Lab 2 Report
Elise Song (eys29)
Katherine Zhou (kz273)

Introduction

In this lab, we implemented a five stage pipelined processor that executes all TinyRV2 instructions. In the alternative design, bypassing was added to decrease latency caused by stalling for data and control hazards. This included understanding the given ISA to add the necessary hardware in the datapath and corresponding control signals from the control unit for each instruction. We learned about pipelined processor microprocessor microarchitecture in lecture, such as the five stages (F, D, X, M, and W), how instructions step through the pipeline stages, how hazards require stalling/squashing, and how bypassing can decrease stalling due to hazards but requires new stalling/squashing logic. The lecture content was quite abstract, pipeline diagrams being an example, so we were able to put the abstraction into practice in this lab. This lab also required a high level of organization and consistency since the pipeline logic quintuples the number of signals of a single cycle processor for the five stages. Thus, the encapsulation design principle was especially important for the Control Unit and Datapath only shared necessary communication. We intentionally had the incremental approach of implementation and testing to ensure that each pipeline stage was separate and the control logic was correct.

Alternative Design

The processor implemented in this lab is a five stage pipelined processor. The stages are fetch, decode, execute, memory, and writeback, shortened to F, D, X, M, and W. In the F stage, the instruction is fetched from memory. In the D stage, the instructions are decoded. This is where we know what instruction it actually is, ie. is it an add, a branch, a jump, etc. In this stage, we also find out which source and destination registers are used in the instruction, as well as the immediate if there is one. In the X stage, the instruction is executed using the registers and/or immediates that are specified and decoded in the D stage. In the M stage, we handle memory read and write requests. This is when memory data and address requests are returned. Finally, in the W stage, we write the result of the instruction into the register file if needed. In the baseline design, all data hazards are handled using stalling. The datapath diagram of our baseline processor datapath is shown below in Figure 1.

