UNIVERSITÄT DES SAARLANDES

Prof. Dr. Jan Reineke

System Architecture SS 2024 – Project I

Hardware Design with Chisel

Project Modalities

The project starts on June 07, 2024 with the release of this description. It consists of two parts:

- In the first part, you will implement arithmetic circuits in Chisel.
- In the second part, you will extend a given RISC-V processor implementation in Chisel.

You may work on the project in *groups of two to three people*. If you have formed a group, please create a corresponding team on your personal page in our CMS until

Wednesday, June 12, 2024, 23:59.

One person per group must submit the group's solution of the project to our CMS system by

Friday, June 28, 2024, 23:59.

Follow the instructions in the next section to generate your submission. A total of 32 points (plus 6 bonus points) can be achieved. Projects submitted late will be graded with 0 points. Submissions that do not pass our sanitizer tests (details below) will also be graded with 0 points.

Tools and Setup

The skeleton files for the project are available at the following public repository:

`https://gitlab.cs.uni-saarland.de/reineke/sysarch-project-one`.

Simply clone the repository to get started. We do recommend using a version control system like `git` to manage your project within your team. For instance, you can use the department's GitLab server to host your repository.

To work on the project you will need *Java* and *Scala*, including *sbt* ("Simple Build Tool"). To install all of these, follow the instructions at `www.scala-lang.org/download/`. This should leave you with a working Scala environment, including *sbt*, and if not already installed, *Java*. More information on how to install *sbt* can also be found on the *sbt* website. Being able to run `sbt -script-version` in your terminal is a good indicator that *sbt* is installed correctly.

sbt is used to compile the project, run tests, and even to create the zip file to be submitted to the CMS. The following table gives an overview of the most important *sbt* commands in the context of this project:

Command	Description
`sbt compile`	Compile the project.
`sbt clean`	Clean the project.
`sbt test`	Run all the tests in the project.
`sbt "testOnly *TestName*"`	Run a specific test (quotation marks necessary).
`sbt check`	Run the sanitizer tests.
`sbt createSubmission`	Create a submission file (inside folder `submissions`).
`sbt checkAndSubmission`	Run the sanitizer tests and create a submission file if they pass.

We provide you with a basic set of tests, which can be found in the `src/test/scala` folder. Some of these already work with the skeleton code, while other tests will only pass once you have implemented the corresponding functionality. We do recommend to create additional tests to ensure the correctness of your implementation. You can add such tests to `src/test/scala`.

To test your processor on RISC-V assembly programs, we provide you with a program loader. Familiarize yourself with the existing program descriptions in the `src/test/resources` folder. You can add your own programs in that folder and create corresponding tests in `ProcessorTest.scala` using the `runProgram` function[1]. Feel free to add your own tests to the existing test files, since the tests will not be part of the submission.

We recommend using VSCode with the *Scala* extension and the Metals language server for development. Therefore we included a `.vscode` folder in the skeleton files. This includes an `extensions.json` file that will suggest to install the necessary extensions when you open the project in VSCode.

Submission

Run `sbt check` to check if your project passes the sanitizer tests. You should only submit your project if it does! Create a submission file by running `sbt createSubmission`. The submission file will be created in the `submissions` folder. Upload this file without further modifications in the CMS.

Collaboration Policy

You may work on the project in groups of up to three people. Briefly describe how each team member contributed to the project in the file `CONTRIBUTIONS.md`. There, also include a brief statement whether you have used any AI support for the project, and if so, which tools were used. We may ask individual team members to explain their contributions in more detail orally. We also expect that all team members have a good understanding of the entire project. We may grade the project with 0 points for individual members if they have not made a significant contribution or do not have a reasonable understanding of the project as a whole.

The work you hand in as a team must be your own. Copying another group's work or allowing your work to be copied by others is a serious academic offense and will be treated as such.

