ECE411: Computer Organization and Design
MP4 report

Group: M2
Members: Zeduo Yu, Pengyang Zhou, Xiwei Wang
TA: Abishek Venkit
December 16, 2021

1. **Introduction**

   In this report we will introduce the architecture of our design of a CPU and its features, as well as the performance of our CPU. On the whole, we implemented an out-of-order CPU using Tomasulo's algorithm with speculative techniques, plus a dynamic branch predictor. We chose to go for an out-of-order CPU because it is a perfect chance to solidify what we have learned in lectures, and a good test of our capabilities in hardware design. What's more, the idea of out-of-order CPU is so cool and we could not hold back the excitement to implement it. The experience of implementing it indeed honed our skills.

2. **Project overview**

   The goal of the project is to implement the out-of-order RV32I Processor. Given the complexity of out-of-order CPU, our primary goal is to ensure that the design is correct and functional. We divided the whole project into many small parts and independently implemented and verified the functions of some modules. There were indeed a lot of problems in the process of integration. But with good communication within our team, we were able to quickly locate the inconsistencies in most cases and solve them efficiently.

   Based on ensuring correctness, we tried to optimize the performance. The speed of our final version was several times faster than the version that had just passed all the tests. In the process of improving performance, we also discovered some issues that were latent before, especially when we tried to fit more steps into shorter cycles. It is worth celebrating that we finally solved these bugs and had a highly completed project.

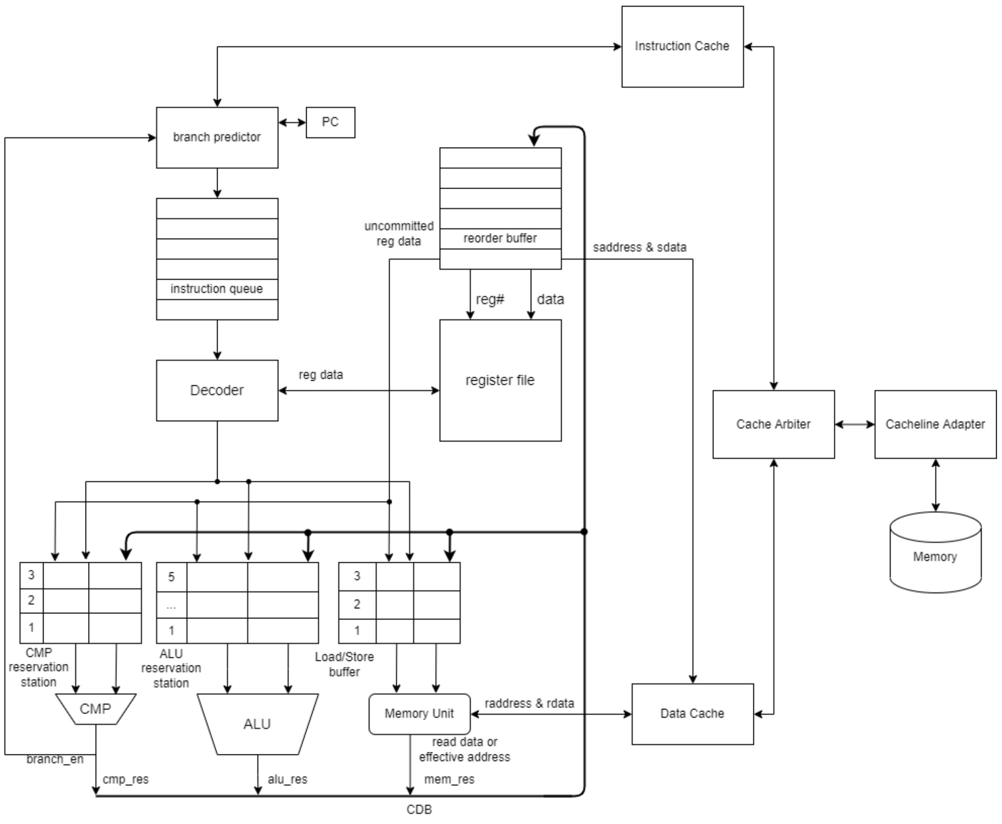
3. **Design description**

   1. **Overview**

      The project is based on the Tomasulo algorithm with some modifications to facilitate the implementation and accommodate the RV32I Base Integer Instruction set.

      The Branch Predictor reads one instruction per cycle from the I cache and passes the instruction to the Instruction Queue. When neither the Reorder Buffer nor Reservation Stations are full, the Decoder reads the empty entry number in the Reorder Buffer, the instruction from the Instruction Queue, and the value of registers from the Reg file. In the next cycle, the Decoder tells the Reorder Buffer the type of the instruction as well as the destination register involved and sends the value of the register to the corresponding Reservation Station or Load Store Buffer. When the Reservation Station or Load Store Buffer has the result, the reorder Buffer will be told and mark the entry as ready. Both the Reservation station and the Load Store Buffer can monitor the values provided in the Reorder Buffer to get the desired operands. Inside the Reorder Buffer, a pointer to the current entry will keep waiting until it is ready, and commit it and move to the next entry. Thus the out-of-order CPU can commit the instructions in order.

