Chipyard Documentation

Release 1.6.2

Berkeley Architecture Research

Mar 01, 2022

Contents

Chipyard is a framework for designing and evaluating full-system hardware using agile teams. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of systems-on-chip. This work is supported by the NSF CCRI ENS Chipyard Award #201662.

Important: New to Chipyard? Jump to the *Initial Repository Setup* page for setup instructions.

CHAPTER 1

Getting Help

If you have a question about Chipyard that isn't answered by the existing documentation, feel free to ask for help on the Chipyard Google Group.

CHAPTER 2

Table of Contents

2.1 Chipyard Basics

These sections will walk you through the basics of the Chipyard framework:

- First, we will go over the components of the framework.
- Next, we will go over how to understand how Chipyard configures its designs.
- Then, we will go over initial framework setup.

Hit next to get started!

2.1.1 Chipyard Components

Generators

The Chipyard Framework currently consists of the following RTL generators:

Processor Cores

Rocket Core An in-order RISC-V core. See *Rocket Core* for more information.

BOOM (Berkeley Out-of-Order Machine) An out-of-order RISC-V core. See *Berkeley Out-of-Order Machine (BOOM)* for more information.

CVA6 Core An in-order RISC-V core written in System Verilog. Previously called Ariane. See *CVA6 Core* for more information.

Ibex Core An in-order 32 bit RISC-V core written in System Verilog. See *Ibex Core* for more information.

Accelerators

Hwacha A decoupled vector architecture co-processor. Hwacha currently implements a non-standard RISC-V extension, using a vector architecture programming model. Hwacha integrates with a Rocket or BOOM core using the RoCC (Rocket Custom Co-processor) interface. See *Hwacha* for more information.

Gemmini A matrix-multiply accelerator targeting neural-networks

SHA3 A fixed-function accelerator for the SHA3 hash function. This simple accelerator is used as a demonstration for some of the Chipyard integration flows using the RoCC interface.

System Components:

icenet A Network Interface Controller (NIC) designed to achieve up to 200 Gbps.

sifive-blocks System components implemented by SiFive and used by SiFive projects, designed to be integrated with the Rocket Chip generator. These system and peripheral components include UART, SPI, JTAG, I2C, PWM, and other peripheral and interface devices.

AWL (Analog Widget Library) Digital components required for integration with high speed serial links.

testchipip A collection of utilities used for testing chips and interfacing them with larger test environments.

Tools

Chisel A hardware description library embedded in Scala. Chisel is used to write RTL generators using meta-programming, by embedding hardware generation primitives in the Scala programming language. The Chisel compiler elaborates the generator into a FIRRTL output. See *Chisel* for more information.

FIRRTL An intermediate representation library for RTL description of digital designs. FIRRTL is used as a formalized digital circuit representation between Chisel and Verilog. FIRRTL enables digital circuits manipulation between Chisel elaboration and Verilog generation. See *FIRRTL* for more information.

Barstools A collection of common FIRRTL transformations used to manipulate a digital circuit without changing the generator source RTL. See *Barstools* for more information.

Dsptools A Chisel library for writing custom signal processing hardware, as well as integrating custom signal processing hardware into an SoC (especially a Rocket-based SoC).

Dromajo A RV64GC emulator primarily used for co-simulation and was originally developed by Esperanto Technologies. See *Dromajo* for more information.

Toolchains

riscv-tools A collection of software toolchains used to develop and execute software on the RISC-V ISA. The include compiler and assembler toolchains, functional ISA simulator (spike), the Berkeley Boot Loader (BBL) and proxy kernel. The riscv-tools repository was previously required to run any RISC-V software, however, many of the riscv-tools components have since been upstreamed to their respective open-source projects (Linux, GNU, etc.). Nevertheless, for consistent versioning, as well as software design flexibility for custom hardware, we include the riscv-tools repository and installation in the Chipyard framework.

esp-tools A fork of riscv-tools, designed to work with the Hwacha non-standard RISC-V extension. This fork can also be used as an example demonstrating how to add additional RoCC accelerators to the ISA-level simulation (Spike) and the higher-level software toolchain (GNU binutils, riscv-opcodes, etc.)

Software

FireMarshal FireMarshal is the default workload generation tool that Chipyard uses to create software to run on its platforms. See *FireMarshal* for more information.

Sims

Verilator Verilator is an open source Verilog simulator. The `verilator` directory provides wrappers which construct Verilator-based simulators from relevant generated RTL, allowing for execution of test RISC-V programs on the simulator (including vcd waveform files). See *Verilator (Open-Source)* for more information.

VCS VCS is a proprietary Verilog simulator. Assuming the user has valid VCS licenses and installations, the `vcs` directory provides wrappers which construct VCS-based simulators from relevant generated RTL, allowing for execution of test RISC-V programs on the simulator (including vcd/vpd waveform files). See *Synopsys VCS (License Required)* for more information.

FireSim FireSim is an open-source FPGA-accelerated simulation platform, using Amazon Web Services (AWS) EC2 F1 instances on the public cloud. FireSim automatically transforms and instruments open-hardware designs into fast (10s-100s MHz), deterministic, FPGA-based simulators that enable productive pre-silicon verification and performance validation. To model I/O, FireSim includes synthesizeable and timing-accurate models for standard interfaces like DRAM, Ethernet, UART, and others. The use of the elastic public cloud enable FireSim to scale simulations up to thousands of nodes. In order to use FireSim, the repository must be cloned and executed on AWS instances. See *FireSim* for more information.

Prototyping

FPGA Prototyping FPGA prototyping is supported in Chipyard using SiFive's `fpga-shells`. Some examples of FPGAs supported are the Xilinx Arty 35T and VCU118 boards. For a fast and deterministic simulation with plenty of debugging tools, please consider using the *FireSim* platform. See *Prototyping Flow* for more information on FPGA prototypes.

VLSI

Hammer Hammer is a VLSI flow designed to provide a layer of abstraction between general physical design concepts to vendor-specific EDA tool commands. The HAMMER flow provide automated scripts which generate relevant tool commands based on a higher level description of physical design constraints. The Hammer flow also allows for re-use of process technology knowledge by enabling the construction of process-technology-specific plug-ins, which describe particular constraints relating to that process technology (obsolete standard cells, metal layer routing constraints, etc.). The Hammer flow requires access to proprietary EDA tools and process technology libraries. See *Core Hammer* for more information.

2.1.2 Development Ecosystem

Chipyard Approach

The trend towards agile hardware design and evaluation provides an ecosystem of debugging and implementation tools, that make it easier for computer architecture researchers to develop novel concepts. Chipyard hopes to build on this prior work in order to create a singular location to which multiple projects within the Berkeley Architecture Research can coexist and be used together. Chipyard aims to be the "one-stop shop" for creating and testing your own unique System on a Chip (SoC).

Chisel/FIRRTL

One of the tools to help create new RTL designs quickly is the Chisel Hardware Construction Language and the FIRRTL Compiler. Chisel is an embedded language within Scala that provides a set of libraries to help hardware designers create highly parameterizable RTL. FIRRTL on the other hand is a compiler for hardware which allows the user to run FIRRTL passes that can do dead code elimination, circuit analysis, connectivity checks, and much more! These two tools in combination allow quick design space exploration and development of new RTL.

RTL Generators

Within this repository, all of the Chisel RTL is written as generators. Generators are parametrized programs designed to generate RTL code based on configuration specifications. Generators can be used to generate Systems-on-Chip (SoCs) using a collection of system components organized in unique generator projects. Generators allow you to create a family of SoC designs instead of a single instance of a design!

2.1.3 Configs, Parameters, Mixins, and Everything In Between

A significant portion of generators in the Chipyard framework use the Rocket Chip parameter system. This parameter system enables for the flexible configuration of the SoC without invasive RTL changes. In order to use the parameter system correctly, we will use several terms and conventions:

Parameters

It is important to note that a significant challenge with the Rocket parameter system is being able to identify the correct parameter to use, and the impact that parameter has on the overall system. We are still investigating methods to facilitate parameter exploration and discovery.

Configs

A *config* is a collection of multiple generator parameters being set to specific values. Configs are additive, can override each other, and can be composed of other configs (sometimes referred to as config fragments). The naming convention for an additive config or config fragment is `With<YourConfigName>`, while the naming convention for a non-additive config will be `<YourConfig>`. Configs can take arguments which will in-turn set parameters in the design or reference other parameters in the design (see *Parameters*).

This example shows a basic config fragment class that takes in zero arguments and instead uses hardcoded values to set the RTL design parameters. In this example, `MyAcceleratorConfig` is a Scala case class that defines a set of variables that the generator can use when referencing the `MyAcceleratorKey` in the design.

```
class WithMyAcceleratorParams extends Config((site, here, up) => {
  case BusWidthBits => 128
  case MyAcceleratorKey =>
    MyAcceleratorConfig(
      rows = 2,
      rowBits = 64,
      columns = 16,
      hartId = 1,
      someLength = 256)
})
```

This next example shows a "higher-level" additive config fragment that uses prior parameters that were set to derive other parameters.

```
class WithMyMoreComplexAcceleratorConfig extends Config((site, here, up) => {
  case BusWidthBits => 128
  case MyAcceleratorKey =>
    MyAcceleratorConfig(
      Rows = 2,
      rowBits = site(SystemBusKey).beatBits,
      hartId = up(RocketTilesKey, site).length)
})
```

The following example shows a non-additive config that combines or "assembles" the prior two config fragments using ++. The additive config fragments are applied from the right to left in the list (or bottom to top in the example). Thus, the order of the parameters being set will first start with the `DefaultExampleConfig`, then `WithMyAcceleratorParams`, then `WithMyMoreComplexAcceleratorConfig`.

```
class SomeAdditiveConfig extends Config(
  new WithMyMoreComplexAcceleratorConfig ++
  new WithMyAcceleratorParams ++
  new DefaultExampleConfig
)
```

The `site`, `here`, and `up` objects in `WithMyMoreComplexAcceleratorConfig` are maps from configuration keys to their definitions. The `site` map gives you the definitions as seen from the root of the configuration hierarchy (in this example, `SomeAdditiveConfig`). The `here` map gives the definitions as seen at the current level of the hierarchy (i.e. in `WithMyMoreComplexAcceleratorConfig` itself). The `up` map gives the definitions as seen from the next level up from the current (i.e. from `WithMyAcceleratorParams`).

Cake Pattern / Mixin

A cake pattern or mixin is a Scala programming pattern, which enable "mixing" of multiple traits or interface definitions (sometimes referred to as dependency injection). It is used in the Rocket Chip SoC library and Chipyard framework in merging multiple system components and IO interfaces into a large system component.

This example shows the Chipyard default top that composes multiple traits together into a fully-featured SoC with many optional components.

```
class DigitalTop(implicit p: Parameters) extends ChipyardSystem
  with testchipip.CanHavePeripheryCustomBootPin // Enables optional custom boot pin
  with testchipip.HasPeripheryBootAddrReg // Use programmable boot address register
  with testchipip.CanHaveTraceIO // Enables optionally adding trace IO
  with testchipip.CanHaveBackingScratchpad // Enables optionally adding a backing␣
↪scratchpad
  with testchipip.CanHavePeripheryBlockDevice // Enables optionally adding the block␣
↪device
  with testchipip.CanHavePeripheryTLSerial // Enables optionally adding the backing␣
↪memory and serial adapter
  with sifive.blocks.devices.i2c.HasPeripheryI2C // Enables optionally adding the␣
↪sifive I2C
  with sifive.blocks.devices.pwm.HasPeripheryPWM // Enables optionally adding the␣
↪sifive PWM
  with sifive.blocks.devices.uart.HasPeripheryUART // Enables optionally adding the␣
↪sifive UART
  with sifive.blocks.devices.gpio.HasPeripheryGPIO // Enables optionally adding the␣
↪sifive GPIOs
  with sifive.blocks.devices.spi.HasPeripherySPIFlash // Enables optionally adding␣
↪the sifive SPI flash controller
  with sifive.blocks.devices.spi.HasPeripherySPI // Enables optionally adding the␣
↪sifive SPI port
```

(continues on next page)

(continued from previous page)

```
  with icenet.CanHavePeripheryIceNIC // Enables optionally adding the IceNIC for␣
↪FireSim
  with chipyard.example.CanHavePeripheryInitZero // Enables optionally adding the␣
↪initzero example widget
  with chipyard.example.CanHavePeripheryGCD // Enables optionally adding the GCD␣
↪example widget
  with chipyard.example.CanHavePeripheryStreamingFIR // Enables optionally adding the␣
↪DSPTools FIR example widget
  with chipyard.example.CanHavePeripheryStreamingPassthrough // Enables optionally␣
↪adding the DSPTools streaming-passthrough example widget
  with nvidia.blocks.dla.CanHavePeripheryNVDLA // Enables optionally having an NVDLA
  with chipyard.clocking.HasChipyardPRCI // Use Chipyard reset/clock distribution
  with fftgenerator.CanHavePeripheryFFT // Enables optionally having an MMIO-based␣
↪FFT block
{
  override lazy val module = new DigitalTopModule(this)
}

class DigitalTopModule[+L <: DigitalTop](l: L) extends ChipyardSystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with sifive.blocks.devices.i2c.HasPeripheryI2CModuleImp
  with sifive.blocks.devices.pwm.HasPeripheryPWMModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIFlashModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIModuleImp
  with chipyard.example.CanHavePeripheryGCDModuleImp
  with freechips.rocketchip.util.DontTouch
```

There are two "cakes" or mixins here. One for the lazy module (ex. `CanHavePeripherySerial`) and one for the lazy module implementation (ex. `CanHavePeripherySerialModuleImp` where `Imp` refers to implementation). The lazy module defines all the logical connections between generators and exchanges configuration information among them, while the lazy module implementation performs the actual Chisel RTL elaboration.

In the `DigitalTop` example class, the "outer" `DigitalTop` instantiates the "inner" `DigitalTopModule` as a lazy module implementation. This delays immediate elaboration of the module until all logical connections are determined and all configuration information is exchanged. The `System` outer base class, as well as the `CanHavePeriphery<X>` outer traits contain code to perform high-level logical connections. For example, the `CanHavePeripherySerial` outer trait contains code to optionally lazily instantiate the `SerialAdapter`, and connect the `SerialAdapter`'s TileLink node to the Front bus.

The `ModuleImp` classes and traits perform elaboration of real RTL. For example, the `CanHavePeripherySerialModuleImp` trait optionally physically connects the `SerialAdapter` module, and instantiates queues.

In the test harness, the SoC is elaborated with `val dut = p(BuildTop)(p)`.

After elaboration, the system submodule of `ChipTop` will be a `DigitalTop` module, which contains a `SerialAdapter` module (among others), if the config specified for that block to be instantiated.

From a high level, classes which extend `LazyModule` *must* reference their module implementation through `lazy val module`, and they *may* optionally reference other lazy modules (which will elaborate as child modules in the module hierarchy). The "inner" modules contain the implementation for the module, and may instantiate other normal modules OR lazy modules (for nested Diplomacy graphs, for example).

The naming convention for an additive mixin or trait is `CanHave<YourMixin>`. This is shown in the `Top` class where things such as `CanHavePeripherySerial` connect a RTL component to a bus and expose signals to the top-level.

Additional References

Another description of traits/mixins and config fragments is given in *Keys, Traits, and Configs*. Additionally, a brief explanation of some of these topics (with slightly different naming) is given in the following video: https://www.youtube.com/watch?v=Eko86PGEoDY.

Note: Chipyard uses the name "config fragments" over "config mixins" to avoid confusion between a mixin applying to a config or to the system `Top` (even though both are technically Scala mixins).

2.1.4 Initial Repository Setup

Requirements

Chipyard is developed and tested on Linux-based systems.

Warning: It is possible to use this on macOS or other BSD-based systems, although GNU tools will need to be installed; it is also recommended to install the RISC-V toolchain from `brew`.

Warning: Working under Windows is not recommended.

In CentOS-based platforms, we recommend installing the following dependencies:

```
#!/bin/bash

set -ex

sudo yum groupinstall -y "Development tools"
sudo yum install -y gmp-devel mpfr-devel libmpc-devel zlib-devel vim git java java-
↪devel

# Install SBT https://www.scala-sbt.org/release/docs/Installing-sbt-on-Linux.html
↪#Red+Hat+Enterprise+Linux+and+other+RPM-based+distributions
# sudo rm -f /etc/yum.repos.d/bintray-rpm.repo
# Use rm above if sbt installed from bintray before.
curl -L https://www.scala-sbt.org/sbt-rpm.repo > sbt-rpm.repo
sudo mv sbt-rpm.repo /etc/yum.repos.d/

sudo yum install -y sbt texinfo gengetopt
sudo yum install -y expat-devel libusb1-devel ncurses-devel cmake
↪"perl(ExtUtils::MakeMaker)"
# deps for poky
sudo yum install -y python38 patch diffstat texi2html texinfo subversion chrpath git␣
↪wget
# deps for qemu
sudo yum install -y gtk3-devel
# deps for firemarshal
sudo yum install -y python38-pip python38-devel rsync libguestfs-tools makeinfo expat␣
↪ctags
# Install GNU make 4.x (needed to cross-compile glibc 2.28+)
sudo yum install -y centos-release-scl
```

(continues on next page)

(continued from previous page)

```
sudo yum install -y devtoolset-8-make
# install DTC
sudo yum install -y dtc
sudo yum install -y python

# install verilator
git clone http://git.veripool.org/git/verilator
cd verilator
git checkout v4.034
autoconf && ./configure && make -j$(nproc) && sudo make install
```

In Ubuntu/Debian-based platforms (Ubuntu), we recommend installing the following dependencies. These dependencies were written based on Ubuntu 16.04 LTS and 18.04 LTS - If they don't work for you, you can try out the Docker image (*Pre-built Docker Image*) before manually installing or removing dependencies:

```
#!/bin/bash

set -ex

sudo apt-get install -y build-essential bison flex software-properties-common curl
sudo apt-get install -y libgmp-dev libmpfr-dev libmpc-dev zlib1g-dev vim default-jdk␣
↪default-jre
# install sbt: https://www.scala-sbt.org/release/docs/Installing-sbt-on-Linux.html
↪#Ubuntu+and+other+Debian-based+distributions
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␣
↪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
sudo apt-get install -y python
# install git >= 2.17
sudo add-apt-repository ppa:git-core/ppa -y
sudo apt-get update
sudo apt-get install git -y

# install verilator
sudo apt-get install -y autoconf
git clone http://git.veripool.org/git/verilator
cd verilator
git checkout v4.034
autoconf && ./configure && make -j$(nproc) && sudo make install
```

Note: When running on an Amazon Web Services EC2 FPGA-development instance (for FireSim), FireSim includes

a machine setup script that will install all of the aforementioned dependencies (and some additional ones).

Setting up the Chipyard Repo

Start by fetching Chipyard's sources. Run:

```
git clone https://github.com/ucb-bar/chipyard.git
cd chipyard
# checkout latest official chipyard release
# note: this may not be the latest release if the documentation version !=␣
↪"stable"
git checkout 1.6.2
./scripts/init-submodules-no-riscv-tools.sh
```

This will initialize and checkout all of the necessary git submodules. This will also validate that you are on a tagged branch, otherwise it will prompt for confirmation.

When updating Chipyard to a new version, you will also want to rerun this script to update the submodules. Using git directly will try to initialize all submodules; this is not recommended unless you expressly desire this behavior.

Building a Toolchain

The *toolchains* directory contains toolchains that include a cross-compiler toolchain, frontend server, and proxy kernel, which you will need in order to compile code to RISC-V instructions and run them on your design. Currently there are two toolchains, one for normal RISC-V programs, and another for Hwacha (`esp-tools`). For custom installations, Each tool within the toolchains contains individual installation procedures within its README file. To get a basic installation (which is the only thing needed for most Chipyard use-cases), just the following steps are necessary. This will take about 20-30 minutes. You can expedite the process by setting a `make` environment variable to use parallel cores: `export MAKEFLAGS=-j8`.

```
./scripts/build-toolchains.sh riscv-tools # for a normal risc-v toolchain
```

Note: If you are planning to use the Hwacha vector unit, or other RoCC-based accelerators, you should build the esp-tools toolchain by adding the `esp-tools` argument to the script above. If you are running on an Amazon Web Services EC2 instance, intending to use FireSim, you can also use the `--ec2fast` flag for an expedited installation of a pre-compiled toolchain.

Once the script is run, a `env.sh` file is emitted that sets the `PATH`, `RISCV`, and `LD_LIBRARY_PATH` environment variables. You can put this in your `.bashrc` or equivalent environment setup file to get the proper variables, or directly include it in your current environment:

```
source ./env.sh
```

These variables need to be set for the `make` system to work properly.

Pre-built Docker Image

An alternative to setting up the Chipyard repository locally is to pull the pre-built Docker image from Docker Hub. The image comes with all dependencies installed, Chipyard cloned, and toolchains initialized. This image sets up baseline Chipyard (not including FireMarshal, FireSim, and Hammer initializations). Each image comes with a tag that corresponds to the version of Chipyard cloned/set-up in that image. Not including a tag during the pull will pull the image with the latest version of Chipyard. First, pull the Docker image. Run:

```
sudo docker pull ucbbar/chipyard-image:<TAG>
```

To run the Docker container in an interactive shell, run:

```
sudo docker run -it ucbbar/chipyard-image bash
```

What's Next?

This depends on what you are planning to do with Chipyard.

- If you intend to run a simulation of one of the vanilla Chipyard examples, go to *Software RTL Simulation* and follow the instructions.
- If you intend to run a simulation of a custom Chipyard SoC Configuration, go to *Simulating A Custom Project* and follow the instructions.
- If you intend to run a full-system FireSim simulation, go to *FPGA-Accelerated Simulation* and follow the instructions.
- If you intend to add a new accelerator, go to Basic customization and follow the instructions.
- If you want to learn about the structure of Chipyard, go to *Chipyard Components*.
- If you intend to change the generators (BOOM, Rocket, etc) themselves, see *Included RTL Generators*.
- If you intend to run a tutorial VLSI flow using one of the Chipyard examples, go to *ASAP7 Tutorial* and follow the instructions.
- If you intend to build a chip using one of the vanilla Chipyard examples, go to *Building A Chip* and follow the instructions.

Upgrading Chipyard Release Versions

In order to upgrade between Chipyard versions, we recommend using a fresh clone of the repository (or your fork, with the new release merged into it).

Chipyard is a complex framework that depends on a mix of build systems and scripts. Specifically, it relies on git submodules, on sbt build files, and on custom written bash scripts and generated files. For this reason, upgrading between Chipyard versions is **not** as trivial as just running `git submodule update --recursive`. This will result in recursive cloning of large submodules that are not necessarily used within your specific Chipyard environments. Furthermore, it will not resolve the status of stale state generated files which may not be compatible between release versions.

If you are an advanced git user, an alternative approach to a fresh repository clone may be to run `git clean -dfx`, and then run the standard Chipyard setup sequence. This approach is dangerous, and **not-recommended** for users who are not deeply familiar with git, since it "blows up" the repository state and removes all untracked and modified files without warning. Hence, if you were working on custom un-committed changes, you would lose them.

If you would still like to try to perform an in-place manual version upgrade (**not-recommended**), we recommend at least trying to resolve stale state in the following areas:

- Delete stale `target` directories generated by sbt.
- Delete jar collateral generated by FIRRTL (`lib/firrtl.jar`)
- Re-generate generated scripts and source files (for example, `env.sh`)
- Re-generating/deleting target software state (Linux kernel binaries, Linux images) within FireMarshal

This is by no means a comprehensive list of potential stale state within Chipyard. Hence, as mentioned earlier, the recommended method for a Chipyard version upgrade is a fresh clone (or a merge, and then a fresh clone).

2.2 Simulation

Chipyard supports two classes of simulation:

1. Software RTL simulation using commercial or open-source (Verilator) RTL simulators
2. FPGA-accelerated full-system simulation using FireSim

Software RTL simulators of Chipyard designs run at O(1 KHz), but compile quickly and provide full waveforms. Conversely, FPGA-accelerated simulators run at O(100 MHz), making them appropriate for booting an operating system and running a complete workload, but have multi-hour compile times and poorer debug visibility.

Click next to see how to run a simulation.

2.2.1 Software RTL Simulation

Verilator (Open-Source)

Verilator is an open-source LGPL-Licensed simulator maintained by Veripool. The Chipyard framework can download, build, and execute simulations using Verilator.

Synopsys VCS (License Required)

VCS is a commercial RTL simulator developed by Synopsys. It requires commercial licenses. The Chipyard framework can compile and execute simulations using VCS. VCS simulation will generally compile faster than Verilator simulations.

To run a VCS simulation, make sure that the VCS simulator is on your `PATH`.

Choice of Simulator

First, we will start by entering the Verilator or VCS directory:

For an open-source Verilator simulation, enter the `sims/verilator` directory

```
# Enter Verilator directory
cd sims/verilator
```

For a proprietry VCS simulation, enter the `sims/vcs` directory

```
# Enter VCS directory
cd sims/vcs
```

Simulating The Default Example

To compile the example design, run `make` in the selected verilator or VCS directory. This will elaborate the `RocketConfig` in the example project.

Note: The elaboration of `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.

An executable called `simulator-chipyard-RocketConfig` will be produced. This executable is a simulator that has been compiled based on the design that was built. You can then use this executable to run any compatible RV64 code. For instance, to run one of the riscv-tools assembly tests.

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

Note: In a VCS simulator, the simulator name will be `simv-chipyard-RocketConfig` instead of `simulator-chipyard-RocketConfig`.

The makefiles have a `run-binary` rule that simplifies running the simulation executable. It adds many of the common command line options for you and redirects the output to a file.

```
make run-binary BINARY=$RISCV/riscv64-unknown-elf/share/riscv-tests/isa/rv64ui-p-
↪simple
```

Alternatively, we can run a pre-packaged suite of RISC-V assembly or benchmark tests, by adding the make target `run-asm-tests` or `run-bmark-tests`. For example:

```
make run-asm-tests
make run-bmark-tests
```

Note: Before running the pre-packaged suites, you must run the plain `make` command, since the elaboration command generates a `Makefile` fragment that contains the target for the pre-packaged test suites. Otherwise, you will likely encounter a `Makefile` target error.

Makefile Variables and Commands

You can get a list of useful Makefile variables and commands available from the Verilator or VCS directories. simply run `make help`:

```
# Enter Verilator directory
cd sims/verilator
make help

# Enter VCS directory
cd sims/vcs
make help
```

Simulating A Custom Project

If you later create your own project, you can use environment variables to build an alternate configuration.

In order to construct the simulator with our custom design, we run the following command within the simulator directory:

```
make SBT_PROJECT=... MODEL=... VLOG_MODEL=... MODEL_PACKAGE=... CONFIG=... CONFIG_
↪PACKAGE=... GENERATOR_PACKAGE=... TB=... TOP=...
```

Each of these make variables correspond to a particular part of the design/codebase and are needed so that the make system can correctly build and make a RTL simulation.

The `SBT_PROJECT` is the `build.sbt` project that holds all of the source files and that will be run during the RTL build.

The `MODEL` and `VLOG_MODEL` are the top-level class names of the design. Normally, these are the same, but in some cases these can differ (if the Chisel class differs than what is emitted in the Verilog).

The `MODEL_PACKAGE` is the Scala package (in the Scala code that says `package ...`) that holds the `MODEL` class.

The `CONFIG` is the name of the class used for the parameter config while the `CONFIG_PACKAGE` is the Scala package it resides in.

The `GENERATOR_PACKAGE` is the Scala package that holds the Generator class that elaborates the design.

The `TB` is the name of the Verilog wrapper that connects the `TestHarness` to VCS/Verilator for simulation.

Finally, the `TOP` variable is used to distinguish between the top-level of the design and the `TestHarness` in our system. For example, in the normal case, the `MODEL` variable specifies the `TestHarness` as the top-level of the design. However, the true top-level design, the SoC being simulated, is pointed to by the `TOP` variable. This separation allows the infrastructure to separate files based on the harness or the SoC top level.

Common configurations of all these variables are packaged using a `SUB_PROJECT` make variable. Therefore, in order to simulate a simple Rocket-based example system we can use:

```
make SUB_PROJECT=yourproject
./simulator-<yourproject>-<yourconfig> ...
```

All `make` targets that can be applied to the default example, can also be applied to custom project using the custom environment variables. For example, the following code example will run the RISC-V assembly benchmark suite on the Hwacha subproject:

```
make SUB_PROJECT=hwacha run-asm-tests
```

Finally, in the `generated-src/<...>-<package>-<config>/` directory resides all of the collateral and Verilog source files for the build/simulation. Specifically, the SoC top-level (`TOP`) Verilog file is denoted with `*.top.v` while the `TestHarness` file is denoted with `*.harness.v`.

Fast Memory Loading

The simulator loads the program binary over a simulated serial line. This can be quite slow if there is a lot of static data, so the simulator also allows data to be loaded from a file directly into the DRAM model.

```
make run-binary BINARY=test.riscv LOADMEM=testdata.hex LOADMEM_ADDR=81000000
```

The `.hex` file should be a text file with a hexadecimal number on each line.

```
deadbeef
0123
```

Each line uses little-endian order, so this file would produce the bytes "ef be ad de 01 23". `LOADMEM_ADDR` specifies which address in memory (in hexadecimal) to write the first byte to. The default is 0x81000000.

A special target that facilitates automatically generating a hex file for an entire elf RISC-V exectuable and then running the simulator with the appropriate flags is also available.

```
make run-binary-hex BINARY=test.riscv
```

Generating Waveforms

If you would like to extract waveforms from the simulation, run the command `make debug` instead of just `make`.

For a Verilator simulation, this will generate a vcd file (vcd is a standard waveform representation file format) that can be loaded to any common waveform viewer. An open-source vcd-capable waveform viewer is GTKWave.

For a VCS simulation, this will generate a vpd file (this is a proprietary waveform representation format used by Synopsys) that can be loaded to vpd-supported waveform viewers. If you have Synopsys licenses, we recommend using the DVE waveform viewer.

Additional Verilator Options

When building the verilator simulator there are some additional options:

```
make VERILATOR_THREADS=8 NUMACTL=1
```

The `VERILATOR_THREADS=<num>` option enables the compiled Verilator simulator to use `<num>` parallel threads. On a multi-socket machine, you will want to make sure all threads are on the same socket by using `NUMACTL=1` to enable `numactl`. By enabling this, you will use Chipyard's `numa_prefix` wrapper, which is a simple wrapper around `numactl` that runs your verilated simulator like this: `$(numa_prefix) ./simulator-<name> <simulator-args>`. Note that both these flags are mutually exclusive, you can use either independently (though it makes sense to use `NUMACTL` just with `VERILATOR_THREADS=8` during a Verilator simulation).

2.2.2 FPGA-Accelerated Simulation

FireSim

FireSim is an open-source cycle-accurate FPGA-accelerated full-system hardware simulation platform that runs on cloud FPGAs (Amazon EC2 F1). FireSim allows RTL-level simulation at orders-of-magnitude faster speeds than software RTL simulators. FireSim also provides additional device models to allow full-system simulation, including memory models and network models.

FireSim currently supports running only on Amazon EC2 F1 FPGA-enabled virtual instances. In order to simulate your Chipyard design using FireSim, if you have not already, follow the initial EC2 setup instructions as detailed in the FireSim documentation. Then clone Chipyard onto your FireSim manager instance, and setup your Chipyard repository as you would normally.

Next, initalize FireSim as a library in Chipyard by running:

```
# At the root of your chipyard repo
./scripts/firesim-setup.sh --fast
```

`firesim-setup.sh` initializes additional submodules and then invokes firesim's `build-setup.sh` script adding `--library` to properly initialize FireSim as a library submodule in chipyard. You may run `./sims/firesim/build-setup.sh --help` to see more options.

Finally, source the following environment at the root of the firesim directory:

```
cd sims/firesim
# (Recommended) The default manager environment (includes env.sh)
source sourceme-f1-manager.sh
```

Note: Every time you want to use FireSim with a fresh shell, you must source this `sourceme-f1-manager.sh`

At this point you're ready to use FireSim with Chipyard. If you're not already familiar with FireSim, please return to the FireSim Docs, and proceed with the rest of the tutorial.

Running your Design in FireSim

Converting a Chipyard config (one in `chipyard/src/main/scala` to run in FireSim is simple, and can be done either through the traditional configuration system or through FireSim's build-recipes scheme.

A FireSim simulation requires 3 additional config fragments:

- `WithFireSimConfigTweaks` modifies your design to better fit the FireSim usage model. This is composed of multiple smaller config fragments. For example, the removal of clock-gating (using the `WithoutClockGating` config fragment) which is required for correct functioning of the compiler. This config fragment also includes other config fragments such as the inclusion of UART in the design, which although may technically be optional,is *strongly* recommended.
- `WithDefaultMemModel` provides a default configuration for FASED memory models in the FireSim simulation. See the FireSim documentation for details. This config fragment is currently included by default within `WithFireSimConfigTweaks`, so it isn't neccessary to add in separately, but it is required if you choose not to use `WithFireSimConfigTweaks`.
- `WithDefaultFireSimBridges` sets the `IOBinders` key to use FireSim's Bridge system, which can drive target IOs with software bridge models running on the simulation host. See the FireSim documentation for details.

The simplest method to add this config fragments to your custom Chipyard config is through FireSim's build recipe scheme. After your FireSim environment is setup, you will define your custom build recipe in `sims/firesim/deploy/deploy/config_build_recipes.ini`. By prepending the FireSim config fragments (separated by _) to your Chipyard configuration, these config fragments will be added to your custom configuration as if they were listed in a custom Chisel config class definition. For example, if you would like to convert the Chipyard `LargeBoomConfig` to a FireSim simulation with a DDR3 memory model, the appropriate FireSim `TARGET_CONFIG` would be `DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_WithFireSimConfigTweaks_chipyard.LargeBoomConfig`. Note that the FireSim config fragments are part of the `firesim.firesim` scala package and therefore there do not need to be prefixed with the full package name as opposed to the Chipyard config fragments which need to be prefixed with the chipyard package name.

An alternative method to prepending the FireSim config fragments in the FireSim build recipe is to create a new "permanent" FireChip custom configuration, which includes the FireSim config fragments. We are using the same target (top) RTL, and only need to specify a new set of connection behaviors for the IOs of that module. Simply create a matching config within `generators/firechip/src/main/scala/TargetConfigs` that inherits your config defined in `chipyard`.

```
class FireSimRocketConfig extends Config(
  new WithDefaultFireSimBridges ++
  new WithDefaultMemModel ++
  new WithFireSimConfigTweaks ++
  new chipyard.RocketConfig)
```

While this option seems to require the maintenance of additional configuration code, it has the benefit of allowing for the inclusion of more complex config fragments which also accept custom arguments (for example, `WithDefaultMemModel` can take an optional argument‘‘)

2.3 Included RTL Generators

A Generator can be thought of as a generalized RTL design, written using a mix of meta-programming and standard RTL. This type of meta-programming is enabled by the Chisel hardware description language (see *Chisel*). A standard RTL design is essentially just a single instance of a design coming from a generator. However, by using meta-programming and parameter systems, generators can allow for integration of complex hardware designs in automated ways. The following pages introduce the generators integrated with the Chipyard framework.

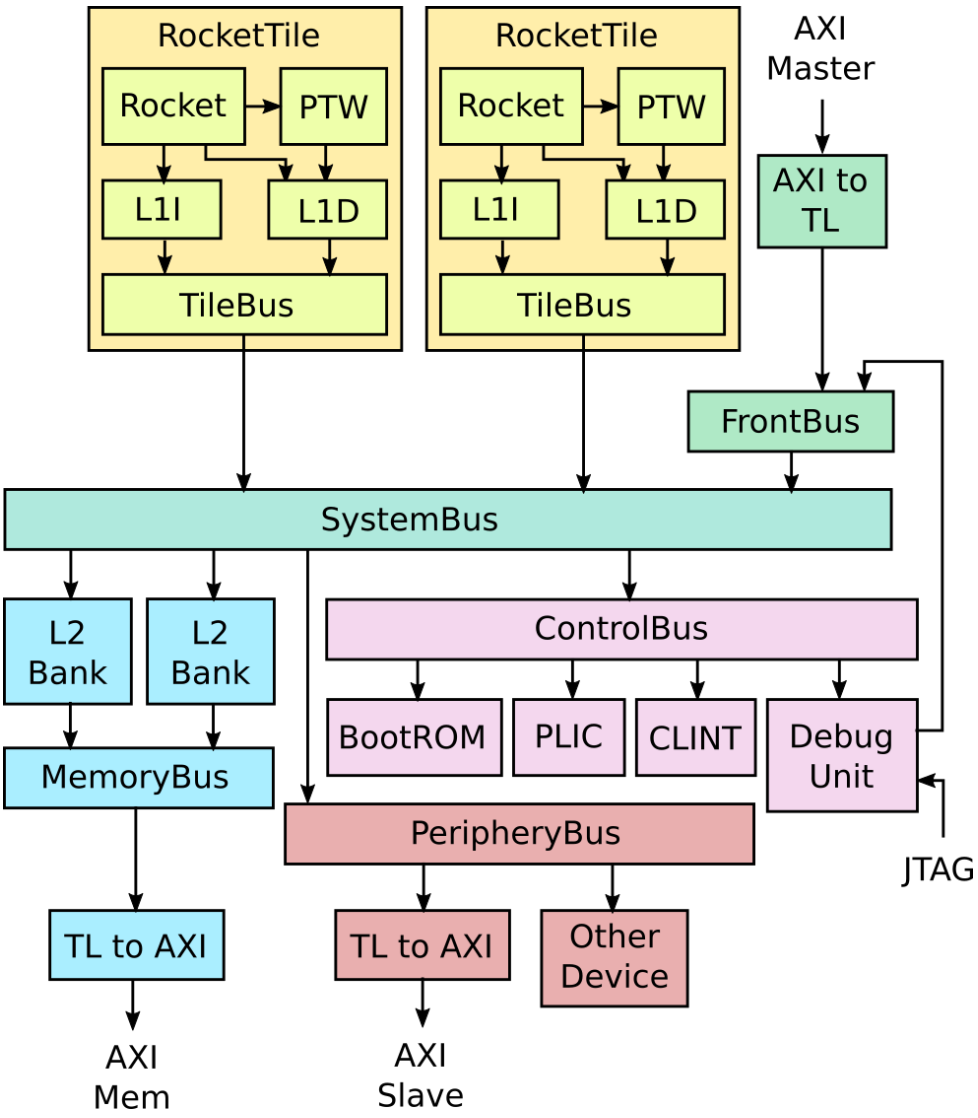
Chipyard bundles the source code for the generators, under the `generators/` directory. It builds them from source each time (although the build system will cache results if they have not changed), so changes to the generators themselves will automatically be used when building with Chipyard and propagate to software simulation, FPGA-accelerated simulation, and VLSI flows.

2.3.1 Rocket Chip

Rocket Chip generator is an SoC generator developed at Berkeley and now supported by SiFive. Chipyard uses the Rocket Chip generator as the basis for producing a RISC-V SoC.

Rocket Chip is distinct from *Rocket core*, the in-order RISC-V CPU generator. Rocket Chip includes many parts of the SoC besides the CPU. Though Rocket Chip uses Rocket core CPUs by default, it can also be configured to use the BOOM out-of-order core generator or some other custom CPU generator instead.

A detailed diagram of a typical Rocket Chip system is shown below.



Tiles

The diagram shows a dual-core `Rocket` system. Each `Rocket` core is grouped with a page-table walker, L1 instruction cache, and L1 data cache into a `RocketTile`.

The `Rocket` core can also be swapped for a `BOOM` core. Each tile can also be configured with a RoCC accelerator that connects to the core as a coprocessor.

Memory System

The tiles connect to the `SystemBus`, which connect it to the L2 cache banks. The L2 cache banks then connect to the `MemoryBus`, which connects to the DRAM controller through a TileLink to AXI converter.

To learn more about the memory hierarchy, see *Memory Hierarchy*.

MMIO

For MMIO peripherals, the `SystemBus` connects to the `ControlBus` and `PeripheryBus`.

The `ControlBus` attaches standard peripherals like the BootROM, the Platform-Level Interrupt Controller (PLIC), the core-local interrupts (CLINT), and the Debug Unit.

The BootROM contains the first stage bootloader, the first instructions to run when the system comes out of reset. It also contains the Device Tree, which is used by Linux to determine what other peripherals are attached.

The PLIC aggregates and masks device interrupts and external interrupts.

The core-local interrupts include software interrupts and timer interrupts for each CPU.

The Debug Unit is used to control the chip externally. It can be used to load data and instructions to memory or pull data from memory. It can be controlled through a custom DMI or standard JTAG protocol.

The `PeripheryBus` attaches additional peripherals like the NIC and Block Device. It can also optionally expose an external AXI4 port, which can be attached to vendor-supplied AXI4 IP.

To learn more about adding MMIO peripherals, check out the *MMIO Peripherals* section.

