An Introduction to Declarative CPU Design and FPGA Development using the Chipyard SoC Design Framework

Alexander Lukens

Karl Hallsby

Faculty Advisor: Dr. Jia Wang

Illinois Institute of Technology

Last Edited: June 24, 2021

Contents

Chapter 1

Setup

1.1 Introduction

This document is intended to serve as a record of the work performed for the ECE 497 special project supervised by Professor Jia Wang during the Spring 2021 semester. In this document, we will specify how our project repository was created, outline issues we ran into, and provide guidance on how to better setup the Chipyard Framework.

Chipyard is a framework for designing, elaborating, simulating, testing, and building RISC-V CPU designs. It provides the functionality to define a set of standard CPU designs, but also allows for the end-user to describe their own custom designs and integrate them as first-class citizens in the framework. It also provides a toolkit for verifying that elaborated CPU designs meet the RISC-V ISA (Instruction Set Architecture) standard, ensuring designed chips are compliant. There are also tools for writing the elaborated designs out to FPGA (Field Programmable Gate Array) bitstreams, so that simulation can be sped up and execution can occur on Softcores. Lastly, Chipyard includes tools for a VLSI-design workflow, to implement the elaborated CPU design on actual silicon.

1.2 Project Environment

The first step to using the Chipyard Framework is creating a project environment and obtaining all of the Chipyard dependencies. In this document, we assume you are using Ubuntu 20.04 LTS in a virtual machine with the following system specifications, or running the Docker container with the host machine having **at least**:

- 4 cores
- 16 GB of RAM, or more
- 250 GB disk image

Much of the disk space that has been allocated will be utilized, as the entire RISC-V toolchain and Xilinx Vivado Suite require a large amount of disk space.

This document will work equally well in other distributions (Fedora, CentOS, OpenSuSE, Archlinux, etc.), so long as the versions of the dependencies are matched. Chipyard has explicit support for CentOS, extending to Fedora and RHEL as well.

Using Linux as the native operating system, rather than as a virtual machine is the preferred way of working with Chipyard. This gives the running system *all* available system resources and removes the virtual machine execution penalty.

1.2.1 Docker Container

To ease the distribution of the required dependencies, we have included a Dockerfile for building an environment very similar to ours. Listing 1.1 shows how to build the image, *after* having cloned our repository.

```
$ DOCKER_BUILDKIT=1 docker build -t ece497:deliverable ./
```

Listing 1.1: Build Docker Image

Note that this image will take a *very* long time to build. It not only creates a new Ubuntu image for you to work with, it also:

1. Updates Ubuntu to the latest version
2. Fetches and builds Verilator from source
3. Fetches Chipyard
4. Initializes Chipyard's submodule dependencies
5. Builds the RISC-V toolchain Chipyard relies on

Once the Docker image is built, you can spawn as many instances as you need, using the command in Listing 1.2.

```
$ docker run -it --user chipyard ece497:deliverable /bin/bash
```

Listing 1.2: Run and Enter Docker Container

1.3 Document Typesetting

This document makes use of a variety of different fonts and colors to denote different aspects of this work. Each of the different font settings are explained here.

`Teletype Text` Computing-related topics/items. This is typically used to denote terms you will see in this document, repository, and other materials surroudning this topic, but do not correlate to any of the other options presented in this list.

`file/path` A relative file path. This is typically used with `chipyard/subdir`, meaning you should move to the specified subdirectory or file *inside* of the Chipyard subdirectory. File paths will only look this way when a file is specified by itself. In a command, this highlighting will not be present.

`/file/path` An absolute file path. When specified this way, you must provide the entire path specified. File paths will only look this way when a file is specified by itself. In a command, this highlighting will not be present.

Blue Text A link to another location within this document. Clicking other word(s) that look like this will take you to a different place inside this document.

URL Link A link to take you outside of this document.

`$ cmd-to-run` A command to run in your terminal. We assume you are using the Bash shell.

In addition, we make use of `man` syntax here. This means that text inside of angle brackes is mandatory, inside of square brackets is optional, and the vertical pipe is an "or." An example is shown in Listing 1.3.

> Text inside black boxes, like this one, is meant to provide an area for notes that should be remembered. For example, some of these provide reminders to parallelize code compilation to speed up the process.

```
$ command <mandatory-arg> [optional-arg-1] [optional-arg-2a | optional-arg-2b]
```

Listing 1.3: man Syntax

1.4 Building Chipyard

Here, we present the necessary steps to retrieving all the dependencies required to set up Chipyard for local development and simulation use. The larger code listings shown in this document is gathered in the code subdirectory of this document's project directory. We developed this documentation using version **1.4.0**[1] of Chipyard.

1.4.1 Chipyard Dependencies

To gather the Chipyard dependencies, follow the Chipyard documentation closely. Specifically, the Section 1.4 of the documentation outlines how to prepare your operating system for development using the Chipyard framework.

A paraphrased reproduction of these steps are shown below.

Retrieve/Install Dependencies

Chipyard relies on numerous dependencies and libraries to read files and build the required Verilog files. In addition, Chipyard relies on `sbt` (Scala Build Tool), because a majority of Chipyard and its dependencies are written in Scala.

Listing 1.4 is a script that handles fetching and installing all the dependencies for you. Note that this does **not** work for installing the dependencies for Linux distributions that do not use the `apt` package manager.

Build Verilator from Source

Chipyard's documentation recommends building Verilator (an open-source (System)Verilog simulator and compiler) from source.

A small script has been provided that handles this for you in Listing 1.5. Note that this does **not** work for installing the dependencies required to build Verilator for Linux distributions that do not use the `apt` package manager.

Fetching Chipyard and its Direct Dependencies

In addition to the library and external programs that Chipyard depends on, it also uses git submodules to track direct dependencies. Direct dependencies are projects that Chipyard directly relies on. These include SiFive's CPU designs, the BOOM CPU design, Rocket-Chip, and several others.

Listing 1.6 has been provided that handles this for you.

1.4.2 Building a Toolchain

To compile programs from C to RISC-V instructions, there are several tools you need, when grouped together is called a toolchain. Your cloned Chipyard repository contains a script to install these. You can run the script to build a good general-purpose toolchain using Listing 1.7 or Listing 1.8 while inside your local copy of the cloned Chipyard repository.

[1] Git Commit Hash: 58076cfb260a3be502d6d1c25b577da39277a7fc

```
$ #!/usr/bin/env bash
$
$ set -ex
$
$ sudo apt-get install -y curl
$ sudo apt-get install -y build-essential bison flex
$ sudo apt-get install -y libgmp-dev libmpfr-dev libmpc-dev zlib1g-dev vim git default-jdk
  ↪  default-jre
$ # install sbt:
  ↪  https://www.scala-sbt.org/release/docs/Installing-sbt-on-Linux.html#Ubuntu+and+other+Debian-based+
$ echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee -a
  ↪  /etc/apt/sources.list.d/sbt.list
$ curl -sL
  ↪  "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823"
  ↪  | sudo apt-key add
$ sudo apt-get update
$ sudo apt-get install -y sbt
$ sudo apt-get install -y texinfo gengetopt
$ sudo apt-get install -y libexpat1-dev libusb-dev libncurses5-dev cmake
$ # deps for poky
$ sudo apt-get install -y python3.8 patch diffstat texi2html texinfo subversion chrpath git
  ↪  wget
$ # deps for qemu
$ sudo apt-get install -y libgtk-3-dev gettext
$ # deps for firemarshal
$ sudo apt-get install -y python3-pip python3.8-dev rsync libguestfs-tools expat ctags
$ # install DTC
$ sudo apt-get install -y device-tree-compiler
```

Listing 1.4: Fetch Chipyard Dependencies using `apt` on Ubuntu

```
$ #!/usr/bin/env bash
$
$ set -ex
$
$ # Dependencies for building verilator from source
$ sudo apt install autoconf automake
$
$ # install verilator
$ git clone http://git.veripool.org/git/verilator
$ cd verilator
$ git checkout v4.034
$ autoconf && ./configure && make -j$(nproc) && sudo make install
```

Listing 1.5: Building Verilator from Source on Debian-derivative Linux Distributions

```
$ #!/usr/bin/env bash
$
$ git clone https://github.com/ucb-bar/chipyard.git
$ cd chipyard
$ ./scripts/init-submodules-no-riscv-tools.sh
```

Listing 1.6: Fetch Chipyard and Submodules

```
$ ./scripts/build-toolchains.sh riscv-tools
```

Listing 1.7: Build RISC-V Toolchain

```
$ export MAKEFLAGS=-j[N]; ./scripts/build-toolchains.sh riscv-tools}
```

Listing 1.8: Parallel Build RISC-V Toolchain

Environment Variables

Once the toolchain is built, an environment-setup script is emitted to the root of your local copy of Chipyard, with the name `env.sh` (located at `chipyard/env.sh`). This file is a bash script that changes your `PATH`, `RISCV`, and `LD_LIBRARY_PATH` environment variables so that Chipyard can find everything it needs.

To alleviate any issues that may occur due to misconfigured or non-existent environment variables, we recommend you do one of the following:

1. Add the line `$ source /path/to/chipyard/env.sh` to the end of your `.bashrc` file in your home directory. After adding this to your `.bashrc` file, restart your shell. Or re-source your `bashrc` (`$ source .bashrc`) and continue.

2. Install the `direnv` package and use it to automatically change your environment variables for you, instead of having them constantly loaded the way the previous option does.

1.5 Example CPU Design

In this section, we show how to build and simulate the default CPU Chipyard defines. This particular CPU is relatively easy to elaborate, requiring just 6 GB of memory.

1.5.1 Building the Example Design

To build the example design that Chipyard defines, all you must do is enter one of the simulator directories and type `$ make` . This **does** require that both the RISC-V toolchain you built and Verilator Verilog simulator be loaded into your environment (see Section 1.4.2). If one of these is not available, `make` will print out an error message why it is failing.

We strongly recommended that you parallelize the elaboration of the CPU design. You can achieve this by passing the `-j [N]` flag to `$ make`. You may replace the `[N]` with a number to indicate the number of your CPU cores to use for building.
If you omit the `[N]` entirely, the build system will use ALL cores!
The elaboration of the default `RocketConfig` requires about 6.5 GB of main memory. Otherwise the process will fail with `$ make: *** [firrtl_temp] Error 137` which is most likely related to limited resources. Other configurations might require even more main memory [1].
Using many cores increases the amount of system memory required, so be sure that you do not request too many cores be used if you are limited on memory.

The commands to run, in order, are:

1. `$ cd chipyard/sims/verilator`
2. `$ make`

Finishing the elaboration of the design produces an **executable** called `simulator-chipyard-RocketConfig`. This executable is capable of running any RISC-V compatible code.