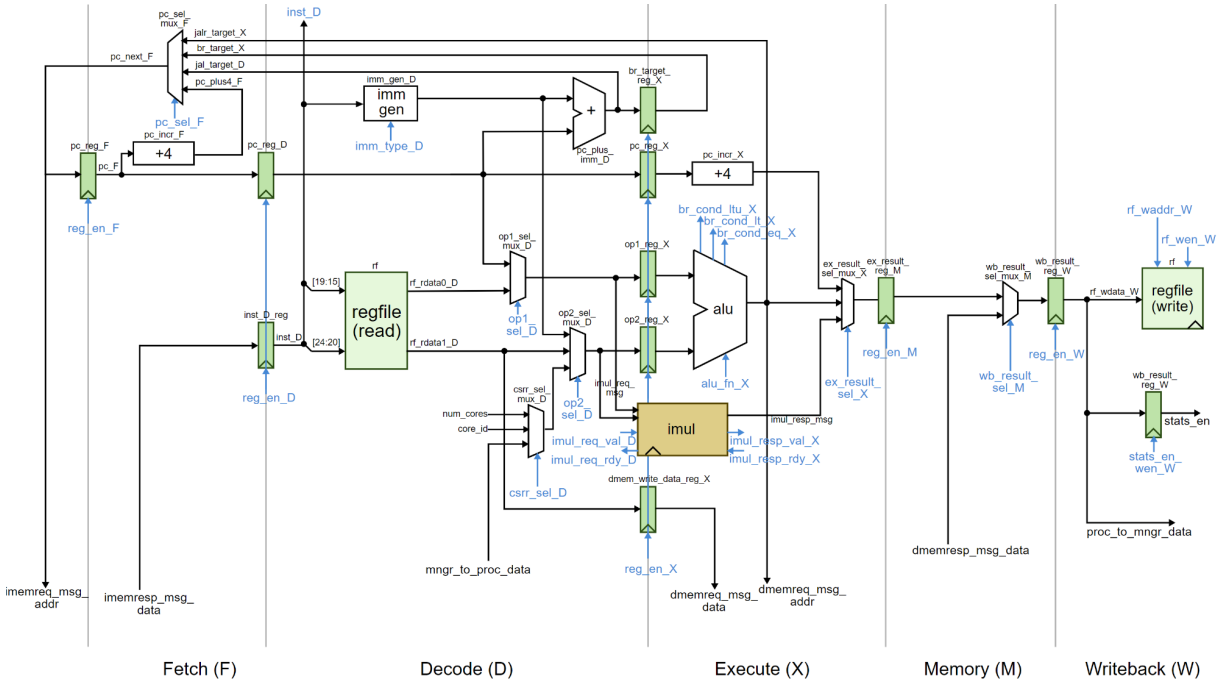

Figure 1: Baseline design datapath diagram

One exception to this standard datapath is the multiply instruction. The multiply instruction is implemented using the alternative design of our iterative multiplier from Lab 1. In the D stage if the instruction is a multiply, then

a request signal, `imul_istream_val`, is sent from the datapath to the multiplier. A corresponding signal, `imul_istream_rdy`, is sent from the multiplier back to the D stage of the processor to indicate that the multiplier is ready to accept a multiply instruction. In the X stage, the processor will receive the `imul_ostream_val` signal from the multiplier, which indicates that the multiplier is finished with its current instruction. In turn, the processor sends the `imul_ostream_rdy` signal to the multiplier, which is a signal that indicates that the processor is ready to accept the output of the multiplier. These signals are important for determining when the processor needs to stall, since it must stall in the D stage if the multiplier is not ready to receive a new instruction, and in the X stage if the multiplier is not finished executing the current instruction which it has.

The alternative design is very similar to the baseline design, the only major difference being that in the datapath we add multiple bypass paths. These bypass paths allow us to reduce the need for stalling, since many data dependencies can be resolved by taking the result from a later stage and "forwarding" it to the D stage of our processor to use the updated data. This allows us to reduce the number of cycles needed per instruction, which improves our overall processor performance. Our alternative design is fully bypassed, meaning that there are bypass paths from the X, M, and W stages back to the D stage.

In order to implement the bypass paths, we needed to add two bypass muxes in the D stage. These are four input muxes, where input 0 is the register file data, input 1 is the output of the X stage, input 2 is the output of the M stage, and input 3 is the output of the W stage. There is a mux for both the outputs of the register file read. The datapath diagram of our fully bypassed datapath is shown below in Figure 2.