We will check all submissions for plagiarism (against submissions from other groups, as well as submissions from previous years). Submissions that were created by modifying another project, for example by changing variable names, are also considered to be plagiarized. Plagiarized submissions will be graded with 0 points and will be reported to the examination board as a cheating attempt; this may lead to expulsion from the university.

Examples of permitted collaboration:

- Talk to other groups about how to complete the project **at a high level**.
- Ask a tutor for help if you are confused or stuck.

Examples of collaboration that is not permitted:

- Help another group debug their code by bringing up your own solution and comparing the two.
- Using a friend's code from a previous year as a starting point and then modifying it.
- Copying any portion of another group's code even if you modify it.
- Sharing portions of your code with another group.

Questions and Problems

If you encounter any issues while working on the project that you cannot solve in your group, we will be happy to help you in the forum or during the office hours. In general, however, we expect that you have already done some work on your own (own tests, debugging outputs, online research, etc.) to solve the issue. Non-specific questions like "Does the code compile?", "Is this correct?", or "What needs to be done in problem X.Y?" will not be answered.

[1] Note that the provided assembler does not support labels, and so you will need to provide the address offsets directly in the assembly code.

Arithmetic Circuits in Chisel (10 (+2) Points)

The skeleton files for this part of the project are in the folder `src/main/scala/arithmetic` and the corresponding tests are in the folder `src/test/scala/arithmetic`.

Problem 1.1: Division Circuit (5 Points)

In the lectures, we have seen circuits for addition, subtraction and multiplication, but not for division. Integer division of two n-bit unsigned binary numbers A and B can be implemented as a *sequential circuit*, based on the grade school division method (`https://en.wikipedia.org/wiki/Long_division`). The division $\frac{\langle A \rangle}{\langle B \rangle}$ according to the school method can be expressed algorithmically as follows:

```
for i = n-1 to 0
  R' = 2 * R + A[i]
  if (R' < B) then Q[i] = 0, R = R'
              else Q[i] = 1, R = R'-B
```

At the end of the loop, Q holds the quotient and R the remainder of the division. Check what happens if $\langle B \rangle = 0$.

To get started, manually compute $\frac{7}{3}$ using this algorithm with pen and paper.

Now design the corresponding sequential circuit based on the provided skeleton. The interface of your circuit is given in the skeleton and it should not be changed: The division circuit has two n-bit inputs `dividend` and `divisor`, a boolean input `start`, an implicit clock input, two n-bit outputs `quotient` and `remainder`, and a boolean output `done`.

The computation should start when `start` is set to true. n clock cycles later the outputs `quotient` and `remainder` should contain the correct results and `done` should be set to true until the next division is initiated. Whenever `start` $= 1$ occurs the circuit should start over with the current input values. The circuit should ignore changes to inputs other than `start` after a computation has been started. In the following section, we provide some additional guidance.

Notes The sequential circuit shall perform *one* iteration of the loop in the pseudo code per cycle. Multiplication in hardware is expensive. Therefore, express the multiplication by a cheaper shift operation. You may use the built-in Chisel operators for subtraction (-) and comparison (<).

As suggested in the skeleton, use three n-bit registers `remainder`, `quotient`, and `divisor` to store the current state of the sequential circuit: Register `remainder` should store the current value of the remainder. Register `quotient` should store the remaining bits of the input dividend ($A[i, 0]$ in terms of the pseudo code above) and the computed bits of the quotient so far ($Q[n-1, i+1]$ in terms of the pseudo code above). Register `divisor` should store the divisor. You will need additional registers to keep track of the timing of the computation.

You can use the provided divider test to test your solution. We will use additional tests internally to evaluate your submission that cover other cases.

Problem 1.2: Bonus: Faster Division (2 Bonus Points)

In some cases, the correct result of the division can be determined in fewer than n cycles. Identify conditions that can be checked cheaply in a circuit under which the loop in the division algorithm can be terminated early.

Based on your findings, extend your implementation from Problem 1.1 to perform the division in fewer cycles.