1.5.2 Running the Example Design

To run arbitrary code, the executable takes the ELF (Executable and Linkable Format) file of the program to run as a parameter. An example of the command to run is shown in Listing 1.9.

```
$ ./simulator-chipyard-RocketConfig
  ↪ $RISCV/riscv64-unknown-elf/share/riscv-tests/isa/rv64ui-p-simple
```

Listing 1.9: Run Arbitrary RISC-V Programs using Example Design

Chipyard also provides a quality-of-life `make` target when running these programs, shown in Listing 1.10.

```
$ make run-binary BINARY=<path/to/riscv/elf>
```

Listing 1.10: `make` command to run arbitrary RISC-V programs using Example Design

Using the `make` target also allows the built design to accept many common command line options, including redirecting `STDOUT` to a file.

1.5.3 Simulating the Example Design

Similar to Section 1.5.2, the simulations are actually just a set of RISC-V programs designed to test the built designs. There are two main commands for running the simulation test suite: Listings 1.11 and 1.12.

```
$ make run-asm-tests
```

Listing 1.11: Run Compliance Tests to RISC-V ISA

```
$ make run-bmark-tests
```

Listing 1.12: Run Benchmark Tests

> To parallelize the Verilator simulator, you must pass the `VERILATOR_THREADS` variable to `$ make`. As an example, `$ make VERILATOR_THREADS=4` will inform Verilator to use 4 cores/threads when simulating the design.
> Note that if you pass the `-j[N]` flag to `make`, then **each** spawned thread will use the specified number of `VERILATOR_THREADS`. The result is that you will need `N` × `VERILATOR_THREADS` to simulate the design. If you do not have that many cores on your hosting CPU, the simulation will be greatly slowed.

1.6 Xilinx Vivado Suite Installation

It is important to install the Xilinx Vivado Suite if any work regarding an FPGA is to be conducted. The suite features tools a variety of tools used in teh design, building, and testing of hardware designs using Softcores.

Vivado, one of the programs in the suite, is used for all aspects of managing FPGAs. It handles the setup process for the FPGA, writing the bitstream to the FPGA, among many other features.

We used the "offline installation" version of the Xilinx Unified Installer (version 2020.2), so no 3rd party libraries would need to be installed. Xilinx is supported for a variety of operating systems, including Ubuntu[2]

When conducting the installation, be sure to select the "Vitis" installation target instead of just selecting "Vivado". Installing Vitis will install both Vivado, and all other Xilinx tools needed for implementing FPGA projects.

1.7 Other Useful Projects

1.7.1 Freedom E SDK

This repository is maintained by SiFive, and provides several useful tools for designing, uploading, and debugging software to FPGA devices [17]. This repository is specifically meant for use with SiFive IP, but can still be utilized for Chipyard projects with some modification.

For setting up this repository with its dependencies and compiling the necessary programs, refer to their Prerequisites section.

1.7.2 Freedom Tools

This repository is maintained by SiFive [18]. It will be used to generate several tools that will be used during this project, such as:

- The GCC cross-compiler for RISC-V (and many extension sets of RISC-V)
- OpenOCD, which assists users in debugging their FPGA designs
- RISC-V QEMU for system testing through emulation
- And other useful software.

These tools take a considerable amount of time and disk space to compile so it is best to run `make` as `$ bash` make -j‘nproc‘ to parallelize compiling. Note that this will consume many system resources, and you should be prepared to have an unresponsive machine while the system is building these tools.

[2] Xilinx only officially offers support for Ubuntu 16.04.2 LTS, but it should work on any Ubuntu version since then.

Chapter 2

Repository Deep Dive

In this section, we briefly discuss each of the subdirectories present within the root of Chipyard, take note of any particularly important files, and demonstrate how this entire system is put together.

2.1 Languages Used in Chipyard

There are several programming languages used in the construction of Chipyard that you should be at least familiar with. They are:

- Make. Discussed in Section 2.2
- Scala
- Chisel/FIRRTL (Flexible Intermediate Representation Register Transfer Language). Both of these are programs that work using files written in Scala `DSL` (Domain-Specific Language)s.
- Verilog

In short, Make and its Makefiles are used to glue all the separate parts of the Chipyard framework together. By calling a single `make` command, and possibly providing flags, all the necessary dependencies are found locally, and placed in the right search locations. It also handles the process of directing `sbt` to work on the proper files.

Each of the generators and Chipyard itself parameterizes the Verilog code using Scala. Verilog is the lowest-level "programming language" used in this framework. It defines the semantic behavior of circuits. Scala is then used to allow multiple of the same Verilog module to be composed together to form a final design.

The most direct example of this is the use of Scala to parameterize the number of Rocket cores to include in the generated CPU design. Listing 2.1 is a good example of this.

```
1 class QuadRocketConfig extends Config(
2     new freechips.rocketchip.subsystem.WithNBigCores(4) ++
3     new chipyard.config.AbstractConfig)
```

Listing 2.1: Example of Scala-Parameterized Verilog

In Listing 2.1, the constructor `WithNBigCores` will return a class of big Rocket-Chip cores. Providing an integer to this constructor returns that number of core objects. This informs FIRRTL about the proper system setup, so that it can elaborate the design, creating the final product. If you want eight, twelve, or even thirty-two cores, simply change the passed integer. You will also hosting system that has enough resources to elaborate the design you specify. This also makes the Verilog writer's job easier, because they only need to write their hardware description module for a single CPU, rather than having to worry about multiple processors.

2.2 Makefiles, or the Glue of this Framework

Chipyard makes **heavy** use of Makefiles to pull together and automate various parts of the build system. Variables and/or values that are shared between different ways of building systems are higher in the directory structure.

Thus, some of the most overarching commands and variables for this project are defined in `chipyard/variables.mk`. One of the first things defined within this file are numerous output messages.

2.2.1 SUB_PROJECT

The first notable part of the `variables.mk` file is the `SUB_PROJECT` defaulting variable. This variable allows for easy re-configuration of the entire framework to support elaborating your own CPU designs. By changing this file between one of the well-defined options, one can easily re-use major portions of Chipyard's architecture.

For example, to switch from a CPU defined by Chipyard to one that is uses the Hwacha accelerator, one just needs to say `make SUB_PROJCT=hwacha`, and all the necessary configuration variables are changed.

2.2.2 Building Each Subproject

The next notable part of this file is its large *ifeq ... endif* blocks. Each of these defines a different subproject that can be built and elaborated upon by Chipyard and its surrounding framework. These subproject defining blocks each define multiple higher-level variables that are used to build and test each of the CPUs. Each of the variables is important, and Chipyard provides documentation for each variable inside `variables.mk`. Additional information that we gathered through trial-and-error is presented below.

`SUB_PROJECT` This corresponds to one of the projects in the `chipyard/generators` directory. More formally, it is defined by one of the entries in the `build.sbt` files in the respective generators directory, and by the main `build.sbt` file in the root of Chipyard.

`SBT_PROJECT` This corresponds to a top-level of the repository of the chip to build. This is where many of the higher-level constructs, such as the test harness and test bench are defined from.

`MODEL` The model is the top-level module of the project that should be used by Chisel. Normally, this should be defined to the be same as the test harness, but does not necessarily have to be.

`VLOG_MODEL` This is the top-level module of the project that should be used by FIRRTL/Verilog. Like `MODEL`, this is usually the same as the test harness, but does not necessarily need to be.

`MODEL_PACKAGE` This is the Scala package that is used to find the overall model of the CPU. This should correspond to the `package <packageName>` in a Scala CPU configuration file.

`CONFIG` This defines the parameters that should be used for the project. Typically, this is used to select one of the CPU configurations defined in the `SBT_PROJECT`.

`CONFIG_PACKAGE` This is the Scala package that defines the `Config` class. This file **MUST** contain the class definition for `Config`, meaning `object Config` must be present.

`GENERATOR_PACKAGE` This is the Scala package that defines the `Generator` class. This file **MUST** contain the class definition for `Generator`, meaning `object Generator` must be present.

`TB` This defines the test bench wrapper that extends over the test harness to allow for simulation in a Verilog simulator.

`TOP` This is the top-level module of the project. Typically, this is the module instantiated by the test harness.

2.3 build.sbt

There are two main build.sbt files that you should be aware of. There is a build.sbt for each of the generator subdirectories. These define some metadata information about each project, such as the name of the design, the authors of the design, the targeted sbt version, and others.

However, the build.sbt file in the root of Chipyard is a metadata file not just for Chipyard itself, but also pulls together all the dependencies in chipyard/generators/ so that they all can be elaborated upon with Chipyard.

This file also should be used for defining your *own* CPU. Note that this means you are building your own Verilog code which defines the generation rules for a CPU. However, you must be careful not to introduce circular dependencies into the dependency graph between the CPU generation and elaboration tools. Even though Scala has support for lazy evaluation, it does not completely extend to dependency evaluation, and the entire system can fall apart. This does *not* mean that you use this to build a new CPU on top of the architecture already defined by Chipyard, or any other CPU. However, you can use other CPU-generating systems inside your design.

2.3.1 About

The primary way to simulate SoC (System on a Chip)s designed using the Chipyard framework is with Verilator simulations. The directory for verilator is chipyard/sims/verilator. An example simulation can be run by using `$ make` in the chipyard/verilator/ directory. Running the `$ make` command produces a simulator executable in the verilator/ directory.

Custom Chipyard configs can be simulated by running `$ make CONFIG=<your custom config>`. For example, if your project name was "TestConfig", running `$ make CONFIG=TestConfig` would create an executable called simulator-chipyard-TestConfig in the verilator/ directory. Custom RISC-V code can be run by using the command `$ ./simulator-chipyard-TestConfig /path/to/riscv/executable` from the chipyard/sims/verilator directory.

2.4 Generators

In this section, we look at each of the subdirectories inside the chipyard/generators subdirectory in turn. Each of the CPU generators presented below are unique in their implementation of the open RISC-V ISA. Some of the generators are accelerators; they are meant to be implemented as add-on processors to main CPUs. For example, the SHA3 accelerator is meant to be implemented with a Rocket-Chip as its main CPU.

2.4.1 BOOM

BOOM (Berkeley Out-of-Order Machine) is a CPU defined and built by Univerity of California at Berkeley that implements the RISC-V ISA. Its claim to fame is that it can execute RISC-V instructions out-of-order, thereby drastically improving performance. It is designed to be highly performant, synthesizable, and parameterizable.

BOOM includes support for the following operations:

- Floating Point (IEEE 754–2008)
- Atomic Operations
- Caching
- Virtual Memory

In addition, BOOM supports external debugging. Microarchitectural documentation can be found here. The GitHub organization and its development can be found here.

