Lab 2 Report: Pipelined Processor

Group 71: Yibin Xu (yx623)
Yanwen Zhu (yz2949)

Section 1: Introduction

The lab offers an exploration into the design of pipelined processor microarchitectures. We focus on the TinyRV2 ISA, which forces two designs: a baseline five-stage processor pipeline that utilizes stalling to manage data hazards, and an alternative design that uses bypassing to resolve data hazards and enhance performance, both of which use hardware speculation to resolve control hazards and can run simple C programs which don't use system calls. We aim to learn about the ISA, how to implement these instructions and implement the basic pipeline microarchitecture that we learned in lectures. We also get familiar with stall and bypassing microarchitecture implementations and successfully resolve data and control hazards using hardware methods that we learned in lectures. We also use many test cases to test each instruction and run several benchmark programs to evaluate the performance of the two processors. The baseline and alternative designs are both similar to the processor we learned about during the lecture, which utilizes a register to stall the data or bypass it if we encounter a data hazard.The data path of the alternative design uses the same structure as the slide shown in class. Notably, the results reveal that the alternative design achieved tasks in fewer cycles and less time than the baseline design for identical instructions. Because the baseline design has to wait until the previous instruction is finished, the alternative design can utilize the bypass units. It also concludes that CPI of alternative design < CPI of Baseline design < CPI of MultiCycle design. At last, we also compare the two designs to evaluate the trade-off between improving performance and increasing hardware complexity of alternative design.

Section 2: Baseline Design

2.1 Overview

The lab assignment centers on developing a five-stage stalling processor compatible with the TinyRV2 ISA. The processor consists of a datapath for data flow and a pipelined control unit for management. For clarity, each component, including a parent connector, is in separate files. The processor works through five stages: fetching, decoding, arithmetic operations, memory access, and register writing. While a baseline datapath is provided, we must further develop and modify the design. As more instructions are added, updates to the datapath and control unit are expected.

Our baseline design for developing a five-stage stalling processor compatible with the TinyRV2 ISA is shown in Figure 3. The processor consists mainly of a datapath, control unit, immediate generator, ALU unit, and Top modules. Our Baseline design takes the assembly code and performs the corresponding operation. In each instruction cycle, the instruction will be processed through five stages: F – Fetch instructions, increment PC; D – Decode instructions, read register, handle jumps; X – Arithmetic operations, branch comparison; M – Access data memory; W – Write back the register file.

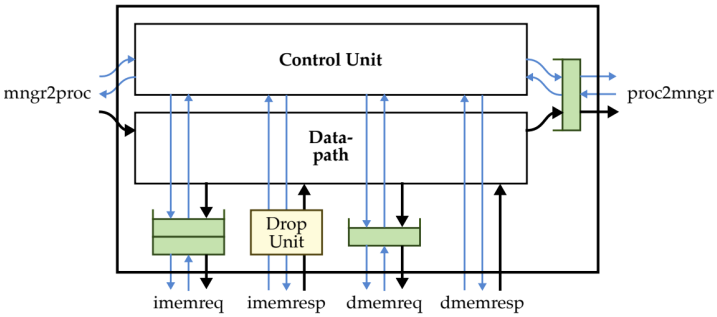


Figure 1. Processor Datapath and Control Composition

For example a MUL instruction, *mul x3, x2, x1*:

- The Mngr2proc initiates by reading the MUL instruction into the processor.
- During the Fetch stage, the Control Unit issues a mul request signal to imemreq. In response, imemresp sends back the instruction details to the datapath.
- In the Decode stage, the Control Unit dispatches request signals for the operands x2 and x1 to dmemreq. Subsequently, dmemresp returns the requisite data information to the datapath.
- The processor then sequentially advances through the Execution, Memory, and Write-back stages for the MUL instruction.
- Finally, the computed result is relayed to proc2mngr."

2.2 Data diagram

In our baseline design, encountering a data hazard prompts a complete halt of the processor's operations. The processor remains in this stalled state until all data hazards are resolved, after which it resumes its regular processing. To illustrate, should there be an attempt to read the value of x1 immediately following a write operation to the same register x1, the processor will be stalled. This ensures that the write operation to x1 completes successfully before any subsequent reads are performed. Also, we use the multiplier from lab 1 to replace the * operation in the ALU. When a MUL instruction is encountered by the processor, a preliminary check is initiated to determine if the multiplier is prepared to accept data from the datapath. If affirmative, the multiplier ingests the data and, upon

completion of its operation, generates an output readiness signal. In the subsequent step, before transmitting the result, the multiplier queries the processor to confirm its readiness to receive the computed output. Once the processor signals its preparedness, the multiplier dispatches the result to the datapath and concurrently transmits a data-valid signal, notifying the processor of the successful data transfer."

