Chipyard Documentation

Berkeley Architecture Research

Mar 15, 2020

Contents

1 Quick Start **3**

1.1 System Requirements . . . 3
1.2 Setting up the Chipyard Repo . . . 3
1.3 Installing the RISC-V Tools . . . 3
1.4 What's Next? . . . 4
1.5 Getting Help . . . 4
1.6 Table of Contents . . . 4
1.6.1 Chipyard Basics . . . 4
1.6.2 Simulation . . . 12
1.6.3 Included RTL Generators . . . 15
1.6.4 Development Tools . . . 28
1.6.5 VLSI Flow . . . 32
1.6.6 Customization . . . 39
1.6.7 Target Software . . . 57
1.6.8 Advanced Concepts . . . 58
1.6.9 TileLink and Diplomacy Reference . . . 68

2 Indices and tables **87**

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. New to Chipyard? Jump to the *Chipyard Basics* page for more info.

CHAPTER 1

Quick Start

1.1 System 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.

1.2 Setting up the Chipyard Repo

Start by fetching Chipyard's sources. Run:

```
git clone https://github.com/ucb-bar/chipyard.git
cd chipyard
./scripts/init-submodules-no-riscv-tools.sh
```

This will initialize and checkout all of the necessary git submodules.

1.3 Installing the RISC-V Tools

We need to install the RISC-V toolchain in order to be able to run RISC-V programs using the Chipyard infrastructure. 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`. To build the toolchains, you should run:

```
./scripts/build-toolchains.sh
```

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.

Finally, set up Chipyard's environment variables and put the newly built toolchain on your path:

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

1.4 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 *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.

1.5 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.

1.6 Table of Contents

1.6.1 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!

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.

Ariane Core An in-order RISC-V core written in System Verilog. See *Ariane 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).

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 wrapper) 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 wrapper) 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.

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.

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!

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 Top(implicit p: Parameters) extends System
  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
```

(continues on next page)

(continued from previous page)

```
  with testchipip.CanHavePeripherySerial // Enables optionally adding the TSI serial-
↪adapter and port
  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 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
{
  override lazy val module = new TopModule(this)
}

class TopModule[+L <: Top](l: L) extends SystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with testchipip.CanHavePeripheryBlockDeviceModuleImp
  with testchipip.CanHavePeripherySerialModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with icenet.CanHavePeripheryIceNICModuleImp
  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 `Top` example class, the "outer" `Top` instantiates the "inner" `TopModule` 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 = Module(LazyModule(Top))`. After elaboration, the result will be a `Top` 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).

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
curl https://bintray.com/sbt/rpm/rpm | sudo tee /etc/yum.repos.d/bintray-sbt-rpm.repo
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 python36 patch diffstat texi2html texinfo subversion chrpath git␣
↪wget
# deps for qemu
sudo yum install -y gtk3-devel
# deps for firemarshal
sudo yum install -y python36-pip python36-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
sudo yum install -y devtoolset-8-make
# install DTC
sudo yum install -y dtc

# install verilator
git clone http://git.veripool.org/git/verilator
cd verilator
```

(continues on next page)

(continued from previous page)

```
git checkout v4.028
autoconf && ./configure && make -j16 && sudo make install
```

In Ubuntu/Debian-based platforms (Ubuntu), we recommend installing the following dependencies:

```
#!/bin/bash

set -ex

sudo apt-get install -y build-essential bison flex
sudo apt-get install -y libgmp-dev libmpfr-dev libmpc-dev zlib1g-dev vim git default-
↪jdk default-jre
# install sbt: https://www.scala-sbt.org/release/docs/Installing-sbt-on-Linux.html
echo "deb https://dl.bintray.com/sbt/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 python3.6 patch diffstat texi2html texinfo subversion chrpath␣
↪git wget
# deps for qemu
sudo apt-get install -y libgtk-3-dev
# deps for firemarshal
sudo apt-get install -y python3-pip python3.6-dev rsync libguestfs-tools expat ctags
# install DTC
sudo apt-get install -y device-tree-compiler

# install verilator
git clone http://git.veripool.org/git/verilator
cd verilator
git checkout v4.028
autoconf && ./configure && make -j16 && 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).