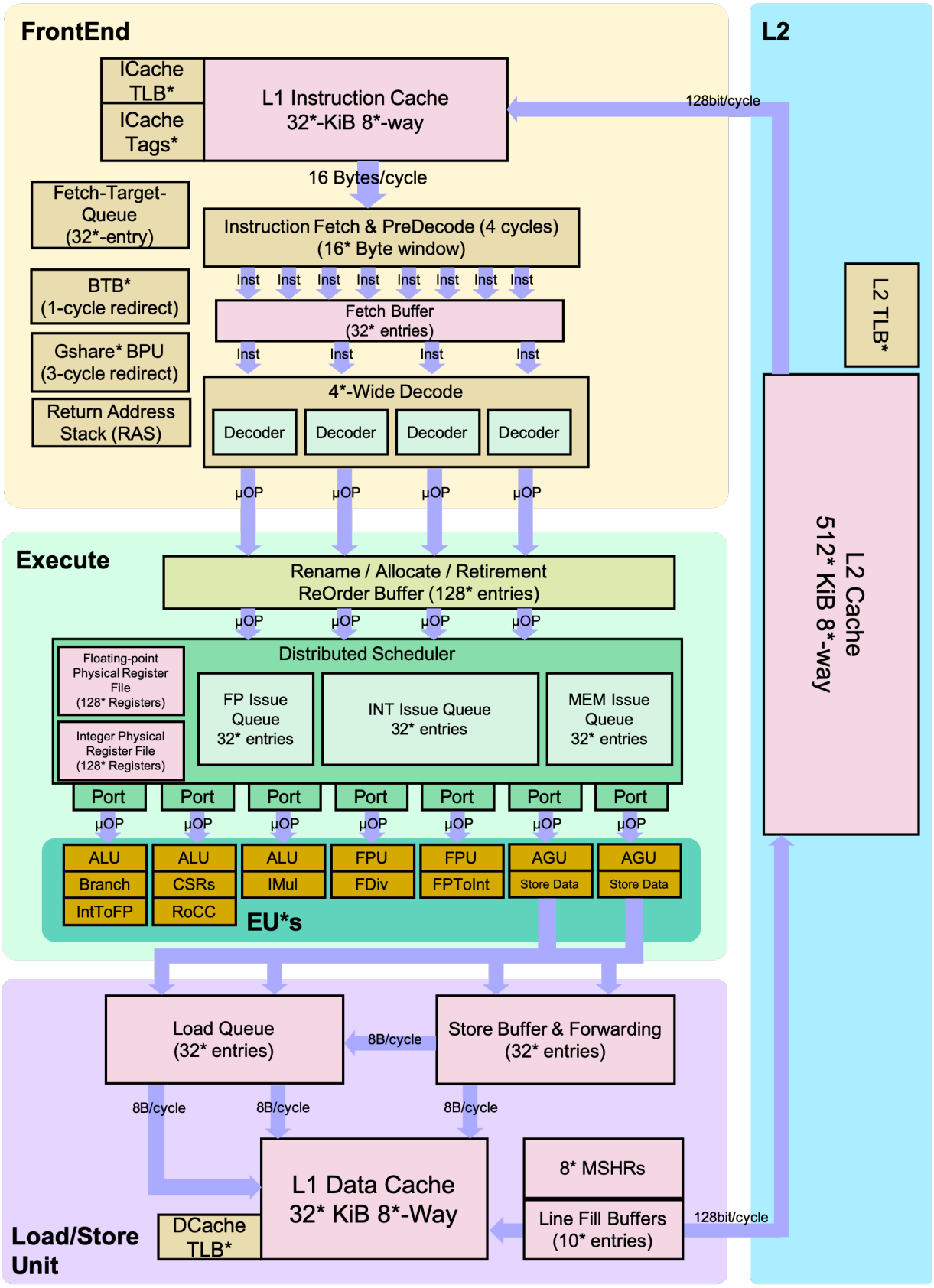

Figure 2.1: BOOM Pipeline [34]

A detailed flowchart of BOOM's pipeline design is shown in Figure 2.1. It is not necessary to understand the BOOM design to use it in Chipyard. We include it as an example of a complex system that we would prefer the computer handle building, rather than us.

These CPU definitions are used by Chipyard when elaborating CPU designs defined by the end-programmer.

2.4.2 Chipyard

This is the main source of truth inside this repository and the location where all of the code required to get these disparate CPUs to work and build together is located. Typically, very little editing is needed to be done here. Most of the editing in this repository comes in the form of defining your own CPUs, which are themselves defined in terms of other CPUs or other lower-level Chipyard constructs.

The main point of interest is the `chipyard/generators/chipyard/src/main/scala/config` directory. This houses CPU design configuration files written in Scala. Each of them defines a different class of CPU, ranging from BOOM to cva6, to RISC-V Sodor configurations. Each CPU design that can be built using Chipyard is a class defined in one of these design configuration files.

2.4.3 cva6

cva6 is a 6-stage, single issue, in-order CPU. This means that unlike the BOOM design, instructions are *always* executed in order. It fully implements the 64-bit RISC-V instruction set, and several extensions, including:

I Base Integer (the base 64-bit instruction set)

M Integer Multiplication and Division

A Atomic Operations

C Compression/Decompression Operations

In addition, this design supports all three privilege levels defined in the RISC-V ISA:

M Machine-mode, the most privileged mode. Designed for the bootloader, firmware, controlling physical resources, and handling interrupts. It is *not* interruptible, and will never be stopped by actions happening in S or U mode.

S Supervisor-mode. Runs the kernel, kernel modules, device drivers, and hypervisors.

U User-mode, the least privileged mode. Only runs user processes.

By supporting all three privilege levels and the extensions, this chip can run a full Unix-like operating system.

2.4.4 Gemmini

Gemmini is not a CPU; instead, it is a CPU *accelerator*, implemented alongside another CPU, such as a BOOM or Rocket-Chip design. It is intended for hardware-level matrix operations, such as matrix multiplication, machine learning, and other SIMD (Single Instruction Multiple Data) operations. A general logic design for the Gemmini accelerators is shown in Figure 2.2.

Gemmini is intended for integration with other Rocket-Chips, but can be configured to work with BOOM chips as well.

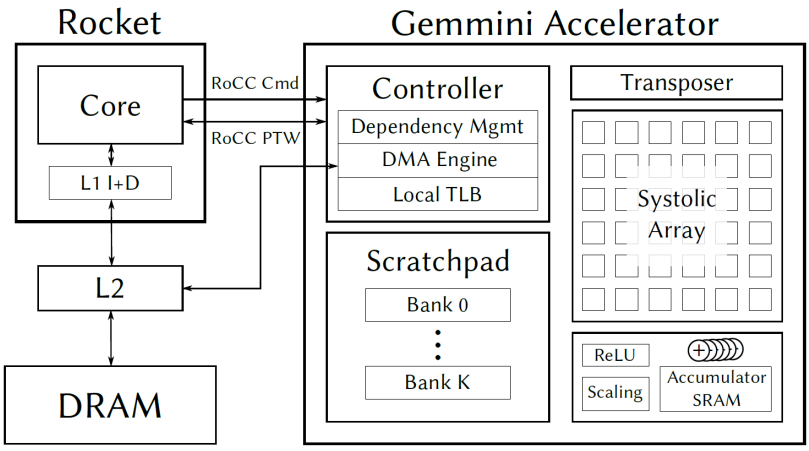

Figure 2.2: Gemmini Accelerator Logic Design [24]

2.4.5 Hwacha

Like the Gemmini, Hwacha is meant to be an accelerator. Hwacha is a co-processor, designed to be run with other CPU processors, namely the Rocket-Chips. Hwacha, like Gemmini, is another SIMD co-processor, but designed to work with vectors instead of matrices.

In terms of computing, the difference between a vector and a matrix depends on the context. A vector could be a list of characters, integers, floating-point values, or booleans. On the other hand, a matrix mirrors its mathematical definition, of an $m \times n$ dimensional construct containing values.

However, in this case, when we say the Gemmini works on matrices and the Hwacha works on matrices, we mean their hardware design for computation. The Hwacha is intended to work on long strings of data at the same time. The Gemmini instead works by performing matrix operations (such as matrix multiplication) in a single operation. In a processor that does not support matrix operations, matrix multiplication is *very* expensive, because each row and column of a matrix multiplication must be computed separately. With hardware-level matrix operations, the entire matrix multiplication can be done with a single instruction, *greatly* speeding up execution.

The Hwacha uses several alternative concepts in its design, including:

- A configurable register file, which is defined by software
- A runtime-variable vector length register
- Aggressive prefetching of memory, due to constant-stride memory accesses
- Resolving memory references as early as possible.

These all feed into Hwacha's goal of maximizing the efficiency of an in-order vector microarchitecture. It was designed to be usable with another CPU to run an operating system that supports unified virtual memory and restartable exceptions. A block-level diagram of the major components in the Hwacha coprocessor is shown in Figure 2.3

2.4.6 Icenet

Icenet is a Generator that is no less interesting, but less applicable to our uses. It is intended to provide Ethernet-based networking components to support the FireSim design simulator. Like all other components in Chipyard, this is also parameterizable, allowing for multiple NICs to be defined. Because we were more focused on getting generated RISC-V images written to an FPGA for local testing, we did not investigate this particular generator. The overall design of Icenet is shown in Figure 2.4.

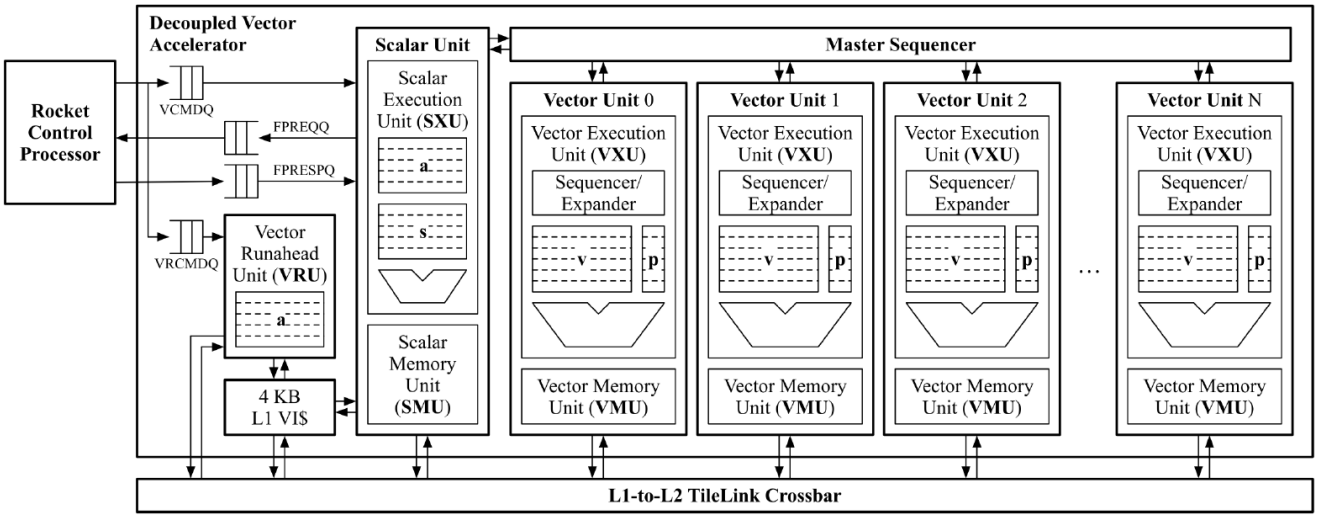

Figure 2.3: Hwacha Design [15, p. 11]

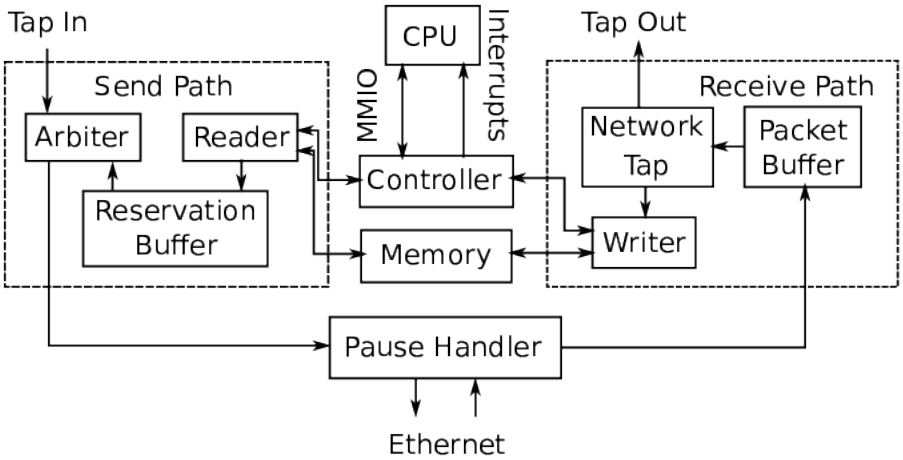

Figure 2.4: Icenet Design

Icenet's controller works by exposing a set of MMIO (Memory-Mapped Input/Output) registers to the CPU. These registers define where in memory to read data from/write data to. The design also has a reservation queue buffer so that TileLink responses that come out of order are provided to the CPU in the proper order. Icenet also provides a kernel driver so that the NIC has a full Linux networking stack in userspace.