Section 3. Alternative Design

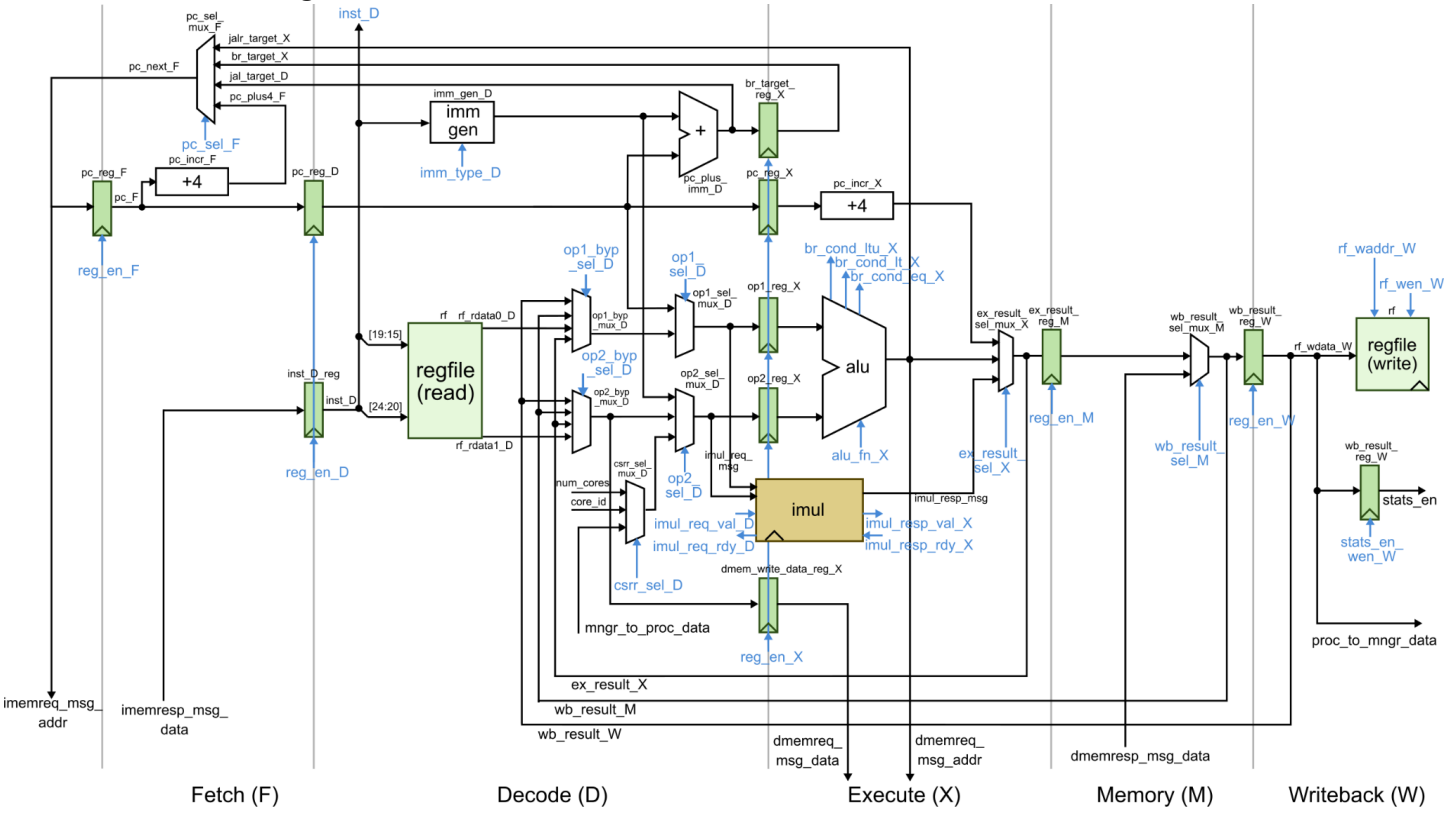
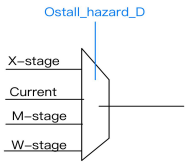


