Lab2 Report

111550076 楊子睬

October 6, 2024

1 Introduction

This report discusses the implementation and optimization of several components in a RISC-V pipeline, focusing on the enhancement of the control unit, the integration of bypassing mechanisms, and the inclusion of a pipelined muldiv unit. Key objectives included ensuring correct data hazard handling, improving instructions per cycle (IPC) through bypassing, and optimizing complex operations like multiplication and division by adding multi-stage pipeline support. The testing methodology involved assembly-level test cases for various bypass scenarios, while further extensions included adding instructions such as MULH, MULHU, and MULHSU, ensuring their correctness and performance under different operational conditions. The overall aim was to streamline pipeline performance, reduce stalls, and handle data dependencies more effectively.

2 Design

2.1 Objective 1: Enhancing the Control Unit

2.1.1 R-type/immediate instruction

I modified the Instruction Decode section in control unit. For register-register R-type instructions, `op0` and `op1` correspond to `rs1` and `rs2`. Then, select the ALU or muldiv function, and write back to `rd`. For immediate-based instructions, `op1` is the immediate value, which may be either I-type or U-type, the remaining setup follows the same pattern as the R-type instructions.

2.1.2 memory instruction

Regarding memory instructions, `op1` is also the immediate value. In the case of load instructions, the memory request is `ld`, and there will be a write-back. For store instructions, the memory request is `st`, but there is no write-back. Additionally, we need to adjust the data length with word, half word, byte and unsigned operation.

2.1.3 jump and branch instruction

Finally, for jump and branch instructions, the jump section selects the type for the PC mux. For `jal`, it selects `PC + immediate value`, and for `jalr`, it selects the value of `rs + immediate value`. Additionally, in the `jalr` instruction, `PC + 4` is written back to `rd`. As for branches, the branch type and ALU operation are selected. For `bne` and `beq`, `xor` is used, while other branch types use `sub` since they require different comparison logic. However, the branch taken logic was incomplete, so I spent some time writing the logic to handle the taken branches properly.

Overall, the task involves wiring things up according to the rules specified in `riscv-isa.txt` while developing a deeper understanding of the core control and datapath to implement the appropriate connections. I will spend more time discussing the remaining two parts.

2.2 Objective 2: Implementing Bypassing

2.2.1 bypass/stall control signal

In the control unit, I first modified the write enable signal to become bypass and stall signals based on the textbook description. The `bypass_wen` is only used when the instruction is an ALU or muldiv operation, and it is placed in the bypass operation select signal's write enable (`wen`). This is because bypassing can only occur when the `rd` result is ready during the execution stage.

The conditions for bypassing are derived from the three original stall data hazards: execution, memory, and write-back hazards. I also designed `localparam` to make the maintenance easier. Finally, these two bypass signals are sent to the data path for `op0` and `op1` selection.

```
wire rf_wen_stall_Xhl = is_load_Xhl  ? rf_wen_Xhl : 1'd0;
wire [2:0] byp_op0_mux_sel_Dhl = inst_val_Dhl &&
      ( rs1_en_Dhl && inst_val_Xhl && rf_wen_bypass_Xhl
      && ( rs1_addr_Dhl == rf_waddr_Xhl ) && ( rf_waddr_Xhl != 5'd0 ) )
      ? byp_exe :inst_val_Dhl && ( rs1_en_Dhl && inst_val_Mhl && rf_wen_Mhl
      && ( rs1_addr_Dhl == rf_waddr_Mhl )&& ( rf_waddr_Mhl != 5'd0 ) )
        ? byp_mem :inst_val_Dhl &&( rs1_en_Dhl && inst_val_Whl && rf_wen_Whl
      && ( rs1_addr_Dhl == rf_waddr_Whl )&& ( rf_waddr_Whl != 5'd0 ) )
      ? byp_wb : byp_none;
// byp_op1_mux_sel_Dhl is similar
```

The `stall_wen` is used when the instruction is a load operation, and it is placed within the original data hazard stall `wen` check. If the previous instruction is a load, we must wait until the memory stage is completed to retrieve the data from the memory, so a one-cycle stall is required. In the instruction decode part, I added `is_load` and `is_alu` signals in the decode stage, which are passed through the pipeline to the execution stage.

```
wire rf_wen_stall_Xhl = is_load_Xhl  ? rf_wen_Xhl : 1'd0;
wire stall_hazard_Dhl = inst_val_Dhl && (
   ( rs1_en_Dhl && inst_val_Xhl && rf_wen_stall_Xhl
   && ( rs1_addr_Dhl == rf_waddr_Xhl ) && ( rf_waddr_Xhl != 5'd0 ) )
   || ( rs2_en_Dhl && inst_val_Xhl && rf_wen_stall_Xhl
   && ( rs2_addr_Dhl == rf_waddr_Xhl )&& ( rf_waddr_Xhl != 5'd0 ) ));
```