2.4.7 NVDLA

NVDLA, short for NVIDIA Deep Learning Accelerator, is a domain-specific SoC, designed for deep learning. It is optimized for deep learning tasks, such as Convolutional Neural Networks and computer vision. It is targeted for edge-computing devices, primarily for IoT (Internet of Things). This particular SoC design was *not* investigated, as it is both proprietary and outside the scope of this research.

2.4.8 RISC-V Sodor

Sodor is a collection of several different simple integer pipelines for RISC-V. Each one implements the full RISC-V 32b user-level integer base (RV32I). **None** of the cores support virtual memory, and all of them interface with a simple asynchronous single-cycle block of memory.

There are five different pipelines:

1-Stage An ISA simulator.

2-Stage Demonstrates pipelining in Chisel.

3-Stage Uses sequential memory, supporting both Harvard and Princeton CPU architectures.

5-Stage Toggle between bypassed or interlocked memory.

Bus-Based A CPU based around a central bus.

This particular set of cores would be most applicable for courses in computer architecture, as the cores are relatively simple and easy to modify. In addition, the repository for these cores also already includes undergraduate laboratory materials.

2.4.9 Rocket-Chip

Rocket Chip is a SoC design generator that outputs synthesizable RTL (Register Transfer Language). This allows for this SoC to be composed of computer-generated cores, caches, and interconnects. The Rocket Chip is a general-purpose processor core that uses the RISC-V ISA, and includes support for virtual memory. It is a superset of the BOOM, because it provides both an in-order execution core generator (Rocket), and the specialized BOOM configuration for out-of-order execution. A diagram of the Rocket Chip's design is shown in Figure 2.5.

Rocket is different from other chips because it can be used to create larger heterogenous SoCs. This means that a single package can be composed of not only Rocket CPUs, but *also* custom accelerators, co-processors, or completely independent cores, or even a mix of all three.

Overall, the Rocket chip is an SoC design that is most useful for general-purpose computing platforms and CPU and FPGA research. Its open development platform and Extensible design allows for anyone to use the already defined designs and extend on top of them. The Rocket Chip design is the paragon of RISC-V extensibility due to its support for adding additional processing units on-board that perform tasks in hardware rather than software.

2.4.10 SHA3

The SHA3 generator is a parameterized accelerator compute unit, like the Gemmini and Hwacha. Because, it is parameterized, its overall characteristics and funtionality can change based on the input parameters to the design system. It was designed to be an example of making an add-on computational unit for the

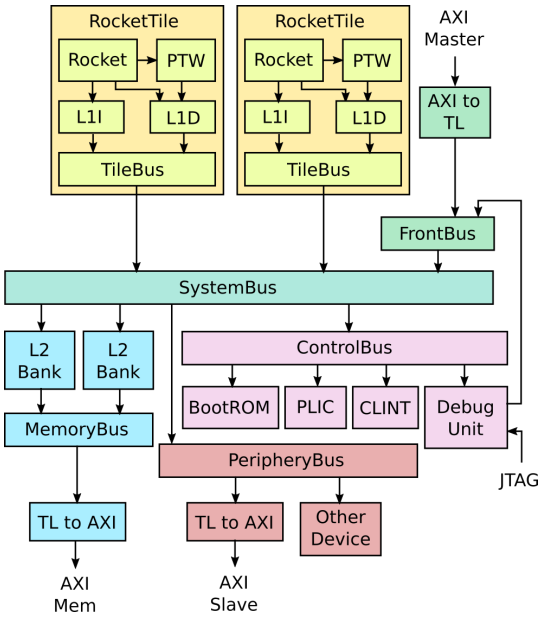

Figure 2.5: Rocket Chip Design [2]

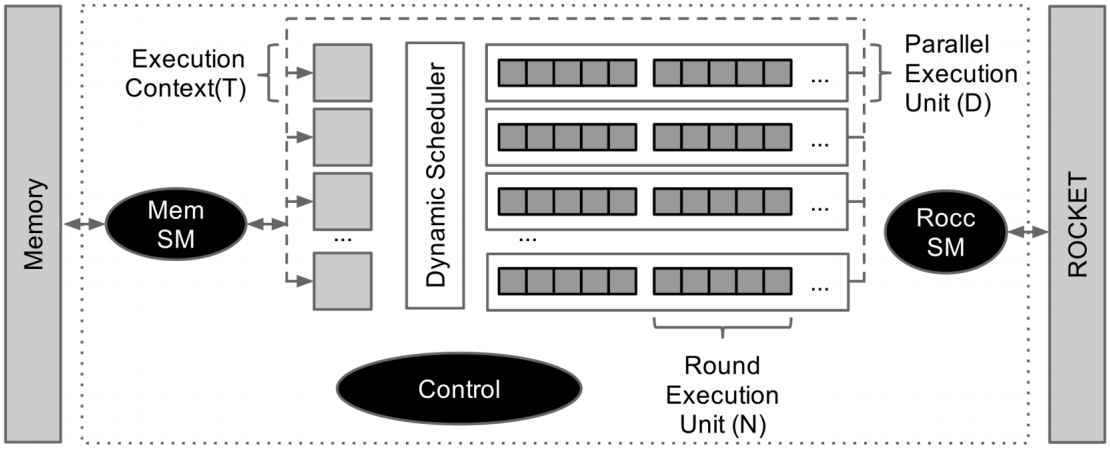

Figure 2.6: SHA3 Accelerator Design [14, p. 3]

Rocket-Chip and BOOM cores. The overall design of the accelerator is given in Figure 2.6, drawn from [14] and fetched from [30].

The design is a fully featured SHA3 computing unit that can be given the memory address whose contents are to be hashed, the computation to perform, and the memory location where to store the result. All of this is done in a "set and forget" way, where the main processor gives the accelerator the required information and the accelerator is free to run on its own.

To add a SHA3 accelerator to your Rocket-Chip or BOOM design, follow the example code shown in Listing 2.2.

```
1 class Sha3RocketConfig extends Config(
2   new sha3.WithSha3Accel ++                                    // add SHA3 rocc accelerator
3   new freechips.rocketchip.subsystem.WithNBigCores(1) ++
4   new chipyard.config.AbstractConfig)
```

Listing 2.2: Add SHA3 Accelerator to Rocket Design

2.4.11 SiFive Blocks

This generator is mainly intended to be used as a building block for other, higher-level, modules. It defines many common RTL blocks that are typically used for SiFive's other projects. Because Chipyard requires integration with many other technologies, they also rely on the code that is already defined in the `sifive-blocks` repository.

2.4.12 SiFive Cache

Like SiFive Blocks, SiFive's cache is intended to build higher-level modules. It defines the necessary terms to paramterize and generate the required RTL to create a last-level inclusive cache. Cache coherence is enforced using an invalidation policy. SiFive Cache is intended to be a drop-in replacement for Rocket-Chip's `tilelink.BroadcastHub` coherence manager.

2.4.13 `testchipip`

This last generator module is used to integrate proprietary intellectual property (IP) components with the rest of the generated system. It provides:

1. Clock utilities
2. Utilities to interface SERDES to and from TileLink
3. Custom serial interfaces for debugging with simulator interface
4. TileLink splitter, switcher
5. Several other components

2.5 Custom Configurations

If defining a custom configuration that is based on the work already done in Chipyard and you want to pull in the all the other work from all other generators, then the new configuration class **must** be defined in the `chipyard/generators/chipyard/src/main/scala/config/` directory.

> Note that this case is distinctly different than when you define your own custom parameterized RTL Verilog to generate **new** modules. In the case you are defining a new processor design, but you want to continue building off the work of the code that is already defined, you **must** place your custom class in Chipyard's `config/` directory.
> If you are defining a completely new design (a new accelerator for instance), then you can separate this definition out to a different directory in the `chipyard/generator/` directory entirely. You can also track that new directory as a git submodule of Chipyard. Then, Chipyard can import and dynamically handle your new project as a dependency of a higher-level definition that generates the CPU design.

In this section, we show how to create a new custom configuration that combines the already-existing projects. We will be building a processor using four medium-sized Rocket-Chip general-purpose cores, the default memory configuration, and will attach a SHA3 accelerator to the design for hardware-accelerated SHA3 calculation.

We will cover how to write a generated design image out to an FPGA in Chapter 4, rather than here, because the process is the same for every generated chip.

2.5.1 File and Class Creation

Start by creating the `NewTestConfig.scala` file in the `chipyard/generators/chipyard/src/main/scala/config/` directory. Inside of this file, we will start defining the `NewTestConfig` class, which will declare the desired configuration.

> Example configurations of various homogenous and heterogenous chips can be found in `RocketConfigs.scala`, in the same directory.

We want our processor design to be quite simple, so we are designing a processor that uses only in-order execution. In addition, we want to design a processor that has multiple cores on it because a single CPU design for general-purpose use is not enough today. Lastly, we want to include a SHA3 accelerator module to allow us to perform SHA3 computations in hardware.

In short, our design will have:

1. Multiple Rocket-Chip cores. We are going to start with four medium-sized cores. There are both larger and smaller sizes already defined.
2. A single SHA3 accelerator.

Using the design parameters we have defined, we can write a processor definition class that *declaratively* describes the resulting CPU we want. The code for this is shown in Listing 2.3 and is also available alongside the source code for this document.

2.5.2 Building

You will need to move to the `chipyard/sims/verilator` directory before building your design. To build the design, it is as simple as `$ make CONFIG=NewTestConfig`.

