313551137 官劉翔

Introduction/Abstract: In this lab, I implemented a two-wide superscalar in-order RISC-V processor with dual fetch capability. The goal was to enhance processor performance by fetching two instructions per cycle while maintaining correct execution. Currently, I have completed the dual-fetch portion (Part 1) where the processor fetches two instructions but issues them one at a time, alternating between the two pipelines.

Design:

Riscvdualfetch:

For this lab, I focused on implementing a dual-fetch superscalar RISC-V processor that can fetch two instructions per cycle while maintaining proper instruction execution. The primary challenge was modifying the control logic to handle multiple instructions simultaneously while preserving program correctness.

Pipeline Stage Modifications

F stage:

The most significant changes were made to the fetch and decode stages to support dual instruction fetching. I modified the PC control logic to fetch two instructions per cycle, implementing this through a triple-ported memory interface that allows two instruction fetches and one data memory access simultaneously.

For the instruction queue management, I implemented dedicated queues for both fetched instructions:

```
wire imemresp0_queue_en_Fhl = (stall_Dhl && imemresp0_val) || (imemresp0_val && case4_hazard_stall);
wire imemresp0_queue_val_next_Fhl
    = (case4_hazard_stall) || (stall_Dhl && ( imemresp0_val || imemresp0_queue_val_Fhl ));
```

These queues ensure that both instructions are properly buffered when pipeline stalls occur, preventing instruction loss.

F->D Stage Management

I redesigned the F->D stage transition to handle special cases when instructions stall. When no stalls occur, both fetched instructions progress normally through the pipeline. However, when the first instruction stalls but the second doesn't, it will trigger squash_first_D_inst. Thus, I implemented a mechanism using first_F_isBubble_flag to track and manage instruction alternation which alternate let the first and second instruction from F stage into D stage This ensures that when one instruction needs to stall, we can properly shift the second instruction into position while maintaining program order.

Steering Logic

I developed a steering mechanism that handles four distinct instruction pairing scenarios:

1. Two ALU instructions: Both instructions proceed normally
2. ALU followed by Non-ALU: Instructions are swapped to optimize pipeline usage
3. Non-ALU followed by ALU: Instructions maintain their order
4. Two Non-ALU instructions: Only the first instruction proceeds while the second stalls

For Part 1 implementation, I modified this logic to effectively implement single-issue behavior while maintaining dual-fetch capability by always inserting NOPs into Pipeline B. This approach provides a foundation for implementing full dual-issue capability in Part 2.

The design also includes hazard detection and handling through case4_hazard_stall, which helps manage pipeline stalls when processing multiple non-ALU instructions. This ensures proper instruction ordering and execution while preventing potential conflicts.

#Control Logic Implementation

A critical component was the handling of instruction alternation between pipelines. I implemented this through a steering_mux_sel_Dhl control signal that determines which instruction gets processed first. For Part 1, I focused on ensuring that only one instruction progresses at a time:

```
assign instA_Dhl = irA_Dhl;
assign instB_Dhl = 32'h00000013; // NOP instruction for Pipeline B
```

I modify comprehensive stall logic to handle various hazard conditions:

General pipeline stalls with stall_X0hl and stall_X1hl

The stall logic was particularly complex as it needed to consider both fetched instructions:

```
verilogCopywire stall_0_Dhl = (stall_X0hl || stall_0_muldiv_use_Dhl || stall_0_load_use_Dhl);
wire stall_1_Dhl = (stall_X0hl || stall_1_reg);
```

Branch and Jump Handling

For branch and jump instructions, I implemented special handling to ensure correct program flow:

```
wire brj_taken_Dhl = (steering_mux_sel_Dhl) ? (inst_val_Dhl && cs1[`RISCV_INST_MSG_J_EN])
                                                               : (inst_val_Dhl && cs0[`RISCV_INST_MSG_J_EN]);
```

When a branch is taken, both fetched instructions need to be properly managed. I implemented a mechanism to kill both instructions in the fetch stage when necessary:

```
verilogCopywire squash_Fhl = (inst_val_Dhl && brj_taken_Dhl)
                      || (inst_val_X0hl && brj_taken_X0hl);
```

Riscvssc: First, I modified the is_alu signal implementation to accurately identify instruction types:

```
wire is_alu_0_Dhl = (!inst_val_Dhl) || (inst0_opcode == 7'b0010011) ||
                          (inst0_opcode == 7'b0110011 && !cs0[`RISCV_INST_MSG_MULDIV_EN]);
```

I successfully implemented comprehensive hazard detection mechanisms to handle WAW, WAR, and RAW hazards between simultaneously fetched instructions

```
wire WAW_hazard_Dhl = (inst_val_Dhl && (rd0 == rd1)) ? 1'b1 : 1'b0;
wire RAW_hazard_Dhl = (inst_val_Dhl && ((rd0 == rs11) || (rd0 == rs21)
                                             || (rd1 == rs10) || (rd1 == rs20))) ? 1'b1 : 1'b0;
```

Additionally, I optimized the bypass network by reorganizing the bypass order to prioritize more recent results. While I haven't yet implemented a formal scoreboard structure, this enhanced bypass mechanism effectively enables true dual-issue execution by properly handling instruction dependencies and data forwarding between both pipelines.

