Chipyard
An Agile RISC-V SoC Design Framework with in-order cores, out-of-order cores, accelerators, and more

EE241B Tutorial
Written by Daniel Grubb (2020)

1 Overview

In this lab, we will explore the Chipyard framework. Chipyard is an integrated design, simulation, and implementation framework for open source hardware development developed here at UC Berkeley. Chipyard is open-source online and is based on the Chisel and FIRRTL hardware description libraries, as well as the Rocket Chip SoC generation ecosystem. Chipyard brings together much of the work on hardware design methodology from Berkeley over the last decade into a single repo that guarantees version compatibility between the projects it submodules.

A designer can use Chipyard to build a RISC-V-based SoC, from RTL development integrated with Rocketchip, to cloud FPGA emulation with FireSim, to physical design with the Hammer framework we've been using this semester. Information about Chisel can be found in https://www.chisel-lang.org/. While you will not be required to write any Chisel code in this lab, basic familiarity with the language will be helpful in understanding many of the components in the system and how they are put together. An initial introduction to Chisel can be found in the Chisel bootcamp: https://github.com/freechipsproject/chisel-bootcamp. Detailed documentation of Chisel functions can be found in https://www.chisel-lang.org/api/SNAPSHOT/index.html.

There is a lot in Chipyard so we will only be able to explore a part of it, but hopefully you will get a sense of its capabilities. utilize it in your future research projects that require some its elements. We will simulate a Rocketchip-based design at the RTL level, and then synthesize and place-and-route it in ASAP7 using the Hammer flow.

2 Getting Started

First, we will need to setup our Chipyard workspace. For this lab, please work on the `eda-{1-8}@eecs.berkeley.edu` machines and in the `/scratch/` directory on the machine. This lab will likely generate too much data for it to fit in your home directory. Run the commands below.

```
% cd /scratch/userA
% git clone ~ee241/spring20-labs/chipyard
% cd chipyard
% ./scripts/init-submodules-no-riscv-tools.sh
% cd vlsi
% git submodule update --init hammer-cadence-plugins
% source sourceme.sh
```

You may have noticed while initializing your Chipyard repo that there are many submodules. Chipyard is built to allow the designer to generate complex configurations from different projects includ-