If you want to get the bonus points, please specify in the file `CONTRIBUTIONS.md` which conditions you have identified and how you have implemented them in your circuit.

Problem 1.3: Vector Unit (5 Points)

Modern processors often have vector units that can perform the same arithmetic or logic operation on multiple data elements in parallel. Your task is to implement a generator that creates a vector unit for a given scalar operation.

More specifically, you should implement a generator `ParallelUnit` that takes the following parameters:

- `arraySize`: the number of parallel units in the vector unit.
- `vectorSize`: the number of elements in the input and output vectors.
  You can assume that `vectorSize` is a multiple of `arraySize`.
- `unitWidth`: the bit width of the input and output elements.
- `comp`: a function that creates a computational unit for the scalar operation.

The interface of a computational unit is given by the abstract class `ComputationalUnit` in the skeleton file.

We are providing a skeleton for the `ParallelUnit`, which fixes the interface of the module. The parallel unit should instantiate `arraySize` many computational units, which is already done in the skeleton.

These computational units should be used to perform the scalar operation on all vectorSize many elements of the input vector in as few cycles as possible. For example, if `vectorSize` is equal to `arraySize`, then the outputs should be computed after a single cycle delay (i.e., *not* in the same cycle). In general, the outputs should be provided at the outputs after `vectorSize/arraySize` cycles. Completion should be indicated by the `done` signal, while the `start` signal should initiate the computation (potentially aborting ongoing computations).

RISC-V Processor (22 (+4) Points)

The skeleton files for this part of the project are in the folder `src/main/scala/RISC-V` and the corresponding tests are in the folder `src/test/scala/RISC-V`.

It is advisable to write your own tests to check that your solutions to the following problems work. Your tests will not be used for grading.

Problem 2.1: Modular RISC-V Implementation (0 Points)

Familiarize yourself with the (almost complete) Chisel implementation of a RISC-V processor and the associated test infrastructure. The machine so far supports most of the base instruction set RV32I with the exception of the load instructions and unconditional jumps, which you are supposed to add in Problems 2.2 and 2.3.

Even though this task doesn't give you any points, take your time: once you have understood the structure of the design, it will be much easier to work on the following problems. Please read the last section which gives an overview of the processor.

We will frequently refer to the RISC-V specification in the following problems. It is available at `https://riscv.org/specifications/` ("Volume 1: Unprivileged Specification"). An overview of instructions encodings is given in Chapter 34.

Problem 2.2: Function Calls (5 Points)

To support function calls, RISC-V has two types of unconditional jumps:

- *jump and link* (`jal rd, imm`)
- *jump and link register* (`jalr rd, rs1, imm`)

Look up the encoding and operation of these instructions in Chapter 2.5.1 of the RISC-V specification and implement them by appropriately adapting the decoder, the control unit, and the RV32I execution unit. In contrast to the specification, your implementation does not need to generate instruction-address-misaligned exceptions.

Problem 2.3: Load Instructions (6 Points)

The processor currently does not support any load instructions. The base instruction set RV32I includes the following load instructions: *load word* (`lw`), *load byte* (`lb`), *load byte unsigned* (`lbu`), *load halfword* (`lh`), and *load halfword unsigned* (`lhu`).

Look up their encoding and operation in Chapter 2.6 of the RISC-V specification and implement the instructions by appropriately adapting the provided implementation.

Problem 2.4: Multiplication and Division Instructions (5 + 6 Points)

The RISC-V M-Extension adds instructions for multiplication and division. Your task is to implement this extension by appropriately adapting the skeleton given in `MultiplicationUnit.scala` and `DivisionUnit.scala`. You can use the Chisel operators for multiplication and division, i.e., you do not need to implement custom arithmetic circuits for these operations.

Multiplication

Implement the multiplication instructions `mul`, `mulh`, `mulhsu`, `mulhu`.
See Chapter 13.1 of the RISC-V specification for details.