We strongly recommend that:

- You alter the amount of memory you allocate to the JVM (Java Virtual Machine), as having too little can cause the elaboration to fail.
- You parallelize the building by passing `make` the `-j N` flag, `make -j N CONFIG=NewTestConfig`. If you do *not* pass the integer `N` to the `-j` flag, then Chipyard will use all available processors.

```
package chipyard
import freechips.rocketchip.config.{Config}
import freechips.rocketchip.diplomacy.{AsynchronousCrossing}

class NewTestConfig extends Config(
    new sha3.WithSha3Accel ++                                    // Add SHA3 accelerator
    new freechips.rocketchip.subsystem.WithDefaultMemPort ++ // Default Rocket chip
     ↪  memory subsystem configuration
    new freechips.rocketchip.subsystem.WithNMedCores(4) ++   // 4 Medium-sized In-Order
     ↪  Rocket Cores
    new chipyard.config.AbstractConfig)
```

Listing 2.3: NewTestConfig.scala Contents

> When building your custom configurations, you will likely require *significantly* more memory than what was required for the example design described in Section 1.5.1. You will likely also need to change the amount of heap memory the JVM has available to it. This can be done by editing the chipyard/variables.mk file. Change the line JAVA_HEAP_SIZE ?= 8G to a larger value (fractional values can be used, but are not recommended).

The passing of CONFIG to the make command is because of the values defined in chipyard/variables.mk. See Section 2.2.2 for a more detailed explanation of what each variable is designed to do.

Notes about Altering JVM Behavior

The arguments the JVM uses when running FIRRTL, Chisel, and all other programs written in Scala and/or Java are controlled by the chipyard/variables.mk file.

As of the time of last editing this document (June 24, 2021), the portion of the file concerned with this is shown in Listing 2.4.

```
###################################################################################################
# java arguments used in sbt
###################################################################################################
JAVA_HEAP_SIZE ?= 8G
JAVA_OPTS ?= -Xmx$(JAVA_HEAP_SIZE) -Xss8M -XX:MaxPermSize=256M
```

Listing 2.4: Altering JVM Behavior

By editing the JAVA_OPTS variable, additional parameters can be specified for *every* JVM invocation. This would be the location to inform the JVM how large a thread's stack may be, the garbage collection algorithm to use, an any other behavior-altering configurations.

2.5.3 Testing

When running *any* make command on your custom configuration, you *must* provide the CONFIG parameter. Every time you want to simulate or run tests on your new design and use make, you must specify everything needed to select that particular design. For example, to run the RISC-V compliance tests with our example custom design, you must enter `$ make CONFIG=NewTestConfig` (you can also add VERILATOR_THREADS=N to parallelize the execution of Verilator). If additional make variables were defined when the design was built, you must include them in the simulation command as well.

> If you do not provide the necessary `CONFIG` parameter, `make` will assume you are running the default processor design and look for that particular set of files. You can configure what `make` will assume is the default by editing the `chipyard/variables.mk` makefile.

2.5.4 Simulating

Just like when Testing the design, when simulating the design using `make` as a helping tool, you must provide the necessary makefile variable configurations every time you simulate/run a program.

Chapter 3

Simulators

Chipyard currently has support for three simulators:

1. Verilator
2. VCS
3. FireSim

Each of these are perfectly suitable for their task. However, each one of these comes with its own benefits and drawbacks. These will be discussed in their respective sections.

3.1 Verilator

Verilator is an open-source (System)Verilog compiler and simulator. Because it is open-source and written in a relatively high-level language (C/C++), it can be compiled to any platform. In addition, it offers a wide range of functionality for Chipyard. However, its biggest drawback right now is that it cannot perform validation simulations of procesor designs that would be implemented on a FPGA.

Throughout the entire research project that was conducted, this was the only simulator that was used. This was due to our relatively minor requirements of the simulations and our focus on proceesor implementation on local FPGA hardware. However, in the laboratory, this would be an invaluable feature for students to have available to them, as FPGAs are likely not always available.

3.2 VCS

VCS is a closed-source, proprietary (System)Verilog simulator and verifier. It is trusted by some of the largest hardware design companies in the world and is quite powerful. It is currently used to simulate designs using all the available features of x86-based microprocessors. In addition, VCS is the only Verilog simulator that can be used to simulate the Arty FPGA. Lastly, Arty support in VCS is still in active development, and is only on Chipyard's git `arty-sim` branch.

Due to the proprietary nature of this product, this particular (System)Verilog simulator was not investigated. Verilator suited our needs, and provided the right amount of support that we needed. However, in a larger organization, or one that requires formal verification of their design, this would be the appropriate tool to use.

3.3 FireSim

FireSim is an interesting technology that allows a system designer to upload a generated design to AWS (Amazon Web Services) and test it there. By using the Icenet generator, the FPGA design can be made to have networking capabilities. AWS/Amazon is then able to write this bitstream out to their FPGAs for closer to

real-time testing. The value of this is that near normal FPGA speeds can be reached from an environment that appears to be composed completely in software (from the developer's point of view).

Because of the limited resources of the original developers, FireSim was not investigated. However, this has the opportunity to be an invaluable stepping stone in the processor design process. By simulating an processor at near-FPGA speeds in an environment the developer sees as software, more rapid prototyping is possible.

Chapter 4

FPGA Implementation

This chapter is devoted to discussing how to implement a Chipyard-generated processor design on a **local** FPGA for quicker testing and general use. Throughout the research project this manual was originally completed in, the Arty FPGA was used. An image of the Arty board can be seen in Figure 4.1. The Arty board is built using a Xilinx FPGA module and then Arty creates a board surrounding the particular chip.

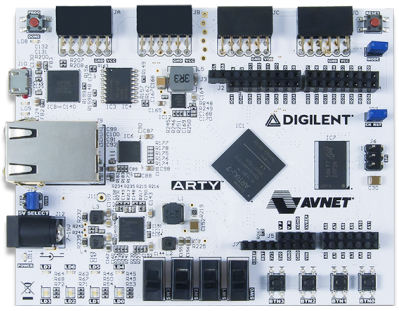

Figure 4.1: Arty FPGA

4.1 About

The Chipyard Framework contains initial support for FPGA development and simulation of SoC designs. At the moment this support is very limited, and is in active development. As of June 24, 2021, the best support for FPGA Development for the Arty 35T FPGA comes from a branch of Chipyard called arty-spi-flash. This branch fixes the UART (Universal Asynchronous Reciever-Transmitter) implementation and enables the SPI (Serial Peripheral Interface) flash storage on the Arty FPGA to allow users to store programs.

4.2 Prerequisites

To assist with the proper setup, we approached the FPGA implementation of an SoC by following the "SiFive Freedom E310 Arty FPGA Dev Kit Getting Started Guide" [19]. This outlined many of the steps we would eventually need to take, starting with purchasing an Olimex JTAG Debugger [4]. Once the final image is flashed to the FPGA, the debugger will allow the user to upload C programs and execute them on the RISC-V processor. Without the JTAG (Joint Test Action Group) Debugger, we were unable to upload programs to the FPGA, so this is a necessity.

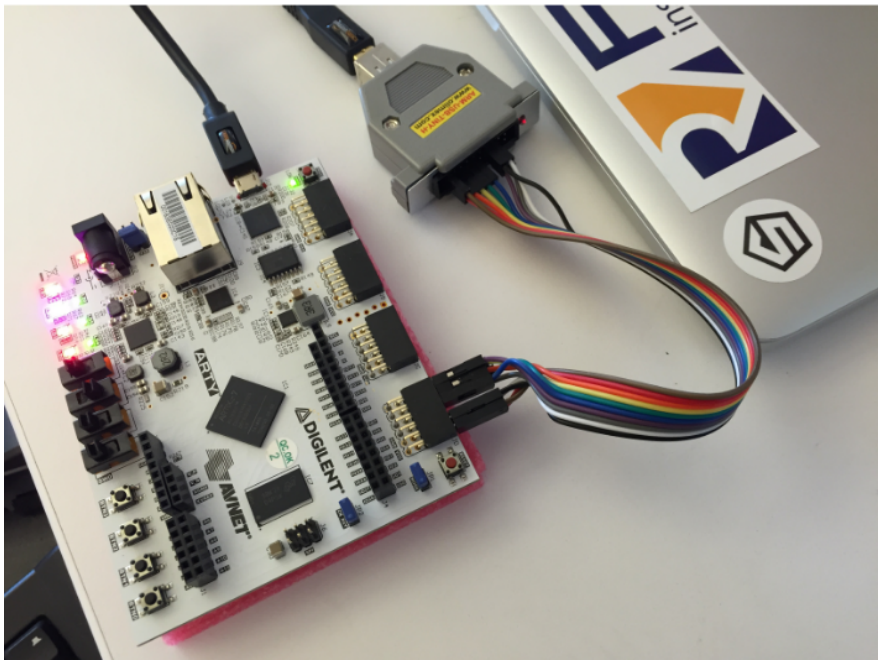

Figure 4.2: Olimex Debugger Setup [19, p. 5]

Signal Name	ARM-USB-TINY-H Pin Number	Suggested Jumper Color	Arty PMOD JD Pin Number
VREF	1	red	12
VREF	2	brown	6 (VCC)
nTRST	3	orange	2
TDI	5	yellow	7
TMS	7	green	8
TCK	9	blue	3
TDO	13	purple	1
GND	14	black	5 (GND)
nRST	15	grey	9
GND	16	white	11

Table 4.1: Olimex Pin Connections Setup [19, p. 4]

4.3 Customizing an FPGA Image

In Chipyard, the directory used for all FPGA prototyping functionality is chipyard/fpga, located in the root directory. Inside this directory there are several important files.

4.3.1 Configuration Directory

The configuration directory for the Arty FPGA is located under chipyard/fpga/src/main/scala/arty/. This directory includes several useful files, including Configs.scala, HarnessBinders.scala, IOBinders.scala, and TestHarness.scala.

Configs.scala

This file is where custom Chipyard SoC configurations are stored and utilized for FPGA prototyping. There are several custom configuration parameters for the Arty FPGA in this file. The default configuration utilized to make an image for the Arty board is labeled as `TinyRocketArtyConfig`, which then utilized the custom parameters defined for `WithArtyTweaks` and `WithDefaultPeripherals` in the same file. The addresses specified in the `WithDefaultParameters` configurations alter how the memory mapped peripherals for the Arty FPGA will be connected.

HarnessBinders.scala

This file is where custom "harness binders" [1] for the Arty FPGA are defined. These harness binders utilize pins specified in the master `ArtyShell` pin definitions file for the Arty board (provided by Digilent for use in Xilinx Vivado). This file is located at chipyard/fpga/fpga-shells/src/main/scala/shell/xilinx/ArtyShell.scala. The pin mappings in this file are the same mappings that would be utilized when interfacing with the Arty board when designing in Xilinx Vivado. Harness binders in this file are provided for the JTAG interface, SPI flash, and the UART connector. These three connections are critical to the FPGA design, and to ensuring that programs can be successfully uploaded and run on the FPGA. It is important to note that the harness binders connects to the `TestHarness`, and not the physical IO [26]. This was done such that separation could be created between the simulated and physical designs. To connect to the physical IO pins, IOBinders are also needed.