Figure 2. Alt Design Data path and control signal



Our alternative design is a fully bypassed processor which is based on the baseline design and adds the control and datapath for both bypass and squash implementation instead of using stalling to avoid data hazards. The optimization force avoids data hazards by forwarding values from later pipeline stages to earlier stages in a combinational path which can reduce the number of cycles. To correctly perform the bypass operation, we need to place two 2-bit bypass mux in both *rf_rdata0_D* and *rf_rdata1_D*. Those mux are controlled by two control signals from *ProcAltCtrl.v* to select the data from four datapaths: *rf_rdata0_D, ex_result_X, wb_result_M, and rf_wdata_W*(labeled as purple in Figure 1).

Figure 3. Mux for bypass control

The control signal for those two mux comes from the *ostall_hazard_D* from the Baseline design. The purpose of *ostall_hazard_D* in Baseline design is to detect if the result calculated by the previous instruction is used
by the current instruction. We can declare an *always_comb* function to assign *ostall_hazard_D* to different mux select. For example, if the control unit thinks there is a hazard appearing, it will send out the correct *op1_byp_sel_D* signal base where the instruction wants the data to forward from.

Section 4: Testing Strategy

The testing in this lab is much more complicated than in the previous lab. The instruction that we implemented requires a unit test case for testing the functionality, stall, and bypass. To make our test more efficient, we split the test case into three different categories:

- Directed testing(Testing the functionality of the instruction)
- Comprehensive Testing
- Unit Testing

4.1 Directed testing

Reg-Reg

In the Reg-Reg (RR) test case, we test functionality by determining if register x10 holds the value of 10 upon completion of all instruction executions. The evaluation extends to the mechanisms of stall and bypass across the X, M, and W stages, drawing from the results of the first instruction. As depicted in Figure 5, the second instruction focuses on the stall and bypass features linked to the X stage of the initial instruction; the third delves into those from the W stage, while the fourth centers on mechanisms originating from the M stage of the first instruction. We also test the rs2 dependence and forwarding as well.

We also test the extreme case for Reg-Reg, for example having the:

- largest 32-bit data add the largest 32-bit data.
- smallest 32-bit data add the smallest 32-bit data.
- largest 32-bit data add the smallest 32-bit data.
- Shift the max step we can shift for the shift instruction
- Shift the negitive step we can shift for the shift instruction

Reg-Imm

For the R-I test case, implement a similar test method as the RR test case. We also test the rs1 and rs2 dependence and forwarding. Also, add a negative value for the test case test the stall, and bypass it. Sine most of the immediate number for Reg-Imm is limited between 1 <= X <= 12, so the maximum number is limits to 0x0000 0fff.

Memory

In the memory test case, we initiate by allocating a specific memory space and populating designated memory addresses with predetermined values. Through the process of reading from and writing to these allocated memory locations, we can effectively assess whether the memory interactions align with our expectations.

Jump

In the jump test case, our evaluation technique differs from the conventional approach of examining the waveform and verifying the PC values. We introduce an instruction, *addi x6, x6, 10*, immediately following the jump instruction. If the value in register *x6* subsequently becomes 10, this indicates a failure in the jump instruction, meaning it did not direct to the correct memory location. Conversely, if the value remains 0, it signifies a successful jump. Additionally, this outcome confirms that the processor efficiently flushed extraneous data present in the F and D stages of the register, ensuring proper instruction execution.

Branch

In the Branch test case, the evaluation methodology is similar to that of the jump instruction. However, an additional layer of scrutiny is applied to ascertain not only the destination of the branch but also whether the branch instruction indeed initiates a jump as expected.

4.2 Comprehensive Testing

We have conducted tests on the bypass mechanism for all the instructions outlined in the Reg-Reg section. During these tests, we observed instances where it was necessary to stall the processor to ensure accurate results. For instance, there are moments when we must pause operations and wait for the multiplier to generate its output before the processor can resume and continue processing subsequent instructions.

