Design Documentation

Samuel Thomas (sgt43) Elisha Sword (eds88)

September 20, 2017

1 Introduction

The purpose of this document is to communicate the design of our 5-stage pipelined MIPS processor. We go into detail of the design of each of the five stages in the pipeline as well as the pipeline registers that connect the stages. This document assumes knowledge of MIPS and of logic gates. It is not meant to be instructive in these regards.

A pipelined processor is a processor that executes each instruction by passing it through different stages. At the end of each stage, the results are stored. At the beginning of each stage, the stage grabs the output from the previous stages. Although pipelining increases the number of clock cycles that it takes to execute each instruction, we can reduce the clock speed and increase the throughput of our processor.

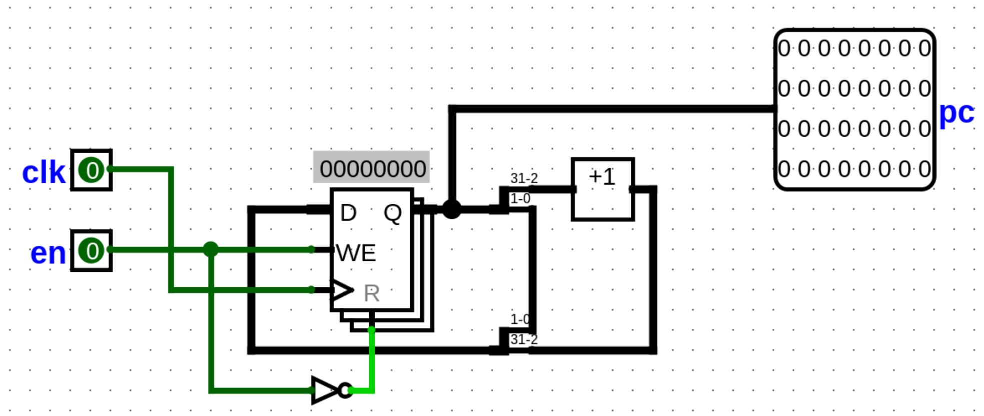
2 Overview

We present a 5-staged pipeline design. Instructions are fetched in the first stage. Instructions are decoded and registers are read in the second stage. The third stage executes an instruction. The fourth stage writes values to memory if required. The fifth stage writes the results of the execute stage back into the register file.

3 Instruction Fetch Stage

This stage fetches the next instruction to execute. We store all of the instructions in Program ROM and keep track of where we are in the execution with `PC`. Every clock cycle the `PC` is incremented.

3.1 Circuit Diagram



3.1.1 PC Incrementer

We store the `PC` in a 32-bit register. Every clock cycle, we increment the upper 30 bits by 1 with a `+1 Incrementer` without touching the lower 2 bits. This gives us the desired effect of advancing the program counter by four every clock cycle. The `en` input is an on/off switch. When turned off, it resets the counter.

3.1.2 Latch

We will store the fetched instruction in the latch at the end of this stage. We are not supporting jump instructions so we do not store `PC+4`.

3.2 Correctness Constraints

Fetch the next instruction on the rising edge of the clock cycle.

- Correctly increment `PC` by 4 every clock cycle.

