# Computer Architecture Report for Lab 2

資科工所 張勝迪 312551006

## 1. Introduction

This project focuses on optimizing a RISC-V-based CPU through three key tasks aimed at improving performance. First, we expanded the instruction table to support the complete RV32I and RV32M instruction sets under the existing stall mechanism. Next, we introduced bypassing logic to minimize performance degradation caused by stalling, enabling instructions to execute faster through data forwarding from later stages to the D stage. Lastly, we replaced the original iterative muldiv unit with a pipelined version, integrating it into the CPU to further enhance instruction throughput and efficiency.

## 2. Design

### a. Implementation Methodology

To ensure that the data read from registers is always consistent with the latest value, I focused on ensuring that when the **op0_mux** or **op1_mux** selects values to be read from the register file, the mux should be able to access the most recent values, even when the register file hasn't been updated yet. To achieve this, I implemented additional multiplexers (referred to as **rs_mux**). These multiplexers check whether the needed register values were recently updated in the X, M, or W stages, and if so, the muxes select the corresponding bypassed values instead of stalling.
For example, when reading from rs1, we check if:

- The rs1 read is enabled.
- The instruction in the X stage is valid (not a bubble).
- The instruction in the X stage requires a writeback.
- The writeback register address in the X stage is not 0.
- The rs1 address matches the writeback register address in the X stage.

If all these conditions are met, we set the **rs1_mux** to select the bypassed value from the X stage. The same logic is applied for the M and W stages. This process ensures that data dependencies can be resolved without needing to stall the pipeline, significantly improving performance.
After implementing this basic bypassing mechanism, there remained an issue with load-use data hazards. Since the load instruction gets its result in the M stage, we cannot bypass its value during the X stage. This requires stalling for one cycle to avoid using incorrect data. Specifically, I added logic in the D stage to check whether the instruction in the X stage is a load. If it is, and a subsequent instruction has a dependency on the load's result, we insert a one-cycle stall. Once the load instruction reaches the M stage, the value is bypassed back to the decode stage, allowing the

dependent instruction to proceed with correct data. This approach resolves the load-use hazard while minimizing stalls.

**b. Design Justification**

In implementing bypassing, the key change was to eliminate most stalls. In the original design, if an instruction in the decode stage had a  RAW data dependency with any of the previous three instructions, the pipeline would stall for up to three cycles to wait for the data to be written back, allowing the instruction to access the correct values. However, stalling causes an increase in cycle count, negatively impacting performance. My approach was to bypass data from the X (execute), M (memory), and W (writeback) stages back to the decode stage instead of stalling. The primary motivation for these changes was to reduce the negative performance impact of stalling. By bypassing data between pipeline stages, we eliminate most stalls, allowing the CPU to continue executing instructions without unnecessary delays. This is particularly important for improving the throughput and overall performance of the processor.
The decision to stall only in the case of load-use hazards was based on the fact that load instructions cannot have their results bypassed in earlier stages. Introducing a single-cycle stall in this scenario is necessary to ensure correct program execution, but it minimizes the overall number of cycles wasted on stalls.
By combining bypassing logic with minimal stalling, this design achieves a good balance between performance improvement and correctness, effectively addressing data hazards in the pipeline.

## 3. Testing Methodology

For the Testing Methodology, I focused on ensuring the correctness of each module and the overall system by testing individual components and edge cases. Initially, I verified the functionality of each newly added instruction by running unit tests to confirm the correct handling of data dependencies and bypass logic. For the bypassing mechanism, I created test cases with various data hazards, including RAW (Read After Write) hazards, to ensure proper forwarding without stalling. Additionally, I simulated **load-use hazards**, where stalling was required for correctness, and tested pipeline performance with different types of hazards to confirm proper stall insertion. I also ran corner cases involving complex instruction combinations, such as dependent arithmetic and memory operations, to ensure accurate handling under all conditions.
This is a snippet of the test case I designed, which includes testing the correctness of load-use and checking the forwarding handling **by inserting different numbers of NOPs** during division operations:

```
TEST_LD_SRC0_BYP( 0, lw, 0, tdata_0, 0x000000ff )
TEST_LD_SRC0_BYP( 1, lw, 4, tdata_0, 0x00007f00 )
```

```
TEST_RR_DEST_BYP( 0, div, 143, 11, 13 )
TEST_RR_DEST_BYP( 1, div, 154, 11, 14 )
TEST_RR_DEST_BYP( 2, div, 165, 11, 15 )
TEST_IMM_DEST_BYP( 0, addi, 13, 11, 24 )
TEST_RR_DEST_BYP( 0, and, 0xff00ff00, 0x0f0f0f0f, 0x0f000f00 )
TEST_SW_SRC01_BYP( 0, 0, sw, 0xaabbccdd,  0,  tdata_4, 0xaabbccdd )
```

Finally, full system testing with benchmark programs was conducted to evaluate overall performance improvements.