Division

Implement the division instructions `div`, `divu`, `rem`, `remu`.
See Chapter 13.2 of the RISC-V specification for details.

Problem 2.5: Bonus: Division (4 Bonus Points)

Use your implementation of the division circuit from Problem 1.1 to implement the division instructions from the previous problem. As the division circuit is not combinatorial, you will need to set appropriate stall signals to ensure that the processor waits for the division to complete before proceeding. For this problem adapt the skeleton given in `BonusDivisionUnit.scala` so that your solution to Problem 2.4 stays intact.

Overview of the RISC-V Processor

In this section you will find a brief overview of the RISC-V processor that you will be working on in the second part of the project.

The processor is designed to be very modular and easy to extend. While it is similar to the one we discussed in the lecture, there are some significant differences in its implementation.

Chisel Enums

We have defined some enums in the skeleton code to make the implementation easier to read and maintain. Notably, there is `RISCV_TYPE` which defines the type of the instruction, you can obtain it by concatenating the `opcode`, `funct3` and `funct7` fields of the instruction. You can obtain a type's components by using the `.getOP`, `.getFunct3` and `.getFunct7` methods. You should not need to change anything here, but you might want to use the enums in your implementation.

There are also `ALU_CONTROL`, which defines the operation of the ALU, `ALU_OP_1_SEL` and `ALU_OP_2_SEL`, which define the sources of the operands for the ALU, `REG_WRITE_SEL`, which defines the destination register for writeback, and `NEXT_PC_SELECT`, which defines the source of the next program counter value.

The Execution Units

The processor has multiple execution units, each responsible for a subset of the instructions. A processor might have multiple execution units, but each of them should implement a disjoint set of instructions. An execution unit can "claim" an instruction by setting the `valid` signal. This indicates that this execution unit will execute the instruction. If the instruction needs multiple cycles, the execution unit should use the `stall` signal `STALL_REASON.EXECUTION_UNIT` to indicate that the instruction has not yet been completed. Once the stall output is set to `STALL_REASON.NO_STALL`, the top level will proceed to the next instruction.

The skeleton provides a generic implementation of a processor that takes a list of execution unit generators, instantiates the execution units and selects the "correct" execution unit for each instruction, i.e. the execution unit that claims the instruction.

You can find the interface of the execution units in Figure 1.