DMA

You can also add DMA devices that read and write directly from the memory system. These are attached to the `FrontendBus`. The `FrontendBus` can also connect vendor-supplied AXI4 DMA devices through an AXI4 to TileLink converter.

To learn more about adding DMA devices, see the *Adding a DMA Device* section.

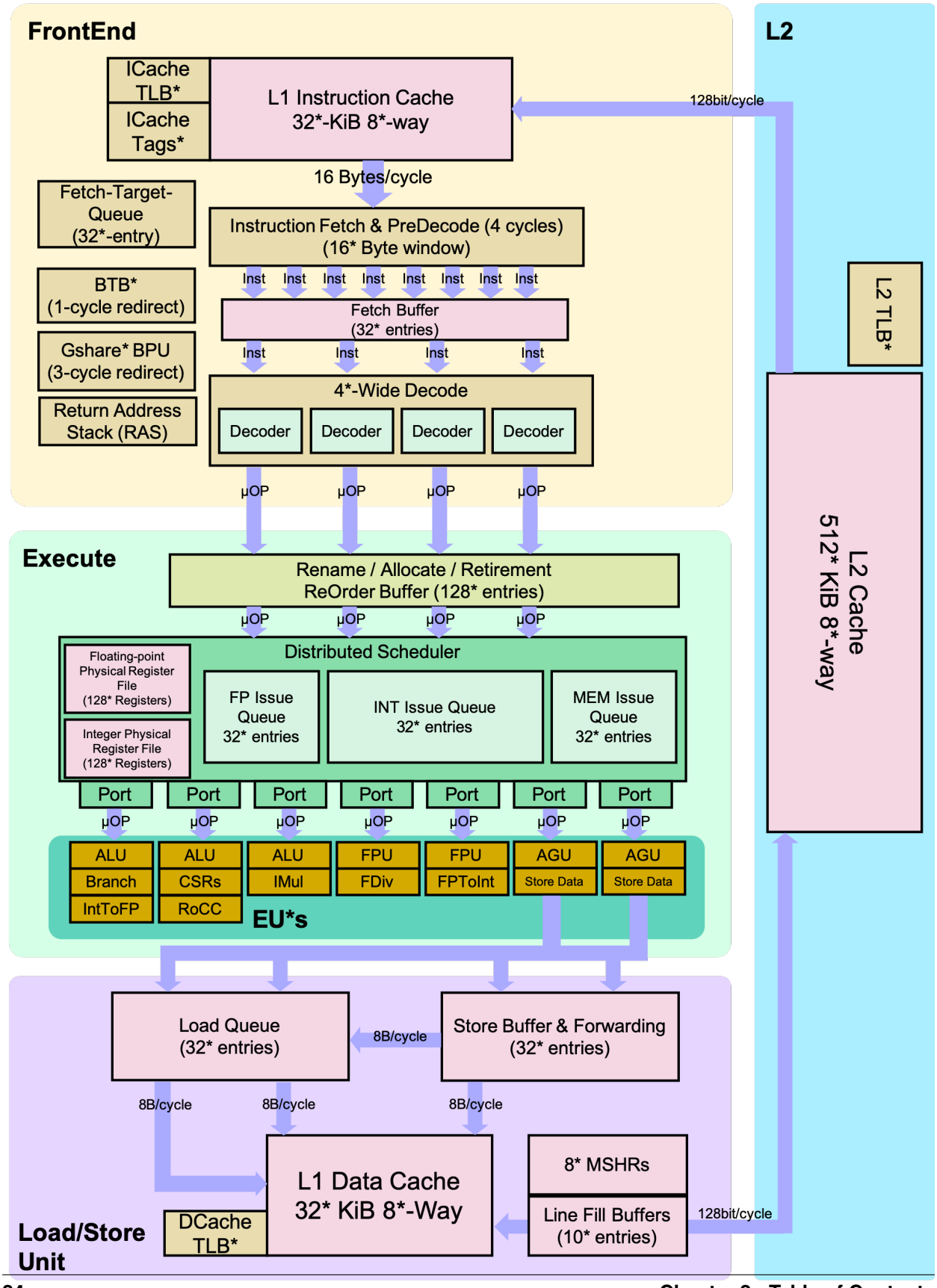
2.3.2 Rocket Core

Rocket is a 5-stage in-order scalar processor core generator, originally developed at UC Berkeley and currently supported by SiFive. The *Rocket core* is used as a component within the *Rocket Chip SoC generator*. A Rocket core combined with L1 caches (data and instruction caches) form a *Rocket tile*. The *Rocket tile* is the replicable component of the *Rocket Chip SoC generator*.

The Rocket core supports the open-source RV64GC RISC-V instruction set and is written in the Chisel hardware construction language. It has an MMU that supports page-based virtual memory, a non-blocking data cache, and a front-end with branch prediction. Branch prediction is configurable and provided by a branch target buffer (BTB), branch history table (BHT), and a return address stack (RAS). For floating-point, Rocket makes use of Berkeley's Chisel implementations of floating-point units. Rocket also supports the RISC-V machine, supervisor, and user privilege levels. A number of parameters are exposed, including the optional support of some ISA extensions (M, A, F, D), the number of floating-point pipeline stages, and the cache and TLB sizes.

For more information, please refer to the GitHub repository, technical report or to this Chisel Community Conference video.

2.3.3 Berkeley Out-of-Order Machine (BOOM)



The Berkeley Out-of-Order Machine (BOOM) is a synthesizable and parameterizable open source RV64GC RISC-V core written in the Chisel hardware construction language. It serves as a drop-in replacement to the Rocket core given by Rocket Chip (replaces the RocketTile with a BoomTile). BOOM is heavily inspired by the MIPS R10k and the Alpha 21264 out-of-order processors. Like the R10k and the 21264, BOOM is a unified physical register file design (also known as "explicit register renaming"). Conceptually, BOOM is broken up into 10 stages: Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback and Commit. However, many of those stages are combined in the current implementation, yielding seven stages: Fetch, Decode/Rename, Rename/Dispatch, Issue/RegisterRead, Execute, Memory and Writeback (Commit occurs asynchronously, so it is not counted as part of the "pipeline").

Additional information about the BOOM micro-architecture can be found in the BOOM documentation pages.

2.3.4 Hwacha

The Hwacha project is developing a new vector architecture for future computer systems that are constrained in their power and energy consumption. The Hwacha project is inspired by traditional vector machines from the 70s and 80s, and lessons learned from our previous vector-thread architectures such as Scale and Maven The Hwacha project includes the Hwacha microarchitecture generator, as well as the `XHwacha` non-standard RISC-V extension. Hwacha does not implement the RISC-V standard vector extension proposal.

For more information on the Hwacha project, please visit the Hwacha website.

To add the Hwacha vector unit to an SoC, you should add the `hwacha.DefaultHwachaConfig` config fragment to the SoC configurations. The Hwacha vector unit uses the RoCC port of a Rocket or BOOM *tile*, and by default connects to the memory system through the *System Bus* (i.e., directly to the L2 cache).

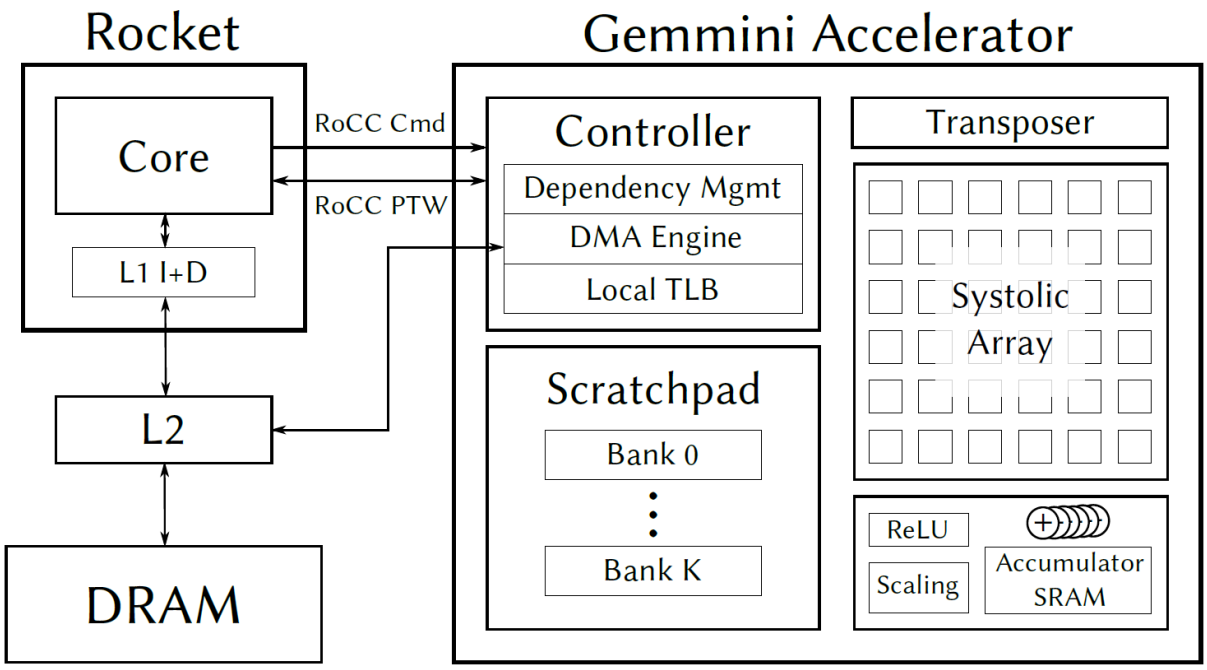
To change the configuration of the Hwacha vector unit, you can write a custom configuration to replace the `DefaultHwachaConfig`. You can view the `DefaultHwachaConfig` under generators/hwacha/src/main/scala/configs.scala to see the possible configuration parameters.

Since Hwacha implements a non-standard RISC-V extension, it requires a unique software toolchain to be able to compile and assemble its vector instructions. To install the Hwacha toolchain, run the `./scripts/build-toolchains.sh esp-tools` command within the root Chipyard directory. This may take a while, and it will install the `esp-tools-install` directory within your Chipyard root directory. `esp-tools` is a fork of `riscv-tools` (formerly a collection of relevant software RISC-V tools) that was enhanced with additional non-standard vector instructions. However, due to the upstreaming of the equivalent RISC-V toolchains, `esp-tools` may not be up-to-date with the latest mainline version of the tools included in it.

2.3.5 Gemmini

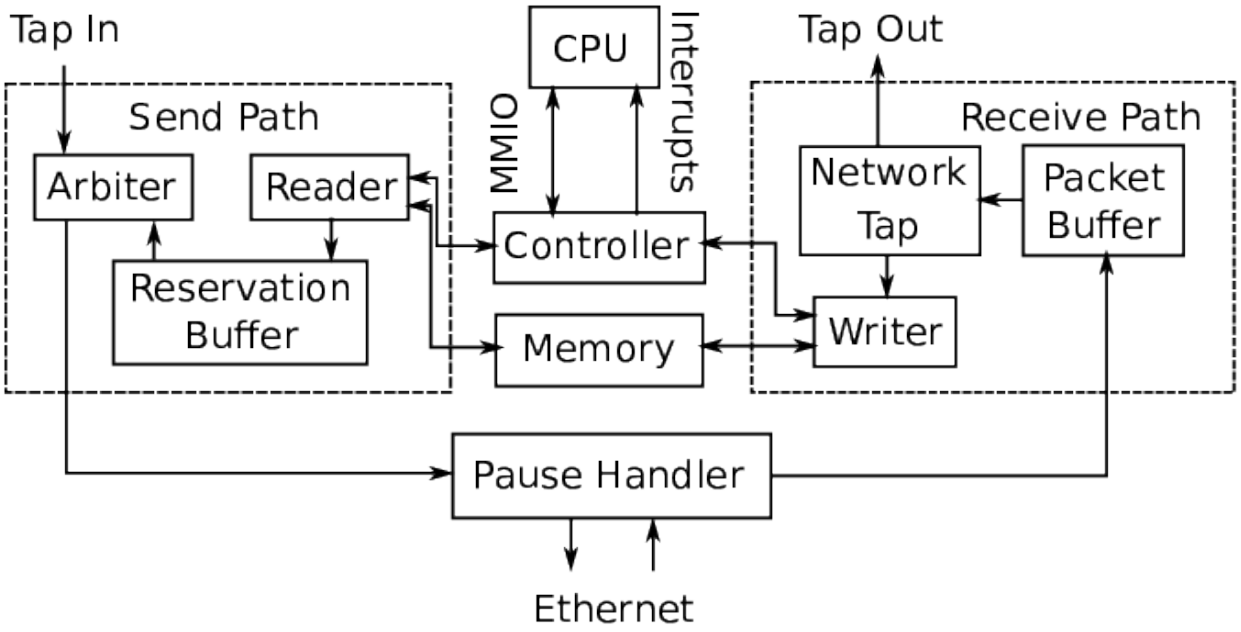
The Gemmini project is developing a full-system, full-stack DNN hardware exploration and evaluation platform. Gemmini enables architects to make useful insights into how different components of the system and software stack (outside of just the accelerator itself) interact to affect overall DNN performance.

Check out Gemmini's documentation to learn how to generate, simulate, and profile DNN accelerators with Gemmini and Chipyard.



2.3.6 IceNet

IceNet is a library of Chisel designs related to networking. The main component of IceNet is IceNIC, a network interface controller that is used primarily in FireSim for multi-node networked simulation. A diagram of IceNet's microarchitecture is shown below.



There are four basic parts of the NIC: the *Controller*, which takes requests from and sends responses to the CPU; the *Send Path*, which reads data from memory and sends it out to the network; the *Receive Path*, which receives data from the network and writes it to memory; and, optionally, the *Pause Handler*, which generates Ethernet pause frames for

the purpose of flow control.

Controller

The controller exposes a set of MMIO registers to the CPU. The device driver writes to registers to request that packets be sent or to provide memory locations to write received data to. Upon the completion of a send request or packet receive, the controller sends an interrupt to the CPU, which clears the completion by reading from another register.

Send Path

The send path begins at the reader, which takes requests from the controller and reads the data from memory.

Since TileLink responses can come back out-of-order, we use a reservation queue to reorder responses so that the packet data can be sent out in the proper order.

The packet data then goes to an arbiter, which can arbitrate access to the outbound network interface between the NIC and one or more "tap in" interfaces, which come from other hardware modules that may want to send Ethernet packets. By default, there are no tap in interfaces, so the arbiter simply passes the output of the reservation buffer through.

Receive Path

The receive path begins with the packet buffer, which buffers data coming in from the network. If there is insufficient space in the buffer, it will drop data at packet granularity to ensure that the NIC does not deliver incomplete packets.

From the packet buffer, the data can optionally go to a network tap, which examines the Ethernet header and select packets to be redirected from the NIC to external modules through one or more "tap out" interfaces. By default, there are no tap out interfaces, so the data will instead go directly to the writer, which writes the data to memory and then sends a completion to the controller.

Pause Handler

IceNIC can be configured to have pause handler, which sits between the send and receive paths and the Ethernet interface. This module tracks the occupancy of the receive packet buffer. If it sees the buffer filling up, it will send an Ethernet pause frame out to the network to block further packets from being sent. If the NIC receives an Ethernet pause frame, the pause handler will block sending from the NIC.

Linux Driver

The default Linux configuration provided by firesim-software contains an IceNet driver. If you launch a FireSim image that has IceNIC on it, the driver will automatically detect the device, and you will be able to use the full Linux networking stack in userspace.

Configuration

To add IceNIC to your design, add `HasPeripheryIceNIC` to your lazy module and `HasPeripheryIceNICModuleImp` to the module implementation. If you are confused about the distinction between lazy module and module implementation, refer to *Cake Pattern / Mixin*.

Then add the `WithIceNIC` config fragment to your configuration. This will define `NICKey`, which IceNIC uses to determine its parameters. The config fragment takes two arguments. The `inBufFlits` argument is the number of 64-bit flits that the input packet buffer can hold and the `usePauser` argument determines whether or not the NIC will have a pause handler.

2.3.7 Test Chip IP

Chipyard includes a Test Chip IP library which provides various hardware widgets that may be useful when designing SoCs. This includes a *Serial Adapter*, *Block Device Controller*, *TileLink SERDES*, *TileLink Switcher*, *TileLink Ring Network*, and *UART Adapter*.

Serial Adapter

The serial adapter is used by tethered test chips to communicate with the host processor. An instance of RISC-V frontend server running on the host CPU can send commands to the serial adapter to read and write data from the memory system. The frontend server uses this functionality to load the test program into memory and to poll for completion of the program. More information on this can be found in *Chipyard Boot Process*.

Block Device Controller

The block device controller provides a generic interface for secondary storage. This device is primarily used in FireSim to interface with a block device software simulation model. The default Linux configuration in firesim-software

To add a block device to your design, add the `WithBlockDevice` config fragment to your configuration.

TileLink SERDES

The TileLink SERDES in the Test Chip IP library allow TileLink memory requests to be serialized so that they can be carried off chip through a serial link. The five TileLink channels are multiplexed over two SERDES channels, one in each direction.

There are three different variants provided by the library, `TLSerdes` exposes a manager interface to the chip, tunnels A, C, and E channels on its outbound link, and tunnels B and D channels on its inbound link. `TLDesser` exposes a client interface to the chip, tunnels A, C, and E on its inbound link, and tunnels B and D on its outbound link. Finally, `TLSerdesser` exposes both client and manager interface to the chip and can tunnel all channels in both directions.

For an example of how to use the SERDES classes, take a look at the `SerdesTest` unit test in the Test Chip IP unit test suite.

TileLink Switcher

The TileLink switcher is used when the chip has multiple possible memory interfaces and you would like to select which channels to map your memory requests to at boot time. It exposes a client node, multiple manager nodes, and a select signal. Depending on the setting of the select signal, requests from the client node will be directed to one of the manager nodes. The select signal must be set before any TileLink messages are sent and be kept stable throughout the remainder of operation. It is not safe to change the select signal once TileLink messages have begun sending.

For an example of how to use the switcher, take a look at the `SwitcherTest` unit test in the Test Chip IP unit tests.

TileLink Ring Network

TestChipIP provides a TLRingNetwork generator that has a similar interface to the TLXbar provided by RocketChip, but uses ring networks internally rather than crossbars. This can be useful for chips with very wide TileLink networks (many cores and L2 banks) that can sacrifice cross-section bandwidth to relieve wire routing congestion. Documentation on how to use the ring network can be found in *The System Bus*. The implementation itself can be found here, and may serve as an example of how to implement your own TileLink network with a different topology.

UART Adapter

The UART Adapter is a device that lives in the TestHarness and connects to the UART port of the DUT to simulate communication over UART (ex. printing out to UART during Linux boot). In addition to working with `stdin/stdout` of the host, it is able to output a UART log to a particular file using `+uartlog=<NAME_OF_FILE>` during simulation.

By default, this UART Adapter is added to all systems within Chipyard by adding the `WithUART` and `WithUARTAdapter` configs.

SPI Flash Model

The SPI flash model is a device that models a simple SPI flash device. It currently only supports single read, quad read, single write, and quad write instructions. The memory is backed by a file which is provided using `+spiflash#=<NAME_OF_FILE>`, where `#` is the SPI flash ID (usually `0`).

2.3.8 SiFive Generators

Chipyard includes several open-source generators developed and maintained by SiFive. These are currently organized within two submodules named `sifive-blocks` and `sifive-cache`.

Last-Level Cache Generator

`sifive-cache` includes last-level cache geneator. The Chipyard framework uses this last-level cache as an L2 cache. To use this L2 cache, you should add the `freechips.rocketchip.subsystem.WithInclusiveCache` config fragment to your SoC configuration. To learn more about configuring this L2 cache, please refer to the *Memory Hierarchy* section.

Peripheral Devices

`sifive-blocks` includes multiple peripheral device generators, such as UART, SPI, PWM, JTAG, GPIO and more.

These peripheral devices usually affect the memory map of the SoC, and its top-level IO as well. To integrate one of these devices in your SoC, you will need to define a custom config fragment with the approriate address for the device using the Rocket Chip parameter system. As an example, for a GPIO device you could add the following config fragment to set the GPIO address to `0x10012000`. This address is the start address for the GPIO configuration registers.

```
class WithGPIO extends Config((site, here, up) => {
  case PeripheryGPIOKey => Seq(
    GPIOParams(address = 0x10012000, width = 4, includeIOF = false))
})
```

Additionally, if the device requires top-level IOs, you will need to define a config fragment to change the top-level configuration of your SoC. When adding a top-level IO, you should also be aware of whether it interacts with the test-harness.

This example instantiates a top-level module with include GPIO ports, and then ties-off the GPIO port inputs to 0 (`false.B`).

Finally, you add the relevant config fragment to the SoC config. For example:

```
class GPIORocketConfig extends Config(
  new chipyard.config.WithGPIO ++                                  // add GPIOs to the␣
↪peripherybus
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

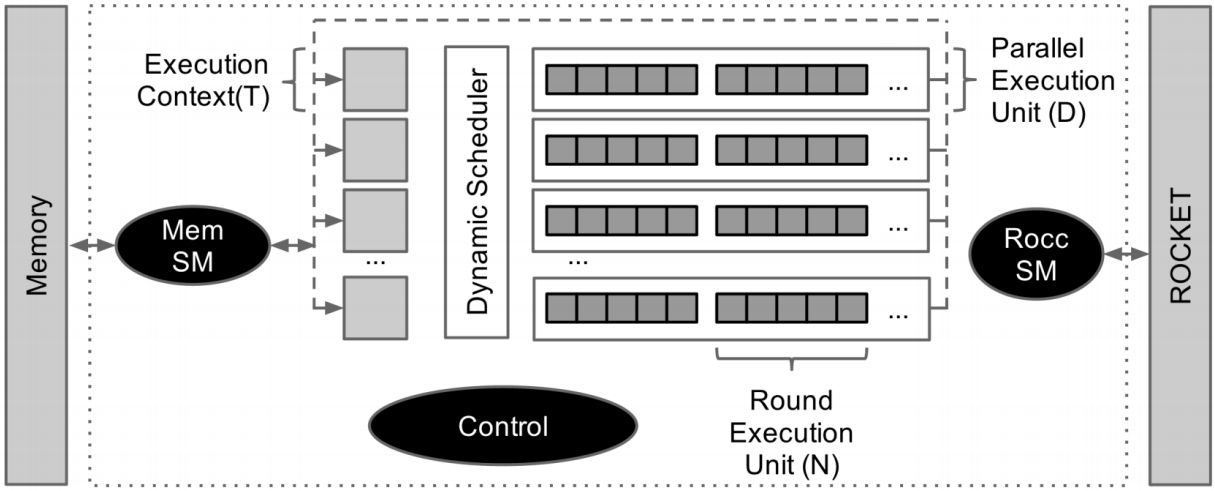
Some of the devices in `sifive-blocks` (such as GPIO) may already have pre-defined config fragments within the Chipyard example project. You may be able to use these config fragments directly, but you should be aware of their addresses within the SoC address map.

2.3.9 SHA3 RoCC Accelerator

The SHA3 accelerator is a basic RoCC accelerator for the SHA3 hashing algorithm. We like using SHA3 in Chipyard tutorial content because it is a self-contained, simple example of integrating a custom accelerator into Chipyard.

Introduction

Secure hashing algorithms represent a class of hashing functions that provide four attributes: ease of hash computation, inability to generate the message from the hash (one-way property), inability to change the message and not the hash (weakly collision free property), and inability to find two messages with the same hash (strongly collision free property). The National Institute of Standards and Technology (NIST) recently held a competition for a new algorithm to be added to its set of Secure Hashing Algorithms (SHA). In 2012 the winner was determined to be the Keccak hashing function and a rough specification for SHA3 was established. The algorithm operates on variable length messages with a sponge function, and thus alternates between absorbing chunks of the message into a set of state bits and permuting the state. The absorbing is a simple bitwise XOR while the permutation is a more complex function composed of several operations, χ, θ, ρ, π, ι, that all perform various bitwise operations, including rotations, parity calculations, XORs, etc. The Keccak hashing function is parameterized for different sizes of state and message chunks but for this accelerator we will only support the Keccak-256 variant with 1600 bits of state and 1088 bit message chunks. A diagram of the SHA3 accelerator is shown below.



Technical Details

The accelerator is designed around three sub-systems, an interface with the processor, an interface with memory, and the actual hashing computation system. The interface with the processor is designed using the ROCC interface for coprocessors integrating with the RISC-V Rocket/BOOM processor. It includes the ability to transfer two 64 bit words

to the co-processor, the request for a return value, and a small field for the function requested. The accelerator receives these requests using a ready/valid interface. The ROCC instruction is parsed and the needed information is stored into a execution context. The execution context contains the memory address of the message being hashed, the memory address to store the resulting hash in, the length of the message, and several other control fields.

Once the execution context is valid the memory subsystem then begins to fetch chunks of the message. The memory subsystem is fully decoupled from the other subsystems and maintains a single full round memory buffers. The accelerators memory interface can provide a maximum of one 64 bit word per cycle which corresponds to 17 requests needed to fill a buffer (the size is dictated by the SHA3 algorithm). Memory requests to fill these buffers are sent out as rapidly as the memory interface can handle, with a tag field set to allow the different memory buffers requests to be distinguished, as they may be returned out of order. Once the memory subsystem has filled a buffer the control unit absorbs the buffer into the execution context, at which point the execution context is free to begin permutation, and the memory buffer is free to send more memory requests.

After the buffer is absorbed, the hashing computation subsystem begins the permutation operations. Once the message is fully hashed, the hash is written to memory with a simple state machine.

Using a SHA3 Accelerator

Since the SHA3 accelerator is designed as a RoCC accelerator, it can be mixed into a Rocket or BOOM core by overriding the `BuildRoCC` key. The config fragment is defined in the SHA3 generator. An example configuration highlighting the use of this config fragment is shown here:

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

The SHA3 example baremetal and Linux tests are located in the SHA3 repository. Please refer to its README.md for more information on how to run/build the tests.

2.3.10 CVA6 Core

CVA6 (previously called Ariane) is a 6-stage in-order scalar processor core, originally developed at ETH-Zurich by F. Zaruba and L. Benini. The *CVA6 core* is wrapped in an *CVA6 tile* so it can be used as a component within the *Rocket Chip SoC generator*. The core by itself exposes an AXI interface, interrupt ports, and other misc. ports that are connected from within the tile to TileLink buses and other parameterization signals.

Warning: Since the core uses an AXI interface to connect to memory, it is highly recommended to use the core in a single-core setup (since AXI is a non-coherent memory interface).

While the core itself is not a generator, we expose the same parameterization that the CVA6 core provides (i.e. change branch prediction parameters).

Warning: This target does not support Verilator simulation at this time. Please use VCS.

For more information, please refer to the GitHub repository.

2.3.11 Ibex Core

Ibex is a parameterizable RV32IMC embedded core written in SystemVerilog, currently maintained by lowRISC. The *Ibex core* is wrapped in an *Ibex tile* so it can be used with the *Rocket Chip SoC generator*. The core exposes a custom memory interface, interrupt ports, and other misc. ports that are connected from within the tile to TileLink buses and other parameterization signals.

Warning: The Ibex mtvec register is 256 byte aligned. When writing/running tests, ensure that the trap vector is also 256 byte aligned.

Warning: The Ibex reset vector is located at BOOT_ADDR + 0x80.

While the core itself is not a generator, we expose the same parameterization that the Ibex core provides so that all supported Ibex configurations are available.

For more information, see the GitHub repository for Ibex.

2.3.12 FFT Generator

The FFT generator is a parameterizable fft accelerator.

Configuration

The following configuration creates an 8-point FFT:

```
class FFTRocketConfig extends Config(
  new fftgenerator.WithFFTGenerator(baseAddr=0x2000, numPoints=8, width=16, decPt=8)␣
↪++ // add 8-point mmio fft at 0x2000 with 16bit fixed-point numbers.
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

`baseAddress` specifies the starting address of the FFT's read and write lanes. The FFT write lane is always located at `baseAddress`. There is 1 read lane per output point; since this config specifies an 8-point FFT, there will be 8 read lanes. Read lane `i` (which can be loaded from to retrieve output point `i`) will be located at `baseAddr + 64bits (assuming 64bit system) + (i * 8)`. `baseAddress` should be 64-bit aligned

`width` is the size of input points in binary. A width of `w` means that each point will have `w` bits for the real component and `w` bits for the imaginary component, yielding a total of *2w* bits per point. `decPt` is the location of the decimal point in the fixed-precision representation of each point's real and imaginary value. In the Config above, each point is *32* bits wide, with *16* bits used to represent the real component and *16* bits used to represent the imaginary component. Within the *16* bits for each component, the *8* LSB are used to represent the decimal component of the value and the remaining (8) MSB are used to represent the integer component. Both the real and imaginary components use a fixed-precision representation.

To build a simulation of this example Chipyard config, run the following commands:

```
cd sims/verilator # or "cd sims/vcs"
make CONFIG=FFTRocketConfig
```

Usage and Testing

Points are passed into the FFT via the single write lane. In C pseudocode, this might look like:

```
for (int i = 0; i < num_points; i++) {
    // FFT_WRITE_LANE = baseAddress
    uint32_t write_val = points[i];
    volatile uint32_t* ptr = (volatile uint32_t*) FFT_WRITE_LANE;
    *ptr = write_val;
}
```

Once the correct number of inputs are passed in (in the config above, 8 values would be passed in), the read lanes can be read from (again in C pseudocode):

```
for (int i = 0; i < num_points; i++) {
    // FFT_RD_LANE_BASE = baseAddress + 64bits (for write lane)
    volatile uint32_t* ptr_0 = (volatile uint32_t*) (FFT_RD_LANE_BASE + (i * 8));
    uint32_t read_val = *ptr_0;
}
```

The `fft.c` test file in the `tests/` directory can be used to verify the fft's functionality on an SoC built with `FFTRocketConfig`.

Acknowledgements

The code for the FFT Generator was adapted from the ADEPT Lab at UC Berkeley's Hydra Spine project.

Authors for the original project (in no particular order):

- **James Dunn, UC Berkeley (dunn [at] eecs [dot] berkeley [dot] edu)**
  - `Deserialize.scala`
  - `Tail.scala`
  - `Unscramble.scala`
- **Stevo Bailey (stevo.bailey [at] berkeley [dot] edu)**
  - `FFT.scala`

2.3.13 NVDLA

NVDLA is an open-source deep learning accelerator developed by NVIDIA. The *NVDLA* is attached as a TileLink peripheral so it can be used as a component within the *Rocket Chip SoC generator*. The accelerator by itself exposes an AXI memory interface (or two if you use the "Large" configuration), a control interface, and an interrupt line. The main way to use the accelerator in Chipyard is to use the NVDLA SW repository that was ported to work on FireSim Linux. However, you can also use the accelerator in baremetal simulations (refer to `tests/nvdla.c`).

For more information on both the HW architecture and the SW, please visit their website.

NVDLA Software with FireMarshal

Located at `software/nvdla-workload` is a FireMarshal-based workload to boot Linux with the proper NVDLA drivers. Refer to that `README.md` for more information on how to run a simulation.

2.3.14 Sodor Core

Sodor is a collection of 5 simple RV32MI cores designed for educational purpose. The *Sodor core* is wrapped in an tile during generation so it can be used as a component within the *Rocket Chip SoC generator*. The cores contain a small scratchpad memory to which the program are loaded through a TileLink slave port, and the cores **DO NOT** support external memory.

The five available cores and their corresponding generator configuration are:

- 1-stage (essentially an ISA simulator) - `Sodor1StageConfig`
- 2-stage (demonstrates pipelining in Chisel) - `Sodor2StageConfig`
- 3-stage (uses sequential memory; supports both Harvard (`Sodor3StageConfig`) and Princeton (`Sodor3StageSinglePortConfig`) versions)
- 5-stage (can toggle between fully bypassed or fully interlocked) - `Sodor5StageConfig`
- "bus"-based micro-coded implementation - `SodorUCodeConfig`

For more information, please refer to the GitHub repository.

2.4 Development Tools

The Chipyard framework relays heavily on a set of Scala-based tools. The following pages will introduce them, and how we can use them in order to generate flexible designs.

2.4.1 Chisel

Chisel is an open-source hardware description language embedded in Scala. It supports advanced hardware design using highly parameterized generators and supports things such as Rocket Chip and BOOM.

After writing Chisel, there are multiple steps before the Chisel source code "turns into" Verilog. First is the compilation step. If Chisel is thought as a library within Scala, then these classes being built are just Scala classes which call Chisel functions. Thus, any errors that you get in compiling the Scala/Chisel files are errors that you have violated the typing system, messed up syntax, or more. After the compilation is complete, elaboration begins. The Chisel generator starts elaboration using the module and configuration classes passed to it. This is where the Chisel "library functions" are called with the parameters given and Chisel tries to construct a circuit based on the Chisel code. If a runtime error happens here, Chisel is stating that it cannot "build" your circuit due to "violations" between your code and the Chisel "library". However, if that passes, the output of the generator gives you an FIRRTL file and other misc collateral! See *FIRRTL* for more information on how to get a FIRRTL file to Verilog.

For an interactive tutorial on how to use Chisel and get started please visit the Chisel Bootcamp. Otherwise, for all things Chisel related including API documentation, news, etc, visit their website.

2.4.2 FIRRTL

FIRRTL is an intermediate representation of your circuit. It is emitted by the Chisel compiler and is used to translate Chisel source files into another representation such as Verilog. Without going into too much detail, FIRRTL is consumed by a FIRRTL compiler (another Scala program) which passes the circuit through a series of circuit-level transformations. An example of a FIRRTL pass (transformation) is one that optimizes out unused signals. Once the transformations are done, a Verilog file is emitted and the build process is done.

For more information on please visit their website.

2.4.3 Treadle and FIRRTL Interpreter

Treadle and FIRRTL Interpreter are circuit simulators that directly execute FIRRTL (specifically low-firrtl IR). Treadle is the replacement for FIRRTL Interpreter but FIRRTL Interpreter is still used within some projects. Treadle is useful for simulating modules in a larger SoC design. Many projects use Treadle for interactive debugging and a low-overhead simulator.

2.4.4 Chisel Testers

Chisel Testers is a library for writing tests for Chisel designs. It provides a Scala API for interacting with a DUT. It can use multiple backends, including things such as Treadle and Verilator. See *Treadle and FIRRTL Interpreter* and *Software RTL Simulation* for more information on these simulation methods.

2.4.5 Dsptools

Dsptools is a Chisel library for writing custom signal processing hardware. Additionally, dsptools is useful for integrating custom signal processing hardware into an SoC (especially a Rocket-based SoC).

Some features:

- Complex type
- Typeclasses for writing polymorphic hardware generators * For example, write one FIR filter generator that works for real or complex inputs
- Extensions to Chisel testers for fixed point and floating point types
- A diplomatic implementation of AXI4-Stream
- Models for verifying APB, AXI-4, and TileLink interfaces with chisel-testers
- DSP building blocks

2.4.6 Barstools

Barstools is a collection of useful FIRRTL transformations and compilers to help the build process. Included in the tools are a MacroCompiler (used to map Chisel memory constructs to vendor SRAMs), FIRRTL transforms (to separate harness and top-level SoC files), and more.

Mapping technology SRAMs (MacroCompiler)

If you are planning on building a real chip, it is likely that you will plan on using some amount of static random access memory, or SRAM. SRAM macros offer superior storage density over flip-flop arrays at the cost of restricting the number of read or write transactions that can happen in a cycle. Unlike in Verilog, these types of sequential memory elements are first-class primitives in Chisel and FIRRTL (`SeqMem` elements). This allows Chisel designs to contain abstract instantiations of sequential memory elements without knowing the underlying implementation or process technology.

Modern CAD tools typically cannot synthesize SRAMs from a high-level RTL description. This, unfortunately, requires the designer to include the SRAM instantiation in the source RTL, which removes its process portability. In Verilog-entry designs, it is possible to create a layer of abstraction that allows a new process technology to implement a specific sequential memory block in a wrapper module. However, this method can be fragile and laborious.

The FIRRTL compiler contains a transformation to replace the `SeqMem` primitives called `ReplSeqMem`. This simply converts all `SeqMem` instances above a size threshold into external module references. An external module reference

is a FIRRTL construct that enables a design to reference a module without describing its contents, only its inputs and outputs. A list of unique SRAM configurations is output to a `.conf` file by FIRRTL, which is used to map technology SRAMs. Without this transform, FIRRTL will map all `SeqMem` s to flip-flop arrays with equivalent behavior, which may lead to a design that is difficult to route.

The `.conf` file is consumed by a tool called MacroCompiler, which is part of the *Barstools* scala package. MacroCompiler is also passed an `.mdf` file that describes the available list of technology SRAMs or the capabilities of the SRAM compiler, if one is provided by the foundry. Typically a foundry SRAM compiler will be able to generate a set of different SRAMs collateral based on some requirements on size, aspect ratio, etc. (see *SRAM MDF Fields*). Using a user-customizable cost function, MacroCompiler will select the SRAMs that are the best fit for each dimensionality in the `.conf` file. This may include over provisioning (e.g. using a 64x1024 SRAM for a requested 60x1024, if the latter is not available) or arraying. Arraying can be done in both width and depth, as well as to solve masking constraints. For example, a 128x2048 array could be composed of four 64x1024 arrays, with two macros in parallel to create two 128x1024 virtual SRAMs which are combinationally muxed to add depth. If this macro requires byte-granularity write masking, but no technology SRAMs support masking, then the tool may choose to use thirty-two 8x1024 arrays in a similar configuration. You may wish to create a cache of your available SRAM macros either manually, or via a script. A reference script for creating a JSON of your SRAM macros is in the asap7 technology library folder. For information on writing `.mdf` files, look at MDF on github and a brief description in *SRAM MDF Fields* section.

The output of MacroCompiler is a Verilog file containing modules that wrap the technology SRAMs into the specified interface names from the `.conf`. If the technology supports an SRAM compiler, then MacroCompiler will also emit HammerIR that can be passed to Hammer to run the compiler itself and generate design collateral. Documentation for SRAM compilers is forthcoming.

MacroCompiler Options

MacroCompiler accepts many command-line parameters which affect how it maps `SeqMem` s to technology specific macros. This highest level option `--mode` specifies in general how MacroCompiler should map the input `SeqMem` s to technology macros. The `strict` value forces MacroCompiler to map all memories to technology macros and error if it is unable to do so. The `synflops` value forces MacroCompiler to map all memories to flip flops. The `compileandsynflops` value instructs MacroCompiler to use the technology compiler to determine sizes of technology macros used but to then create mock versions of these macros with flip flops. The `fallbacksynflops` value causes MacroCompiler to compile all possible memories to technology macros but when unable to do so to use flip flops to implement the remaining memories. The final and default value, `compileavailable`, instructs MacroCompiler to compile all memories to the technology macros and do nothing if it is unable to map them.

Most of the rest of the options are used to control where different inputs and outputs are expected and produced. The option `--macro-conf` is the file that contains the set of input `SeqMem` configurations to map in the `.conf` format described above. The option `--macro-mdf` also describes the input `SeqMem` s but is instead in the `.mdf` format. The option `--library` is an `.mdf` description of the available technology macros that can be mapped to. This file could be a list of fixed size memories often referred to as a cache of macros, or a description of what size memories could be made available through some technology specific process (usually an SRAM compiler), or a mix of both. The option `--use-compiler` instructs MacroCompiler that it is allowed to use any compilers listed in the `--library` specification. If this option is not set MacroCompiler will only map to macros directly listed in the `--library` specification. The `--verilog` option specifies where MacroCompiler will write the verilog containing the new technology mapped memories. The `--firrtl` option similarly specifies where MacroCompiler will write the FIRRTL that will be used to generate this verilog. This option is optional and no FIRRTL will be emitted if it is not specified. The `--hammer-ir` option specifies where MacroCompiler will write the details of which macros need to be generated from a technology compiler. This option is not needed if `--use-compiler` is not specified. This file can then be passed to HAMMER to have it run the technology compiler producing the associated macro collateral. The `--cost-func` option allows the user to specify a different cost function for the mapping task. Because the mapping of memories is a multi-dimensional space spanning performance, power, and area, the cost function setting of MacroCompiler allows the user to tune the mapping to their preference. The default option is a reasonable heuristic that attempts to minimize the number of technology macros instantiated per `SeqMem`

without wasting too many memory bits. There are two ways to add additional cost functions. First, you can simply write another one in scala and call *registerCostMetric* which then enables you to pass its name to this command-line flag. Second, there is a pre-defined *ExternalMetric* which will execute a program (passed in as a path) with the MDF description of the memory being compiled and the memory being proposed as a mapping. The program should print a floating point number which is the cost for this mapping, if no number is printed MacroCompiler will assume this is an illegal mapping. The `--cost-param` option allows the user to specify parameters to pass to the cost function if the cost function supports that. The `--force-synflops [mem]` options allows the user to override any heuristics in MacroCompiler and force it to map the given memory to flip-flops. Likewise, the `--force-compile [mem]` option allows the user to force MacroCompiler to map the given `mem` to a technology macro.