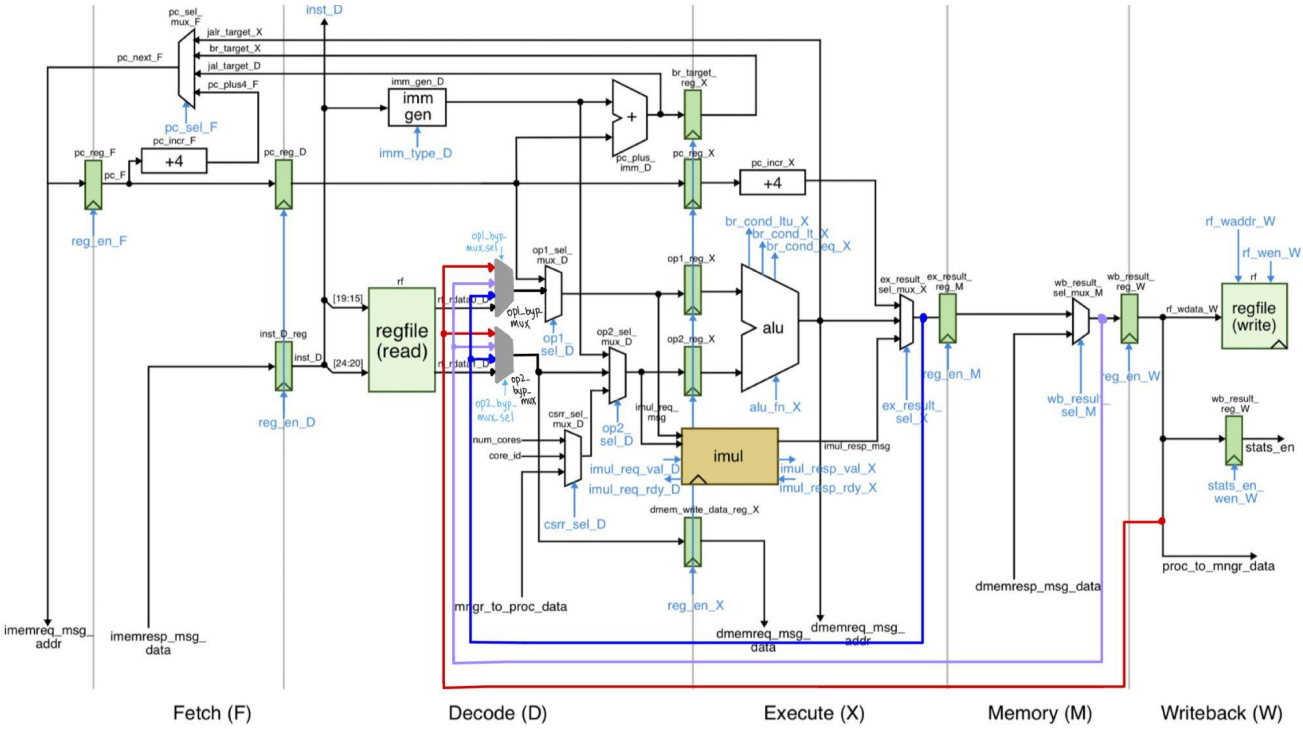

Figure 2: Alternative design datapath diagram (fully bypassed processor)

Additionally, we needed to add bypass signals to the control unit of the processor. To derive these signals, we looked at when a value would need to be bypassed, which is when the register to write in an instruction in a later stage matched a register to read in the D stage. An example pipeline diagram of a simple series of instructions that would use bypassing is shown below in Figure 3. The red and blue letters indicate bypassing from one stage to the other. As we can see, the result of the addi instruction from the X stage is bypassed and sent to the D stage of the add instruction. Additionally, the result of the addi in the M stage and the result of the add in the X stage is bypassed and sent to the D stage of the sub instruction. This demonstrates the need for bypass paths from multiple stages to the D stage, not just from X to D, which further emphasizes the need for a fully bypassed processor.

`addi x1, x1, 5`	F	D	X	M	W									
`add x2, x1, x3`		F	D	X	M	W								
`sub x3, x2, x1`			F	D	X	M	W							

Figure 3: Pipeline diagram to demonstrate bypassing

The bypass signals for each stage are derived in the D stage, and are all very similar to each other. The bypass signals for input one and input two are symmetrical for each stage. We check that the instruction in the stage we are bypassing from is not a load because if it is, we cannot bypass and must stall the processor. Thus, we bypass when a source register in the D stage is the same as the destination register in a later stage of the pipeline, and the instruction in the later stage is not a load. The bypass signals for each stage are used to set the bypass mux select signal so that our mux is choosing the correct input value to use for the current instruction in D.

Although bypassing allows us to reduce the need for stalling, it does not eliminate the need for stalling completely. The processor still needs to stall when there are load use dependencies, so we needed to change the stall signals from the baseline design. To derive the stall signals, we looked at how the load instruction works. Again, the stall signals are similar from stage to stage, and all the stall signals are derived in the D stage, since this is when we want to stall the processor. If the instruction in a later stage (X, M, or W) of the pipeline is a load, and the destination register of the load is equal to a source register of the instruction in D, then the processor will stall.

Our design implements the design patterns of modularity through the separation of the datapath and the control unit modules. In the datapath module, everything is broken down further into smaller modules for each hardware component. In the top level module, we declare the datapath and control modules. The datapath module declares our multiplier module, along with many other modules in the datapath. This demonstrates a hierarchical design. Additionally, our processor demonstrates the encapsulation design principle, since the memory and multiplier interfaces only need to interact with the datapath through valid and ready signals, and do not need details of the implementation.

