Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Electrical and Computer Engineering

ADS I Class Project

Hardware Design with Chisel, an object-oriented HDL

Chair of Electronic Design Automation

Entwurf informationstechnischer Systeme

Prof. Dr.-Ing. Wolfgang Kunz
M.Sc. Tobias Jauch

Contents

1 General Information

Our lectures Architecture of Digital Systems I (EIT-EIS-571-V-4) and Architecture of Digital Systems II (EIT-EIS-573-V-4), deal with the basic principles of computer architecture for CPU cores and SoCs. We offer this class project as an add-on to ADS I to provide students a chance to deepen their knowledge and understanding of these topics and to gain first hands-on experience in designing hardware systems in Chisel.

2 Chisel: A modern Hardware Description Language

Chisel (Constructing Hardware in a Scala Embedded Language) is an open-source HDL based on the programming language Scala. It is used to describe digital systems at the register-transfer level (RTL) and is therefore on the same abstraction level as VHDL and (System) Verilog.
The advantages of Chisel over other HDLs are its object-oriented and functional programming characteristics inherited from Scala. This allows a modern coding style and better IDE support compared to VHDL or System Verilog. Additionally, while VHDL and Verilog were introduced as hardware simulators, Chisel is a hardware generator intended to synthesize hardware designs. In general, a Chisel design is a scala program describing how to build a specific Chisel graph that represents the hardware design. During the compilation and generation of the Chisel graph, several checks are done to guarantee a working circuit. The checks include syntax, invalid circuit designs, unconnected wires, etc.
Afterward, the behavior of the designed circuit can be simulated with FIRRTL (Flexible Internal Representation for RTL). Setting inputs and analyzing the circuit's outputs in simulation can be used to test and validate the behavior of the design. Unlike other hardware description languages, Chisel is doing checks whether the design is synthesizable before the simulation is done, during the generation of the Chisel graph. The intermediate representation in FIRRTL can later be translated into System Verilog code which enables FPGA emulation of the circuit design and even the construction of ASICs.

3 Getting Started

Prerequisites

To take this course you should have a basic knowledge of digital circuits. Here at RPTU, undergrad courses teaching those topics are for example Grundlagen der Informationsverarbeitung and Labor Digitaltechnik I. If you got your Bachelor's degree from another university, you probably took similar courses.

Prepare your Workspace

In general, it is best to use a machine running Linux Ubuntu 20.04 or newer for this course. Other Linux distributions, Windows (with Linux subsystem) or macOS might be suitable as

well, but if you want to use them you need to figure out how to install and build everything for yourself, if the provided instructions don't work.

An Editor / IDE for Chisel

Chisel designs can be created in any editor (nano, Vim, etc.) For larger projects, it can be helpful to use an integrated development environment (IDE) as an editor for coding. IDEs provide helpful functions like the integration of version control systems (e.g. Git) or syntax highlighting and other helpful features to keep a better overview over big code bases.

In case you already have worked with an IDE that you favor, feel free to use it. If this area is new to you, you can either start with a basic text editor or try using IntelliJ. IntelliJ is a professional IDE that also provides support for Chisel. As a student, you can get a free license here.

Java / Scala / sbt

First of all, as Scala (the underlying programming language of Chisel) uses the java virtual machine (JVM), your workspace must be able to run java. Afterward, you need some more tools like the Scala compiler (scalac), the Scala build tool (sbt), and some others, to use all features of Chisel. Luckily, Scala provides all of these tools in one combined script. You just need to execute the following command in your terminal (on Linux) and the script will automatically check if you have some of the tools already present and install the missing ones.

```
$ curl -fL https://github.com/coursier/launchers/raw/master/cs-x86_64-pc-linux.
    gz | gzip -d > cs && chmod +x cs && ./cs setup
```

If you need more information, just check the official scala install page.

