RPTU Kaiserslautern-Landau
Fachbereich Elektrotechnik und Informationstechnik
Entwurf informationstechnischer Systeme

Assignment 2 Single-Cycle RISC-V Core

The goal of our second assignment is to implement a simple 32-bit RISC-V processor supporting parts of the RV32I instruction set architecture. This 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.

Structure of the Project

Before you start with the tasks, make yourself familiar with the directory for the second assignment (02_single-cycle_RISC-V_core) in your forked reository. The Chisel project already contains a skeleton of the core. The actual implementation will be done in the core module (core.scala), while SimpleRISCV32I.scala serves as a wrapper to connect the core tile to the outside world.

The wrapper interacts with the testbench in SimpleRISCV32I_tb.scala. The testbench runs a simplified software program on the core, 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 (SimpleRISCV32I_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.

How to Start

This assignment includes two main tasks: Completing the implementation of the simplified RISC-V core and extending the test program and test bench to validate your implementation. In principle, the order in which you work through these two tasks does not matter. To some people, it feels more intuitive to create the design first and test it afterwards. However, this can lead to writing tests under the influence of your own implementation. If you have misunderstood the task or misinterpreted the specification, it is likely that you will also design the tests that way. In the worst case, all tests might pass although the implementation is faulty.
To avoid this, you can reverse the order: First, the tests are designed based on the specification, then the design is implemented. This way, the design can be already be tested during the design process. If all tests pass, the design is finished. This method is called "test-driven development".
However, the best method is to have the tests and implementation designed by different people. This reduces the influence of the user's own interpretation of the specification and ensures that the design is checked by at least one other person.

Specification

The goal of this task is to implement a simple 32-bit RISC-V processor supporting parts of the RV32I instruction set architecture. This 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. Your final design should be able to process all R-type instructions and the "ADDI" instruction (I-type) from the RISC-V 32 bit "I" instruction set.

The core initializes an instruction memory "mem" with a capacity of 4096 32-bit words. It loads the content of this memory from a binary file specified by the BinaryFile parameter. The core also initializes a program counter (PC) register "PC" with an initial value of 0. Moreover, it contains a register file "regFile" with 32 entries. Register x0 is hard-wired to always contain the value "0".

The Fetch Stage reads the current instruction from "mem" based on the value of the program counter PC. The Decode Stage extracts various fields (opcode, rd, funct3, rs1, funct7, and rs2) from the fetched instruction, according to the RISC-V ISA specification. The core determines the operands for ALU operations based on the instruction type and stores them into two variables "operandA" and "operandB". For R-Type instructions, both operands are read from regFile. For I-Type instructions, operandA is read from regFile, while operandB is an immediate value, encoded in the instruction. The code defines a set of boolean signals (isADD, isADDI, etc.) to identify the type of instruction based on the opcode and function fields. These signals are then used to determine the operation to be performed in the Execute Stage. In the Execute Stage, ALU operations are performed based on the instruction type and the operands. The result is stored in "aluResult".
Note that some instructions (e.g., arithmetic shifts or comparations) only make sense when performed on signed integer numbers. Chisel operators that can be used to perform various arithmetic operations can be found here, along with their explanation.
As the core does not include memory operations, the Memory Stage can be left empty. The result of the ALU operation ("aluResult") is propagated to "writeBackData" in the Write Back Stage. The code writes the value contained ba "writeBackData" to the destination register in the regFile specified by "rd". The result of the ALU operation is also provided as output to "io.check_res". This signal will be used by our test cases to validate the actions performed by our core. In the end, the program counter PC is incremented by 4 in every cycle, as this core does not include jumps or branches. It simply fetches instructions sequentially.

Task 2.1 Preparation

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

1. Which specific instructions do you need to implement?
2. How is the "NOP" (no operation) instruction implementen in a RISC-V core?
3. How can you read from and write to the register file?

Task 2.2 Implementation

Implement a simple RV32I core according to the specification above by using the provided Chisel project.

Task 2.3 Test your Implementation

Check whether your design runs and produces correct results by writing test cases for every instruction you implemented. The test bench (SimpleRISCV32I_tb.scala) already contains a basis that needs to be extended to cover all instructions. While the scala testbench only checks the results that your "check_res" output provides, new test cases need to be added to the "Binary File" in HEX assembly. A RISC-V encoder and a RISC-V interpreter might be helpful for you to achieve this task.