MP4 Checkpoint 1 Deliverables

Progress Report, Roadmap, Paper Designs

November 11th, 2020

Alex Vetsavong
Peter Kircher
Mohan Li

Responsibilities: What did people do for this checkpoint?

Datapath:
Mohan started and laid out all the functional units, pipeline registers, and corrected some syntax errors in the included packages from rv32i_types.sv introduced when implementing a rv32i_ctrl_word struct.

Control ROM:
Alex created the Control ROM module, and extended the rv32i_types.sv and rv32i_mux_types.sv files to allow for a control word struct to be passed down the pipeline. Debugging the Control ROM module and the signals being set in the control word was also a part of his responsibility.

Testing methodology:
Peter helped in debugging and fixing compile-time and run-time errors/warnings. Alex was also the one who exposed the necessary RVFI monitor signals to the top level module and worked with Peter in order to figure out a halting condition necessary for the AG in CP1.

Changes from Design Checkpoint:
Compared to the original design, the regfilemux_sel and mem_byte_enable logic is more detailed and allows for aligned memory accesses for loads and stores. Before, these read masks and write masks were not being shifted correctly in the proposed design, which prevented correctness in the register file throughout execution.

Roadmap: Who's doing what?

Alex:
Will be working on hazard detection and branch prediction, moving forward. For now, only static branch-not-taken prediction will be handled, and the implementation of the pipeline flush will be determined past the paper design due date.

Mohan:
Will be providing the design for and implementation of the data forwarding unit, and the logic associated with it.

Peter:
Will handle implementation of the arbiter logic and the use of parameter memory for the split caches for the upcoming checkpoint. This will involve handling mem_resp and stalling the pipeline or forwarding data until the memory access is done.

CP2 Paper Designs
Hazard Detection and Forwarding:

Problem Statement

For checkpoint 2, we need to implement solutions to data hazards, specifically RAW type. WAW & WAR hazards do not apply to our design.

An example of RAW hazard: Registers' values are updated in WB stage; rs1_out and rs2_out are from ID stage.

	1	2	3	4	5
Write R1	IF	ID	EX	MEM	WB
Read R1		IF	ID1	EX1	MEM
Read R1			IF	ID2	EX2
Read R1				IF	ID3

Solution

1. Without Forwarding, a simple solution to WB - ID3 in cycle 5 is: Performing register file reads @negedge, and writes @posedge. - Change regfile.sv.

2. **Hazard Detection:**
   If (rd of previous 2 operations == source registers for the current ALU/CMP operation):
   - Use registers to preserve the values of ID.rs1, ID.rs2 to EX stage for comparison.

3. **Forwarding:**
   Select the forwarded result as the ALU input rather than the value read from regfile.
   We need to forward:
   1. Alu_out from ex/mem registers - ALU / CMP instructions (higher priority than wb.alu_out)
   2. Alu_out from mem/wb registers - ALU / CMP instructions
   3. D_mem_rdata from wb stage only - Load instructions

Explanation of 3.3: Load Instructions: Stalls Are Required
LW x1, (addr)
ADD x1, x1, x1
This example reads the previous rd 1 cycle after the Load Register instruction.
However, we will not be able to get d_mem_rdata from Data Cache until the MEM stage.

We need to add hardware - a pipeline interlock, to preserve the correct execution pattern.
It needs to (1) detect a hazard (added to the pseudocode) (2) stall the pipeline for 1 cycle:

