**Required parts of the design:**

For this checkpoint, we worked toward getting the base of the 5-stage pipeline processor that would be capable of decoding instructions, and interfaced with a magic memory.

**Major component divisions:**

1. Conceptualizing the datapath and coming up with the paper design, as well as choosing the form of implementation and the division of work: Entire group met twice to discuss the same.

2. Implementing the Control Word design to decode each instruction: Code primarily written by **Johan**, using reference from MP2 to set each signal for all muxes in the control word – replaced the FSM from MP2.

3. Constructing the datapath, creating all the components of each stage:

Fetch: PC logic

Decode: Containing the Register file, and incorporating the Control ROM logic

Execute: Containing the ALU with inputs from the 2 ALU muxes, CMP mux, and the branch logic

Memory: Containing the magic memory controlled by the Control ROM.   
Writeback: Containing the MUX that chooses where to store values back into the Regfile.

This section was primarily written by **Quinn** and **Geitanksha**.

4. Verification and debugging:

We faced a lot of small issues in our code, several typos and logical misconceptions. We all worked on it to fix those bugs.

**Functionality:**  
The Processor was capable of most RV32I instructions, with a couple of exceptions like CSRR, EBREAK and ECALL. We did not implement forwarding or branch prediction, and also used Magic Memory for the interface.

**Testing:**

For testing, we started out by making sure that the program code provided by the TAs executed correctly by using Verdi to understand all the signals, and check if the registers stored the correct value at the end, which covered most instructions. We also wrote simple assembly to test every instruction that we had implemented, logical, bitwise and arithmetic. We put predefined values in with appropriate NOPs and checked if the result we expected in the registers was achieved.

**Next checkpoint:**  
  
**Required parts of the design:**

1. Forwarding: This involves taking the values generated by the ALU (alu\_out) and the Memory (rdata) and sending them to the inputs of the ALU (via 2 muxes) in order to handle the cases where a source operand of a newer instruction in the pipeline requires an older instruction’s value before it is committed.
2. Branch Hazards: This involves making sure that if a branch is not taken in the code, we correctly flush the pipeline and don’t commit or execute any instructions we were not meant to.
3. RVFI Monitor: This involves setting up the RVFI monitor by adjusting the signals going into the monitor from the writeback stage of the design.
4. Arbiter and cache interfacing: This involves making sure that the arbiter is able to interact with the instruction cache and the data cache simultaneously and use the arbiter to do so.

While we have not specifically divided up work, these are the 4 ways in which we will split up the work. The general paper description is put up on the Draw.io file on the GitHub Repository.

**The Arbiter:**

This is going to have 3 major states: wait, data\_fetch, and instruction\_fetch. There are 7 possible scenarios we have considered so far:  
1.

2. I-cache interaction only

3. D-cache interaction only

4. Both caches interact: The I-cache is given priority

5. D cache attempts to write to I-cache location (suspected to be a not-use case, as structural hazards are not being tested)

The following 2 will not occur if we stall, but should we choose to optimize it is a possibility, but it will be handled by first-come-first-serve:

6. D-cache interaction then I-cache interaction

7. I-cache interaction then D-cache interaction