RPTU Kaiserslautern-Landau
Fachbereich Elektrotechnik und Informationstechnik
Entwurf informationstechnischer Systeme

Assignment 4 Pipelined RISC-V Core

The fourth assignment transforms the multi-cycle processor from task 3 into a 5-stage pipeline. The implemented ISA will stay the same simplified subset of RV32I as before. The pipeline features five stages that run in parallel, but is not (yet) equipped with a unit for hazard detection and resolution / prevention.

Structure of the Project

Before you start with the tasks, make yourself familiar with the directory for the fourth assignment (04_pipelined_RISC-V_core) in your forked repository. The Chisel project already contains a skeleton of the core. The actual implementation will be done in the core module (core.scala), while PipelinedRISCV32I.scala serves as a wrapper to connect the core tile to the outside world.

The wrapper interacts with the testbench in PipelinedRISCV32I_tb.scala. The testbench runs a simplified software program by passing the BinaryFile to the core. The core reads the content of the binary file into its instruction memory and executes it line by line. The test program in BinaryFile consists of 32-bit RISC-V assembly instructions encoded in HEX. BinaryFile_dump does not contribute to the testbench, but lists the assembly instructions in a human-readable way.
Remember, the testbench itself (PipelinedRISCV32I_tb.scala) only checks the expected results of the instructions and runs the clock. When adding new instructions to the test program, think of the result you expect from the core and add the corresponding check to the testbench.

Specification

The goal of this task is to implement a pipelined 5-stage 32-bit RISC-V processor supporting parts of the RV32I instruction set architecture. The RV32I core is still kept basic and does not include features like memory operations, exception handling, or branch instructions. It is designed for a simplified subset of the RISC-V ISA, mainly focusing on ALU operations and basic instruction execution.

This task is based on the core design implemented in task 3 and features the same characteristics. The only difference is that the processor stages are no longer called sequentially but act in parallel to process multiple instructions at a time. The execution flow stays the same as for the multi-cycle core:

1. Instructions are fetched from the instruction memory in the IF stage.
2. The fetched instruction is decoded in the ID stage, and the corresponding micro-operation code is determined.
3. The EX stage executes the operation using the operands.
4. The MEM stage does not perform any memory operations in this design.
5. The result is written back to the register file in the WB stage.

With different operations being processed concurrently, the pipeline stages need to keep track of which instruction they are currently processing. *Uopc* (enum data type) defines micro-operation codes representing ALU operations according to the RV32I subset used in the previous tasks. A pipeline stage can choose what operation to perform based on the *upoc* it receives. Compared to various boolean signals (e.g., *isADD*, *isSUB*, etc.) that would need to be stored and propagated, only one signal for the *uopc* is needed this way. The functional design of the individual stages can be reused from task 3, but each stage should now be implemented as a module. In addition to the previous assignment, barriers between the pipeline stages need to be implemented to store intermediate results and signals. An additional top module (*PipelinedRV32Icore*) is used to connect all pipeline stages, barriers and the register file. It interfaces with the external world through *check_res*, which is the result produced by the core.

Task 4.1 Preparation

Before you start with the implementation, please answer the following questions:

1. What effect does the implementation of the pipeline stages as individual modules have?
2. How does the interaction with the register file change, if it is implemented as a seperate module?
3. What would happen if you reuse the test cases from task 2 or task 3? How do you need to adapt these test cases to fit the pipelined implementation? What problems are arising now, that did not exist for the singe- and multi-cycle implementation?

Task 4.2 Implementation

Implement the pipelined 5-stage 32-bit RISC-V processor core according to the specification above by using the provided Chisel project.

Task 4.3 Test your Implementation

Check whether your design runs and produces correct results by adapting the test cases you created in the previous assignments. The test bench is located in PipelinedRISCV32I_tb.scala, test cases need to be added in HEX assembly to the "Binary File" in the "programs" folder.
A RISC-V encoder and a RISC-V interpreter might be helpful for you to achieve this task.