4.3 Unit Testing

ALU Test: The ALU is a combinational logic circuit, so there is no need to consider signal delays. The ALU is responsible for performing arithmetic operations, shifts, and generating signals to compare input values. We have considered various test cases, beginning with direct tests for arithmetic operations. These tests cover scenarios such as positive numbers with positive numbers, positive numbers with negative numbers, negative numbers with negative numbers, larger (or smaller) numbers with larger (or smaller) numbers, numbers with zero, and numbers with the low (middle or high) order bits masked off. We also examine cases involving sparse numbers with many zeros but few ones, dense numbers with many ones but few zeros, overflow situations, and other corner cases. For shift instructions, we test several corner cases, such as shifting by 31 bits, to ensure the correctness of the instructions. Additionally, we pay special attention to the arithmetic shift operation to ensure that the sign bit is properly copied for different cases. For signal comparisons between input values, we test these three instructions similar to how we tested arithmetic operations. Additionally, for signed signal comparisons, we focus on specific corner cases and pay extra attention to the sign bit. Likewise, we also perform random tests on the ALU.

Immediate Generate Test: The module is used to generate five types of immediate values. Similar to the ALU, it's a combinational logic and does not involve random delays. We complete the testing by selecting different corner cases based on the immediate value formats.

Dpath Test (for baseline and alternate): Our tests cover whether the newly added modules in the datapath can function properly, including muxes, registers, integer multiplication (imul), and whether the data paths for various instruction executions are correct (including I-type, R-type, jumps and branch instruction, etc.), with random delays. We begin with direct tests. For R-type instructions except for the "mul", the differences lie only in the ALU operation. Therefore, we test the "add" and "mul" instructions. We initialize the machine code with "add x4, x2, x3" to set the signals in the F stage's "imemresp_msg_data," and then verify the correctness of the outputs of all registers (e.g., op1_reg_x, op2_reg_x1, etc.) and muxes (such as op1_sel_mux_D, op2_sel_mux_D, ex_result_sel_mux_X, etc.) involved in the execution of the "add" instruction. The test for "mul" is similar, with the difference that "ex_result_sel_mux_X" selects the output signal from "imul" as the output result. For R-type instructions other than "lui" and "auipc," the differences also lie only in the ALU operation. Like R-type instructions, we initialize the machine code for "addi x4, x2, 1" to set the signals in the F stage's and then confirm the correctness of the enable signals for all registers and the select signals for muxes involved in the execution of I-type instructions. Testing for memory instructions is also similar. We initialize with the machine code

for "sw" and "lw" and then verify the correctness of the "dmem_write_data_reg_X" register and the result of the "wb_result_sel_mux_M" mux. Testing for jump instructions is also initialized in a similar manner. Here, we verify the results of the "pc_sel_mux_F" and "ex_result_sel_mux_X" muxes and the result of the "pc_reg_F" register after a PC jump occurs. The testing for branch instructions is similar to jump instructions, but we also test scenarios where a branch does not occur.

Ctrl Test(for baseline and alternate): Our tests cover whether the newly added signals in the control unit can function properly and whether the control signals generated by various instructions at different pipeline stages are correct, including bypass and stall signals, even involving random delays. We begin with tests for R-type instructions, except for the "mul" instruction, where the only difference lies in the generated "alu_fn" signal. Therefore, we test the "add" and "mul" instructions. We initialize the machine code for "add x4, x2, x3" to set the "inst_D" signal in the M stage and then verify whether the enable signals for registers (e.g., op1_reg_x, op2_reg_x1, etc.) and the select signals for muxes (such as op1_sel_mux_D, op2_sel_mux_D, ex_result_sel_mux_X, etc.) involved in the execution of R-type instructions are correct at the corresponding stage. Testing for "mul" is similar, with the difference that "ex_result_sel_mux_X" selects the output signal from "imul" as the output result. For "mul" instructions, we also test the "val/req" interface and the corresponding stall signals and consider random delays. For I-type instructions, except for "lui" and "auipc," the only difference is in the generated "alu_fn" signal. Similar to R-type instructions, we initialize the "inst_D" signal with the machine code for "addi x4, x2, 1" and verify the correctness of the enable signals for registers and the select signals for muxes involved in the execution of I-type instructions. Testing for memory instructions is similar to the above, with the additional consideration of the "val/req" interface and the corresponding stall signals, and accounting for random delays. Testing for jump instructions is initialized and tested in the same way, with a focus on testing the stall signals. For "jal" instructions, the "squash" signal is generated in the D stage, while for "jalr" instructions, the "squash" signal is generated in the X stage. Testing for branch instructions is divided into "branch taken" and "branch not taken" scenarios. In the "branch taken" scenario, the test is similar to "jalr" instructions. In the "branch taken" scenario, only the "select" signal of "pc_sel_mux_F" is different. Additionally, no "squash" signal is generated. For the alternate design, testing for bypass signals is also necessary. First, we consider simple RAW scenarios (excluding "lw"): When the "inst_rs1_D" (or "inst_rs2_D") in the D stage of the pipeline is equal to "rf_waddr_X" ("rf_waddr_M" or "rf_waddr_W") in the X (or M or W) stage, the corresponding bypass signal is active. For load-use RAW scenarios, bypass is only effective in the M stage, and stall is required in the X stage. These conditions are also tested.