Checking out the sources

After cloning this repo, you will need to initialize all of the submodules.

```
git clone https://github.com/ucb-bar/chipyard.git
cd chipyard
./scripts/init-submodules-no-riscv-tools.sh
```

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.

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

# OR

./scripts/build-toolchains.sh esp-tools # for a modified risc-v toolchain with Hwacha␣
↪vector instructions
```

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

1.6.2 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. Conversly, 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 visability.

Click next to see how to run a simulation.

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.

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`.

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.

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.

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.

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. 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)
```

Only 3 additional config fragments are needed.

- `WithFireSimConfigTweaks` modifies your design to better fit the FireSim usage model. For example, FireSim designs typically include a UART. Technically, adding this in is optional, but *strongly* recommended.
- `WithDefaultMemModel` sets the external memory model in the FireSim simulation. See the FireSim documentation for details.
- `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.

1.6.3 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

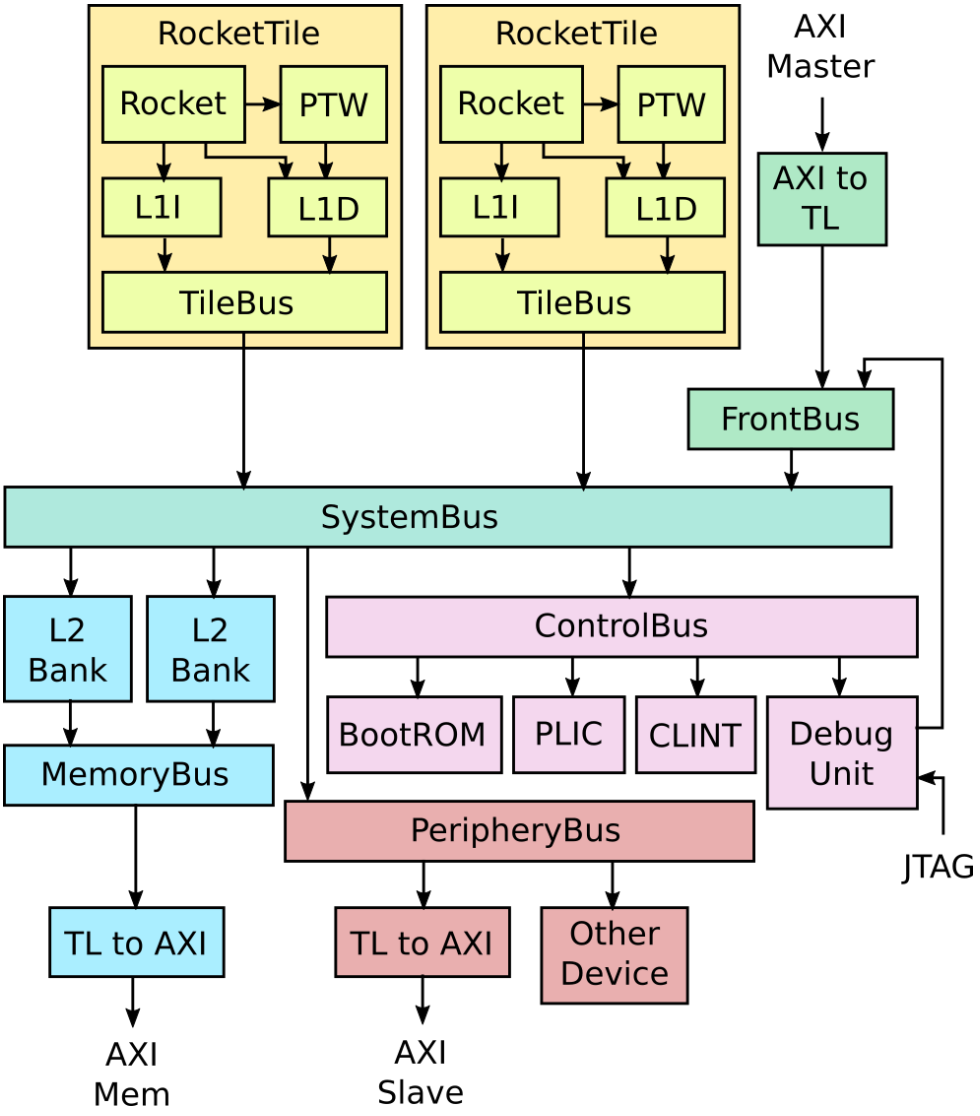
themselves will automatically be used when building with Chipyard and propagate to software simulation, FPGA-accelerated simulation, and VLSI flows.