Testing Strategy

Our overall testing strategy made use of black box testing, white box testing, directed testing, random testing, value testing, dependency testing, and delay testing for every instruction. We also used unit tests for the immediate generation unit and the ALU to ensure that there were no bugs there that could carry down into our processor implementation. Testing was done using pytest in Python. We made sure to run our tests on the functional level model first to guarantee the correctness of the test cases.

We made use of both black box and white box testing in order to ensure the overall functionality of our processor. Majority of our test cases do not take into account the specifics of the implementation of the processor itself, which is black box testing. This helps us to make sure that our tests are not passing solely because of some detail of the processor implementation, and that the overall correctness is maintained for all instructions. However, we also use white box testing to zoom in on specific instruction sequences to make sure that they are executing properly. In particular, we wanted to test that our load instruction worked correctly in conjunction with data dependent instructions, specifically the multiply instruction, to make sure stalling was working. We also tested combining jump and branch instructions to ensure that the correct PC was jumped/branched to, and that the other instruction would be squashed correctly. We found that white box testing was equally as important as black box testing, since these specifics were necessary to establish the functionality of our processor.

We also used directed tests to test specific edge cases of each instruction. For example, for shift instructions we test shifting by 0 bits and by 31 bits, which is the maximum number of bits we can shift by. For SLT (set less than) instructions, we tested that the output was correct when the two inputs were equal to each other.

Although directed tests are great at ensuring correct behavior of edge cases, it does not generate enough tests to achieve sufficient coverage. Therefore we made use of random tests to get a large number of tests with random inputs to ensure that our processor can handle these inputs, to potentially catch cases that we did not think of, and to add coverage to our tests. For these tests, it was important to fix the random seed so that the tests were replicable for debugging purposes.

All of the tests mentioned above are also value tests, since we are comparing the value that is generated by the test output to the actual output of our processor to make sure that they match up.

In addition to the testing described above, we also used dependency tests. These tests insert a specified number of nop instructions before a source register read or a destination register write. This was an important part of

testing our alternative design because the combination of a decreased number of nops and the change of the stall signals to only stall for load use meant that if our bypass paths were incorrect, these tests would not pass. We also tested cases where a source register and destination register are the same.

The final category of tests done were delay tests. For these tests, we used a python helper function to turn delays on. The delays are randomly generated, and we run the delay tests on our random input test case to get the maximum amount of coverage. Having the delay tests pass with our random tests gave us confidence that the functionality of our processor was correct.

A table detailing some specifics of our test cases is shown below in Table 1. Not every test case is shown to reduce redundancy. Only tests that were particularly interesting for a distinct instruction are listed specifically, otherwise only broad categories of instructions are listed.

Table 1: Summary of test cases

Instruction type	**Test type**	**Test description and justification**
Register-register, register-immediate, jump, branch	Destination dependency test	Inserts a specified number of nop's before the output is written to the destination register. These tests are specifically for testing the destination bypass path in the alternative design by varying the number of nops between the instruction and the read out of the destination register to ensure that bypassing is functional.
Register-register, register-immediate, jump, branch	Source dependency test	Inserts a specified number of nop's before a value is put into one or both of the source registers. These tests are specifically for testing the source bypass paths in the alternative design by varying the number of nops between writing the source register and reading it in the next instruction to ensure that bypassing is functional.
Register-register, register-immediate	Source and destination equal test	These tests are for the case where a source register is the same as the destination register, the two source registers are the same, or all of the registers are the same to guarantee that our processor is able to handle these situations. We want to be able to read a source register and write the result back into it for correct operation.
Register-register, register-immediate, memory	Random test	These tests are meant to add coverage by testing random inputs and checking that the outputs are what we expect them to be.
Register-register, register-immediate, jump, branch, memory	Delay test	These tests are for varying the sink and source delay of our processor.
Register-register: SUB	Subtracting a negative number	This ensures that subtracting a negative number behaves the same as adding a positive number.
Register-register: MUL	Multiply and load word data dependency test	This tests the correctness of our stall signals by creating a data dependency between the destination of a multiply instruction and the source of a load instruction. We want to ensure that the load reads the correct value out of its destination register.
Register-register: SHIFTS	Shifting by minimum and maximum value	This tests edge cases for shift instructions. Shifting by 0 should push the input value to the output, and shifting by 31