4 Structure of a Chisel Project

Every Chisel (and Scala) project needs is organized in the following structure:

```
- <project name>
    - project (sbt uses this for its own files)
        - build.properties
    - build.sbt (sbt's build definition file)
    - src
        - main
            - scala (main code of your design)
                - <file>.scala
        - test
            - scala (testcases for your design)
                - <testcase>.scala
```

This structure is provided for each task of the class project as the main focus is on designing the hardware. In case you want to set up a Chisel or Scala project on your own, you can use the command

```
$ sbt new scala/scala3.g8
```

and enter a project name when prompted. An empty structure as shown above is then generated at the location you executed the command.

5 Chisel Basics

First of all, we need to make an important distinction between Scala and Chisel: While Scala is a programming language with abstract concepts, Chisel represents synthesizable hardware circuits. To achive this, Chisel uses several concepts that are embedded into the Scala programming language, but differ quite significantly when compared to Scala's corresponding concepts.

Data Types

In general, Chisel deals with three data types: **UInt**, **SInt**, and **Bool**. In contrast, pure Scala uses Int, Boolean and other types. Note that only Chisel data types are suitable to describe hardware! The underlying reason is that a Chisel type "Bool" is a reference to a wire that always holds the value "1", i.e. that is connected to the V_{DD} plane in a real hardware circuit. A Scala variable of type "Boolean" is just a theoretical concept of the boolean values "true" or "false" and therefore differs significantly from its Chisel counterpart.

```
1.U     // unsigned integer with value "1" (uses 1-bit)
8.U     // unsigned integer with value "8" (uses 4-bits)

5.S     // signed integer with value "5" (uses 4-bits)
-8.S    // signed integer with value "-8" (uses 4-bits)

true.B  // boolean literals
false.B
```

Wire

A wire in a real-world circuit is an electrical connection that is able to carry a current or voltage level but cannot store it in any way. If the driving power source of the wire changes, the state of the wire also changes immediately. In Chisel, the datatype "Wire" refers to a bundle of connections which can be defined in number and interpreted data format (e.g. unsigned integer). As these bundles later refer to physical wires that can only carry two distinctive voltage levels to represent a logical "1" or "0", we need as many wires in a bundle as we need bits to represent the data we want to transfer. For example, if we want to work with unsigned integer numbers between 0 and 15, a width of four connections in the wire bundle will be suffcient:

```
val signal = Wire(UInt(4.W))
```

Wires without further connections are useless, as they only act as "transmitters" of binary encoded information. Therefore, we need drivers that put these information onto the wires. In the following code snippet, a hardwired value of "3" is connected to signalA, while signalB is then driven by signalA.

```
val signalA = Wire(UInt(2.W))
val signalB = Wire(UInt(2.W))

signalA := 3.U
signalB := signalA
```

To keep the circuit sythesizable, a wire should not be driven by multiple sources at a time and circular dependencies should also never occur. Otherwiese, the resulting information on the wire is undefined, leading to unwanted behaviour (or simply an error during the compilation of your Chisel code).

Combinatory vs. Sequential Circuits

If only wires are used in a design, we get a **combinatory circuit**. This means that the output of the circuit only depends in its current inputs. If we also take previous events into account to calculate the output, we get a **sequential circuit**. Sequential circuits need some kind of storage element, to be able to save previous results or states. The storage elements are called registers. Physically, registers are built using flip-flops (also called latches).

Register

In Chisel, registers are used quite simlar to wires. They need to be specified and provided with a datatype and width:

```
val register = Reg(UInt(4.W))
```

As registers store and keep their assigned value, it is also possible to initialize them with a predefined value. In this case, the datatype in the register is implicitly derived from the initial value, e.g. "10.U" denotes a register of type "UInt":

```
val register = RegInit(10.U(4.W))
```

Registers can be used to drive wires or other registers. The important difference compared to wires is that a register updates its value with the next clock tick and not immediately. In contrast to wires, this allows circulary connections, for example two registers swapping values every clock cycle:

```
val registerA = RegInit(1.U(2.W))
val registerB = RegInit(3.U(2.W))

regA := regB
regB := regA
```

Clock

Digital circuits are driven by clock ticks that denote at which points in time things change in a circuit. The time between two clock ticks is defined as a clock cycle. While VHDL designs need an explicit incorporation of the clock signal. Chisel design handle the clock implicitly, so you don’t care about this topic much in the beginning.