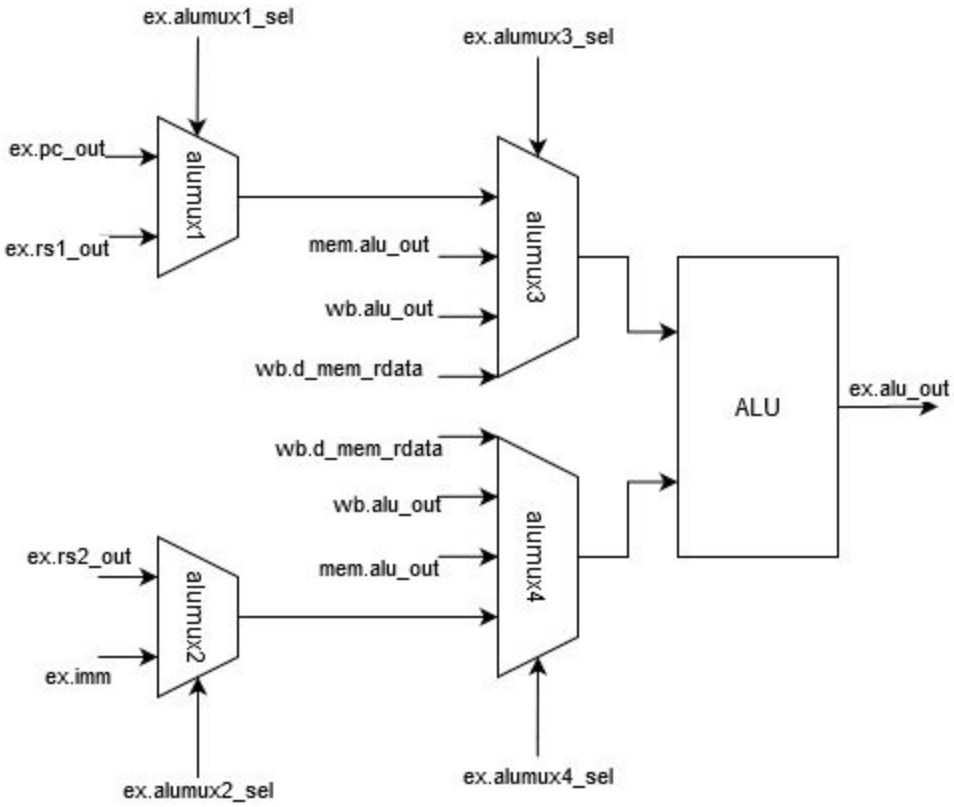
LW R1	IF	ID	EX	MEM	WB
Read R1		IF	ID	STALL	EX
Read R1			IF	STALL	ID
				STALL	IF

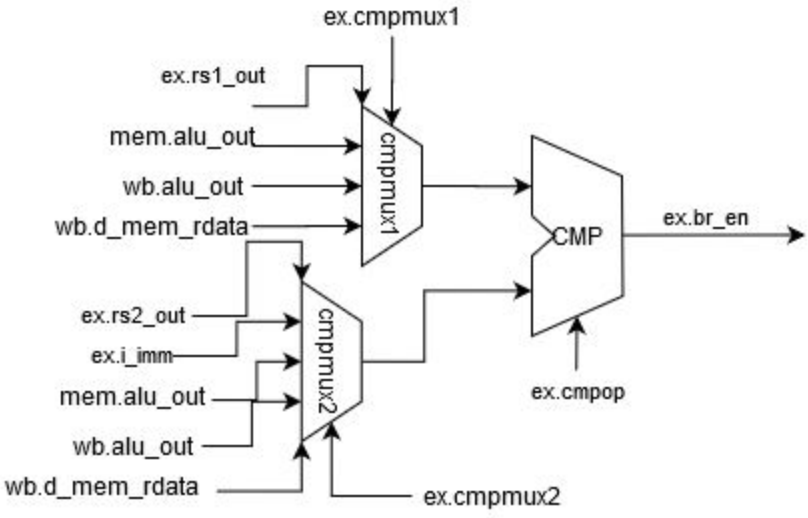
This can be done by preventing flopping the inputs to (1) IF/ID, (2) ID/EX registers. In other words, the inputs to IF and ID stages stay the same.
The EX, MEM, WB stages see an extra NOP inserted.