SRAM MDF Fields

Technology SRAM macros described in MDF can be defined at three levels of detail. A single instance can be defined with the *SRAMMacro* format. A group of instances that share the number and type of ports but vary in width and depth can be defined with the *SRAMGroup* format. A set of groups of SRAMs that can be generated together from a single source like a compiler can be defined with the *SRAMCompiler* format.

At the most concrete level the *SRAMMAcro* defines a particular instance of an SRAM. That includes its functional attributes such as its width, depth, and number of access ports. These ports can be read, write, or read and write ports, and the instance can have any number. In order to correctly map these functional ports to the physical instance, each port is described in a list of sub-structures, in the parent instance's structure. Each port is only required to have an address and data field, but can have many other optional fields. These optional fields include a clock, write enable, read enable, chip enable, mask and its granularity. The mask field can have a different granularity than the data field, e.g. it could be a bit mask or a byte mask. Each field must also specify its polarity, whether it is active high or active low.

The specific JSON file format described above is here. A reference cache of SRAMs from the nangate45 technology library is available here.

In addition to these functional descriptions of the SRAM there are also other fields that specify physical/implementation characteristics. These include the threshold voltage, the mux factor, as well as a list of extra non-functional ports.

The next level of detail, an *SRAMGroup* includes a range of depths and widths, as well as a set of threshold voltages. A range has a lower bound, upper bound, and a step size. The least concrete level, an *SRAMCompiler* is simply a set of *SRAMGroups*.

Separating the Top module from the TestHarness module

Unlike the FireSim and Software simulation flows, a VLSI flow needs to separate the test harness and the chip (a.k.a. DUT) into separate files. This is necessary to facilitate post-synthesis and post-place-and-route simulation, as the module names in the RTL and gate-level verilog files would collide. Simulations, after your design goes through a VLSI flow, will use the verilog netlist generated from the flow and will need an untouched test harness to drive it. Separating these components into separate files makes this straightforward. Without the separation the file that included the test harness would also redefine the DUT which is often disallowed in simulation tools. To do this, there is a FIRRTL `App` in *Barstools* called `GenerateTopAndHarness`, which runs the appropriate transforms to elaborate the modules separately. This also renames modules in the test harness so that any modules that are instantiated in both the test harness and the chip are uniquified.

Note: For VLSI projects, this `App` is run instead of the normal FIRRTL `App` to elaborate Verilog.

Macro Description Format

The SRAM technology macros and IO cells are described in a json format called Macro Description Format (MDF). MDF is specialized for each type of macro it supports. The specialization is defined in their respective sections.

Mapping technology IO cells

Like technology SRAMs, IO cells are almost always included in digital ASIC designs to allow pin configurability, increase the voltage level of the IO signal, and provide ESD protection. Unlike SRAMs, there is no corresponding primitive in Chisel or FIRRTL. However, this problem can be solved similarly to `SeqMems` by leveraging the strong typing available in these scala-based tools. We are actively developing a FIRRTL transform that will automatically configure, map, and connect technology IO cells. Stay tuned for more information!

In the meantime, it is recommended that you instantiate the IO cells in your Chisel design. This, unfortunately, breaks the process-agnostic RTL abstraction, so it is recommended that inclusion of these cells be configurable using the `rocket-chip` parameterization system. The simplest way to do this is to have a config fragment that when included updates instantiates the IO cells and connects them in the test harness. When simulating chip-specific designs, it is important to include the IO cells. The IO cell behavioral models will often assert if they are connected incorrectly, which is a useful runtime check. They also keep the IO interface at the chip and test harness boundary (see *Separating the Top module from the TestHarness module*) consistent after synthesis and place-and-route, which allows the RTL simulation test harness to be reused.

2.4.7 Dromajo

Dromajo is a RV64GC functional simulator designed for co-simulation. To use it as a co-simulator, it requires you to pass the committed trace of instructions coming from the core into the tool. Within Chipyard, this is done by connecting to the *TracePort'* signals that are piped to the top level of the DUT. While the Rocket core does have a *TracePort*, it does not provide the committed write data that Dromajo requires. Thus, Dromajo uses the *ExtendedTracePort* only probided by BOOM (BOOM is the only core that supports Dromajo co-simulation). An example of a divergence and Dromajo's printout is shown below.

```
[error] EMU PC ffffffe001055d84, DUT PC ffffffe001055d84
[error] EMU INSN 14102973, DUT INSN 14102973
[error] EMU WDATA 00000000000220d6, DUT WDATA 00000000000220d4
[error] EMU MSTATUS a000000a0, DUT MSTATUS 00000000
[error] DUT pending exception -1 pending interrupt -1
```

Dromajo shows the divergence compared to simulation (PC, inst, inst-bits, write data, etc) and also provides the register state on failure. It is useful to catch bugs that affect architectural state before a simulation hangs or crashes.

To use Dromajo with BOOM, refer to *Debugging RTL* section on Dromajo.

2.5 VLSI Flow

The Chipyard framework aims to provide wrappers for a general VLSI flow. In particular, we aim to support the Hammer physical design generator flow.

2.5.1 Building A Chip

In this section, we will discuss many of the ASIC-specific transforms and methodologies within Chipyard. For the full documentation on how to use the VLSI tool flow, see the Hammer Documentation.

Transforming the RTL

Building a chip requires specializing the generic verilog emitted by FIRRTL to adhere to the constraints imposed by the technology used for fabrication. This includes mapping Chisel memories to available technology macros such as SRAMs, mapping the input and output of your chip to connect to technology IO cells, see *Barstools*. In addition to these required transformations, it may also be beneficial to transform the RTL to make it more amenable to hierarchical physical design easier. This often includes modifying the logical hierarchy to match the physical hierarchy through grouping components together or flattening components into a single larger module.

Modifying the logical hierarchy

Building a large or complex chip often requires using hierarchical design to place and route sections of the chip separately. In addition, the design as written in Chipyard may not have a hierarchy that matches the physical hierarchy that would work best in the place and route tool. In order to reorganize the design to have its logical hierarchy match its physical hierarchy there are several FIRRTL transformations that can be run. These include grouping, which pull several modules into a larger one, and flattening, which dissolves a modules boundary leaving its components in its containing module. These transformations can be applied repeatedly to different parts of the design to arrange it as the physical designer sees fit. More details on how to use these transformations to reorganize the design hierarchy are forthcoming.

Creating a floorplan

An ASIC floorplan is a specification that the place-and-route tools will follow when placing instances in the design. This includes the top-level chip dimensions, placement of SRAM macros, placement of custom (analog) circuits, IO cell placement, bump or wirebond pad placement, blockages, hierarchical boundaries, and pin placement.

Much of the design effort that goes into building a chip involves developing optimal floorplans for the instance of the design that is being manufactured. Often this is a highly manual and iterative process which consumes much of the physical designer's time. This cost becomes increasingly apparent as the parameterization space grows rapidly when using tools like Chisel- cycle times are hampered by the human labor that is required to floorplan each instance of the design. The Hammer team is actively developing methods of improving the agility of floorplanning for generator-based designs, like those that use Chisel. The libraries we are developing will emit Hammer IR that can be passed directly to the Hammer tool without the need for human intervention. Stay tuned for more information.

In the meantime, see the Hammer Documentation for information on the Hammer IR floorplan API. It is possible to write this IR directly, or to generate it using simple python scripts. While we certainly look forward to having a more featureful toolkit, we have built many chips to date in this way.

Running the VLSI tool flow

For the full documentation on how to use the VLSI tool flow, see the Hammer Documentation. For an example of how to use the VLSI in the context of Chipyard, see *ASAP7 Tutorial*.

2.5.2 Core Hammer

Hammer is a physical design flow which encourages reusability by partitioning physical design specifications into three distinct concerns: design, CAD tool, and process technology. Hammer wraps around vendor specific technologies and tools to provide a single API to address ASIC design concerns. Hammer allows for reusability in ASIC design while still providing the designers leeway to make their own modifications.

For more information, read the Hammer paper and see the GitHub repository and associated documentation.

Hammer implements a VLSI flow using the following high-level constructs:

Actions

Actions are the top-level tasks Hammer is capable of executing (e.g. synthesis, place-and-route, etc.)

Steps

Steps are the sub-components of actions that individually addressable in Hammer (e.g. placement in the place-and-route action).

Hooks

Hooks are modifications to steps or actions that are programmatically defined in a Hammer configuration.

2.5.3 Configuration (Hammer IR)

To configure a Hammer flow, supply a set `yaml` or `json` configuration files that chooses the tool and technology plugins and versions as well as any design specific configuration options. Collectively, this configuration API is referred to as Hammer IR and can be generated from higher-level abstractions.

The current set of all available Hammer APIs is codified here.

2.5.4 Tool Plugins

Hammer supports separately managed plugins for different CAD tool vendors. You may be able to acquire access to the included Cadence, Synopsys, and Mentor plugins repositories with permission from the respective CAD tool vendor. The types of tools (by Hammer names) supported currently include:

- synthesis
- par
- drc
- lvs
- sram_generator
- sim
- power
- pcb

Several configuration variables are needed to configure your tool plugin of choice.

First, select which tool to use for each action by setting `vlsi.core.<tool_type>_tool` to the name of your tool, e.g. `vlsi.core.par_tool: "innovus"`.

Then, point Hammer to the folder that contains your tool plugin by setting `vlsi.core.<tool_type>_tool_path`. This directory should include a folder with the name of the tool, which itself includes a python file `__init__.py` and a yaml file `defaults.yml`. Customize the version of the tool by setting `<tool_type>.<tool_name>.version` to a tool specific string.

The `__init__.py` file should contain a variable, `tool`, that points to the class implementing this tool. This class should be a subclass of `Hammer<tool_type>Tool`, which will be a subclass of `HammerTool`. The class should implement methods for all the tool's steps.

The `defaults.yml` file contains tool-specific configuration variables. The defaults may be overridden as necessary.

2.5.5 Technology Plugins

Hammer supports separately managed technology plugins to satisfy NDAs. You may be able to acquire access to certain pre-built technology plugins with permission from the technology vendor. Or, to build your own tech plugin, you need at least a `<tech_name>.tech.json` and `defaults.yml`. An `__init__.py` is optional if there are any technology-specific methods or hooks to run.

The ASAP7 plugin is a good starting point for setting up a technology plugin because it is an open-source example that is not suitable for taping out a chip. Refer to Hammer's documentation for the schema and detailed setup instructions.

Several configuration variables are needed to configure your technology of choice.

First, choose the technology, e.g. `vlsi.core.technology: asap7`, then point to the location with the PDK tarball with `technology.<tech_name>.tarball_dir` or pre-installed directory with `technology.<tech_name>.install_dir` and (if applicable) the plugin repository with `vlsi.core.technology_path`.

Technology-specific options such as supplies, MMMC corners, etc. are defined in their respective `vlsi.inputs..` configurations. Options for the most common use case are already defined in the technology's `defaults.yml` and can be overridden by the user.

2.5.6 Using Hammer To Place and Route a Custom Block

Important: In order to use the Hammer VLSI flow, you need access to Hammer tools and technology plugins. You can obtain these by emailing hammer-plugins-access@lists.berkeley.edu with a request for which plugin(s) you would like access to. Make sure your email includes your github ID and proof (through affiliation or otherwise) that you have licensed access to relevant tools.

Initialize the Hammer Plug-ins

In the Chipyard root, run:

```
./scripts/init-vlsi.sh <tech-plugin-name>
```

This will pull the Hammer & CAD tool plugin submodules, assuming the technology plugins are available on github. Currently only the asap7 technology plugin is available on github. If you have additional private technology plugins (this is a typical use-case for proprietry process technologies with require NDAs and secure servers), you can clone them directly into VLSI directory with the name `hammer-<tech-plugin-name>-plugin`. For example, for an imaginary process technology called tsmintel3:

```
cd vlsi
git clone git@my-secure-server.berkeley.edu:tsmintel3/hammer-tsmintel3-plugin.git
```

Next, we define the Hammer environment into the shell:

```
cd vlsi    # (if you haven't done so yet)
export HAMMER_HOME=$PWD/hammer
source $HAMMER_HOME/sourceme.sh
```

Note: Some VLSI EDA tools are supported only on RHEL-based operating systems. We recommend using Chipyard on RHEL7 and above. However, many VLSI server still have old operating systems such as RHEL6, which have software packages older than the basic chipyard requirements. In order to build Chipyard on RHEL6, you will likely

need to use tool packages such as devtoolset (for example, devtoolset-8) and/or build from source gcc, git, gmake, make, dtc, cc, bison, libexpat and liby.

Setting up the Hammer Configuration Files

The first configuration file that needs to be set up is the Hammer environment configuration file `env.yml`. In this file you need to set the paths to the EDA tools and license servers you will be using. You do not have to fill all the fields in this configuration file, you only need to fill in the paths for the tools that you will be using. If you are working within a shared server farm environment with an LSF cluster setup (for example, the Berkeley Wireless Research Center), please note the additional possible environment configuration listed in the *Advanced Environment Setup* segment of this documentation page.

Hammer relies on YAML-based configuration files. While these configuration can be consolidated within a single files (as is the case in the ASAP7 tutorial *ASAP7 Tutorial* and the `nangate45` OpenRoad example), the generally suggested way to work with an arbitrary process technology or tools plugins would be to use three configuration files, matching the three Hammer concerns - tools, tech, and design. The `vlsi` directory includes three such example configuration files matching the three concerns: `example-tools.yml`, `example-tech.yml`, and `example-design.yml`.

The `example-tools.yml` file configures which EDA tools hammer will use. This example file uses Cadence Innovus, Genus and Voltus, Synopsys VCS, and Mentor Calibre (which are likely the tools you will use if you're working in the Berkeley Wireless Research Center). Note that tool versions are highly sensitive to the process-technology in-use. Hence, tool versions that work with one process technology may not work with another.

The `example-design.yml` file contains basic build system information (how many cores/threads to use, etc.), as well as configurations that are specific to the design we are working on such as clock signal name and frequency, power modes, floorplan, and additional constraints that we will add later on.

Finally, the `example-tech.yml` file is a template file for a process technology plugin configuration. We will copy this file, and replace its fields with the appropriate process technology details for the tech plugin that we have access to. For example, for the `asap7` tech plugin, we will replace the <tech_name> field with "asap7" and the path to the process technology files installation directory. The technology plugin (which for ASAP7 is within Hammer) will define the technology node and other parameters.

We recommend copying these example configuration files and customizing them with a different name, so you can have different configuration files for different process technologies and designs (e.g. create tech-tsmintel3.yml from example-tech.yml)

Building the Design

After we have set the configuration files, we will now elaborate our Chipyard Chisel design into Verilog, while also performing the required transformations in order to make the Verilog VLSI-friendly. Additionally, we will automatically generate another set of Hammer configuration files matching to this design, which will be used in order to configure the physical design tools. We will do so by calling `make buildfile` with appropriate Chipyard configuration variables and Hammer configuration files. As in the rest of the Chipyard flows, we specify our SoC configuration using the `CONFIG` make variable. However, unlike the rest of the Chipyard flows, in the case of physical design we might be interested in working in a hierarchical fashion and therefore we would like to work on a single module. Therefore, we can also specify a `VLSI_TOP` make variable with the same of a specific Verilog module (which should also match the name of the equivalent Chisel module) which we would like to work on. The makefile will automatically call tools such as Barstools and the MacroCompiler (*Barstools*) in order to make the generated Verilog more VLSI friendly. By default, the MacroCompiler will attempt to map memories into the SRAM options within the Hammer technology plugin. However, if you are working with a new process technology and prefer to work with flip-flop arrays, you can configure the MacroCompiler using the `MACROCOMPILER_MODE` make variable. For example, if your technology

plugin does not have an SRAM compiler ready, you can use the `MACROCOMPILER_MODE='--mode synflops'` option (Note that synthesizing a design with only flipflops is very slow and will often may not meet constraints).

We call the `make buildfile` command while also specifying the name of the process technology we are working with (same `tech_name` for the configuration files and plugin name) and the configuration files we created. Note, in the ASAP7 tutorial ((*ASAP7 Tutorial*)) these configuration files are merged into a single file called `example-asap7.yml`.

Hence, if we want to monolithically place and route the entire SoC, the relevant command would be

```
make buildfile CONFIG=<chipyard_config_name> tech_name=<tech_name> INPUT_CONFS=
↪"example-design.yml example-tools.yml example-tech.yml"
```

In a more typical scenario of working on a single module, for example the Gemmini accelerator within the GemminiRocketConfig Chipyard SoC configuration, the relevant command would be:

```
make buildfile CONFIG=GemminiRocketConfig VLSI_TOP=Gemmini tech_name=tsmintel3 INPUT_
↪CONFS="example-design.yml example-tools.yml example-tech.yml"
```

Running the VLSI Flow

Running a basic VLSI flow using the Hammer default configurations is fairly simple, and consists of simple `make` command with the previously mentioned Make variables.

Synthesis

In order to run synthesis, we run `make syn` with the matching Make variables. Post-synthesis logs and collateral will be saved in `build/<config-name>/syn-rundir`. The raw QoR data (area, timing, gate counts, etc.) will be found in `build/<config-name>/syn-rundir/reports`.

Hence, if we want to monolithically synthesize the entire SoC, the relevant command would be:

```
make syn CONFIG=<chipyard_config_name> tech_name=<tech_name> INPUT_CONFS="example-
↪design.yml example-tools.yml example-tech.yml"
```

In a more typical scenario of working on a single module, for example the Gemmini accelerator within the GemminiRocketConfig Chipyard SoC configuration, the relevant command would be:

```
make syn CONFIG=GemminiRocketConfig VLSI_TOP=Gemmini tech_name=tsmintel3 INPUT_CONFS=
↪"example-design.yml example-tools.yml example-tech.yml"
```

It is worth checking the final-qor.rpt report to make sure that the synthesized design meets timing before moving to the place-and-route step.

Place-and-Route

In order to run place-and-route, we run `make par` with the matching Make variables. Post-PnR logs and collateral will be saved in `build/<config-name>/par-rundir`. Specifically, the resulting GDSII file will be in that directory with the suffix `*.gds`. and timing reports can be found in `build/<config-name>/par-rundir/timingReports`. Place-and-route is requires more design details in contrast to synthesis. For example, place-and-route requires some basic floorplanning constraints. The default `example-design.yml` configuration file template allows the tool (specifically, the Cadence Innovus tool) to use it's automatic floorplanning capability within the top level of the design (`ChipTop`). However, if we choose to place-and-route a specific block which is not the SoC top level, we need to change the top-level path name to match the `VLSI_TOP` make parameter we are using.

Hence, if we want to monolitically place-and-route the entire SoC with the default tech plug-in parameters for power-straps and corners, the relevant command would be:

```
make par CONFIG=<chipyard_config_name> tech_name=<tech_name> INPUT_CONFS="example-
↪design.yml example-tools.yml example-tech.yml"
```

In a more typical scenario of working on a single module, for example the Gemmini accelerator within the GemminiRocketConfig Chipyard SoC configuration,

```
vlsi.inputs.placement_constraints:
  - path: "Gemmini"
    type: toplevel
    x: 0
    y: 0
    width: 300
    height: 300
    margins:
      left: 0
      right: 0
      top: 0
      bottom: 0
```

The relevant `make` command would then be:

```
make par CONFIG=GemminiRocketConfig VLSI_TOP=Gemmini tech_name=tsmintel3 INPUT_CONFS=
↪"example-design.yml example-tools.yml example-tech.yml"
```

Note that the width and height specification can vary widely between different modulesi and level of the module hierarchy. Make sure to set sane width and height values. Place-and-route generally requires more fine-grained input specifications regarding power nets, clock nets, pin assignments and floorplanning. While the template configuration files provide defaults for automatic tool defaults, these will usually result in very bad QoR, and therefore it is recommended to specify better-informed floorplans, pin assignments and power nets. For more information about cutomizing theses parameters, please refer to the *Customizing Your VLSI Flow in Hammer* sections or to the Hammer documentation. Additionally, some Hammer process technology plugins do not provide default values for required settings such as tool paths and pin assignments (for example, ASAP7). In those cases, these constraints will need to be specified manually in the top-level configuration yml files, as is the case in the `example-asap7.yml` configuration file.

Place-and-route tools are very sensitive to process technologes (significantly more sensitive than synthesis tools), and different process technologies may work only on specific tool versions. It is recommended to check what is the appropriate tool version for the specific process technology you are working with.

Note: If you edit the yml configuration files in between synthesis and place-and-route, the `make par` command will automatically re-run synthesis. If you would like to avoid that and are confident that your configuration file changes do not affect synthesis results, you may use the `make redo-par` command instead with the variable `HAMMER_EXTRA_ARGS='-p <your-changed.yml>'`.

Power Estimation

Power estimation in Hammer can be performed in one of two stages: post-synthesis (post-syn) or post-place-and-route (post-par). The most accurate power estimation is post-par, and it includes finer grained details of the places instances and wire lengths. Post-par power estimation can be based on static average signal toggles rates (also known as "static power estimation"), or based on simulation-extracted signal toggle data (also known as "dynamic power estimation").

Warning: In order to run post-par power estimation, make sure that a power estimation tool (such as Cadence Voltus) has been defined in your `example-tools.yml` file. Make sure that the power estimation tool (for example, Cadence Voltus) version matches the physical design tool (for example, Cadence Innovus) version, otherwise you will encounter a database mismatch error.

Simulation-exacted power estimation often requires a dedicated testharness for the block under evalution (DUT). While the Hammer flow supports such configurations (further details can be found in the Hammer documentation), Chipyard's integrated flows support an automated full digital SoC simulation-extracted post-par power estimation through the integration of software RTL simulation flows with the Hammer VLSI flow. As such, full digital SoC simulation-extracted power estimation can be performed by specifying a simple binary executable with the associated `make` command.

```
make power-par BINARY=/path/to/baremetal/binary/rv64ui-p-addi.riscv CONFIG=<chipyard_
↪config_name> tech_name=tsmintel3 INPUT_CONFS="example-design.yml example-tools.yml␣
↪example-tech.yml"
```

The simulation-extracted power estimation flow implicitly uses Hammer's gate-level simulation flow (in order to generate the `saif` activity data file). This gate-level simulation flow can also be run independantly from the power estimation flow using the `make sim-par` command.

Note: The gate-level simulation flow (and there the simulation-extracted power-estimation) is currently integrated only with the Synopsys VCS simulation (Verilator does not support gate-level simulation. Support for Cadence Incisive is work-in-progress)

Signoff

During chip tapeout, you will need to perform sign-off check to make sure the generated GDSII can be fabricated as intended. This is done using dedicated signoff tools that perform design rule checking (DRC) and layout versus schematic (LVS) verification. In most cases, placed-and-routed designs will not pass DRC and LVS on first attempts due to nuanced design rules and subtle/silent failures of the place-and-route tools. Passing DRC and LVS will often requires adding manual placement constraints to "force" the EDA tools into certain patterns. If you have placed-and-routed a design with the goal of getting area and power estimates, DRC and LVS are not strictly neccessary and the results will likely be quite similar. If you are intending to tapeout and fabricate a chip, DRC and LVS are mandatory and will likely requires multiple-iterations of refining manual placement constraints. Having a large number of DRC/LVS violations can have a significant impact on the runtime of the place-and-route procedure (since the tools will try to fix each of them several times). A large number of DRC/LVS violations may also be an indication that the design is not necessarily realistic for this particular process technology, which may have power/area implications.

Since signoff checks are required only for a complete chip tapeout, they are currently not fully automated in Hammer, and often require some additional manual inclusion of custom Makefiles associated with specific process technologies. However, the general steps from running signoff within Hammer (under the assumption of a fully automated tech plug-in) are Make commands similar to the previous steps.

In order to run DRC, the relevant `make` command is `make drc`. As in the previous stages, the make command should be accompanied by the relevant configuration Make variables:

```
make drc CONFIG=GemminiRocketConfig VLSI_TOP=Gemmini tech_name=tsmintel3 INPUT_CONFS=
↪"example-design.yml example-tools.yml example-tech.yml"
```

DRC does not emit easily audited reports, as the rule names violated can be quite esoteric. It is often more productive to rather use the scripts generated by Hammer to open the DRC error database within the appropriate tool. These

generated scripts can be called from `./build/<config-name>/drc-rundir/generated-scripts/view_drc`.

In order to run LVS, the relevant `make` command is `make lvs`. As in the previous stages, the make command should be accompanied by the relevant configuration Make variables:

```
make lvs CONFIG=GemminiRocketConfig VLSI_TOP=Gemmini tech_name=tsmintel3 INPUT_CONFS=
↪"example-design.yml example-tools.yml example-tech.yml"
```

LVS does not emit easily audited reports, as the violations are often cryptic when seen textually. As a result it is often more productive to visually see the LVS issues using the generated scripts that enable opening the LVS error database within the appropriate tool. These generated scripts can be called from `./build/<config-name>/lvs-rundir/generated-scripts/view_lvs`.

Customizing Your VLSI Flow in Hammer

Advanced Environment Setup

If you have access to a shared LSF cluster and you would like Hammer to submit it's compute-intensive jobs to the LSF cluster rather than your login machine, you can add the following code segment to your `env.yml` file (completing the relevant values for the bsub binary path, the number of CPUs requested, and the requested LSF queue):

```
#submit command (use LSF)
vlsi.submit:
    command: "lsf"
    settings: [{"lsf": {
        "bsub_binary": "</path/to/bsub/binary/bsub>",
        "num_cpus": <N>,
        "queue": "<lsf_queu>",
        "extra_args": ["-R", "span[hosts=1]"]
        }
    }]
    settings_meta: "append"
```

Composing a Hierarchical Design

For large designs, a monolithic VLSI flow may take the EDA tools a very long time to process and optimize, to the extent that it may not be feasable sometimes. Hammer supports a hierarchical physical design flow, which decomposes the design into several specified sub-components and runs the flow on each sub-components separetly. Hammer is then able to assemble these blocks together into a top-level design. This hierarchical approach speeds up the VLSI flow for large designs, especially designs in which there may me multiple instantiations of the same sub-components(since the sub-component can simply be replicated in the layout). While hierarchical physical design can be performed in multiple ways (top-down, bottom-up, abutment etc.), Hammer currently supports only the bottom-up approach. The bottom-up approach traverses a tree representing the hierarchy starting from the leaves and towards the direction of the root (the "top level"), and runs the physical design flow on each node of the hierarchy tree using the previously layed-out children nodes. As nodes get closer to the root (or "top level") of the hierarchy, largers sections of the design get layed-out.

The Hammer hierarchical flow relies on a manually-specified descrition of the desired heirarchy tree. The specification of the heirarchy tree is defined based on the instance names in the generated Verilog, which sometime make this specification challenging due to inconsisent instance names. Additionally, the specification of the heirarchy tree is intertwined with the manual specification of a floorplan for the design.

For example, if we choose to specifiy the previously mentioned `GemminiRocketConfig` configuration in a hierarchical fashion in which the Gemmini accelerator and the last-level cache are run separetly from the top-level SoC,

we would replace the floorplan example in `example-design.yml` from the *Place-and-Route* section with the following specification:

```
vlsi.inputs.hiearchical.top_module: "ChipTop"
vlsi.inputs.hierarchical.mode: manual"
vlsi.inputs.manual_modules:
  - ChipTop:
    - RocketTile
    - InclusiveCache
  - RocketTile:
    - Gemmini
vlsi.manual_placement_constraints:
  - ChipTop
    - path: "ChipTop"
      type: toplevel
      x: 0
      y: 0
      width: 500
      height: 500
      margins:
        left: 0
        right: 0
        top: 0
        bottom: 0
  - RocketTile
    - path: "chiptop.system.tile_prci_domain.tile"
      type: hierarchical
      master: ChipTop
      x: 0
      y: 0
      width: 250
      height: 250
      margins:
        left: 0
        right: 0
        top: 0
        bottom: 0
  - Gemmini
    - path: "chiptop.system.tile_prci_domain.tile.gemmini"
      type: hierarchical
      master: RocketTile
      x: 0
      y: 0
      width: 200
      height: 200
      margins:
        left: 0
        right: 0
        top: 0
        bottom: 0
  - InclusiveCache
    - path: "chiptop.system.subsystem_l2_wrapper.l2"
      type: hierarchical
      master: ChipTop
      x: 0
      y: 0
      width: 100
      height: 100
```

(continues on next page)

(continued from previous page)

```
    margins:
      left: 0
      right: 0
      top: 0
      bottom: 0
```

In this specification, `vlsi.inputs.hierarchical.mode` indicates the manual specification of the heirarchy tree (which is the only mode currently supported by Hammer), `vlsi.inputs.hiearchical.top_module` sets the root of the hierarchical tree, `vlsi.inputs.hierarchical.manual_modules` enumerates the tree of hierarchical modules, and `vlsi.inputs.hierarchical.manual_placement_constraints` enumerates the floorplan for each module.

Customizing Generated Tcl Scripts

The `example-vlsi` python script is the Hammer entry script with placeholders for hooks. Hooks are additional snippets of python and TCL (via `x.append()`) to extend the Hammer APIs. Hooks can be inserted using the `make_pre/post/replacement_hook` methods as shown in the `example-vlsi` entry script example. In this particular example, a list of hooks is passed in the `get_extra_par_hooks` function in the `ExampleDriver` class. Refer to the Hammer documentation on hooks for a detailed description of how these are injected into the VLSI flow.

2.5.7 ASAP7 Tutorial

The `vlsi` folder of this repository contains an example Hammer flow with the SHA-3 accelerator and a dummy hard macro. This example tutorial uses the built-in ASAP7 technology plugin and requires access to the included Cadence and Mentor tool plugin submodules. Cadence is necessary for synthesis & place-and-route, while Mentor is needed for DRC & LVS.

Project Structure

This example gives a suggested file structure and build system. The `vlsi/` folder will eventually contain the following files and folders:

- `Makefile`, `sim.mk`, `power.mk`
  - Integration of Hammer's build system into Chipyard and abstracts away some Hammer commands.
- `build`
  - Hammer output directory. Can be changed with the `OBJ_DIR` variable.
  - Will contain subdirectories such as `syn-rundir` and `par-rundir` and the `inputs.yml` denoting the top module and input Verilog files.
- `env.yml`
  - A template file for tool environment configuration. Fill in the install and license server paths for your environment.
- `example-vlsi`
  - Entry point to Hammer. Contains example placeholders for hooks.
- `example-asap7.yml`, `example-tools.yml`
  - Hammer IR for this tutorial.

- `example-design.yml`, `example-nangate45.yml`, `example-tech.yml`
  - Hammer IR not used for this tutorial but provided as templates.
- `generated-src`
  - All of the elaborated Chisel and FIRRTL.
- `hammer`, `hammer-<vendor>-plugins`, `hammer-<tech>-plugin`
  - Core, tool, tech repositories.
- `view_gds.py`
  - A convenience script to view a layout using gdstk or gdspy. Only use this for small layouts (i.e. smaller than the TinyRocketConfig example) since the gdstk-produced SVG will be too big and gdspy's GUI is very slow for large layouts!

Prerequisites

- Python 3.4+
- numpy and gdstk or gdspy packages. Note: gdspy must be version 1.4.
- Genus, Innovus, Voltus, VCS, and Calibre licenses
- For ASAP7 specifically (README for more details):
  - First, download the ASAP7 v1p7 PDK (we recommend shallow-cloning or downloading an archive of the repository). Then, download the encrypted Calibre decks tarball tarball to a directory of choice (e.g. the root directory of the PDK) but do not extract it like the instructions say. The tech plugin is configured to extract the tarball into a cache directory for you.
  - If you have additional ASAP7 hard macros, their LEF & GDS need to be 4x upscaled @ 4000 DBU precision.

Initial Setup

In the Chipyard root, run:

```
./scripts/init-vlsi.sh asap7
```

to pull the Hammer & plugin submodules. Note that for technologies other than `sky130` or `asap7`, the tech submodule must be added in the `vlsi` folder first.

Pull the Hammer environment into the shell:

```
cd vlsi
export HAMMER_HOME=$PWD/hammer
source $HAMMER_HOME/sourceme.sh
```

Building the Design

To elaborate the `TinyRocketConfig` and set up all prerequisites for the build system to push the design and SRAM macros through the flow:

```
make buildfile CONFIG=TinyRocketConfig
```

The CONFIG=TinyRocketConfig selects the target generator config in the same manner as the rest of the Chipyard framework. This elaborates a stripped-down Rocket Chip in the interest of minimizing tool runtime.

For the curious, make buildfile generates a set of Make targets in build/hammer.d. It needs to be re-run if environment variables are changed. It is recommended that you edit these variables directly in the Makefile rather than exporting them to your shell environment.

Running the VLSI Flow

example-vlsi

This is the entry script with placeholders for hooks. In the ExampleDriver class, a list of hooks is passed in the get_extra_par_hooks. Hooks are additional snippets of python and TCL (via x.append()) to extend the Hammer APIs. Hooks can be inserted using the make_pre/post/replacement_hook methods as shown in this example. Refer to the Hammer documentation on hooks for a detailed description of how these are injected into the VLSI flow.

example-asap7.yml

This contains the Hammer configuration for this example project. Example clock constraints, power straps definitions, placement constraints, and pin constraints are given. Additional configuration for the extra libraries and tools are at the bottom.

First, set technology.asap7.tarball_dir to the absolute path to the directory where the downloaded the ASAP7 Calibre deck tarball lives. If it is not in the PDK's root directory, then also set technology.asap7.pdk_install_dir and technology.asap7.stdcell_install_dir.

Synthesis

```
make syn CONFIG=TinyRocketConfig
```

Post-synthesis logs and collateral are in build/syn-rundir. The raw quality of results data is available at build/syn-rundir/reports, and methods to extract this information for design space exploration are a work in progress.

Place-and-Route

```
make par CONFIG=TinyRocketConfig
```

After completion, the final database can be opened in an interactive Innovus session via ./build/par-rundir/generated-scripts/open_chip.

Intermediate database are written in build/par-rundir between each step of the par action, and can be restored in an interactive Innovus session as desired for debugging purposes.

Timing reports are found in build/par-rundir/timingReports. They are gzipped text files.

gdspy can be used to view the final layout, but it is somewhat crude and slow (wait a few minutes for it to load):

```
./view_gds.py build/chipyard.TestHarness.TinyRocketConfig/par-rundir/ChipTop.gds
```

By default, this script only shows the M2 thru M4 routing. Layers can be toggled in the layout viewer's side pane and view_gds.py has a mapping of layer numbers to layer names.

DRC & LVS

To run DRC & LVS, and view the results in Calibre:

```
make drc CONFIG=TinyRocketConfig
./build/drc-rundir/generated-scripts/view-drc
make lvs CONFIG=TinyRocketConfig
./build/lvs-rundir/generated-scripts/view-lvs
```

Some DRC errors are expected from this PDK, as explained in the ASAP7 plugin readme. Furthermore, the dummy SRAMs that are provided in this tutorial and PDK do not have any geometry inside, so will certainly cause DRC errors.

Simulation

Simulation with VCS is supported, and can be run at the RTL- or gate-level (post-synthesis and post-P&R). The simulation infrastructure as included here is intended for running RISC-V binaries on a Chipyard config. For example, for an RTL-level simulation:

```
make sim-rtl CONFIG=TinyRocketConfig BINARY=$RISCV/riscv64-unknown-elf/share/riscv-
↪tests/isa/rv32ui-p-simple
```

Post-synthesis and post-P&R simulations use the `sim-syn` and `sim-par` make targets, respectively.

Appending `-debug` and `-debug-timing` to these make targets will instruct VCS to write a SAIF + VPD (or FSDB if the `USE_FSDB` flag is set) and do timing-annotated simulations, respectively. See the `sim.mk` file for all available targets.

Power/Rail Analysis

Post-P&R power and rail (IR drop) analysis is supported with Voltus:

```
make power-par CONFIG=TinyRocketConfig
```

If you append the `BINARY` variable to the command, it will use the activity file generated from a `sim-<syn/par>-debug` run and report dynamic power & IR drop from the toggles encoded in the waveform.

To bypass gate-level simulation, you will need to run the power tool manually (see the generated commands in the generated `hammer.d` buildfile). Static and active (vectorless) power & IR drop will be reported.

2.5.8 Sky130 Tutorial

The `vlsi` folder of this repository contains an example Hammer flow with the SHA-3 accelerator and a dummy hard macro. This example tutorial uses the built-in Sky130 technology plugin and requires access to the included Cadence and Mentor tool plugin submodules. Cadence is necessary for synthesis & place-and-route, while Mentor is needed for DRC & LVS.

Project Structure

This example gives a suggested file structure and build system. The `vlsi/` folder will eventually contain the following files and folders:

- `Makefile`, `sim.mk`, `power.mk`

  - Integration of Hammer's build system into Chipyard and abstracts away some Hammer commands.

- `build`
  - Hammer output directory. Can be changed with the `OBJ_DIR` variable.
  - Will contain subdirectories such as `syn-rundir` and `par-rundir` and the `inputs.yml` denoting the top module and input Verilog files.
- `env.yml`
  - A template file for tool environment configuration. Fill in the install and license server paths for your environment.
- `example-vlsi-sky130`
  - Entry point to Hammer. Contains example placeholders for hooks.
- `example-sky130.yml`, `example-tools.yml`
  - Hammer IR for this tutorial.
- `example-design.yml`, `example-nangate45.yml`, `example-tech.yml`
  - Hammer IR not used for this tutorial but provided as templates.
- `generated-src`
  - All of the elaborated Chisel and FIRRTL.
- `hammer`, `hammer-<vendor>-plugins`, `hammer-<tech>-plugin`
  - Core, tool, tech repositories.

Prerequisites

- Python 3.4+
- numpy package
- Genus, Innovus, Voltus, VCS, and Calibre licenses
- Sky130 PDK, install using these directions

Initial Setup

In the Chipyard root, run:

```
./scripts/init-vlsi.sh sky130
```

to pull the Hammer & plugin submodules. Note that for technologies other than `sky130` or `asap7`, the tech submodule must be added in the `vlsi` folder first.

Pull the Hammer environment into the shell:

```
cd vlsi
export HAMMER_HOME=$PWD/hammer
source $HAMMER_HOME/sourceme.sh
```

Building the Design

To elaborate the `TinyRocketConfig` and set up all prerequisites for the build system to push the design and SRAM macros through the flow:

```
make buildfile tech_name=sky130 CONFIG=TinyRocketConfig
```

The `CONFIG=TinyRocketConfig` selects the target generator config in the same manner as the rest of the Chipyard framework. This elaborates a stripped-down Rocket Chip in the interest of minimizing tool runtime.

For the curious, `make buildfile` generates a set of Make targets in `build/hammer.d`. It needs to be re-run if environment variables are changed. It is recommended that you edit these variables directly in the Makefile rather than exporting them to your shell environment.

Running the VLSI Flow

example-vlsi-sky130

This is the entry script with placeholders for hooks. In the `ExampleDriver` class, a list of hooks is passed in the `get_extra_par_hooks`. Hooks are additional snippets of python and TCL (via `x.append()`) to extend the Hammer APIs. Hooks can be inserted using the `make_pre/post/replacement_hook` methods as shown in this example. Refer to the Hammer documentation on hooks for a detailed description of how these are injected into the VLSI flow.

example-sky130.yml

This contains the Hammer configuration for this example project. Example clock constraints, power straps definitions, placement constraints, and pin constraints are given. Additional configuration for the extra libraries and tools are at the bottom.

First, set `technology.sky130.sky130A/sky130_nda/openram_lib` to the absolute path of the respective directories containing the Sky130 PDK and SRAM files. See the Sky130 Hammer plugin README for details about the PDK setup.

Synthesis

```
make syn tech_name=sky130 CONFIG=TinyRocketConfig
```

Post-synthesis logs and collateral are in `build/syn-rundir`. The raw quality of results data is available at `build/syn-rundir/reports`, and methods to extract this information for design space exploration are a work in progress.

Place-and-Route

```
make par tech_name=sky130 CONFIG=TinyRocketConfig
```

After completion, the final database can be opened in an interactive Innovus session via `./build/par-rundir/generated-scripts/open_chip`.

Intermediate database are written in `build/par-rundir` between each step of the `par` action, and can be restored in an interactive Innovus session as desired for debugging purposes.

Timing reports are found in `build/par-rundir/timingReports`. They are gzipped text files.

DRC & LVS

To run DRC & LVS, and view the results in Calibre:

```
make drc tech_name=sky130 CONFIG=TinyRocketConfig
./build/chipyard.TestHarness.TinyRocketConfig-ChipTop/drc-rundir/generated-scripts/
↪view_drc
make lvs tech_name=sky130 CONFIG=TinyRocketConfig
./build/chipyard.TestHarness.TinyRocketConfig-ChipTop/lvs-rundir/generated-scripts/
↪view_lvs
```

Some DRC errors are expected from this PDK, especially with regards to the SRAMs, as explained in the Sky130 Hammer plugin README. For this reason, the `example-vlsi-sky130` script black-boxes the SRAMs for DRC/LVS analysis.