      Below is our overall design.



2. **Milestones**
   a. **Checkpoint1**
   We discussed the connections between the modules in detail, writing down concrete specifications to define the inputs and outputs of each module. We did not write too much code at this checkpoint but spent most of the time considering the possible boundary cases. Admittedly there were a lot of issues not discussed appeared in the later implementation process. But the detailed documentation did help us get a better idea of what we were going to implement.
   b. **Checkpoint2**
   We had the out-of-order CPU that could pass the tests of checkpoint1 and 2.

   We did unit tests for the module Instruction Queue and Decoder. However, The rest of the module is difficult to validate individually because most modules need to work with Reorder Buffer as shown in the overview. Nevertheless, the correct behavior of Reorder Buffer is hard to simulate. The way we adopted was that the person who implemented the Reorder Buffer ran ModelSim, observed whether the signals input to the Reorder Buffer by other models were in line with expectations, and told the team members who implemented the module with the problem about the irrationality. We finally verified the accuracy of the design by comparing the results of running the test code with the MP2 CPU.

   The logic of the trap signal was also implemented at this time, identifying the condition of one instruction jumping to the address of itself.

c. **Checkpoint3**

We set up the cache arbiter and integrated the design with the I cache and D cache. The I cache is a modified version of the one-way given cache such that all reads can complete in one cycle. The D cache is a similar two-way associative cache as MP3. The reason is to consider the continuity of instructions and the randomness of data.

We put a lot of effort into verifying checkpoint3 and the competition's test code. The most likely problem to go wrong is in JAL and JALR instructions, especially after the flush operation or in the case of issuing instructions. The way to test is the same as the previous checkpoint. We successfully passed all the tests and started discussing possible optimizations.

3. Advanced design options

a. Out-of-order Execution

i. Overview

This is the backbone of our CPU design, which is taken into consideration from the very beginning. The scheme we used is Tomasulo's algorithm with speculative techniques (Reorder Buffer). The overall structure is displayed above.

ii. Trade-offs

There are numerous trade-offs when designing, such as the number of entries in each reservation station, instruction queue and the ROB. The cache size is also another parameter we need to fine tune. We started by setting everything a moderate value, such as 3, 5 or 7. Based on the actual performance and profiling data, we continued to fine tune the parameters to optimize the overall performance.

iii. Performance

Note: since branch predictor is an inseparable part of our design, the following data were collected with branch predictor connected.

	comp1.s	comp2_i.s	comp3.s
Execution Time	702920ns	3040950ns	894860ns
Percentage of cycles issuing instructions	86.50%	43.54%	70.57%
icache hit rate	99.96%	97.80%	99.96%
dcache hit rate	99.80%	98.17%	96.10%
ALU RS utilization	26.21%	22.27%	46.11%
CMP RS	34.52%	42.86%	42.77%

utilization			
LSB RS utilization	70.98%	39.48%	73.77%

b. **Branch Predictor**

i. **Overview**

For Tomasulo's algorithm, the overhead is extremely high if CPU encounters a branch instruction (b branch instruction or jalr) and stalls, however, the overhead is also high if CPU calculates wrong next PC and fetches wrong instructions after a branch instruction. In this way, the branch predictor is necessary to guarantee the efficiency of the CPU. For b branch predictor, we implemented a 2-bit local branch history table, and for jalr, we implemented a return address stack.

ii. **Trade-offs**

The first trade-off is between accuracy and area. If the branch predictor has larger entries, it will usually have larger accuracy and the programs will run faster. However, larger branch predictors lead to more registers and logical circuits, so more power is consumed. In this way, first, we enable the branch predictor to replace the oldest-issued predictor entry when the predictor is full. Then, we adjust the size of the predictor entry to balance the area and accuracy.

Another trade-off is about critical paths due to updating branch predictors. That is, the reorder buffer first determines whether branch predictors should be updated, then it sends signals to branch predictors and branch predictors update the entries. However, both reorder buffer and branch predictors have complicated logic so the updating has a very long logic and will not be finished in time. For example, the maximum frequency of the CPU is less than 100MHz with branch predictors. In this way, we decided to use the pipeline to solve the critical paths. That is, in the first cycle, the reorder buffer should determine branch predictor updating signals and temporarily store those signals. And in the second cycle, the branch predictors should be updated with those signals. In this way, the critical paths are solved and the maximum frequency is larger than 100MHz again.

iii. **Performance**

Note: the metric of accuracy is the percentage of correctly predicted b branch instructions and predicted jalr and jal instructions within all branch instructions.

	comp1.s	comp2_i.s	comp3.s
Accuracy	89.52%	84.39%	89.12%

4. Conclusion

Our achievements have reached our design objectives. We have successfully implemented an out-of-order CPU with Tomasulo's algorithm, and we also implemented a 2-bit local branch history table and a return address stack. We have successfully speeded up the execution of programs on the out-of-order CPU and reduced the branch misprediction overhead. For critical path issues in updating branch predictors, we use pipeline to temporarily store the update signals and guarantee the maximum frequency of the CPU.

We have learned a lot of techniques around designing the CPU microarchitecture, determining interface details, verifying and correcting the design, profiling parameters to improve the results.