Joseph Liba, Marta Taulet, Tianzhi Wu

Final Report

Superscalar Overview: Superscalar architecture contains multiple functional units, one of which has cycle time greater than 1. The main advantage of this processor is that multiple instructions that take advantage of different functional units will be able to run in parallel. The order that the instructions commit can vary. However, for our implementation, we use an in-order issue, out of order commit.

Implementation Goals: The functional units we used in this design were the standard ALU (integer version from base RISC-V processor) and MULTi. We would have liked to implement the ALUf, however, it was not so easy to implement using Verilog HDL.

The multiply unit will take 4 cycles long. It will run on MULT instructions as specified by the RISC-V manual. In the case another MULT instruction tries to run before a previous MULT instruction is complete, the processor will stall.

Although other superscalar architectures have a separate dispatch stage and commit stage, we opted for combining the dispatch stage with the decode stage. Furthermore, rather than having a commit stage, we allow for out of order writeback.

To deal with data hazards, stalling is done. The pipelined processor given to us by Prof. Kinsy following the mid-project report was not fully bypassed. Due to time constraints, we did not try to complete the processor and fully bypass. All resolutions are done with stall.

In the ideal scenario, a MULT instruction follows 3 ALU instructions with no data hazards. In this scenario, MULT runs simultaneously alongside the ALU instructions.

Designation of Tasks:

Marta Taulet - Creating assembly code that follows the following scenario:

1) MULT
2) MULT with no dependencies with Instruction 1
3) ALU with dependencies on instruction 2
4) MULT
5) ALU with no dependencies
6) ALU with no dependencies
7) ALU with no dependencies

Decode the MULT instruction and send it to the appropriate ALU unit.

Joseph Liba - Create the MULT pipeline so that an ALU and MULT instruction can run simultaneously in different pipelines.

Derive the appropriate stall signal and deal with hazard resolution.

Design some out of order execution.

Overall waveform debugging.

Tianzhi Wu did not show up at all for part 2 of the project.

Decoding the MULT instructions

The decode stage is responsible for analyzing the instruction and determining what execution unit it has to send it to in the next cycle. In our case, it has to decide between the regular ALU and the MULT modules. To implement the correct checks in our code, we had to look at the RISC-V manual closely to decide what were the identification fields in the instruction that made the instruction types unique.

In the RISC-V Instruction Set, the MULT instructions have the same opcode as the conventional ALU register-register operations so we could not identify them by that field only. The bits in func3 are also the same as ADD and SUB so that was not a good solution either. Finally, we realized the func7 field was the only unique set of bits for the MULT instruction.

We had to write our own MULT testbech to make sure the MULT module performed correctly and handled dependencies stalling at the right times. One of the additional things we learned while writing the binary instructions is the naming convention that the RISC-V ISA uses for the registers. It was confusing at first because it does not follow the intuitive approach in which decimal 6 == binary 00110, but we quickly picked it up.

Implementation of the MULT Unit

The MULT unit uses higher order Verilog design. It sets MULT_result = A * B. In order to simulate the instruction taking 4 cycles, the MULT unit itself stalls for 3 cycles then calculates the result in the 4th instruction.

The MULT unit stores the inputs on the first cycle. That way, it will still know the inputs even when they are overwritten by a following ALU instruction.

The MULT unit also contains the MULT_stall signal. The MULT_stall indicates that a MULT instruction is still calculating. The MULT_stall signal is output by the execution stage and send to the stall_control unit.

Design of the MULT Pipeline

The MULT pipeline takes 2 cycles to complete rather than 1. This is so that it can pipeline the data directly from execute to writeback since there is no MULT memory instruction. However, because it pipes the data directly, you do not want an ALU instruction in the memory stage to intersect with a MULT instruction coming from the execute stage at the writeback stage. Therefore, MULT takes 2 cycles. In future implementation, one optimization could be to make the pipeline take 1 cycle and just check the timing so there is no real intersection. This could optimize multiple MULT instructions that occur in a row, where they could entirely skip over the memory stage.

The MULT pipeline runs simultaneously with the standard ALU pipeline. This way, data from an ALU instruction could go around the data in the MULT pipeline.

Deriving the stall signal

The standard stall signal was used, but in addition, a stall was added in the case that a MULT tries to run while a MULT is already calculating. So long as the stall works and data dependencies are checked, data hazards are resolved and instructions can execute out of order.