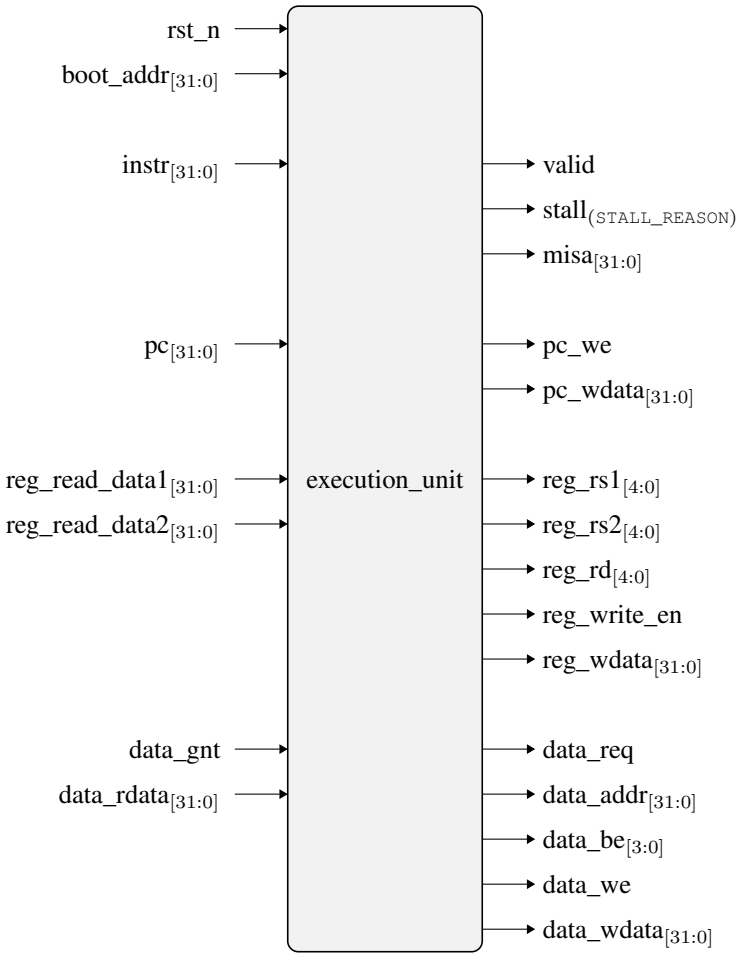


Figure 1: Execution unit interface

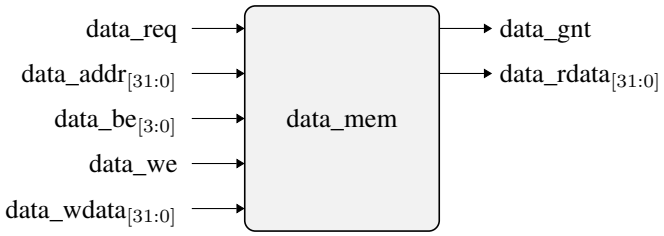


Figure 2: Data memory interface

The Memory Interface

The instruction and data memory are not directly implemented in Chisel but emulated during the test. This also means that these memories are not combinatorial but operate based on requests, which introduces at least one cycle delay for each memory access. In real systems the memory hierarchy is much more complex and can introduce multiple cycles of delay for each memory access.

The emulated memory has a little-endian byte order, i.e. the least significant byte of a word is stored at the lowest address.

Generally speaking the processor sets all parameters of the request (i.e. the address, which byte to read/write, etc.) and sets the `req` signal to indicate a new request. The memory then processes the request and eventually responds with a `gnt` signal to indicate that the request has been granted. The processor can then read the data from the memory.

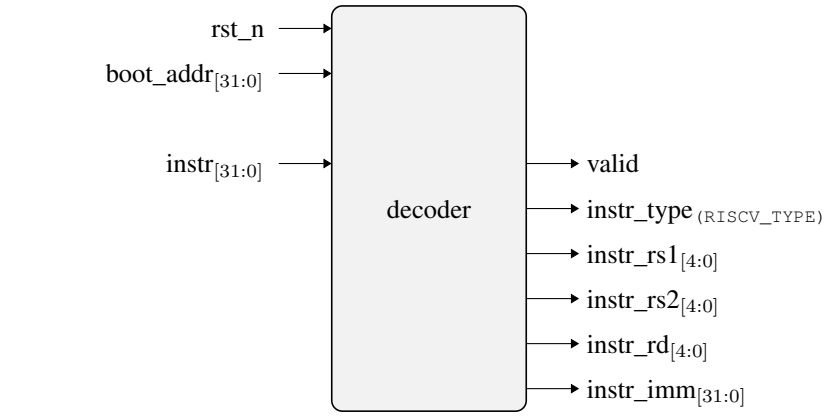
When implementing the load requests, you might want to have a look at how the store requests are implemented, especially how they stall the core until the store request has been completed.

In Figure 2 you can see the interface of the data memory.

