Introduction:

In this report, we discuss the development and design of a RISC-V processor utilizing Tomasulo's algorithm, enhanced with several advanced features to improve performance. The processor design incorporates reservation stations for the ALU and multiplier units, a dedicated buffer for load and store instructions, and integrates both an instruction and a data cache. This implementation serves as an exploration into advanced processor design techniques within the realm of computer architecture.

Project Overview:

The project's primary goal was to implement a functional and efficient RISC-V processor by applying Tomasulo's algorithm, known for allowing dynamic instruction scheduling and out-of-order execution, which can significantly enhance CPU performance. It could run ALU operations, multiplier operations, control instructions, and memory instructions. The project was inspired by the need to understand and implement complex processor features practically as a part of our course. We divided each week of the project into separate parts, each handled by different team members focusing on specific components such as ALU or multiplier operations, memory management, and branch prediction. Regular meetings ensured that individual contributions were working properly, and challenges such as debugging and integration were managed collaboratively.

Design Description:

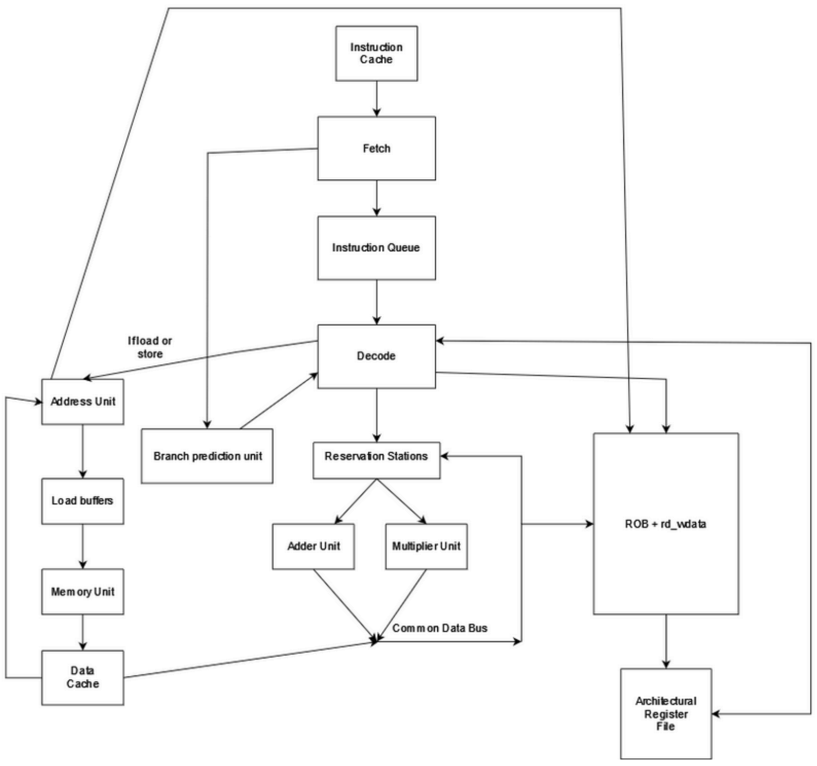
1. **Overview:** The design was structured around the implementation of Tomasulo's algorithm, which uses reservation stations for managing instruction execution

dynamically, coupled with a reorder buffer to maintain correct instruction order. It has two types of reservation stations, a branch predictor, divider, and a load/store buffer.

2. Milestones

2.1. **Checkpoint 1:** At checkpoint 1, our processor was able to fetch instructions using a parameterizable queue. We also created a datapath for our final processor (with branch prediction), which is pictured below.

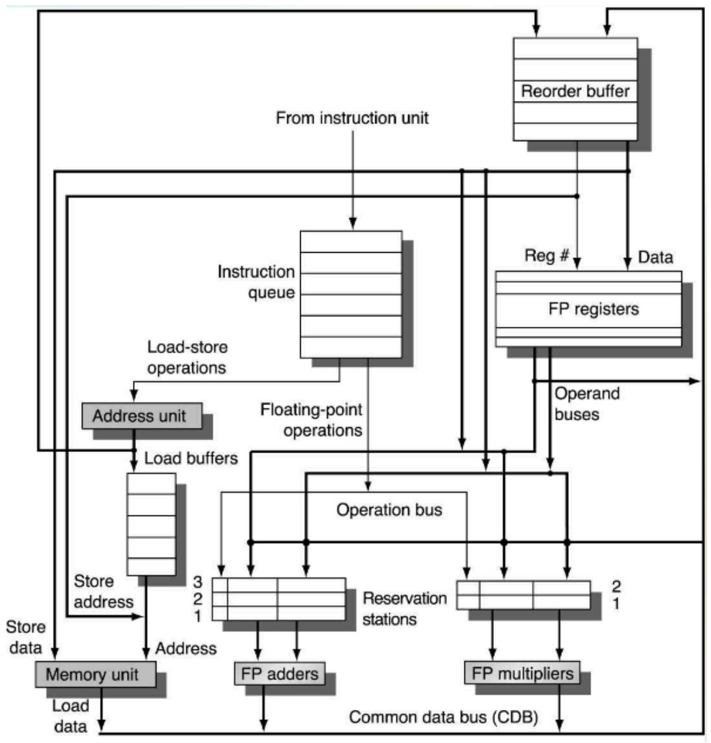
We tested our code for checkpoint 1 using assembly test cases and ensuring that the Verdi waveform was correct. Ensured that each instruction is fetched correctly and is added to the instruction queue.