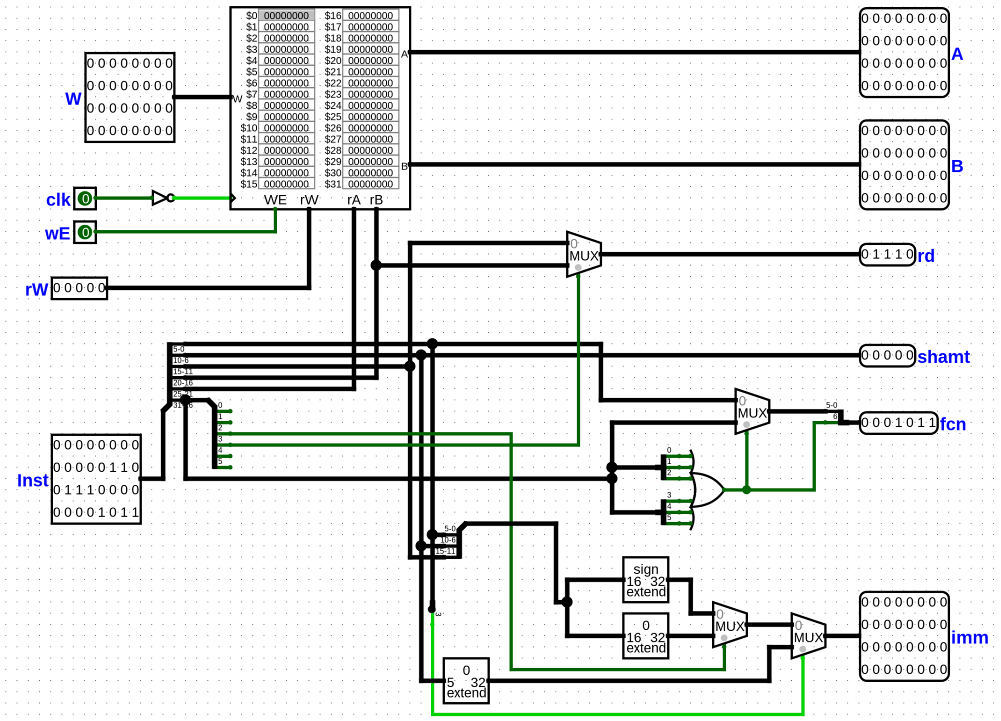
3.3 Testing

Test that the instructions are fetched in the correct order.

4 Instruction Decoding Stage

This is the stage where our register file lives. In the first half of the clock cycle, we write input `W` to register `rW` if `wE` is high. In the second half of the clock cycle, we read registers `rA` and registers `rB` and output them as `A` and `B` respectively. We also split the opcode into `rd`, `shamt`, `fcn`, and `imm` so that future stages simpler.

4.1 Circuit Diagram



4.1.1 Outputs

- `A` and `B` are read from the register file based on `rA` (bits 21-25) and `rB` (bits 16-20).
- The output `rd` is determined by `bit 3` of the opcode. When this bit is low, `rd = bits 11-15`. When this bit is high, `rd = bits 16-20`.
- `shamt` is always equal to `bits 6-10`.
- `fcn` is a 7-bit number that will be used in the execute stage to determine what operation to give to the ALU. The lower 6-bits of this number from either the opcode `bits 0-5` or, if the opcode is all 0s, from `bits 26-31`. The upper bit of `fcn` is 0 if the opcode is all 0s and 1 otherwise.
- `imm` is `bits 0-15` (either sign extended or zero extended depending on `bit 2` of the `fcn` part of the instruction), or `bits 6-10` zero extended. This is determined by `bit 3` of the opcode. The purpose of this is to have a 32-bit zero to compare against in the execute stage.

4.2 Correctness Constraints

- `W` is correctly written to `rW` in the first half of the stage.
- `A` and `B` are correctly read from the registers based on `rA` and `rB`.
- `rD` is correctly chosen based on the instruction.
- `shamt` is fetched correctly from the instruction.
- `fcn` correctly chooses between the opcode and the special function portion of the instruction.
- `imm` correctly outputs sign-extended lower 16-bits, zero-extended lower 16-bits, and 32-bit zero.

