CS 3410 Project 3 Design Documentation

Michael Maitland (mtm68) Jared Matthews (jm2534) Philip Ayoub (pja66)

April 18, 2020

1 Introduction

The purpose of this document is to document the design and implementation of a RISCV processor and its subcomponents (ALU, Register File, PC, pipeline registers, etc.) and the connections between them. We provide an explanation for any parts of your processor or any sub-components that are unusual or potentially confusing; documentation provides us another avenue of communicating our intended design and implementation.

In this project we extended the RISCV processor design started in Project 2 by including support for memory interaction and more complex instructions; we implemented all instructions mentioned in the last project in addition to load/stores, jumps and branches. Building off the release circuit published on Course Management System that encompassed our work up through the last project, we also made use of the provided blackbox components where appropriate, and per specification we did not describe their functionality extensively nor test these components.

We used Git throughout this project to maintain version control of our circuit and to ensure that the availability of a backup for our work, and we submitted a Git log documenting these efforts. Our group had no restrictions on how often we intended to commit, but we did so often and with the intent of providing helpful messages to the other group members; we believe our log reflects this.

The intended audience of this document is us, the creators of our circuit, so that we document our work on the project. Additionally, this document is intended for anyone reviewing our circuit so that they understand our design.

The references we used are:

- RISC V SPEC: VOLUME 1: USER LEVEL ISA
- RISC V ONE PAGE REFERENCE SHEET

In summary, this document provides a general overview of the functionality of the processor, with special attention paid to components specific to making the instructions in table B functional. We describe each stage of the pipeline with the goal of emphasizing its role in the processor and conclude with a discussion of our testing methodology.

2 Overview

The processor is a five-stage modified Harvard-architecture pipelined RISCV design implementing stalls, memory-execute forwarding, writeback-execute forwarding, and providing support for core RISCV commands. It it shown schematically in fig. 1 with the functional relationships between subcomponents emphasized.

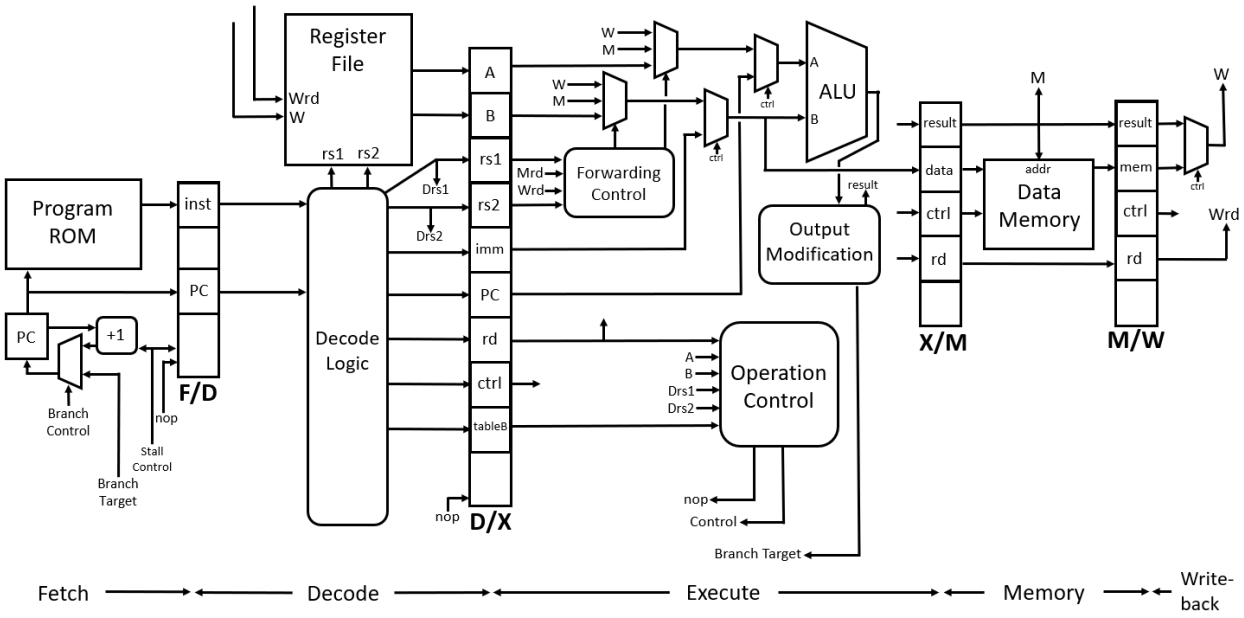

Figure 1: RISCV processor overview emphasizing the functional and logical pathways in the design. Some connections were simplified or truncated to maintain readability.