Simulation

Simulation with VCS is supported, and can be run at the RTL- or gate-level (post-synthesis and post-P&R). The simulation infrastructure as included here is intended for running RISC-V binaries on a Chipyard config. For example, for an RTL-level simulation:

```
make sim-rtl CONFIG=TinyRocketConfig BINARY=$RISCV/riscv64-unknown-elf/share/riscv-
↪tests/isa/rv64ui-p-simple
```

Post-synthesis and post-P&R simulations use the `sim-syn` and `sim-par` make targets, respectively.

Appending `-debug` and `-debug-timing` to these make targets will instruct VCS to write a SAIF + VPD (or FSDB if the `USE_FSDB` flag is set) and do timing-annotated simulations, respectively. See the `sim.mk` file for all available targets.

Power/Rail Analysis

Post-P&R power and rail (IR drop) analysis is supported with Voltus:

```
make power-par tech_name=sky130 CONFIG=TinyRocketConfig
```

If you append the `BINARY` variable to the command, it will use the activity file generated from a `sim-<syn/par>-debug` run and report dynamic power & IR drop from the toggles encoded in the waveform.

To bypass gate-level simulation, you will need to run the power tool manually (see the generated commands in the generated `hammer.d` buildfile). Static and active (vectorless) power & IR drop will be reported.

2.5.9 Advanced Usage

Alternative RTL Flows

The Make-based build system provided supports using Hammer without using RTL generated by Chipyard. To push a custom Verilog module through, one only needs to append the following environment variables to the `make buildfile` command (or edit them directly in the Makefile).

```
CUSTOM_VLOG=<your Verilog files>
VLSI_TOP=<your top module>
```

`CUSTOM_VLOG` breaks the dependency on the rest of the Chipyard infrastructure and does not start any Chisel/FIRRTL elaboration. `VLSI_TOP` selects the top module from your custom Verilog files.

Under the Hood

To uncover what is happening under the hood, here are the commands that are executed:

For `make syn`:

```
./example-vlsi -e /path/to/env.yml -p /path/to/example.yml -p /path/to/inputs.yml --
↪obj_dir /path/to/build syn
```

`example-vlsi` is the entry script as explained before, `-e` provides the environment yml, `-p` points to configuration yml/jsons, `--obj_dir` speficies the destination directory, and `syn` is the action.

For `make par`:

```
./example-vlsi -e /path/to/env.yml -p /path/to/syn-output-full.json -o /path/to/par-
↪input.json --obj_dir /path/to/build syn-to-par
./example-vlsi -e /path/to/env.yml -p /path/to/par-input.json --obj_dir /path/to/
↪build par
```

A `syn-to-par` action translates the synthesis output configuration into an input configuration given by `-o`. Then, this is passed to the `par` action.

For more information about all the options that can be passed to the Hammer command-line driver, please see the Hammer documentation.

Manual Step Execution & Dependency Tracking

It is invariably necessary to debug certain steps of the flow, e.g. if the power strap settings need to be updated. The underlying Hammer commands support options such as `--to_step`, `--from_step`, and `--only_step`. These allow you to control which steps of a particular action are executed.

Make's dependency tracking can sometimes result in re-starting the entire flow when the user only wants to re-run a certain action. Hammer's build system has "redo" targets such as `redo-syn` and `redo-par` to run certain actions without typing out the entire Hammer command.

Say you need to update some power straps settings in `example.yml` and want to try out the new settings:

```
make redo-par HAMMER_REDO_ARGS='-p example.yml --only_step power_straps'
```

Hierarchical RTL/Gate-level Simulation, Power Estimation

With the Synopsys plugin, hierarchical RTL and gate-level simulation is supported using VCS at the chip-level. Also, post-par power estimation with Voltus in the Cadence plugin is also supported. Special Make targets are provided in the `vlsi/` directory in `sims.mk` and `power.mk`. Here is a brief description:

- `sim-rtl`: RTL-level simulation
  - `sim-rtl-debug`: Also write a VPD waveform
- `sim-syn`: Post-synthesis gate-level simulation

  - `sim-syn-debug`: Also write a VPD waveform
  - `sim-syn-timing-debug`: Timing-annotated with VPD waveform
- `sim-par`: Post-par gate-level simulation
  - `sim-par-debug`: Also write a VPD waveform
  - `sim-par-timing-debug`: Timing-annotated with VPD waveform
- `power-par`: Post-par power estimation
  - Note: this will run `sim-par` first
- `redo-` can be appended to all above targets to break dependency tracking, like described above.
- `-$(VLSI_TOP)` suffixes denote simulations/power analysis on a submodule in a hierarchical flow. Note that you must provide the testbenches for these modules since the default testbench only simulates a Chipyard-based `ChipTop` DUT instance.

The simulation configuration (e.g. binaries) can be edited for your design. See the `Makefile` and refer to Hammer's documentation for how to set up simulation parameters for your design.

2.6 Customization

These guides will walk you through customization of your system-on-chip:

- Contructing heterogenous systems-on-chip using the existing Chipyard generators and configuration system.
- How to include your custom Chisel sources in the Chipyard build system
- Adding custom core
- Adding custom RoCC accelerators to an existing Chipyard core (BOOM or Rocket)
- Adding custom MMIO widgets to the Chipyard memory system by Tilelink or AXI4, with custom Top-level IOs
- Adding custom Dsptools based blocks as MMIO widgets.
- Standard practices for using Keys, Traits, and Configs to parameterize your design
- Customizing the memory hierarchy
- Connect widgets which act as TileLink masters
- Adding custom blackboxed Verilog to a Chipyard design

We also provide information on:

- The boot process for Chipyard SoCs
- Examples of FIRRTL transforms used in Chipyard, and where they are specified

We recommend reading all these pages in order. Hit next to get started!

2.6.1 Heterogeneous SoCs

The Chipyard framework involves multiple cores and accelerators that can be composed in arbitrary ways. This discussion will focus on how you combine Rocket, BOOM and Hwacha in particular ways to create a unique SoC.

Creating a Rocket and BOOM System

Instantiating an SoC with Rocket and BOOM cores is all done with the configuration system and two specific config fragments. Both BOOM and Rocket have config fragments labelled `WithN{Small|Medium|Large|etc.}BoomCores(X)` and `WithNBigCores(X)` that automatically create `X` copies of the core/tile[1]. When used together you can create a heterogeneous system.

The following example shows a dual core BOOM with a single core Rocket.

```
class DualLargeBoomAndSingleRocketConfig extends Config(
  new boom.common.WithNLargeBooms(2) ++                    // add 2 boom cores
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++   // add 1 rocket core
  new chipyard.config.AbstractConfig)
```

Adding Hwachas

Adding a Hwacha accelerator is as easy as adding the `DefaultHwachaConfig` so that it can setup the Hwacha parameters and add itself to the `BuildRoCC` parameter. An example of adding a Hwacha to all tiles in the system is below.

```
class HwachaLargeBoomAndHwachaRocketConfig extends Config(
  new chipyard.config.WithHwachaTest ++
  new hwacha.DefaultHwachaConfig ++                        // add hwacha to all␣
↪harts
  new boom.common.WithNLargeBooms(1) ++                    // add 1 boom core
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++   // add 1 rocket core
  new chipyard.config.AbstractConfig)
```

In this example, Hwachas are added to both BOOM tiles and to the Rocket tile. All with the same Hwacha parameters.

Assigning Accelerators to Specific Tiles with MultiRoCC

Located in `generators/chipyard/src/main/scala/config/fragments/RoCCFragments.scala` is a config fragment that provides support for adding RoCC accelerators to specific tiles in your SoC. Named `MultiRoCCKey`, this key allows you to attach RoCC accelerators based on the `hartId` of the tile. For example, using this allows you to create a 8 tile system with a RoCC accelerator on only a subset of the tiles. An example is shown below with two BOOM cores, and one Rocket tile with a RoCC accelerator (Hwacha) attached.

```
class DualLargeBoomAndHwachaRocketConfig extends Config(
  new chipyard.config.WithMultiRoCC ++                                  // support␣
↪heterogeneous rocc
  new chipyard.config.WithMultiRoCCHwacha(0) ++                         // put hwacha␣
↪on hart-0 (rocket)
  new hwacha.DefaultHwachaConfig ++                                     // set␣
↪default hwacha config keys
  new boom.common.WithNLargeBooms(2) ++                                 // add 2 boom␣
↪cores
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++                // add 1␣
↪rocket core
  new chipyard.config.AbstractConfig)
```

[1] Note, in this section "core" and "tile" are used interchangeably but there is subtle distinction between a "core" and "tile" ("tile" contains a "core", L1D/I$, PTW). For many places in the documentation, we usually use "core" to mean "tile" (doesn't make a large difference but worth the mention).

The `WithMultiRoCCHwacha` config fragment assigns a Hwacha accelerator to a particular `hartId` (in this case, the `hartId` of `0` corresponds to the Rocket core). Finally, the `WithMultiRoCC` config fragment is called. This config fragment sets the `BuildRoCC` key to use the `MultiRoCCKey` instead of the default. This must be used after all the RoCC parameters are set because it needs to override the `BuildRoCC` parameter. If this is used earlier in the configuration sequence, then MultiRoCC does not work.

This config fragment can be changed to put more accelerators on more cores by changing the arguments to cover more `hartId`'s (i.e. `WithMultiRoCCHwacha(0,1,3,6,...)`).

Since config fragments are applied from right-to-left (or bottom-to-top as they are formatted here), the right-most config fragment specifying a core (which is `freechips.rocketchip.subsystem.WithNBigCores` in the example above) gets the first hart ID. Consider this config:

```
class RocketThenBoomHartIdTestConfig extends Config(
  new boom.common.WithNLargeBooms(2) ++
  new freechips.rocketchip.subsystem.WithNBigCores(3) ++
  new chipyard.config.AbstractConfig)
```

This specifies an SoC with three Rocket cores and two BOOM cores. The Rocket cores would have hart IDs 0, 1, and 2, while the BOOM cores would have hard IDs 3 and 4. On the other hand, consider this config which reverses the order of those two fragments:

```
class BoomThenRocketHartIdTestConfig extends Config(
  new freechips.rocketchip.subsystem.WithNBigCores(3) ++
  new boom.common.WithNLargeBooms(2) ++
  new chipyard.config.AbstractConfig)
```

This also specifies an SoC with three Rocket cores and two BOOM cores, but because the BOOM config fragment is evaluated before the Rocket config fragment, the hart IDs are reversed. The BOOM cores would have hart IDs 0 and 1, while the Rocket cores would have hard IDs 2, 3, and 4.

2.6.2 Integrating Custom Chisel Projects into the Generator Build System

Warning: This section assumes integration of custom Chisel through git submodules. While it is possible to directly commit custom Chisel into the Chipyard framework, we heavily recommend managing custom code through git submodules. Using submodules decouples development of custom features from development on the Chipyard framework.

While developing, you want to include Chisel code in a submodule so that it can be shared by different projects. To add a submodule to the Chipyard framework, make sure that your project is organized as follows.

```
yourproject/
    build.sbt
    src/main/scala/
        YourFile.scala
```

Put this in a git repository and make it accessible. Then add it as a submodule to under the following directory hierarchy: `generators/yourproject`.

The `build.sbt` is a minimal file which describes metadata for a Chisel project. For a simple project, the `build.sbt` can even be empty, but below we provide an example build.sbt.

```
organization := "edu.berkeley.cs"
```

(continues on next page)

(continued from previous page)

```
version := "1.0"

name := "yourproject"

scalaVersion := "2.12.4"
```

```
cd generators/
git submodule add https://git-repository.com/yourproject.git
```

Then add `yourproject` to the Chipyard top-level build.sbt file.

```
lazy val yourproject = (project in file("generators/yourproject")).
↪settings(commonSettings).dependsOn(rocketchip)
```

You can then import the classes defined in the submodule in a new project if you add it as a dependency. For instance, if you want to use this code in the `chipyard` project, change the final line in build.sbt to the following.

```
lazy val chipyard = (project in file(".")).settings(commonSettings).
↪dependsOn(testchipip, yourproject)
```

2.6.3 Adding a custom core

You may want to integrate a custom RISC-V core into the Chipyard framework. This documentation page provides step-by-step instructions on how to achieve this.

Note: RoCC is currently not supported by cores other than Rocket and BOOM. Please use Rocket or BOOM as the RoCC base core if you need to use RoCC.

Note: This page contains links to the files that contains important definitions in the Rocket chip repository, which is maintained separately from Chipyard. If you find any discrepancy between the code on this page and the code in the source file, please report it through GitHub issues!

Wrap Verilog Module with Blackbox (Optional)

Since Chipyard uses Scala and Chisel, if the top module of your core is not in Chisel, you will first need to create a Verilog blackbox for it so that it can be processed by Chipyard. See *Incorporating Verilog Blocks* for instructions.

Create Parameter Case Classes

Chipyard will generate a core for every `InstantiableTileParams` object it discovered in the `TilesLocated(InSubsystem)` key. This object is derived from‘‘TileParams‘‘, a trait containing the information needed to create a tile. All cores must have their own implementation of `InstantiableTileParams`, as well as `CoreParams` which is passed as a field in `TileParams`.

`TileParams` holds the parameters for the tile, which include parameters for all components in the tile (e.g. core, cache, MMU, etc.), while `CoreParams` contains parameters specific to the core on the tile. They must be implemented as case classes with fields that can be overridden by other config fragments as the constructor parameters. See the appendix at the bottom of the page for a list of variable to be implemented. You can also add custom fields to them, but standard fields should always be preferred.

`InstantiableTileParams[TileType]` holds the constructor of `TileType` on top of the fields of `TileParams`, where `TileType` is the tile class (see the next section). All custom cores will also need to implement `instantiate()` in their tile parameter class to return a new instance of the tile class `TileType`.

`TileParams` (in the file BaseTile.scala) , `InstantiableTileParams` (in the file BaseTile.scala), `CoreParams` (in the file Core.scala), and `FPUParams` (in the file FPU.scala) contains the following fields:

```
trait TileParams {
  val core: CoreParams                    // Core parameters (see below)
  val icache: Option[ICacheParams]        // Rocket specific: I1 cache option
  val dcache: Option[DCacheParams]        // Rocket specific: D1 cache option
  val btb: Option[BTBParams]              // Rocket specific: BTB / branch predictor␣
↪option
  val hartId: Int                         // Hart ID: Must be unique within a design␣
↪config (This MUST be a case class parameter)
  val beuAddr: Option[BigInt]             // Rocket specific: Bus Error Unit for Rocket␣
↪Core
  val blockerCtrlAddr: Option[BigInt]     // Rocket specific: Bus Blocker for Rocket␣
↪Core
  val name: Option[String]                // Name of the core
}

abstract class InstantiableTileParams[TileType <: BaseTile] extends TileParams {
  def instantiate(crossing: TileCrossingParamsLike, lookup: LookupByHartIdImpl)
                 (implicit p: Parameters): TileType
}

trait CoreParams {
  val bootFreqHz: BigInt                // Frequency
  val useVM: Boolean                    // Support virtual memory
  val useUser: Boolean                  // Support user mode
  val useSupervisor: Boolean            // Support supervisor mode
  val useDebug: Boolean                 // Support RISC-V debug specs
  val useAtomics: Boolean               // Support A extension
  val useAtomicsOnlyForIO: Boolean      // Support A extension for memory-mapped IO␣
↪(may be true even if useAtomics is false)
  val useCompressed: Boolean            // Support C extension
  val useVector: Boolean = false        // Support V extension
  val useSCIE: Boolean                  // Support custom instructions (in custom-0 and␣
↪custom-1)
  val useRVE: Boolean                   // Use E base ISA
  val mulDiv: Option[MulDivParams]      // *Rocket specific: M extension related␣
↪setting (Use Some(MulDivParams()) to indicate M extension supported)
  val fpu: Option[FPUParams]            // F and D extensions and related setting (see␣
↪below)
  val fetchWidth: Int                   // Max # of insts fetched every cycle
  val decodeWidth: Int                  // Max # of insts decoded every cycle
  val retireWidth: Int                  // Max # of insts retired every cycle
  val instBits: Int                     // Instruction bits (if 32 bit and 64 bit are␣
↪both supported, use 64)
  val nLocalInterrupts: Int             // # of local interrupts (see SiFive interrupt␣
↪cookbook)
  val nPMPs: Int                        // # of Physical Memory Protection units
  val pmpGranularity: Int               // Size of the smallest unit of region for PMP␣
↪unit (must be power of 2)
  val nBreakpoints: Int                 // # of hardware breakpoints supported (in RISC-
↪V debug specs)
  val useBPWatch: Boolean               // Support hardware breakpoints
```

(continues on next page)

(continued from previous page)

```
  val nPerfCounters: Int              // # of supported performance counters
  val haveBasicCounters: Boolean      // Support basic counters defined in the RISC-V␣
↪counter extension
  val haveFSDirty: Boolean            // If true, the core will set FS field in␣
↪mstatus CSR to dirty when appropriate
  val misaWritable: Boolean           // Support writable misa CSR (like variable␣
↪instruction bits)
  val haveCFlush: Boolean             // Rocket specific: enables Rocket's custom␣
↪instruction extension to flush the cache
  val nL2TLBEntries: Int              // # of L2 TLB entries
  val mtvecInit: Option[BigInt]       // mtvec CSR (of V extension) initial value
  val mtvecWritable: Boolean          // If mtvec CSR is writable

  // Normally, you don't need to change these values (except lrscCycles)
  def customCSRs(implicit p: Parameters): CustomCSRs = new CustomCSRs

  def hasSupervisorMode: Boolean = useSupervisor || useVM
  def instBytes: Int = instBits / 8
  def fetchBytes: Int = fetchWidth * instBytes
  // Rocket specific: Longest possible latency of Rocket core D1 cache. Simply set it␣
↪to the default value 80 if you don't use it.
  def lrscCycles: Int

  def dcacheReqTagBits: Int = 6

  def minFLen: Int = 32
  def vLen: Int = 0
  def sLen: Int = 0
  def eLen(xLen: Int, fLen: Int): Int = xLen max fLen
  def vMemDataBits: Int = 0
}

case class FPUParams(
  minFLen: Int = 32,           // Minimum floating point length (no need to change)
  fLen: Int = 64,              // Maximum floating point length, use 32 if only single␣
↪precision is supported
  divSqrt: Boolean = true,     // Div/Sqrt operation supported
  sfmaLatency: Int = 3,        // Rocket specific: Fused multiply-add pipeline latency␣
↪(single precision)
  dfmaLatency: Int = 4         // Rocket specific: Fused multiply-add pipeline latency␣
↪(double precision)
)
```

Most of the fields here (marked "Rocket spcific") are originally designed for the Rocket core and thus contain some implementation-specific details, but many of them are general enough to be useful for other cores. You may ignore any fields marked "Rocket specific" and use their default values; however, if you need to store additional information with meaning or usage similar to these "Rocket specific" fields, it is recommended to use these fields instead of creating your own custom fields.

You will also need a `CanAttachTile` class to add the tile config into the config system, with the following format:

```
case class MyTileAttachParams(
  tileParams: MyTileParams,
  crossingParams: RocketCrossingParams
) extends CanAttachTile {
  type TileType = MyTile
  val lookup = PriorityMuxHartIdFromSeq(Seq(tileParams))
```

(continues on next page)

(continued from previous page)

```
}
```

During elaboration, Chipyard will look for subclasses of `CanAttachTile` in the config system and instantiate a tile from the parameters in this class for every such class it found.

Note: Implementations may choose to ignore some fields here or use them in a non-standard way, but using an inaccurate value may break Chipyard components that rely on them (e.g. an inaccurate indication of supported ISA extension will result in an incorrect test suite being generated) as well as any custom modules that use them. ALWAYS document any fields you ignore or with altered usage in your core implementation, and if you are implementing other devices that would look up these config values, also document them. "Rocket specific" values are generally safe to ignore, but you should document them if you use them.

Create Tile Class

In Chipyard, all Tiles are diplomatically instantiated. In the first phase, diplomatic nodes which specify Tile-to-System interconnects are evaluated, while in the second "Module Implementation" phase, hardware is elaborated. See *TileLink and Diplomacy Reference* for more details. In this step, you will need to implement a tile class for your core, which specifies the constraints on the core's parameters and the connections with other diplomatic nodes. This class usually contains Diplomacy/TileLink code only, and Chisel RTL code should not go here.

All tile classes implement `BaseTile` and will normally implement `SinksExternalInterrupts` and `SourcesExternalNotifications`, which allow the tile to accept external interrupt. A typical tile has the following form:

```
class MyTile(
  val myParams: MyTileParams,
  crossing: ClockCrossingType,
  lookup: LookupByHartIdImpl,
  q: Parameters)
  extends BaseTile(myParams, crossing, lookup, q)
  with SinksExternalInterrupts
  with SourcesExternalNotifications
{

  // Private constructor ensures altered LazyModule.p is used implicitly
  def this(params: MyTileParams, crossing: TileCrossingParamsLike, lookup:␣
↪LookupByHartIdImpl)(implicit p: Parameters) =
    this(params, crossing.crossingType, lookup, p)

  // Require TileLink nodes
  val intOutwardNode = IntIdentityNode()
  val masterNode = visibilityNode
  val slaveNode = TLIdentityNode()

  // Implementation class (See below)
  override lazy val module = new MyTileModuleImp(this)

  // Required entry of CPU device in the device tree for interrupt purpose
  val cpuDevice: SimpleDevice = new SimpleDevice("cpu", Seq("my-organization,my-cpu",
↪"riscv")) {
    override def parent = Some(ResourceAnchors.cpus)
    override def describe(resources: ResourceBindings): Description = {
      val Description(name, mapping) = super.describe(resources)
```

(continues on next page)

(continued from previous page)

```
      Description(name, mapping ++
                        cpuProperties ++
                        nextLevelCacheProperty ++
                        tileProperties)
    }
  }

  ResourceBinding {
    Resource(cpuDevice, "reg").bind(ResourceAddress(hartId))
  }

  // TODO: Create TileLink nodes and connections here.
```

Connect TileLink Buses

Chipyard uses TileLink as its onboard bus protocol. If your core doesn't use TileLink, you will need to insert converters between the core's memory protocol and TileLink within the Tile module. in the tile class. Below is an example of how to connect a core using AXI4 to the TileLink bus with converters provided by Rocket chip:

```
  (tlMasterXbar.node  // tlMasterXbar is the bus crossbar to be used when this core /␣
↪tile is acting as a master; otherwise, use tlSlaveXBar
    := memoryTap
    := TLBuffer()
    := TLFIFOFixer(TLFIFOFixer.all) // fix FIFO ordering
    := TLWidthWidget(masterPortBeatBytes) // reduce size of TL
    := AXI4ToTL() // convert to TL
    := AXI4UserYanker(Some(2)) // remove user field on AXI interface. need but in␣
↪reality user intf. not needed
    := AXI4Fragmenter() // deal with multi-beat xacts
    := memAXI4Node) // The custom node, see below
```

Remember, you may not need all of these intermediate widgets. See *Diplomatic Widgets* for the meaning of each intermediate widget. If you are using TileLink, then you only need the tap node and the TileLink node used by your components. Chipyard also provides converters for AHB, APB and AXIS, and most of the AXI4 widgets has equivalent widget for these bus protocol; see the source files in `generators/rocket-chip/src/main/scala/amba` for more info.

If you are using some other bus protocol, you may implement your own converters, using the files in `generators/rocket-chip/src/main/scala/amba` as the template, but it is not recommended unless you are familiar with TileLink.

`memAXI4Node` is an AXI4 master node and is defined as following in our example:

```
  // # of bits used in TileLink ID for master node. 4 bits can support 16 master␣
↪nodes, but you can have a longer ID if you need more.
  val idBits = 4
  val memAXI4Node = AXI4MasterNode(
    Seq(AXI4MasterPortParameters(
      masters = Seq(AXI4MasterParameters(
        name = "myPortName",
        id = IdRange(0, 1 << idBits))))))
  val memoryTap = TLIdentityNode() // Every bus connection should have their own tap␣
↪node
```

where `portName` and `idBits` (number of bits to represent a port ID) are the parameter provides by the tile. Make sure to read *TileLink Node Types* to check out what type of nodes Chipyard supports and their parameters!

Also, by default, there are boundary buffers for both master and slave connections to the bus when they are leaving the tile, and you can override the following two functions to control how to buffer the bus requests/responses: (You can find the definition of these two functions in the class `BaseTile` in the file BaseTile.scala)

```
// By default, their value is "TLBuffer(BufferParams.none)".
protected def makeMasterBoundaryBuffers(implicit p: Parameters): TLBuffer
protected def makeSlaveBoundaryBuffers(implicit p: Parameters): TLBuffer
```

You can find more information on `TLBuffer` in *Diplomatic Widgets*.

Create Implementation Class

The implementation class contains the parameterized, actual hardware that depends on the values resolved by the Diplomacy framework according to the info provided in the Tile class. This class will normally contains Chisel RTL code. If your core is in Verilog, you will need to instantiate the black box class that wraps your Verilog implementation and connect it with the buses and other components. No Diplomacy/TileLink code should be in this class; you should only connect the IO signals in TileLink interfaces or other diplomatically defined components, which are located in the tile class.

The implementation class for your core is of the following form:

```
class MyTileModuleImp(outer: MyTile) extends BaseTileModuleImp(outer){
  // annotate the parameters
  Annotated.params(this, outer.myParams)

  // TODO: Create the top module of the core and connect it with the ports in "outer"

  // If your core is in Verilog (assume your blackbox is called "MyCoreBlackbox"),␣
↪instantiate it here like
  //   val core = Module(new MyCoreBlackbox(params...))
  // (as described in the blackbox tutorial) and connect appropriate signals. See the␣
↪blackbox tutorial
  // (link on the top of the page) for more info.
  // You can look at https://github.com/ucb-bar/cva6-wrapper/blob/master/src/main/
↪scala/CVA6Tile.scala
  // for a Verilog example.

  // If your core is in Chisel, you can simply instantiate the top module here like␣
↪other Chisel module
  // and connect appropriate signal. You can even implement this class as your top␣
↪module.
  // See https://github.com/riscv-boom/riscv-boom/blob/master/src/main/scala/common/
↪tile.scala and
  // https://github.com/chipsalliance/rocket-chip/blob/master/src/main/scala/tile/
↪RocketTile.scala for
  // Chisel example.
```

If you create an AXI4 node (or equivalents), you will need to connect them to your core. You can connect a port like this:

```
  outer.memAXI4Node.out foreach { case (out, edgeOut) =>
    // Connect your module IO port to "out"
    // The type of "out" here is AXI4Bundle, which is defined in generators/rocket-
↪chip/src/main/scala/amba/axi4/Bundles.scala
    // Please refer to this file for the definition of the ports.
    // If you are using APB, check APBBundle in generators/rocket-chip/src/main/scala/
↪amba/apb/Bundles.scala
```

(continues on next page)

(continued from previous page)

```
    // If you are using AHB, check AHBSlaveBundle or AHBMasterBundle in generators/
↪rocket-chip/src/main/scala/amba/ahb/Bundles.scala
    // (choose one depends on the type of AHB node you create)
    // If you are using AXIS, check AXISBundle and AXISBundleBits in generators/
↪rocket-chip/src/main/scala/amba/axis/Bundles.scala
  }
```

Connect Interrupt

Chipyard allows a tile to either receive interrupts from other devices or initiate interrupts to notify other cores/devices. In the tile that inherited `SinksExternalInterrupts`, one can create a `TileInterrupts` object (a Chisel bundle) and call `decodeCoreInterrupts()` with the object as the argument. Note that you should call this function in the implementation class since it returns a Chisel bundle used by RTL code. You can then read the interrupt bits from the `TileInterrupts` bundle we create above. The definition of `TileInterrupts` (in the file Interrupts.scala) is

```
class TileInterrupts(implicit p: Parameters) extends CoreBundle()(p) {
  val debug = Bool() // debug interrupt
  val mtip = Bool() // Machine level timer interrupt
  val msip = Bool() // Machine level software interrupt
  val meip = Bool() // Machine level external interrupt
  val seip = usingSupervisor.option(Bool()) // Valid only if supervisor mode is␣
↪supported
  val lip = Vec(coreParams.nLocalInterrupts, Bool())  // Local interrupts
}
```

Here is an example on how to connect these signals in the implementation class:

```
  // For example, our core support debug interrupt and machine-level interrupt, and␣
↪suppose the following two signals
  // are the interrupt inputs to the core. (DO NOT COPY this code - if your core␣
↪treat each type of interrupt differently,
  // you need to connect them to different interrupt ports of your core)
  val debug_i = Wire(Bool())
  val mtip_i = Wire(Bool())
  // We create a bundle here and decode the interrupt.
  val int_bundle = new TileInterrupts()
  outer.decodeCoreInterrupts(int_bundle)
  debug_i := int_bundle.debug
  mtip_i := int_bundle.meip & int_bundle.msip & int_bundle.mtip
```

Also, the tile can also notify other cores or devices for some events by calling following functions in `SourcesExternalNotifications` from the implementation class: (These functions can be found in in the trait `SourcesExternalNotifications` in the file Interrupts.scala)

```
def reportHalt(could_halt: Option[Bool]) // Triggered when there is an unrecoverable␣
↪hardware error (halt the machine)
def reportHalt(errors: Seq[CanHaveErrors]) // Varient for standard error bundle␣
↪(Rocket specific: used only by cache when there's an ECC error)
def reportCease(could_cease: Option[Bool], quiescenceCycles: Int = 8) // Triggered␣
↪when the core stop retiring instructions (like clock gating)
def reportWFI(could_wfi: Option[Bool]) // Triggered when a WFI instruction is executed
```

Here is an example on how to use these functions to raise interrupt.

```
  // This is a demo. You should call these function according to your core
  // Suppose that the following signal is from the decoder indicating a WFI␣
↪instruction is received.
  val wfi_o = Wire(Bool())
  outer.reportWFI(Some(wfi_o))
  // Suppose that the following signal indicate an unreconverable hardware error.
  val halt_o = Wire(Bool())
  outer.reportHalt(Some(halt_o))
  // Suppose that our core never stall for a long time / stop retiring. Use None to␣
↪indicate that this interrupt never fires.
  outer.reportCease(None)
```

Create Config Fragments to Integrate the Core

To use your core in a Chipyard config, you will need a config fragment that will create a `TileParams` object of your core in the current config. An example of such config will be like this:

```
class WithNMyCores(n: Int = 1, overrideIdOffset: Option[Int] = None) extends␣
↪Config((site, here, up) => {
  case TilesLocated(InSubsystem) => {
    // Calculate the next available hart ID (since hart ID cannot be duplicated)
    val prev = up(TilesLocated(InSubsystem), site)
    val idOffset = overrideIdOffset.getOrElse(prev.size)
    // Create TileAttachParams for every core to be instantiated
    (0 until n).map { i =>
      MyTileAttachParams(
        tileParams = MyTileParams(hartId = i + idOffset),
        crossingParams = RocketCrossingParams()
      )
    } ++ prev
  }
  // Configurate # of bytes in one memory / IO transaction. For RV64, one load/store␣
↪instruction can transfer 8 bytes at most.
  case SystemBusKey => up(SystemBusKey, site).copy(beatBytes = 8)
  // The # of instruction bits. Use maximum # of bits if your core supports both 32␣
↪and 64 bits.
  case XLen => 64
})
```

Chipyard looks up the tile parameters in the field `TilesLocated(InSubsystem)`, whose type is a list of `InstantiableTileParams`. This config fragment simply appends new tile parameters to the end of this list.

Now you have finished all the steps to prepare your cores for Chipyard! To generate the custom core, simply follow the instructions in *Integrating Custom Chisel Projects into the Generator Build System* to add your project to the build system, then create a config by following the steps in *Heterogeneous SoCs*. You can now run most desired workflows for the new config just as you would for the built-in cores (depending on the functionality your core supports).

If you would like to see an example of a complete third-party Verilog core integrated into Chipyard, `generators/ariane/src/main/scala/CVA6Tile.scala` provides a concrete example of the CVA6 core. Note that this particular example includes additional nuances with respect to the interaction of the AXI interface with the memory coherency system.

2.6.4 RoCC vs MMIO

Accelerators or custom IO devices can be added to your SoC in several ways:

- MMIO Peripheral (a.k.a TileLink-Attached Accelerator)
- Tightly-Coupled RoCC Accelerator

These approaches differ in the method of the communication between the processor and the custom block.

With the TileLink-Attached approach, the processor communicates with MMIO peripherals through memory-mapped registers.

In contrast, the processor communicates with a RoCC accelerators through a custom protocol and custom non-standard ISA instructions reserved in the RISC-V ISA encoding space. Each core can have up to four accelerators that are controlled by custom instructions and share resources with the CPU. RoCC coprocessor instructions have the following form.

```
customX rd, rs1, rs2, funct
```

The X will be a number 0-3, and determines the opcode of the instruction, which controls which accelerator an instruction will be routed to. The `rd`, `rs1`, and `rs2` fields are the register numbers of the destination register and two source registers. The `funct` field is a 7-bit integer that the accelerator can use to distinguish different instructions from each other.

Note that communication through a RoCC interface requires a custom software toolchain, whereas MMIO peripherals can use that standard toolchain with appropriate driver support.

2.6.5 Adding a RoCC Accelerator

RoCC accelerators are lazy modules that extend the `LazyRoCC` class. Their implementation should extends the `LazyRoCCModule` class.

```
class CustomAccelerator(opcodes: OpcodeSet)
    (implicit p: Parameters) extends LazyRoCC(opcodes) {
  override lazy val module = new CustomAcceleratorModule(this)
}

class CustomAcceleratorModule(outer: CustomAccelerator)
    extends LazyRoCCModuleImp(outer) {
  val cmd = Queue(io.cmd)
  // The parts of the command are as follows
  // inst - the parts of the instruction itself
  //   opcode
  //   rd - destination register number
  //   rs1 - first source register number
  //   rs2 - second source register number
  //   funct
  //   xd - is the destination register being used?
  //   xs1 - is the first source register being used?
  //   xs2 - is the second source register being used?
  // rs1 - the value of source register 1
  // rs2 - the value of source register 2
  ...
}
```

The `opcodes` parameter for `LazyRoCC` is the set of custom opcodes that will map to this accelerator. More on this in the next subsection.

The `LazyRoCC` class contains two TLOutputNode instances, `atlNode` and `tlNode`. The former connects into a tile-local arbiter along with the backside of the L1 instruction cache. The latter connects directly to the L1-L2 crossbar. The corresponding Tilelink ports in the module implementation's IO bundle are `atl` and `tl`, respectively.

The other interfaces available to the accelerator are `mem`, which provides access to the L1 cache; `ptw` which provides access to the page-table walker; the `busy` signal, which indicates when the accelerator is still handling an instruction; and the `interrupt` signal, which can be used to interrupt the CPU.

Look at the examples in `generators/rocket-chip/src/main/scala/tile/LazyRoCC.scala` for detailed information on the different IOs.

Adding RoCC accelerator to Config

RoCC accelerators can be added to a core by overriding the `BuildRoCC` parameter in the configuration. This takes a sequence of functions producing `LazyRoCC` objects, one for each accelerator you wish to add.

For instance, if we wanted to add the previously defined accelerator and route custom0 and custom1 instructions to it, we could do the following.

```
class WithCustomAccelerator extends Config((site, here, up) => {
  case BuildRoCC => Seq((p: Parameters) => LazyModule(
    new CustomAccelerator(OpcodeSet.custom0 | OpcodeSet.custom1)(p)))
})

class CustomAcceleratorConfig extends Config(
  new WithCustomAccelerator ++
  new RocketConfig)
```

To add RoCC instructions in your program, use the RoCC C macros provided in `tests/rocc.h`. You can find examples in the files `tests/accum.c` and `charcount.c`.

2.6.6 MMIO Peripherals

The easiest way to create a MMIO peripheral is to use the `TLRegisterRouter` or `AXI4RegisterRouter` widgets, which abstracts away the details of handling the interconnect protocols and provides a convenient interface for specifying memory-mapped registers. Since Chipyard and Rocket Chip SoCs primarily use Tilelink as the on-chip interconnect protocol, this section will primarily focus on designing Tilelink-based peripherals. However, see `generators/chipyard/src/main/scala/example/GCD.scala` for how an example AXI4 based peripheral is defined and connected to the Tilelink graph through converters.

To create a RegisterRouter-based peripheral, you will need to specify a parameter case class for the configuration settings, a bundle trait with the extra top-level ports, and a module implementation containing the actual RTL.

For this example, we will show how to connect a MMIO peripheral which computes the GCD. The full code can be found in `generators/chipyard/src/main/scala/example/GCD.scala`.

In this case we use a submodule `GCDMMIOChiselModule` to actually perform the GCD. The `GCDModule` class only creates the registers and hooks them up using `regmap`.

```
class GCDMMIOChiselModule(val w: Int) extends Module
  with HasGCDIO
{
  val s_idle :: s_run :: s_done :: Nil = Enum(3)

  val state = RegInit(s_idle)
  val tmp   = Reg(UInt(w.W))
  val gcd   = Reg(UInt(w.W))

  io.input_ready := state === s_idle
  io.output_valid := state === s_done
```

(continues on next page)

(continued from previous page)

```
  io.gcd := gcd

  when (state === s_idle && io.input_valid) {
    state := s_run
  } .elsewhen (state === s_run && tmp === 0.U) {
    state := s_done
  } .elsewhen (state === s_done && io.output_ready) {
    state := s_idle
  }

  when (state === s_idle && io.input_valid) {
    gcd := io.x
    tmp := io.y
  } .elsewhen (state === s_run) {
    when (gcd > tmp) {
      gcd := gcd - tmp
    } .otherwise {
      tmp := tmp - gcd
    }
  }

  io.busy := state =/= s_idle
}
```

```
trait GCDModule extends HasRegMap {
  val io: GCDTopIO

  implicit val p: Parameters
  def params: GCDParams
  val clock: Clock
  val reset: Reset


  // How many clock cycles in a PWM cycle?
  val x = Reg(UInt(params.width.W))
  val y = Wire(new DecoupledIO(UInt(params.width.W)))
  val gcd = Wire(new DecoupledIO(UInt(params.width.W)))
  val status = Wire(UInt(2.W))

  val impl = if (params.useBlackBox) {
    Module(new GCDMMIOBlackBox(params.width))
  } else {
    Module(new GCDMMIOChiselModule(params.width))
  }

  impl.io.clock := clock
  impl.io.reset := reset.asBool

  impl.io.x := x
  impl.io.y := y.bits
  impl.io.input_valid := y.valid
  y.ready := impl.io.input_ready

  gcd.bits := impl.io.gcd
  gcd.valid := impl.io.output_valid
  impl.io.output_ready := gcd.ready
```

(continues on next page)

(continued from previous page)

```
  status := Cat(impl.io.input_ready, impl.io.output_valid)
  io.gcd_busy := impl.io.busy

  regmap(
    0x00 -> Seq(
      RegField.r(2, status)), // a read-only register capturing current status
    0x04 -> Seq(
      RegField.w(params.width, x)), // a plain, write-only register
    0x08 -> Seq(
      RegField.w(params.width, y)), // write-only, y.valid is set on write
    0x0C -> Seq(
      RegField.r(params.width, gcd))) // read-only, gcd.ready is set on read
}
```

Advanced Features of RegField Entries

`RegField` exposes polymorphic `r` and `w` methods that allow read- and write-only memory-mapped registers to be interfaced to hardware in multiple ways.

- `RegField.r(2, status)` is used to create a 2-bit, read-only register that captures the current value of the `status` signal when read.
- `RegField.r(params.width, gcd)` "connects" the decoupled handshaking interface `gcd` to a read-only memory-mapped register. When this register is read via MMIO, the `ready` signal is asserted. This is in turn connected to `output_ready` on the GCD module through the glue logic.
- `RegField.w(params.width, x)` exposes a plain register via MMIO, but makes it write-only.
- `RegField.w(params.width, y)` associates the decoupled interface signal `y` with a write-only memory-mapped register, causing `y.valid` to be asserted when the register is written.

Since the ready/valid signals of `y` are connected to the `input_ready` and `input_valid` signals of the GCD module, respectively, this register map and glue logic has the effect of triggering the GCD algorithm when `y` is written. Therefore, the algorithm is set up by first writing `x` and then performing a triggering write to `y`. Polling can be used for status checks.