		bits should push zero to the output regardless of what the input is for the logical shifts. For an arithmetic shift, the sign is preserved so the output will be either 0 or -1 depending on the sign of the input.
Register-immediate: ADDI	Add-immediate and load word data dependency test	This tests to see if the load use stalling is correct with an ordinary instruction (any instruction that is not a multiply). We want to see that the load will read the correct memory address when it is after the addi. We want to see that the addi will use the correct input when the load is before the addi.
Memory: SW	Data/Address Source/Destination Dependency Test	Since our implementation of the Store instruction required additional hardware, we wanted to differentiate bugs due to the wrong data being written or the wrong address being written to, especially when bypassing. This also tests the source dependency of the load instruction.
Memory	Sequences of consecutive memory instructions	We wanted to ensure that consecutive memory instructions would work.
Jump, Branch	Jumping/branching to an instruction before the current instruction	We identified this as an edge case and wrote a directed test.
Jump, Branch	Sequences of consecutive jump and branch instructions	We identified this as an edge case in lecture where the jump/branch instruction first should take priority over the next jump/branch instruction.
Jump: JAL	BEQ and JAL as back to back instructions to make sure the correct instruction is executed.	We want to make sure that the processor is able to execute these instructions one after the other with varying delays (from the delay tests) and execute the correct instruction after the jump or branch.

After passing all of these tests, we felt that we achieved enough coverage to be confident enough to conclude that our processor was operating correctly.

Evaluation

Simulation Quantitative Analysis:

We ran the Base and Alt pipelined processors against five C program benchmarks: vvadd-unopt, vvadd-opt, cmult, bsearch, and mfilt. Figure 6 and Figure 7 show the number of cycles and CPI of the benchmarks respectively.

The vvadd benchmarks are vector-vector addition programs: adding two arrays together. The unoptimized version adds one element of each array at one time while the optimized version adds 4 elements of each array at a time. Vector-vector addition includes load, add, store, add-immediate, and branch instructions. We expected the Alt design unoptimized vvadd to have decreased average cycles/instr since the Base design would have to stall due to dependencies. Figure 4 illustrates the pipeline diagram for the unoptimized vvadd program. The bypassed processor had a steady state of 11 cycles per iteration, and the base design had a steady state of 20 cycles. So, the alternative design reduced the steady state almost by half. The program iterates 100 times so we expected a significant performance increase for the bypassed processor. The optimized version contains no dependencies so we did not expect any difference between the two designs.

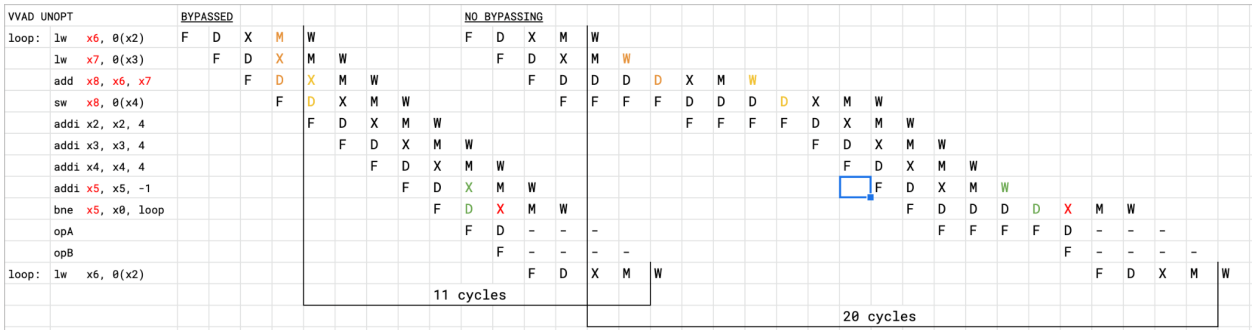

Figure 4: Pipeline diagram for unoptimized vector-vector add.
**Stages of the same color represent a dependency arrow*

