

Chipyard SoC Design Framework

Alexander Lukens

Karl Hallsby

Illinois Institute of Technology

Last Edited: March 25, 2021

Contents

1	Setup	1
1.1	Introduction	1
1.2	Project Environment	1
1.3	Building Chipyard	1
1.3.1	Chipyard Dependencies	1
1.3.2	Xilinx Vivado Suite Installation	3
1.4	Other Useful Projects	4
1.4.1	Freedom E SDK	4
1.4.2	Freedom Tools	4
2	Repository Deep Dive	5
2.1	Verilator Simulator	5
2.1.1	About	5
2.1.2	Generators	5
2.1.3	Custom Configurations	5
2.1.4	FPGA Implementation	5
	Bibliography	6

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.

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, running in a virtual machine with at least:

- 4 cores
- 8 GB of RAM
- 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, so long as the versions of the dependencies are matched. Chipyard also has explicit support for CentOS, which extends to Fedora and RHEL as well. In addition, installing and using Linux natively works as well.

1.3 Building Chipyard

Here, we present the necessary steps to retrieving all the dependencies required to set up Chipyard for local development and simulation use. All of the code shown in the listings of this section is gathered in the `code` subdirectory.

1.3.1 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

Chippyard relies on numerous dependencies and libraries to read files and build the required Verilog files. In addition, Chippyard relies on **sbt**, the Scala Build Tool, as a majority of Chippyard and its dependencies are written in Scala.

[Listing 1](#) 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.

```

1  #!/usr/bin/env bash
2
3  set -ex
4
5  sudo apt-get install -y build-essential bison flex
6  sudo apt-get install -y libgmp-dev libmpfr-dev libmpc-dev zlib1g-dev vim git default-jdk
   ↪ default-jre
7  # install sbt: https://www.scala-sbt.org/release/docs/Installing-sbt-on-Linux.html
8  echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a
   ↪ /etc/apt/sources.list.d/sbt.list
9  curl -sL
   ↪ "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823"
   ↪ | sudo apt-key add
10 sudo apt-get update
11 # Scala Built Tool
12 sudo apt-get install -y sbt
13 # For the 'info' command and parsing command line arguments
14 sudo apt-get install -y texinfo gengetopt
15 # Additional dependencies
16 sudo apt-get install -y libexpat1-dev libusb-dev libncurses5-dev cmake
17 # deps for poky
18 sudo apt-get install -y python3.6 patch diffstat texi2html texinfo subversion chrpath git
   ↪ wget
19 # deps for qemu
20 sudo apt-get install -y libgtk-3-dev gettext
21 # deps for firemarshal
22 sudo apt-get install -y python3-pip python3.6-dev rsync libguestfs-tools expat ctags
23 # install DTC
24 sudo apt-get install -y device-tree-compiler

```

Listing 1: Fetch Chippyard Dependencies using **apt** on Ubuntu

Build Verilator from Source

Chippyard'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 2](#). 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 Chippyard and its Direct Dependencies

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

```
1  #!/usr/bin/env bash
2
3  set -ex
4
5  # Dependencies for building verilator from source
6  sudo apt install autoconf automake
7
8  # install verilator
9  git clone http://git.veripool.org/git/verilator
10 cd verilator
11 git checkout v4.034
12 autoconf && ./configure && make -j$(nproc) && sudo make install
```

Listing 2: Building Verilator from Source

Listing 3 has been provided that handles this for you.

```
1  #!/usr/bin/env bash
2
3  git clone https://github.com/ucb-bar/chipyard.git
4  cd chipyard
5  ./scripts/init-submodules-no-riscv-tools.sh
```

Listing 3: Fetch Chipyard and Submodules

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 `./scripts/build-toolchains.sh riscv-tools` while inside your local copy of the cloned Chipyard repository.

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`. 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, there are two possible quality of life improvements you can make.

1. Add the line `source /path/to/chipyard/env.sh` to your `.bashrc` file in your home directory. After adding this to your `.bashrc` file, restart your shell, or re-source your `.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.

1.3.2 Xilinx Vivado Suite Installation

The [Xilinx Vivado Suite](#) is important to be installed if any work regarding an FPGA is to be conducted. While performing this work and preparing this document, 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 for a variety of operating systems, including Ubuntu¹ 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.

1.4 Other Useful Projects

1.4.1 Freedom E SDK

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

1.4.2 Freedom Tools

[This repository](#) is maintained by SiFive. 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 the `make` command as `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.

¹Xilinx 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

2.1 Verilator Simulator

2.1.1 About

The primary way to simulate SoCs’ designed using the Chipyard framework is via Verilator simulations. The directory for verilator is `chipyard/sims/verilator`. An example simulation can be run by using `make` in the 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.

2.1.2 Generators

Chipyard Generator

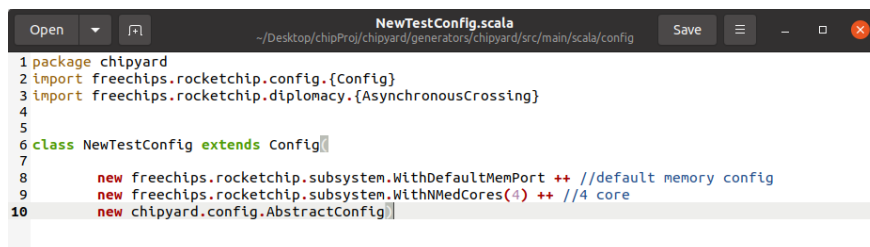
SHA3 Accelerators

2.1.3 Custom Configurations

Custom Configs can be created in the directory `chipyard/generators/chipyard/src/main/scala/config/`. For example, I created a new scala file called `NewTestConfig.scala` in the directory, allowing me to create a simulator from a class inside the `NewTestConfig.scala` file. Example Configs can be found in `RocketConfigs.scala` in the same directory.

2.1.4 FPGA Implementation

About



```
1 package chipyard
2 import freechips.rocketchip.config.{Config}
3 import freechips.rocketchip.diplomacy.{AsynchronousCrossing}
4
5
6 class NewTestConfig extends Config{
7
8     new freechips.rocketchip.subsystem.WithDefaultMemPort ++ //default memory config
9     new freechips.rocketchip.subsystem.WithNMedCores(4) ++ //4 core
10    new chipyard.config.AbstractConfig{
```

Figure 2.1: `NewTestConfig.scala`

Bibliography

- [Ami+20] 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](https://doi.org/10.1109/MM.2020.2996616).