2.2.2 bypass datapath

For the data path, I handled the basic execution `op0` and `op1` sources, the store instruction's write data, and the `jalr`'s `rs1` data, as `rs1` or `rs2` may result in data hazards. Starting with the execution source, I first checked if `rs1` or `rs2` should be selected as the input. If so, the bypass signal is used to determine the source. During execution hazards, `execute_mux_out_Xhl` is used as the `rs`; for memory and write-back hazards, `wb_mux_out_Mhl` and `wb_mux_out_Whl` are used as the `rs`. If the original mux is not rs of regfile, the bypass is ignored, and the original multiplication signal is used for selection. Fig. 1 show the fully bypass datapath.

For the store instruction's write data, I modified the `wdata_Dhl` selection. If a bypass occurs, the same sources as above are used; otherwise, the default source `rf_rdata1_Dhl` is used. Similarly, the `jalr`'s jump register, `jump_reg_rf_rdata0_Dhl`, uses the bypass sources when available, and defaults to `rf_rdata0_Dhl` when there is no bypass.Then I completes all the required bypass logic in the data path.

The design principle was to follow the original coding style, adhering to the format for signal naming and port connections, while ensuring all instruction behaviors are handled correctly, leaving no data hazard paths unchecked. This is a fundamental yet very challenging task.

2.3 Objective 3: Integrating a Pipelined MulDiv Unit

2.3.1 add two dummy stage

To extend the core' s pipeline, I added two dummy stages (M2, M3) between the memory and write-back stages. The control signals and data originally passed to the write-back stage are now passed through two additional stages using an always block. The detailed architecture can be seen in Fig. 2 . Due to the additional stages, the bypass and stall conditions also needed to be adjusted. Specifically, I introduced new bypass routes from M2 to the execution stage and from M3 to the execution stage. Every point requiring bypass logic had to be updated accordingly.

Since the assignment requires that the pipeline muldiv unit takes four stages to produce a result, I extended the is_md signal from the decode stage all the way to M2. When the execution stage needs to use the rd from the previous execution, memory, or memory2 stages, a stall is introduced if those stages are handling a muldiv instruction. The stall persists until the M3 stage, as that is when the muldiv unit outputs its value. However, in other cases, M1 and M2 can still perform bypassing. Therefore, I divided the write enable logic into two cases and integrated them separately into the bypass and stall condition checks.

2.3.2 integrate pipelined muldiv unit

Next, the pipelined muldiv unit was integrated into the system. To synchronize the first stage of the muldiv unit with the execution stage, the function signal and rs data from the decode stage were passed in. The `muldivreq_val` was set to `muldivreq_val_Dhl && inst_val_Dhl`, ensuring the muldiv function is triggered only for md instructions. The `muldivreq_rdy` signal was connected to `stall_Dhl` to prevent stalls until the muldiv unit is ready, although this setting is conservative as the muldiv unit typically remains ready.

The `stall_Xhl` and `stall_Mhl` signals synchronize the first two stages of the pipeline with the muldiv unit, avoiding write-back conflicts. The M2 and M3 dummy stages do not generate stalls, so no additional design was needed. The output is generated in the M3 stage.

The `muidiv_mux_sel` signal was passed through the pipeline to control whether the M3 stage outputs the high or low 32 bits. I set `muldivresp_rdy` to 1, ensuring no stalls in M3. The write-back mux checks `muldivresp_val` and selects the muldiv output if valid, completing object 3.

```
// rdy/val signal
assign muldivreq_val = muldivreq_val_Dhl && inst_val_Dhl;
wire stall_muldiv_Dhl = ( muldivreq_val_Dhl && inst_val_Dhl && !muldivreq_rdy );
assign muldivresp_rdy = 1'b1;
```

2.3.3 design principle

The key design elements are the addition of pipeline stages and the integration of a 4-stage muldiv unit. First, minimal signals originally passed to the write-back stage were supplemented with a basic `stage_inst_val` for bubble detection, and the bypass and stall conditions were adjusted for the added stages, following consistent naming and coding conventions.

Second, the 4-stage muldiv unit was synchronized with the X, M, M2, and M3 stages. This required careful analysis of stalls, signal connections, and timing logic to avoid write-back conflicts. While the dummy stages have no real behavior, handling stall and val/rdy interfaces was simplified. Future stage merging with more complex behaviors may require additional adjustments.