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.

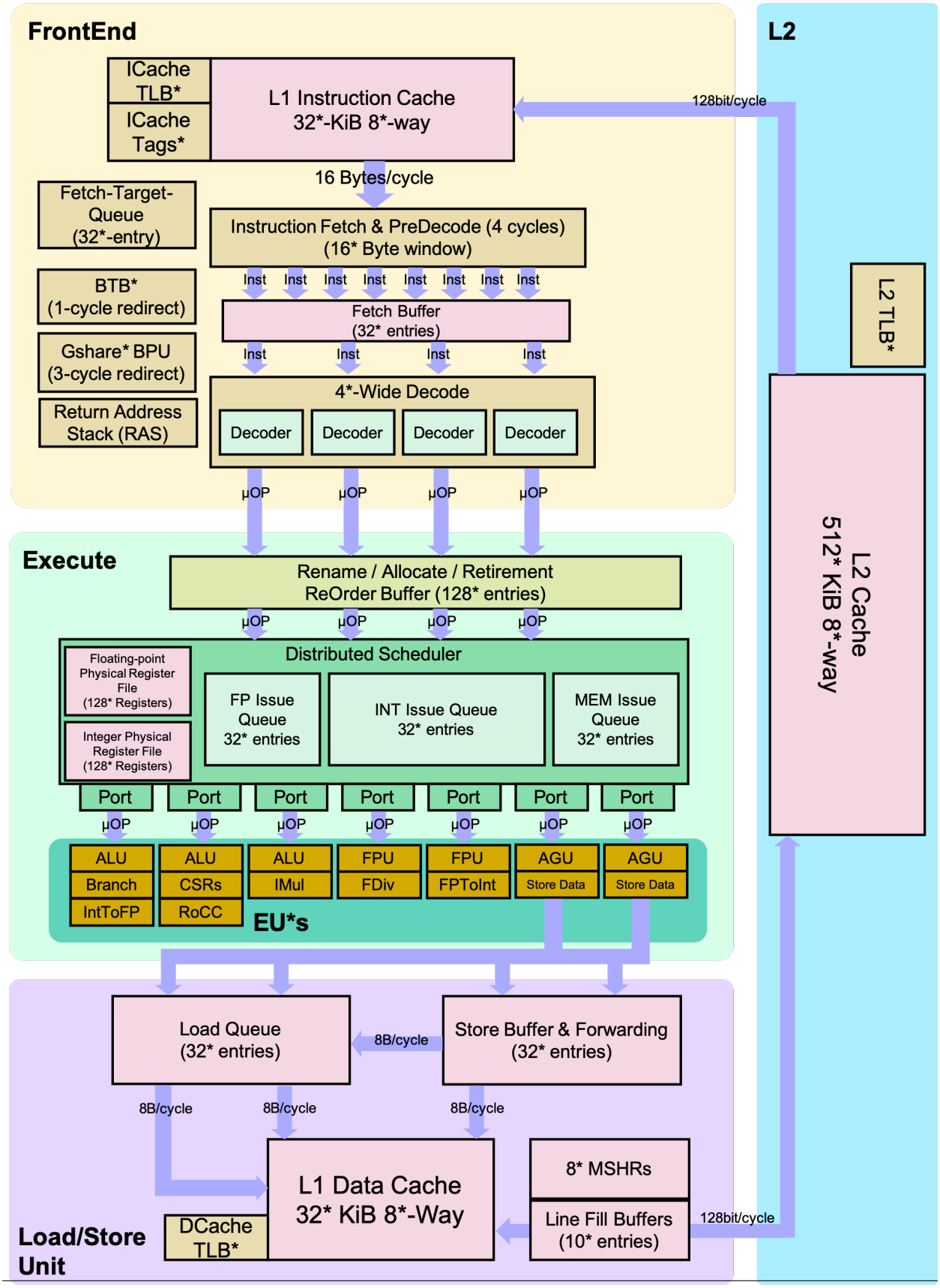
Rocket Core

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.

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.

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.

Gemmini

The Gemmini project is developing a systolic-array based matrix multiplication unit generator for the investigation of software/hardware implications