Undergraduate Research Information

Kevin Laeufer <laeufer@berkeley.edu>

About me:

- PhD student advised by Koushik Sen and Jonathan Bachrach
- part of the Adept lab (5th floor of Soda Hall)
- working on better testing for circuits written in Chisel
- I have already advised 4 Berkeley undergraduate students, two of which are still working with me
- Planning to graduate May 2023 (so you would be able to do research with me for ~4 semesters)

Undergraduate Research Philosophy

- Undergraduate research is primarily about you, not the project you are working on
- We will talk about your personal goals at the beginning of the semester:
    - Is there a certain technical and/or soft skill you would like to work on? (e.g., becoming a better presenter, a better coder, learn how to read research papers, etc)
    - Are you planning to apply to grad school? Are you trying to apply to industry internships?
    - Do you want to attend academic conferences?
- We will meet once every week during the semester
- No hard deadlines/goal that you need to achieve
- I do expect you to spend 5-8 hours per week working on your project (ideally you can find a regular, continuous time slot)
- If you are busy (e.g., because of exams or for personal reasons) you just need to tell me and we can *skip* that week, we cancel the meeting and you won't have to do any work.
- Once you are making progress on your project (after 1-2 semesters) I will ask you to present
    - at the group meeting with Koushik Sen and students
    - at the Adept lab retreat poster sessions with industrial sponsors and other undergraduate and graduate students
- We will have a retrospective meeting at the end of the semester to determine if you want to continue your research and to reflect on your goals.

What is Chisel?

All project suggestions explore new verification features in the context of a domain specific language called Chisel. If you have taken CS61C, you might remember building a RISC-V processor as one of your projects. In CS61C you used a GUI to connect elementary building

blocks with wires to form your processor. This is called *schematic entry*. Chisel is a library written in the Scala programming language that allows you to easily write programs that connect signals together to make a digital circuit. For example, if you want to add the bits of wires `a` and `b` together and call the result `c`, in Chisel you can write `val c = a + b`. It is important to note though, that Chisel *does not* translate a Scala program into a circuit. Instead it allows you to write a Scala program that outputs a circuit, just like a python program using matplotlib might output a graph. If you want to learn Chisel, I recommend going through the Chisel bootcamp which features a series of exercises that you can do in a notebook without having to install anything on your computer.

Projects *(in arbitrary order)*

Project: low-overhead interface between a C++ simulation and a test running on the JVM

Skills: low-level Java programming, C/C++, systems programming, performance measurements
In order to check whether a circuit works as expected, we need to write some tests for it. The chiseltest library makes that easy by providing a high level-interface in Scala that you can write your tests against. In order to quickly execute these tests, the circuit is converted into a C++ simulation using Verilator. The C++ sources are then compiled into a shared library which is loaded into the Scala program using the JNA library. Unfortunately JNA has some overhead that slows down the simulation. In this project you would learn how to call C code from Java/Scala using the JNI. Then you would investigate how to load shared libraries at runtime using dlopen. With that knowledge you will speed up our simulator interface by replacing our use of JNA with a small C library and JNI. At the end of the project you will contribute those changes to the chiseltest project so that the whole community can benefit from your performance improvement.

Project: Java backend for ESSENT

Skills: low-level Java programming, programming in Scala, compilers, performance analysis of code on the Java Virtual Machine, firrtl IR
In order to check whether a circuit works as expected, we need to write some tests for it. In order to test the circuit we need to simulate it (as opposed to manufacturing a real chip, which would take way too long…). For Chisel circuits, there are multiple simulators that we are using, depending on the size of the circuit we are trying to simulator and the length of the test we want to execute. The most important performance metrics are: (1) how long does it take to generate/compile the simulation (2) how fast is the simulation, i.e., how many cycles can we simulate per second. Currently there are two open-source simulators that we mostly use: (1) treadle is an interpreter written in Scala; it quickly generates a simulation, but the simulations can be fairly slow, once the circuit grows bigger (2) Verilator turns our circuit into a C++ simulation model which then needs to be compiled to a binary; thus the compilation for the simulator takes a long time, but once it is compiled, the simulation runs very quickly.

For this project, you are going to explore a third option: You are going to build a simulator that generates Java code that is compiled and then loaded into the testing process. The idea here is that the compilation speed will be faster than with Verilator, while the execution speed will be much better than with treadle. To quickly prototype this, you are going to extend the existing ESSENT simulator to generate Java instead of C++ code. Then you will create a set of benchmarks to test compilation and simulation speeds across all existing simulators + your new prototype simulator.