Combinational Logic for ALU MUXes and CMP MUXes

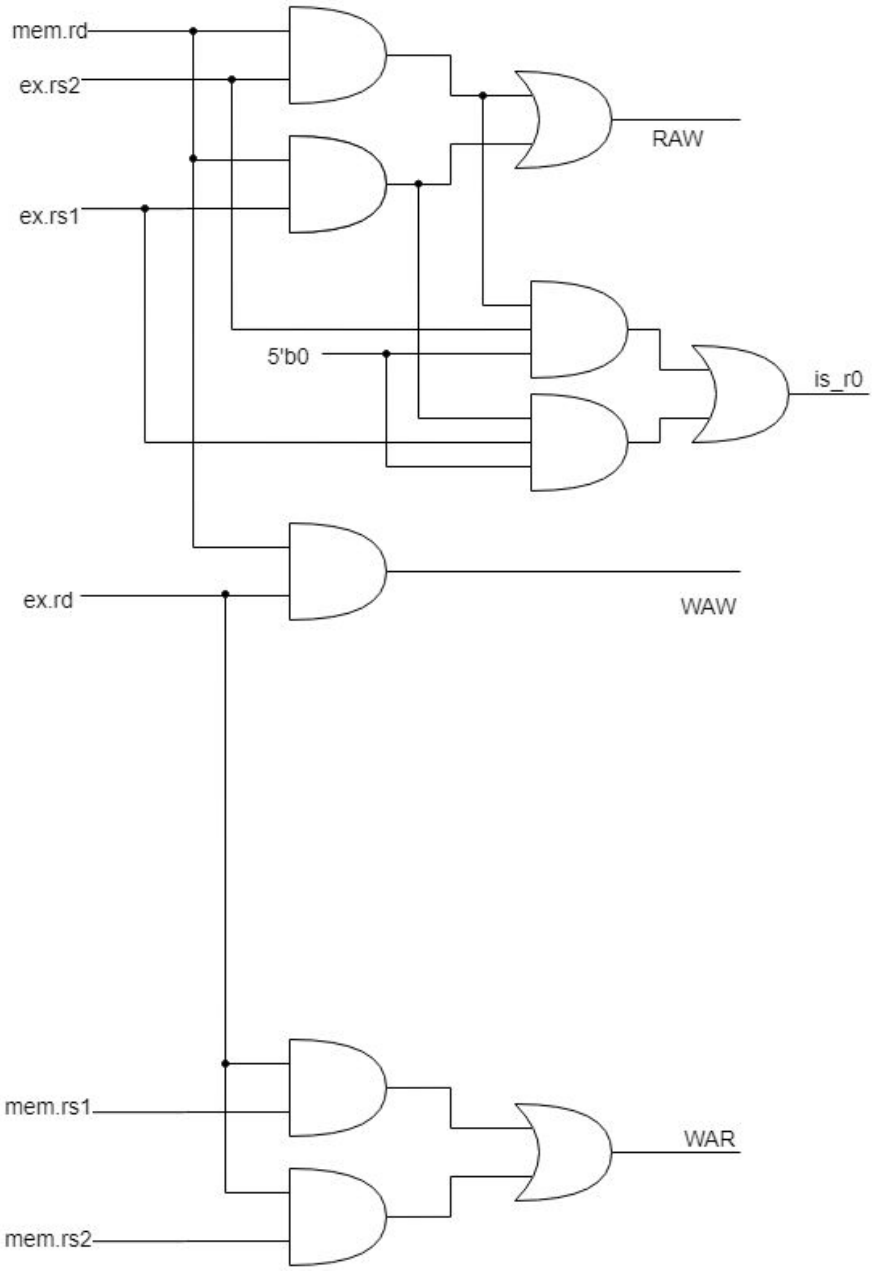
```
// Pseudocode - updated alumux1/2 logic:
If alumux1_sel == rs1_out:
    If ex.rs1 == mem.rd:
        If mem.opcode == Arithmetic Instructions
            Alumux1_out = mem.alu_out
        If mem.opcode == Load Register:
            // stalling! unreachable?
    // Must be else if, because we prioritize data from mem over data
    // from wb. This accounts for consecutive writes followed by a
    // read.
    Else If ex.rs1 == wb.rd:
        If wb.opcode == Arithmetic Instructions
            Alumux1_out = wb.alu_out
        If wb.opcode == Load Register:
            Alumux1_out = wb.d_mem_rdata
// Similar design for cmpmux 1/2
```

Design diagram for new ALU MUXes and CMP MUXes (tentative)





Tentative Design for Data Hazard Detection

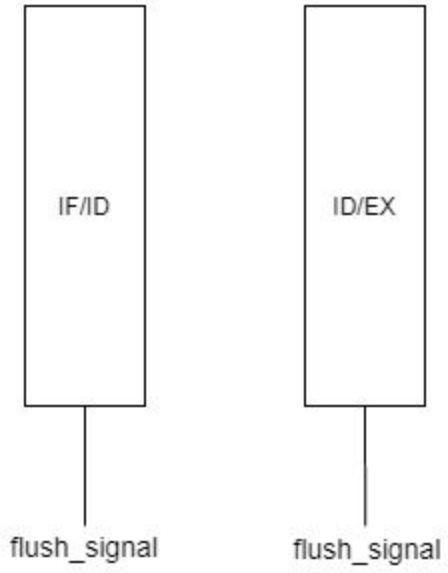


Currently, we handle RAW hazards in more detail than the others, as those hazards are alleviated by the register file being double-pumped (reads occuring on the positive edge, writes on the negative edge) and the fact that our execution stage has constant time usage for each instruction, currently.