of such integrated SoC accelerators. It is inspired by recent trends in machine learning accelerators for edge and mobile SoCs.

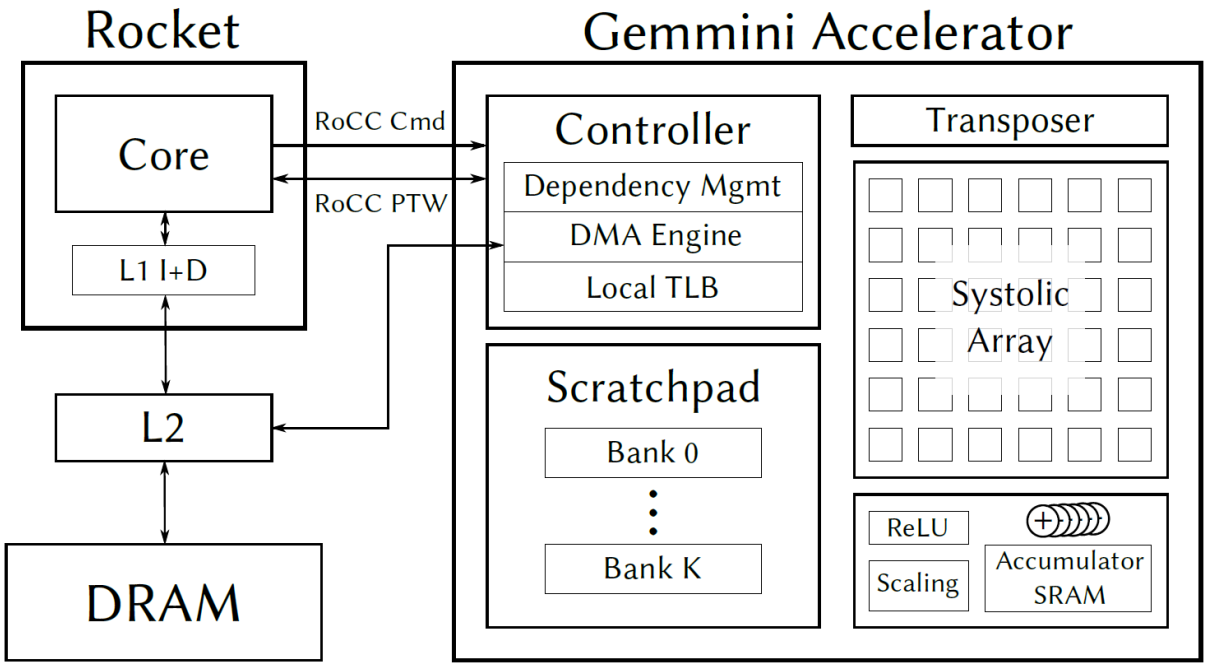
Gemmini is implemented as a RoCC accelerator with non-standard RISC-V custom instructions. The Gemmini unit uses the RoCC port of a Rocket or BOOM *tile*, and by default connects to the memory system through the System Bus (i.e., directly to the L2 cache).