Connecting by TileLink

Once you have these classes, you can construct the final peripheral by extending the `TLRegisterRouter` and passing the proper arguments. The first set of arguments determines where the register router will be placed in the global address map and what information will be put in its device tree entry. The second set of arguments is the IO bundle constructor, which we create by extending `TLRegBundle` with our bundle trait. The final set of arguments is the module constructor, which we create by extends `TLRegModule` with our module trait. Notice how we can create an analogous AXI4 version of our peripheral.

```
class GCDTL(params: GCDParams, beatBytes: Int)(implicit p: Parameters)
  extends TLRegisterRouter(
    params.address, "gcd", Seq("ucbbar,gcd"),
    beatBytes = beatBytes)(
      new TLRegBundle(params, _) with GCDTopIO)(
      new TLRegModule(params, _, _) with GCDModule)

class GCDAXI4(params: GCDParams, beatBytes: Int)(implicit p: Parameters)
  extends AXI4RegisterRouter(
```

(continues on next page)

(continued from previous page)

```
    params.address,
    beatBytes=beatBytes)(
      new AXI4RegBundle(params, _) with GCDTopIO)(
      new AXI4RegModule(params, _, _) with GCDModule)
```

Top-level Traits

After creating the module, we need to hook it up to our SoC. Rocket Chip accomplishes this using the cake pattern. This basically involves placing code inside traits. In the Rocket Chip cake, there are two kinds of traits: a `LazyModule` trait and a module implementation trait.

The `LazyModule` trait runs setup code that must execute before all the hardware gets elaborated. For a simple memory-mapped peripheral, this just involves connecting the peripheral's TileLink node to the MMIO crossbar.

```
trait CanHavePeripheryGCD { this: BaseSubsystem =>
  private val portName = "gcd"

  // Only build if we are using the TL (nonAXI4) version
  val gcd = p(GCDKey) match {
    case Some(params) => {
      if (params.useAXI4) {
        val gcd = LazyModule(new GCDAXI4(params, pbus.beatBytes)(p))
        pbus.toSlave(Some(portName)) {
          gcd.node :=
          AXI4Buffer () :=
          TLToAXI4 () :=
          // toVariableWidthSlave doesn't use holdFirstDeny, which TLToAXI4() needsx
          TLFragmenter(pbus.beatBytes, pbus.blockBytes, holdFirstDeny = true)
        }
        Some(gcd)
      } else {
        val gcd = LazyModule(new GCDTL(params, pbus.beatBytes)(p))
        pbus.toVariableWidthSlave(Some(portName)) { gcd.node }
        Some(gcd)
      }
    }
    case None => None
  }
}
```

Note that the `GCDTL` class we created from the register router is itself a `LazyModule`. Register routers have a TileLink node simply named "node", which we can hook up to the Rocket Chip bus. This will automatically add address map and device tree entries for the peripheral. Also observe how we have to place additional AXI4 buffers and converters for the AXI4 version of this peripheral.

For peripherals which instantiate a concrete module, or which need to be connected to concrete IOs or wires, a matching concrete trait is necessary. We will make our GCD example output a `gcd_busy` signal as a top-level port to demonstrate. In the concrete module implementation trait, we instantiate the top level IO (a concrete object) and wire it to the IO of our lazy module.

```
trait CanHavePeripheryGCDModuleImp extends LazyModuleImp {
  val outer: CanHavePeripheryGCD
  val gcd_busy = outer.gcd match {
    case Some(gcd) => {
      val busy = IO(Output(Bool()))
```

(continues on next page)

(continued from previous page)

```
      busy := gcd.module.io.gcd_busy
      Some(busy)
    }
    case None => None
  }
}
```

Constructing the DigitalTop and Config

Now we want to mix our traits into the system as a whole. This code is from `generators/chipyard/src/main/scala/DigitalTop.scala`.

```
class DigitalTop(implicit p: Parameters) extends ChipyardSystem
  with testchipip.CanHavePeripheryCustomBootPin // Enables optional custom boot pin
  with testchipip.HasPeripheryBootAddrReg // Use programmable boot address register
  with testchipip.CanHaveTraceIO // Enables optionally adding trace IO
  with testchipip.CanHaveBackingScratchpad // Enables optionally adding a backing␣
↪scratchpad
  with testchipip.CanHavePeripheryBlockDevice // Enables optionally adding the block␣
↪device
  with testchipip.CanHavePeripheryTLSerial // Enables optionally adding the backing␣
↪memory and serial adapter
  with sifive.blocks.devices.i2c.HasPeripheryI2C // Enables optionally adding the␣
↪sifive I2C
  with sifive.blocks.devices.pwm.HasPeripheryPWM // Enables optionally adding the␣
↪sifive PWM
  with sifive.blocks.devices.uart.HasPeripheryUART // Enables optionally adding the␣
↪sifive UART
  with sifive.blocks.devices.gpio.HasPeripheryGPIO // Enables optionally adding the␣
↪sifive GPIOs
  with sifive.blocks.devices.spi.HasPeripherySPIFlash // Enables optionally adding␣
↪the sifive SPI flash controller
  with sifive.blocks.devices.spi.HasPeripherySPI // Enables optionally adding the␣
↪sifive SPI port
  with icenet.CanHavePeripheryIceNIC // Enables optionally adding the IceNIC for␣
↪FireSim
  with chipyard.example.CanHavePeripheryInitZero // Enables optionally adding the␣
↪initzero example widget
  with chipyard.example.CanHavePeripheryGCD // Enables optionally adding the GCD␣
↪example widget
  with chipyard.example.CanHavePeripheryStreamingFIR // Enables optionally adding the␣
↪DSPTools FIR example widget
  with chipyard.example.CanHavePeripheryStreamingPassthrough // Enables optionally␣
↪adding the DSPTools streaming-passthrough example widget
  with nvidia.blocks.dla.CanHavePeripheryNVDLA // Enables optionally having an NVDLA
  with chipyard.clocking.HasChipyardPRCI // Use Chipyard reset/clock distribution
  with fftgenerator.CanHavePeripheryFFT // Enables optionally having an MMIO-based␣
↪FFT block
{
  override lazy val module = new DigitalTopModule(this)
}

class DigitalTopModule[+L <: DigitalTop](l: L) extends ChipyardSystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with sifive.blocks.devices.i2c.HasPeripheryI2CModuleImp
```

(continues on next page)

(continued from previous page)

```
  with sifive.blocks.devices.pwm.HasPeripheryPWMModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIFlashModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIModuleImp
  with chipyard.example.CanHavePeripheryGCDModuleImp
  with freechips.rocketchip.util.DontTouch
```

Just as we need separate traits for `LazyModule` and module implementation, we need two classes to build the system. The `DigitalTop` class contains the set of traits which parameterize and define the `DigitalTop`. Typically these traits will optionally add IOs or peripherals to the `DigitalTop`. The `DigitalTop` class includes the pre-elaboration code and also a `lazy val` to produce the module implementation (hence `LazyModule`). The `DigitalTopModule` class is the actual RTL that gets synthesized.

And finally, we create a configuration class in `generators/chipyard/src/main/scala/config/RocketConfigs.scala` that uses the `WithGCD` config fragment defined earlier.

```
class WithGCD(useAXI4: Boolean, useBlackBox: Boolean) extends Config((site, here, up)␣
↪=> {
  case GCDKey => Some(GCDParams(useAXI4 = useAXI4, useBlackBox = useBlackBox))
})
```

```
class GCDTLRocketConfig extends Config(
  new chipyard.example.WithGCD(useAXI4=false, useBlackBox=false) ++          // Use␣
↪GCD Chisel, connect Tilelink
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

Testing

Now we can test that the GCD is working. The test program is in `tests/gcd.c`.

```
#include "mmio.h"

#define GCD_STATUS 0x2000
#define GCD_X 0x2004
#define GCD_Y 0x2008
#define GCD_GCD 0x200C

unsigned int gcd_ref(unsigned int x, unsigned int y) {
  while (y != 0) {
    if (x > y)
      x = x - y;
    else
      y = y - x;
  }
  return x;
}

// DOC include start: GCD test
int main(void)
{
  uint32_t result, ref, x = 20, y = 15;

  // wait for peripheral to be ready
```

(continues on next page)

(continued from previous page)

```
  while ((reg_read8(GCD_STATUS) & 0x2) == 0) ;

  reg_write32(GCD_X, x);
  reg_write32(GCD_Y, y);


  // wait for peripheral to complete
  while ((reg_read8(GCD_STATUS) & 0x1) == 0) ;

  result = reg_read32(GCD_GCD);
  ref = gcd_ref(x, y);

  if (result != ref) {
    printf("Hardware result %d does not match reference value %d\n", result, ref);
    return 1;
  }
  return 0;
}
// DOC include end: GCD test
```

This just writes out to the registers we defined earlier. The base of the module's MMIO region is at 0x2000 by default. This will be printed out in the address map portion when you generate the Verilog code. You can also see how this changes the emitted `.json` addressmap files in `generated-src`.

Compiling this program with `make` produces a `gcd.riscv` executable.

Now with all of that done, we can go ahead and run our simulation.

```
cd sims/verilator
make CONFIG=GCDTLRocketConfig BINARY=../../tests/gcd.riscv run-binary
```

Dsptools is a Chisel library that aids in writing custom signal processing accelerators. It does this by: * Giving types and helpers that allow you to express mathematical operations more directly. * Typeclasses that let you write polymorphic generators, for example an FIR filter generator that works for both real- and complex-valued filters. * Structures for packaging DSP blocks and integrating them into a rocketchip-based SoC. * Test harnesses for testing DSP circuits, as well as VIP-style drivers and monitors for DSP blocks.

The Dsptools repository has more documentation.

2.6.7 Dsptools Blocks

A `DspBlock` is the basic unit of signal processing functionality that can be integrated into an SoC. It has a AXI4-stream interface and an optional memory interface. The idea is that these `DspBlocks` can be easily designed, unit tested, and assembled lego-style to build complex functionality. A `DspChain` is one example of how to assemble `DspBlocks`, in which case the streaming interfaces are connected serially into a pipeline, and a bus is instatiated and connected to every block with a memory interface.

Chipyard has example designs that integrate a `DspBlock` to a rocketchip-based SoC as an MMIO peripheral. The custom `DspBlock` has a `ReadQueue` before it and a `WriteQueue` after it, which allow memory mapped access to the streaming interfaces so the rocket core can interact with the `DspBlock`[1]. This section will primarily focus on designing Tilelink-based peripherals. However, through the resources provided in Dsptools, one could also define an

[1] `ReadQueue` and `WriteQueue` are good illustrations of how to write a `DspBlock` and how they can be integrated into rocket, but in a real design a DMA engine would be preferred. `ReadQueue` will stall the processor if you try to read an empty queue, and `WriteQueue` will stall if you try to write to a full queue, which a DMA engine can more elegantly avoid. Furthermore, a DMA engine can do the work of moving data, freeing the processor to do other useful work (or sleep).

AXI4-based peripheral by following similar steps. Furthermore, the examples here are simple, but can be extended to implement more complex accelerators, for example an OFDM baseband or a spectrometer.

For this example, we will show you how to connect a simple FIR filter created using Dsptools as an MMIO peripheral as shown in the figure above. The full code can be found in `generators/chipyard/src/main/scala/example/dsptools/GenericFIR.scala`. That being said, one could substitute any module with a ready valid interface in the place of the FIR and achieve the same results. As long as the read and valid signals of the module are attached to those of a corresponding `DSPBlock` wrapper, and that wrapper is placed in a chain with a `ReadQueue` and a `WriteQueue`, following the general outline establised by these steps will allow you to interact with that block as a memory mapped IO.

The module `GenericFIR` is the overall wrapper of our FIR module. This module links together a variable number of `GenericFIRDirectCell` submodules, each of which performs the computations for one coefficient in a FIR direct form architecture. It is important to note that both modules are type-generic, which means that they can be instantiated for any datatype `T` that implements `Ring` operations (e.g. addition, multiplication, identities).

```
class GenericFIR[T<:Data:Ring](genIn:T, genOut:T, coeffs: Seq[T]) extends Module {
  val io = IO(GenericFIRIO(genIn, genOut))

  // Construct a vector of genericFIRDirectCells
  val directCells = Seq.fill(coeffs.length){ Module(new GenericFIRDirectCell(genIn,␣
↪genOut)).io }

  // Construct the direct FIR chain
  for ((cell, coeff) <- directCells.zip(coeffs)) {
    cell.coeff := coeff
  }

  // Connect input to first cell
  directCells.head.in.bits.data := io.in.bits.data
  directCells.head.in.bits.carry := Ring[T].zero
  directCells.head.in.valid := io.in.valid
  io.in.ready := directCells.head.in.ready

  // Connect adjacent cells
  // Note that .tail() returns a collection that consists of all
  // elements in the inital collection minus the first one.
  // This means that we zip together directCells[0, n] and
  // directCells[1, n]. However, since zip ignores unmatched elements,
  // the resulting zip is (directCells[0], directCells[1]) ...
  // (directCells[n-1], directCells[n])
  for ((current, next) <- directCells.zip(directCells.tail)) {
    next.in.bits := current.out.bits
    next.in.valid := current.out.valid
    current.out.ready := next.in.ready
  }

  // Connect output to last cell
  io.out.bits.data := directCells.last.out.bits.carry
  directCells.last.out.ready := io.out.ready
  io.out.valid := directCells.last.out.valid

}
```

```
class GenericFIRDirectCell[T<:Data:Ring](genIn: T, genOut: T) extends Module {
  val io = IO(GenericFIRCellIO(genIn, genOut))
```

(continues on next page)

(continued from previous page)

```
  // Registers to delay the input and the valid to propagate with calculations
  val hasNewData = RegInit(0.U)
  val inputReg = Reg(genIn.cloneType)

  // Passthrough ready
  io.in.ready := io.out.ready

  // When a new transaction is ready on the input, we will have new data to output
  // next cycle. Take this data in
  when (io.in.fire) {
    hasNewData := 1.U
    inputReg := io.in.bits.data
  }

  // We should output data when our cell has new data to output and is ready to
  // recieve new data. This insures that every cell in the chain passes its data
  // on at the same time
  io.out.valid := hasNewData & io.in.fire
  io.out.bits.data := inputReg

  // Compute carry
  // This uses the ring implementation for + and *, i.e.
  // (a * b) maps to (Ring[T].prod(a, b)) for whicever T you use
  io.out.bits.carry := inputReg * io.coeff + io.in.bits.carry
}
```

Creating a DspBlock

The first step in attaching the FIR filter as a MMIO peripheral is to create an abstract subclass of `DspBlock` the wraps around the `GenericFIR` module. Streaming outputs and inputs are packed and unpacked into `UInt` s. If there were control signals, this is where they'd go from raw IOs to memory mapped. The main steps of this process are as follows.

1. Instantiate a `GenericFIR` within `GenericFIRBlock`.
2. Attach the ready and valid signals from the in and out connections.
3. Cast the module input data to the input type of `GenericFIR` (`GenericFIRBundle`) and attach.
4. Cast the output of `GenericFIR` to `UInt` and attach to the module output.

```
abstract class GenericFIRBlock[D, U, EO, EI, B<:Data, T<:Data:Ring]
(
  genIn: T,
  genOut: T,
  coeffs: Seq[T]
)(implicit p: Parameters) extends DspBlock[D, U, EO, EI, B] {
  val streamNode = AXI4StreamIdentityNode()
  val mem = None

  lazy val module = new LazyModuleImp(this) {
    require(streamNode.in.length == 1)
    require(streamNode.out.length == 1)

    val in = streamNode.in.head._1
    val out = streamNode.out.head._1
```

(continues on next page)

(continued from previous page)

```
    // instantiate generic fir
    val fir = Module(new GenericFIR(genIn, genOut, coeffs))

    // Attach ready and valid to outside interface
    in.ready := fir.io.in.ready
    fir.io.in.valid := in.valid

    fir.io.out.ready := out.ready
    out.valid := fir.io.out.valid

    // cast UInt to T
    fir.io.in.bits := in.bits.data.asTypeOf(GenericFIRBundle(genIn))

    // cast T to UInt
    out.bits.data := fir.io.out.bits.asUInt
  }
}
```

Note that at this point the `GenericFIRBlock` does not have a type of memory interface specified. This abstract class can be used to create different flavors that use AXI-4, TileLink, AHB, or whatever other memory interface you like like.

Connecting DspBlock by TileLink

With these classes implemented, you can begin to construct the chain by extending `GenericFIRBlock` while using the `TLDspBlock` trait via mixin.

```
class TLGenericFIRBlock[T<:Data:Ring]
(
  val genIn: T,
  val genOut: T,
  coeffs: Seq[T]
)(implicit p: Parameters) extends
GenericFIRBlock[TLClientPortParameters, TLManagerPortParameters, TLEdgeOut, TLEdgeIn,␣
↪TLBundle, T](
    genIn, genOut, coeffs
) with TLDspBlock
```

We can then construct the final chain by utilizing the `TLWriteQueue` and `TLReadeQueue` modules found in `generators/chipyard/src/main/scala/example/dsptools/DspBlocks.scala`. The chain is created by passing a list of factory functions to the constructor of `TLChain`. The constructor then automatically instantiates these `DspBlocks`, connects their stream nodes in order, creates a bus, and connects any `DspBlocks` that have memory interfaces to the bus.

```
class TLGenericFIRChain[T<:Data:Ring] (genIn: T, genOut: T, coeffs: Seq[T], params:␣
↪GenericFIRParams)(implicit p: Parameters)
  extends TLChain(Seq(
    TLWriteQueue(params.depth, AddressSet(params.writeAddress, 0xff))(_),
    { implicit p: Parameters =>
      val fir = LazyModule(new TLGenericFIRBlock(genIn, genOut, coeffs))
      fir
    },
    TLReadQueue(params.depth, AddressSet(params.readAddress, 0xff))(_)
  ))
```

Top Level Traits

As in the previous MMIO example, we use a cake pattern to hook up our module to our SoC.

```
trait CanHavePeripheryStreamingFIR extends BaseSubsystem {
  val streamingFIR = p(GenericFIRKey) match {
    case Some(params) => {
      val streamingFIR = LazyModule(new TLGenericFIRChain(
        genIn = FixedPoint(8.W, 3.BP),
        genOut = FixedPoint(8.W, 3.BP),
        coeffs = Seq(1.F(0.BP), 2.F(0.BP), 3.F(0.BP)),
        params = params))
      pbus.toVariableWidthSlave(Some("streamingFIR")) { streamingFIR.mem.get :=␣
↪TLFIFOFixer() }
      Some(streamingFIR)
    }
    case None => None
  }
}
```

Note that this is the point at which we decide the datatype for our FIR. You could create different configs that use different types for the FIR, for example a config that instantiates a complex-valued FIR filter.

Constructing the Top and Config

Once again following the path of the previous MMIO example, we now want to mix our traits into the system as a whole. The code is from `generators/chipyard/src/main/scala/DigitalTop.scala`

```
class DigitalTop(implicit p: Parameters) extends ChipyardSystem
  with testchipip.CanHavePeripheryCustomBootPin // Enables optional custom boot pin
  with testchipip.HasPeripheryBootAddrReg // Use programmable boot address register
  with testchipip.CanHaveTraceIO // Enables optionally adding trace IO
  with testchipip.CanHaveBackingScratchpad // Enables optionally adding a backing␣
↪scratchpad
  with testchipip.CanHavePeripheryBlockDevice // Enables optionally adding the block␣
↪device
  with testchipip.CanHavePeripheryTLSerial // Enables optionally adding the backing␣
↪memory and serial adapter
  with sifive.blocks.devices.i2c.HasPeripheryI2C // Enables optionally adding the␣
↪sifive I2C
  with sifive.blocks.devices.pwm.HasPeripheryPWM // Enables optionally adding the␣
↪sifive PWM
  with sifive.blocks.devices.uart.HasPeripheryUART // Enables optionally adding the␣
↪sifive UART
  with sifive.blocks.devices.gpio.HasPeripheryGPIO // Enables optionally adding the␣
↪sifive GPIOs
  with sifive.blocks.devices.spi.HasPeripherySPIFlash // Enables optionally adding␣
↪the sifive SPI flash controller
  with sifive.blocks.devices.spi.HasPeripherySPI // Enables optionally adding the␣
↪sifive SPI port
  with icenet.CanHavePeripheryIceNIC // Enables optionally adding the IceNIC for␣
↪FireSim
  with chipyard.example.CanHavePeripheryInitZero // Enables optionally adding the␣
↪initzero example widget
  with chipyard.example.CanHavePeripheryGCD // Enables optionally adding the GCD␣
↪example widget
  with chipyard.example.CanHavePeripheryStreamingFIR // Enables optionally adding the␣
↪DSPTools FIR example widget
```

(continues on next page)

(continued from previous page)

```
  with chipyard.example.CanHavePeripheryStreamingPassthrough // Enables optionally␣
↪adding the DSPTools streaming-passthrough example widget
  with nvidia.blocks.dla.CanHavePeripheryNVDLA // Enables optionally having an NVDLA
  with chipyard.clocking.HasChipyardPRCI // Use Chipyard reset/clock distribution
  with fftgenerator.CanHavePeripheryFFT // Enables optionally having an MMIO-based␣
↪FFT block
{
  override lazy val module = new DigitalTopModule(this)
}

class DigitalTopModule[+L <: DigitalTop](l: L) extends ChipyardSystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with sifive.blocks.devices.i2c.HasPeripheryI2CModuleImp
  with sifive.blocks.devices.pwm.HasPeripheryPWMModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIFlashModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIModuleImp
  with chipyard.example.CanHavePeripheryGCDModuleImp
  with freechips.rocketchip.util.DontTouch
```

Finally, we create the configuration class in `generators/chipyard/src/main/scala/config/RocketConfigs.scala` that uses the `WithFIR` mixin defined in `generators/chipyard/src/main/scala/example/dsptools/GenericFIR.scala`.

```
class WithStreamingFIR extends Config((site, here, up) => {
  case GenericFIRKey => Some(GenericFIRParams(depth = 8))
})
```

```
class StreamingFIRRocketConfig extends Config (
  new chipyard.example.WithStreamingFIR ++                    // use top with tilelink-
↪controlled streaming FIR
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

FIR Testing

We can now test that the FIR is working. The test program is found in `tests/streaming-fir.c`.

```
#define PASSTHROUGH_WRITE 0x2000
#define PASSTHROUGH_WRITE_COUNT 0x2008
#define PASSTHROUGH_READ 0x2100
#define PASSTHROUGH_READ_COUNT 0x2108

#define BP 3
#define BP_SCALE ((double)(1 << BP))

#include "mmio.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
```

(continues on next page)

(continued from previous page)

```
uint64_t roundi(double x)
{
  if (x < 0.0) {
    return (uint64_t)(x - 0.5);
  } else {
    return (uint64_t)(x + 0.5);
  }
}

int main(void)
{
  double test_vector[15] = {1.0, 2.0, 3.0, 4.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.5, 0.25, 0.
↪125, 0.125};
  uint32_t num_tests = sizeof(test_vector) / sizeof(double);
  printf("Starting writing %d inputs\n", num_tests);

  for (int i = 0; i < num_tests; i++) {
    reg_write64(PASSTHROUGH_WRITE, roundi(test_vector[i] * BP_SCALE));
  }

  printf("Done writing\n");
  uint32_t rcnt = reg_read32(PASSTHROUGH_READ_COUNT);
  printf("Write count: %d\n", reg_read32(PASSTHROUGH_WRITE_COUNT));
  printf("Read count: %d\n", rcnt);

  int failed = 0;
  if (rcnt != 0) {
    for (int i = 0; i < num_tests - 3; i++) {
      uint32_t res = reg_read32(PASSTHROUGH_READ);
      // double res = ((double)reg_read32(PASSTHROUGH_READ)) / BP_SCALE;
      double expected_double = 3*test_vector[i] + 2*test_vector[i+1] + test_
↪vector[i+2];
      uint32_t expected = ((uint32_t)(expected_double * BP_SCALE + 0.5)) & 0xFF;
      if (res == expected) {
        printf("\n\nPass: Got %u Expected %u\n\n", res, expected);
      } else {
        failed = 1;
        printf("\n\nFail: Got %u Expected %u\n\n", res, expected);
      }
    }
  } else {
    failed = 1;
  }

  if (failed) {
    printf("\n\nSome tests failed\n\n");
  } else {
    printf("\n\nAll tests passed\n\n");
  }

  return 0;
}
```

The test feed a series of values into the fir and compares the output to a golden model of computation. The base of the module's MMIO write region is at 0x2000 and the base of the read region is at 0x2100 by default.

Compiling this program with `make` produces a `streaming-fir.riscv` executable.

Now we can run our simulation.

```
cd sims/verilator
make CONFIG=StreamingFIRRocketConfig BINARY=../../tests/streaming-fir.riscv run-binary
```

2.6.8 Keys, Traits, and Configs

You have probably seen snippets of Chisel referencing keys, traits, and configs by this point. This section aims to elucidate the interactions between these Chisel/Scala components, and provide best practices for how these should be used to create a parameterized design and configure it.

We will continue to use the GCD example.

Keys

Keys specify some parameter which controls some custom widget. Keys should typically be implemented as **Option types**, with a default value of `None` that means no change in the system. In other words, the default behavior when the user does not explicitly set the key should be a no-op.

Keys should be defined and documented in sub-projects, since they generally deal with some specific block, and not system-level integration. (We make an exception for the example GCD widget).

```
case object GCDKey extends Field[Option[GCDParams]](None)
```

The object within a key is typically a `case class XXXParams`, which defines a set of parameters which some block accepts. For example, the GCD widget's `GCDParams` parameterizes its address, operand widths, whether the widget should be connected by Tilelink or AXI4, and whether the widget should use the blackbox-Verilog implementation, or the Chisel implementation.

```
case class GCDParams(
  address: BigInt = 0x2000,
  width: Int = 32,
  useAXI4: Boolean = false,
  useBlackBox: Boolean = true)
```

Accessing the value stored in the key is easy in Chisel, as long as the `implicit p: Parameters` object is being passed through to the relevant module. For example, `p(GCDKey).get.address` returns the address field of `GCDParams`. Note this only works if `GCDKey` was not set to `None`, so your Chisel should check for that case!

Traits

Typically, most custom blocks will need to modify the behavior of some pre-existing block. For example, the GCD widget needs the `DigitalTop` module to instantiate and connect the widget via Tilelink, generate a top-level `gcd_busy` port, and connect that to the module as well. Traits let us do this without modifying the existing code for the `DigitalTop`, and enables compartmentalization of code for different custom blocks.

Top-level traits specify that the `DigitalTop` has been parameterized to read some custom key and optionally instantiate and connect a widget defined by that key. Traits **should not** mandate the instantiation of custom logic. In other words, traits should be written with `CanHave` semantics, where the default behavior when the key is unset is a no-op.

Top-level traits should be defined and documented in subprojects, alongside their corresponding keys. The traits should then be added to the `DigitalTop` being used by Chipyard.

Below we see the traits for the GCD example. The Lazy trait connects the GCD module to the Diplomacy graph, while the Implementation trait causes the `DigitalTop` to instantiate an additional port and concretely connect it to the GCD module.

```
trait CanHavePeripheryGCD { this: BaseSubsystem =>
  private val portName = "gcd"

  // Only build if we are using the TL (nonAXI4) version
  val gcd = p(GCDKey) match {
    case Some(params) => {
      if (params.useAXI4) {
        val gcd = LazyModule(new GCDAXI4(params, pbus.beatBytes)(p))
        pbus.toSlave(Some(portName)) {
          gcd.node :=
          AXI4Buffer () :=
          TLToAXI4 () :=
          // toVariableWidthSlave doesn't use holdFirstDeny, which TLToAXI4() needsx
          TLFragmenter(pbus.beatBytes, pbus.blockBytes, holdFirstDeny = true)
        }
        Some(gcd)
      } else {
        val gcd = LazyModule(new GCDTL(params, pbus.beatBytes)(p))
        pbus.toVariableWidthSlave(Some(portName)) { gcd.node }
        Some(gcd)
      }
    }
    case None => None
  }
}
// DOC include end: GCD lazy trait

// DOC include start: GCD imp trait
trait CanHavePeripheryGCDModuleImp extends LazyModuleImp {
  val outer: CanHavePeripheryGCD
  val gcd_busy = outer.gcd match {
    case Some(gcd) => {
      val busy = IO(Output(Bool()))
      busy := gcd.module.io.gcd_busy
      Some(busy)
    }
    case None => None
  }
}
```

These traits are added to the default `DigitalTop` in Chipyard.

```
class DigitalTop(implicit p: Parameters) extends ChipyardSystem
  with testchipip.CanHavePeripheryCustomBootPin // Enables optional custom boot pin
  with testchipip.HasPeripheryBootAddrReg // Use programmable boot address register
  with testchipip.CanHaveTraceIO // Enables optionally adding trace IO
  with testchipip.CanHaveBackingScratchpad // Enables optionally adding a backing␣
↪scratchpad
  with testchipip.CanHavePeripheryBlockDevice // Enables optionally adding the block␣
↪device
  with testchipip.CanHavePeripheryTLSerial // Enables optionally adding the backing␣
↪memory and serial adapter
  with sifive.blocks.devices.i2c.HasPeripheryI2C // Enables optionally adding the␣
↪sifive I2C
```

(continues on next page)

(continued from previous page)

```
  with sifive.blocks.devices.pwm.HasPeripheryPWM // Enables optionally adding the␣
↪sifive PWM
  with sifive.blocks.devices.uart.HasPeripheryUART // Enables optionally adding the␣
↪sifive UART
  with sifive.blocks.devices.gpio.HasPeripheryGPIO // Enables optionally adding the␣
↪sifive GPIOs
  with sifive.blocks.devices.spi.HasPeripherySPIFlash // Enables optionally adding␣
↪the sifive SPI flash controller
  with sifive.blocks.devices.spi.HasPeripherySPI // Enables optionally adding the␣
↪sifive SPI port
  with icenet.CanHavePeripheryIceNIC // Enables optionally adding the IceNIC for␣
↪FireSim
  with chipyard.example.CanHavePeripheryInitZero // Enables optionally adding the␣
↪initzero example widget
  with chipyard.example.CanHavePeripheryGCD // Enables optionally adding the GCD␣
↪example widget
  with chipyard.example.CanHavePeripheryStreamingFIR // Enables optionally adding the␣
↪DSPTools FIR example widget
  with chipyard.example.CanHavePeripheryStreamingPassthrough // Enables optionally␣
↪adding the DSPTools streaming-passthrough example widget
  with nvidia.blocks.dla.CanHavePeripheryNVDLA // Enables optionally having an NVDLA
  with chipyard.clocking.HasChipyardPRCI // Use Chipyard reset/clock distribution
  with fftgenerator.CanHavePeripheryFFT // Enables optionally having an MMIO-based␣
↪FFT block
{
  override lazy val module = new DigitalTopModule(this)
}

class DigitalTopModule[+L <: DigitalTop](l: L) extends ChipyardSystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with sifive.blocks.devices.i2c.HasPeripheryI2CModuleImp
  with sifive.blocks.devices.pwm.HasPeripheryPWMModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIFlashModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIModuleImp
  with chipyard.example.CanHavePeripheryGCDModuleImp
  with freechips.rocketchip.util.DontTouch
```

Config Fragments

Config fragments set the keys to a non-default value. Together, the collection of config fragments which define a configuration generate the values for all the keys used by the generator.

For example, the `WithGCD` config fragment is parameterized by the type of GCD widget you want to instantiate. When this config fragment is added to a config, the `GCDKey` is set to a instance of `GCDParams`, informing the previously mentioned traits to instantiate and connect the GCD widget appropriately.

```
class WithGCD(useAXI4: Boolean, useBlackBox: Boolean) extends Config((site, here, up)␣
↪=> {
  case GCDKey => Some(GCDParams(useAXI4 = useAXI4, useBlackBox = useBlackBox))
})
```

We can use this config fragment when composing our configs.

```
class GCDTLRocketConfig extends Config(
  new chipyard.example.WithGCD(useAXI4=false, useBlackBox=false) ++          // Use␣
↪GCD Chisel, connect Tilelink
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

Note: Readers who want more information on the configuration system may be interested in reading *Context-Dependent-Environments*.

2.6.9 Adding a DMA Device

DMA devices are Tilelink widgets which act as masters. In other words, DMA devices can send their own read and write requests to the chip's memory system.

For IO devices or accelerators (like a disk or network driver), instead of having the CPU poll data from the device, we may want to have the device write directly to the coherent memory system instead. For example, here is a device that writes zeros to the memory at a configured address.

```
package chipyard.example

import chisel3._
import chisel3.util._
import freechips.rocketchip.subsystem.{BaseSubsystem, CacheBlockBytes}
import freechips.rocketchip.config.{Parameters, Field, Config}
import freechips.rocketchip.diplomacy.{LazyModule, LazyModuleImp, IdRange}
import testchipip.TLHelper

case class InitZeroConfig(base: BigInt, size: BigInt)
case object InitZeroKey extends Field[Option[InitZeroConfig]](None)

class InitZero(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeClientNode(
    name = "init-zero", sourceId = IdRange(0, 1))

  lazy val module = new InitZeroModuleImp(this)
}

class InitZeroModuleImp(outer: InitZero) extends LazyModuleImp(outer) {
  val config = p(InitZeroKey).get

  val (mem, edge) = outer.node.out(0)
  val addrBits = edge.bundle.addressBits
  val blockBytes = p(CacheBlockBytes)

  require(config.size % blockBytes == 0)

  val s_init :: s_write :: s_resp :: s_done :: Nil = Enum(4)
  val state = RegInit(s_init)

  val addr = Reg(UInt(addrBits.W))
  val bytesLeft = Reg(UInt(log2Ceil(config.size+1).W))

  mem.a.valid := state === s_write
  mem.a.bits := edge.Put(
```

(continues on next page)

(continued from previous page)

```
    fromSource = 0.U,
    toAddress = addr,
    lgSize = log2Ceil(blockBytes).U,
    data = 0.U)._2
  mem.d.ready := state === s_resp

  when (state === s_init) {
    addr := config.base.U
    bytesLeft := config.size.U
    state := s_write
  }

  when (edge.done(mem.a)) {
    addr := addr + blockBytes.U
    bytesLeft := bytesLeft - blockBytes.U
    state := s_resp
  }

  when (mem.d.fire) {
    state := Mux(bytesLeft === 0.U, s_done, s_write)
  }
}

trait CanHavePeripheryInitZero { this: BaseSubsystem =>
  implicit val p: Parameters

  p(InitZeroKey) .map { k =>
    val initZero = LazyModule(new InitZero()(p))
    fbus.fromPort(Some("init-zero"))() := initZero.node
  }
}


// DOC include start: WithInitZero
class WithInitZero(base: BigInt, size: BigInt) extends Config((site, here, up) => {
  case InitZeroKey => Some(InitZeroConfig(base, size))
})
// DOC include end: WithInitZero
```

```
class DigitalTop(implicit p: Parameters) extends ChipyardSystem
  with testchipip.CanHavePeripheryCustomBootPin // Enables optional custom boot pin
  with testchipip.HasPeripheryBootAddrReg // Use programmable boot address register
  with testchipip.CanHaveTraceIO // Enables optionally adding trace IO
  with testchipip.CanHaveBackingScratchpad // Enables optionally adding a backing␣
↪scratchpad
  with testchipip.CanHavePeripheryBlockDevice // Enables optionally adding the block␣
↪device
  with testchipip.CanHavePeripheryTLSerial // Enables optionally adding the backing␣
↪memory and serial adapter
  with sifive.blocks.devices.i2c.HasPeripheryI2C // Enables optionally adding the␣
↪sifive I2C
  with sifive.blocks.devices.pwm.HasPeripheryPWM // Enables optionally adding the␣
↪sifive PWM
  with sifive.blocks.devices.uart.HasPeripheryUART // Enables optionally adding the␣
↪sifive UART
  with sifive.blocks.devices.gpio.HasPeripheryGPIO // Enables optionally adding the␣
↪sifive GPIOs
```

(continues on next page)

(continued from previous page)

```
  with sifive.blocks.devices.spi.HasPeripherySPIFlash // Enables optionally adding␣
↪the sifive SPI flash controller
  with sifive.blocks.devices.spi.HasPeripherySPI // Enables optionally adding the␣
↪sifive SPI port
  with icenet.CanHavePeripheryIceNIC // Enables optionally adding the IceNIC for␣
↪FireSim
  with chipyard.example.CanHavePeripheryInitZero // Enables optionally adding the␣
↪initzero example widget
  with chipyard.example.CanHavePeripheryGCD // Enables optionally adding the GCD␣
↪example widget
  with chipyard.example.CanHavePeripheryStreamingFIR // Enables optionally adding the␣
↪DSPTools FIR example widget
  with chipyard.example.CanHavePeripheryStreamingPassthrough // Enables optionally␣
↪adding the DSPTools streaming-passthrough example widget
  with nvidia.blocks.dla.CanHavePeripheryNVDLA // Enables optionally having an NVDLA
  with chipyard.clocking.HasChipyardPRCI // Use Chipyard reset/clock distribution
  with fftgenerator.CanHavePeripheryFFT // Enables optionally having an MMIO-based␣
↪FFT block
{
  override lazy val module = new DigitalTopModule(this)
}

class DigitalTopModule[+L <: DigitalTop](l: L) extends ChipyardSystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with sifive.blocks.devices.i2c.HasPeripheryI2CModuleImp
  with sifive.blocks.devices.pwm.HasPeripheryPWMModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIFlashModuleImp
  with sifive.blocks.devices.spi.HasPeripherySPIModuleImp
  with chipyard.example.CanHavePeripheryGCDModuleImp
  with freechips.rocketchip.util.DontTouch
```

We use `TLHelper.makeClientNode` to create a TileLink client node for us. We then connect the client node to the memory system through the front bus (fbus). For more info on creating TileLink client nodes, take a look at *Client Node*.

Once we've created our top-level module including the DMA widget, we can create a configuration for it as we did before.

```
class WithInitZero(base: BigInt, size: BigInt) extends Config((site, here, up) => {
  case InitZeroKey => Some(InitZeroConfig(base, size))
})
```

```
class InitZeroRocketConfig extends Config(
  new chipyard.example.WithInitZero(0x88000000L, 0x1000L) ++   // add InitZero
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

2.6.10 Incorporating Verilog Blocks

Working with existing Verilog IP is an integral part of many chip design flows. Fortunately, both Chisel and Chipyard provide extensive support for Verilog integration.

Here, we will examine the process of incorporating an MMIO peripheral that uses a Verilog implementation of Greatest Common Denominator (GCD) algorithm. There are a few steps to adding a Verilog peripheral:

- Adding a Verilog resource file to the project
- Defining a Chisel `BlackBox` representing the Verilog module
- Instantiating the `BlackBox` and interfacing `RegField` entries
- Setting up a chip `Top` and `Config` that use the peripheral

Adding a Verilog Blackbox Resource File

As before, it is possible to incorporate peripherals as part of your own generator project. However, Verilog resource files must go in a different directory from Chisel (Scala) sources.

```
generators/yourproject/
    build.sbt
    src/main/
        scala/
        resources/
            vsrc/
                YourFile.v
```

In addition to the steps outlined in the previous section on adding a project to the `build.sbt` at the top level, it is also necessary to add any projects that contain Verilog IP as dependencies to the `tapeout` project. This ensures that the Verilog sources are visible to the downstream FIRRTL passes that provide utilities for integrating Verilog files into the build process, which are part of the `tapeout` package in `barstools/tapeout`.

```
lazy val tapeout = conditionalDependsOn(project in file("./tools/barstools/tapeout/"))
  .dependsOn(chisel_testers, example, yourproject)
  .settings(commonSettings)
```

For this concrete GCD example, we will be using a `GCDMMIOBlackBox` Verilog module that is defined in the `chipyard` project. The Scala and Verilog sources follow the prescribed directory layout.

```
generators/chipyard/
    build.sbt
    src/main/
        scala/
            example/
                GCD.scala
        resources/
            vsrc/
                GCDMMIOBlackBox.v
```

Defining a Chisel BlackBox

A Chisel `BlackBox` module provides a way of instantiating a module defined by an external Verilog source. The definition of the blackbox includes several aspects that allow it to be translated to an instance of the Verilog module:

- An `io` field: a bundle with fields corresponding to the portlist of the Verilog module.
- A constructor parameter that takes a `Map` from Verilog parameter name to elaborated value
- One or more resources added to indicate Verilog source dependencies

Of particular interest is the fact that parameterized Verilog modules can be passed the full space of possible parameter values. These values may depend on elaboration-time values in the Chisel generator, as the bitwidth of the GCD calculation does in this example.