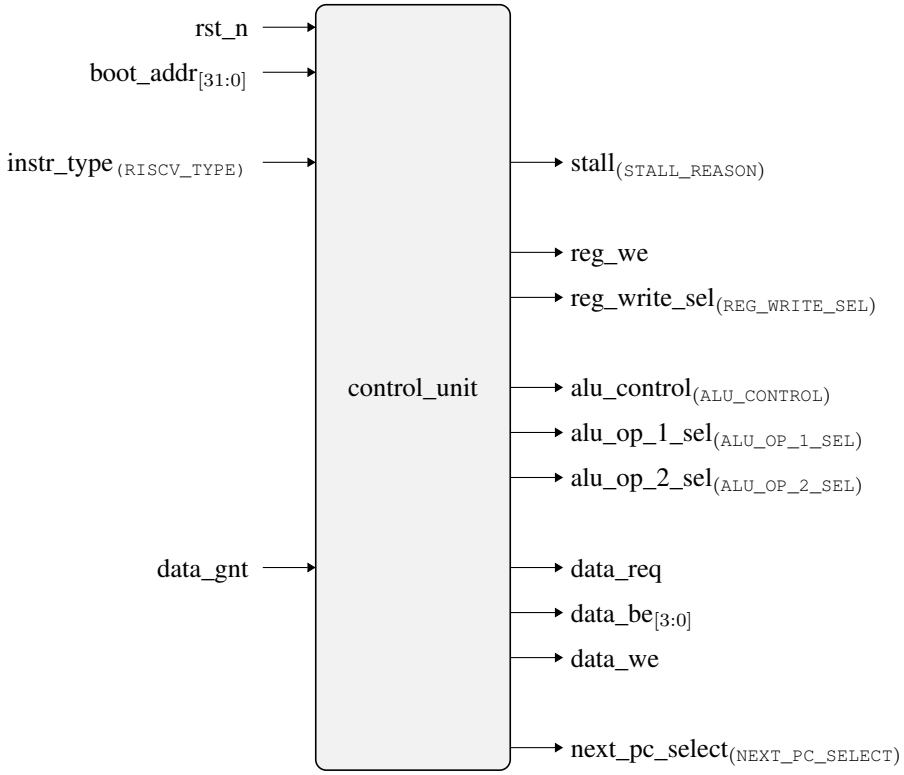
The instruction memory is similar to the data memory but only supports read requests. In this project, you should not need to query the instruction memory directly, the skeleton code already implements the instruction fetch and provides the correct instruction to the execution unit.

The `RV32I` Module

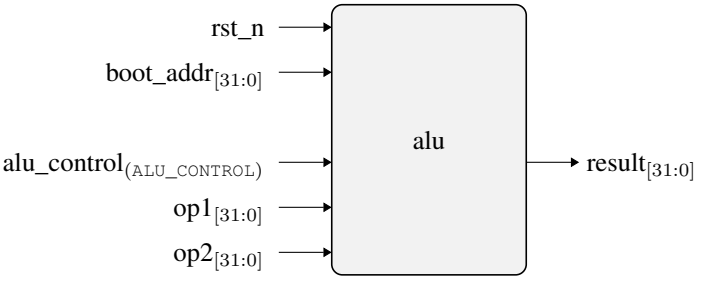
The `RV32I` module implements the base RISC-V ISA. It has several modules which you already know from the lecture. You can see their interfaces in Figure 3. As you can see, all modules have a `rst_n` signal to reset the module (active low) and the `boot_addr` signal corresponds to the program counter after a reset. If you add new registers to a module, make sure to reset them on a reset signal. For Problems 2.2 and 2.3 you will need to adapt those modules, however you should not need to change the interfaces of the modules.



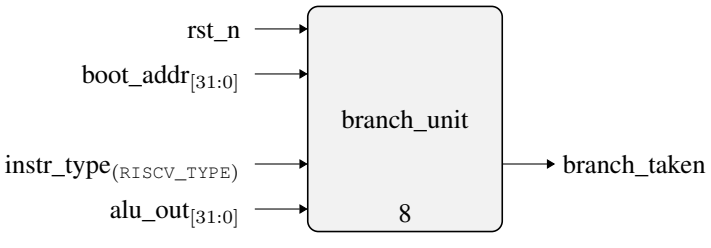
(a) Decoder



(b) Control Unit



(c) ALU



(d) Branch Unit

Figure 3: Modules of the RV32I processor