Section 5: Evaluation

We use two provided benchmarks and one self-written benchmark to evaluate the performance, specifically in terms of Cycles Per Instruction (CPI), across three different processor designs: MultiCycle, Base, and Alternative. Following the evaluation, it was observed that all benchmarks were successfully executed, with the Alternative design showcasing the most impressive performance through the lowest CPI. This was closely followed by the Base design, while the MultiCycle design lagged behind.

Section 5.1 CPI

In order to accurately determine the CPI for each benchmark, it is imperative to gather comprehensive data on both the total number of instructions executed and the total number of cycles taken:

$$CPI = \frac{Total\ Cycle\ Count}{Total\ Instruction\ Count}$$

The total instruction count for each benchmarks is calculated as follows:

Ubench_vvadd_opt:	5+ 21 * 25 + 7	= 537
Ubench_vvadd_unopt:	5 + 9* 100 + 7	= 906
Matrix :	9+4+(1+(1+13*2+6)*2+9)*2+4+4	= 173

Section 5.2 Matrix benchmarks

Matrix benchmarks employ a two-by-two matrix multiplication algorithm. We have allocated memory space to store the data values (1, 2, 3, 4) for both matrix 1 and matrix 2. Following the computation, we write the resultant matrix directly into the memory space immediately succeeding that of matrix 2.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 7 & 10 \\ 15 & 22 \end{pmatrix}$$

Section 5.3 Table of CPI Result

Benchmark	Design	Inst count	Cycle count	CPI	Time of cycle	Time
vvadd_opt (max_delay = 0)	FL Base Alt	537 537 537	2505 651 651	4.66 1.21 1.21	N/A 25τ 31τ	N/A 16.3kτ 20.2kτ
vvadd_unopt (max_delay = 0)	FL Base Alt	906 906 906	4005 2076 1276	4.42 2.29 1.41	N/A 25τ 31τ	N/A 52kτ 39.6kτ

Matrix (max_delay = 0)	FL Base Alt	173 173 173	644 382 307	3.72 2.21 1.77	N/A 25τ 31τ	N/A 9.6kτ 9.5kτ
vvadd_opt (max_delay = 5)	FL Base Alt	537 537 537	3338 1322 1322	6.21 2.46 2.46	N/A 25τ 31τ	N/A 33.3kτ 41kτ
vvadd_unopt (max_delay = 5)	FL Base Alt	906 906 906	5236 2883 2263	5.78 3.18 2.50	N/A 25τ 31τ	N/A 72.1τ 70.2kτ
Matrix (max_delay = 5)	FL Base Alt	173 173 173	802 463 432	4.63 2.67 2.50	N/A 25τ 31τ	N/A 11.6kτ 13.4kτ