3 Testing Methodology

I added an assembly test (riscv-byp.S) to specifically test various bypass scenarios. This includes register-to-register bypass, immediate instruction bypass, a single stall during load-use situations followed by bypass, and bypass when store word write data encounters a hazard. I made sure that the subsequent instructions continuously use the previous instruction' s rd data to verify that all three bypass paths are correctly connected without causing stalls. When sw needs to store the result of a previous register calculation to memory, it can do so without stalling. Additionally, the original test cases already covered bypass scenarios, and all passed successfully. Finally, the test passed the complex bmark operations, validating that all bypass paths are functional. I also added extra test cases for mulh, mulhu, and mulhsu, which will be explained in the extension section. Below are part of my test cases.

```
// imm tests
li x1, 1
li x2, 2
addi x3, x2, 20
add  x4, x3, x2
slli x2, x2, 1
mul  x5, x4, x2
addi x6, x5, 12
TEST_CHECK_EQ(x6, 108)
```

```
//sw data tests
li x2, 1
li x3, 2
add x4, x3, x2
sw x4, 0(x1)
lw x6, 0(x1)
 TEST_CHECK_EQ( x6,3 )
```

Table 1: instructions per cycle on ubmark

ubmark	riscvstall		riscvbyp		riscvlong	
	cycles	ipc	cycles	ipc	cycles	ipc
bin-search	3096	0.3327	1415	0.7279	1415	0.7279
cmplx-mult	16846	0.1111	15325	0.1222	2550	0.7345
masked-filter	16271	0.2928	14015	0.3399	6389	0.7457
vvadd	570	0.7947	471	0.9618	471	0.9618

4 Evaluation (Table 1)

5 Discussion

From Table 1, we can see that the optimization effect of bypassing is most significant in bin-search. Since binary search relies heavily on the result of the previous loop iteration to make the next decision, there is strong data dependency. When bypassing is introduced in riscvstall, the instructions per cycle (IPC) more than doubled. However, in vvadd, the improvement is less noticeable, around 17%, because it mainly performs vector operations, accessing independent memory locations with little data dependency.

Bypassing has the smallest impact when there are many multiplication operations, as seen in cmplx-mult, which involves extensive computation of real and imaginary parts. Here, the muldiv unit in the execution stage typically takes 33 cycles, causing frequent stalls. Consequently, the impact of bypassing is limited because the bottleneck is the muldiv unit's cycle count rather than data forwarding. Similarly, in masked-filter, which implements a convolution-based image processing filter, the use of numerous multiplication operations also limits the benefit of bypassing. However, when we pipelined the multiplication unit into 4 stages and allowed parallel execution, the IPC improved significantly by six to seven times.

Thus, we can conclude that bypassing is less effective in functions with minimal data dependency and that to improve performance in multiplication-intensive operations, the design of the muldiv unit must be optimized to reduce the number of cycles.

6 Extensions

This section adds the MULH, MULHU, and MULHSU instructions, similar to object-1. I first updated the inst-msg for these instructions, then modified the control unit' s instruction decode to include the mulu and mulsu functions. The md bit select was adjusted to the upper 32 bits, and the muldiv unit now computes both mulu and mulsu, with output selected via a function code mux.

For testing, I generated random large numbers using Python and created assembly test cases to verify different operations and bypass scenarios. Below is the main adjust in pipelined muldiv unit in Verilog.

```
wire [63:0] productu = a_reg * b_reg;
wire [63:0] productsu_raw = a_unsign * b_reg;
wire [63:0] productsu = ( a_reg[31] == 1'b1 ) ? ( ~productsu_raw + 1'b1 ): productsu_raw;
```