3 The Fetch Stage

3.1 Description

Fig. 2 depicts our implementation of a RISCV fetch stage, in which we make use of the provided IncrementPC template as well as some of our own designs. Specifically, we grouped the constitutive registers of the F/D pipeline register into a single subciruit. The bottom of the frame shows control logic from the execute stage; if a program calls for a branch or jump, the multiplexor (mux), incoming 32-bit address ("Branch Target", fig. 1), and mux control are sufficient to ensure this transition correctly executes.

Since taken branches and jumps from execute render successive instructions in fetch and decode invalid, the processor "zaps" those stages by injecting nops to keep continuity in the pipeline. Thus, the mux control for switching to the branch target is the same as the nop control for the F/D pipeline register. Similarly, we see that stall control in the event of load-use hazards is passed to the F/D pipeline and the program counter register, both of which would have writes disabled for stalling.

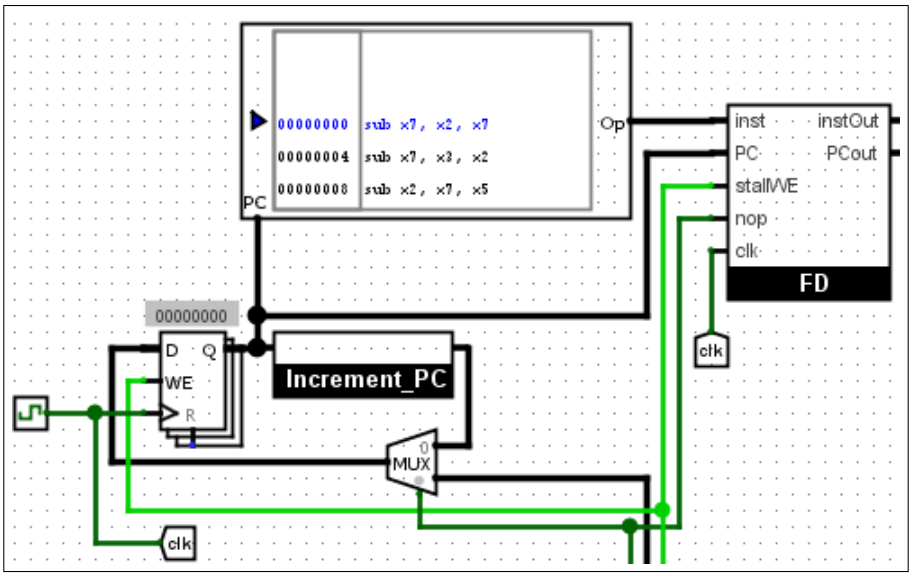

Figure 2: Fetch stage showing ROM, F/D pipeline register, incrementer, and stall and nop control

3.2 Subcircuit Diagrams

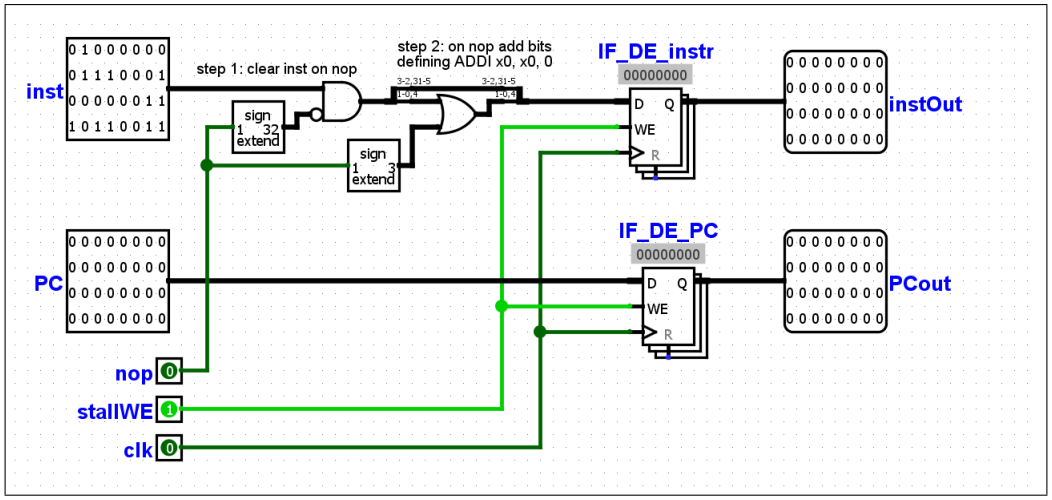

Figure 3: Fetch/Decode pipeline register and associated control circuit. The registers are write-disabled in the event of a stall. If a nop is required, the control logic injects an ADDI x0,x0,0 signal in accordance with the RISCV specification and as described by the comments in the figure.

