Lab 2: Pipelined Processor

Asma Ansari (ara89) & Nathan Vogt (ncv7)

I. Introduction

Pipelined processors divide each part of the data path into individual stages such that more than one instruction can be passed into the data path. However, this implementation can cause data hazards which can be resolved by either stalling instructions in a particular stage or bypassing data from later stages back into earlier stages. In this lab, we implement two versions of a five-stage pipelined processor, one version includes stalling while the other includes both stalling and bypassing to resolve data hazards.

The five stages that the data path is split into are Fetch (F), Decode (D), Execute (X), Memory (M), and Writeback (W). Both the baseline and alternative processor designs contain the same stages in the data path, but the alternative processor can link data from the X, M, and W stages back to the D stage to solve data hazards and optimize the number of cycles per instruction (CPI). Furthermore, the alternative design still contains stalling due to the fact that certain instructions may need more than one cycle to calculate or read/write values that newer instructions are attempting to read/write.

II. Alternative Design

The alternative design builds on our baseline implementation by including bypassing from X, M, and W stages to the D stage as a way to resolve data hazards quicker. Although stalling alone can address data hazards, it is not the most efficient solution since it slows down the progression of newer instructions.

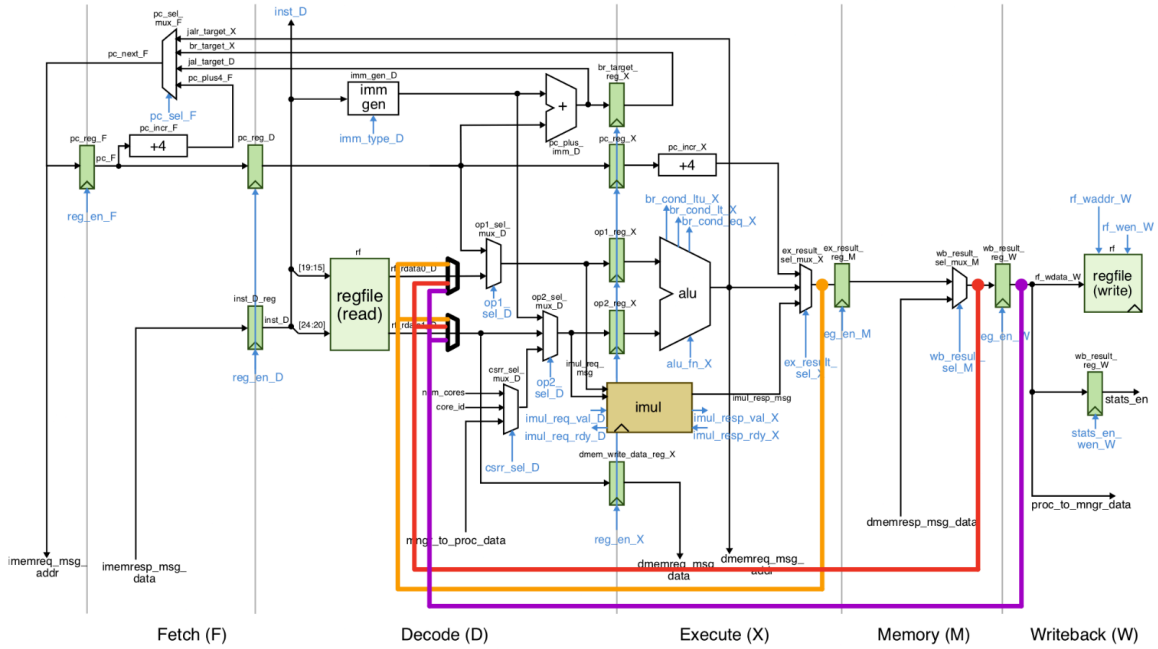

Figure 1: Data path diagram for the alternative design.
Credit: Figure 13 – ECE 4750, Fall 2024, Lab 2: Pipelined Processor

In the diagram above, the purple, orange, and red wires connect the outputs of the X and M stages, the input of the W stage, and the outputs of the regfile to two separate muxes drawn in black. These muxes are placed after the regfile so that any instructions that require the use of either or both outputs of the regfile can get the appropriate

values for the X stage. In the baseline design, newer instructions would be stalled as an older instruction passed through the pipeline before accessing updated values.

