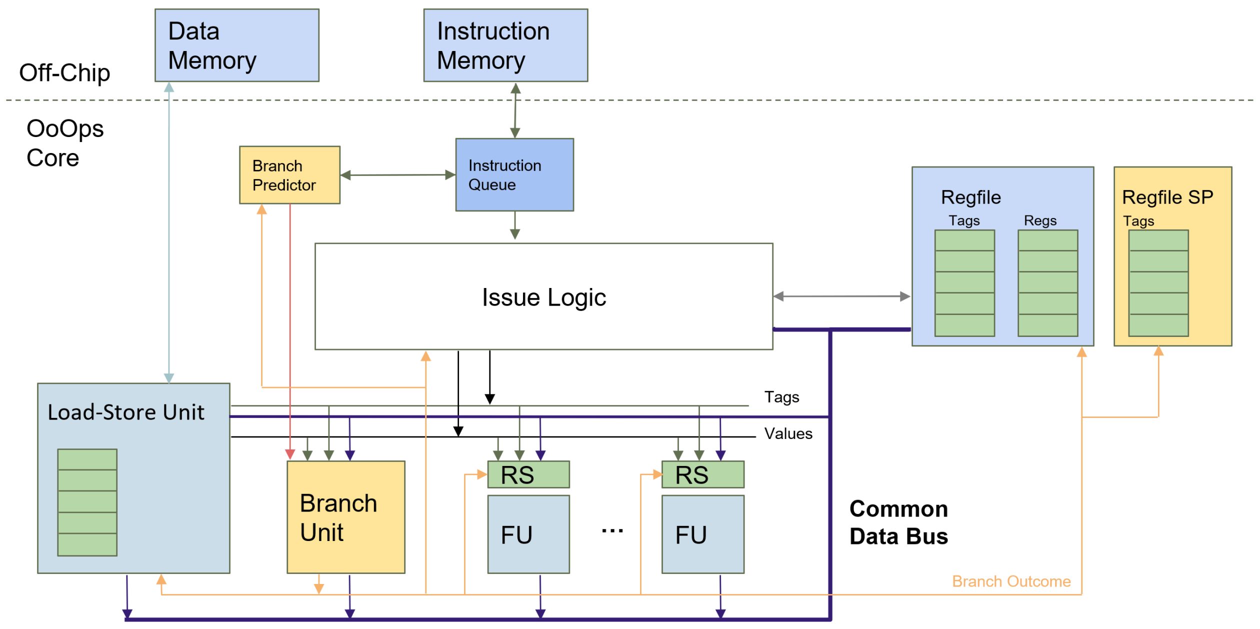
OoOps: Out-of-Order Execution

Rusdan Crook and Oleh Kondratyuk   June 9th, 2022



1 Core system/design

OoOps is an Out-of-Order execution CPU implementing a subset of the rv32i RISCV instruction set. The system is built on top of a hardware implementation of Tomasulo's algorithm. This central feature allows for dynamic scheduling of a program's instructions. Instructions can continue to execute in the face of load-store latencies, branch evaluation latencies, and circumstances where a given instruction may not have all of its operands computed. Tomasulo's Algorithm

Our usage of Tomasulo's introduces a form of register renaming by associating a logical register with a tag, a piece of data that points to a unique functional unit. This renaming indicates whether the value in a register is fresh or is currently being computed. In the latter case, the tag will point to its producing functional unit, and in the former case, the tag will be set to zero. An instruction and its tags are issued to a register known as a "reservation station", which will compare its tags against tag-value pairs broadcasted through a common data bus. The reservation station holds these values until the operands are ready (with updates provided by the common data bus), once ready, the reservation station issues its instruction to its associated functional unit. The common data bus is one of the central components of Tomasulo's algorithm, where the bus broadcasts a tag and associated value to much of the design: all reservation stations, the load-store unit, the register file and issue logic.

The brain of OoOps is the issue logic, which is responsible for checking availability of reservation stations, monitoring the common data bus to issue the most-recent data, interacting with the register file, and reading from the instruction queue.

2 Design Choices

2.1 RISCV Migration

After considering the default ARM thumb instruction set, we concluded it was suboptimal for implementing OoO. Since almost every instruction updated the flag register (but not all the flags), executing instructions out of order is made unnecessarily difficult. Since RISCV does not have any flags (branch condition depends on comparing two register values) the ISA was deemed more compatible with our goals and we migrated to a 32-bit RISCV ISA early in the design process.

2.2 Double Speculation on Branches

We made some simplifying assumptions going into the RTL design. We decided not to go down the path of branch-on-branch speculation. This simplified the branching and speculation logic while not having too much effect on performance since branches usually have some other instructions/work separating them giving time for the first branch to be evaluated before issuing a second branch.

2.3 In-Order Load-Stores

This was simply a pragmatic design choice that was made with the ticking clock in mind. We were behind where we wanted to be in terms of RTL and made this choice as time-saving measure. With more time, Out-of-Order load-stores would be one of the top priorities for making our project a true OoO cpu core. Speculation Implementation We decided to use a simple global correlated branch predictor that was simple to implement in RTL while still being fairly accurate once trained. A lot of the difficulty with implementing speculation was in tracking speculative instructions and squashing them in the event of a misprediction. This was done with a branch unit that acted much like a reservation station dedicated to branch instructions, where it stores the instruction until the operands it needs become available. Once the condition is evaluated, the information (that a condition has been evaluated and whether the prediction was correct) is broadcast to nearly every unit. This updates the branch predictor, causes reservation stations to either continue processing the no-longer-speculative instruction or squash the instruction there, resets the instruction fifo in the event of a misprediction, causes the regfile to overwrite speculative tags on misprediction and updates the speculation fsm in the issue logic. In effect speculation added a new fsm state to various modules and additional corner cases that had to be considered when coding the RTL. We unfortunately wrote the load-store unit in a fifo structure (since we decided on In-order execution), so in order to flush mispredicted load-store instructions, they had to be tagged with flags indicating that instruction was invalid and read out of the load-store fifo, introducing some additional latency in the event of a misprediction.

2.4 Parameterizable RTL

We decided to write our RTL to have as many parameterizable components as possible. While this takes more effort than simply hard-coding everything in the design, it makes the code more flexible and reusable for future projects. The instruction fifo and load-store buffers have variable sizes (in powers of 2) and can be adjusted by changing a single line. Our data_types.sv file acted as a top-level configuration file defining all the top-level parameters, enumerated and struct types. To change the number of reservation stations in the top level configuration is as simple as editing two parameters (a "number of" parameter and a TAG array (which must contain unique tags for the reservation stations)).

3 Verification Methodology

To verify our design, we wrote a collection of test programs in our CPU's subset of RISCV assembly. Since these programs were written in an assembly language, we wrote a custom-RISCV assembler to convert these programs into machine-usable code. The programs were loaded into an instruction memory model in the design's testbench. Our programs were written in lockstep with the development of new features. An initial set of programs tests the OoOps CPU's ability to handle conventional RAW hazards and WAR and WAW hazards introduced by OoO execution (`asm_gen/addi_instr_only.txt`, `asm_gen/alu_instr_OOO_test.txt`). Later programs target load-store and branching functionality, ensuring our in-order load-store policy and issue-but-not execute branch prediction policy (`asm_gen/for_loop.txt`, `asm_gen/bubble_sort.txt`). Due to time-constraints, we decided on using bubble sort as the design's litmus test due. After randomizing the memory to random values the program sorts the first 30 memory address locations. This program is key for testing the design's ability to execute out of order, handle branches, load-stores and flush speculative instructions on branch mispredictions.