Conditional Statements

Hardware designs usually contain parts that behave different based on an input, internal state or value. To implement conditional execution in Chisel, the "when"-statement is used.

```
when(condition1)
{
  ... // execute if condition1 is fulfilled, skip rest
} .elsewhen(condition2) {
  ... // execute if condition 1 is false and condition2 is fulfilled, skip rest
} .otherwise {
  ... // execute if none of the conditions above is fulfilled
}
```

In addition, a "switch"-statement decides which action to take based on data instead of boolean conditions.

```
switch(variable){
  is(value_a) {
    ... // execute if variable holds value_a
  } is(value_b) {
    ... // execute if variable holds value_b
  ...
}
```

Pitfalls: Chisel vs. Scala

Many things that are totally legitimate in Scala can lead to problems when they are used to describe a Chisel circuit. The main reason for these problems is that Chisel, as exlained earlier, is just a way to generate a graph from a Scala code that can be sythesized to an transistor-level circuit.

val vs. var

In pure Scala, val and var both denote variables. While val is immutable (i.e. it can not be changed after its creation), var is mutable (and can also be changed after being created). Unless you know exactly what you are doing, you should only use val when creating new signals or registers in Chisel designs.

```
val signal = Wire(UInt(4.W))
```

A Scala variable like *signal* points to a Chisel node in the graph representation. This node is a wire of type UInt. *signal* should not change the Chisel node it points to, to avoid confusion and unexpected behavior! If you use var instead of val, it is possible to (accidentially) change the pointer to a new node:

```
var signal = io.a
signal := true.B
signal = io.b
signal := false.B
```

In this example, *signal* first points to node *io.a* and is then assigned the boolean value true.B. The third line changes *signal* to point to *io.b* which is then set to false.B.
In Verilog, this example would look like this:

```
module Example(
    output io_a,
    output io_b
);

  assign io_a = 1'h1;
  assign io_b = 1'h0;
endmodule
```

As you can see, using *var* might work, but it is very hard to predict the output on RTL, even in very small designs. In the context of this course, only stick to *val* when creating new signals!

= vs. :=

In the examples above, you may already have noticed that two different "equal"-signs are used. While "=" denotes a Scala assignment that makes a variable point to a certain note of the Chisel graph, ":=" assignes a new value to the Chisel node that the variable is pointing to. Mixing up these two different assignments can be dangerous and may result in unexpected behavior. If you are assigning values to Chisel variables based on conditions (i.e. when-statements), "=" assignments are ignoring the condition and will be assigned in any case.

Making things easier for yourself

We will cover "real" debugging later in this course, but there are some measures that you can already use while writing the code to make things easier for yourself.

Comments

Everything that is not completely obvious (and that's more than you would think) should be explained in a short comment. Code is meant to be read, not only written! Even your own code can be confusing once you didn't look at it for a while.

```
val signal = Wire(UInt(4.W)) // this is a comment explaining what "signal" does
```

Assertions

Assertions can help you to find the rootcause of bugs in your system. It is recommended that after every bug you found and fixed, you add an assertion that would have caught this bug. If an assertion is not fulfilled, it will trigger the simulation to throw an error and print a specified message. In large code bases, adding the module name can help to locate the failing assertion faster.

```
when (write_to_queue)
{
  queue(idx).valid := io.valid
  queue(idx).data  := io.data
```

```
  assert( !(write_to_queue && queue(idx).valid), "[Module Name] Valid queue
      entry gets overwritten")
}
```