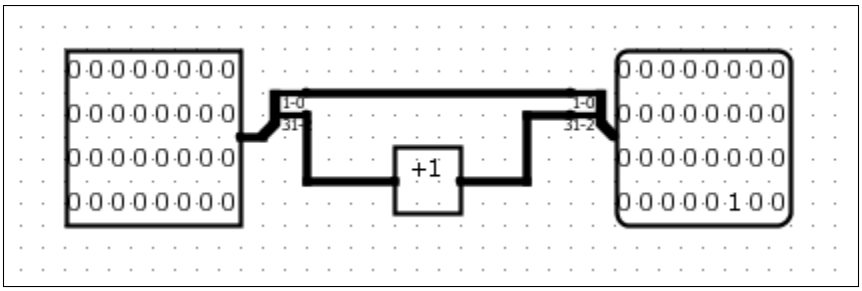

Figure 4: Provided incrementer subcircuit for the program counter.

4 The Decode Stage

4.1 Description

The following diagram depicts the top-level of the Decode Stage (fig. 6). The "Decode" subcircuit is the primary source of control logic for the processor, and its breadth of output is apparent in the size of the D/X register. Like the Fetch stage, the pipeline register is a single subcircuit with minor supplemental control logic. This logic is necessary for the injection of nops, but this differs in its implementation here compared to the hardware in F/D since only the write-enabling and branch control bits of the instruction are zeroed rather than an entire instruction inserted.

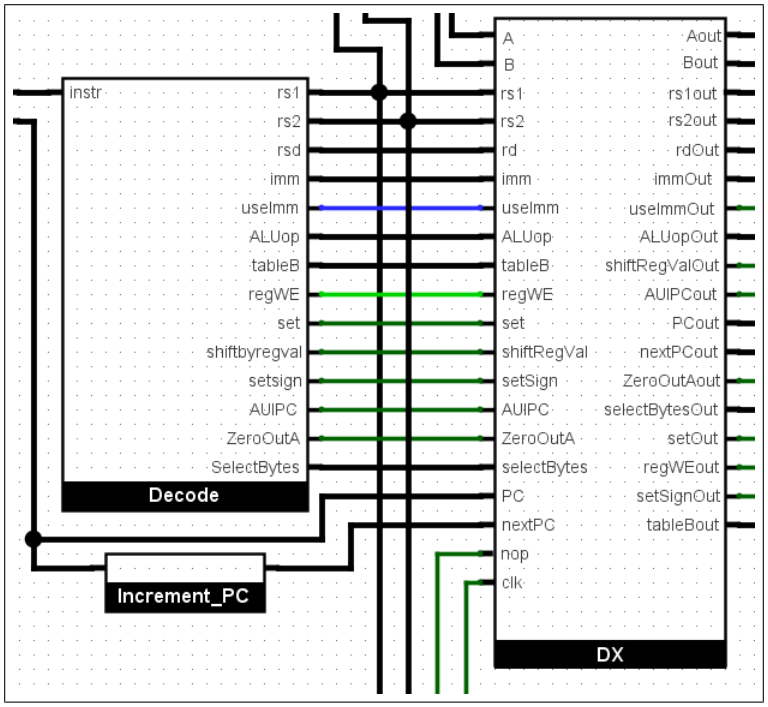

Figure 5: The Decode section of the processor showing the Decode subcircuit and D/X register. The additional program counter incrementer is integral to successful JAL and JALR command execution. As with Fetch, control signals from the execute stage are necessary to stall and inject nops.

4.2 Subcircuit Diagrams

A key aspect of the Decode subcircuit is its differentiation between table A and table B instructions; as mentioned, the architecture for the former (bitwise, shift, arithmetic, LUI, and AUIPC) operations was provided, and our design supports these as well as the more complex table B jumps, branches, and memory operations.