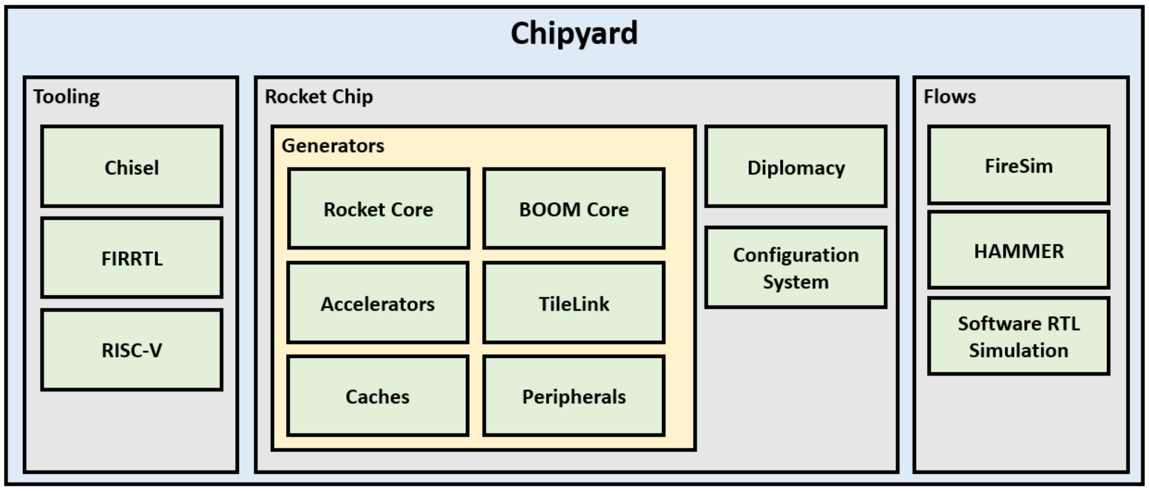

Figure 1: Chipyard Components

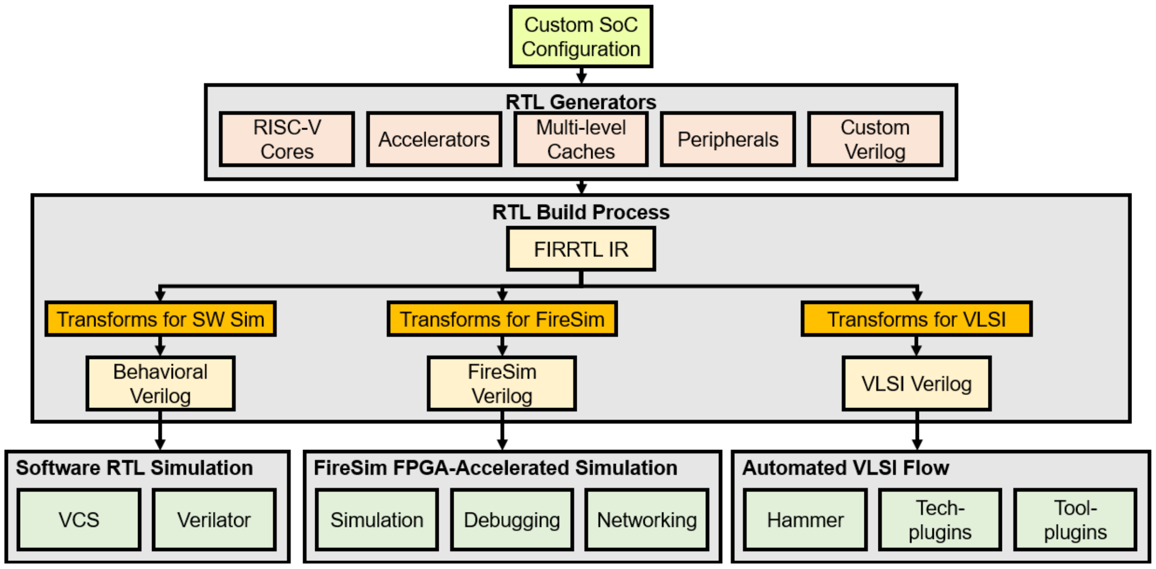

Figure 2: Chipyard Flow

ing the in-order Rocket Chip core, the out-of-order BOOM core, the systolic array Gemmini, and many other components needed to build a chip. Thankfully, Chipyard has some great documentation, which can be found here. You can find most of these in the `chipyard/generators/` directory. All of these modules are built as generators (a core driving point of using Chisel), which means that each piece is parameterized and can be fit together with some of the functionality in Rocket Chip (check out the TileLink and Diplomacy references in the Chipyard documentation). You can find the Chipyard specific code and its configs in `chipyard/generators/chipyard/src/main/scala/config`. You can look at examples of how your own Chisel modules or verilog black-box modules can be integrated into a Rocket Chip-based SoC in `chipyard/generators/chipyard/src/main/scala/example`. Many times, an accelerator block is connected to the Rocket core with a memory-mapped interface over the system bus. This allows the core to configure and read from the block. Again, there is far too much to discuss fully here, but you can really put together a system very quickly using the infrastructure of Chipyard.

3 Chipyard Simulation and Design Benchmarking

3.1 RTL Simulation

Many Chipyard Chisel-based design looks something like a Rocket core connected to some kind of "accelerator" (eg. a DSP block like an FFT module). When building something like that, you would typically build your "accelerator" generator in Chisel, and unit test it using ChiselTesters. You can then write integration tests (eg. a baremetal C program) which can then be simulated with your Rocketchip and "accelerator" block together to test end-to-end system functionality. Chipyard provides the infrastructure to help you do this for both VCS (Synopsys) and Verilator (open-source). In this lab, we are just focusing on a Rocket core in isolation, so we will run some assembly tests on a Rocket config. You can edit the configs being used in `chipyard/variables.mk`. In this case, we will work with a Rocket config that is a basic Rocket core (includes L1 data and instruction caches) with an on-chip scratchpad memory. Go to the `chipyard/sims/vcs/` directory and run:

```
% make SUB_PROJECT=scratchpad
% make run-asm-tests
```