## 4. Evaluation

| Benchmark | Processor | Num Cycles | Num Inst | IPC |
|---|---|---|---|---|
| **ubmark-vvadd.c** | stall | 570 | 453 | 0.794737 |
| | byp | 471 | 453 | 0.961783 |
| | long | 471 | 453 | 0.961783 |
| **ubmark-cmplx-mult.c** | stall | 16846 | 1873 | 0.111184 |
| | byp | 15325 | 1873 | 0.122219 |
| | long | 2750 | 1873 | 0.681091 |
| **ubmark-masked-filt.c** | stall | 16271 | 4764 | 0.292791 |
| | byp | 14015 | 4764 | 0.339922 |
| | long | 6553 | 4764 | 0.726995 |
| **ubmark-bin-search.c** | stall | 3096 | 1030 | 0.332687 |
| | byp | 1415 | 1030 | 0.727915 |
| | long | 1415 | 1030 | 0.727915 |

## 5. Discussion

### a. Performance differences

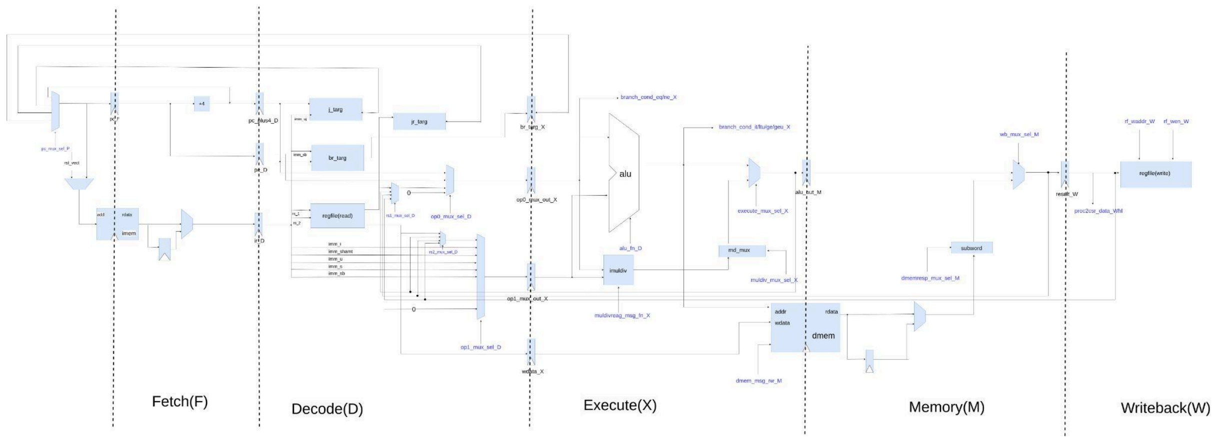
From the results in Section 4, it can be seen that bypassing generally optimizes stalls; however, the extent of improvement varies depending on the test program. Specifically, in ubmark-cmplx-mult.c, the IPC difference between the two is minimal, while there is a significant difference in ubmark-bin-search.c. Based on my design logic, it can be inferred that this is because the former does not have a large amount of data dependency, making the presence or absence of a bypassing mechanism have almost no effect on CPU cycles. Conversely, if the latter frequently activates the stall mechanism due to data dependencies, it results in an increased cycle count, which demonstrates the performance differences we observed. Additionally, the difference between the long and bypassing implementations depends on the number of muldiv operations; the more operations there are, the more cycles the long implementation can be saved.

**b. Trade-offs between bypassing and stalling**

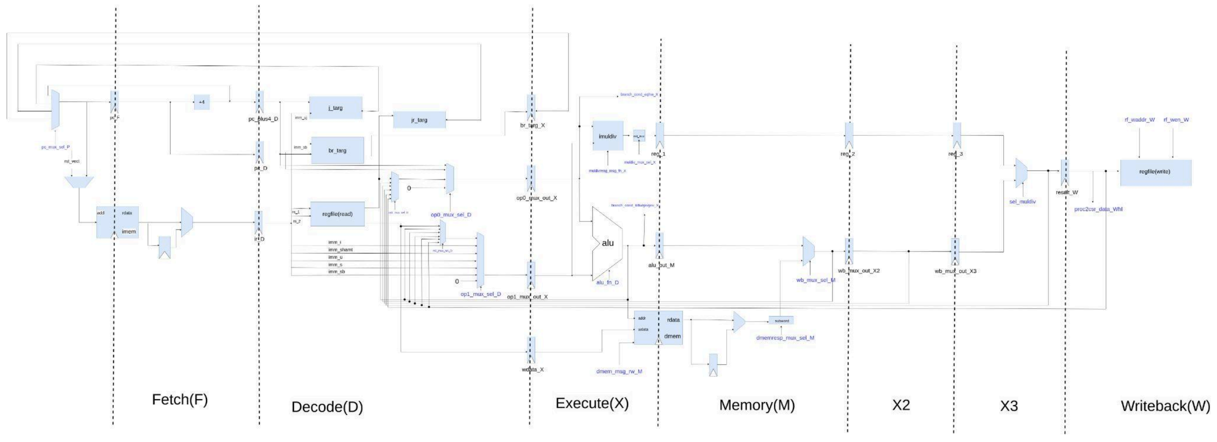
Based on the test results, we find that bypassing reduces the number of cycles for instructions in many tests, thereby lowering the CPI. However, the design approach it employs relies on additional hardware and decision logic. For example, in my implemented CPU, an rs_mux was added, which increases the time spent per cycle. Therefore, we can conclude that bypassing, compared to the stall mechanism, l**owers the CPI** but **increases the clock cycle time**. According to the Iron Law of Processor Performance, the overall performance must consider the balance between the two.

## 6. Figure

### a. Bypassing (5-stage)



### b. Pipelined Muldiv (7-stage)


alu
imuldiv
regfile(read)
regfile(write)
dmem
imem
Fetch(F)
Decode(D)
Execute(X)
Memory(M)
X2
X3
Writeback(W)