The cmult benchmark is a complex multiplication C program that multiplies complex numbers made up of real and imaginary numbers. The benchmark contains multiply, load, store, add-immediate, and branch instructions. Since the program contains dependencies, we expected the Alt design to perform slightly better than the Base design. This is because our multiplier takes more than one cycle to complete an instruction, so it will always have to stall the processor regardless of data dependencies. Figure 5 illustrates the pipeline diagram of the cmult program. The bypassed processor has a steady state of 19 cycles per iteration in the program. The base design has a steady state of 22 cycles per iteration, which is not a significant difference. However, the cmult program iterates 100 times so the bypassed processor still performs slightly better than the base design.

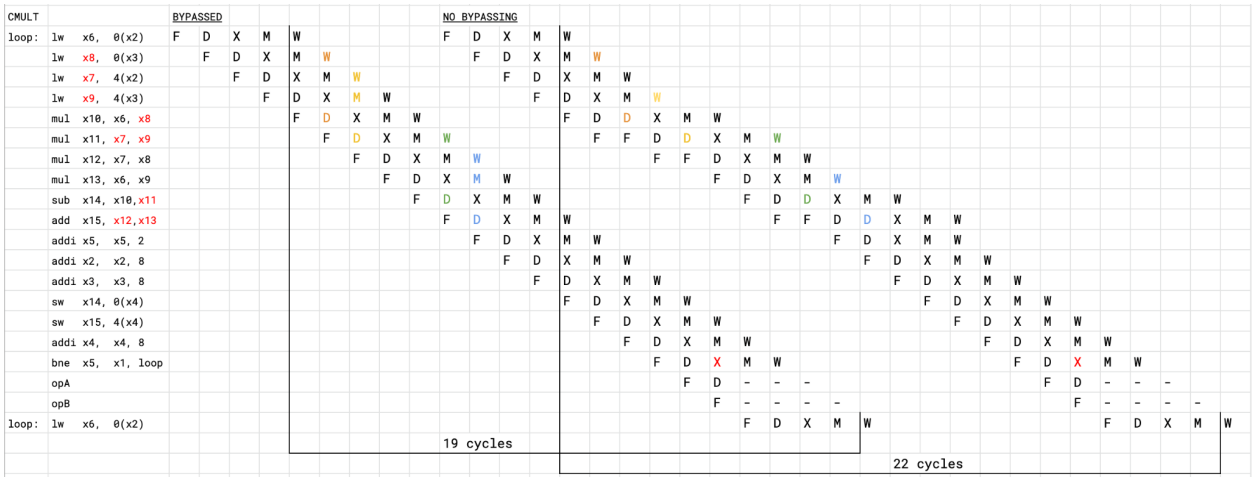

Figure 5: Pipeline diagram for complex multiplication of real and imaginary numbers
**Stages of the same color represent a dependency arrow*

The bsearch benchmark executes a binary search algorithm on a dictionary. The program contained a fair amount of dependencies so we expected better performance with the bypassed processor since the base design would have to stall to account for the dependencies. The program includes branch, register-immediate, register-register, jump, and memory instructions.

The mfilt benchmark applies a median filter on a 2 dimensional array. The program contains many dependencies so we expected a significant difference between the two processors. The program is made up of all types of instructions like the bsearch benchmark.