This will build a VCS-simulator for the config specified in `variables.mk` and will run a suite of RISC-V assembly tests. If you're unfamiliar with Chisel, running this command will elaborate your design. This means that the Chisel code, embedded in Scala, is compiled and run through the FIRRTL backend, and eventually Verilog is generated. We can use all the same VLSI tools while simulating and building our design, but a designer can use Chisel to more powerfully generate RTL. You can look at the results of the simulations in `chipyard/sims/vcs/results/` to verify that the different test passed.

Q: 1. In your lab report, include the last 10 lines of the output of one of the assembly tests you ran. It should include the * PASSED *** flag. You may stop the simulations part way through if you don't want to run all of the assembly tests.**

3.2 FireSim

A key component of Chipyard is FireSim, which is an open-source cycle-accurate FPGA-accelerated full-system hardware simulation platform that runs on cloud FPGAs (Amazon EC2 F1). This allows for simulations of your system at orders-of-magnitude faster speeds than software simulation. A key point of FireSim is that it is cycle-accurate since it actually models things like DRAM access latency by transforming parts of your design. Emulating your design normally on an FPGA does not model these system-level aspects that your actual chip will run with. Using FireSim is outside the scope of this lab, but it is worth looking in to.

4 VLSI Flow

4.1 Design Elaboration

The Hammer flow we have used throughout the semester is integrated into Chipyard. A project setup similar to the ones we have previously used is in `chipyard/vlsi`. In this directory, run

```
% make srams
```

This will elaborate our Rocket design similarly to the `sims/` directory, but this time, the required hard SRAM macros will be generated by the macrocompiler in the `barstools` submodule. This macrocompiler step is required because the memories available in the technology will not match the ones required by your design exactly. This step gives you the list of technology-available SRAMs that map to your design and fits the connections into the correct parts of your RTL. The SRAMs available in ASAP7 can be found in `hammer/src/hammer-vlsi/technology/asap7/`.

We are using a slightly different config from the one we simulated, with smaller caches and a smaller scratchpad for the purpose of simplifying the physical design for this lab. The elaboration results can be found in `vlsi/generated-src/`. There are many files that are generated in this folder that relate to FIRRTL compilation of the Chisel design and of course the final Verilog output files. All of the file names are pre-fixed with the name of the config used. In the `generated-src` directory (all prefixed with `chipyard.TestHarness.SmallScratchpadRocketConfig`), `top.mems.conf` describes the parameters of the memories in your design and top.mems.v shows the actual Verilog instantiations of the individual SRAM blocks used that correspond to the total memories in `top.mems.conf` (the names of the memory blocks will match the Verilog module names).

Q: What is the breakdown of ASAP7 SRAM blocks for each of the memories in the design? (this can be found by looking at the files described above.)

`generated-src/` also includes your elaborated Verilog in top.v, files related to your system's device tree, and even a `graphml` file that visualizes the diplomacy graph of the different components in your system (can be viewed with an online viewer for instance).

4.2 Synthesis

Now that the design is elaborated, we can leverage the Hammer infrastructure we have used this semester to physically build our system in much the same way as before. The Hammer entry point runs through `example-vlsi` (setup to easily include additional custom Hammer hooks) and our Hammer config is in `example.yml`. Here you can see we have again constrained our top-level clock to be 50 MHz. It is pretty straightforward to close timing for Rocketchip in the 100's of MHz with limited physical design input using Hammer out-of-the-box, but we are running it at this lower frequency to ease our design constraints. Run the following (again uses the auto-generated Make fragment in `build/`).

```
% make syn
```

This step should take up to about 1 hour. If you are ssh'd directly into the machine (not using X2go, etc.), you should use a utility like `tmux` to make sure that you don't lose your run if you lose your connection or log off. When it completes, you can look at the results just like before in `syn-rundir/reports/` to confirm your design passed timing.