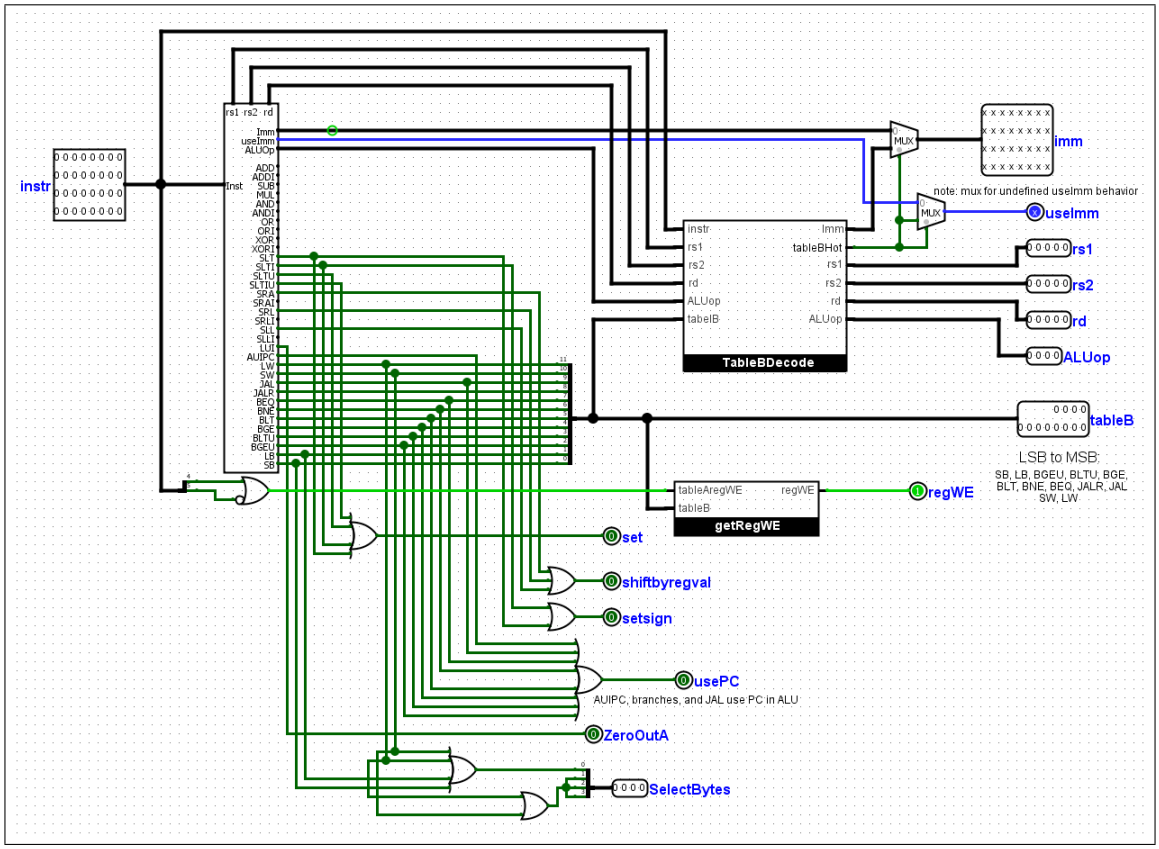

Figure 6: Decode subcircuit with included "Table A" instruction blackbox. Decode builds control signals for the rest of the processor using the output from this black box as well as its own manipulation of the instruction bits from Fetch.