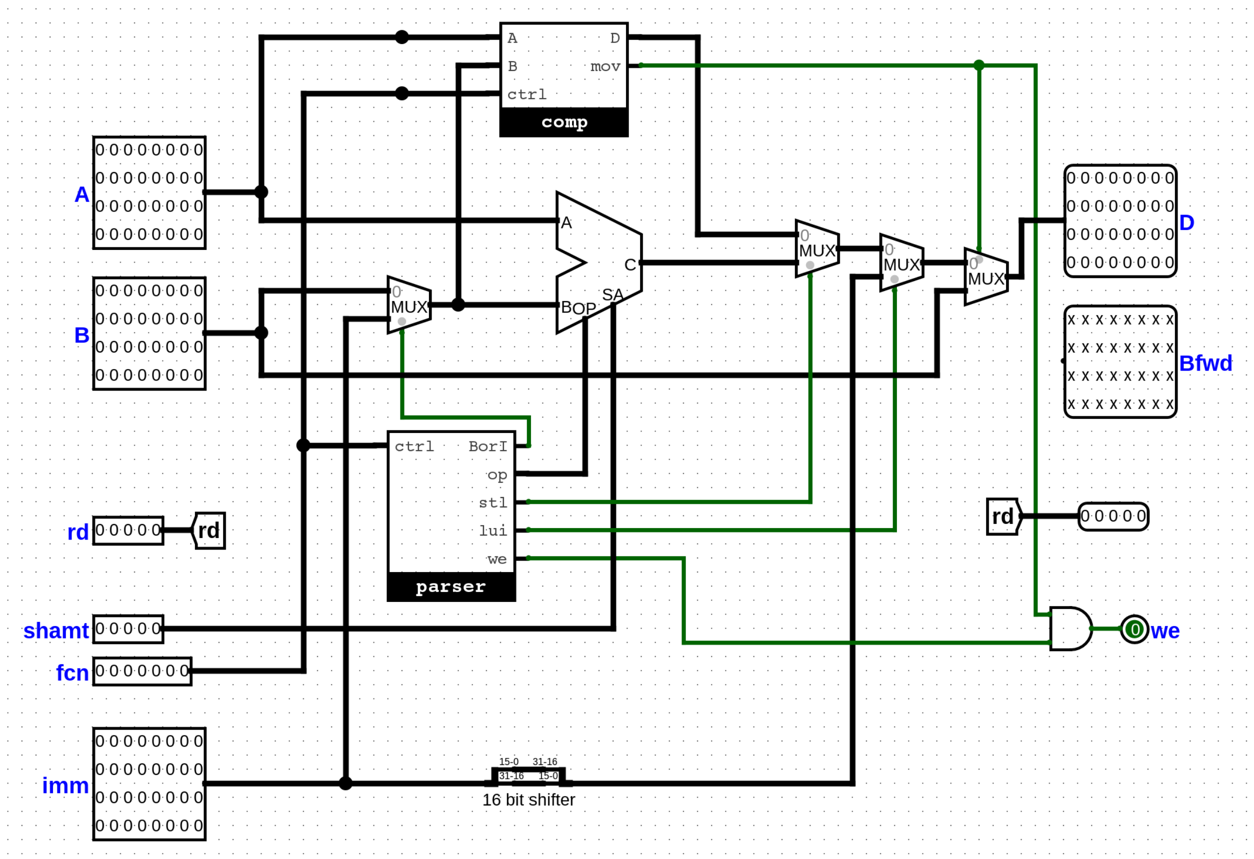
4.3 Testing

Test vectors will be written to test each possible opcode to see if the above conditions are true. To test if the values are correctly written in the first half of the stage, we will write a value to a register and attempt to output the value from the same register.

5 Execute Stage

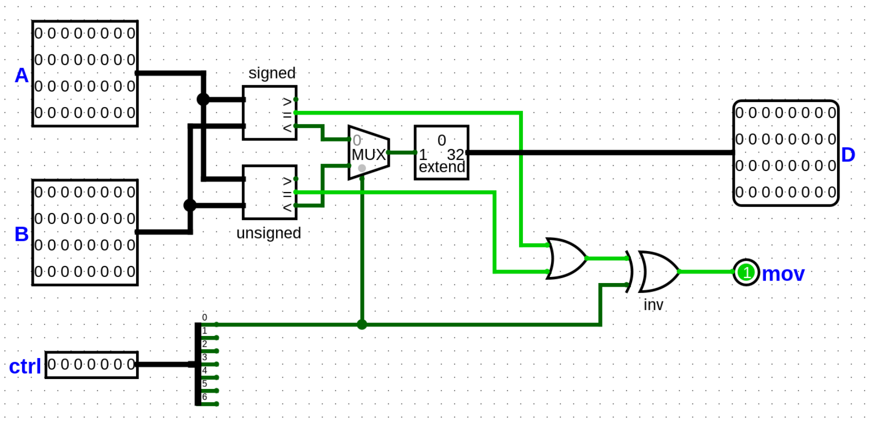
This stage executes all of the instructions that we will implement in this project.

5.1 Circuit Diagram



There are two executing subcircuits (`comp` and `ALU`) and one logic subcircuit (`parser`). `Comp` performs comparison operations for `SLT`, `SLTU`, `SLTI`, `SLTIU`, `MOVN`, and `MOVZ`. `ALU` performs all of the other arithmetic operations. `parser` parses the `fcn` input into an opcode for the ALU as well as other control logic that chooses between different outputs.

5.1.1 Comp



This circuit uses an unsigned and signed comparator to perform $A < B$. We also use the comparators to perform $A = B$. An `XOR` gate is used to invert this when `bit 0` is high. `Bit 0` of `ctrl` also switches between signed and unsigned.

5.1.2 Parser

All the bits referenced below refer to bits of `fcn`.

- `Bit 0-2` determine the 3 upper bits of the ALU opcode. The lowest bit is always 0.
- `Bit 6` chooses whether or not to replace `B` with an immediate. We choose an immediate with `Bit 6` is high. The one exception to this is when we are performing `MOV` instructions. In this case, we also want to replace `B` with an immediate (which will always be 0).
- If `Bit 0-3` are all high, then we are performing `LUI`. Here we replace the output of the `ALU` with `imm` shifted left by 16.
- Finally, we always want to output `we` except when `MOVN` or `MOVZ` return false.

5.2 Correctness Constraints

- `we` should be on for all instructions except when `MOVN` or `MOVZ` return false.
- Addition, Subtraction, Shifting, Logic operations, Comparison operations should all execute as expected.