Testing Methodology

I designed a specific test file (riscv-lab3test.S) to verify two critical features of the dual-issue processor: WAW (Write-After-Write) hazard handling and non-ALU instruction execution. The testing strategy is divided into several parts:

WAW Hazard Testing: Basic WAW test: Two consecutive multiplication instructions writing to the same target register (x4)

TEST_RR_OP(mul, 5, 4, 20)   // x4 = 5 * 4

TEST_RR_OP(mul, 6, 3, 18)   // x4 = 6 * 3

This test ensures the processor correctly handles WAW hazards, with the result of the second instruction (18) being the final value stored in x4

Source-Equals-Destination WAW Testing:

Using the same register as both source and destination to increase test complexity

TEST_RR_SRCS_EQ_DEST(mul, 5, 25)   // x2 = x2 * x2 (5 * 5)

TEST_RR_SRCS_EQ_DEST(mul, 6, 36)   // x2 = x2 * x2 (6 * 6)

This scenario tests the processor's ability to handle WAW hazards with self-dependent operations

Non-ALU Instruction Testing:

Consecutive multiplication instruction test:

TEST_RR_OP(mul, 7, 6, 42)   // First multiply

TEST_RR_OP(mul, 8, 5, 40)   // Second multiply

Multiply followed by divide test:

TEST_RR_OP(mul, 10, 5, 50)   // Multiply first

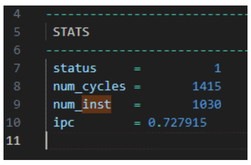
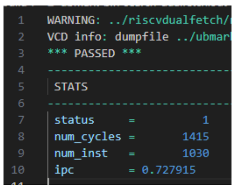
TEST_RR_OP(div, 50, 5, 10)   // Divide using previous result

These tests verify if the processor can correctly handle two consecutive non-ALU instructions, especially those requiring the same hardware resource (multiply/divide unit)

Finally, riscvdualfetch and riscvssc all pass this costume test.

#Evaluation:

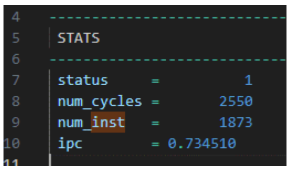
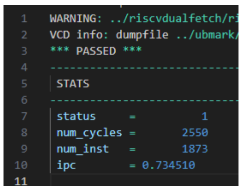
In ubmark-bin-search-byp.out

riscvlong	riscvdualfetch	riscvssc
		

```
 ----------------------------
 STATS
 ----------------------------
 status     =        1
 num_cycles =     1415
 num_inst   =     1030
 ipc        = 0.727915
```

```
WARNING: ../riscvdualfetch/
VCD info: dumpfile ../ubmar
*** PASSED ***
 ----------------------------
 STATS
 ----------------------------
 status     =        1
 num_cycles =     1415
 num_inst   =     1030
 ipc        = 0.727915
```

In ubmark-cmplx-mult-byp.out

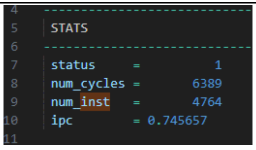
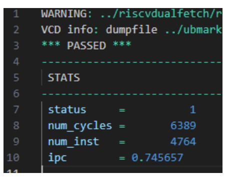
riscvlong	riscvdualfetch	riscvssc
		

```
 ----------------------------
 STATS
 ----------------------------
 status     =        1
 num_cycles =     2550
 num_inst   =     1873
 ipc        = 0.734510
```

```
WARNING: ../riscvdualfetch/ri
VCD info: dumpfile ../ubmark/
*** PASSED ***
 ----------------------------
 STATS
 ----------------------------
 status     =        1
 num_cycles =     2550
 num_inst   =     1873
 ipc        = 0.734510
```

In ubmark-masked-filter-byp.out

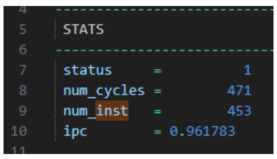
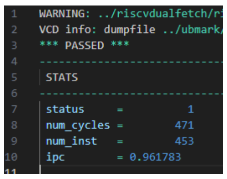
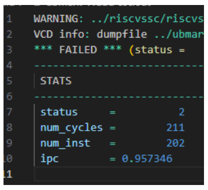
riscvlong	riscvdualfetch	riscvssc

```
------------------------------
STATS
------------------------------
status     =        1
num_cycles =     6389
num_inst   =     4764
ipc        = 0.745657
```

```
WARNING: ../riscvdualfetch/r
VCD info: dumpfile ../ubmark
*** PASSED ***
------------------------------
STATS
------------------------------
status     =        1
num_cycles =     6389
num_inst   =     4764
ipc        = 0.745657
```

