ECE 411: mp_ooo Progress Report

Checkpoint 1:
Akhil: Draw a design block diagram (using draw.io or similar, not handwritten!) of the basic OoO datapath [5]
Sam: Implement a parameterizable queue (depth and width) [5]
Ryan: Implement instruction fetch and integrate it with magic memory and your parameterizable queue [7.5] (Your program counter should be initialized to 0x60000000)

We tested the parameterizable queue and fetch through two test benches, fifo_tb and fetch_tb. In fifo_tb, we filled up the entire queue to the size of its depth to check it does not push to the queue when it is full. Similarly, we popped the entire queue to check it does not read anything when the queue is empty. We also test concurrent push and pop and if the queue becomes empty on reset signal.

In fetch_tb, we first check the implementation of instruction fetch and its integration with magic memory by checking the value that is being read/popped from the queue. We also use ordinary memory to test if the queue gets populated correctly when fetching instructions at varying cycles. The fmax of this design is approximately 1.534GHz as the minimum clock is 0.652ns.

Roadmap for Checkpoint 2:
Akhil: Integrate the provided multiplier into your processor as a functional unit [5]
Ryan / Sam: Your design must support out-of-order execution of the arithmetic operations (you must implement ROB, RS, CDB, etc.) [5]

For checkpoint 2, we will use the provided multiplier and make changes if necessary to integrate into the processor. We will test the implementation by writing test cases for multiply operation similar to what was done for mp_pipeline using RVFI. Out-of-order execution for arithmetic operation will be implemented through ROB, RS, CDB, RRF, and RAT. RVFI

Checkpoint 2:
Sam: Integrate the provided multiplier into your processor as a functional unit [5]
Akhil/Ryan/Sam: Your design must support out-of-order execution of the arithmetic operations (you must implement ROB, RS, CDB, etc.) [5]

We have created 2 functional units so far: a multiplication unit and an ALU.The provided shift add multiplier is instantiated in the multiplication unit in order to execute multiplication operations. Both functional units output ready signals when they’re ready to issue the instruction, and the CDB chooses which functional unit’s instruction to send for issue. Our CDB prioritizes

multiplication instructions over ALU instructions and will send the multiplication instruction to the ROB first in the event of a conflict.

Going off of the block diagram we created in CP1, we have created an instruction decoder that takes an instruction from the instruction queue and sends its architectural registers to the Rename stage. The Rename stage refers to the RAT to determine the mapping of the architectural register and replaces the architectural registers in the instruction with the physical registers that correspond to them on the RAT. The Rename stage will then send this information along with the ROB ID to our reservation station, which will determine the correct functional units to send the instruction to and update the ROB accordingly. The CDB then takes the instruction of the available functional unit, and if there's a conflict between a multiplication unit and an ALU, the multiplication instruction will be prioritized, and updates the ROB with the instruction and ID, and updates the RAT with registers that aren't in use anymore. Once the instruction is committed, the physical registers will be sent to the RRF and Free Lists so that they can be used in the next instruction for renaming. The fmax of this design is 469MHz and the minimum clock is 2.13ns.

Roadmap for Checkpoint 3:
Akhil: Add support for control instructions [3]
Ryan: Add support for memory instructions [3]
Sam: Integrate with your mp_cache with competition memory (Instruction and Data) [3]

For Checkpoint 3, we will add support for the remaining control and memory instructions by building a branch predictor to enable branch predictions and enabling flushing of any branch mispredictions. We will also enable loads and store instructions in our processor and will enable the appropriate stalling logic for them. Finally, we will integrate our cache with the competition memory to increase performance.