Table1.Quantitatively Evaluation Results

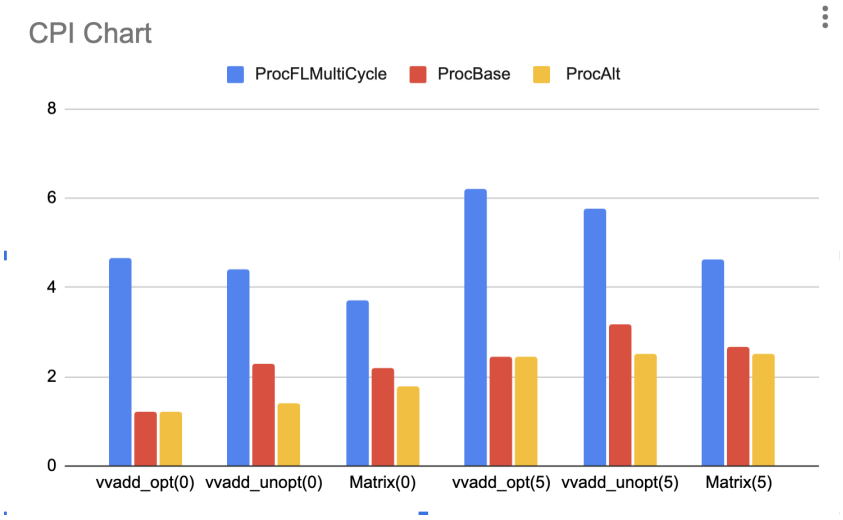


Figure 5. CPI Chart For All Benchmarks

Section 5.4 Analysis

When max_delay is 0, ProcAlt and ProcBase show highly efficient performance in the Ubench_vvadd_opt benchmark, both yielding a CPI of 1.21, whereas ProcFLMultiCycle lags with a CPI of 4.66. This pattern holds in the Ubench_vvadd_unopt and Matrix benchmarks, with ProcAlt consistently outperforming the others, showcasing its efficiency and optimization for different types of computational loads. However, when max_delay is increased to 5, all processor configurations experience a decreasing in performance, but the impact is most pronounced on ProcFLMultiCycle, which displays the highest CPIs across all benchmarks. Despite this, ProcAlt demonstrates resilience, maintaining a great performance under varied conditions. Overall, while an increase in max_delay generally results in performance drops, ProcAlt consistently shows robustness, outshining ProcBase and significantly outperforming ProcFLMultiCycle across all benchmarks. The multicycle approach requires 5 clock cycles for each instruction, regardless of the instruction's simplicity. This is why it typically has the longest cycle time and highest CPI (Cycles Per Instruction). However, for both the baseline and alternative designs, we employ a pipeline structure. Although we still implement stalls to handle data, control, and address hazards in the baseline design, the CPI is considerably reduced, showcasing the efficiency of pipelining. We can anticipate that the rate of performance improvement will diminish as we progressively allocate more resources.

The design complexity in our project significantly influences resource utilization, with alternative design consuming the most resources, followed by baseline and multicycle designs. A notable resource usage jump occurs when moving from multicycle to baseline design due to their substantial differences. In terms of energy, the increased resource demand for the baseline and alternative designs elevates energy consumption per cycle due to the need for more signal outputs and data storage registers. However, this is offset by a reduction in total cycles, especially when running larger programs, showcasing these designs' efficiency for extensive computational tasks. The alternative design consistently exhibits the lowest cycle time across numerous tests.

Section 6: Conclusion

From the evaluation results, it is easy to find that the processor with bypass design has the best performance. We find that four out of six benchmark tests take less number of cycles for processors with bypass design and CPI increases from 24.8% to 62.41% depending on the specific program, although the cycle time increases by about 16% because of the longer critical paths introduced by the bypass muxes. However, the overall latency was improved because the execution time was reduced from 1.2% to 24.5%. The processor that uses stalling uses a stable and reliable way to handle data hazards, reducing design complexity and potential bugs. However, due to the existence of data hazards, the processor needs to stall the execution of instructions, which will lead to a waste of processor resources, reducing execution efficiency and resulting in reduced processor throughput. The processor that uses bypassing can reduce or eliminate stalls caused by data hazards and improve execution efficiency and throughput. However, complex hardware logic is required to detect data risks and forward data. Overall, the full bypassing strategy provides higher performance, but at the cost of increased complexity and power consumption. The stalling strategy, on the other hand, is simple and stable to implement, but has lower performance. In general, the processor that uses stalling typically spends more time executing programs with complex data dependencies because these programs are more likely to cause pipeline stall. When performing relatively simple programs, the processor that uses bypassing can reduce performance degradation because it allows more flexibility in obtaining relevant data.Choosing to use Stall or Bypass design depends on the specific application scenario and performance requirements. Different applications may require different trade-offs. The Stall design is suitable for complex programs and data dependencies, and the Bypass design is suitable for simple programs and lower latency requirements. Optimization often needs to be based on specific uses, performance requirements, and data dependencies. Different applications may require different trade-offs to achieve optimal performance and efficiency.

Section 7 Work Distribution

Yibin Xu and Yanwen Zhu have collaborated seamlessly on both the Baseline and Alternative designs. Yibin Xu took the lead in writing assembly code for RISC-V instruction testing and CPI testing, while Yanwen Zhu focused on crafting unit tests and ensuring comprehensive coverage.