Verilog GCD port list and parameters

```
module GCDMMIOBlackBox
  #(parameter WIDTH)
   (
    input                  clock,
    input                  reset,
    output                 input_ready,
    input                  input_valid,
    input [WIDTH-1:0]      x,
    input [WIDTH-1:0]      y,
    input                  output_ready,
    output                 output_valid,
    output reg [WIDTH-1:0] gcd,
    output                 busy
    );
```

Chisel BlackBox Definition

```
class GCDMMIOBlackBox(val w: Int) extends BlackBox(Map("WIDTH" -> IntParam(w))) with␣
↪HasBlackBoxResource
  with HasGCDIO
{
  addResource("/vsrc/GCDMMIOBlackBox.v")
}
```

Instantiating the BlackBox and Defining MMIO

Next, we must instantiate the blackbox. In order to take advantage of diplomatic memory mapping on the system bus, we still have to integrate the peripheral at the Chisel level by mixing peripheral-specific traits into a `TLRegisterRouter`. The `params` member and `HasRegMap` base trait should look familiar from the previous memory-mapped GCD device example.

```
trait GCDModule extends HasRegMap {
  val io: GCDTopIO

  implicit val p: Parameters
  def params: GCDParams
  val clock: Clock
  val reset: Reset


  // How many clock cycles in a PWM cycle?
  val x = Reg(UInt(params.width.W))
  val y = Wire(new DecoupledIO(UInt(params.width.W)))
  val gcd = Wire(new DecoupledIO(UInt(params.width.W)))
  val status = Wire(UInt(2.W))

  val impl = if (params.useBlackBox) {
    Module(new GCDMMIOBlackBox(params.width))
  } else {
    Module(new GCDMMIOChiselModule(params.width))
  }

  impl.io.clock := clock
  impl.io.reset := reset.asBool
```

(continues on next page)

(continued from previous page)

```
  impl.io.x := x
  impl.io.y := y.bits
  impl.io.input_valid := y.valid
  y.ready := impl.io.input_ready

  gcd.bits := impl.io.gcd
  gcd.valid := impl.io.output_valid
  impl.io.output_ready := gcd.ready

  status := Cat(impl.io.input_ready, impl.io.output_valid)
  io.gcd_busy := impl.io.busy

  regmap(
    0x00 -> Seq(
      RegField.r(2, status)), // a read-only register capturing current status
    0x04 -> Seq(
      RegField.w(params.width, x)), // a plain, write-only register
    0x08 -> Seq(
      RegField.w(params.width, y)), // write-only, y.valid is set on write
    0x0C -> Seq(
      RegField.r(params.width, gcd))) // read-only, gcd.ready is set on read
}
```

Defining a Chip with a BlackBox

Since we've parameterized the GCD instantiation to choose between the Chisel and the Verilog module, creating a config is easy.

```
class GCDAXI4BlackBoxRocketConfig extends Config(
  new chipyard.example.WithGCD(useAXI4=true, useBlackBox=true) ++             // Use␣
↪GCD blackboxed verilog, connect by AXI4->Tilelink
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

You can play with the parameterization of the mixin to choose a TL/AXI4, BlackBox/Chisel version of the GCD.

Software Testing

The GCD module has a more complex interface, so polling is used to check the status of the device before each triggering read or write.

```
int main(void)
{
  uint32_t result, ref, x = 20, y = 15;

  // wait for peripheral to be ready
  while ((reg_read8(GCD_STATUS) & 0x2) == 0) ;

  reg_write32(GCD_X, x);
  reg_write32(GCD_Y, y);


  // wait for peripheral to complete
```

(continues on next page)

(continued from previous page)

```
  while ((reg_read8(GCD_STATUS) & 0x1) == 0) ;

  result = reg_read32(GCD_GCD);
  ref = gcd_ref(x, y);

  if (result != ref) {
    printf("Hardware result %d does not match reference value %d\n", result, ref);
    return 1;
  }
  return 0;
}
```

Support for Verilog Within Chipyard Tool Flows

There are important differences in how Verilog blackboxes are treated by various flows within the Chipyard framework. Some flows within Chipyard rely on FIRRTL in order to provide robust, non-invasive transformations of source code. Since Verilog blackboxes remain blackboxes in FIRRTL, their ability to be processed by FIRRTL transforms is limited, and some advanced features of Chipyard may provide weaker support for blackboxes. Note that the remainder of the design (the "non-Verilog" part of the design) may still generally be transformed or augmented by any Chipyard FIRRTL transform.

- Verilog blackboxes are fully supported for generating tapeout-ready RTL
- HAMMER workflows offer robust support for integrating Verilog blackboxes
- FireSim relies on FIRRTL transformations to generate a decoupled FPGA simulator. Therefore, support for Verilog blackboxes in FireSim is currently limited but rapidly evolving. Stay tuned!
- Custom FIRRTL transformations and analyses may sometimes be able to handle blackbox Verilog, depending on the mechanism of the particular transform

As mentioned earlier in this section, `BlackBox` resource files must be integrated into the build process, so any project providing `BlackBox` resources must be made visible to the `tapeout` project in `build.sbt`

2.6.11 Memory Hierarchy

The L1 Caches

Each CPU tile has an L1 instruction cache and L1 data cache. The size and associativity of these caches can be configured. The default `RocketConfig` uses 16 KiB, 4-way set-associative instruction and data caches. However, if you use the `WithNMedCores` or `WithNSmallCores` configurations, you can configure 4 KiB direct-mapped caches for L1I and L1D.

If you only want to change the size or associativity, there are config fragments for those too. See *Config Fragments* for how to add these to a custom `Config`.

```
new freechips.rocketchip.subsystem.WithL1ICacheSets(128) ++  // change rocket I$
new freechips.rocketchip.subsystem.WithL1ICacheWays(2) ++    // change rocket I$
new freechips.rocketchip.subsystem.WithL1DCacheSets(128) ++  // change rocket D$
new freechips.rocketchip.subsystem.WithL1DCacheWays(2) ++    // change rocket D$
```

You can also configure the L1 data cache as an data scratchpad instead. However, there are some limitations on this. If you are using a data scratchpad, you can only use a single core and you cannot give the design an external DRAM. Note that these configurations fully remove the L2 cache and mbus.

```
class ScratchpadOnlyRocketConfig extends Config(
  new testchipip.WithSerialPBusMem ++
  new chipyard.config.WithL2TLBs(0) ++
  new freechips.rocketchip.subsystem.WithNBanks(0) ++
  new freechips.rocketchip.subsystem.WithNoMemPort ++           // remove offchip mem␣
↪port
  new freechips.rocketchip.subsystem.WithScratchpadsOnly ++     // use rocket l1␣
↪DCache scratchpad as base phys mem
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

This configuration fully removes the L2 cache and memory bus by setting the number of channels and number of banks to 0.

The System Bus

The system bus is the TileLink network that sits between the tiles and the L2 agents and MMIO peripherals. Ordinarily, it is a fully-connected crossbar, but TestChipIP provides a version that uses a ring network instead. This can be useful when taping out larger systems. To use the ring network system bus, simply add the `WithRingSystemBus` config fragment to your configuration.

```
class RingSystemBusRocketConfig extends Config(
  new testchipip.WithRingSystemBus ++                          // Ring-topology system␣
↪bus
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

The SiFive L2 Cache

The default `RocketConfig` provided in the Chipyard example project uses SiFive's InclusiveCache generator to produce a shared L2 cache. In the default configuration, the L2 uses a single cache bank with 512 KiB capacity and 8-way set-associativity. However, you can change these parameters to obtain your desired cache configuration. The main restriction is that the number of ways and the number of banks must be powers of 2.

Refer to the `CacheParameters` object defined in sifive-cache for customization options.

The Broadcast Hub

If you do not want to use the L2 cache (say, for a resource-limited embedded design), you can create a configuration without it. Instead of using the L2 cache, you will instead use RocketChip's TileLink broadcast hub. To make such a configuration, you can just copy the definition of `RocketConfig` but omit the `WithInclusiveCache` config fragment from the list of included mixims.

If you want to reduce the resources used even further, you can configure the Broadcast Hub to use a bufferless design. This config fragment is `freechips.rocketchip.subsystem.WithBufferlessBroadcastHub`.

The Outer Memory System

The L2 coherence agent (either L2 cache or Broadcast Hub) makes requests to an outer memory system consisting of an AXI4-compatible DRAM controller.

The default configuration uses a single memory channel, but you can configure the system to use multiple channels. As with the number of L2 banks, the number of DRAM channels is restricted to powers of two.

```
new freechips.rocketchip.subsystem.WithNMemoryChannels(2)
```

In VCS and Verilator simulation, the DRAM is simulated using the `SimAXIMem` module, which simply attaches a single-cycle SRAM to each memory channel.

Instead of connecting to off-chip DRAM, you can instead connect a scratchpad and remove the off-chip link. This is done by adding a fragment like `testchipip.WithBackingScratchpad` to your configuration and removing the memory port with `freechips.rocketchip.subsystem.WithNoMemPort`.

```
class MbusScratchpadRocketConfig extends Config(
  new testchipip.WithBackingScratchpad ++                    // add mbus backing␣
↪scratchpad
  new freechips.rocketchip.subsystem.WithNoMemPort ++        // remove offchip mem port
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

If you want a more realistic memory simulation, you can use FireSim, which can simulate the timing of DDR3 controllers. More documentation on FireSim memory models is available in the FireSim docs.

2.6.12 Chipyard Boot Process

This section will describe in detail the process by which a Chipyard-based SoC boots a Linux kernel and the changes you can make to customize this process.

BootROM and RISC-V Frontend Server

The BootROM contains both the first instructions to run when the SoC is powered on as well as the Device Tree Binary (dtb) which details the components of the system. The assembly for the BootROM code is located in generators/testchipip/src/main/resources/testchipip/bootrom/bootrom.S. The BootROM address space starts at `0x10000` (determined by the `BootROMParams` key in the configuration) and execution starts at address `0x10040` (given by the linker script and reset vector in the `BootROMParams`), which is marked by the `_hang` label in the BootROM assembly.

The Chisel generator encodes the assembled instructions into the BootROM hardware at elaboration time, so if you want to change the BootROM code, you will need to run `make` in the bootrom directory and then regenerate the Verilog. If you don't want to overwrite the existing `bootrom.S`, you can also point the generator to a different bootrom image by overriding the `BootROMParams` key in the configuration.

```
class WithMyBootROM extends Config((site, here, up) => {
  case BootROMParams =>
    BootROMParams(contentFileName = "/path/to/your/bootrom.img")
})
```

The default bootloader simply loops on a wait-for-interrupt (WFI) instruction as the RISC-V frontend-server (FESVR) loads the actual program. FESVR is a program that runs on the host CPU and can read/write arbitrary parts of the target system memory using the Tethered Serial Interface (TSI).

FESVR uses TSI to load a baremetal executable or second-stage bootloader into the SoC memory. In *Software RTL Simulation*, this will be the binary you pass to the simulator. Once it is finished loading the program, FESVR will write to the software interrupt register for CPU 0, which will bring CPU 0 out of its WFI loop. Once it receives the interrupt, CPU 0 will write to the software interrupt registers for the other CPUs in the system and then jump to the beginning of DRAM to execute the first instruction of the loaded executable. The other CPUs will be woken up by the first CPU and also jump to the beginning of DRAM.

The executable loaded by FESVR should have memory locations designated as *tohost* and *fromhost*. FESVR uses these memory locations to communicate with the executable once it is running. The executable uses *tohost* to send commands to FESVR for things like printing to the console, proxying system calls, and shutting down the SoC. The *fromhost* register is used to send back responses for *tohost* commands and for sending console input.

The Berkeley Boot Loader and RISC-V Linux

For baremetal programs, the story ends here. The loaded executable will run in machine mode until it sends a command through the *tohost* register telling the FESVR to power off the SoC.

However, for booting the Linux Kernel, you will need to use a second-stage bootloader called the Berkeley Boot Loader, or BBL. This program reads the device tree encoded in the boot ROM and transforms it into a format compatible with the Linux kernel. It then sets up virtual memory and the interrupt controller, loads the kernel, which is embedded in the bootloader binary as a payload, and starts executing the kernel in supervisor mode. The bootloader is also responsible for servicing machine-mode traps from the kernel and proxying them over FESVR.

Once BBL jumps into supervisor mode, the Linux kernel takes over and begins its process. It eventually loads the `init` program and runs it in user mode, thus starting userspace execution.

The easiest way to build a BBL image that boots Linux is to use the FireMarshal tool that lives in the firesim-software repository. Directions on how to use FireMarshal can be found in the FireSim documentation. Using FireMarshal, you can add custom kernel configurations and userspace software to your workload.

2.6.13 Adding a Firrtl Transform

Similar to how LLVM IR passes can perform transformations and optimizations on software, FIRRTL transforms can modify Chisel-elaborated RTL. As mentioned in Section *FIRRTL*, transforms are modifications that happen on the FIRRTL IR that can modify a circuit. Transforms are a powerful tool to take in the FIRRTL IR that is emitted from Chisel and run analysis or convert the circuit into a new form.

Where to add transforms

In Chipyard, the FIRRTL compiler is called multiple times to create a "Top" file that contains the DUT and a "Harness" file containing the test harness, which instantiates the DUT. The "Harness" file does not contain the DUT's module definition or any of its submodules. This is done by the `tapeout` SBT project (located in `tools/barstools/tapeout`) which calls `GenerateTopAndHarness` (a function that wraps the multiple FIRRTL compiler calls and extra transforms).

```
# NOTE: These *_temp intermediate targets will get removed in favor of make 4.3␣
↪grouped targets (&: operator)
.INTERMEDIATE: firrtl_temp
$(TOP_TARGETS) $(HARNESS_TARGETS): firrtl_temp
        @echo "" > /dev/null

firrtl_temp: $(FIRRTL_FILE) $(ANNO_FILE) $(VLOG_SOURCES)
        $(call run_scala_main,tapeout,barstools.tapeout.transforms.
↪GenerateTopAndHarness,\
                --allow-unrecognized-annotations \
                --output-file $(TOP_FILE) \
                --harness-o $(HARNESS_FILE) \
                --input-file $(FIRRTL_FILE) \
                --syn-top $(TOP) \
                --harness-top $(VLOG_MODEL) \
                --annotation-file $(ANNO_FILE) \
```

(continues on next page)

(continued from previous page)

```
                --top-anno-out $(TOP_ANNO) \
                --top-dotf-out $(sim_top_blackboxes) \
                --top-fir $(TOP_FIR) \
                --harness-anno-out $(HARNESS_ANNO) \
                --harness-dotf-out $(sim_harness_blackboxes) \
                --harness-fir $(HARNESS_FIR) \
                $(REPL_SEQ_MEM) \
                $(HARNESS_CONF_FLAGS) \
                --target-dir $(build_dir) \
                --log-level $(FIRRTL_LOGLEVEL) \
                $(EXTRA_FIRRTL_OPTIONS))
        touch $(sim_top_blackboxes) $(sim_harness_blackboxes)
```

If you look inside of the tools/barstools/tapeout/src/main/scala/transforms/Generate.scala file, you can see that FIRRTL is invoked twice, once for the "Top" and once for the "Harness". If you want to add transforms to just modify the DUT, you can add them to `topTransforms`. Otherwise, if you want to add transforms to just modify the test harness, you can add them to `harnessTransforms`.

For more information on Barstools, please visit the *Barstools* section.

Examples of transforms

There are multiple examples of transforms that you can apply and are spread across the FIRRTL ecosystem. Within FIRRTL there is a default set of supported transforms located in https://github.com/freechipsproject/firrtl/tree/master/src/main/scala/firrtl/transforms. This includes transforms that can flatten modules (`Flatten`), group modules together (`GroupAndDedup`), and more.

Transforms can be standalone or can take annotations as input. Annotations are used to pass information between FIRRTL transforms. This includes information on what modules to flatten, group, and more. Annotations can be added to the code by adding them to your Chisel source or by creating a serialized annotation `json` file and adding it to the FIRRTL compiler (note: annotating the Chisel source will automatically serialize the annotation as a `json` snippet into the build system for you). **The recommended way to annotate something is to do it in the Chisel source, but not all annotation types have Chisel APIs**.

The example below shows two ways to annotate the signal using the `DontTouchAnnotation` (makes sure that a particular signal is not removed by the "Dead Code Elimination" pass in FIRRTL):

- use the Chisel API/wrapper function called `dontTouch` that does this automatically for you (more dontTouch information):
- directly annotate the signal with the `annotate` function and the `DontTouchAnnotation` class if there is no Chisel API for it (note: most FIRRTL annotations have Chisel APIs for them)

```
class TopModule extends Module {
    ...
    val submod = Module(new Submodule)
    ...
}

class Submodule extends Module {
    ...
    val some_signal := ...

    // MAIN WAY TO USE `dontTouch`
    // how to annotate if there is a Chisel API/wrapper
    chisel3.dontTouch(some_signal)
```

(continues on next page)

(continued from previous page)

```
    // how to annotate WITHOUT a Chisel API/wrapper
    annotate(new ChiselAnnotation {
        def toFirrtl = DontTouchAnnotation(some_signal.toNamed)
    })

    ...
}
```

Here is an example of the `DontTouchAnnotation` when it is serialized:

```
[
    {
        "class": "firrtl.transforms.DontTouchAnnotation",
        "target": "~TopModule|Submodule>some_signal"
    }
]
```

In this case, the specific syntax depends on the type of annotation and its fields. One of the easier ways to figure out the serialized syntax is to first try and find a Chisel annotation to add to the code. Then you can look at the collateral that is generated from the build system, find the `*.anno.json`, and find the proper syntax for the annotation.

Once `yourAnnoFile.json` is created then you can add `-faf yourAnnoFile.json` to the FIRRTL compiler invocation in `common.mk`.

```
# NOTE: These *_temp intermediate targets will get removed in favor of make 4.3␣
↪grouped targets (&: operator)
.INTERMEDIATE: firrtl_temp
$(TOP_TARGETS) $(HARNESS_TARGETS): firrtl_temp
        @echo "" > /dev/null

firrtl_temp: $(FIRRTL_FILE) $(ANNO_FILE) $(VLOG_SOURCES)
        $(call run_scala_main,tapeout,barstools.tapeout.transforms.
↪GenerateTopAndHarness,\
                --allow-unrecognized-annotations \
                --output-file $(TOP_FILE) \
                --harness-o $(HARNESS_FILE) \
                --input-file $(FIRRTL_FILE) \
                --syn-top $(TOP) \
                --harness-top $(VLOG_MODEL) \
                --annotation-file $(ANNO_FILE) \
                --top-anno-out $(TOP_ANNO) \
                --top-dotf-out $(sim_top_blackboxes) \
                --top-fir $(TOP_FIR) \
                --harness-anno-out $(HARNESS_ANNO) \
                --harness-dotf-out $(sim_harness_blackboxes) \
                --harness-fir $(HARNESS_FIR) \
                $(REPL_SEQ_MEM) \
                $(HARNESS_CONF_FLAGS) \
                --target-dir $(build_dir) \
                --log-level $(FIRRTL_LOGLEVEL) \
                $(EXTRA_FIRRTL_OPTIONS))
        touch $(sim_top_blackboxes) $(sim_harness_blackboxes)
```

If you are interested in writing FIRRTL transforms please refer to the FIRRTL documentation located here: https://github.com/freechipsproject/firrtl/wiki.

2.6.14 IOBinders and HarnessBinders

In Chipyard we use special `Parameters` keys, `IOBinders` and `HarnessBinders` to bridge the gap between digital system IOs and TestHarness collateral.

IOBinders

The `IOBinder` functions are responsible for instantiating IO cells and IOPorts in the `ChipTop` layer.

`IOBinders` are typically defined using the `OverrideIOBinder` or `ComposeIOBinder` macros. An `IOBinder` consists of a function matching `Systems` with a given trait that generates IO ports and IOCells, and returns a list of generated ports and cells.

For example, the `WithUARTIOCells` IOBinder will, for any `System` that might have UART ports (`HasPeripheryUARTModuleImp`, generate ports within the `ChipTop` (`ports`) as well as IOCells with the appropriate type and direction (`cells2d`). This function returns a the list of generated ports, and the list of generated IOCells. The list of generated ports is passed to the `HarnessBinders` such that they can be connected to `TestHarness` devices.

```
class WithUARTIOCells extends OverrideIOBinder({
  (system: HasPeripheryUARTModuleImp) => {
    val (ports: Seq[UARTPortIO], cells2d) = system.uart.zipWithIndex.map({ case (u,␣
↪i) =>
      val (port, ios) = IOCell.generateIOFromSignal(u, s"uart_${i}", system.
↪p(IOCellKey), abstractResetAsAsync = true)
      (port, ios)
    }).unzip
    (ports, cells2d.flatten)
  }
})
```

HarnessBinders

The `HarnessBinder` functions determine what modules to bind to the IOs of a `ChipTop` in the `TestHarness`. The `HarnessBinder` interface is designed to be reused across various simulation/implementation modes, enabling decoupling of the target design from simulation and testing concerns.

- For SW RTL or GL simulations, the default set of `HarnessBinders` instantiate software-simulated models of various devices, for example external memory or UART, and connect those models to the IOs of the `ChipTop`.
- For FireSim simulations, FireSim-specific `HarnessBinders` instantiate `Bridges`, which faciliate cycle-accurate simulation across the simulated chip's IOs. See the FireSim documentation for more details.
- In the future, a Chipyard FPGA prototyping flow may use `HarnessBinders` to connect `ChipTop` IOs to other devices or IOs in the FPGA harness.

Like `IOBinders`, `HarnessBinders` are defined using macros (`OverrideHarnessBinder`, `ComposeHarnessBinder`), and match `Systems` with a given trait. However, `HarnessBinders` are also passed a reference to the `TestHarness` (`th: HasHarnessSignalReferences`) and the list of ports generated by the corresponding `IOBinder` (`ports: Seq[Data]`).

For exmaple, the `WithUARTAdapter` will connect the UART SW display adapter to the ports generated by the `WithUARTIOCells` described earlier, if those ports are present.

```
class WithUARTAdapter extends OverrideHarnessBinder({
  (system: HasPeripheryUARTModuleImp, th: HasHarnessSignalReferences, ports:␣
↪Seq[UARTPortIO]) => {
```

(continues on next page)

(continued from previous page)

```
    UARTAdapter.connect(ports)(system.p)
  }
})
```

The `IOBinder` and `HarnessBinder` system is designed to enable decoupling of concerns between the target design and the simulation system.

For a given set of chip IOs, there may be not only multiple simulation platforms ("harnesses", so-to-speak), but also multiple simulation strategies. For example, the choice of whether to connect the backing AXI4 memory port to an accurate DRAM model (`SimDRAM`) or a simple simulated memory model (`SimAXIMem`) is isolated in `HarnessBinders`, and does not affect target RTL generation.

Similarly, for a given simulation platform and strategy, there may be multiple strategies for generating the chip IOs. This target-design configuration is isolated in the `IOBinders`.

2.7 Target Software

Chipyard includes tools for developing target software workloads. The primary tool is FireMarshal, which manages workload descriptions and generates binaries and disk images to run on your target designs. Workloads can be bare-metal, or be based on standard Linux distributions. Users can customize every part of the build process, including providing custom kernels (if needed by the hardware).

FireMarshal can also run your workloads on high-performance functional simulators like Spike and Qemu. Spike is easily customized and serves as the official RISC-V ISA reference implementation. Qemu is a high-performance functional simulator that can run nearly as fast as native code, but can be challenging to modify.

To initialize additional software repositories, such as wrappers for Coremark, SPEC2017, and workloads for the NVDLA, run the following script. The submodules are located in the `software` directory.

```
./scripts/init-software.sh
```

2.7.1 FireMarshal

FireMarshal is a workload generation tool for RISC-V based systems. It currently only supports the FireSim FPGA-accelerated simulation platform.

Workloads in FireMarshal consist of a series of **Jobs** that are assigned to logical nodes in the target system. If no jobs are specified, then the workload is considered `uniform` and only a single image will be produced for all nodes in the system. Workloads are described by a `json` file and a corresponding workload directory and can inherit their definitions from existing workloads. Typically, workload configurations are kept in `workloads/` although you can use any directory you like. We provide a few basic workloads to start with including buildroot or Fedora-based linux distributions and bare-metal.

Once you define a workload, the `marshal` command will produce a corresponding boot-binary and rootfs for each job in the workload. This binary and rootfs can then be launched on qemu or spike (for functional simulation), or installed to a platform for running on real RTL (currently only FireSim is automated).

To get started, checkout the full FireMarshal documentation.

2.7.2 The RISC-V ISA Simulator (Spike)

Spike is the golden reference functional RISC-V ISA C++ sofware simulator. It provides full system emulation or proxied emulation with HTIF/FESVR. It serves as a starting point for running software on a RISC-V target. Here is a

highlight of some of Spikes main features:

- Multiple ISAs: RV32IMAFDQCV extensions
- Multiple memory models: Weak Memory Ordering (WMO) and Total Store Ordering (TSO)
- Privileged Spec: Machine, Supervisor, User modes (v1.11)
- Debug Spec
- Single-step debugging with support for viewing memory/register contents
- Multiple CPU support
- JTAG support
- Highly extensible (add and test new instructions)

In most cases, software development for a Chipyard target will begin with functional simulation using Spike (usually with the addition of custom Spike models for custom accelerator functions), and only later move on to full cycle-accurate simulation using software RTL simulators or FireSim.

Spike comes pre-packaged in the RISC-V toolchain and is available on the path as `spike`. More information can be found in the Spike repository.

2.7.3 Baremetal RISC-V Programs

To build baremetal RISC-V programs to run in simulation, we use the riscv64-unknown-elf cross-compiler and a fork of the libgloss board support package. To build such a program yourself, simply invoke the cross-compiler with the flags "-fno-common -fno-builtin-printf -specs=htif_nano.specs" and the link with the arguments "-static -specs=htif_nano.specs". For instance, if we want to run a "Hello, World" program in baremetal, we could do the following.

```
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

```
$ riscv64-unknown-elf-gcc -fno-common -fno-builtin-printf -specs=htif_nano.specs -c␣
↪hello.c
$ riscv64-unknown-elf-gcc -static -specs=htif_nano.specs hello.o -o hello.riscv
$ spike hello.riscv
Hello, World!
```

For more examples, look at the tests/ directory in the chipyard repository.

For more information about the libgloss port, take a look at its README.

2.8 Advanced Concepts

The following sections are advanced topics about how to Chipyard works, how to use Chipyard, and special features of the framework. They expect you to know about Chisel, Parameters, configs, etc.

2.8.1 Tops, Test-Harnesses, and the Test-Driver

The three highest levels of hierarchy in a Chipyard SoC are the `ChipTop` (DUT), `TestHarness`, and the `TestDriver`. The `ChipTop` and `TestHarness` are both emitted by Chisel generators. The `TestDriver` serves as our testbench, and is a Verilog file in Rocket Chip.

ChipTop/DUT

`ChipTop` is the top-level module that instantiates the `System` submodule, usually an instance of the concrete class `DigitalTop`. The vast majority of the design resides in the `System`. Other components that exist inside the `ChipTop` layer are generally IO cells, clock receivers and multiplexers, reset synchronizers, and other analog IP that needs to exist outside of the `System`. The `IOBinders` are responsible for instantiating the IO cells for `ChipTop` IO that correspond to IO of the `System`. The `HarnessBinders` are responsible for instantiating test harness collateral that connects to the `ChipTop` ports. Most types of devices and testing collateral can be instantiated using custom `IOBinders` and `HarnessBinders`.

System/DigitalTop

The system module of a Rocket Chip SoC is composed via cake-pattern. Specifically, `DigitalTop` extends a `System`, which extends a `Subsystem`, which extends a `BaseSubsystem`.

BaseSubsystem

The `BaseSubsystem` is defined in `generators/rocketchip/src/main/scala/subsystem/BaseSubsystem.scala`. Looking at the `BaseSubsystem` abstract class, we see that this class instantiates the top-level buses (frontbus, systembus, peripherybus, etc.), but does not specify a topology. We also see this class define several `ElaborationArtefacts`, files emitted after Chisel elaboration (e.g. the device tree string, and the diplomacy graph visualization GraphML file).

Subsystem

Looking in generators/chipyard/src/main/scala/Subsystem.scala, we can see how Chipyard's `Subsystem` extends the `BaseSubsystem` abstract class. `Subsystem` mixes in the `HasBoomAndRocketTiles` trait that defines and instantiates BOOM or Rocket tiles, depending on the parameters specified. We also connect some basic IOs for each tile here, specifically the hartids and the reset vector.

System

`generators/chipyard/src/main/scala/System.scala` completes the definition of the `System`.

- `HasHierarchicalBusTopology` is defined in Rocket Chip, and specifies connections between the top-level buses
- `HasAsyncExtInterrupts` and `HasExtInterruptsModuleImp` adds IOs for external interrupts and wires them appropriately to tiles
- `CanHave...AXI4Port` adds various Master and Slave AXI4 ports, adds TL-to-AXI4 converters, and connects them to the appropriate buses
- `HasPeripheryBootROM` adds a BootROM device

Tops

A SoC Top then extends the `System` class with traits for custom components. In Chipyard, this includes things like adding a NIC, UART, and GPIO as well as setting up the hardware for the bringup method. Please refer to *Communicating with the DUT* for more information on these bringup methods.

TestHarness

The wiring between the `TestHarness` and the Top are performed in methods defined in traits added to the Top. When these methods are called from the `TestHarness`, they may instantiate modules within the scope of the harness, and then connect them to the DUT. For example, the `connectSimAXIMem` method defined in the `CanHaveMasterAXI4MemPortModuleImp` trait, when called from the `TestHarness`, will instantiate `SimAXIMems` and connect them to the correct IOs of the top.

While this roundabout way of attaching to the IOs of the top may seem to be unnecessarily complex, it allows the designer to compose custom traits together without having to worry about the details of the implementation of any particular trait.

TestDriver

The `TestDriver` is defined in `generators/rocketchip/src/main/resources/vsrc/TestDriver.v`. This Verilog file executes a simulation by instantiating the `TestHarness`, driving the clock and reset signals, and interpreting the success output. This file is compiled with the generated Verilog for the `TestHarness` and the `Top` to produce a simulator.

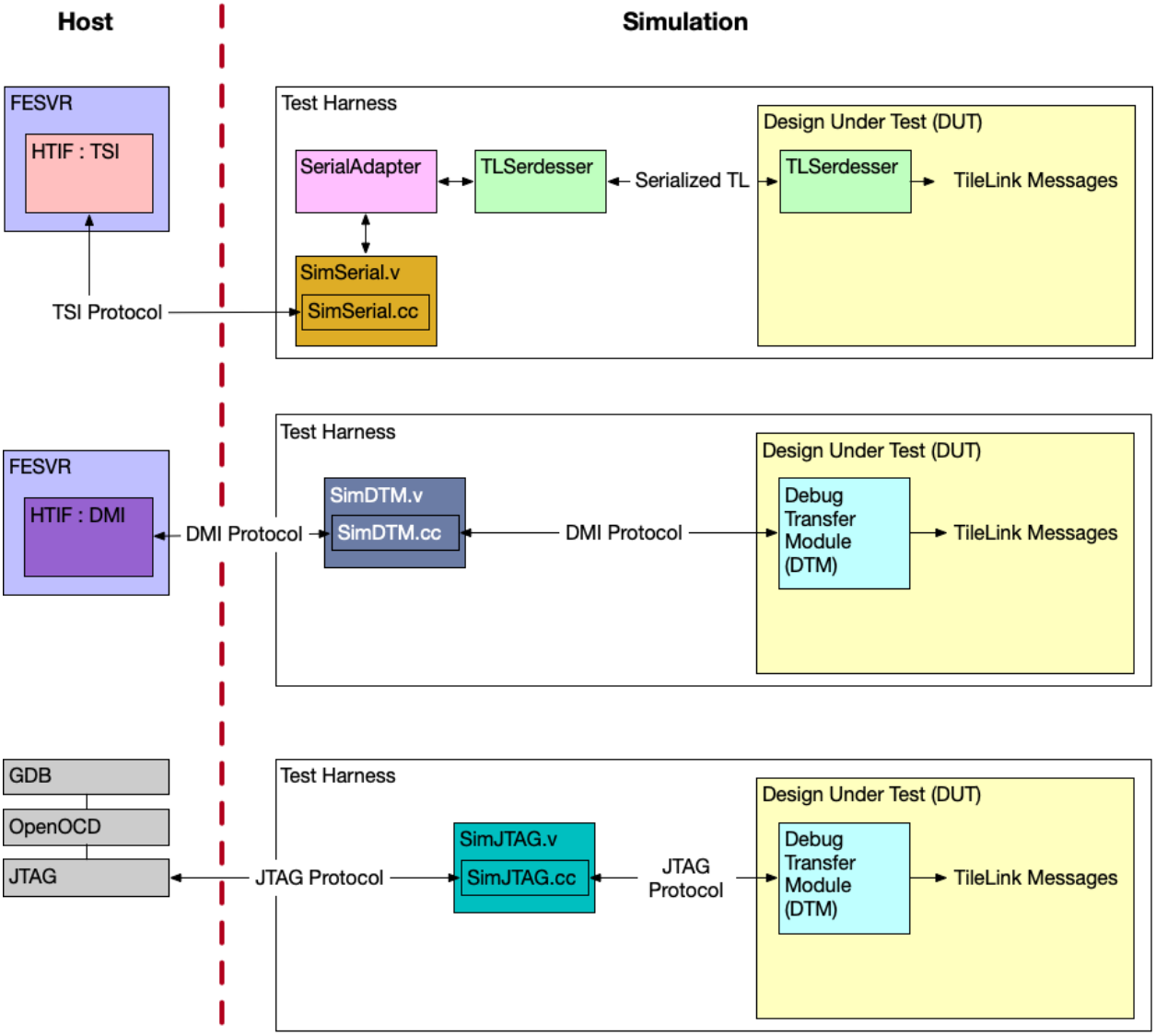
2.8.2 Communicating with the DUT

There are two types of DUTs that can be made: *tethered* or *standalone* DUTs. A *tethered* DUT is where a host computer (or just host) must send transactions to the DUT to bringup a program. This differs from a *standalone* DUT that can bringup itself (has its own bootrom, loads programs itself, etc). An example of a tethered DUT is a Chipyard simulation where the host loads the test program into the DUTs memory and signals to the DUT that the program is ready to run. An example of a standalone DUT is a Chipyard simulation where a program can be loaded from an SDCard out of reset. In this section, we mainly describe how to communicate to tethered DUTs.

There are two ways the host (otherwise known as the outside world) can communicate with a tethered Chipyard DUT:

- Using the Tethered Serial Interface (TSI) or the Debug Module Interface (DMI) with the Front-End Server (FESVR) to communicate with the DUT
- Using the JTAG interface with OpenOCD and GDB to communicate with the DUT

The following picture shows a block diagram view of all the supported communication mechanisms split between the host and the simulation.



Using the Tethered Serial Interface (TSI) or the Debug Module Interface (DMI)

If you are using TSI or DMI to communicate with the DUT, you are using the Front-End Server (FESVR) to facilitate communication between the host and the DUT.

Primer on the Front-End Server (FESVR)

FESVR is a C++ library that manages communication between a host machine and a RISC-V DUT. For debugging, it provides a simple API to reset, send messages, and load/run programs on a DUT. It also emulates peripheral devices. It can be incorporated with simulators (VCS, Verilator, FireSim), or used in a bringup sequence for a taped out chip.

Specifically, FESVR uses the Host Target Interface (HTIF), a communication protocol, to speak with the DUT. HTIF is a non-standard Berkeley protocol that uses a FIFO non-blocking interface to communicate with the DUT. It defines a protocol where you can read/write memory, load/start/stop the program, and more. Both TSI and DMI implement this HTIF protocol differently in order to communicate with the DUT.

Using the Tethered Serial Interface (TSI)

By default, Chipyard uses the Tethered Serial Interface (TSI) to communicate with the DUT. TSI protocol is an implementation of HTIF that is used to send commands to the RISC-V DUT. These TSI commands are simple R/W commands that are able to access the DUT's memory space. During simulation, the host sends TSI commands to a simulation stub in the test harness called `SimSerial` (C++ class) that resides in a `SimSerial` Verilog module (both are located in the `generators/testchipip` project). This `SimSerial` Verilog module then sends the TSI command recieved by the simulation stub to an adapter that converts the TSI command into a TileLink request. This conversion is done by the `SerialAdapter` module (located in the `generators/testchipip` project). After the transaction is converted to TileLink, the `TLSerdesser` (located in `generators/testchipip`) serializes the transaction and sends it to the chip (this `TLSerdesser` is sometimes also referred to as a digital serial-link or SerDes). Once the serialized transaction is received on the chip, it is deserialized and masters a TileLink bus on the chip which handles the request. In simulation, FESVR resets the DUT, writes into memory the test program, and indicates to the DUT to start the program through an interrupt (see *Chipyard Boot Process*). Using TSI is currently the fastest mechanism to communicate with the DUT in simulation (compared to DMI/JTAG) and is also used by FireSim.

Using the Debug Module Interface (DMI)

Another option to interface with the DUT is to use the Debug Module Interface (DMI). Similar to TSI, the DMI protocol is an implementation of HTIF. In order to communicate with the DUT with the DMI protocol, the DUT needs to contain a Debug Transfer Module (DTM). The DTM is given in the RISC-V Debug Specification and is responsible for managing communication between the DUT and whatever lives on the other side of the DMI (in this case FESVR). This is implemented in the Rocket Chip `Subsystem` by having the `HasPeripheryDebug` and `HasPeripheryDebugModuleImp` traits. During simulation, the host sends DMI commands to a simulation stub called `SimDTM` (C++ class) that resides in a `SimDTM` Verilog module (both are located in the `generators/rocket-chip` project). This `SimDTM` Verilog module then sends the DMI command recieved by the simulation stub into the DUT which then converts the DMI command into a TileLink request. This conversion is done by the DTM named `DebugModule` in the `generators/rocket-chip` project. When the DTM receives the program to load, it starts to write the binary byte-wise into memory. This is considerably slower than the TSI protocol communication pipeline (i.e. `SimSerial`/`SerialAdapter`/TileLink) which directly writes the program binary to memory.

Starting the TSI or DMI Simulation

All default Chipyard configurations use TSI to communicate between the simulation and the simulated SoC/DUT. Hence, when running a software RTL simulation, as is indicated in the *Software RTL Simulation* section, you are in-fact using TSI to communicate with the DUT. As a reminder, to run a software RTL simulation, run:

```
cd sims/verilator
# or
cd sims/vcs

make CONFIG=RocketConfig run-asm-tests
```

If you would like to build and simulate a Chipyard configuration with a DTM configured for DMI communication, then you must tie-off the serial-link interface, and instantiate the *SimDTM*.

```
class dmiRocketConfig extends Config(
  new chipyard.harness.WithSerialAdapterTiedOff ++                // don't attach an␣
↪external SimSerial
  new chipyard.config.WithDMIDTM ++                               // have debug module␣
↪expose a clocked DMI port
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

Then you can run simulations with the new DMI-enabled top-level and test-harness.

```
cd sims/verilator
# or
cd sims/vcs

make CONFIG=dmiRocketConfig run-asm-tests
```

Using the JTAG Interface

Another way to interface with the DUT is to use JTAG. Similar to the *Using the Debug Module Interface (DMI)* section, in order to use the JTAG protocol, the DUT needs to contain a Debug Transfer Module (DTM) configured to use JTAG instead of DMI. Once the JTAG port is exposed, the host can communicate over JTAG to the DUT through a simulation stub called `SimJTAG` (C++ class) that resides in a `SimJTAG` Verilog module (both reside in the `generators/rocket-chip` project). This simulation stub creates a socket that OpenOCD and GDB can connect to when the simulation is running. The default Chipyard designs instantiate the DTM configured to use JTAG (i.e. `RocketConfig`).

Note: As mentioned, default Chipyard designs are enabled with JTAG. However, they also use TSI/Serialized-TL with FESVR in case the JTAG interface isn't used. This allows users to choose how to communicate with the DUT (use TSI or JTAG).

Debugging with JTAG

Roughly the steps to debug with JTAG in simulation are as follows:

1. Build a Chipyard JTAG-enabled RTL design. Remember default Chipyard designs are JTAG ready.

```
cd sims/verilator
# or
cd sims/vcs

make CONFIG=RocketConfig
```

2. Run the simulation with remote bit-bang enabled. Since we hope to load/run the binary using JTAG, we can pass `none` as a binary (prevents FESVR from loading the program). (Adapted from: https://github.com/chipsalliance/rocket-chip#3-launch-the-emulator)

```
# note: this uses Chipyard make invocation to run the simulation to properly wrap the␣
↪simulation args
make CONFIG=RocketConfig BINARY=none SIM_FLAGS="+jtag_rbb_enable=1 --rbb-port=9823"␣
↪run-binary
```

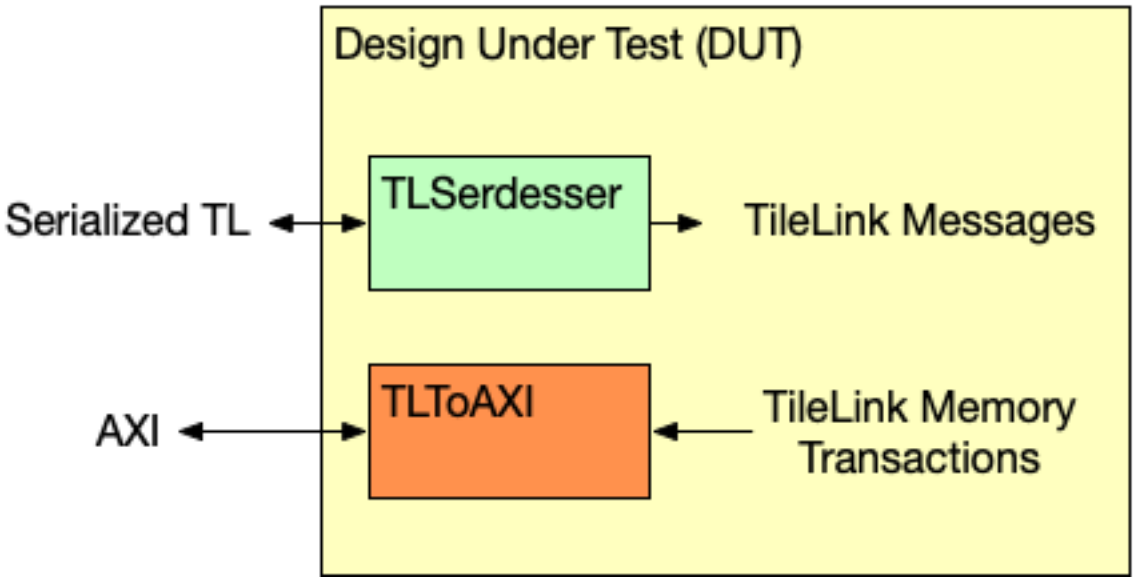
3. Follow the instructions here to connect to the simulation using OpenOCD + GDB.

