313551137 官劉翔

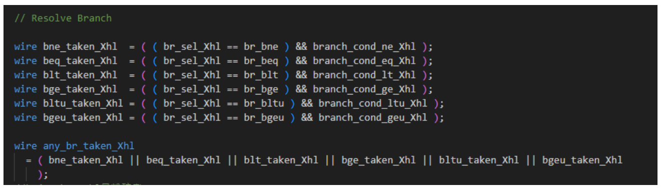
Introduction/Abstract: This lab challenges us to enhance a 5-stage pipelined RISC-V processor by extending its control unit and implementing bypassing. We'll work with the rv32i and rv32m instruction sets, gaining practical experience in processor design and optimization. By transforming a stall-based processor into one with bypassing, we'll explore key concepts in computer architecture and learn how to improve processor performance through advanced microarchitectural techniques. In the end, we have to implement the 7=stage riscvlong pipeline with new 4-stage muldiv pipeline.

Design:

Riscvstall: In this implementation, I focused on extending the control unit of the RISC-V processor. The main modifications were made in the instruction decode stage, specifically within the 'cs' (control signals) case statement.

Key additions include:

- Register-Immediate Arithmetic: Implemented ANDI, XORI, SLLI, SRLI, SRAI, SLTI, and SLTIU instructions.
- Register-Register Arithmetic: Added SUB, SLT, SLTU, SLL, SRL, SRA, AND, OR, and XOR instructions.
- Memory Operations: Expanded support for LB, LBU, LH, LHU, SB, and SH instructions.
- Control Flow: Implemented JALR and additional branch instructions (BEQ, BGE, BLTU, BGEU).This part make me impress since it have to add the condition in any_br_taken_Xhl



```
// Resolve Branch

wire bne_taken_Xhl  = ( ( br_sel_Xhl == br_bne ) && branch_cond_ne_Xhl );
wire beq_taken_Xhl  = ( ( br_sel_Xhl == br_beq ) && branch_cond_eq_Xhl );
wire blt_taken_Xhl  = ( ( br_sel_Xhl == br_blt ) && branch_cond_lt_Xhl );
wire bge_taken_Xhl  = ( ( br_sel_Xhl == br_bge ) && branch_cond_ge_Xhl );
wire bltu_taken_Xhl = ( ( br_sel_Xhl == br_bltu ) && branch_cond_ltu_Xhl );
wire bgeu_taken_Xhl = ( ( br_sel_Xhl == br_bgeu ) && branch_cond_geu_Xhl );

wire any_br_taken_Xhl
  = ( bne_taken_Xhl || beq_taken_Xhl || blt_taken_Xhl || bge_taken_Xhl || bltu_taken_Xhl || bgeu_taken_Xhl
    );
```

- Multiplication and Division: Added support for MUL, DIV, DIVU, REM, and REMU instructions.

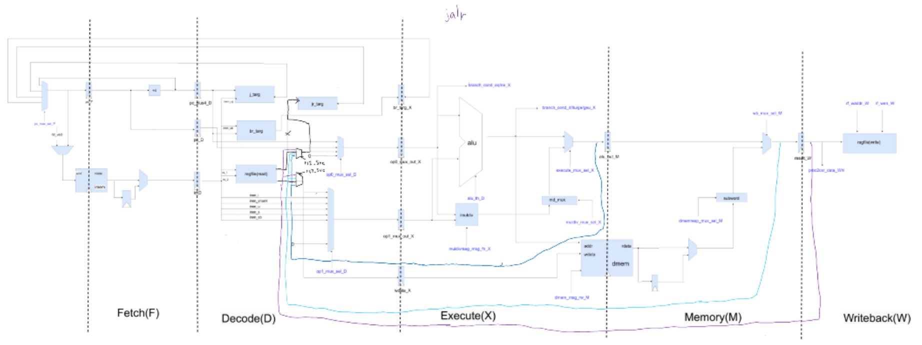
The implementation closely follows the RISC-V specification, maintaining the existing datapath structure while expanding its capabilities.

The design decisions were driven by the need to balance functionality with simplicity. Each new instruction was carefully integrated to work with the existing 5-stage pipeline, ensuring that data hazards and control hazards could still be handled effectively by the existing stalling and forwarding mechanisms.

#No significant deviations from the provided datapath were necessary, as the existing structure was sufficiently flexible to accommodate these new instructions.

Riscvbyp:

This is a designed datapath:

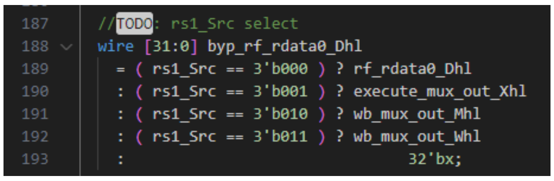


I focused on extending the RISC-V processor to incorporate bypassing mechanisms. The main modifications were made in both the control unit (riscv_CoreCtrl) and the datapath (riscv_CoreDpath) modules.