In summary, it checks to see if the destination of the previous instruction is the same as one of the sources in the current instruction, and then compares whether those sources are Register 0.

When actually implemented, we will also need to check whether the instruction actually uses that specific register source, i.e. how immediate instructions do not use rs2.

Proposed Control Hazard Detection



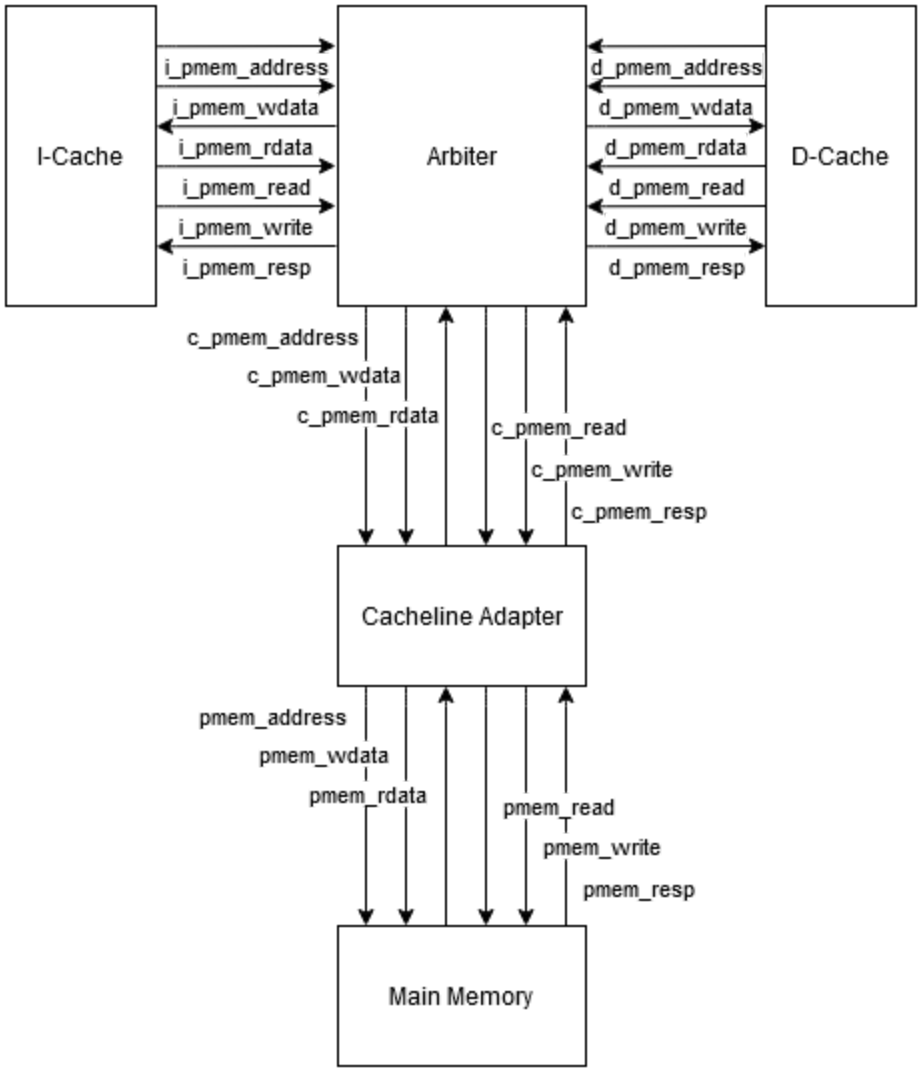
assign flush_signal = is_branch

Modify registers on flush_signal assertion to insert nop

Modify registers to take in a stall signal instead of constant high assertion

As seen in the diagram, we propose that a signal, called is_branch, that specifies whether or not an instruction takes a branch, is created. We assign a flush_signal to monitor is_branch, and we will use that to flush the necessary registers of the corresponding stages.

Arbiter:



The arbiter uses a state bit that will determine which way the cache signals are forwarded. The I-Cache and D-Cache signals will be connected (MUX-style) to the cacheline adapter. The state bit change will be on pmem_reads and pmem_writes from the caches. The I-Cache will win in the event of a conflict. The state bit can switch to the serve_d state immediately following pmem_resp (on a d_pmem_read/write). The arbiter will also register its inputs in order to lower the path of signals to memory.