IOBinders.scala

This file is where the physical IO pins are connected to the harness binders defined previously. In this file are custom configurations for the SPI flash and JTAG connectors that will be utilized in the default design. In the future, additional IOBinders for other peripherals on the Arty FPGA should be implemented.

TestHarness.scala

This file is where miscellaneous connections are made between pins, global clock and reset variable are defined, and the Harness Binders are actually applied to the SoC design. No modification to this file should be needed in order to implement new peripheral devices in the future.

The generated-src directory is the directory in which all files created from compiling the SoC design and FPGA image will be stored. This means that the memory map, FPGA bitstream, and other important files will be stored in this directory. This directory can be found at chipyard/fpga/generated-src. This directory will not be created until a FPGA design run is initiated.

4.3.2 Makefile

Inside the Makefile is where you are able to define a custom subproject as shown in Section 2.2.1 and Section 2.2.2. This allows users to control what files are compiled and generated for the FPGA image. This is highly recommended as it simplifies the workflow for repeated compilation attempts.

[1] Harness Binders are utilized by Chipyard to connect the name of pins in HDL to the pins in the `TestHarness` Verilog model. To connect to the physical pins of the FPGA board, IOBinders will map the TestHarness pins to the physical pins of the FPGA.

4.4 Generating the FPGA Image

4.4.1 Syntax

Much like generating a verilog simulation, there are many options when it comes to generating an FPGA image in Chipyard. The complete syntax is as follows: [27]

```
$ make SBT_PROJECT=... MODEL=... VLOG_MODEL=... MODEL_PACKAGE=... CONFIG=...
  ↪ CONFIG_PACKAGE=... GENERATOR_PACKAGE=... TB=... TOP=... BOARD=... FPGA_BRAND=...
  ↪ [-j[N]] bitstream
```

Listing 4.1: Command to generate FPGA image using long format

The condensed syntax, as implemented in section 4.3.2 is as follows:

```
$ make SUB_PROJECT=<sub_project> [-j[N]] bitstream
```

Listing 4.2: Command to generate FPGA image using subproject.

```
$ ifeq ($(SUB_PROJECT),artyCustom) # Customize SUB_PROJECT name to artyCustom
$         SBT_PROJECT       ?= fpga_platforms
$         MODEL             ?= ArtyFPGATestHarness #Still utilize Arty Test Harness
$         VLOG_MODEL        ?= ArtyFPGATestHarness
$         MODEL_PACKAGE     ?= chipyard.fpga.arty
$         CONFIG            ?= ArtyCustomConfig #Change SoC configuration to build
$         CONFIG_PACKAGE    ?= chipyard.fpga.arty #Package where custom config can be found
$         GENERATOR_PACKAGE ?= chipyard
$         TB                ?= none # unused
$         TOP               ?= ChipTop
$         BOARD             ?= arty
$         FPGA_BRAND        ?= xilinx
$ endif
```

Listing 4.3: Example of sub project variables customization

It is highly recommended to make use of the `-jN` flag to allow multi-threading and speed up generation. Without multi-threading the process of generating an image can take multiple hours.

When generating the default Arty image, we noticed that the script would produce an error about `Failed to meet timing by $timing_slack`. However, we found that the image still appears to run correctly, so further investigation is needed into why this error occurs.

Upon successful generation, the bitstream file will be saved to `chipyard/fpga/generated-src/<long_project_name>/obj/<ConfigName>.bit`.

The branch of chipyard with the best functionality for the Arty board is currently the `arty-spi-flash` branch, so use of this branch is recommended.

4.4.2 Creating MCS File

After generating a bitstream file, one should utilize Xilinx Vivado to create a MCS (Memory Configuration File Format) file in order to save the design to the SPI flash on the Arty board. This will allow the design to be automatically reloaded onto the Arty board after power is disconnected.

To begin, open Xilinx Vivado, and enter the `Hardware Manager` (shown in Figure 4.3). Under the `Tools` dropdown, select `Generate Memory Configuration File`.

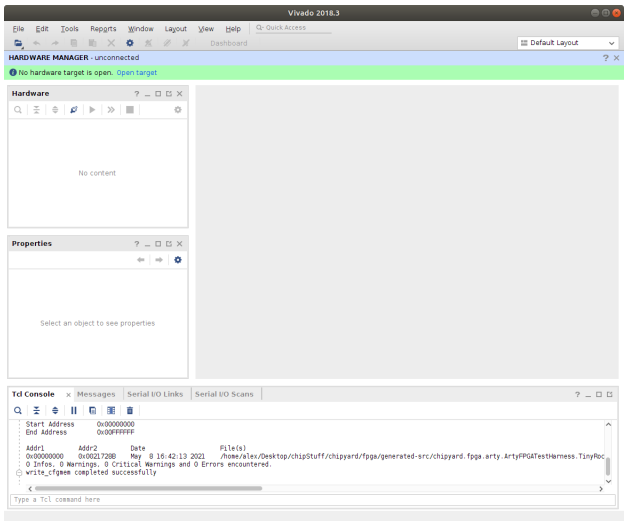

Figure 4.3: Vivado Hardware Manager Window

Inside this wizard (shown in Figure 4.4), there are several options that must be filled.

Memory Part Select the memory part present on your specific FPGA board. For the Arty board we used, this was the `s25fl128sxxxxxx0-spi-x1_x2_x4` device.

Filename Specify a new output location and name for the MCS file that will be generated.

Interface Specify SPIx4 for the Arty board. Other options include a x1 or x2 wide SPI interface.

Bitfile Select the bitstream file generated previously by Chipyard.

Datafile In theory, this should allow one to upload a .hex or .elf program file to be run by the Arty board, however in our experience we had better success when uploaded using the JTAG debugger after flashing the FPGA.

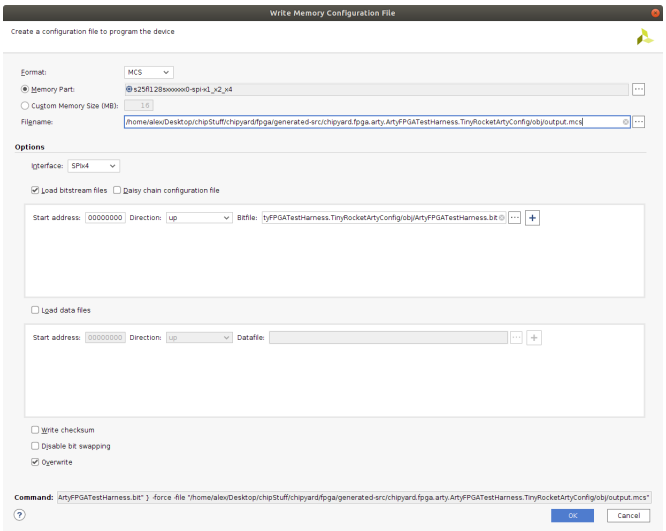

Figure 4.4: Vivado MCS Generation Window

4.5 Using the FPGA Image

4.5.1 Flashing the Image

In order to flash the MCS file generated in Section 4.4.2, first open Vivado Hardware Manager (Figure 4.3) and connect the Arty board via USB. Inside Hardware Manager, click the `Open target prompt` and select `Auto Connect` at the top of the window to connect to the Arty board. In the Hardware section of the window, the Arty board will now be shown as `xc7a35t_0` (shown in Figure 4.5).

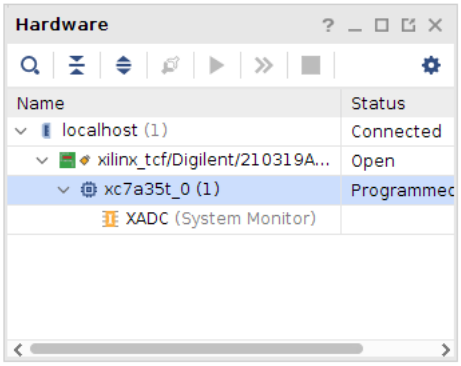

Figure 4.5: Hardware section of Vivado Hardware Manager

Next, right-click on `xc7a35t_0`, select `Add Configuration Memory Device`, and navigate to the same memory configuration device selected previously (Figure 4.6). After clicking `OK`, you will be prompted to program the configuration memory device (Figure 4.7). In this prompt, select the MCS file generated previously. After selecting `OK`, flashing of the image will commence. At this point, if the wrong memory part was selected, you will receive an error message depicting the correct memory part. If this occurs, repeat creating the MCS file (Section 4.4.2), adding the memory configuration device, and try again.

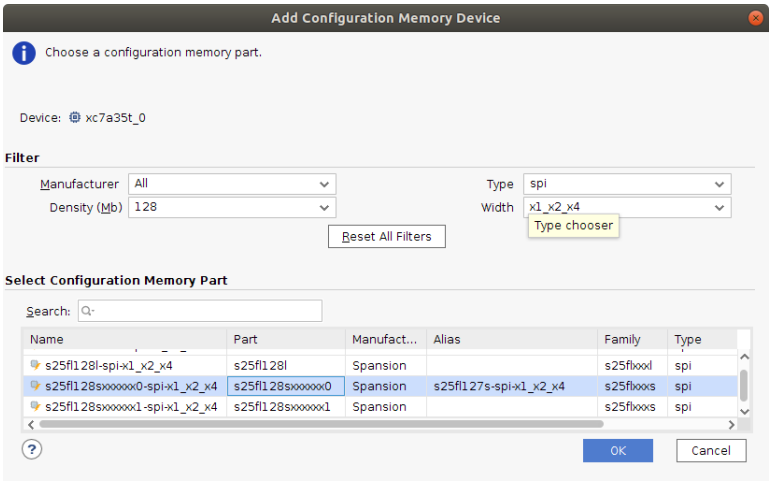

Figure 4.6: Add Configuration Memory Device

After the flashing of the MCS file, ensure that jumper JP1 is shorted so that the FPGA boots from the SPI flash, and press the `PROG` button on the Arty board. The `DONE` LED on the Arty board should be lit, indicating that flashing the FPGA image was a success. We can now proceed to compile and upload C programs to the FPGA.

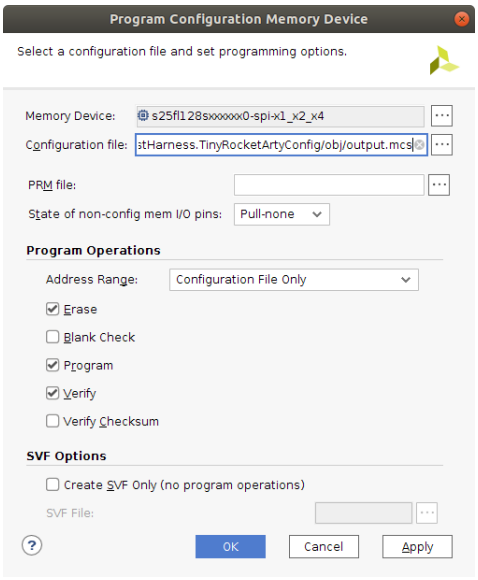

Figure 4.7: Program Configuration Memory Device

4.5.2 Compiling Programs

In order to compile programs for the Arty board, we utilized the Freedom E SDK (Mentioned in Section 1.7.1) project provided by SiFive [17]. To correctly configure the Freedom E SDK for the Chipyard Arty implementation, we must first specify the JTAG debugger that will be used in this project. This can be done in the file `/freedom-e-sdk/bsp/freedom-e310-arty/openocd.cfg`. In this file, edit line 9 to `set protocol jtag` and line 14 to `set connection probe`. These settings will allow the Freedom E SDK to utilize the Olimex debugger purchased for this project.