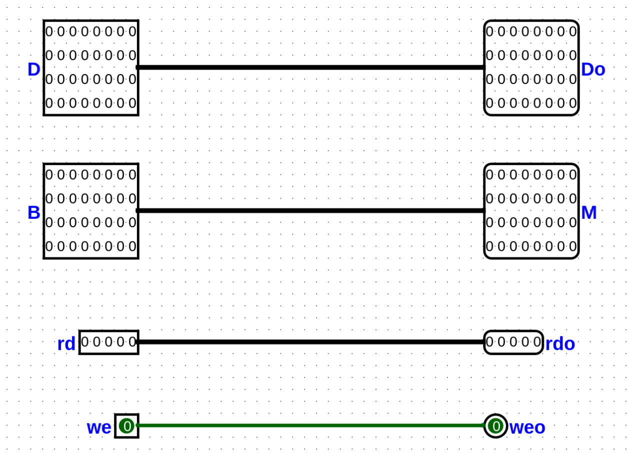
5.3 Testing

We will write text vectors to test all valid inputs to see if the circuit outputs the correct results.

6 Memory Stage

In this project, this stage doesn't do anything. It simply forwards it's inputs to the next stage.

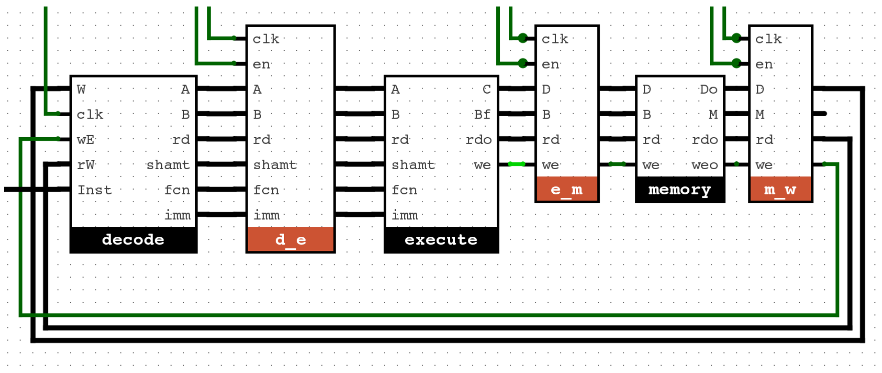
6.1 Circuit Diagram



7 Write-Back Stage

For this project, there is no logic in this stage. It simply pipes `D`, `rd`, and `we` back to the decode subcircuit where it is written to the register file.

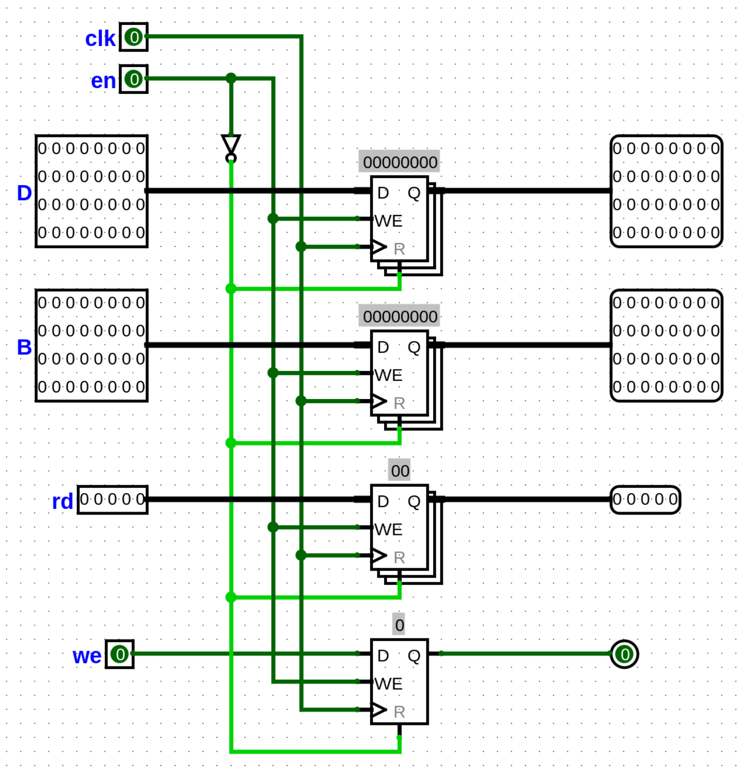
7.1 Circuit Diagram



This diagram shows more than just the writeback. For this project, the writeback stage simply consists of wires. For the next project, we will need to add logic to chose whether to write `D` or `M` to the register file.

7.2 Pipeline Registers

In between each stage, there is a pipeline register. These simply contain a register for each input that are written to on the rising edge of the clock. Below is the implementation of the execute-memory pipeline register. All of the others are implemented in a similar fashion.



8 Summary

In this document, we presented a design for a limited 5-stage pipelined MIPS processor. Such a design greatly increases the performance of our processor by allowing multiple instructions to be executed simultaneously.