1	1	21
	1	8	9	2	1	1	21
	2	218	215	156	104	1	694
	3	442	432	319	216	1	1410
	4	680	660	491	337	1	2169
	5	932	899	672	467	1	2971
	6	1198	1149	862	606	1	3816

Looking at the instruction mixes below, the percentage of each instruction category is very close to constant. Since factorial growth is very fast, the factorial numbers larger than six cannot be computed on a processor with a 12-bit accumulator. Additionally, the instruction mixes are identical for the two benchmarks.

Compared to the multiplication benchmarks, these benchmarks branch approximately 10% less, but store about 5% more. Since branches and stores cause stalls, this benchmark's mix suggests that it might have a lower percentage of clocks that stall.

		Operation Type				
	N	LOAD/LOADI %	STORE/STOREI %	ALU %	BRANCH %	MISC %
SPACE OPTIMIZED FACTORIAL	0	38.10	42.86	9.52	4.76	4.76
	1	38.10	42.86	9.52	4.76	4.76
	2	31.21	30.89	22.61	15.18	0.11
	3	31.15	30.62	22.71	15.47	0.05
	4	31.14	30.45	22.70	15.67	0.03
	5	31.14	30.31	22.68	15.85	0.03
	6	31.15	30.18	22.64	16.01	0.02

		Operation Type				
	N	LOAD/LOADI %	STORE/STOREI %	ALU %	BRANCH %	MISC %
SPEED OPTIMIZED FACTORIAL	0	38.10	42.86	9.52	4.76	4.76
	1	38.10	42.86	9.52	4.76	4.76
	2	31.41	30.98	22.48	14.99	0.14
	3	31.35	30.64	22.62	15.32	0.07
	4	31.35	30.43	22.64	15.54	0.05
	5	31.37	30.26	22.62	15.72	0.03
	6	31.39	30.11	22.59	15.88	0.03

D. Factorial Benchmark Pipeline Performance

The performance metrics of the factorial metrics are in the image below. Notably the percentage of clocks that stall is nearly identical to two multiplication benchmarks, despite having a smaller sum of store and branch percents.

	N	TOTAL INSTRUCTIONS	TOTAL CLOCKS	ACTUAL CLOCKS	TOTAL STALLS	% OF CLOCKS THAT STALL
SPACE OPTIMIZED FACTORIAL	0	21	65	61	40	65.57
	1	21	65	61	40	65.57
	2	942	2406	2402	1460	60.78
	3	1907	4859	4855	2948	60.72
	4	2916	7424	7420	4504	60.70
	5	3969	10101	10097	6128	60.69
	6	5066	12890	12886	7820	60.69

	N	TOTAL INSTRUCTIONS	TOTAL CLOCKS	ACTUAL CLOCKS	TOTAL STALLS	% OF CLOCKS THAT STALL
SPEED OPTIMIZED FACTORIAL	0	21	65	61	40	65.57
	1	21	65	61	40	65.57
	2	694	1774	1770	1076	60.79
	3	1410	3590	3586	2176	60.68
	4	2169	5513	5509	3340	60.63
	5	2971	7543	7539	4568	60.59
	6	3816	9680	9676	5860	60.56

E. Final Benchmark Comparison

For a benchmark to do well in the pipeline design, it must minimize stores and branches because they stall the pipeline. The '*speed'* multiplication benchmark's combined store and branch percentage is approximately 45% when N is not small. The '*space'* multiplication benchmark is approximately 50% stores and branches, while the two factorial benchmarks are approximately 45%. Initially, this led us to believe that the factorial benchmarks might have a lower percentage of clocks that store. However, all the benchmarks stall approximately 60% of the time. This could have a couple of explanations. The most likely seems to be, that the number of branch instructions in a benchmark's mix does not reflect the number of stalls because stalls occur when the branch is taken.

The question arises, which benchmark was better suited for this pipeline design? While this may be subjective, the authors of this paper believe that percentage of clocks that stalled is the best metric. None of the benchmarks are better suited than the other, because their percentage of time spent stalling is approximately equal.

V. DISCUSSION

After evaluating the performance of the previous benchmarks, a big question remains. Did the pipeline design improve performance? This is a difficult question to answer, because the metric used to compare the approaches is vague. By clock cycle alone, the pipeline benchmarks are significantly worse. This is because the single cycle machine assumed a cpi of 1. However, the pipeline design treats a clock as the time required to progress an instruction through one stage. The pipelined design could only achieve a cpi of 1 if there was no stalling.

It seems that the time to execute the combinational logic of any instruction in the pipeline would be the same time to execute the combinational logic in the single stage design. Therefore, five of the pipeline's 'clocks' seem to be a single clock of the single stage datapath. There are many legitimate arguments against this. For example, a single stage of the pipeline can only be as fast as the slowest stage of the pipeline. Therefore, you cannot say that is $1/n^{th}$ the time of the single stage design, where n is the number of stages. However, the authors of this report can find no other way to standardize the time of a single 'clock' to facilitate a fair comparison. Without the ability to

standardize the length of a ‘clock’, we are regrettably left with intuition. However, the pipelined version is undoubtedly faster than the single stage design.

VI. CONCLUSION AND FUTURE WORK

Pipelining is an excellent way to speed up program execution time. However, the performance comes at a complexity that can be difficult to manage. While this design would does speed up execution time, it has considerable issues. Primarily, the design halts whenever a store instruction is read. This is done to prevent accidentally using bad data in the instructions that immediately come after a store. However, there is a chance that the data used by the instructions immediately following the store access data at addresses that the store is not modifying. In other words, there is a chance that not stalling would cause no issues. Considering this, it would be enjoyable to modify this design with smarter store logic, only halting where necessary. Speculatively, this could radically reduce the 60% stall metric.