Anne Potter (ap674) and Ashley Heckman (agh93)
Lab Report 2
ECE 4750
10/20/22

Lab Report 2

1. Introduction

In this lab, we implemented two different pipelined processors for the TinyRV2 instruction set architecture: one that uses stalling to resolve data hazards, and one that is fully bypassed but still needs to occasionally stall to resolve data hazards. Both designs are five-stage processors that pass data through the pipeline to complete the given instructions. The stalling implementation, or the baseline design, uses stalling to resolve data hazards. It does this by checking to see if the destination that we're writing to in any given stage is the same destination that we're writing to in a later stage. If it is, the instruction that came later (and is earlier in the pipeline) needs to stall. The bypassed implementation, or the alternative design, exploits the fact that data is available before it is written back to the register file to decrease the number of cycles necessary to produce the output. It does this by passing data from later stages in the pipeline back to the decode stage. This lab is intended to give us a deeper understanding of the TinyRV2 instruction set, as well as to familiarize us with and give us hands-on experience with the material we have covered in lecture. These topics include the Verilog hardware description language, pipelining, stalling, bypassing, and other microarchitectural techniques. The processor integrates the multiplier from the first lab and is the second step towards completing a full multicore system.

2. Alternative Design

The pipelined processor has five stages: fetch, decode, execute, memory, and writeback. Fetch is where the instruction is retrieved from memory and the PC is updated. Decode is where the instruction is read into usable fragments of information, register operands are read, data is passed into the multiplier, and the jump and link (jal) instruction is handled. Execute is when the branch comparisons take place, the operands are used to compute values (through multiplication, the alu, etc.), and the memory request is sent. Memory is where data from the memory request comes into our processor. Writeback is when the register file is written. In the baseline design, if an instruction that is later in the pipeline will write to a register that an instruction earlier in the pipeline needs, the processor stalls. In the alternative design, we take advantage of the fact that the data that we're going to write to the register is frequently computed before the writeback stage, and we pass that value back to the instruction in the earlier stage of the pipeline. This eliminates the need for stalling in almost every situation.

The alternative implementation is similar to the baseline design, but it differs from the original implementation by taking advantage of the pipeline properties. The datapath for both designs is implemented as a series of modules that represent each component, with comments surrounding all of the components that make it clear which component is in which stage. Each module has inputs and outputs, and we can wire the modules together through shared variables.

The alternative design's datapath differs from the baseline design's datapath only by the addition of bypassing capabilities. The fully bypassed processor can send data to the decode stage from the end of the execute, memory, and writeback stages. We implement this change by including two extra muxes in the decode stage. These muxes are placed very early in the decode stage, and they choose between the data from the register file, the data passed back from the execute stage, the data passed back from the memory stage, and the data passed back from the writeback stage. We bypass the data from right before the register write in each stage to ensure that we have the correct value and not a value that we don't care about. The baseline datapath is shown below in *Figure 1*, and the modified alternative datapath is shown in *Figure 2*.