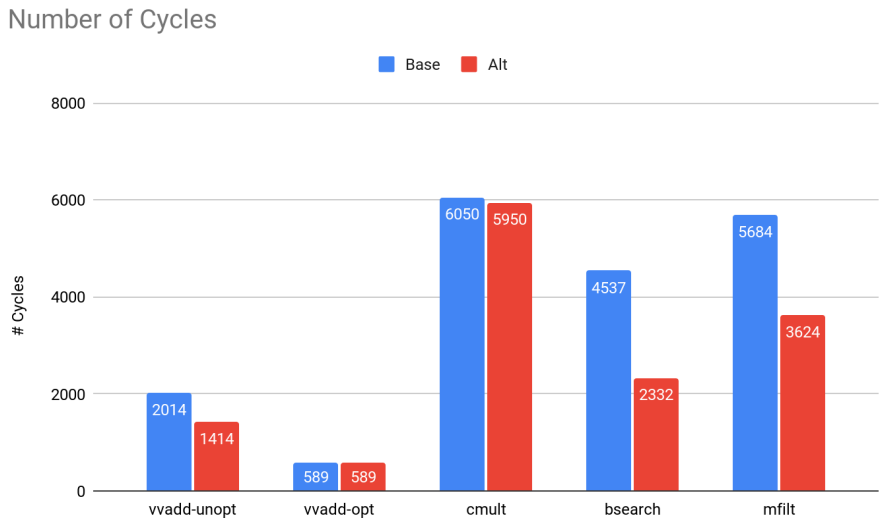

Figure 6: Total number of cycles for simulation benchmarks

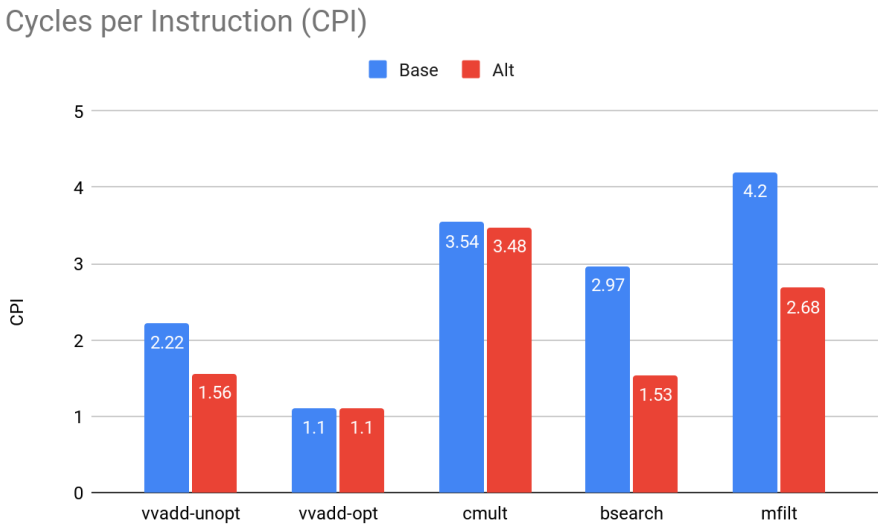

Figure 7: Cycles per instruction for simulation benchmarks

As shown in Figure 6 and 7, the total number of cycles and cycles/instruction decreased for every benchmark except for optimized vector-vector add. This was expected though since the optimized vector-vector add program did not contain any dependencies. The small difference between the base and alt design steady state for the cmult benchmark showed a small increase in performance compared to other benchmarks. So the bypassed processor performs better depending on the program, specifically if the program contains dependencies that cause a lot of stalls in the unbypassed processor that can be resolved by bypassing. It is important to note that the bypassed processor in the worst case would perform like the base design like in the case of the cmult benchmark.

Table 2: Percent Difference between Alt and Base number of cycles and CPI

Benchmark	% Difference # Cycles	% Difference CPI
vvadd-unopt	42.43%	42.31%
vvadd-opt	0.00%	0.00%
cmult	1.68%	1.72%
bsearch	94.55%	94.12%
mfilt	56.84%	56.72%
Average	39.10%	38.97%

Table 2 contains the percent differences of number of cycles and CPI between the Alt and Base design, calculated by the equation % diff = base - alt / alt. On average, the number of cycles decreases by 39.10% and the CPI decreases by 38.97%. In the case where there are no dependencies, there is no difference between the Alt and Base design.

