Computer Architecture Report for Lab 3

資科工所 張勝迪 312551006

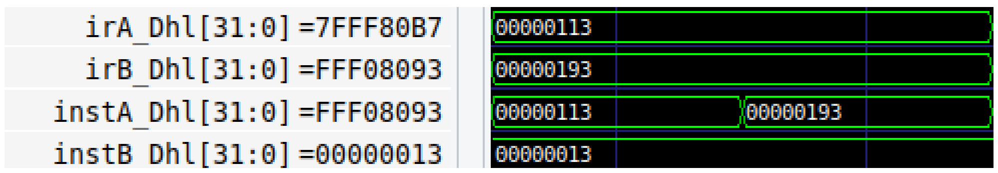
1. Introduction

This assignment involves implementing a 2-wide pipeline, which means fetching two instructions at once. The task is divided into two parts: the first is sequentially feeding one instruction into the pipeline at a time, while the second part is to load both instructions into the pipeline simultaneously whenever possible. If a stall is required due to unavoidable WAW or RAW hazards, or for other waiting conditions, only one instruction will proceed. Additionally, the assignment includes maintaining a scoreboard to replace the original bypassing mechanism.

2. Design

a. Implementation Methodology

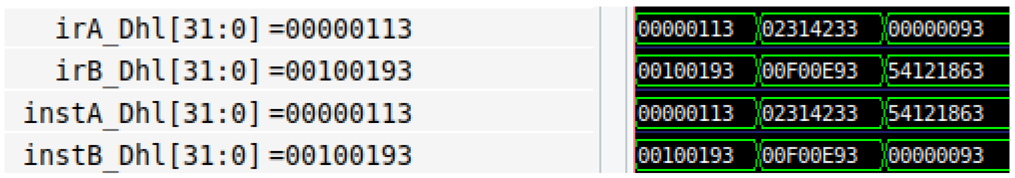
(1) In Part 1 of the design, the pipeline processes instructions sequentially, so there is no need to account for dependencies between the two fetched instructions. In this simplified approach, instructions are allowed to enter the pipeline in order, ensuring that instA_Dhl is the only active instruction while instB_Dhl is treated as a NOP (no-operation) instruction. This choice to nullify instB_Dhl reflects the single-pipeline limitation, where only one instruction is actively processed per cycle. Consequently, dependencies or hazards do not need to be managed in this phase, and we can disregard the B pipeline for the time being.



```
irA_Dhl[31:0] =7FFF80B7    00000113
irB_Dhl[31:0] =FFF08093    00000193
instA_Dhl[31:0] =FFF08093  00000113  00000193
instB_Dhl[31:0] =00000013  00000013
```

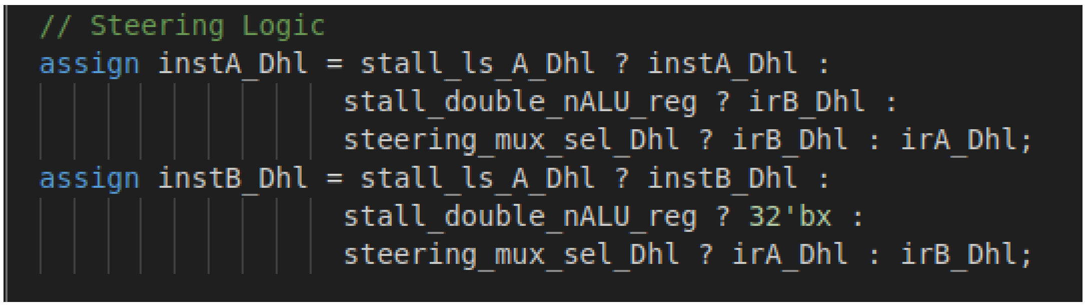
Figure 1

(2) In Part 2 of the design, we need to determine which of the two instructions fetched together should enter pipeline A and which should enter pipeline B. The result and my approach are respectively illustrated in Figure 2 and Figure 3.



```
irA_Dhl[31:0] =00000113    00000113  02314233  00000093
irB_Dhl[31:0] =00100193    00100193  00F00E93  54121863
instA_Dhl[31:0] =00000113  00000113  02314233  54121863
instB_Dhl[31:0] =00100193  00100193  00F00E93  00000093
```