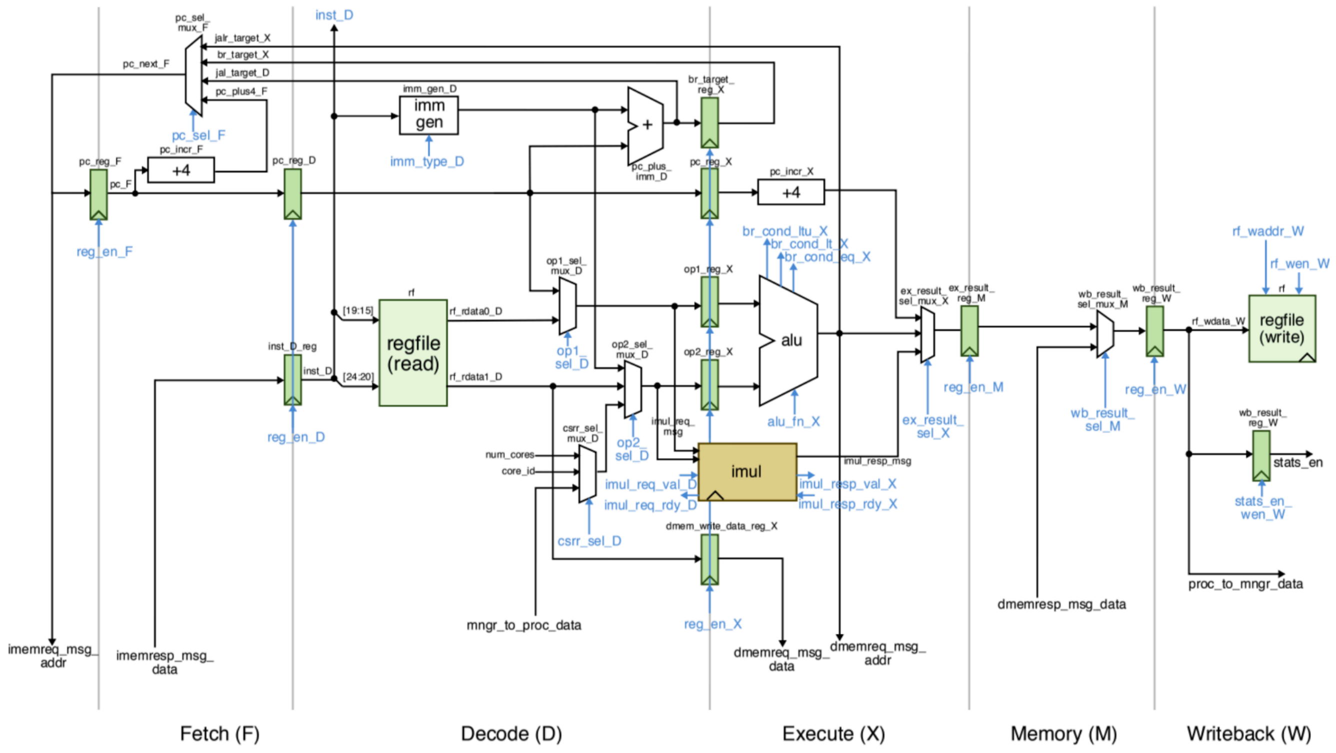

Figure 1: Pipelined Processor with Stalling Datapath (Baseline)