Project: System Verilog Assertions for Chisel

Skills: programming in Scala, formal methods, automatons, compilers, firrtl IR
How do we detect bugs in our circuit? In order to detect whether something is going wrong while executing the circuit simulation, we need to specify what behaviors are allowed and which ones are not. One mechanism for specifying requirements is the `assert` statement. If there is a signal `a` that should always be less than 4, you can write the following in Chisel:

```
assert(a < 4.U)
```

While Chisel already supports these simple assertions over boolean expressions, many interesting properties involve more than one time step. Synchronous digital circuits like CPUs are driven by a clock which determines how often the state is updated. We want to be able to write properties like, *if a was less than 4 in the previous cycle, it should now be less than 5*. With a new Chisel extension you can write:

```
when(past(a < 4.U)) { assert(a < 5.U) }
```

However, the simple `past` statement is not powerful enough to describe more powerful properties, like: *after b is high and while c is high, a should be less than 4*
These kinds of temporal properties are available in commercial tools for the old industry standard languages Verilog and VHDL. In this project you will explore adding support for assertions similar to the System Verilog Assertion (SVA) language defined in the SystemVerilog standard to Chisel. You will first familiarize yourself with the current basic assertion support offered by Chisel, study some SVA examples, and explanations on how SVA can be implemented. We will also try to find and review some papers on the topic. The goal of the project is to contribute your changes to the Chisel and chiseltest project in order to make them available to the community. Going further (probably after 2-3 semesters), one interesting research question is whether we can add optimizations to your SVA compiler in order to more efficiently check the SVA properties in formal verification, in simulation or in an FPGA accelerated simulation.

Project: coverage trace generation for Chisel

Skills: programming in Scala, formal methods, circuits, firrtl IR, SMT Solvers
Traditionally we test circuits by executing a simulation of them and writing a program (called a test bench) that will supply inputs every cycle and check the outputs of the circuit. Another way

to go about testing is to specify properties that should always hold (see the *System Verilog Assertions for Chisel* project for more info about assertions) and then to exhaustively test all input combinations to see if a property can be violated. Since a circuit accepts new inputs every cycle of its execution, it quickly becomes impossible to just brute-force every combination of input bits. Instead we can convert the circuit into a declarative representation that can be used by a *formal engine* to try and find a combination of inputs that will trigger a property violation after *k* cycles. This is called bounded model checking and is already supported by Chisel and the chiseltest library.
Instead of trying to find property violations, we could also try to make sure that we can execute certain scenarios. So instead of the user asking us to *prove that a is always less than 4*. The user might demand that we give them a trace (a series of inputs to the circuit) that shows that a can be greater or equal to 4. This is useful to (1) explore how the circuit works, by asking for traces that demonstrate certain features (2) uncovering bugs (if the intended feature cannot be demonstrated, then there must be something wrong with the circuit).
In this project you will study the formal coverage implementation for Verilog provided by SymbiYosys. Then you will add formal coverage support for Chisel circuits to the chiseltest library. The final goal of the project is to contribute your coverage implementation to the chiseltest library in order to make it available to all Chisel users. We also want to compare your implementation to SymbiYosys, qualitatively and quantitatively.

Project: mutation testing for Chisel circuits

Skills: programming in Scala, formal methods, circuits, firrtl IR
I work on tools and techniques to more quickly test circuits written in Chisel. However, how can we actually know whether these techniques work? One way of assessing the quality of a test is to see how many bugs it finds. But how do we get a circuit that has enough bugs to provide us with interesting feedback about the performance of our testing technique. The approach we are going to investigate in this project is called mutation testing. The basic idea is to add a single small change to the circuit and then to try and see if the test can detect the change. For this project you will learn about the firrtl compiler which can be used to programmatically change Chisel circuits. Once you are able to automatically introduce small changes (i.e., mutations) into the circuit, we need to make sure that we can filter out the changes that might not be detectable. There are numerous reasons why a change might not be detectable, but a simple example would be changing part of the code that is never executed. Thus to the outside world, the original circuit as well as the mutated one behave exactly the same.
During the course of this project we will study an open source mutation testing implementation for Verilog circuits as well as some academic papers on mutation testing. Then you will implement a version for Chisel circuits. The final goal of the project is to benchmark and study your implementation and to open-source your code so that others in the Chisel community can use it.

Project: taint tracking in Chisel

Skills: programming in Scala, firrtl IR, compilers, hardware security

Taint tracking is a program analysis technique in which we want to answer the question whether the output of a circuit is influenced by a certain input or state element. Most often, the term *influenced* means that if we changed the value at the input, the value at the output would change. This analysis is normally done in a conservative way, i.e., unless we can be sure that A does not influence B, we have to assume that it does. One popular use of taint tracking is to check security properties, such as making sure that the value of secret inputs does not show up in any output that can be seen by the public.

As part of this project we are going to read and understand prior work on tracking information flow in Chisel circuits as well as study the taint analysis framework provided by the LLVM project. We will then come up with a generic taint analysis framework for Chisel circuits that you will implement in Scala as an extension to the firrtl compiler.