Documentation for the Freedom E SDK can be found at the Freedom Metal Library Github page. It is important to note that much of the functionality regarding physical features (GPIO, Buttons, LEDs, etc.) are currently not functional on the Chipyard SoC implementation for the Arty. Additionally, we found that some functionality in the Metal library was buggy. For example, sending output to the serial terminal did not work when using `printf()`, but did work when utilizing `putc()`. In the future, compatibility and functionality should be improved for this C library.

In the Freedom E SDK, default and custom projects are defined in the directory `/freedom-e-sdk/software/`. To compile a project for the Arty board, run the following command:

```
1 $ make TARGET=freedom-e310-arty PROGRAM=<program> CONFIGURATION=debug software
```

Listing 4.4: Command used to compile a program for the Arty board in the Freedom E SDK

4.5.3 Uploading Programs to the FPGA

Before running a program on the Arty board, it is important to start a serial connection to the Arty board, so that you can view the serial output from the Arty board. Other documentation has declared that the design should output to UART at a baud rate of 115200, however we have found that the correct baud rate is 57600. The serial terminal we utilized for this project is CuteCom [11], due to its ease of adjusting settings and graphical interface, although other options include GNU Screen, minicom, PuTTY, etc.

To load a default or custom project onto the Arty board, use the following syntax:

Pressing the reset button on the Arty board will cause the board to reinitialize and run the program again.

```
$ make TARGET=freedom-e310-arty PROGRAM=<program> CONFIGURATION=debug upload
```

Listing 4.5: Command used to flash a program to the Arty board

Other useful make targets include debug, simulate, clean, and help. For example, to simulate the `hello` program in SPIKE [22], use the following command:

```
$ make TARGET=spike PROGRAM=hello CONFIGURATION=debug simulate
```

Listing 4.6: Command used to simulate a program in SPIKE

Chapter 5

Future Work

In this chapter, we present opportunities that we feel can be leveraged due to the initial work we did during this research project. We have chosen to break these opportunities up into several categories, each presented in a section below.

5.1 Additional Research

Chipyard is a large and very complicated piece of software. Simply getting over the initial hurdle of getting Chipyard to work and getting the generated FPGA image written out comprised a majority of our work. If we had more time to investigate Chipyard and become more familiar with its inner workings, we would like to further explore:

- Creating custom heterogenous CPU designs, elaborating them, and writing them out to an FPGA.
- Booting a minimal Linux kernel on the generated Softcore.
- Adding additional peripheral device functionality for the Arty and other FPGA boards in Chipyard (buttons, LEDs, GPIO, etc.)
- Writing a new C programming library for the Chipyard FPGA framework, and incorperating JTAG debugger functionality directly into the framework.
- Modeling performance of Softcore RISC-V designs versus discrete implementations.

5.2 Academic Applications

RISC-V offers a lot to the academic world because it is a open-source CPU design. This offers the chance to investigate the inner workings of the CPU and its implementation significantly more than many other architectures. In fact, because RISC-V has been created by the University of California, Berkeley, there are already academic materials available for use.

In the sections below, we discuss several courses offered at Illinois Institute of Technology that, we believe, would be perfect candidates for a revamp using RISC-V and Chipyard.

5.2.1 ECE 242 — Digital Computers and Computing

ECE 242 is intended to be an introductory course to some of the lowest levels of digital computing. Namely, this involves in-depth discussion around both CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) architectures, their differences, and how to write assembly [7].

Currently, the RISC architecture taught is MIPS. We feel that this is not going to help students in their future work, so we suggest teaching RISC-V instead. The base instruction set is not much more complicated, with just forty-seven (47) instructions for the user-level 32-bit integer instruction set [13, pp. 9–26]. Because

the RISC-V ISA is significantly more modern, many of the concepts learned here will still translate to other RISC ISAs.

By virtue of being more modern, students will gain an appreciation and knowledge of an architecture that they are far more likely to encounter in their career. This is because RISC-V supports mixing and matching extensions, so the processors can actually be designed for anything from an embedded microcomputer to high performance computing.

5.2.2 ECE 441 — Microcomputers and Embedded Computing

ECE 441 teaches the concept of embedded computing in more depth. It handles more advanced microprocessor features, such as hardware interrupting, memory design, and MMIO [8].

In previous iterations of this course, the SANPER-1 Educational Lab Unit was used. This system is based on the Motorola 68000 series microprocessor. Although there would be some work required to move from a CISC architecture to a RISC one, we feel it is appropriate given how the world has already and will continue to move forward. Although the principles in the MC68k are sound, they are also quite outdated. We suggest a RISC-V CPU that implements the RV32E integer instruction set [13, p. 25] because this course is particularly focused on embedded computing.

RISC-V processors can be built with the ability to have their bus cycles interrupted, which is a key feature of the SANPER. In addition, highly desired features can be implemented in the RISC-V processor's hardware, by extending the already defined instructions with new ones. This customization and flexibility is already making RISC-V a major competitor in the industrial world. Taking this same flexible system and bringing it to academia will allow for further RISC-V environment maturation and more academically up-to-date graduates.

Taking this a step further, to keep the devices up-to-date, using an FPGA might be appropriate as well. This would allow for greater diversity in CPU design exposure during hte actual laboratory session. If need be, key functionality can be added to the system between laboratory sessions.

5.2.3 ECE 485 — Computer Organization and Design

ECE 485 is designed to teach fundamental concepts of computer architecture, organization, and design [9]. All of these topics can be covered and explored in even *more* depth by having access to an Extensible ISA.

Many of the sub-projects that Chipyard makes use of would be very appropriate for a reworked version of this course. The cva6 and RISC-V Sodor would be perfect for this course. They are already very small designs, implementing minimal functionality.

The RISC-V Sodor design would be best for introducing topics, because it has multiple stages that support different levels of simulation. Because the simulations are done completely in software, there is minimal student overhead for testing new designs and learning about how the system is designed. The different stages allow for students to view and simulate progressively more complex CPU designs.

The cva6 could be used as a simpler example of a full CPU design, as it supports multiple extensions and multiple privilege levels. Because the cva6 is a single issue, in-order design, the circuitry is less complex than similar chips (Rocket-Chip and BOOM). This makes the cva6 perfect to show component integration onto a single device.

We believe focusing the revamped version of this course around an FPGA would also be most appropriate, as students could make new designs on-the-fly and test them. This could open a completely different world up to this course. Being able to not only learn about processor architecture and design, but the chance to implement this functionality on an FPGA would make everything being learned tangible.

About the Authors

Alexander Lukens

Alexander is an undergraduate engineering student studying towards a Bachelor's Degree in Electrical Engineering at Illinois Institute of Technology. He is entering his final semester of study, with an emphasis on FPGA design and Computer architecture. When not pursuing his degree, Alexander works at the Idea Shop Prototyping Lab where he helps students from all majors design meaningful models. He is proficient in using electronics prototyping workstations, 3D Printers, Laser Cutters, CNC Mills, and CAD software. My current areas of interest are SoC design and rich web-app development After graduation, Alexander intends to pursue a graduate degree in Computer Engineering with a focus on integrated circuit design. He is passionate about open source software, and intends to give back to the Computer Engineering community via contributing to open source projects.

Alexander Lukens

Karl Hallsby

Karl is a co-terminal student pursuing his Bachelor's and Master's of Science in Computer Engineering at Illinois Institute of Technology. He is currently in his fourth year of undergraduate studies, and is preparing for his graduate work. He has experience in circuit analysis, hardware design, systems programming, application programming, and programming language theory and design. Karl is also a member of the CyberHawks cybersecurity student organization, and has been involved in capture-the-flag competitions and protocol analyses. His professional interests lie in hardware/software co-design, operating systems, and computer cybersecurity, and further extend to programming languages and exotic/novel/esoteric operating systems.

Karl Hallsby

Bibliography

[1] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs." In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[2] Krste Asanović et al. *The Rocket Chip Generator*. report. Apr. 15, 2016. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf (visited on 04/09/2021).

[3] Chips Alliance. *Rocket Chip Generator*. Apr. 9, 2021. URL: https://github.com/chipsalliance/rocket-chip (visited on 04/12/2021).

[4] Digikey. *ARM-USB-TINY-H*. 2021. URL: https://www.digikey.com/en/products/detail/olimex-ltd/ARM-USB-TINY-H/3471388.

[5] Farzad Farshchi, Qijing Huang, and Heechul Yun. "Integrating NVIDIA Deep Learning Accelerator (VNDLA) with RISC-V SoC on FireSim." In: *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)* (2019), pp. 21–25. URL: https://arxiv.org/pdf/1903.06495.pdf (visited on 04/12/2021).

[6] Hasan Genc et al. "Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures." In: *University of California, Berkeley* (Dec. 7, 2019). DOI: 1911.09925. URL: https://arxiv.org/pdf/1911.09925.pdf (visited on 04/09/2021).

[7] Illinois Institute of Technology. *ECE 242: Computer Organization and Design*. Illinois Institute of Technology. 2021. URL: http://bulletin.iit.edu/search/?P=ECE%20242 (visited on 05/05/2021).

[8] Illinois Institute of Technology. *ECE 441: Computer Organization and Design*. Illinois Institute of Technology. 2021. URL: http://bulletin.iit.edu/search/?P=ECE%20441 (visited on 05/05/2021).

[9] Illinois Institute of Technology. *ECE 485: Computer Organization and Design*. Illinois Institute of Technology. 2021. URL: http://bulletin.iit.edu/search/?P=ECE%20485 (visited on 05/05/2021).

[10] Sagar Karandikar, David Biancolin, and Alon Amid. *FireSim: Scalable FPGA-acceleratred Cycle-Accurate Hardware Simulation in the Cloud*. RISC-V Summing 2018 Tutorial. 2018. URL: https://riscv.org/wp-content/uploads/2018/12/Tutorial-Easy-to-use-FPGA-Accelerated-Hardware-Simulation-of-RISC-V-Hardware-Designs-with-FireSim-on-Amazon-EC2-F1-Amid-Karandikar-Biancolin-.pdf (visited on 04/09/2021).

[11] Alexander Neundorf. *CuteCom*. May 8, 2021. URL: http://cutecom.sourceforge.net/ (visited on 05/08/2021).

[12] OpenHW Group. *cva6*. 2021. URL: https://github.com/openhwgroup/cva6 (visited on 04/09/2021).