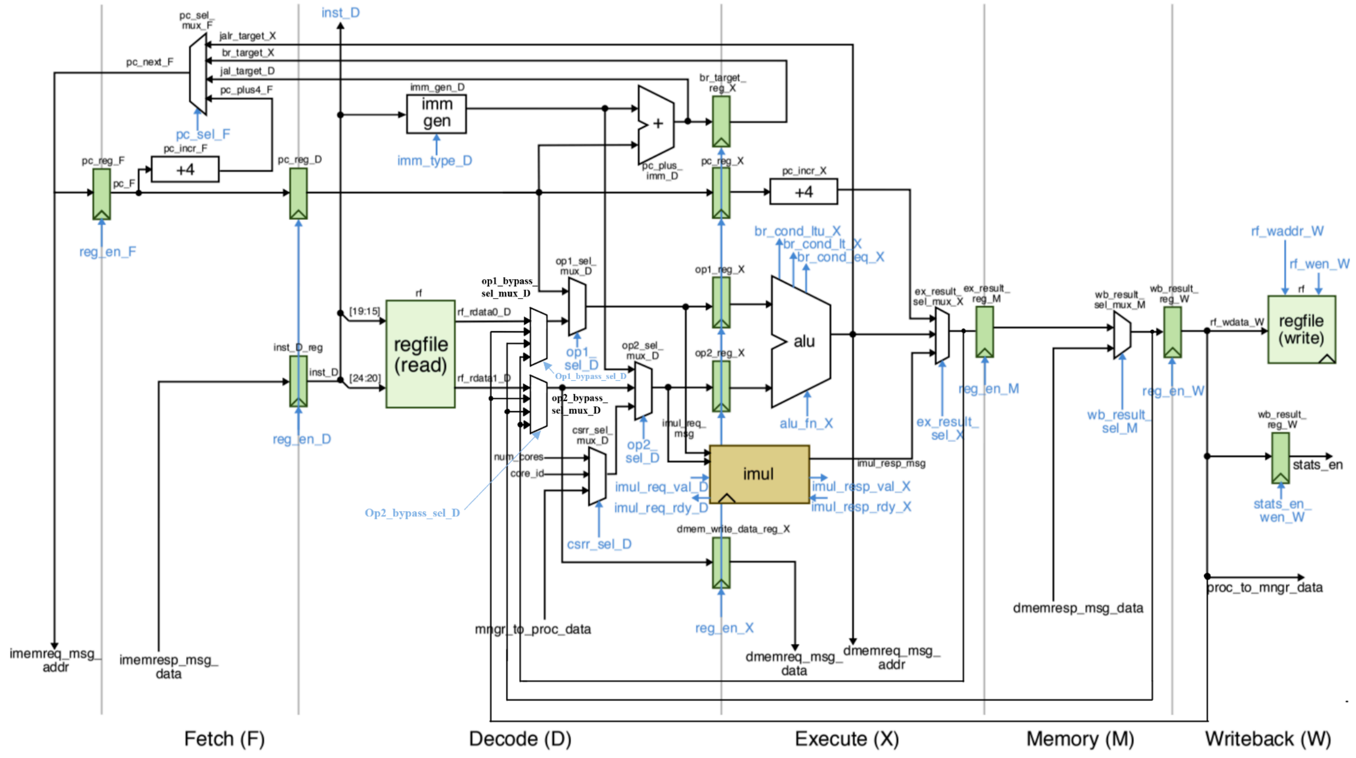

Figure 2: Fully Bypassed Pipelined Processor Datapath (Alternative)

The alternative design's control unit has a lot of changes from the baseline design. In the baseline design, if there are any data hazards, the pipeline simply stalls until they are resolved. In the alternative design, however, the processor tries to bypass before deciding to stall. The logic for the bypassing in the alternative design is identical to the logic for stalling in the baseline design in all stages except for the execute stage. To implement bypassing in the execute stage, we need to make sure that the instruction currently in the execute stage is not a load. This is because the data that we're going to store in the destination register isn't ready until the memory stage with a load. We only need to check that the instruction isn't a load in the execute stage because the memory stage is the latest that the data we're writing to the register file will become available for any instruction; if we're in a later stage, we don't need to check to make sure that the data is available - we already know that it is. Because the data from a load isn't available until the memory stage, if the load is in the execute stage and the next instruction (currently in the decode stage) needs that value, the next instruction must be stalled for one cycle. The execute stage is the only stage that will cause a stall due to data hazards. *Table 1* shows the stall then bypass that needs to occur with a load followed by an instruction that reads that value. *Table 2* shows the bypassing without stalling that happens for every other instruction and for the load when the next instruction doesn't need to read its value. The control signals for the bypass muxes are updated based on what we're bypassing, and we give priority to bypasses from earlier in the pipeline (the later instructions).

lw x2, 0(x1)	F	D	X	M	W		
addi x2, x2, 0		F	D	D	X	M	W

Table 1: Stall then Bypass with LW Dependency

add x1, x3, x3	F	D	X	M	W	
Mul x2, x1, x3		F	D	X	M	W

Table 2: Bypassing without LW Dependency

The bypassing eliminates cycles that would have been used to wait for that data to exit the writeback stage. The addition of bypassing to the alternative implementation allows us to limit stalling to two situations: when a load word instruction is immediately followed by an instruction that uses its output, or when the multiplier is still doing calculations for a multiply instruction. *Table 3* shows the pipeline diagram for the baseline design for the same sequence of instructions as in *Table 2*. It is obvious that the sequence in *Table 2* completes in fewer stages. Incorporating bypassing greatly increases our throughput.

add x1, x3, x3	F	D	X	M	W				
mul x2, x1, x3		F	D	D	D	D	X	M	W

Table 3: Same Sequence as Table 2, but with no Bypassing

When implementing our alternative design, we tried to keep as much in common with the baseline design as possible. We did this because the baseline design reflects the patterns and principles of modularity, hierarchy, and encapsulation. It exhibits modularity by decomposing the design into two modules: the datapath and the control unit. Within these modules, we are calling smaller modules. The design is broken down into very simple pieces. The design shows hierarchy with this decomposition into smaller modules. The top level calls the datapath and control unit modules, which then call their child modules, and so on. It's very easy to follow and see what module calls what, and to clearly see what the higher level modules are. The design displays encapsulation by hiding implementation details, such as the processor latency, from the interface by using the val/rdy signals. Our alternative design displays modularity, hierarchy, and encapsulation for the same reasons that our baseline design does.