Requires

Chisel also provides a require() function that throws a run-time error if the condition is not met during compilation. Parameter values can be guarded by require() statements, but it may also be used to codify your own assumptions. This is useful, when your design only acts correct for specific values, e.g. a power of two.

```
require(isPow2(num_entries)) // number of entries in the queue should be 2^x
```

6 Starting the Class Project

Now it is time for some hands-on practice! For this class project, we provide a GitHub repository with the basic structure of every task and all the documentation you need:

- ADS I Class Project GitHub Repo

6.1 Version Control with Git

Git is the most common version control system in the world today. GitHub, on the other hand, is a cloud-based service that offers a platform for storing and managing projects using Git. Generally, working with GitHub is based on some simple principles. There are some very complex commands, too, but you usually won't need them in your everyday work with Git. To install Git, you can use the following command:

```
$ sudo apt install git-all
```

To avoid confusion, this document will only talk about the exact steps that you need for the class project. Additional information can be found online, for example, here: Introduction to Git

Step 1: Forking the Main Repository

- Go to the GitHub page of the repository (ADSProject).
- Click on the "Fork" button at the top-right corner of the page.
- This will create a copy of the repository under your GitHub account.

Step 2: Cloning the Forked Repository

- Open a terminal or command prompt on your local machine.
- Navigate to the directory where you want to clone the repository.

- Run the following command, replacing <your-account> with the name of your GitHub account:

```
$ git clone https://github.com/<your-account>/ADSProject
```

Step 3: Creating a New Branch

- Change into the cloned repository directory.
- Run the following command to create a new branch that starts from the exact state as the main branch you're currently on.
- Replace 'XXX' with your group number:

```
$ git checkout -b <GroupXXX>
```

Step 4: Pushing the New Branch to your Fork on GitHub

- Your forked repository on GitHub doesn't know about the new branch you created, yet.
- Push the new branch to your repository with the following command:

```
$ git push --set-upstream origin <your branch name>
```

Step 5: Pushing Changes to the New Branch

- Make the desired changes to the files in your local repository.
- Run the following commands to stage and commit your changes:

```
$ git add <file-you-want-to-add>  // "add ." will add all new files
$ git commit -a -m "Your commit message"
```

- Finally, push your changes to your repository using the following command:

```
$ git push
```

Whenever you want to add, commit and push changes to your repo, you can do this by repeating step 5. If multiple people work in the same directory, you should always pull the changes other people pushed recently to avoid conflicts (e.g., by editing the same region of the same file simultaneously). Getting the latest state is simply done by using *git pull*:

```
$ git pull
```

That's it! You have now successfully forked a repository, cloned it, created a new branch, and pushed your changes to the new branch in your own repository on GitHub.

6.2 Running the Tasks

The repository contains a directory for each assignment, e.g., "01_warm-up". Each assignment already contains a basic Chisel project following the structure described in section 4. Navigate to the warm-up directory on your local clone and open a terminal. To generate or run a Chisel project, you need two commands:

```
$ sbt run
```

generates the System Verilog code for your Chisel design and stores it in the folder "generated-src".

```
$ sbt test
```

runs the test cases located in "src/test/scala" on the design. The terminal output shows, whether the tests passed successfully or failed. Additionally, the directory "test_run_dir" contains ".vcd" files for each test. These files contain waveforms and can be analyzed with waveform viewers such as gtkwave. Waveforms of simulated test cases can be useful for debugging, especially when it comes to larger and more complex designs.

Continuous Integration Pipeline in GitHub

The repository is equipped with a so-called "CI pipeline", that will run the test cases automatically every time a new update is pushed to the main branch. Two badges at the top of the readme file indicate whether the test cases were run successfully or not. In the beginning, only the test cases for "01_warm-up" are marked as passing, while the test cases for all other tasks fail. The reason for this is that the warum-up task only contains test cases for the implemented example module, while the other tasks already come with first test cases for the hardware design that you will implement during the class project.