2.2. **Checkpoint 2:** At checkpoint 2, our processor was capable of holding instructions in the queue until the reservation stations were ready. It was also capable of holding instructions in reservation stations until they were ready. It was capable of

doing alu operations and multiplication instructions. It can send several instructions to the ROB out of order and the ROB will commit them in order. The architectural register file is connected to the reservation stations and the ROB through the common data bus.

Our testing strategies for checkpoint 2 was verdi and assembly test cases. We used the provided assembly test cases, ooo_test.s and dependency_test.s to ensure that the ROB was committing the correct data. We looked at verdi to confirm that the instructions were reaching the ROB out of order, but were then being committed in the correct order by the ROB as efficiently as possible.



2.3. Checkpoint 3: At checkpoint 3, our processor was capable of holding instructions in the queue until reservation stations were ready and capable of holding instructions in reservation stations until they were ready. It could do alu operations and multiplication instructions. It could send several instructions to the ROB out of order and the ROB will commit them in order. It was capable of doing branch and jump operations and flushing the mispredicted instructions. It also used an instruction and data cache and was capable of doing load and store functions using a load buffer.

Our testing strategy for checkpoint 3 was verdi and assembly test cases. We made assembly test cases to test the branch and jump functions separately to ensure they were jumping to the correct pc value and were flushing mispredicted values. We also used assembly test cases to test the load and store instructions, and compared that with what the spike log should be from mp_pipeline. We ran coremark successfully on our processor once we had finished every part of cp3.

3. Advanced design options

3.1. Early Branch Recovery:

3.1.1. **Design:** Control Buffer tracks branch/jump instructions and operands. Branch mispredictions are detected when operands are ready. Then we use mis_tag_start and mis_tag_end to identify flushing range and clear entries younger than mispredicted branches inside ROB. Other stages clear matching entries based on ROB tags

3.1.2. **Testing:** Used our own branch test first, then used coremark.

3.1.3. **Performance:** Our design doesn't perform significantly better, and the area increased a lot. Reason for not improving is that our ROB size is small and there are not many instructions to recover.

3.2. Branch Predictor

3.2.1. **Design:** Implemented a gshare branch predictor with branch history table and global history registers. The predictor generates a prediction by indexing into the BHT using the XOR of relevant 10 bits of branch instruction address and the global history. The prediction is based on the most significant bit of the corresponding BHT entry.

3.2.2. **Testing:** Used our own branch test first, then used coremark.

3.2.3. **Performance:** Improve the IPC by about 30% but the overall performance is not as good as a 2-bit saturating branch predictor since it takes more area due to the branch history table. The prediction accuracy for gshare is higher.

3.3. Memory Disambiguation

3.3.1. **Design:** Built a store buffer decouples stores from the memory unit. Our design only supports in order load/store. With the store buffer, we can forward data from Store Buffer to loads in LSQ, before data storing to memory. And storing it in memory when rob commits.

3.3.2. **Testing:** Used our own memory test with dependencies first, then used coremark.

3.3.3. **Performance:** Slightly increase the IPC. But the in-order load store queue is the bottleneck of delay. Could improve performance if separate load and store.

3.4. Divider

3.4.1. **Design:** Integrated Synopsys DW_div_seq IP that runs in 3 cycles, implemented a state machine to handle the result delay. Shared one reservation station with multiplication operations.

3.4.2. **Testing:** Used our own division test with dependencies first, then used physics_d and rsa_d to test with other instructions.

3.4.3. **Performance:** Lower the delay by 75% for physics, and 60% for rsa.

Additional Observations:

Throughout the project, we noticed that integrating multiple parts of the design, like the control instruction functionality with the memory instruction functionality, often caused many unexpected bugs. Integration was where most of these arised, and fixing them caused our area to increase significantly.

Even though we implemented several advanced features, we don't see a huge improvement in performance. We think it might be caused by the limitation of ROB size and the fact that we don't support out-of-order memory operations. Therefore, the memory might be the bottleneck to lower the delay.

Conclusion:

Our project succeeded in implementing a RISC-V processor with Tomasulo's algorithm, effectively demonstrating the practical application and benefits of advanced processor design techniques. The integration of sophisticated branch prediction and memory management features slightly enhanced the processor's performance, although they also increased our area significantly. This project not only reinforced our understanding of complex CPU design concepts but also prepared us for tackling real-world computing challenges using innovative architectural strategies outside of college.