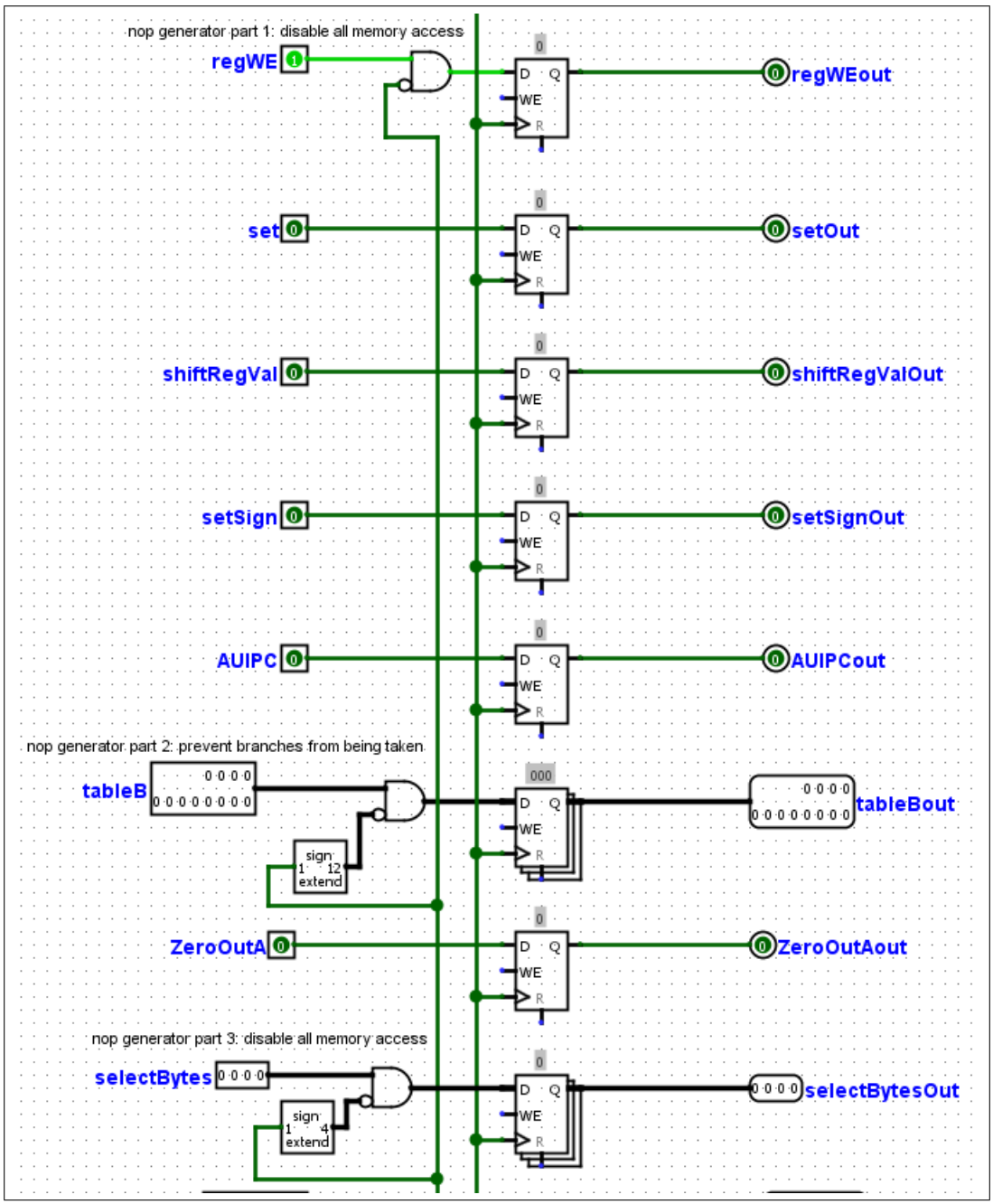

Figure 7: A portion of the D/X pipeline register subcircuit showing control logic for implementing nops: the memory selection bits that define the bytes of a word written to memory are all zeroed, and the write-enable bit for the register file is similarly deasserted. Since table B instruction often alter the program counter, the table B flag is also zeroed to prevent unitedted jumps or branches. The effect of these modification is the injection of a dummy instruction that has no lasting effect on the state of the circuit, i.e. a nop. The control signal being sign extended to achieve this is the nop flag passed from execute.

5 The Execute Stage

5.1 Description

Every implemented RISCV instruction requires some kind of arithmetic or logical operation(s) to be meaningful. Since the foundation for the table A instruction existed prior to our work, we focused on integrating the necessary logic for table B with this established framework. To understand this stage of the pipeline, it is helpful to follow the data streams for A and B, the outputs from r1 and r2 in the decode stage. These