Figure 2



```
// Steering Logic
assign instA_Dhl = stall_ls_A_Dhl ? instA_Dhl :
                   stall_double_nALU_reg ? irB_Dhl :
                   steering_mux_sel_Dhl ? irB_Dhl : irA_Dhl;
assign instB_Dhl = stall_ls_A_Dhl ? instB_Dhl :
                   stall_double_nALU_reg ? 32'bx :
                   steering_mux_sel_Dhl ? irA_Dhl : irB_Dhl;
```

Figure 3

We use the signal steering_mux_sel_Dhl to identify cases where the first instruction is an ALU operation and the second is a non-ALU operation. Only in this situation do we need to swap the instructions; otherwise, the default setup sends the first instruction into pipeline A and the second into pipeline B. Next, we consider cases where both instructions are non-ALU operations. Since both instructions need to enter through pipeline A in this scenario, we need to stall the second instruction, allowing the first one to proceed. Only after the first instruction enters can the second follow. This requires handling across two consecutive cycles, so I maintain a stall_double_nALU_reg register to record the previous cycle's status, allowing the second instruction to enter the pipeline on the second cycle.

Finally, if a store instruction precedes a load, data hazards may arise that aren't visible by register number alone, as offsets also play a role. To prevent address conflicts, we stall the load instruction whenever it follows a store instruction. This ensures all hazards are avoided by pausing the load in load-after-store scenarios.

b. Design Justification

In implementing the 2-wide pipeline, our primary goal was to maximize instruction throughput while effectively managing hazards between concurrently fetched instructions. To achieve this, I introduced a sophisticated control scheme that relies on separate stall signals—stall_A and stall_B—to independently control stalling for the two pipelines. This enables the detection of hazards individually for each instruction, allowing one instruction to progress while the other stalls if necessary, thus optimizing the overall pipeline utilization.

For fine-grained control over stalling, signals like stall_SB_Dhl, stall_RAW_Dhl, and stall_WAR_Dhl are utilized to address specific hazard scenarios. These signals ensure that all Read-After-Write (RAW), Write-After-Read (WAR), and Store-Load data hazards are correctly identified and managed, particularly when dealing with pending registers. By tracking the pending state of registers, the control logic identifies dependencies early and makes stalling decisions that ensure data integrity. This approach prevents the pipeline from introducing errors that could arise from data hazards between instructions at the decode stage and those currently pending for execution.

The stall_double_nALU_reg register plays a crucial role in managing cases where both fetched instructions are non-ALU operations and must enter the same pipeline. This register records the previous cycle's status, enabling the control logic to stagger entry into the pipeline over two cycles without compromising execution order. Similarly, load-after-store hazards are preemptively managed by stalling load instructions that follow store instructions when address conflicts are possible, ensuring safe memory access and preventing misaligned data reads.

3. Testing Methodology

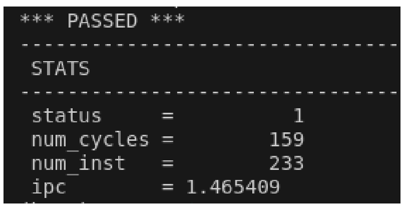
To verify data hazards and ensure correct handling of edge cases, I designed assembly tests that specifically target Write-After-Write (WAW) and Read-After-Write (RAW) scenarios. Here's a breakdown of each hazard and how the test checks for correctness:

a. **WAW Hazard**
   The instruction la x5, tdata_0 loads the address of tdata_0 into x5.
   lw x1, 4(x5) loads the value at the specified memory address into x1.
   li x1, 2 immediately overwrites x1 with the value 2.
   TEST_CHECK_EQ(x1, 2) confirms that the latest write operation to x1 has the correct value (2), validating that the processor correctly handles the overwrite without unintended behavior from the earlier lw operation. This test ensures that any WAW hazard handling logic allows the second write to execute as expected.
b. **RAW Hazard**
   la x8, tdata_0 loads the address of tdata_0 into x8.
   lw x6, 4(x8) loads the value from memory into x6.
   addi x7, x6, 1 performs an addition using x6 as an operand, with the expectation that the loaded value is correctly read and used.
   TEST_CHECK_EQ(x7, 0x00007f01) checks that x7 contains the expected result. This ensures that the RAW dependency between x6 and x7 is managed correctly by the pipeline, with x7 obtaining the correct value from x6 without any bypassing or stalling issues.