Main changes include:

- Bypass Source Selection: Added rs1_Src and rs2_Src signals in the control unit to determine the source of bypassed data for each operand.
- Load-Use Hazard Detection: Implemented logic to detect load-use hazards, which cannot be resolved by bypassing and require stalling.
- Bypassing Logic: In the datapath, added multiplexers (byp_rf_rdata0_Dhl and byp_rf_rdata1_Dhl) to select between register file data and bypassed data from different pipeline stages.

  For example:

  ```
  //TODO: rs1_Src select
  wire [31:0] byp_rf_rdata0_Dhl
    = ( rs1_Src == 3'b000 ) ? rf_rdata0_Dhl
    : ( rs1_Src == 3'b001 ) ? execute_mux_out_Xhl
    : ( rs1_Src == 3'b010 ) ? wb_mux_out_Mhl
    : ( rs1_Src == 3'b011 ) ? wb_mux_out_Whl
    :                         32'bx;
  ```

- Stall Logic Update: Modified the stall logic in the decode stage to account for load-use hazards instead of all data hazards. Besides, I deleting original stall hazard, since the load-use is enough.

These changes allow the processor to forward results from later pipeline stages to earlier stages when possible, reducing the number of stalls due to data dependencies. The load-use hazard detection ensures that the processor still stalls when necessary for load instructions.

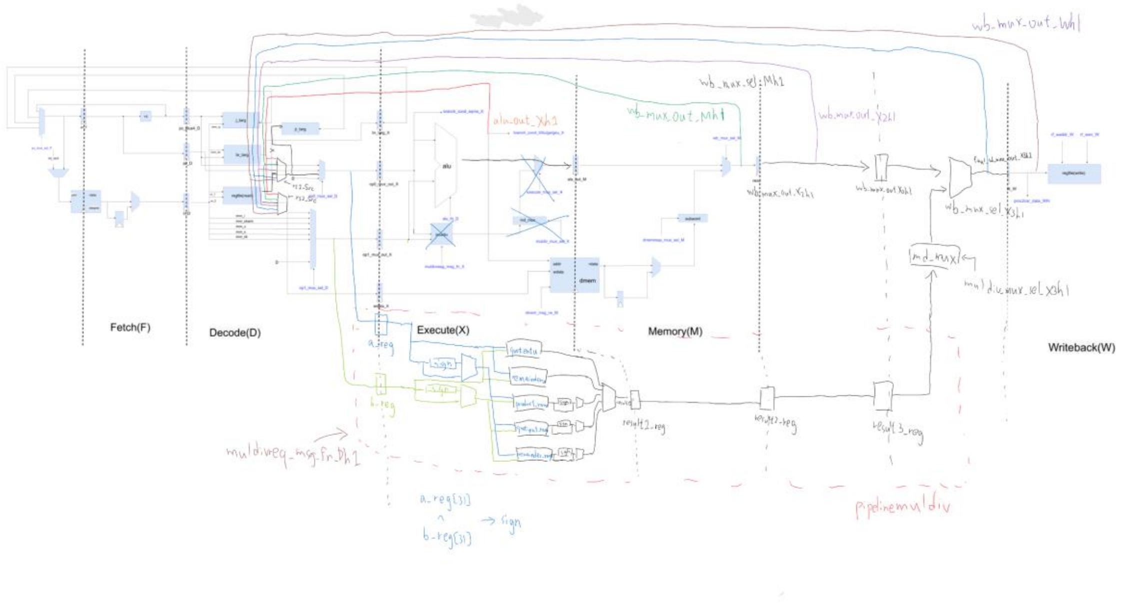
Riscvlong:

I extend the RISCV processor to incorporate a pipelined multiply/divide unit and additional pipeline stages.

Since this part is too difficult, I make this objective step by step

1. Simply add two stages
2. Add the bypass of riscvlong
3. Add riscvlong's muldiv stall
4. Move muldiv_mux and execute_mux to X3 stage
5. Add pipeline muldiv，

This is my pipeline designe



Changes include:

1. Extended Pipeline: Added two new stages (X2 and X3) to the existing 5-stage pipeline, making it a 7-stage pipeline. This aim to minimize pipeline stalls while maintaining correct instruction execution order.
2. Updated Bypassing Logic: Modified the bypassing mechanisms to account for the new pipeline stages, ensuring data hazards are properly handled across all stages.
3. Stall Logic Updates: Adjusted the stall logic to account for the new pipeline stages and the pipelined multiply/divide unit, including handling of new multiply/divide hazards.
4. Mux Relocation: Moved the muldiv_mux and execute_mux to the X3 stage, allowing for later selection of multiply/divide results.
5. Preparation for Pipelined Multiply/Divide Unit: Set up the necessary connections and control signals to integrate a pipelined multiply/divide unit in future iterations.