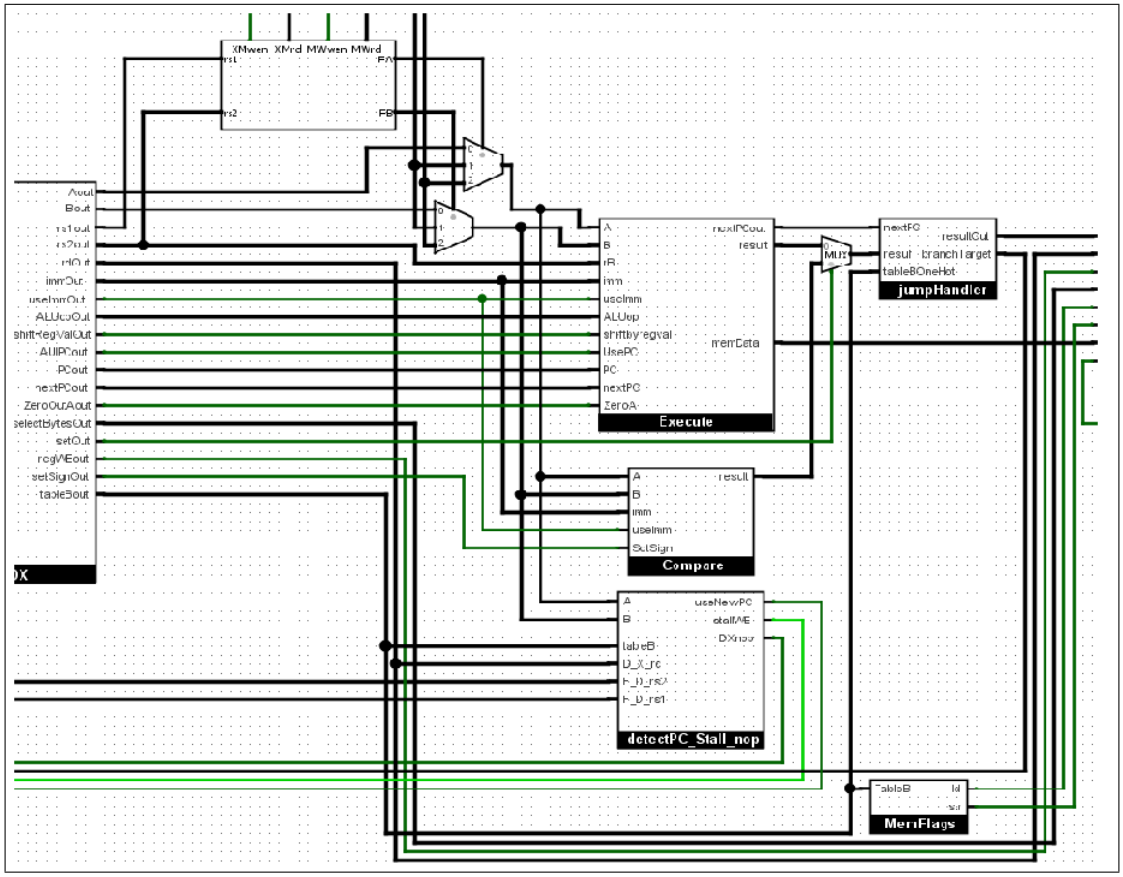

Figure 8: Complete execute stage containing MX and WX forwarding logic, stall detection, nop detection, and ALU output generation/modification via the Execute subcircuit and helper circuit jumpHandler

5.2 Subcircuit Diagrams

We will call this sub-circuit DetectPC_Stall_nop It will have the following inputs:

- A
- B
- tableB
- X/M.Rd
- D/X.rd

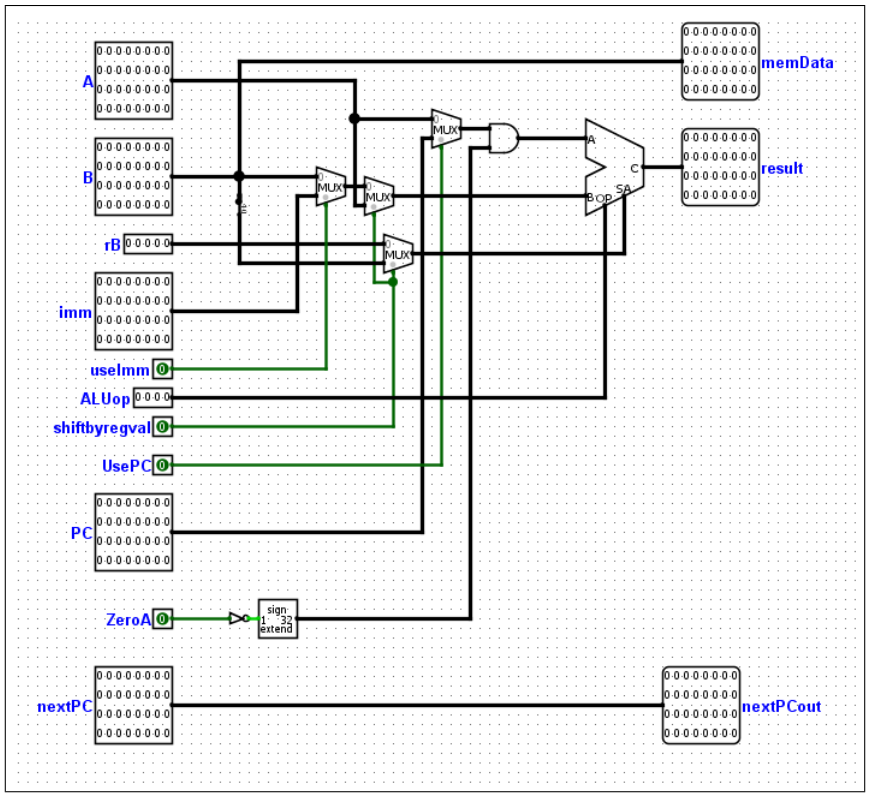

Figure 9:

- F/D.rs2
- F/D.rs1

It will have the following outputs:

- 1 bit flag, UseNewPC (used for mux control)
- 1 bit flag DXnop (used for mux control
- 1 bit flag stallWE, which will be used to turn instructions into a nop

useNewPC is passed back to the fetch stage so that we can decide whether or not to use the nextPC or the one coming from the execute stage (in the case of a jump or a branch).

stallWE is used to in the fetch/decode registers. It is passed into each F/D register in the WE slot. If it is true, then write is enabled, and we are not stalling, otherwise WE is false which indicates a stall.