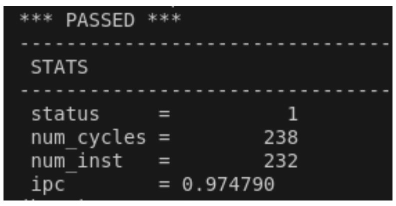
These tests are effective in validating the processor's ability to handle WAW and RAW dependencies by checking that the final register values match expectations even in the presence of data hazards.

4. Evaluation

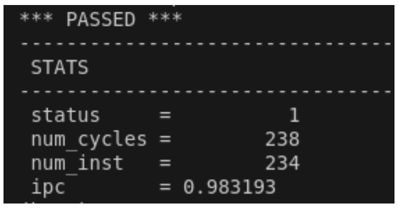
To begin with, I must mention that due to time constraints, I was unable to complete the ubmark section, so I could not provide evaluations for the four .c files. However, my code successfully passes all test cases under the make check command, so I used the results from the .S files in the test/ directory as the basis for my analysis.



```
*** PASSED ***
-------------------------------
 STATS
-------------------------------
 status     =         1
 num_cycles =       159
 num_inst   =       233
 ipc        = 1.465409
```



```
*** PASSED ***
-------------------------------
 STATS
-------------------------------
 status     =         1
 num_cycles =       238
 num_inst   =       232
 ipc        = 0.974790
```



```
*** PASSED ***
-------------------------------
 STATS
-------------------------------
 status     =         1
 num_cycles =       238
 num_inst   =       234
 ipc        = 0.983193
```

riscvssc **riscvdualfetch** **riscvlong**

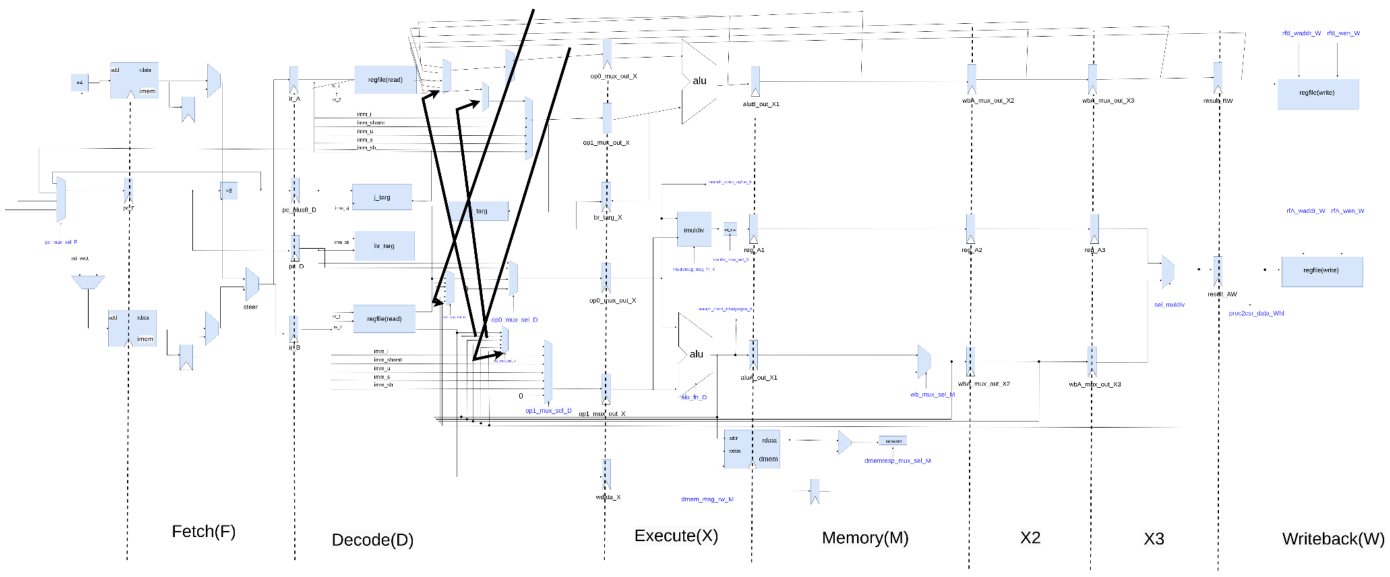
5. Discussion

Given the limitations preventing completion of the ubmark benchmarks, I continued with analysis based on the .S test results.