Note: This section was adapted from the instruction in Rocket Chip and riscv-isa-sim. For more information refer to that documentation: Rocket Chip GDB Docs, riscv-isa-sim GDB Docs

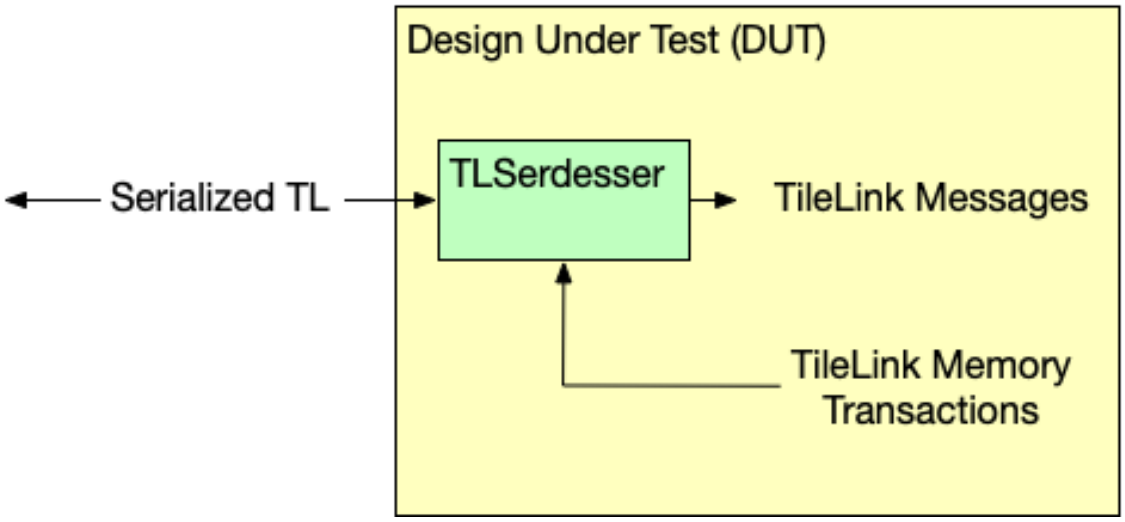
Example Test Chip Bringup Communication

Intro to Typical Chipyard Test Chip

Most, if not all, Chipyard configurations are tethered using TSI (over a serial-link) and have access to external memory through an AXI port (backing AXI memory). The following image shows the DUT with these set of default signals:

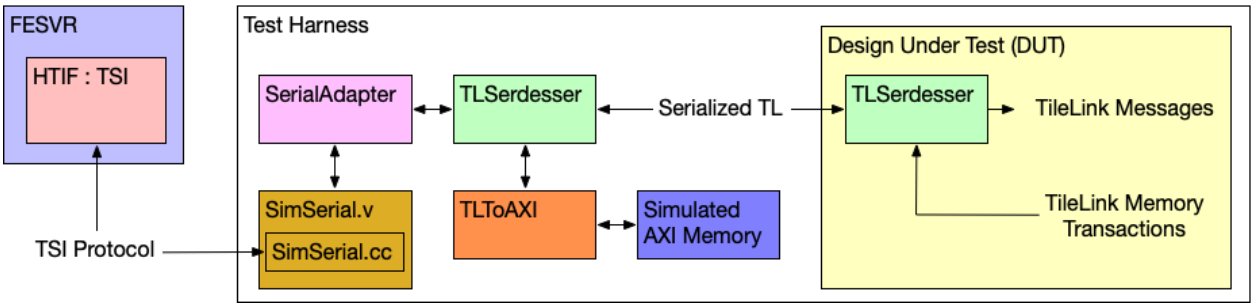


In this setup, the serial-link is connected to the TSI/FESVR peripherals while the AXI port is connected to a simulated AXI memory. However, AXI ports tend to have many signals, and thus wires, associated with them so instead of creating an AXI port off the DUT, one can send the memory transactions over the bi-directional serial-link (`TLSerdesser`) so that the main interface to the DUT is the serial-link (which has comparatively less signals than an AXI port). This new setup (shown below) is a typical Chipyard test chip setup:



Simulation Setup of the Example Test Chip

To test this type of configuration (TSI/memory transactions over the serial-link), most of the same TSI collateral would be used. The main difference is that the TileLink-to-AXI converters and simulated AXI memory resides on the other side of the serial-link.



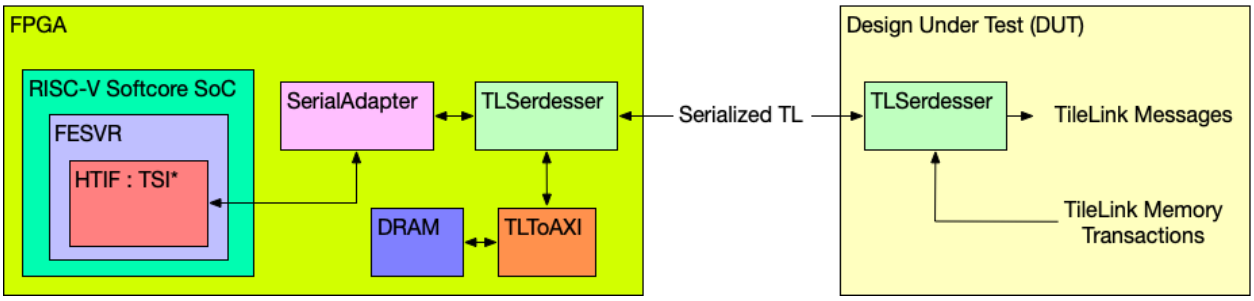
Note: Here the simulated AXI memory and the converters can be in a different clock domain in the test harness than the reference clock of the DUT. For example, the DUT can be clocked at 3.2GHz while the simulated AXI memory can be clocked at 1GHz. This functionality is done in the harness binder that instantiates the TSI collateral, TL-to-AXI converters, and simulated AXI memory. See *Creating Clocks in the Test Harness* on how to generate a clock in a harness binder.

This type of simulation setup is done in the following multi-clock configuration:

```
class MulticlockAXIOverSerialConfig extends Config(
  new chipyard.config.WithSystemBusFrequencyAsDefault ++
  new chipyard.config.WithSystemBusFrequency(250) ++
  new chipyard.config.WithPeripheryBusFrequency(250) ++
  new chipyard.config.WithMemoryBusFrequency(250) ++
  new chipyard.config.WithFrontBusFrequency(50) ++
  new chipyard.config.WithTileFrequency(500, Some(1)) ++
  new chipyard.config.WithTileFrequency(250, Some(0)) ++

  new chipyard.config.WithFbusToSbusCrossingType(AsynchronousCrossing()) ++
  new testchipip.WithAsynchronousSerialSlaveCrossing ++
  new freechips.rocketchip.subsystem.WithAsynchronousRocketTiles(
    AsynchronousCrossing().depth,
    AsynchronousCrossing().sourceSync) ++

  new chipyard.harness.WithSimAXIMemOverSerialTL ++ // add SimDRAM DRAM model for␣
↪axi4 backing memory over the SerDes link, if axi4 mem is enabled
  new chipyard.config.WithSerialTLBackingMemory ++ // remove axi4 mem port in favor␣
↪of SerialTL memory

  new freechips.rocketchip.subsystem.WithNBigCores(2) ++
  new chipyard.config.AbstractConfig)
```

Bringup Setup of the Example Test Chip after Tapeout

Assuming this example test chip is taped out and now ready to be tested, we can communicate with the chip using this serial-link. For example, a common test setup used at Berkeley to evaluate Chipyard-based test-chips includes an FPGA running a RISC-V soft-core that is able to speak to the DUT (over an FMC). This RISC-V soft-core would serve as the host of the test that will run on the DUT. This is done by the RISC-V soft-core running FESVR, sending TSI commands to a `SerialAdapter` / `TLSerdesser` programmed on the FPGA. Once the commands are converted to serialized TileLink, then they can be sent over some medium to the DUT (like an FMC cable or a set of wires connecting FPGA outputs to the DUT board). Similar to simulation, if the chip requests offchip memory, it can then send the transaction back over the serial-link. Then the request can be serviced by the FPGA DRAM. The following image shows this flow:



In fact, this exact type of bringup setup is what the following section discusses: *Introduction to the Bringup Design*.

2.8.3 Debugging RTL

While the packaged Chipyard configs and RTL have been tested to work, users will typically want to build custom chips by adding their own IP, or by modifying existing Chisel generators. Such changes might introduce bugs. This section aims to run through a typical debugging flow using Chipyard. We assume the user has a custom SoC configuration, and is trying to verify functionality by running some software test. We also assume the software has already been verified on a functional simulator, such as Spike or QEMU. This section will focus on debugging hardware.

Waveforms

The default software RTL simulators do not dump waveforms during execution. To build simulators with wave dump capabilities use must use the `debug` make target. For example:

```
make CONFIG=CustomConfig debug
```

The `run-binary-debug` rule will also automatically build a simulator, run it on a custom binary, and generate a waveform. For example, to run a test on `helloworld.riscv`, use

```
make CONFIG=CustomConfig run-binary-debug BINARY=helloworld.riscv
```

VCS and Verilator also support many additional flags. For example, specifying the `+vpdfilesize` flag in VCS will treat the output file as a circular buffer, saving disk space for long-running simulations. Refer to the VCS and Verilator manuals for more information You may use the `SIM_FLAGS` make variable to set additional simulator flags:

```
make CONFIG=CustomConfig run-binary-debug BINARY=linux.riscv SIM_
↪FLAGS=+vpdfilesize=1024
```

Note: In some cases where there is multiple simulator flags, you can write the `SIM_FLAGS` like the following: `SIM_FLAGS="+vpdfilesize=XYZ +some_other_flag=ABC"`.

Print Output

Both Rocket and BOOM can be configured with varying levels of print output. For information see the Rocket core source code, or the BOOM documentation website. In addition, developers may insert arbitrary printfs at arbitrary conditions within the Chisel generators. See the Chisel documentation for information on this.

Once the cores have been configured with the desired print statements, the `+verbose` flag will cause the simulator to print the statements. The following commands will all generate desired print statements:

```
make CONFIG=CustomConfig run-binary-debug BINARY=helloworld.riscv

# The below command does the same thing
./simv-CustomConfig-debug +verbose helloworld.riscv
```

Both cores can be configured to print out commit logs, which can then be compared against a Spike commit log to verify correctness.

Basic tests

`riscv-tests` includes basic ISA-level tests and basic benchmarks. These are used in Chipyard CI, and should be the first step in verifying a chip's functionality. The make rule is

```
make CONFIG=CustomConfig run-asm-tests run-bmark-tests
```

Torture tests

The RISC-V torture utility generates random RISC-V assembly streams, compiles them, runs them on both the Spike functional model and the SW simulator, and verifies identical program behavior. The torture utility can also be configured to run continuously for stress-testing. The torture utility exists within the `tools` directory. To run torture tests, run `make` in the simulation directories:

```
make CONFIG=CustomConfig torture
```

To run overnight tests (repeated random tests), run

```
make CONFIG=CustomConfig TORTURE_ONIGHT_OPTIONS=<overnight options> torture-overnight
```

You can find the overnight options in *overnight/src/main/scala/main.scala* in the torture repo.

Firesim Debugging

Chisel printfs, asserts, Dromajo co-simulation, and waveform generation are also available in FireSim FPGA-accelerated simulation. See the FireSim documentation for more detail.

2.8.4 Debugging BOOM

In addition to the default debugging techniques specified in *Debugging RTL*, single-core BOOM designs can utilize the Dromajo co-simulator (see *Dromajo*) to verify functionality.

Warning: Dromajo currently only works in single-core BOOM systems without accelerators.

Warning: Dromajo currently only works in VCS simulation and FireSim.

Setting up Dromajo Co-simulation

Dromajo co-simulation is setup to work when three config fragments are added to a BOOM config.

- A `chipyard.config.WithTraceIO` config fragment must be added so that BOOM's traceport is enabled.
- A `chipyard.iobinders.WithTraceIOPunchthrough` config fragment must be added to add the `TraceIO` to the `ChipTop`
- A `chipyard.harness.WithSimDromajoBridge` config fragment must be added to instantiate a Dromajo cosimulator in the `TestHarness` and connect it to the `ChipTop`'s `TraceIO`

Once all config fragments are added Dromajo should be enabled.

To build/run Dromajo with a BOOM design, run your configuration the following make commands:

```
# build the default Dromajo BOOM config without waveform dumps
# replace "DromajoBoomConfig" with your particular config
make CONFIG=DromajoBoomConfig ENABLE_DROMAJO=1

# run a simulation with Dromajo
make CONFIG=DromajoBoomConfig ENABLE_DROMAJO=1 BINARY=<YOUR-BIN> run-binary
```

2.8.5 Accessing Scala Resources

A simple way to copy over a source file to the build directory to be used for a simulation compile or VLSI flow is to use the `addResource` function given by FIRRTL. An example of its use can be seen in generators/testchipip/src/main/scala/SerialAdapter.scala. Here is the example inlined:

```
class SimSerial(w: Int) extends BlackBox with HasBlackBoxResource {
  val io = IO(new Bundle {
    val clock = Input(Clock())
    val reset = Input(Bool())
    val serial = Flipped(new SerialIO(w))
    val exit = Output(Bool())
  })

  addResource("/testchipip/vsrc/SimSerial.v")
  addResource("/testchipip/csrc/SimSerial.cc")
}
```

In this example, the `SimSerial` files will be copied from a specific folder (in this case the `path/to/testchipip/src/main/resources/testchipip/...`) to the build folder. The `addResource` path retrieves resources from the `src/main/resources` directory. So to get an item at `src/main/resources/fileA.v` you can use `addResource("/fileA.v")`. However, one caveat of this approach is that to retrieve the file during the FIRRTL compile, you must have that project in the FIRRTL compiler's classpath. Thus, you need to add the SBT project as a dependency to the FIRRTL compiler in the Chipyard `build.sbt`, which in Chipyards case is the `tapeout` project. For example, you added a new project called `myAwesomeAccel` in the Chipyard `build.sbt`. Then you can add it as a `dependsOn` dependency to the `tapeout` project. For example:

```
lazy val myAwesomeAccel = (project in file("generators/myAwesomeAccelFolder"))
  .dependsOn(rocketchip)
  .settings(commonSettings)

lazy val tapeout = conditionalDependsOn(project in file("./tools/barstools/tapeout/"))
  .dependsOn(myAwesomeAccel)
  .settings(commonSettings)
```

2.8.6 Context-Dependent-Environments

Readers may notice that the parameterization system frequently uses `(site, here, up)`. This construct is an artifact of the "context-dependent-environment" system which Chipyard and Rocket Chip both leverage for powerful composable hardware configuration.

The CDE parameterization system provides different "Views" of a single global parameterization. The syntax for accessing a `Field` within a `View` is `my_view(MyKey, site_view)`, where `site_view` is a "global" view that will be passed recursively into various functions and key-lookups in the call-stack of `my_view(MyKey, site_view)`.

Note: Rocket Chip based designs will frequently use `val p: Parameters` and `p(SomeKey)` to lookup the value of a key. `Parameters` is just a subclass of the `View` abstract class, and `p(SomeKey)` really expands into `p(SomeKey, p)`. This is because we consider the call `p(SomeKey)` to be the "site", or "source" of the original key query, so we need to pass in the view of the configuration provided by `p` recursively to future calls through the `site` argument.

Consider the following example using CDEs.

```
case object SomeKeyX extends Field[Boolean](false) // default is false
case object SomeKeyY extends Field[Boolean](false) // default is false
case object SomeKeyZ extends Field[Boolean](false) // default is false

class WithX(b: Boolean) extends Config((site, here, up) => {
  case SomeKeyX => b
})

class WithY(b: Boolean) extends Config((site, here, up) => {
  case SomeKeyY => b
})
```

When forming a query based on a `Parameters` object, like `p(SomeKeyX)`, the configuration system traverses the "chain" of config fragments until it finds a partial function which is defined at the key, and then returns that value.

```
val params = Config(new WithX(true) ++ new WithY(true)) // "chain" together config␣
↪fragments
params(SomeKeyX) // evaluates to true
params(SomeKeyY) // evaluates to true
params(SomeKeyZ) // evaluates to false
```

In this example, the evaluation of `params(SomeKeyX)` will terminate in the partial function defined in `WithX(true)`, while the evaluation of `params(SomeKeyY)` will terminate in the partial function defined in `WithY(true)`. Note that when no partial functions match, the evaluation will return the default value for that parameter.

The real power of CDEs arises from the `(site, here, up)` parameters to the partial functions, which provide useful "views" into the global parameterization that the partial functions may access to determine a parameterization.

Note: Additional information on the motivations for CDEs can be found in Chapter 2 of Henry Cook's Thesis .

Site

`site` provides a `View` of the "source" of the original parameter query.

```
class WithXEqualsYSite extends Config((site, here, up) => {
  case SomeKeyX => site(SomeKeyY) // expands to site(SomeKeyY, site)
}

val params_1 = Config(new WithXEqualsYSite ++ new WithY(true))
val params_2 = Config(new WithY(true) ++ new WithXEqualsYSite)
params_1(SomeKeyX) // evaluates to true
params_2(SomeKeyX) // evaluates to true
```

In this example, the partial function in `WithXEqualsYSite` will look up the value of `SomeKeyY` in the original `params_N` object, which becomes `site` in each call in the recursive traversal.

Here

`here` provides a `View` of the locally defined config, which typically just contains some partial function.

```
class WithXEqualsYHere extends Config((site, here, up) => {
  case SomeKeyY => false
  case SomeKeyX => here(SomeKeyY, site)
}

val params_1 = Config(new WithXEqualsYHere ++ new WithY(true))
val params_2 = Config(new WithY(true) ++ new WithXEqualsYHere)

params_1(SomeKeyX) // evaluates to false
params_2(SomeKeyX) // evaluates to false
```

In this example, note that although our final parameterization in `params_2` has `SomeKeyY` set to `true`, the call to `here(SomeKeyY, site)` only looks in the local partial function defined in `WithXEqualsYHere`. Note that we pass `site` to `here` since `site` may be used in the recursive call.

Up

`up` provides a `View` of the previously defined set of partial functions in the "chain" of partial functions. This is useful when we want to lookup a previously set value for some key, but not the final value for that key.

```
class WithXEqualsYUp extends Config((site, here, up) => {
  case SomeKeyX => up(SomeKeyY, site)
}

val params_1 = Config(new WithXEqualsYUp ++ new WithY(true))
val params_2 = Config(new WithY(true) ++ new WithXEqualsYUp)

params_1(SomeKeyX) // evaluates to true
params_2(SomeKeyX) // evaluates to false
```

In this example, note how `up(SomeKeyY, site)` in `WithXEqualsYUp` will refer to *either* the the partial function defining `SomeKeyY` in `WithY(true)` *or* the default value for `SomeKeyY` provided in the original `case object SomeKeyY` definition, *depending on the order in which the config fragments were used*. Since the order of config fragments affects the the order of the `View` traversal, `up` provides a different `View` of the parameterization in `params_1` and `params_2`.

Also note that again, `site` must be recursively passed through the call to `up`.

2.8.7 Creating Clocks in the Test Harness

Chipyard currently allows the SoC design (everything under `ChipTop`) to have independent clock domains through diplomacy. This implies that some reference clock enters the `ChipTop` and then is divided down into separate clock domains. From the perspective of the `TestHarness` module, the `ChipTop` clock and reset is provided from a clock and reset called `buildtopClock` and `buildtopReset`. In the default case, this `buildtopClock` and `buildtopReset` is directly wired to the clock and reset IO's of the `TestHarness` module. However, the `TestHarness` has the ability to generate a standalone clock and reset signal that is separate from the reference clock/reset of `ChipTop`. This allows harness components (including harness binders) the ability to "request" a clock for a new clock domain. This is useful for simulating systems in which modules in the harness have independent clock domains from the DUT.

Requests for a harness clock is done by the `HarnessClockInstantiator` class in `generators/chipyard/src/main/scala/TestHarness.scala`. This class is accessed in harness components by referencing the Rocket Chip parameters key `p(HarnessClockInstantiatorKey)`. Then you can request a clock and synchronized reset at a particular frequency by invoking the `requestClockBundle` function. Take the following example:

```
        withClockAndReset(th.buildtopClock, th.buildtopReset) {
          val memOverSerialTLClockBundle = p(HarnessClockInstantiatorKey).
↪requestClockBundle("mem_over_serial_tl_clock", memFreq)
          val serial_bits = SerialAdapter.asyncQueue(port, th.buildtopClock, th.
↪buildtopReset)
          val harnessMultiClockAXIRAM = SerialAdapter.connectHarnessMultiClockAXIRAM(
            system.serdesser.get,
            serial_bits,
            memOverSerialTLClockBundle,
            th.buildtopReset)
```

Here you can see the `p(HarnessClockInstantiatorKey)` is used to request a clock and reset at `memFreq` frequency.

Note: In the case that the reference clock entering `ChipTop` is not the overall reference clock of the simulation (i.e. the clock/reset coming into the `TestHarness` module), the `buildtopClock` and `buildtopReset` can differ from the implicit `TestHarness` clock and reset. For example, if the `ChipTop` reference is 500MHz but an extra harness clock is requested at 1GHz, the `TestHarness` implicit clock/reset will be at 1GHz while the `buildtopClock` and `buildtopReset` will be at 500MHz.

2.8.8 Managing Published Scala Dependencies

In preparation for Chisel 3.5, in Chipyard 1.5 Chisel, FIRRTL, the FIRRTL interpreter, and Treadle, were transitioned from being built-from-source to managed as published dependencies. Their submodules have been removed. Switching between published versions can be achieved by changing the versions specified in Chipyard's `build.sbt`.

Lists of available artifacts can be using search.maven.org or mvnrepository.org:

- Chisel3
- FIRRTL
- FIRRTL Interpreter
- Treadle

Publishing Local Changes

Under the new system, the simplest means to make custom source modifications to the packages above is to run `sbt +publishLocal` from within a locally modified clone of each of their respective repositories. This will post your custom variant to your local ivy2 repository, which can generally be found at `~/.ivy2`. See the SBT documentation for more detail.

In practice, this will require the following steps:

1. Check out and modify the desired projects.
2. Take note of, or modify, the versions of each projects (in their `build.sbt`). If you're cloning from `master` generally these will have default versions of `1.X-SNAPSHOT`, where `X` is not-yet-released next major version. You can modify the version string, to say `1.X-<MYSUFFIX>`, to uniquely identify your change.
3. Call `sbt +publishLocal` in each subproject. You may need to rebuild other published dependencies. SBT will be clear about what it is publishing and where it is putting it. The `+` is generally necessary and ensures that all cross versions of the package are published.
4. Update the Chisel or FIRRTL version in Chipyard's `build.sbt` to match the versions of your locally published packages.
5. Use Chipyard as you would normally. Now when you call out to make in Chipyard you should see SBT resolving dependencies to the locally published instances in your local ivy2 repository.
6. When you're finished, consider removing your locally published packages (by removing the appropriate directory in your ivy2 repository) to prevent accidentally reusing them in the future.

A final word of caution: packages you publish to your local ivy repository will be visible to other projects you may be building on your system. For example, if you locally publish Chisel 3.5.0, other projects that depend on Chisel 3.5.0 will preferentially use the locally published variant over the version available on Maven (the "real" 3.5.0). Take care to note versions you are publishing and remove locally published versions once you are done with them.

2.9 TileLink and Diplomacy Reference

TileLink is the cache coherence and memory protocol used by RocketChip and other Chipyard generators. It is how different modules like caches, memories, peripherals, and DMA devices communicate with each other.

RocketChip's TileLink implementation is built on top of Diplomacy, a framework for exchanging configuration information among Chisel generators in a two-phase elaboration scheme. For a detailed explanation of Diplomacy, see the paper by Cook, Terpstra, and Lee.

A brief overview of how to connect simple TileLink widgets can be found in the *MMIO Peripherals* section. This section will provide a detailed reference for the TileLink and Diplomacy functionality provided by RocketChip.

A detailed specification of the TileLink 1.7 protocol can be found on the SiFive website.

2.9.1 TileLink Node Types

Diplomacy represents the different components of an SoC as nodes of a directed acyclic graph. TileLink nodes can come in several different types.

Client Node

TileLink clients are modules that initiate TileLink transactions by sending requests on the A channel and receive responses on the D channel. If the client implements TL-C, it will receive probes on the B channel, send releases on

the C channel, and send grant acknowledgements on the E channel.

The L1 caches and DMA devices in RocketChip/Chipyard have client nodes.

You can add a TileLink client node to your LazyModule using the TLHelper object from testchipip like so:

```
class MyClient(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeClientNode(TLMasterParameters.v1(
    name = "my-client",
    sourceId = IdRange(0, 4),
    requestFifo = true,
    visibility = Seq(AddressSet(0x10000, 0xffff))))

  lazy val module = new LazyModuleImp(this) {
    val (tl, edge) = node.out(0)

    // Rest of code here
  }
}
```

The `name` argument identifies the node in the Diplomacy graph. It is the only required argument for TLClientParameters.

The `sourceId` argument specifies the range of source identifiers that this client will use. Since we have set the range to [0, 4) here, this client will be able to send up to four requests in flight at a time. Each request will have a distinct value in its source field. The default value for this field is `IdRange(0, 1)`, which means it would only be able to send a single request inflight.

The `requestFifo` argument is a boolean option which defaults to false. If it is set to true, the client will request that downstream managers that support it send responses in FIFO order (that is, in the same order the corresponding requests were sent).

The `visibility` argument specifies the address ranges that the client will access. By default it is set to include all addresses. In this example, we set it to contain a single address range `AddressSet(0x10000, 0xffff)`, which means that the client will only be able to access addresses from 0x10000 to 0x1ffff. normally do not specify this, but it can help downstream crossbar generators optimize the hardware by not arbitrating the client to managers with address ranges that don't overlap with its visibility.

Inside your lazy module implementation, you can call `node.out` to get a list of bundle/edge pairs. If you used the TLHelper, you only specified a single client edge, so this list will only have one pair.

The `tl` bundle is a Chisel hardware bundle that connects to the IO of this module. It contains two (in the case of TL-UL and TL-UH) or five (in the case of TL-C) decoupled bundles corresponding to the TileLink channels. This is what you should connect your hardware logic to in order to actually send/receive TileLink messages.

The `edge` object represents the edge of the Diplomacy graph. It contains some useful helper functions which will be documented in *TileLink Edge Object Methods*.

Manager Node

TileLink managers take requests from clients on the A channel and send responses back on the D channel. You can create a manager node using the TLHelper like so:

```
class MyManager(implicit p: Parameters) extends LazyModule {
  val device = new SimpleDevice("my-device", Seq("tutorial,my-device0"))
  val beatBytes = 8
  val node = TLHelper.makeManagerNode(beatBytes, TLSlaveParameters.v1(
    address = Seq(AddressSet(0x20000, 0xfff)),
```

(continues on next page)

(continued from previous page)

```
    resources = device.reg,
    regionType = RegionType.UNCACHED,
    executable = true,
    supportsArithmetic = TransferSizes(1, beatBytes),
    supportsLogical = TransferSizes(1, beatBytes),
    supportsGet = TransferSizes(1, beatBytes),
    supportsPutFull = TransferSizes(1, beatBytes),
    supportsPutPartial = TransferSizes(1, beatBytes),
    supportsHint = TransferSizes(1, beatBytes),
    fifoId = Some(0)))

  lazy val module = new LazyModuleImp(this) {
    val (tl, edge) = node.in(0)
  }
}
```

The `makeManagerNode` method takes two arguments. The first is `beatBytes`, which is the physical width of the TileLink interface in bytes. The second is a TLManagerParameters object.

The only required argument for `TLManagerParameters` is the `address`, which is the set of address ranges that this manager will serve. This information is used to route requests from the clients. In this example, the manager will only take requests for addresses from 0x20000 to 0x20fff. The second argument in `AddressSet` is a mask, not a size. You should generally set it to be one less than a power of two. Otherwise, the addressing behavior may not be what you expect.

The second argument is `resources`, which is usually retrieved from a `Device` object. In this case, we use a `SimpleDevice` object. This argument is necessary if you want to add an entry to the DeviceTree in the BootROM so that it can be read by a Linux driver. The two arguments to `SimpleDevice` are the name and compatibility list for the device tree entry. For this manager, then, the device tree entry would look like

```
L12: my-device@20000 {
    compatible = "tutorial,my-device0";
    reg = <0x20000 0x1000>;
};
```

The next argument is `regionType`, which gives some information about the caching behavior of the manager. There are seven region types, listed below:

1. `CACHED` - An intermediate agent may have cached a copy of the region for you.
2. `TRACKED` - The region may have been cached by another master, but coherence is being provided.
3. `UNCACHED` - The region has not been cached yet, but should be cached when possible.
4. `IDEMPOTENT` - Gets return most recently put content, but content should not be cached.
5. `VOLATILE` - Content may change without a put, but puts and gets have no side effects.
6. `PUT_EFFECTS` - Puts produce side effects and so must not be combined/delayed.
7. `GET_EFFECTS` - Gets produce side effects and so must not be issued speculatively.

Next is the `executable` argument, which determines if the CPU is allowed to fetch instructions from this manager. By default it is false, which is what most MMIO peripherals should set it to.

The next six arguments start with `support` and determine the different A channel message types that the manager can accept. The definitions of the message types are explained in *TileLink Edge Object Methods*. The `TransferSizes` case class specifies the range of logical sizes (in bytes) that the manager can accept for the particular message type. This is an inclusive range and all logical sizes must be powers of two. So in this case, the manager can accept requests with sizes of 1, 2, 4, or 8 bytes.

The final argument shown here is the `fifoId` setting, which determines which FIFO domain (if any) the manager is in. If this argument is set to `None` (the default), the manager will not guarantee any ordering of the responses. If the `fifoId` is set, it will share a FIFO domain with all other managers that specify the same `fifoId`. This means that client requests sent to that FIFO domain will see responses in the same order.

Register Node

While you can directly specify a manager node and write all of the logic to handle TileLink requests, it is usually much easier to use a register node. This type of node provides a `regmap` method that allows you to specify control/status registers and automatically generates the logic to handle the TileLink protocol. More information about how to use register nodes can be found in *Register Router*.

Identity Node

Unlike the previous node types, which had only inputs or only outputs, the identity node has both. As its name suggests, it simply connects the inputs to the outputs unchanged. This node is mainly used to combine multiple nodes into a single node with multiple edges. For instance, say we have two client lazy modules, each with their own client node.

```
class MyClient1(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeClientNode("my-client1", IdRange(0, 1))

  lazy val module = new LazyModuleImp(this) {
    // ...
  }
}

class MyClient2(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeClientNode("my-client2", IdRange(0, 1))

  lazy val module = new LazyModuleImp(this) {
    // ...
  }
}
```

Now we instantiate these two clients in another lazy module and expose their nodes as a single node.

```
class MyClientGroup(implicit p: Parameters) extends LazyModule {
  val client1 = LazyModule(new MyClient1)
  val client2 = LazyModule(new MyClient2)
  val node = TLIdentityNode()

  node := client1.node
  node := client2.node

  lazy val module = new LazyModuleImp(this) {
    // Nothing to do here
  }
}
```

We can also do the same for managers.

```
class MyManager1(beatBytes: Int)(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeManagerNode(beatBytes, TLSlaveParameters.v1(
    address = Seq(AddressSet(0x0, 0xfff))))
```

(continues on next page)

(continued from previous page)

```
  lazy val module = new LazyModuleImp(this) {
    // ...
  }
}

class MyManager2(beatBytes: Int)(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeManagerNode(beatBytes, TLSlaveParameters.v1(
    address = Seq(AddressSet(0x1000, 0xfff))))

  lazy val module = new LazyModuleImp(this) {
    // ...
  }
}

class MyManagerGroup(beatBytes: Int)(implicit p: Parameters) extends LazyModule {
  val man1 = LazyModule(new MyManager1(beatBytes))
  val man2 = LazyModule(new MyManager2(beatBytes))
  val node = TLIdentityNode()

  man1.node := node
  man2.node := node

  lazy val module = new LazyModuleImp(this) {
    // Nothing to do here
  }
}
```

If we want to connect the client and manager groups together, we can now do this.

```
class MyClientManagerComplex(implicit p: Parameters) extends LazyModule {
  val client = LazyModule(new MyClientGroup)
  val manager = LazyModule(new MyManagerGroup(8))

  manager.node :=* client.node

  lazy val module = new LazyModuleImp(this) {
    // Nothing to do here
  }
}
```

The meaning of the `:=*` operator is explained in more detail in the *Diplomacy Connectors* section. In summary, it connects two nodes together using multiple edges. The edges in the identity node are assigned in order, so in this case `client1.node` will eventually connect to `manager1.node` and `client2.node` will connect to `manager2.node`.

The number of inputs to an identity node should match the number of outputs. A mismatch will cause an elaboration error.

Adapter Node

Like the identity node, the adapter node takes some number of inputs and produces the same number of outputs. However, unlike the identity node, the adapter node does not simply pass the connections through unchanged. It can change the logical and physical interfaces between input and output and rewrite messages going through. RocketChip provides a library of adapters, which are catalogued in *Diplomatic Widgets*.

You will rarely need to create an adapter node yourself, but the invocation is as follows.

```
val node = TLAdapterNode(
  clientFn = { cp =>
    // ..
  },
  managerFn = { mp =>
    // ..
  })
```

The `clientFn` is a function that takes the `TLClientPortParameters` of the input as an argument and returns the corresponding parameters for the output. The `managerFn` takes the `TLManagerPortParameters` of the output as an argument and returns the corresponding parameters for the input.

Nexus Node

The nexus node is similar to the adapter node in that it has a different output interface than input interface. But it can also have a different number of inputs than it does outputs. This node type is mainly used by the `TLXbar` widget, which provides a TileLink crossbar generator. You will also likely not need to define this node type manually, but its invocation is as follows.

```
val node = TLNexusNode(
  clientFn = { seq =>
    // ..
  },
  managerFn = { seq =>
    // ..
  })
```

This has similar arguments as the adapter node's constructor, but instead of taking single parameters objects as arguments and returning single objects as results, the functions take and return sequences of parameters. And as you might expect, the size of the returned sequence need not be the same size as the input sequence.

2.9.2 Diplomacy Connectors

Nodes in a Diplomacy graph are connected to each other with edges. The Diplomacy library provides four operators that can be used to form edges between nodes.

:=

This is the basic connection operator. It is the same syntax as the Chisel uni-directional connector, but it is not equivalent. This operator connects Diplomacy nodes, not Chisel bundles.

The basic connection operator always creates a single edge between the two nodes.

:=*

This is a "query" type connection operator. It can create multiple edges between nodes, with the number of edges determined by the client node (the node on the right side of the operator). This can be useful if you are connecting a multi-edge client to a nexus node or adapter node.

:*=

This is a "star" type connection operator. It also creates multiple edges, but the number of edges is determined by the manager (left side of operator), rather than the client. It's useful for connecting nexus nodes to multi-edge manager nodes.

:*=*

This is a "flex" connection operator. It creates multiple edges based on whichever side of the operator has a known number of edges. This can be used in generators where the type of node on either side isn't known until runtime.

2.9.3 TileLink Edge Object Methods

The edge object associated with a TileLink node has several helpful methods for constructing TileLink messages and retrieving data from them.

Get

Constructor for a TLBundleA encoding a `Get` message, which requests data from memory. The D channel response to this message will be an `AccessAckData`, which may have multiple beats.

Arguments:

- `fromSource: UInt` - Source ID for this transaction
- `toAddress: UInt` - The address to read from
- `lgSize: UInt` - Base two logarithm of the number of bytes to be read

Returns:

A `(Bool, TLBundleA)` tuple. The first item in the pair is a boolean indicating whether or not the operation is legal for this edge. The second is the A channel bundle.

Put

Constructor for a TLBundleA encoding a `PutFull` or `PutPartial` message, which write data to memory. It will be a `PutPartial` if the `mask` is specified and a `PutFull` if it is omitted. The put may require multiple beats. If that is the case, only `data` and `mask` should change for each beat. All other fields must be the same for all beats in the transaction, including the address. The manager will respond to this message with a single `AccessAck`.

Arguments:

- `fromSource: UInt` - Source ID for this transaction.
- `toAddress: UInt` - The address to write to.
- `lgSize: UInt` - Base two logarithm of the number of bytes to be written.
- `data: UInt` - The data to write on this beat.
- `mask: UInt` - (optional) The write mask for this beat.

Returns:

A `(Bool, TLBundleA)` tuple. The first item in the pair is a boolean indicating whether or not the operation is legal for this edge. The second is the A channel bundle.

Arithmetic

Constructor for a TLBundleA encoding an `Arithmetic` message, which is an atomic operation. The possible values for the `atomic` field are defined in the `TLAtomics` object. It can be `MIN`, `MAX`, `MINU`, `MAXU`, or `ADD`, which correspond to atomic minimum, maximum, unsigned minimum, unsigned maximum, or addition operations, respectively. The previous value at the memory location will be returned in the response, which will be in the form of an `AccessAckData`.

Arguments:

- `fromSource: UInt` - Source ID for this transaction.
- `toAddress: UInt` - The address to perform an arithmetic operation on.
- `lgSize: UInt` - Base two logarithm of the number of bytes to operate on.
- `data: UInt` - Right-hand operand of the arithmetic operation
- `atomic: UInt` - Arithmetic operation type (from `TLAtomics`)

Returns:

A `(Bool, TLBundleA)` tuple. The first item in the pair is a boolean indicating whether or not the operation is legal for this edge. The second is the A channel bundle.

Logical

Constructor for a TLBundleA encoding a `Logical` message, an atomic operation. The possible values for the `atomic` field are `XOR`, `OR`, `AND`, and `SWAP`, which correspond to atomic bitwise exclusive or, bitwise inclusive or, bitwise and, and swap operations, respectively. The previous value at the memory location will be returned in an `AccessAckData` response.

Arguments:

- `fromSource: UInt` - Source ID for this transaction.
- `toAddress: UInt` - The address to perform a logical operation on.
- `lgSize: UInt` - Base two logarithm of the number of bytes to operate on.
- `data: UInt` - Right-hand operand of the logical operation
- `atomic: UInt` - Logical operation type (from `TLAtomics`)

Returns:

A `(Bool, TLBundleA)` tuple. The first item in the pair is a boolean indicating whether or not the operation is legal for this edge. The second is the A channel bundle.

Hint

Constructor for a TLBundleA encoding a `Hint` message, which is used to send prefetch hints to caches. The `param` argument determines what kind of hint it is. The possible values come from the `TLHints` object and are `PREFETCH_READ` and `PREFETCH_WRITE`. The first one tells caches to acquire data in a shared state. The second one tells cache to acquire data in an exclusive state. If the cache this message reaches is a last-level cache, there won't be any difference. If the manager this message reaches is not a cache, it will simply be ignored. In any case, a `HintAck` message will be sent in response.

Arguments:

- `fromSource: UInt` - Source ID for this transaction.

- `toAddress: UInt` - The address to prefetch
- `lgSize: UInt` - Base two logarithm of the number of bytes to prefetch
- `param: UInt` - Hint type (from TLHints)

Returns:

A `(Bool, TLBundleA)` tuple. The first item in the pair is a boolean indicating whether or not the operation is legal for this edge. The second is the A channel bundle.

AccessAck

Constructor for a TLBundleD encoding an `AccessAck` or `AccessAckData` message. If the optional `data` field is supplied, it will be an `AccessAckData`. Otherwise, it will be an `AccessAck`.

Arguments

- `a: TLBundleA` - The A channel message to acknowledge
- `data: UInt` - (optional) The data to send back

Returns:

The `TLBundleD` for the D channel message.

HintAck

Constructor for a TLBundleD encoding a `HintAck` message.

Arguments

- `a: TLBundleA` - The A channel message to acknowledge

Returns:

The `TLBundleD` for the D channel message.

first

This method take a decoupled channel (either the A channel or D channel) and determines whether the current beat is the first beat in the transaction.