DXnop is used to disable memory and register writing so that the current instruction, when it runs due to the stall does nothin, effectivley making it a nop.

MemFlags Component Inputs

- tableB

Outputs

- 1 bit flag ld
- 1 bit flag str

This component takes in the tableB one hot and outputs 1 for ld if the instrucion is LW or LB (ie a load). It outputs 1 for str if the instruction is SW or SB (ie a store). Both output 0 otherwise.

These two flags are passes to the XM register

jumpHandler Component Inputs

- nextPC
- result
- tableBOneHot

Outputs

- 32 bit output resultOut
- 32 bit branchTarget

This component takes in the the nextPc, result, and tableBOneHot. It decides whether to use the nextPc or the result from the ALU.

6 The Memory Stage

6.1 Description

The memory stage is comparably more simple than the execute stage, relying only on minor logic to handle RAM input and output. Specifically, in the event that an inbound 32-bit address from the ALU has nonzero bit in 12 most significant places, the memory operation is ignored by zeroing-out selectBtyes, as this prevents an otherwise out-of-bounds instruction from accessing memory. Further, in the special case of an LB operation, a small subcircuit sign-extends the RAM output. Forwarding paths are also present in this stage, and head to the 3:1 muxes discussed in Section 5.

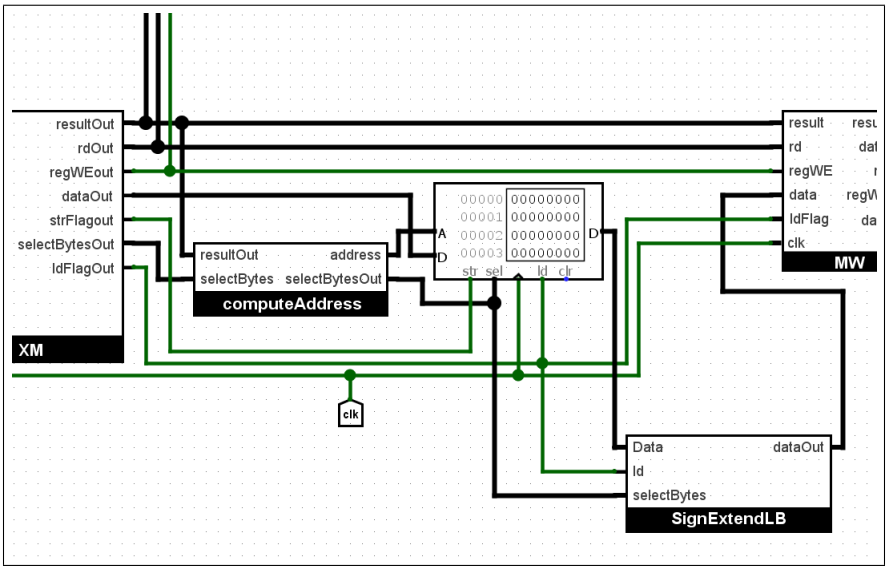

Figure 10: Memory stage incorporating RAM module and supplemental subcircuits for input/output modification. The memory flags from execute determine which function (read/write, if any) that the memory performs at address A. The address

6.2 Subcircuit Diagrams

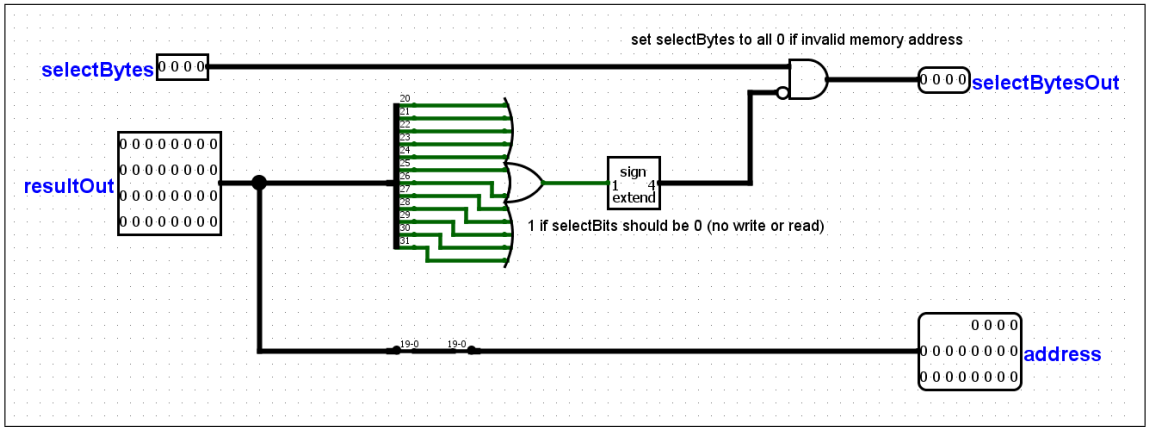

Figure 11: Subcircuit modifying RAM inputs to account for memory operations attempting to write to or read from nonexistent (> 20 bit) addresses.