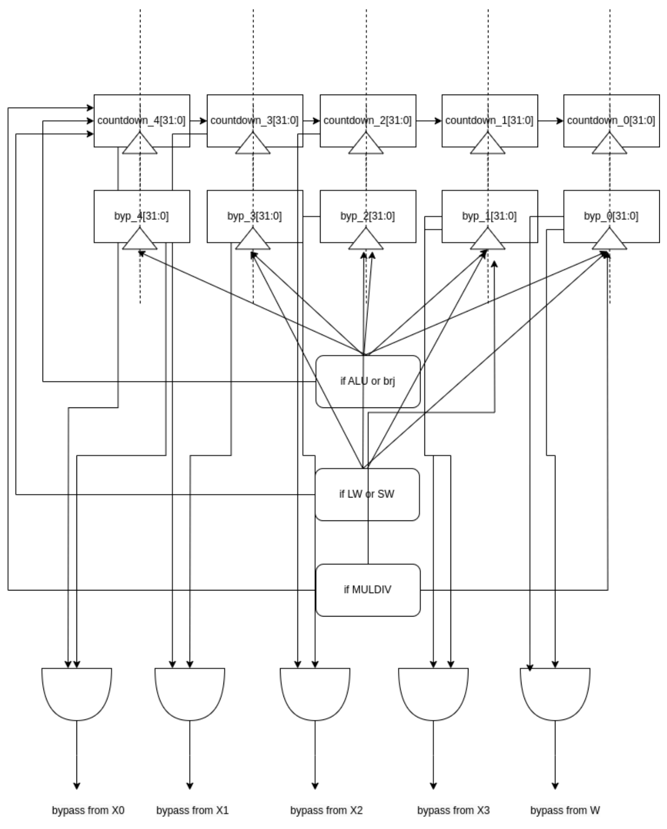
- Dual-fetch, dual-issue (riscvssc): This design fetches two instructions per cycle and can issue both instructions to separate pipelines if there are no dependencies, which maximizes IPC. This approach benefits from the ability to sustain parallelism when instruction dependencies are low, effectively handling workloads with a balanced mix of independent instructions. The high IPC of 1.46 reflects this capability, as the design has fewer stalls and can issue instructions in parallel, reducing overall cycle count.
- Dual-fetch, single-issue (riscvdualfetch): This design also fetches two instructions per cycle, but only issues one instruction at a time. While it aims to improve throughput by preparing more instructions per fetch, the single-issue constraint limits its effectiveness, especially when dependencies are present. Since it cannot issue both fetched instructions simultaneously, dependency stalls increase, which diminishes the intended advantage of fetching two instructions. This limitation can lead to an IPC of 0.97, slightly lower than riscvlong, as the stalled instructions pile up in the decode stage, causing the design to behave less efficiently under dependencies.
- Single-fetch, single-issue (riscvlong): Fetching and issuing one instruction per cycle, this design maintains simplicity and control over instruction flow, potentially benefiting from fewer structural hazards and stalls since only one instruction is processed at a time. With an IPC of 0.98, riscvlong actually performs slightly better than riscvdualfetch because it reduces dependency-induced stalls by focusing on a single instruction path, avoiding the overhead of a secondary instruction in the pipeline.

6. Figure

a. Dual-Issue I4 Processor



b. Scoreboard



countdown_4[31:0]
countdown_3[31:0]
countdown_2[31:0]
countdown_1[31:0]
countdown_0[31:0]
byp_4[31:0]
byp_3[31:0]
byp_2[31:0]
byp_1[31:0]
byp_0[31:0]
if ALU or brj
if LW or SW
if MULDIV
bypass from X0
bypass from X1
bypass from X2
bypass from X3
bypass from W