In section 6.1 you created a new branch named after your group number for your repo. From now on, you will be pushing updates to your code to this branch, so you also need to update the CI pipeline to run the tests accordingly. Navigate to

```
ADSProject/.github/workflows/
```

in your local copy of the repository. If you don't see the ".github" folder, press CTRL+H. The dot in front of a file or folder means that it's hidden, so you first need to make it visible in your directory with this shortkey. If your navigating with the command line, you can access it by using the "cd" command, "ls -a" will show you all files in a directory, including the hidden ones. The workflows-directory contains a <task>.yml file for each task. Changing

```
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

to

```
on:
  push:
```

```
    branches: [ "<GroupXXX>" ] // name of your branch
  pull_request:
    branches: [ "<GroupXXX>" ] // name of your branch
```

will make the CI pipeline run the tests on your working branch (after you pushed the changes).

Although the CI badges in GitHub are a nice feature, please be aware that they only show the correctness of your own test cases! If you design bad test cases that miss certain bugs, a passing CI test will not make your implementation work fine!

6.3 Assignment 1: Warm-up

The first assignment is intended to be a small warm-up task to get in touch with the first steps of writing hardware and test cases in Chisel. The warm-up Chisel project contains a module called "basic_adder" serving as a amall example for a hardware module. The test directory also comes with a test file "BasicAdder.scala" that shows how test cases can be designed for a specific design. The other designs in the file "adder.scala", as well as their test cases will be done by yourself in the warm-up task. A detailed description of the tasks can be found in the "assignment_01.pdf" document.

The other assignment work similarily. A basic Chisel structure is provided in the matching folder in the repository and the tasks are described in the documents located in the "docs" directory.

A Chisel Coding Style Guide

This style guide aims to provide you with best practices and recommendations for writing clean, readable, and maintainable Chisel code. Following these guidelineslookupreference for good coding styles while working on the class project.
The guide covers various aspects of Chisel coding, including indentation and formatting, naming conventions, module organization, signal declarations, connect statements, control flow, commenting, testing and debugging, documentation, and maintainability. Each section provides clear guidelines and examples to illustrate the recommended coding practices.

Indentation and Formatting

- Use a consistent indentation style, such as 2 or 4 spaces (do not use tabs!).
- Align code elements vertically to improve readability.
- Avoid lines longer than 100 characters.

```
class MyModule extends Module {
  val inputSignal = Input(UInt(8.W))
  val outputSignal = Output(UInt(16.W))

  val intermediateSignal =
    inputSignal * 2.U

  outputSignal :=
    intermediateSignal + 1.U
}
```

Naming Conventions

- Use meaningful and descriptive names for modules, signals, and variables.
- Use camel case for module and signal names (e.g., `myModule`, `inputSignal`).
- Avoid using reserved keywords or names that may cause conflicts.

```
class MyModule extends Module {
  val inputSignal = Input(UInt(8.W))
  val outputSignal = Output(UInt(16.W))
  val intermediateSignal = Wire(UInt(16.W))

  intermediateSignal :=
    inputSignal * 2.U

  outputSignal :=
    intermediateSignal + 1.U
}
```

Module Organization

- Separate modules into individual files with clear names.
- Group related modules into directories or packages.
- Follow a modular design approach with reusable components.

```
// MyModule.scala
class MyModule extends Module {
  // Module definition...
}

// SubModule.scala
class SubModule extends Module {
  // Module definition...
}
```

Signal Declarations

- Declare signals with explicit types.
- Use `val` for combinational logic and `Reg` for sequential logic.
- Use Chisel's type inference when possible.

```
class MyModule extends Module {
  val inputSignal = Input(UInt(8.W))
  val outputSignal = Output(UInt(16.W))
  val intermediateSignal = Wire(UInt(16.W))

  intermediateSignal :=
    inputSignal * 2.U

  outputSignal :=
    intermediateSignal + 1.U
}
```