Qualitative Analysis:

The addition of bypass paths in the alternative design affects the critical path of our processor. The critical path is now extended by two additional muxes, since the new critical path would be the same as the old, but instead of ending at the original pipeline register, it takes the bypass path, goes through two muxes in the D stage, and ends in the D/X stage pipeline register. This slightly increases the cycle time of our processor, decreasing the clock frequency. However, since muxes are basic pieces of hardware, the new critical path is not a drastic change, and the cycle time and clock frequency will be minorly impacted.

In addition to increasing the cycle time, the bypass muxes also add area to our baseline design because it adds hardware. However this does not mean that there is a corresponding increase in energy consumption. We can calculate energy consumption of the entire instruction sequence by doing energy/program = energy/cycle * cycles/instruction * instruction/program. We know that the energy per cycle increases and the instructions per program stays the same. Then, the total energy consumption of the program depends on the cycles per instruction. From the simulation we found that the number of cycles per instruction (CPI) of the alternative design was always equal to or less than the CPI of the baseline design. This makes sense because the alternative resolves non-load use data hazards using bypassing instead of stalling, which greatly reduces the CPI of data dependent instructions. Additionally, the bypass muxes do not have a major impact on the energy consumption in themselves, as a four input mux can be made using 16 CMOS transistors, having two additional muxes would add 32 transistors to our design. Therefore, while on their own, they would increase the energy of a single cycle, the improvement in CPI negates the effects of the increased energy consumption per cycle when there is a noticeable difference in CPI. For example, in the binary search benchmark, the CPI is nearly halved from the baseline to the alternative design but in the optimized vector-vector add the CPI stays the same. For the binary search instruction sequence, there would be less total energy consumed, however, in the optimized vector-vector add there would be more total energy consumed.

Overall we feel that our alternative design is an improvement over the baseline design. It allows us to potentially greatly reduce the CPI which is able to increase the processor performance. Although the tradeoff is an increased critical path leading to a decreased clock frequency and potentially consuming more energy, the positive impacts outweigh the downsides. From the simulations, we can conclude that the alternative design gives us the biggest boost in CPI when there are many non load and non multiply data hazards. When there are no data dependencies, the alternative design has the same CPI and therefore a worse overall performance as the baseline design, since it also has a longer critical path and would use more energy per cycle. We can predict this to be true for all instructions of these patterns, that is that an instruction sequence that has many data hazards will in general see improved performance with the alternative design due to the diminished need for stalling, however if those hazards are mainly loads or multiplies, then the benefits will be more minor as the processor will still need to stall.

Conclusion

The bypassed processor design requires the addition of two muxes into the base design datapath. The critical path is affected and increases the cycle time by the mux latency. Additionally, area increases in order to add more hardware to the datapath. Since the CPI and number of cycles have significant decreases in programs that

would depend on stalling without bypassing, energy consumption by the added area is canceled out. Through benchmark simulations we found that on average, the number of cycles decreases by 39.10% and the CPI decreases by 38.97%. When programs contain no dependencies, there is no performance difference between the bypassed and unbypassed design. On any program, the bypassed processor will perform at the base design processor performance or better. Though there are area tradeoffs, the decreased cycle time makes the energy consumption of a bypassed processor comparable to a simple pipelined processor with better performance.

Work Distribution

The majority of code was written during partner coding sessions so that we could discuss what we were implementing and catch each others' mistakes. Katherine added the multiplier from Lab 1 into the datapath. Katherine wrote tests for register-register and register-immediate instructions. Elise wrote tests for jump, branch, and store instructions. Katherine wrote the Alternative Design section, and Elise wrote the Introduction and Conclusion sections. We equally split up the Testing Strategy section based on what tests we wrote, and the Evaluation section by quantitative and quantitative analysis.