3. Testing Strategy

We took an incremental approach to testing our implementation. First, we created unit tests for our immediate generation and alu models, as we wanted to be sure these worked before we started testing the instructions that used these. We then created a new set of tests for one instruction, testing that instruction until we were confident that it was functioning the way we intended. After we were confident that the instruction was working, we repeated this process for the next instruction. Because the interfaces of both designs are the same, we were able to use the same tests for both implementations. Every set of tests began with ad hoc tests of the most simple RISCV instructions for that case. Throughout our testing, we used a mix of directed, random, value, delay, and ad-hoc testing. We did this mix of testing to try to hit as many corner cases in our code as possible and to verify that our designs were working as expected.

At first, we were only concerned with whether our code was producing the correct output. We were confident that the add, lw, bne, csrr, and csrw were correct since they were provided to us. Instead of adding more tests to check these instructions for correct functionality, we used these tests as a way to become familiar with the way to test other arithmetic, memory and branching instructions. We began testing our designs with blackbox testing, which is only concerned about generating the correct output given the inputs, not with the implementation. This allowed us to verify the functionality of our code without worrying about the specifics of what's going on internally.

For each instruction, we created some simple positive and negative tests to test basic functionality, large positive and negative numbers to ensure functionality was consistent over all 32 bits, overflow tests to make sure that was working the way we expected, and some varying addresses to ensure our code is working for all register and memory addresses. We then created some random tests and some corner case tests.

Once we determined that our implementations were producing the correct outputs for a given instruction, we added line tracing, which is white-box, to ensure that we were progressing in the way we intended. We wanted to make sure that we were stalling (and for the alternative design, not stalling) in the way that we expected, and that we were moving down the pipeline as expected. To do this, we ran the test case for the specific instruction with the -s flag. This allowed us to view the line trace, and see how the inputs, outputs, and pipeline stages were changing with each cycle. This felt very much like an ad-hoc test since it relied on us observing what the data looked like.

We decided to use assertion testing because we wanted our test cases to be systematic and automatic; we wanted a test suite where the output is automatically checked for us. We implemented this by explicitly writing assertions that had to be true in order for the test to pass.

We wanted to use a mix of random and directed tests on our designs. We used directed testing to target specific input patterns and corner cases, and we used random testing to improve coverage and further ensure that our designs are functioning properly. With only directed cases, there's a chance that our implementation only works for those specific values, and random testing eases that fear. Random testing improves code coverage and helps bring to light any bugs we may have missed in our directed tests. We implemented directed testing by specifying specific inputs and expected outputs for those inputs. We implemented random testing by generating random numbers for our inputs, then using functions that are known to work, and put that as our expected output. Random testing allows us to randomize all aspects of our tests, like the address, number of delays, and offsets, not just the data we manipulate.

We wanted to use value testing to ensure that our processors were producing the correct outputs given the inputs. If the value of the output is incorrect given the value of the inputs, then the processor is not functioning properly. We did this by using assertions tests to ensure that the expected value of the output is produced. We also wanted to have some delay tests. In addition to ensuring that the processor is producing the correct value, we also want to ensure that it can handle arbitrary source and sink delays. We implemented this by calling some of our previously created test cases with source and/or sink delays. We feel that with all of these different testing strategies and test cases, we have ensured that our code works properly. We have a lot of code coverage with a lot of different types of inputs and instructions.

4. Evaluation

Our alternative design is better than our baseline design. In the worst case scenario for the alternative design, there is a data dependency between a load word instruction and the instruction that immediately follows it, and the processor stalls for one cycle. In the best and average case scenario for the alternative design, there are either no data dependencies or there are data dependencies that don't include a load word instruction needing to bypass data, and the processor doesn't need to stall at all. For the baseline design, the best case scenario is that there are no data dependencies, in which case the processor doesn't stall, and the worst case scenario is that there is a data dependency between instructions immediately following each other, and the processor needs to stall for 3 cycles. The alternative design takes much fewer cycles to complete, so its performance is much better than the baseline design. *Figure 3* and *Figure 4* show the number of instructions and the cycles per instruction of the baseline and alternative designs when running the same programs. We can see that the CPI for the alternative design is always less than or equal to the CPI of the baseline design, and it is usually much less than that of the baseline design. It is clear that the alternative design completes in fewer cycles on average.