7 Figures

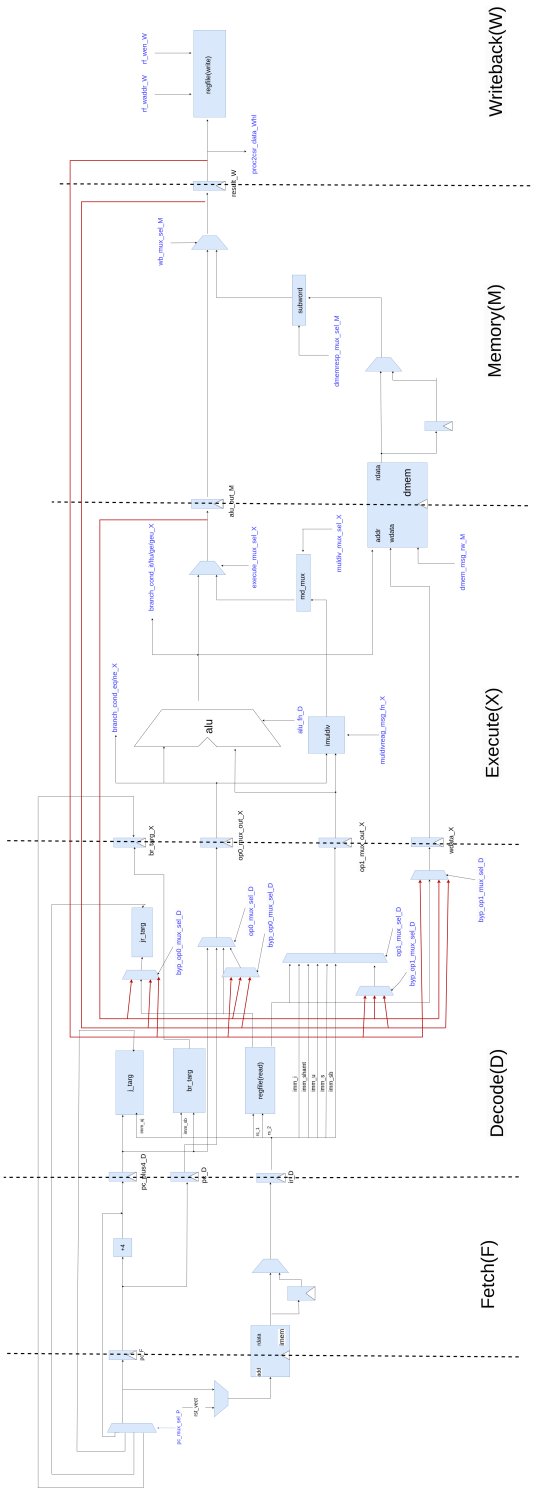

Figure 1: 5 stage bypass datapath

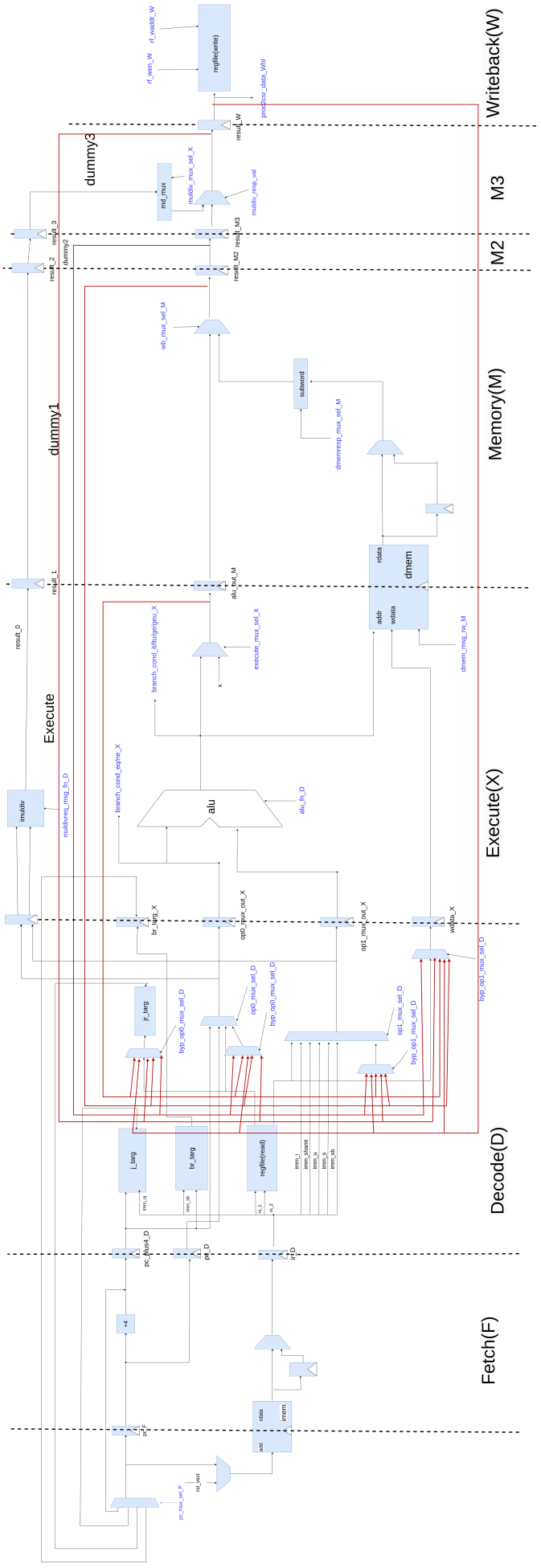

Figure 2: 7 stage bypass datapath