Example:

```
ADDI x1, x2, 3 |  F  D  X  M  W
ADDI x3, x1, 4 |     F  D  X  M  W
```

The example above illustrates a read-after-write (RAW) data hazard. Without bypassing, the second ADDI instruction would have to stall until the first ADDI instruction exits the pipeline. However, since there is a wire that connects the X stage's output to the D stage's input, the updated value stored in x1 can be sent back to the D stage so that the second ADDI instruction can continue progressing. In the baseline design, the second ADDI instruction would have to stall in the D stage until the first ADDI instruction writes the regfile in the W stage to update the value in x1.

There are specific instructions where stalling is still necessary, such as LW and MUL. Our multiplier is variable latency, meaning that instructions in previous stages must wait until the calculation is complete. However, once the processor stops stalling, the output from the multiplier can be bypassed to newer instructions. The LW instruction must progress to the M stage to receive the value from memory which could potentially require an instruction in one of the earlier stages to stall until the output from the M stage has been acquired.

Furthermore, we updated the control unit to include signals for bypassing and adjusted the stalling logic to ensure that stalling only occurs when necessary. In the baseline design, the D stage contained several stall signals. We changed these stalls to bypass signals and created combinational blocks that determined which data value to select (i.e. bypassed value from a later stage or data read from the regfile) for each of the muxes we added in our diagram based on these bypass signals.

We used two wires called "bypass_sel_1" and "bypass_sel_2" to indicate which mux to select between (rs1 and rs2). These were signals produced from the control unit and connected as inputs to the datapath. The combinational block set the value of each mux select based on which bypass signal was high. Each of these bypass signals checked if the source registers' addresses in the X, M, and W stages matched the source registers' addresses in the D stage. If they matched, then the bypass signal would be set high, meaning that stage's register value should be bypassed to the D stage rather than accept the source register's value from the regfile.

Through developing both our baseline and alternative designs, we had to use modularity and encapsulation to simplify the process of adding on to our datapath. Our multiplier, immediate generator, and ALU encapsulate the logic for the functionality it provides to the datapath since we are only concerned with receiving the appropriate outputs from each module. This encapsulation supports the modularity of the datapath's design since each component can be described at a high-level as illustrated by Figure 1. Exposing the internal details of how each module functions is not important at the level of the processor which utilizes the inputs and outputs of these modules to ensure the progression of a program.

III. Testing Strategy

Category	Instructions	Tests
Reg-Reg	add, sub, mul, and, or, xor, slt, sltu, sra, srl, sll	Passes: ☑ Alternative ☑ Baseline Contains: ☑ Directed test cases with manual instruction sequences written out ☑ Template test functions from inst_utils.py to generate instruction sequences and check their behavior ☑ Delay testing

		☑ RAW, WAR, + WAW data hazard tests for ADD
Reg-Imm	addi, ori, andi, xori, slti, sltiu, srai, srli, slli, lui, auipc	Passes: ☑ Alternative ☑ Baseline Contains: ☑ Directed test cases with manual instruction sequences written out ☑ Template test functions from inst_utils.py to generate instruction sequences and check their behavior ☑ Delay testing ☑ RAW, WAR, + WAW data hazard tests for ADD
Memory	lw, sw	Passes: ☑ Alternative ☑ Baseline Contains: ☑ Directed test cases with manual instruction sequences written out ☑ Template test functions from inst_utils.py to generate instruction sequences and check their behavior ☑ Delay testing ☑ RAW + WAR data hazard tests for LW
Branch	bne, beq, blt, bltu, bge, bgeu	Passes: ☑ Alternative ☑ Baseline Contains: ☑ Directed test cases with manual instruction sequences written out ☑ Template test functions from inst_utils.py to generate instruction sequences and check their behavior ☑ Delay testing ☑ RAW, WAW, + WAR data hazard tests for BNE
Jump	jal, jalr	Passes: ☑ Alternative ☑ Baseline Contains: ☑ Directed test cases with manual instruction sequences written out ☐ Template test functions from inst_utils.py to generate instruction sequences and check their behavior ☑ Delay testing ☑ RAW + WAR data hazard tests for JAL
CSR	csrr, csrw	Passes: ☑ Alternative ☑ Baseline Contains: ☑ Directed test cases with manual instruction sequences written out ☑ Template test functions from inst_utils.py to generate instruction sequences and check their behavior ☑ Delay testing