[13] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. Ed. by Andrew Waterman and Krste Asanović. May 7, 2017. URL: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (visited on 05/03/2021).

[14] Colin Schmidt and Adam Izraelevitz. "A Fast Parameterized SHA3 Accelerator." In: *University of California Berkeley - Electrical Engineering and Computer Sciences* (n.d.). URL: https://people.eecs.berkeley.edu/~colins/papers/SHA3.pdf (visited on 04/20/2021).

[15] Colin Schmidt, Alber Ou, and Krste Asanović. *Hwacha V4: Decoupled Data: Parallel Custom Extension*. University of California, Berkeley. Dec. 2018. URL: https://riscv.org/wp-content/uploads/2018/12/Hwacha-A-Data-Parallel-RISC-V-Extension-and-Implementation-Schmidt-Ou-.pdf (visited on 04/09/2019).

[16] SiFive. *block-inclusivecache-sifive*. Apr. 16, 2021. URL: https://github.com/sifive/block-inclusivecache-sifive (visited on 04/20/2021).

[17] SiFive. *Freedom E SDK*. 2021. URL: https://github.com/sifive/freedom-e-sdk.

[18] SiFive. *Freedom Tools*. 2021. URL: https://github.com/sifive/freedom-tools.

[19] SiFive. "SiFive Freedom E310 Arty FPGA Dev Kit Getting Started Guide." In: (2017). URL: https://www.sifive.com/documentation/freedom-soc/freedom-e300-arty-fpga-dev-kit-getting-started-guide/.

[20] SiFive. *sifive-blocks*. Jan. 5, 2021. URL: https://github.com/sifive/sifive-blocks (visited on 04/20/2021).

[21] Fans Sijstermans and Yunsup Lee. *NVIDIA's Deep Learning Accelerator Meets SiFive's Freedom Platform*. 2018. URL: https://riscv.org/wp-content/uploads/2018/12/Nvidias-Deep-Learning-Accelerator-Meets-SiFives-Freedom-Platform-Frans-Sijstermans-and-Yunsup-Lee.pdf (visited on 04/12/2021).

[22] Regents of the University of California. *Spike RISC-V ISA Simulator*. 2021. URL: https://github.com/riscv/riscv-isa-sim.

[23] University of California, Berkeley and FireSim. *icenet*. University of California, Berkeley. Dec. 13, 2021. URL: https://github.com/firesim/icenet (visited on 04/09/2021).

[24] University of California, Berkeley - Berkeley Architecture Research. *Gemmini*. 2021. URL: https://github.com/ucb-bar/gemmini (visited on 04/09/2021).

[25] University of California, Berkeley - Berkeley Architecture Research. *Hwacha*. 2021. URL: https://github.com/ucb-bar/hwacha/tree/62c01f5a8858aa1b827f0f9372a4392d7b596fca (visited on 04/09/2021).

[26] University of California, Berkeley - Berkeley Architecture Research. *IOBinders and HarnessBinders*. 2021. URL: https://chipyard.readthedocs.io/en/latest/Customization/IOBinders.html (visited on 05/07/2021).

[27] University of California, Berkeley - Berkeley Architecture Research. *Prototyping Flow*. 2021. URL: https://chipyard.readthedocs.io/en/latest/Prototyping/General.html (visited on 05/08/2021).

[28] University of California, Berkeley - Berkeley Architecture Research. *RISC-V BOOM*. 2020. URL: https://boom-core.org/ (visited on 04/09/2021).

[29] University of California, Berkeley - Berkeley Architecture Research. *RISC-V Sodor*. Apr. 8, 2021. URL: https://github.com/ucb-bar/riscv-sodor (visited on 04/12/2021).

[30] University of California, Berkeley - Berkeley Architecture Research. *SHA3 RoCC Accelerator*. Dec. 30, 2020. URL: https://github.com/ucb-bar/sha3 (visited on 04/20/2021).

[31] University of California, Berkeley - Berkeley Architecture Research. *SonicBoom: The Berkeley Out-of-Order Machine*. Apr. 11, 2021. URL: https://github.com/riscv-boom/riscv-boom (visited on 04/12/2021).

[32] University of California, Berkeley - Berkeley Architecture Research. *testchipip*. Jan. 11, 2021. URL: https://github.com/ucb-bar/testchipip/tree/6572beb03bc6eb0575269eaf4cc960b72b3ddef3 (visited on 04/20/2021).

[33] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[34] Jerry Zhao et al. *SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine*. report. 2021. URL: https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf (visited on 04/09/2021).

[35] Lirong Zheng et al. "Technologies, applications, and governance in the Internet of Things." English. In: *Internet of Things - Global Technological and Societal Trends*. Ed. by Ovidiu Vermesan and Peter Friess. River Publishers, 2011, pp. 141–175. ISBN: 978-87-92329-67-7.

Acronyms

AWS Amazon Web Services. 21

CISC Complex Instruction Set Computer. 31, 32, *Glossary:* Complex Instruction Set Computer

DSL Domain-Specific Language. 8, 38, 39, *Glossary:* Domain-Specific Language

ELF Executable and Linkable Format. 6, *Glossary:* Executable and Linkable Format

FIRRTL Flexible Intermediate Representation Register Transfer Language. 8, 19, *Glossary:* Flexible Intermediate Representation Register Transfer Language

FPGA Field Programmable Gate Array. 1, 7, 13, 15, 18, 21–23, 25–28, 31, 32, 39, *Glossary:* Field Programmable Gate Array

IoT Internet of Things. 15, *Glossary:* Internet of Things

IR Intermediate Representation. 38, *Glossary:* Intermediate Representation

ISA Instruction Set Architecture. 1, 10, 12, 15, 32, *Glossary:* Instruction Set Architecture

JTAG Joint Test Action Group. 23, 25, 27, 29, 31, *Glossary:* JTAG

JVM Java Virtual Machine. 18, 19, *Glossary:* Java Virtual Machine

MCS Memory Configuration File Format. 26, 28, *Glossary:* MCS

MMIO Memory-Mapped I/O. 15, 32, *Glossary:* Memory-Mapped Input/Output

NIC Network Interface Card. 13, 15

RISC Reduced Instruction Set Computer. 31, 32, *Glossary:* Reduced Instruction Set Computer

RTL Register Transfer Language. 15, 17, 18, *Glossary:* Register Transfer Language

`Sbt` Scala Build Tool. 3, 8, *Glossary:* Scala Build Tool

SIMD Single Instruction Multiple Data. 12, 13, *Glossary:* Single Instruction Multiple Data

SoC System on a Chip. 10, 15, 23, 25, 38, *Glossary:* System on a Chip

SPI Serial Peripheral Interface. 23, 25, 27, *Glossary:* Serial Peripheral Interface

UART Universal Asynchronous Receiver-Transmitter. 23, 25, *Glossary:* Universal Asynchronous Receiver-Transmitter

Glossary

Accelerator A specialized processing unit that performs a single set of tasks very effectively. These can be thought of like DSLs for hardware. Some accelerators are domain-specific SoCs, which are more like a regular CPU design, but are still not general-purpose compute units. 9, 10, 12, 13, 15, 17, 18

Assembly Any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions. Because assembly depends on the machine code instructions, every assembly language is designed for exactly one specific computer architecture.JTAG (Joint Test Action Group) Assembly language may also be called symbolic machine code. 31

Complex Instruction Set Computer A computer in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions. 31

Domain-Specific Language A computer language specialized to a particular application domain. 8

Elaboration The build process of processor design generation. This involves finding all necessary submodules and "gluing" them together using the `TileLink` standard. 1, 5, 6, 18

Executable and Linkable Format A common standard file format for executable files, object code, shared libraries, and core dumps. 6

Extensible An original product, built by someone else, can be **extended** to meet new requirements or to offer new functionality. 15, 32

Field Programmable Gate Array An integrated circuit designed to be configured by a customer or a designer after manufacturing using software. 1

Flexible Intermediate Representation Register Transfer Language An IR (Intermediate Representation) for digital circuits designed as a platform for writing circuit-level transformations. 8

Generator A singular, parameterized design that receives a number of parameters, and returns a number of objects (potentially one, or many) based on the provided information. 13, 39

Instruction Set Architecture An abstract model of a computer. It is also referred to as architecture or computer architecture. A realization of an ISA, such as a central processing unit (CPU), is called an implementation. 1

Intermediate Representation The data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive for further processing, such as optimization and translation. 38

Internet of Things A dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual "things" have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network, often communicate data associated with users and their environments [35]. 15

Java Virtual Machine A virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. 18

JTAG An industry standard connector type for verifying designs and testing printed circuit boards after manufacture. Used for communicating at a low level with the SoC design implemented in Chipyard. 23

Lazy evaluation Computation model where expressions are evaluated as late as possible during program execution. This allows for infinitely recursive structures that do not cause program non-termination. Lazy evaluation tends to be most frequently used in functional programming languages, like Scala. 10

`Man` Command to fetch and open `man`ual pages from the system's informational database. 2, 3

MCS A descriptive file containing the contents of what will be flashed to the FPGA utilizing ASCII format. MCS files include additional information beyond the payload, such as headers and comment information. 26

Memory-Mapped Input/Output Memory-mapped Input/Output uses the same address space to address both memory and I/O devices. The memory and registers of the I/O devices are mapped to (associated with) address values. 15

Multi-thread The act of using multiple processes simultaneously, allowing for parallel computation. 2

Parameterize The ability for an object to receive input parameters to change its behavior. This is the key functionality that allows a Generator to work. 8, 15, 38

Reduced Instruction Set Computer A computer with a small, highly optimized set of instructions, rather than the more specialized set often found in other types of architecture. 31, 39

Register Transfer Language A type of object code a kind of intermediate representation that is very close to assembly language, such as that which is used in a compiler. It is used to describe data flow at the register-transfer level of an architecture. 15

RISC-V The fifth revision of an open-source Reduced Instruction Set Computer architecture, developed at University of California Berkeley. 1, 2, 5, 6, 10, 12, 31, 32

Scala A strong statically typed general-purpose programming language which supports both object-oriented programming and functional programming. Designed to be concise, many of Scala's design decisions are aimed to address criticisms of Java. 39

Scala Build Tool sbt is an open-source Scala-based DSL to express parallel processing task graphs as a build tool for Scala and Java projects, similar to Apache's Maven and Ant. 3

Serial Peripheral Interface A synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems. 23

Single Instruction Multiple Data Operations defined using a single instruction that takes multiple data values in simultaneously. 12

Softcore A digital circuit design (typically a logic core) that can be wholly described and implemented using software and logic synthesis. Such a design is typically written out to an FPGA, but can be written out to other programmable logic devices. 1, 7, 31

Source Source code for a project. In the context of building software, building from source means compiling the project manually. 2–4

SPIKE A RISC-V ISA Simulator that is useful for validating the results of a SoC implementation to ensure the ISA is implemented correctly. 30

System on a Chip An integrated circuit that integrates all or most components of a computer or other electronic system. 10

Universal Asynchronous Receiver-Transmitter Computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable. 23