**Required parts of the design:**

For this checkpoint, we worked toward fixing forwarding hazards, interfacing with 2 caches (D-Cache and I-cache via arbiter), and getting the RVFI set up.

**Major component divisions:**

1. Implementing the Forwarding scheme and deciding the 3 major cases of forwarding:

* Either of the source operands required for a new operation are in the memory stage or the writeback stage, they must be sent to the execute via the ALU MUXes.
* If the memory is not ready when performing a read or write, the pipeline must stall (via arbiter’s signal).

Code primarily written by **Quinn**.

2. Implementing the Arbiter, instantiating caches, and checking functionality, handling branch misprediction (when nops are no longer available, flush the pipeline if the branch is taken when it should not have been). Writing the report and updating the datapath with required changes, especially for arbiter.  
Primarily worked on by **Geitanksha**.

3. Implementing the RVFI monitor, analyzing the signals required and testing the entire system with it, and working on Spike.

Code primarily written by **Johan**.

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, without requiring magic memory, or nops – currently we simply stall. We still have not implemented instructions like CSRR, EBREAK and ECALL. We implemented forwarding or branch prediction, and also used 2 caches for the instruction / data memory.

**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 no NOPs and checked if the result we expected in the registers was achieved. We also tested every version of the Branch scheme.

**Next checkpoint:**  
  
**Possible features considered to be implementing:**

*L2 + Cache System (2)*  
We would add additional caches that would 'consume' evicted values in our L1 cache (L n-1 cache). This will require adding additional logic to the provided cache. The logic would facilitate providing the evicted cacheline to the L2+ caches.

*4-way set associative data cache (4)*The data cache would be a good candidate for this. We would add additional cachelines (and make the indexing differences necessary) to the data cache.

**Advanced Cache Options:**

*Eviction write buffer (4)* is being considered – Quinn has a partially working eviction write

Victim Cache (6)  
We would have to add additional logic to the provided cache (or make our own), to support the changes to the standard cache's FSM. We would need to push the evicted cacheline during our fetch stage, and then use the buffered cacheline to write to the next level cache.

**Branch Prediction Options:**

*Local Branch History Table (2)*

We would have 2 arrays, one with a 2-bit value updated on an FSM (dynamic scheme) and one of the arrays storing the address we need to jump to.

*Global Branch History Table (2)*

We would have the first array be a larger sets of bits, each representing the history of taken or not taken (weakly / strongly) and use the PC to index into it and based on the past history, alter our FSM accordingly.

*Tournament Branch Predictor (5)*

Using the Local Branch History Table and the Global 2-Level Branch History Table, we will implement the Tournament Branch Predictor and maintain the two-bit counter as was presented in the lecture slides. Where the two bit values correspond to: 11: Strongly Local Branch History Table 10: Weakly Local Branch History Table 01: Weakly Global 2-Level Branch History Table 00: Strongly Global 2-level Branch History Table And we will move between these values depending on the rules that predict correctly (on branch taken).

**Prefetching Design Options:**

*Basic Hardware Prefetching (4)*

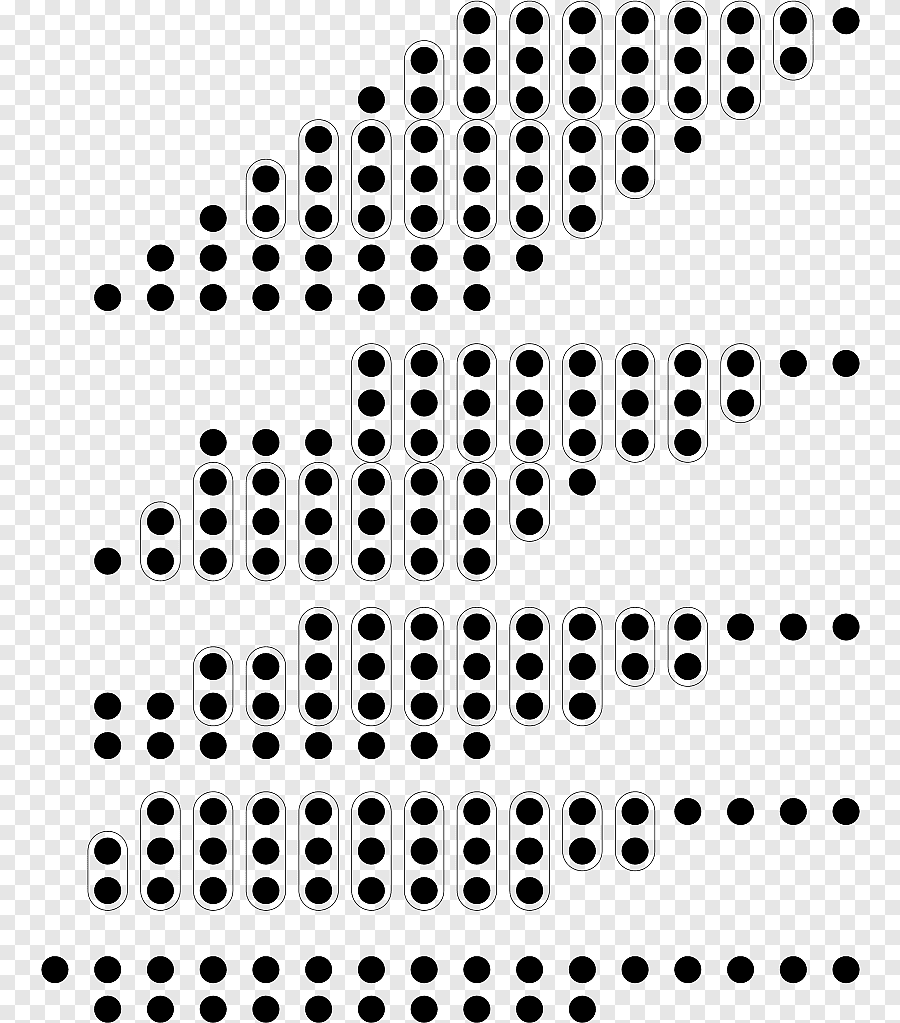
We will implement OBL prefetch, which aids in exploiting spatial locality. We will take a signal from the arbiter to see if there was a miss, after which we will send a call to the arbiter to fetch an i+1 cacheline. But, before that, we will check the cache’s next line and make sure it is not hosting the same address we send a fetch for.

**Difficult Design Options:**

*Basic Multiplier / Wallace Tree (4)*

We would first employ the add-shift multiplier (3) in order to check if we can get the multiplier working as we did for MP1. After this, we will try to employ the Wallace tree:

The Wallace tree’s algorithm is that we do individual partial sums which get added up in one singular step at the end, when all partial products are created.



Instead, we split up the addition into several smaller steps.

We end up summing 3 values at a time, which vastly reduces the amount of partial products that need to be summed up together at the end. We will be using full and half adders for this.