In ubmark-vvadd-byp.out

riscvlong	riscvdualfetch	riscvssc
		

```
------------------------------
STATS
------------------------------
status     =        1
num_cycles =      471
num_inst   =      453
ipc        = 0.961783
```

```
WARNING: ../riscvdualfetch/ri
VCD info: dumpfile ../ubmark/
*** PASSED ***
------------------------------
STATS
------------------------------
status     =        1
num_cycles =      471
num_inst   =      453
ipc        = 0.961783
```

```
WARNING: ../riscvssc/riscvs
VCD info: dumpfile ../ubmar
*** FAILED *** (status =
------------------------------
STATS
------------------------------
status     =        2
num_cycles =      211
num_inst   =      202
ipc        = 0.957346
```

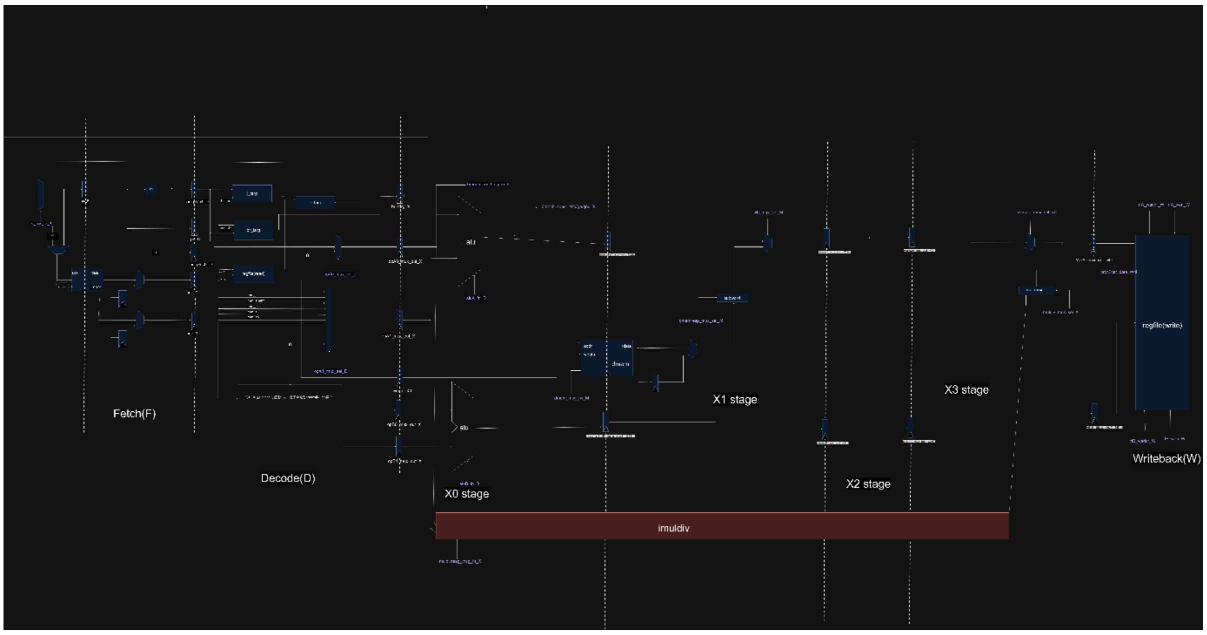
#Discussion:

Analyzing the benchmark results from our four tests (vvadd, bin-search, cmplx-mult, and masked-filter), I observed that riscvlong and riscvdualfetch show nearly identical performance metrics, with IPC around 0.74-0.77. This similarity makes sense given our current implementation where Pipeline B receives NOPs, effectively maintaining single-issue behavior despite dual-fetch capability.

For the upcoming riscvssc implementation, I expect Compute-intensive benchmarks like vvadd and masked-filter should see significant gains due to their ALU-heavy operations that can be easily paired. However, bin-search might show more modest improvements due to its branch-dependent nature limiting instruction-level parallelism.

Our current dual-fetch implementation adds hardware complexity without performance gains, but serves as groundwork for dual-issue capability. The challenge lies in managing increased control logic complexity and bypass network requirements against potential performance benefits. This directly impacts the Iron Law of Processor Performance - while our current implementation only affects cycle time negatively, full dual-issue should improve CPI despite some cycle time cost from increased complexity.

Figure: Maybe the picture is not clear enough, I also hand in the draw.io file in zip.


Fetch(F)
Decode(D)
X0 stage
X1 stage
X2 stage
X3 stage
Writeback(W)
regfile(write)
imuldiv
alu