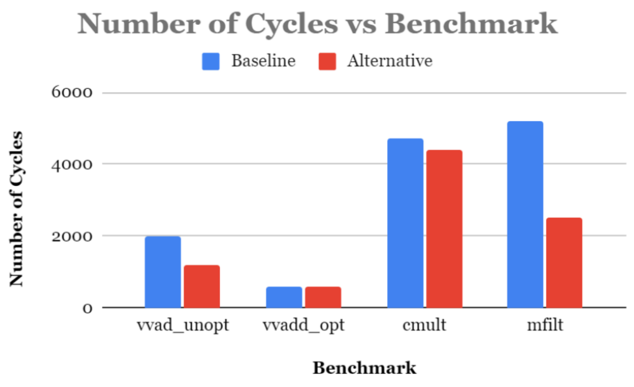

Figure 3: Number of Cycles for Different Benchmarks

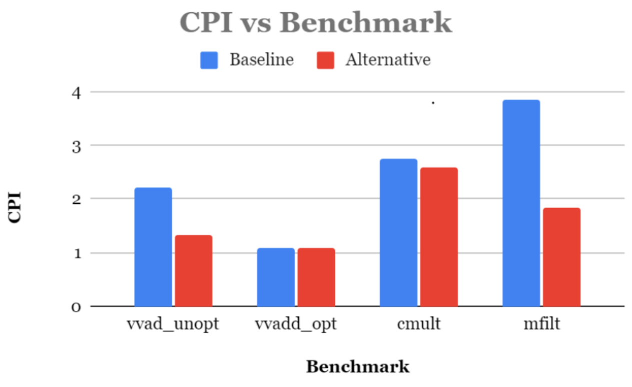

Figure 4: CPI for Different Benchmarks

The cycle time in our alternative design is longer than that in the baseline design. The critical path of the baseline design goes straight from one stage to the next stage, while the critical path of our alternative design goes through one of the later stages, gets bypassed back to the decode stage, and finally ends at the registers between the decode and execute stages. This will be quite a bit longer than the critical path of the baseline design, so the cycle time of the alternative design is longer.

Adding another piece of functionality to our implementation requires additional hardware. Our alternative design does a couple extra checks for stalling and bypassing logic and sends the data to be written back to the decode stage when needed. Since we check additional conditions and since we have two extra muxes, we need quite a few more gates when compared to our baseline design. Thus, our alternative design uses more area than our baseline design.

The additional functionality also means that more energy will be required to process each cycle of the alternative implementation. However, because there will be, on average, much fewer cycles, the overall energy consumption will be smaller in the alternative design than in the baseline design.

5. Conclusion

Our alternative design performed better than our baseline design when there are instructions with dependencies, and it performed very similarly to our baseline design when there are no instructions with dependencies. When there are instructions with data hazards, the alternative design completes in fewer cycles. When there are no instructions with data hazards, the alternative and baseline designs complete in the same number of cycles. Since the cycle time is a little bit longer for the alternative design, it will take a little bit longer to complete when there are no data hazards between instructions. In the average case, however, there are instructions with data hazards, and the alternative design completes in much fewer cycles than the baseline design. Even though the cycle time of the alternative design is a little bit longer, its performance is still much better since it saves so much time by not stalling. In the future, we should use the alternative design.

6. Work Distribution

Anne began the baseline datapath and control system and Ashley completed it. Anne coded the logic for the bypassing and new stalling in the alternative processor, and Ashley debugged it. Anne and Ashley worked together to code the test cases and the alternative design, then debug those test cases and the alternative design. Ashley also completed the evaluations. Anne and Ashley then worked together on the lab report, with Ashley focusing more on the evaluation and alternative design section, and Anne focusing more on the introduction, conclusion, and testing sections.