Arguments:

- `x: DecoupledIO[TLChannel]` - The decoupled channel to snoop on.

Returns:

A `Boolean` which is true if the current beat is the first, or false otherwise.

last

This method take a decoupled channel (either the A channel or D channel) and determines whether the current beat is the last in the transaction.

Arguments:

- `x: DecoupledIO[TLChannel]` - The decoupled channel to snoop on.

Returns:

A `Boolean` which is true if the current beat is the last, or false otherwise.

done

Equivalent to `x.fire() && last(x)`.

Arguments:

- `x: DecoupledIO[TLChannel]` - The decoupled channel to snoop on.

Returns:

A `Boolean` which is true if the current beat is the last and a beat is sent on this cycle. False otherwise.

count

This method take a decoupled channel (either the A channel or D channel) and determines the count (starting from 0) of the current beat in the transaction.

Arguments:

- `x: DecoupledIO[TLChannel]` - The decoupled channel to snoop on.

Returns:

A `UInt` indicating the count of the current beat.

numBeats

This method takes in a TileLink bundle and gives the number of beats expected for the transaction.

Arguments:

- `x: TLChannel` - The TileLink bundle to get the number of beats from

Returns:

A `UInt` that is the number of beats in the current transaction.

numBeats1

Similar to `numBeats` except it gives the number of beats minus one. If this is what you need, you should use this instead of doing `numBeats - 1.U`, as this is more efficient.

Arguments:

- `x: TLChannel` - The TileLink bundle to get the number of beats from

Returns:

A `UInt` that is the number of beats in the current transaction minus one.

hasData

Determines whether the TileLink message contains data or not. This is true if the message is a PutFull, PutPartial, Arithmetic, Logical, or AccessAckData.

Arguments:

- `x:` `TLChannel` - The TileLink bundle to check

Returns:

A `Boolean` that is true if the current message has data and false otherwise.

2.9.4 Register Router

Memory-mapped devices generally follow a common pattern. They expose a set of registers to the CPUs. By writing to a register, the CPU can change the device's settings or send a command. By reading from a register, the CPU can query the device's state or retrieve results.

While designers can manually instantiate a manager node and write the logic for exposing registers themselves, it's much easier to use RocketChip's `regmap` interface, which can generate most of the glue logic.

For TileLink devices, you can use the `regmap` interface by extending the `TLRegisterRouter` class, as shown in *MMIO Peripherals*, or you can create a regular LazyModule and instantiate a `TLRegisterNode`. This section will focus on the second method.

Basic Usage

```
import chisel3._
import chisel3.util._
import freechips.rocketchip.config.Parameters
import freechips.rocketchip.diplomacy._
import freechips.rocketchip.regmapper._
import freechips.rocketchip.tilelink.TLRegisterNode

class MyDeviceController(implicit p: Parameters) extends LazyModule {
  val device = new SimpleDevice("my-device", Seq("tutorial,my-device0"))
  val node = TLRegisterNode(
    address = Seq(AddressSet(0x10028000, 0xfff)),
    device = device,
    beatBytes = 8,
    concurrency = 1)

  lazy val module = new LazyModuleImp(this) {
    val bigReg = RegInit(0.U(64.W))
    val mediumReg = RegInit(0.U(32.W))
    val smallReg = RegInit(0.U(16.W))

    val tinyReg0 = RegInit(0.U(4.W))
    val tinyReg1 = RegInit(0.U(4.W))

    node.regmap(
      0x00 -> Seq(RegField(64, bigReg)),
      0x08 -> Seq(RegField(32, mediumReg)),
      0x0C -> Seq(RegField(16, smallReg)),
      0x0E -> Seq(
        RegField(4, tinyReg0),
```

(continues on next page)

(continued from previous page)

```
        RegField(4, tinyReg1)))
  }
}
```

The code example above shows a simple lazy module that uses the `TLRegisterNode` to memory map hardware registers of different sizes. The constructor has two required arguments: `address`, which is the base address of the registers, and `device`, which is the device tree entry. There are also two optional arguments. The `beatBytes` argument is the interface width in bytes. The default value is 4 bytes. The `concurrency` argument is the size of the internal queue for TileLink requests. By default, this value is 0, which means there will be no queue. This value must be greater than 0 if you wish to decoupled requests and responses for register accesses. This is discussed in *Using Functions*.

The main way to interact with the node is to call the `regmap` method, which takes a sequence of pairs. The first element of the pair is an offset from the base address. The second is a sequence of `RegField` objects, each of which maps a different register. The `RegField` constructor takes two arguments. The first argument is the width of the register in bits. The second is the register itself.

Since the argument is a sequence, you can associate multiple `RegField` objects with an offset. If you do, the registers are read or written in parallel when the offset is accessed. The registers are in little endian order, so the first register in the list corresponds to the least significant bits in the value written. In this example, if the CPU wrote to offset 0x0E with the value 0xAB, `tinyReg0` will get the value 0xB and `tinyReg1` would get 0xA.

Decoupled Interfaces

Sometimes you may want to do something other than read and write from a hardware register. The `RegField` interface also provides support for reading and writing `DecoupledIO` interfaces. For instance, you can implement a hardware FIFO like so.

```
    // 4-entry 64-bit queue
    val queue = Module(new Queue(UInt(64.W), 4))

    node.regmap(
      0x00 -> Seq(RegField(64, queue.io.deq, queue.io.enq)))
```

This variant of the `RegField` constructor takes three arguments instead of two. The first argument is still the bit width. The second is the decoupled interface to read from. The third is the decoupled interface to write to. In this example, writing to the "register" will push the data into the queue and reading from it will pop data from the queue.

You need not specify both read and write for a register. You can also create read-only or write-only registers. So for the previous example, if you wanted enqueue and dequeue to use different addresses, you could write the following.

```
    node.regmap(
      0x00 -> Seq(RegField.r(64, queue.io.deq)),
      0x08 -> Seq(RegField.w(64, queue.io.enq)))
```

The read-only register function can also be used to read signals that aren't registers.

```
val constant = 0xf00d.U

node.regmap(
  0x00 -> Seq(RegField.r(8, constant)))
```

Using Functions

You can also create registers using functions. Say, for instance, that you want to create a counter that gets incremented on a write and decremented on a read.

```
  val counter = RegInit(0.U(64.W))

  def readCounter(ready: Bool): (Bool, UInt) = {
    when (ready) { counter := counter - 1.U }
    // (ready, bits)
    (true.B, counter)
  }

  def writeCounter(valid: Bool, bits: UInt): Bool = {
    when (valid) { counter := counter + 1.U }
    // Ignore bits
    // Return ready
    true.B
  }

  node.regmap(
    0x00 -> Seq(RegField.r(64, readCounter(_))),
    0x08 -> Seq(RegField.w(64, writeCounter(_, _))))
```

The functions here are essentially the same as a decoupled interface. The read function gets passed the `ready` signal and returns the `valid` and `bits` signals. The write function gets passed `valid` and `bits` and returns `ready`.

You can also pass functions that decouple the read/write request and response. The request will appear as a decoupled input and the response as a decoupled output. So for instance, if we wanted to do this for the previous example.

```
  val counter = RegInit(0.U(64.W))

  def readCounter(ivalid: Bool, oready: Bool): (Bool, Bool, UInt) = {
    val responding = RegInit(false.B)

    when (ivalid && !responding) { responding := true.B }

    when (responding && oready) {
      counter := counter - 1.U
      responding := false.B
    }

    // (iready, ovalid, obits)
    (!responding, responding, counter)
  }

  def writeCounter(ivalid: Bool, oready: Bool, ibits: UInt): (Bool, Bool) = {
    val responding = RegInit(false.B)

    when (ivalid && !responding) { responding := true.B }

    when (responding && oready) {
      counter := counter + 1.U
      responding := false.B
    }

    // (iready, ovalid)
    (!responding, responding)
```

(continues on next page)

(continued from previous page)

```
    }

    node.regmap(
      0x00 -> Seq(RegField.r(64, readCounter(_, _))),
      0x08 -> Seq(RegField.w(64, writeCounter(_, _, _))))
```

In each function, we set up a state variable `responding`. The function is ready to take requests when this is false and is sending a response when this is true.

In this variant, both read and write take an input valid and return an output ready. The only difference is that bits is an input for read and an output for write.

In order to use this variant, you need to set `concurrency` to a value larger than 0.

Register Routers for Other Protocols

One useful feature of the register router interface is that you can easily change the protocol being used. For instance, in the first example in *Basic Usage*, you could simply change the `TLRegisterNode` to and `AXI4RegisterNode`.

```
import freechips.rocketchip.amba.axi4.AXI4RegisterNode

class MyAXI4DeviceController(implicit p: Parameters) extends LazyModule {
  val node = AXI4RegisterNode(
    address = AddressSet(0x10029000, 0xfff),
    beatBytes = 8,
    concurrency = 1)

  lazy val module = new LazyModuleImp(this) {
    val bigReg = RegInit(0.U(64.W))
    val mediumReg = RegInit(0.U(32.W))
    val smallReg = RegInit(0.U(16.W))

    val tinyReg0 = RegInit(0.U(4.W))
    val tinyReg1 = RegInit(0.U(4.W))

    node.regmap(
      0x00 -> Seq(RegField(64, bigReg)),
      0x08 -> Seq(RegField(32, mediumReg)),
      0x0C -> Seq(RegField(16, smallReg)),
      0x0E -> Seq(
        RegField(4, tinyReg0),
        RegField(4, tinyReg1)))
  }
}
```

Other than the fact that AXI4 nodes don't take a `device` argument, and can only have a single AddressSet instead of multiple, everything else is unchanged.

2.9.5 Diplomatic Widgets

RocketChip provides a library of diplomatic TileLink and AXI4 widgets. The most commonly used widgets are documented here. The TileLink widgets are available from `freechips.rocketchip.tilelink` and the AXI4 widgets from `freechips.rocketchip.amba.axi4`.

TLBuffer

A widget for buffering TileLink transactions. It simply instantiates queues for each of the 2 (or 5 for TL-C) decoupled channels. To configure the queue for each channel, you pass the constructor a `freechips.rocketchip.diplomacy.BufferParams` object. The arguments for this case class are:

- `depth: Int` - The number of entries in the queue
- `flow: Boolean` - If true, combinationally couple the valid signals so that an input can be consumed on the same cycle it is enqueued.
- `pipe: Boolean` - If true, combinationally couple the ready signals so that single-entry queues can run at full rate.

There is an implicit conversion from `Int` available. If you pass an integer instead of a BufferParams object, the queue will be the depth given in the integer and `flow` and `pipe` will both be false.

You can also use one of the predefined BufferParams objects.

- `BufferParams.default = BufferParams(2, false, false)`
- `BufferParams.none = BufferParams(0, false, false)`
- `BufferParams.flow = BufferParams(1, true, false)`
- `BufferParams.pipe = BufferParams(1, false, true)`

Arguments:

There are four constructors available with zero, one, two, or five arguments.

The zero-argument constructor uses `BufferParams.default` for all of the channels.

The single-argument constructor takes a `BufferParams` object to use for all channels.

The arguments for the two-argument constructor are:

- `ace: BufferParams` - Parameters to use for the A, C, and E channels.
- `bd: BufferParams` - Parameters to use for the B and D channels

The arguments for the five-argument constructor are

- `a: BufferParams` - Buffer parameters for the A channel
- `b: BufferParams` - Buffer parameters for the B channel
- `c: BufferParams` - Buffer parameters for the C channel
- `d: BufferParams` - Buffer parameters for the D channel
- `e: BufferParams` - Buffer parameters for the E channel

Example Usage:

```
// Default settings
manager0.node := TLBuffer() := client0.node

// Using implicit conversion to make buffer with 8 queue entries per channel
manager1.node := TLBuffer(8) := client1.node

// Use default on A channel but pipe on D channel
manager2.node := TLBuffer(BufferParams.default, BufferParams.pipe) := client2.node

// Only add queues for the A and D channel
manager3.node := TLBuffer(
```

(continues on next page)

(continued from previous page)

```
  BufferParams.default,
  BufferParams.none,
  BufferParams.none,
  BufferParams.default,
  BufferParams.none) := client3.node
```

AXI4Buffer

Similar to the *TLBuffer*, but for AXI4. It also takes `BufferParams` objects as arguments.

Arguments:

Like TLBuffer, AXI4Buffer has zero, one, two, and five-argument constructors.

The zero-argument constructor uses the default BufferParams for all channels.

The one-argument constructor uses the provided BufferParams for all channels.

The two-argument constructor has the following arguments.

- `aw: BufferParams` - Buffer parameters for the "ar", "aw", and "w" channels.
- `br: BufferParams` - Buffer parameters for the "b", and "r" channels.

The five-argument constructor has the following arguments

- `aw: BufferParams` - Buffer parameters for the "ar" channel
- `w: BufferParams` - Buffer parameters for the "w" channel
- `b: BufferParams` - Buffer parameters for the "b" channel
- `ar: BufferParams` - Buffer parameters for the "ar" channel
- `r: BufferParams` - Buffer parameters for the "r" channel

Example Usage:

```
// Default settings
slave0.node := AXI4Buffer() := master0.node

// Using implicit conversion to make buffer with 8 queue entries per channel
slave1.node := AXI4Buffer(8) := master1.node

// Use default on aw/w/ar channel but pipe on b/r channel
slave2.node := AXI4Buffer(BufferParams.default, BufferParams.pipe) := master2.node

// Single-entry queues for aw, b, and ar but two-entry queues for w and r
slave3.node := AXI4Buffer(1, 2, 1, 1, 2) := master3.node
```

AXI4UserYanker

This widget takes an AXI4 port that has a user field and turns it into one without a user field. The values of the user field from input AR and AW requests is kept in internal queues associated with the ARID/AWID, which is then used to associate the correct user field to the responses.

Arguments:

- `capMaxFlight: Option[Int]` - (optional) An option which can hold the number of requests that can be inflight for each ID. If `None` (the default), the UserYanker will support the maximum number of inflight requests.

Example Usage:

```
nouser.node := AXI4UserYanker(Some(1)) := hasuser.node
```

AXI4Deinterleaver

Multi-beat AXI4 read responses for different IDs can potentially be interleaved. This widget reorders read responses from the slave so that all of the beats for a single transaction are consecutive.

Arguments:

- `maxReadBytes: Int` - The maximum number of bytes that can be read in a single transaction.

Example Usage:

```
interleaved.node := AXI4Deinterleaver() := consecutive.node
```

TLFragmenter

The TLFragmenter widget shrinks the maximum logical transfer size of the TileLink interface by breaking larger transactions into multiple smaller transactions.

Arguments:

- `minSize: Int` - Minimum size of transfers supported by all outward managers.
- `maxSize: Int` - Maximum size of transfers supported after the Fragmenter is applied.
- `alwaysMin: Boolean` - (optional) Fragment all requests down to minSize (else fragment to maximum supported by manager). (default: false)
- `earlyAck: EarlyAck.T` - (optional) Should a multibeat Put be acknowledged on the first beat or last beat? Possible values (default: `EarlyAck.None`):
  - `EarlyAck.AllPuts` - always acknowledge on first beat.
  - `EarlyAck.PutFulls` - acknowledge on first beat if PutFull, otherwise acknowledge on last beat.
  - `EarlyAck.None` - always acknowledge on last beat.
- `holdFirstDeny: Boolean` - (optional) Allow the Fragmenter to unsafely combine multibeat Gets by taking the first denied for the whole burst. (default: false)

Example Usage:

```
val beatBytes = 8
val blockBytes = 64

single.node := TLFragmenter(beatBytes, blockBytes) := multi.node

axi4lite.node := AXI4Fragmenter() := axi4full.node
```

Additional Notes

- TLFragmenter modifies: PutFull, PutPartial, LogicalData, Get, Hint
- TLFragmenter passes: ArithmeticData (truncated to minSize if alwaysMin)

- TLFragmenter cannot modify acquire (could livelock); thus it is unsafe to put caches on both sides

AXI4Fragmenter

The AXI4Fragmenter is similar to the *TLFragmenter*. The AXI4Fragmenter slices all AXI accesses into simple power-of-two sized and aligned transfers of the largest size supported by the manager. This makes it suitable as a first stage transformation to apply before an AXI4=>TL bridge. It also makes it suitable for placing after TL=>AXI4 bridge driving an AXI-lite slave.

Example Usage:

```
axi4lite.node := AXI4Fragmenter() := axi4full.node
```

TLSourceShrinker

The number of source IDs that a manager sees is usually computed based on the clients that connect to it. In some cases, you may wish to fix the number of source IDs. For instance, you might do this if you wish to export the TileLink port to a Verilog black box. This will pose a problem, however, if the clients require a larger number of source IDs. In this situation, you will want to use a TLSourceShrinker.

Arguments:

- `maxInFlight: Int` - The maximum number of source IDs that will be sent from the TLSourceShrinker to the manager.

Example Usage:

```
// client.node may have >16 source IDs
// manager.node will only see 16
manager.node := TLSourceShrinker(16) := client.node
```

AXI4IdIndexer

The AXI4 equivalent of *TLSourceShrinker*. This limits the number of AWID/ARID bits in the slave AXI4 interface. Useful for connecting to external or black box AXI4 ports.

Arguments:

- `idBits: Int` - The number of ID bits on the slave interface.

Example Usage:

```
// master.node may have >16 unique IDs
// slave.node will only see 4 ID bits
slave.node := AXI4IdIndexer(4) := master.node
```

Notes:

The AXI4IdIndexer will create a `user` field on the slave interface, as it stores the ID of the master requests in this field. If connecting to an AXI4 interface that doesn't have a `user` field, you'll need to use the *AXI4UserYanker*.

TLWidthWidget

This widget changes the physical width of the TileLink interface. The width of a TileLink interface is configured by managers, but sometimes you want the client to see a particular width.

Arguments:

- `innerBeatBytes: Int` - The physical width (in bytes) seen by the client

Example Usage:

```
// Assume the manager node sets beatBytes to 8
// With WidthWidget, client sees beatBytes of 4
manager.node := TLWidthWidget(4) := client.node
```

TLFIFOFixer

TileLink managers that declare a FIFO domain must ensure that all requests to that domain from clients which have requested FIFO ordering see responses in order. However, they can only control the ordering of their own responses, and do not have control over how those responses interleave with responses from other managers in the same FIFO domain. Responsibility for ensuring FIFO order across managers goes to the TLFIFOFixer.

Arguments:

- `policy: TLFIFOFixer.Policy` - (optional) Which managers will the TLFIFOFixer enforce ordering on? (default: `TLFIFOFixer.all`)

The possible values of `policy` are:

- `TLFIFOFixer.all` - All managers (including those without a FIFO domain) will have ordering guaranteed
- `TLFIFOFixer.allFIFO` - All managers that define a FIFO domain will have ordering guaranteed
- `TLFIFOFixer.allVolatile` - All managers that have a RegionType of `VOLATILE`, `PUT_EFFECTS`, or `GET_EFFECTS` will have ordering guaranteed (see *Manager Node* for explanation of region types).

TLXbar and AXI4Xbar

These are crossbar generators for TileLink and AXI4 which will route requests from TL client / AXI4 master nodes to TL manager / AXI4 slave nodes based on the addresses defined in the managers / slaves. Normally, these are constructed without arguments. However, you can change the arbitration policy, which determines which client ports get precedent in the arbiters. The default policy is `TLArbiter.roundRobin`, but you can change it to `TLArbiter.lowestIndexFirst` if you want a fixed arbitration precedence.

Arguments:

All arguments are optional.

- `arbitrationPolicy: TLArbiter.Policy` - The arbitration policy to use.
- `maxFlightPerId: Int` - (AXI4 only) The number of transactions with the same ID that can be inflight at a time. (default: 7)
- `awQueueDepth: Int` - (AXI4 only) The depth of the write address queue. (default: 2)

Example Usage:

```
// Instantiate the crossbar lazy module
val tlBus = LazyModule(new TLXbar)

// Connect a single input edge
tlBus.node := tlClient0.node
// Connect multiple input edges
tlBus.node :=* tlClient1.node
```

(continues on next page)

(continued from previous page)

```
// Connect a single output edge
tlManager0.node := tlBus.node
// Connect multiple output edges
tlManager1.node :*= tlBus.node

// Instantiate a crossbar with lowestIndexFirst arbitration policy
// Yes, we still use the TLArbiter singleton even though this is AXI4
val axiBus = LazyModule(new AXI4Xbar(TLArbiter.lowestIndexFirst))

// The connections work the same as TL
axiBus.node := axiClient0.node
axiBus.node :=* axiClient1.node
axiManager0.node := axiBus.node
axiManager1.node :*= axiBus.node
```

TLToAXI4 and AXI4ToTL

These are converters between the TileLink and AXI4 protocols. TLToAXI4 takes a TileLink client and connects to an AXI4 slave. AXI4ToTL takes an AXI4 master and connects to a TileLink manager. Generally you don't want to override the default arguments of the constructors for these widgets.

Example Usage:

```
axi4slave.node :=
    AXI4UserYanker() :=
    AXI4Deinterleaver(64) :=
    TLToAXI4() :=
    tlclient.node

tlmanager.node :=
    AXI4ToTL() :=
    AXI4UserYanker() :=
    AXI4Fragmenter() :=
    axi4master.node
```

You will need to add an *AXI4Deinterleaver* after the TLToAXI4 converter because it cannot deal with interleaved read responses. The TLToAXI4 converter also uses the AXI4 user field to store some information, so you will need an *AXI4UserYanker* if you want to connect to an AXI4 port without user fields.

Before you connect an AXI4 port to the AXI4ToTL widget, you will need to add an *AXI4Fragmenter* and *AXI4UserYanker* because the converter cannot deal with multi-beat transactions or user fields.

TLROM

The TLROM widget provides a read-only memory that can be accessed using TileLink. Note: this widget is in the `freechips.rocketchip.devices.tilelink` package, not the `freechips.rocketchip.tilelink` package like the others.

Arguments:

- `base: BigInt` - The base address of the memory
- `size: Int` - The size of the memory in bytes

- `contentsDelayed: => Seq[Byte]` - A function which, when called generates the byte contents of the ROM.
- `executable: Boolean` - (optional) Specify whether the CPU can fetch instructions from the ROM (default: `true`).
- `beatBytes: Int` - (optional) The width of the interface in bytes. (default: 4).
- `resources: Seq[Resource]` - (optional) Sequence of resources to add to the device tree.

Example Usage:

```
val rom = LazyModule(new TLROM(
  base = 0x100A0000,
  size = 64,
  contentsDelayed = Seq.tabulate(64) { i => i.toByte },
  beatBytes = 8))
rom.node := TLFragmenter(8, 64) := client.node
```

Supported Operations:

The TLROM only supports single-beat reads. If you want to perform multi-beat reads, you should attach a TLFragmenter in front of the ROM.

TLRAM and AXI4RAM

The TLRAM and AXI4RAM widgets provide read-write memories implemented as SRAMs.

Arguments:

- `address: AddressSet` - The address range that this RAM will cover.
- `cacheable: Boolean` - (optional) Can the contents of this RAM be cached. (default: `true`)
- `executable: Boolean` - (optional) Can the contents of this RAM be fetched as instructions. (default: `true`)
- `beatBytes: Int` - (optional) Width of the TL/AXI4 interface in bytes. (default: 4)
- `atomics: Boolean` - (optional, TileLink only) Does the RAM support atomic operations? (default: `false`)

Example Usage:

```
val xbar = LazyModule(new TLXbar)

val tlram = LazyModule(new TLRAM(
  address = AddressSet(0x1000, 0xfff)))

val axiram = LazyModule(new AXI4RAM(
  address = AddressSet(0x2000, 0xfff)))

tlram.node := xbar.node
axiram := TLToAXI4() := xbar.node
```

Supported Operations:

TLRAM only supports single-beat TL-UL requests. If you set `atomics` to true, it will also support Logical and Arithmetic operations. Use a `TLFragmenter` if you want multi-beat reads/writes.

AXI4RAM only supports AXI4-Lite operations, so multi-beat reads/writes and reads/writes smaller than full-width are not supported. Use an `AXI4Fragmenter` if you want to use the full AXI4 protocol.

2.10 Prototyping Flow

Chipyard supports FPGA prototyping for local FPGAs supported by fpga-shells. This includes popular FPGAs such as the Xilinx VCU118 and the Xilinx Arty 35T board.

Note: While `fpga-shells` provides harnesses for other FPGA development boards such as the Xilinx VC707 and some MicroSemi PolarFire, only harnesses for the Xilinx VCU118 and Xilinx Arty 35T boards are currently supported in Chipyard. However, the VCU118 and Arty 35T examples demonstrate how a user may implement support for other harnesses provided by fpga-shells.

2.10.1 General Setup and Usage

Sources and Submodule Setup

All FPGA prototyping-related collateral and sources are located in the `fpga` top-level Chipyard directory. This includes the `fpga-shells` submodule and the `src` directory that hold both Scala, TCL and other collateral. However, the `fpga-shells` submodule repository is not initialized by default. To initialize the `fpga-shells` submodule repository, run the included initialization script from the Chipyard top-level directory:

```
# in the chipyard top level folder
./scripts/init-fpga.sh
```

Generating a Bitstream

Generating a bitstream for any FPGA target using Vivado is similar to building RTL for a software RTL simulation. Similar to a software RTL simulation (*Simulating A Custom Project*), you can run the following command in the `fpga` directory to build a bitstream using Vivado:

```
make SBT_PROJECT=... MODEL=... VLOG_MODEL=... MODEL_PACKAGE=... CONFIG=... CONFIG_
↪PACKAGE=... GENERATOR_PACKAGE=... TB=... TOP=... BOARD=... FPGA_BRAND=... bitstream

# or

make SUB_PROJECT=<sub_project> bitstream
```

The `SUB_PROJECT` make variable is a way to meta make variable that sets all of the other make variables to a specific default. For example:

```
make SUB_PROJECT=vcu118 bitstream

# converts to

make SBT_PROJECT=fpga_platforms MODEL=VCU118FPGATestHarness VLOG_
↪MODEL=VCU118FPGATestHarness MODEL_PACKAGE=chipyard.fpga.vcu118␣
↪CONFIG=RocketVCU118Config CONFIG_PACKAGE=chipyard.fpga.vcu118 GENERATOR_
↪PACKAGE=chipyard TB=none TOP=ChipTop BOARD=vcu118 FPGA_BRAND=... bitstream
```

Some `SUB_PROJECT` defaults are already defined for use, including `vcu118` and `arty`. These default `SUB_PROJECT`'s setup the necessary test harnesses, packages, and more for the Chipyard make system. Like a software RTL simulation make invocation, all of the make variables can be overridden with user specific values (ex. include the `SUB_PROJECT` with a `CONFIG` and `CONFIG_PACKAGE` override). In most cases, you will just need to

run a command with a `SUB_PROJECT` and an overridden `CONFIG` to point to. For example, building the BOOM configuration on the VCU118:

```
make SUB_PROJECT=vcu118 CONFIG=BoomVCU118Config bitstream
```

That command will build the RTL and generate a bitstream using Vivado. The generated bitstream will be located in your designs specific build folder (`generated-src/<LONG_NAME>/obj`). However, like a software RTL simulation, you can also run the intermediate make steps to just generate Verilog or FIRRTL.

Debugging with ILAs on Supported FPGAs

ILA (integrated logic analyzers) can be added to certain designs for debugging relevant signals. First, open up the post synthesis checkpoint located in the build directory for your design in Vivado (it should be labeled `post_synth.dcp`). Then using Vivado, add ILAs (and other debugging tools) for your design (search online for more information on how to add an ILA). This can be done by modifying the post synthesis checkpoint, saving it, and running `make ... debug-bitstream`. This will create a new bitstream called `top.bit` in a folder named `generated-src/<LONG_NAME>/debug_obj/`. For example, running the bitstream build for an added ILA for a BOOM config.:

```
make SUB_PROJECT=vcu118 CONFIG=BoomVCU118Config debug-bitstream
```

Important: For more extensive debugging tools for FPGA simulations including printf synthesis, assert synthesis, instruction traces, ILAs, out-of-band profiling, co-simulation, and more, please refer to the *FireSim* platform.

2.10.2 Running a Design on VCU118

Basic VCU118 Design

The default Xilinx VCU118 harness is setup to have UART, a SPI SDCard, and DDR backing memory. This allows it to run RISC-V Linux from an SDCard while piping the terminal over UART to the host machine (the machine connected to the VCU118). To extend this design, you can create your own Chipyard configuration and add the `WithVCU118Tweaks` located in `fpga/src/main/scala/vcu118/Configs.scala`. Adding this config fragment will enable and connect the UART, SPI SDCard, and DDR backing memory to your Chipyard design/config.

```
class WithVCU118Tweaks extends Config(
  // harness binders
  new WithUART ++
  new WithSPISDCard ++
  new WithDDRMem ++
  // io binders
  new WithUARTIOPassthrough ++
  new WithSPIIOPassthrough ++
  new WithTLIOPassthrough ++
  // other configuration
  new WithDefaultPeripherals ++
  new chipyard.config.WithTLBackingMemory ++ // use TL backing memory
  new WithSystemModifications ++ // setup busses, use sdboot bootrom, setup ext. mem.␣
↪size
  new chipyard.config.WithNoDebug ++ // remove debug module
  new freechips.rocketchip.subsystem.WithoutTLMonitors ++
  new freechips.rocketchip.subsystem.WithNMemoryChannels(1) ++
  new WithFPGAFrequency(100) // default 100MHz freq
)
```

(continues on next page)

(continued from previous page)

```
class RocketVCU118Config extends Config(
  new WithVCU118Tweaks ++
  new chipyard.RocketConfig)
```

Brief Implementation Description + More Complicated Designs

The basis for a VCU118 design revolves around creating a special test harness to connect the external IOs to your Chipyard design. This is done with the `VCU118TestHarness` in the basic default VCU118 FPGA target. The `VCU118TestHarness` (located in `fpga/src/main/scala/vcu118/TestHarness.scala`) uses `Overlays` that connect to the VCU118 external IOs. Generally, the `Overlays` take an IO from the `ChipTop` (labeled as `topDesign` in the file) when "placed" and connect it to the external IO and generate necessary Vivado collateral. For example, the following shows a UART `Overlay` being "placed" into the design with a IO input called `io_uart_bb`.

```
  // 1st UART goes to the VCU118 dedicated UART

  val io_uart_bb = BundleBridgeSource(() => (new UARTPortIO(dp(PeripheryUARTKey).
↪head)))
  dp(UARTOverlayKey).head.place(UARTDesignInput(io_uart_bb))
```

Here the `UARTOverlayKey` is referenced and used to "place" the necessary connections (and collateral) to connect to the UART. The `UARTDesignInput` is used to pass in the UART IO from the `ChipTop`/`topDesign` to the `Overlay`. Note that the `BundleBridgeSource` can be viewed as a glorified wire (that is defined in the `LazyModule` scope). This pattern is similar for all other `Overlays` in the test harness. They must be "placed" and given a set of inputs (IOs, parameters). The main exception to this pattern is the `Overlay` used to generate the clock(s) for the FPGA.

```
  // place all clocks in the shell
  require(dp(ClockInputOverlayKey).size >= 1)
  val sysClkNode = dp(ClockInputOverlayKey)(0).place(ClockInputDesignInput()).
↪overlayOutput.node

  /*** Connect/Generate clocks ***/

  // connect to the PLL that will generate multiple clocks
  val harnessSysPLL = dp(PLLFactoryKey)()
  harnessSysPLL := sysClkNode

  // create and connect to the dutClock
  println(s"VCU118 FPGA Base Clock Freq: ${dp(DefaultClockFrequencyKey)} MHz")
  val dutClock = ClockSinkNode(freqMHz = dp(DefaultClockFrequencyKey))
  val dutWrangler = LazyModule(new ResetWrangler)
  val dutGroup = ClockGroup()
  dutClock := dutWrangler.node := dutGroup := harnessSysPLL
```

Without going into too much detail, the clocks overlay is placed in the harness and a PLL node (`harnessSysPLL`) generates the necessary clocks specified by `ClockSinkNodes`. For ease of use, you can change the `FPGAFrequencyKey` to change the default clock frequency of the FPGA design.

After the harness is created, the `BundleBridgeSource`'s must be connected to the `ChipTop` IOs. This is done with harness binders and io binders (see `fpga/src/main/scala/vcu118/HarnessBinders.scala` and `fpga/src/main/scala/vcu118/IOBinders.scala`). For more information on harness binders and io binders, refer to *IOBinders and HarnessBinders*.

Introduction to the Bringup Design

An example of a more complicated design used for Chipyard test chips can be viewed in `fpga/src/main/scala/vcu118/bringup/`. This example extends the default test harness and creates new `Overlays` to connect to a DUT (connected to the FMC port). Extensions include another UART (connected over FMC), I2C (connected over FMC), miscellaneous GPIOS (can be connected to anything), and a TSI Host Widget. The TSI Host Widget is used to interact with the DUT from the prototype over a SerDes link (sometimes called the Low BandWidth InterFace - LBWIF) and provide access to a channel of the FPGA's DRAM.

Note: Remember that since whenever a new test harness is created (or the config changes, or the config packages changes, or...), you need to modify the make invocation. For example, `make SUB_PROJECT=vcu118 CONFIG=MyNewVCU118Config CONFIG_PACKAGE=this.is.my.scala.package bitstream`. See *Generating a Bitstream* for information on the various make variables.

Running Linux on VCU118 Designs

As mentioned above, the default VCU118 harness is setup with a UART and a SPI SDCard. These are utilized to both interact with the DUT (with the UART) and load in Linux (with the SDCard). The following steps describe how to build and run buildroot Linux on the prototype platform.

Building Linux with FireMarshal

Since the prototype currently does not have a block device setup for it, we build Linux with the rootfs built into the binary (otherwise known as "initramfs" or "nodisk" version of Linux). To make building this type of Linux binary easy, we will use the FireMarshal platform (see *FireMarshal* for more information).

1. Setup FireMarshal (see *FireMarshal* on the initial setup).
2. By default, FireMarshal is setup to work with FireSim. Instead, we want to target the prototype platform. This is done by switching the FireMarshal "board" from "firechip" to "prototype" using `marshal-config.yaml`:

```
# this assumes you do not have a `marshal-config.yaml` file already setup
echo "board-dir : 'boards/prototype'" > $PATH_TO_FIREMARSHAL/marshal-config.yaml
```

Note: Refer to the FireMarshal docs on more ways to set the board differently through environment variables and more.

---

3. Next, build the workload (a.k.a buildroot Linux) in FireMarshal with the `nodisk` option flag. For the rest of these steps, we will assume you are using the base `br-base.json` workload. This workload has basic support for GPIO and SPI drivers (in addition to the default UART driver) but you can build off it in different workloads (refer to FireMarshal docs on workload inheritance).

```
./marshal -v -d build br-base.json # here the -d indicates --nodisk or initramfs
```

Note: Using the "board" FireMarshal functionality allows any child workload depending on the `br-base.json` workload specification to target a "prototype" platform rather than FireChip platform. Thus, you can re-use existing workloads that depend on `br-base.json` on the prototype platform by just changing the "board"!

---

4. The last step to generate the proper binary is to flatten it. This is done by using FireMarshal's `install` feature which will produce a `*-flat` binary in the `$PATH_TO_FIREMARSHAL/images` directory (in our case `br-base-bin-nodisk-flat`) from the previously built Linux binary (`br-base-bin-nodisk`).

```
./marshal -v -d install -t prototype br-base.json
```

Setting up the SDCard

These instructions assume that you have a spare uSDCard that can be loaded with Linux and other files using two partitions. The 1st partition will be used to store the Linux binary (created with FireMarshal or other means) while the 2nd partition will store a file system that can be accessed from the DUT. Additionally, these instructions assume you are using Linux with `sudo` privileges and `gdisk`, but you can follow a similar set of steps on Mac (using `gpt` or another similar program).

1. Wipe the GPT on the card using `gdisk`. Use the *z* command from the expert menu (opened with 'x', closed with 'm') to zap everything. For rest of these instructions, we assume the SDCard path is `/dev/sdc` (replace this with the path to your SDCard).

```
sudo gdisk /dev/sdc
```

2. Create the new GPT with *o*. Click yes on all the prompts.
3. The VCU118 bootrom assumes that the Linux binary to load into memory will be located on sector 34 of the SDCard. Change the default partition alignment to *1* so you can write to sector *34*. Do this with the *l* command from the expert menu (opened with 'x', closed with 'm').
4. Create a 512MiB partition to store the Linux binary (this can be smaller but it must be larger than the size of the Linux binary). Use *n*, partion number 1 and select sector 34, with size *+1048576* (corresponding to 512MiB). For the type, search for the *apfs* type and use the hex number given.
5. Create a second partition to store any other files with the rest of the SDCard. Use *n* and use the defaults for partition number, starting sector and overall size (expand the 2nd partition to the rest of the SDCard space). For the type, search for the *hfs* and use the hex number given.
6. Write the changes using *w*.
7. Setup the filesystem on the 2nd partition. Note that the `/dev/sdc2` points to the 2nd partition. Use the following command:

```
sudo mkfs.hfs -v "PrototypeData" /dev/sdc2
```

Transfer and Run Linux from the SDCard

After you have a Linux boot binary and the SDCard is setup properly (1st partition at sector 34), you can transfer the binary to the 1st SDCard partition. In this example, we generated a `br-base-bin-nodisk-flat` from FireMarshal and we will load it using `dd`. Note that `sdc1` points to the 1st partition (remember to change the `sdc` to your own SDCard path).

```
sudo dd if=$PATH_TO_FIREMARSHAL/br-base-bin-nodisk-flat of=/dev/sdc1
```

If you want to add files to the 2nd partition, you can also do this now.

After loading the SDCard with Linux and potentially other files, you can program the FPGA and plug in the SDCard. To interact with Linux via the UART console, you can connect to the serial port (in this case called `ttyUSB1`) using something like `screen`:

```
screen -S FPGA_UART_CONSOLE /dev/ttyUSB1 115200
```

Once connected, you should see the binary being loaded as well as Linux output (in some cases you might need to reset the DUT). Sign in as 'root' with password 'fpga'.

2.10.3 Running a Design on Arty

Basic Arty Design

The default Xilinx Arty 35T harness is setup to have JTAG available over the board's PMOD pins, and UART available over its FTDI serial USB adapter. The pin mappings for JTAG signals are identical to those described in the SiFive Freedom E310 Arty 35T Getting Started Guide. The JTAG interface allows a user to connect to the core via OpenOCD, run bare-metal applications, and debug these applications with gdb. UART allows a user to communicate with the core over a USB connection and serial console running on a PC. To extend this design, a user may create their own Chipyard configuration and add the `WithArtyTweaks` located in `fpga/src/main/scala/arty/Configs.scala`. Adding this config. fragment will enable and connect the JTAG and UART interfaces to your Chipyard design.

```
class WithArtyTweaks extends Config(
  new WithArtyJTAGHarnessBinder ++
  new WithArtyUARTHarnessBinder ++
  new WithArtyResetHarnessBinder ++
  new WithDebugResetPassthrough ++
  new WithDefaultPeripherals ++
  new freechips.rocketchip.subsystem.WithNBreakpoints(2))

class TinyRocketArtyConfig extends Config(
  new WithArtyTweaks ++
  new chipyard.TinyRocketConfig)
```

Future peripherals to be supported include the Arty 35T SPI Flash EEPROM, and I2C/PWM/SPI over the Arty 35T GPIO pins. These peripherals are available as part of sifive-blocks.

Brief Implementation Description and Guidance for Adding/Changing Xilinx Collateral

Like the VCU118, the basis for the Arty 35T design is the creation of a special test harness that connects the external IO (which exist as Xilinx IP blackboxes) to the Chipyard design. This is done with the `ArtyTestHarness` in the basic default Arty 35T target. However, unlike the `VCU118TestHarness`, the `ArtyTestHarness` uses no `Overlays`, and instead directly connects chip top IO to the ports of the external IO blackboxes, using functions such as `IOBUF` provided by `fpga-shells`. Unlike the VCU118 and other more complicated test harnesses, the Arty 35T Vivado collateral is not generated by `Overlays`, but rather are a static collection of `create_ip` and `set_properties` statements located in the files within `fpga/fpga-shells/xilinx/arty/tcl` and `fpga/fpga-shells/xilinx/arty/constraints`. If the user wishes to re-map FPGA package pins to different harness-level IO, this may be changed within `fpga/fpga-shells/xilinx/arty/constraints/arty-master.xdc`. The addition of new Xilinx IP blocks may be done in `fpga-shells/xilinx/arty/tcl/ip.tcl`, mapped to harness-level IOs in `arty-master.xdc`, and wired through from the test harness to the chip top using `HarnessBinders` and `IOBinders`. Examples of a simple `IOBinder` and `HarnessBinder` for routing signals (in this case the debug and JTAG resets) from the core to the test harness are the `WithResetPassthrough` and `WithArtyResetHarnessBinder`.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search