Table 1: Table containing instruction types along with whether they pass on the alternative and base designs.

Incremental development and testing was crucial in successfully implementing both the baseline and alternative design. Before considering bypassing, we had to ensure that all 34 instructions we implemented in the baseline design functioned properly. In the lab handout, these instructions were grouped into categories depending on their expected outcome and/or behavior (i.e. memory, jump, branch, etc.). Thus, we started by implementing the instructions that required the least amount of changes to the datapath and/or control unit.

The reg-reg and reg-imm instructions required the use of an ALU to calculate the appropriate outputs, such as addition and subtraction. We implemented all of the required ALU functions and wrote a unit test to ensure that all ALU calculations were correct before implementing the aforementioned instructions. For the MUL instruction specifically, we used our multiplier implementation from Lab 1 in our data path. By proxy, the unit testing for our multiplier is already complete, so wiring the multiplier correctly was more important. For this instruction, we used integration testing to debug our multiplier's val and rdy signals as well as the squash and stall logic required to prevent any data hazards from occurring.

Furthermore, we used black-box testing to test for edge cases by only looking for an expected outcome or behavior from the instruction as it passes through the pipeline. For the majority of instructions, we used delay testing and random value generators to increase the coverage for our baseline implementation. Doing this dual purpose in that we could verify the correctness of our stalling logic as well as our instruction's behavior. For example, generating random values to input for reg-reg or reg-imm instructions tests stalling whether that is due to a variable-latency calculation or data dependency.

Additionally, we utilized an FL model to verify the correctness of our test cases so that any errors thrown from running the tests will be due to an error with the implementation of an instruction or framework. Before implementing each instruction, we developed our test cases and tested them on the FL model. Our test cases were a mix of test cases where we manually wrote a list of instructions as well as templates from the inst_utils.py file. For the jump instructions specifically, it was simpler to write manual test cases that verified different behaviors rather than develop a template function. As illustrated in the table, we tried to maintain a consistent test design across all instruction categories, but ultimately, the goal was to ensure that our instructions functioned correctly in each design.

Another aspect of inst_utils.py is that many of the templates tested dependencies between CSRR/CSRW and the instruction being tested. This type of test was useful to verify our stalling logic and our bypassing logic once we implemented our alternative design. Testing our alternative design involved the same test cases that were used on our baseline design. As described before, we changed the stalling logic in our control unit to include bypassing, so we first verified that our instructions still functioned correctly even with these changes. After that, we used the provided evaluation tests to check whether our bypassing logic was correct. The evaluation tests calculated CPI for the FL, baseline, and alternative designs for various algorithms. Therefore, the alternative design should have the lowest CPI, indicating that the bypassing logic was working.

VCD files and traces assisted us with debugging any instructions that were not working properly in either design. Throughout developing the baseline design, we had to continuously add more datapath until we could accommodate all 34 instructions. The alternative design relied more heavily on correct control unit logic, but we also had to be conscientious of the connections into and out of the muxes within the D stage. Thus, visualizing the waveforms for different signals, such as mux selections, ALU/multiplier functions, enables, register values, and so much more helped us pinpoint whether we had an issue in the control unit and/or datapath for either design.