These changes allow the processor to handle multiply and divide operations more efficiently by overlapping their execution with other instructions, potentially improving overall performance. The extended pipeline and updated bypassing logic

Testing Methodology:

In addition to using the standard test commands:

- make check-asm-riscvstall
- make check-asm-rand-riscvstall

I also utilized individual test execution for more focused debugging:

```
./riscvstall-sim +exe=../tests/build/vmh/riscv-addi.vmh +stats=1
```

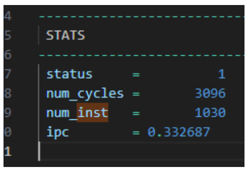
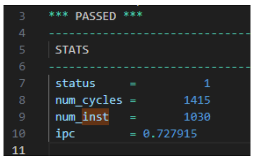
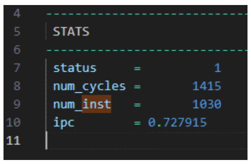
This approach, combined with gtkwave for waveform analysis, was particularly useful in identifying and correcting issues, such as problems with the multiplier in the rand-riscvstall tests.

For riscvbyp, when encountering errors like "RTL-ERROR (time = 18305) riscv_sim.mem.mem : x assertion failed : memreq0_val", I systematically deconstructed the implementation to isolate the issue, which in this case was related to missing inst_val checks.

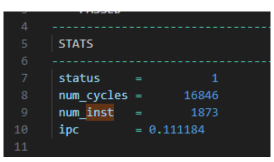
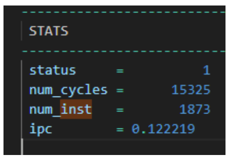
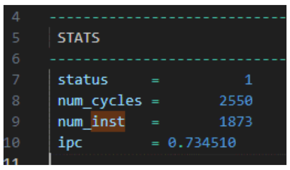
In both rand-riscvlong and rand-riscvbyp tests, gtkwave was instrumental in tracing signal propagation and identifying connection problems in the pipeline.

Evaluation:

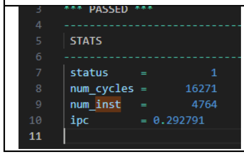
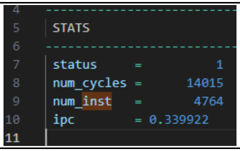
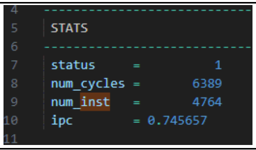
In ubmark-bin-search-byp.out

riscvstall	riscvbyp	riscvlong
 `STATS` `status = 1` `num_cycles = 3096` `num_inst = 1030` `ipc = 0.332687`	 `*** PASSED ***` `STATS` `status = 1` `num_cycles = 1415` `num_inst = 1030` `ipc = 0.727915`	 `STATS` `status = 1` `num_cycles = 1415` `num_inst = 1030` `ipc = 0.727915`

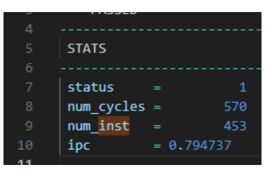
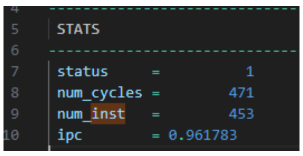
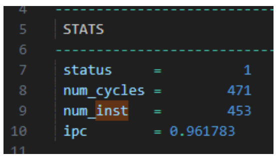
In ubmark-cmplx-mult-byp.out

riscvstall	riscvbyp	riscvlong
 `STATS` `status = 1` `num_cycles = 16846` `num_inst = 1873` `ipc = 0.111184`	 `STATS` `status = 1` `num_cycles = 15325` `num_inst = 1873` `ipc = 0.122219`	 `STATS` `status = 1` `num_cycles = 2550` `num_inst = 1873` `ipc = 0.734510`

In ubmark-masked-filter-byp.out

riscvstall	riscvbyp	riscvlong
 `*** PASSED ***` `STATS` `status = 1` `num_cycles = 16271` `num_inst = 4764` `ipc = 0.292791`	 `STATS` `status = 1` `num_cycles = 14015` `num_inst = 4764` `ipc = 0.339922`	 `STATS` `status = 1` `num_cycles = 6389` `num_inst = 4764` `ipc = 0.745657`

In ubmark-vvadd-byp.out

riscvstall	riscvbyp	riscvlong
 `STATS` `status = 1` `num_cycles = 570` `num_inst = 453` `ipc = 0.794737`	 `STATS` `status = 1` `num_cycles = 471` `num_inst = 453` `ipc = 0.961783`	 `STATS` `status = 1` `num_cycles = 471` `num_inst = 453` `ipc = 0.961783`

Discussion: The riscvbyp consistently outperforms riscvstall across all benchmarks, demonstrating the effectiveness of bypassing in reducing pipeline stalls. The riscvlong shows varied improvements: it matches riscvbyp's performance in simpler tasks like binary search and vector addition, but significantly outperforms both in complex multiplication and masked filtering. This indicates that the extended pipeline and specialized units in riscvlong are particularly beneficial for math-intensive and memory-intensive operations. Bypassing reduces CPI by minimizing pipeline stalls, but may slightly increase clock period due to added logic. The "Instruction Count" remains unchanged, while "Clock Period" might marginally increase.