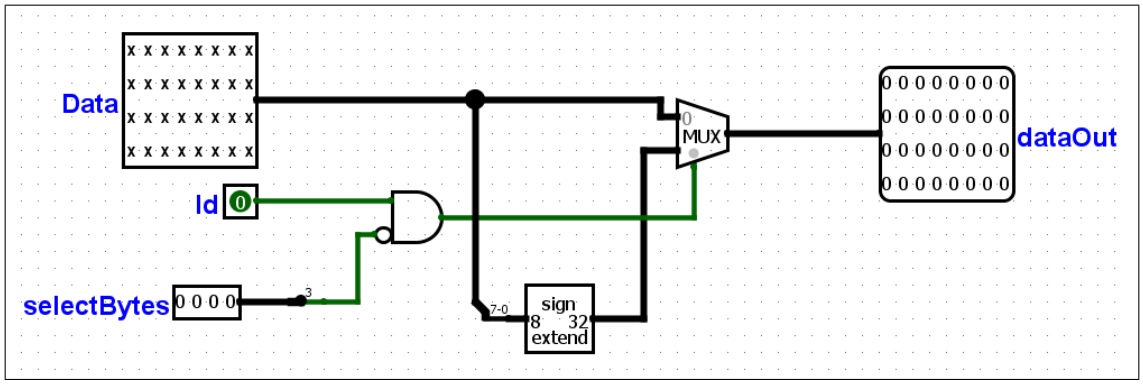

Figure 12: Writeback stage showing memory/ALU output distincation via the mux. The ld flag is carried from the memory stage and sends the output from the memory module to the register file when it is asserted.

7 The Writeback Stage

7.1 Description

The final pipeline stage requires only distributing

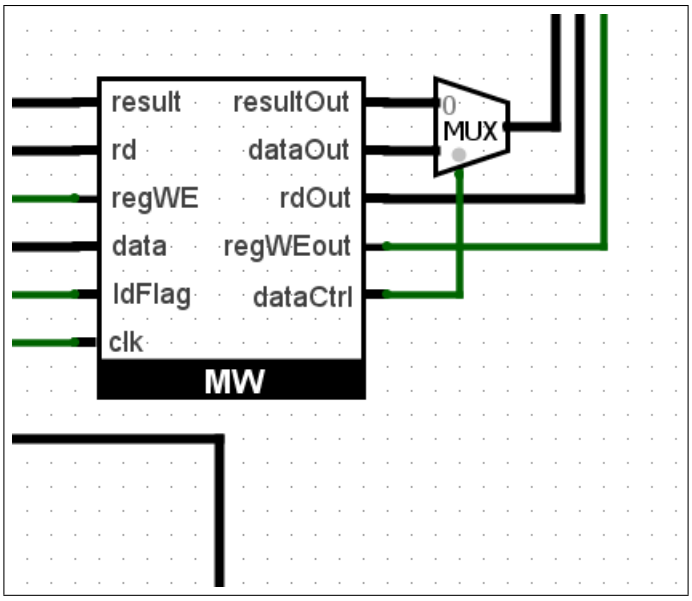

Figure 13: Writeback stage showing memory/ALU output distincation via the mux. The ld flag is carried from the memory stage and sends the output from the memory module to the register file when it is asserted.

8 Testing

We did three different testing methods.

The first method of testing was test vectors in decode. We used python to generate test cases for each instruction in table B. For each of them, we had to figure out each of the output fields for a given instruction. The python generation scripts are in the GitHub repository for reference.

This set of tests ensure that comes out of decode is accurate in the rest of the main circuit.

The second set of tests is the fibonnaci tests. Those are pretty straight forward and shows that we can execute a program that utilizes many different types of instructions.

The last is our RISCVTESTS.txt file. This set of tests tests each individual table B instruction without any hazards, tests each table B instruction for both forwarding and load use hazards, tests table A isntructions in a general manner to ensure that they work as expected.

Some of the notible edge cases in this test file was using negative immediates to make sure bit extension worked, having multiple branches/jumps back to back, making loads and stores where nothing is supposed to happen, and doing loads/stores and uses while changing the same registers and memory addresses.