We also included some more manual instruction sequence test cases in specific instructions that had more clear RAW, WAR, and WAW data hazards to support this form of debugging during our implementation. Our design flow ended up being: (1) create test cases, (2) run on FL model, (3) implement instruction/bypassing, and (4) verify using test cases.

IV. Evaluation

Algorithm	Baseline CPI	Alternative CPI
vvadd-unopt	2.22	1.34
vvadd-opt	1.10	1.10
cmult	3.06	2.88
bsearch	2.97	1.38
mfilt	4.27	2.28
Average	2.724	1.796

Table 2: Evaluation of both the baseline and alternative based on the CPI for various algorithms.

The simulator provided to us utilized the instructions we implemented in our base or alternative design to complete vector-vector add ("vvadd-unopt", "vvadd-opt"), complex multiplication ("cmult"), binary search ("bsearch"), and masked convolution ("mfilt"). Notably, the baseline and alternative design had the same CPI for the "vvadd-opt" algorithm since this algorithm optimized the instructions such that there were no longer any data hazards. Thus, the CPI is not affected since there would be no need for stalling and/or bypassing in either implementation.

There was a significant reduction in CPI for the "bsearch" and "mfilt" algorithms, but "cmult" only had an approximately five percent reduction in its CPI. Complex multiplication requires the use of the multiplier module within our datapath. Since our multiplier is variable latency, the multiplier will stall the X stage (and previous stages) until the calculation is complete. Therefore, this unavoidable stalling prevents data hazards by allowing older instructions to pass through the pipeline and forcing younger instructions to stall until the X stage has a valid output. Any data hazards in between the multiplier's X stage stall were resolved through bypassing as indicated by the smaller CPI.

In addition to all of the registers, muxes, and other relevant modules in the baseline design, the alternative design includes two muxes in the D stage that support bypassing. Therefore, both designs are not drastically different in terms of area, indicating that the tradeoff between area and performance is not significant. However, this additional muxing can affect the critical path which impacts the processor's clock cycle time.

Without bypassing, newer instructions that have data dependencies with older instructions must stall until the updated data values are written in the W stage. However, connecting the outputs of the X and M stage as well as the input of the W stage to the D stage reduces the critical path by eliminating the need for newer instructions to wait to read the data directly from the regfile. As a result, the cycle time should be lower which further indicates the performance benefits of using bypassing.

The energy consumption between each design is also dependent on the average CPI for each design. The average CPI across the five algorithms for the baseline and alternative designs are 2.724 and 1.796 respectively which is a 33% decrease in average CPI. However, with algorithms that use more latency-heavy instructions, such as MUL, energy consumption between designs is nearly equal. Similar to how our multiplier's inputs heavily dictate its energy consumption, each design is only as energy efficient as the instructions that pass through it.

As noted before, the "vvadd-opt" algorithm did not have any data hazards, so both designs had an equivalent CPI and energy usage, but the alternative design uses more area, making that the main tradeoff between the two designs. However, more realistically, most programs will have several data dependencies that will likely cause data hazards, so bypassing is still the most preferable option to resolving these hazards.

V. Conclusion

The evaluation results revealed the importance of design optimization, especially given that the performance benefits were achieved through a small addition to the data path. As mentioned in the previous section, the algorithms that run on the alternative design will determine how much it actually improves performance. On average, the CPI improved by at least half a cycle for each program with the largest improvement coming from the "mfilt" algorithm. Ultimately, the alternative design is the better option for a realistic situation since most, if not all, programs will have data dependencies which may or may not cause data hazards.

As discussed in class, there have been many advances within the chip industry in the last 50 years, but there is a limit to how many transistors can be fit onto a chip as well as its power efficiency. Optimizing our usage of a chip's resources enhances the computational power of that chip without having to change the hardware at a fundamental level. Developing different types of processors diversifies the use cases where each processor performs the best, but having a processor that generally performs well is crucial.

VI. Work Distribution

We implemented the majority of instructions for the baseline design together, but Asma finished the baseline and alternative designs independently. Both group members worked on the lab report prior to submission.