ADS I Class Project
Hardware Design with Chisel
Prof. Dr.-Ing. Wolfgang Kunz
M.Sc. Tobias Jauch

RPTU Kaiserslautern-Landau
Fachbereich Elektrotechnik und Informationstechnik
Entwurf informationstechnischer Systeme

Assignment 3 Multi-Cycle RISC-V Core

In our third assignment, the simple 32-bit RISC-V processor implemented in the previous task will be transformed into a multi-cycle core featuring five individual processor stages. The implemented ISA will stay the same simplified subset of RV32I as before.

Structure of the Project

Before you start with the tasks, make yourself familiar with the directory for the third assignment (03_multi-cycle_RISC-V_core) in your forked repository. As before, the Chisel project already contains a skeleton of the core. The actual implementation will be done in the core module (core.scala), while MultiCycleRISCV32I.scala serves as a wrapper to connect the core tile to the outside world.

The wrapper interacts with the testbench in MultiCycleRISCV32I_tb.scala. The testbench works similar to the testbench in assignment 2. It 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. To give you an example, four basic instructions are already listed there. BinaryFile_dump does not contribute to the testbench, but lists the assembly instructions in a human-readable way.
The testbench itself (MultiCycleRISCV32I_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 5-stage multi-cycle 32-bit RISC-V processor (without pipelining) supporting parts of the RV32I instruction set architecture. The RV32I core is relatively 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. It mainly focuses on ALU operations and basic instruction execution.

The CPU comes with an instruction memory (*IMem*) with 4096 words, each of 32 bits. The content of IMem is loaded from a binary file specified during the instantiation of the MultiCycleRV32Icore module. The CPU has a program counter (PC) and a register file (*regFile*) with 32 registers, each holding a 32-bit value. Register *x0* is hard-wired to zero. Signals are defined as either registers or wires depending on whether they need to be used in the same cycle or in a later cycle. The processor has five stages: fetch, decode, execute, memory, and writeback. The current stage is stored in a register named *stage*.

- **Fetch Stage:** The instruction is fetched from the instruction memory based on the current value of the program counter (PC).
- **Decode Stage:** Instruction fields such as *opcode*, *rd*, *funct3*, and *rs1* are extracted. For R-type instructions, additional fields like *funct7* and *rs2* are extracted. Control signals (*isADD*, *isSUB*, etc.) are set based on the *opcode* and *funct3* values. Operands (*operandA* and *operandB*) are determined based on the instruction type.
- **Execute Stage:** Arithmetic and logic operations are performed based on the control signals and operands. The result is stored in the *aluResult* register.
- **Memory Stage:** No memory operations are implemented in this basic CPU.
- **Writeback Stage:** The result of the operation (*writeBackData*) is written back to the destination register (rd) in the register file. The program counter (PC) is updated for the next instruction.

If the processor state is not in any of the defined stages, an assertion is triggered to indicate an error. The final result (*writeBackData*) is output to the *io.check_res* signal. In the fetch stage, a default value of 0 is assigned to *io.check_res*.

Task 3.1 Preparation

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

1. How can you implement a finite state machine in Chisel?
2. Compared to your single-cycle implementation from task 2, what needs to be changed in terms of the signal's data types? What can stay the same?
3. What would happen if you reuse the test cases from task 2? How do you need to adapt them to fit the multi-cycle implementation?

Task 3.2 Implementation

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

Task 3.3 Test your Implementation

Check whether your design runs and produces correct results by adapting the test cases you created for the single-cycle RISC-V core to the multi-cycle core. The test bench is located in MultiCycleRISCV32I_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.