4.3 Floorplanning

As discussed in Lab 1, floorplanning is a key step to all designs and will have a huge effect on your design's QoR. In this case, we must place the SRAM macros for our Rocket core's L1 caches and scratchpad. `example.yml` has an example of SRAM placement in Hammer under `vlsi.inputs.placement_constraints`. You can see that these macros match the ones we saw in the `mems.v` file we looked at earlier. In these Hammer constraints, you can see that the lower-left hand corner placement is specified (in microns) as well as the orientation (see `hammer/src/hammer-vlsi/defaults.yml` for documentation on all placement options). A common design pattern when placing SRAMs is to "butterfly" them such that the different macro blocks abut each other on multiple sides, while the side with its pins is exposed so that it can be routed to.

Hammer provides a floorplan visualization tool to give you a very quick preview of your floorplan. The visualization tool is called in the `visualize_floorplan` hook in `example-vlsi`. You can run the floorplan tool with `make par` and then kill it with `ctrl+c` after the first step runs. The visualization will be generated at `par-rundir/all.svg`, which can be opened using `display` or a browser.

Q: Include a screenshot of the Hammer generated floorplan visualization output.

The floorplan provided here is fairly random, but you may play around with placement and visualize your changes using this same process. We recommend that you try playing around with the floorplan and attempt to see changes in area, etc. QoR if you have time, since you can kick off jobs and have them run with little supervision.

As noted in a previous lab, if you edit an input yml/json file, the Hammer Make include file will detect this and re-run the flow. If you want to edit a setting without re-running the whole flow, you can run `make redo-STEP HAMMER_REDO_ARGS='-p floorplan.yml'`. This will break the dependencies and just run that step, and additional arguments can be added to the call with the `HAMMER_REDO_ARGS` flag. New input files can add additional Hammer key flags, or it can override keys in the originally included files. Just make sure you are keeping track of what flags you are using when using the redo steps.

4.4 P&R

Now that we have synthesized the design and created a floorplan, we can run through P&R just like before. Note that you can open up the design in Innovus by restoring one of the generated Innovus databases in `par-rundir` after the floorplan step has run to confirm your floorplan made it into the design (switch to floorplan view as opposed to physical view).

```
% make par
```

You can open up the final design in Innovus using `par-rundir/generated-scripts/open_chip` just like in the previous labs. Note that we are using "express" level effort (Hammer key: `par.innovus.design_flow_effort: "express"`) for the sake of runtime.

Q: Include a picture of your design in Innovus with M8 and M9 turned off.

Q: How much setup timing slack is there in the design?

Q: Include a picture of the clock tree debugger for your design from Innovus and comment on the balancing.

5 Rest of the VLSI Flow

Running DRC and LVS is not required for this lab, but you can run them though Hammer just like before. The placement of macros like SRAMs can cause considerable numbers of DRC and LVS errors if placed incorrectly and can cause considerable congestion if placed non-optimally. The floorplan visualization tools in Hammer can help you root out these problems early in your design process.

6 Conclusion

Chipyard is designed to allow you to rapidly build and integrate your design with general purpose control and compute as well as a whole host of other generators. You can then take your design, run some RTL simulations, benchmark it with FireSim, and then push it through the VLSI flow with the technology of your choice using Hammer. The tools integrated with Chipyard, from how you actually build your design (eg. Chisel and generators), to how you verify and benchmark its performance, to how you physically implement it, are meant to enable higher design QoR within an agile hardware design process through increased designer productivity and faster design iteration. We just scratched the surface in this lab, but there are always more interesting features being integrated into Chipyard. We recommend that you continue to explore what you can build with Chipyard given this introduction!

7 Acknowledgements

Thank you to Alon Amid and the whole Chipyard dev team for figures and documentation on Chipyard.