To add a Gemmini unit to an SoC, you should add the `gemmini.DefaultGemminiConfig` config fragment to the SoC configurations. To change the configuration of the Gemmini accelerator unit, you can write a custom configuration to replace the `DefaultGemminiConfig`, which you can view under generators/gemmini/src/main/scala/configs.scala to see the possible configuration parameters.

The example Chipyard config includes the following example SoC configuration which includes Gemmini:

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

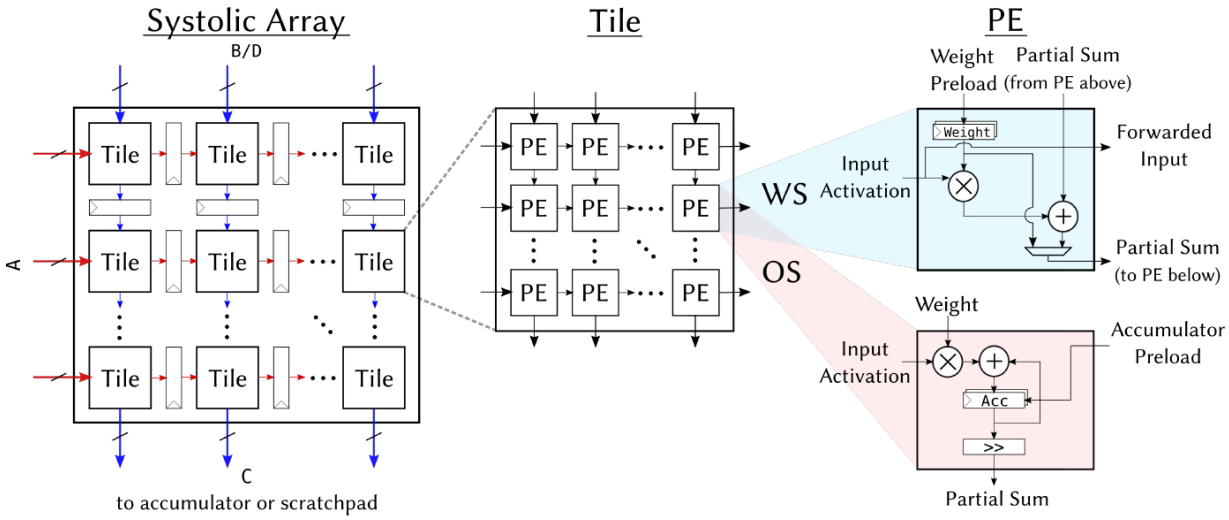
```
cd sims/verilator # or "cd sims/vcs"
make CONFIG=GemminiRocketConfig
```



Generator Parameters

Major parameters of interest include:

- Systolic array dimensions (`tileRows`, `tileColumns`, `meshRows`, `meshColumns`): The systolic array is composed of a 2-level hierarchy, in which each tile is fully combinational, while a mesh of tiles has pipeline registers between each tile.



- Dataflow parameters (`dataflow`): Determine whether the systolic array in Gemmini is output-stationary or

weight-stationary, or whether it supports both dataflows so that programmers may choose between them at runtime.

- Scratchpad and accumulator memory parameters (`sp_banks`, `sp_capacity`, `acc_capacity`): Determine the properties of the Gemmini scratchpad memory: overall capacity of the scratchpad or accumulators (in KiB), and the number of banks the scratchpad is divided into.
- Type parameters (`inputType`, `outputType`, `accType`): Determine the data-types flowing through different parts of a Gemmini accelerator. For example, `inputType` may be an 8-bit fixed-point number, while `accType`, which determines the type of partial accumulations in a matrix multiplication, may be a 32-bit integer. `outputType` only determines the type of the data passed between two processing elements (PEs); for example, an 8-bit multiplication may produce a 16-bit result which must be shared between PEs in a systolic array.
- Access-execute queue parameters (`ld_queue_length`, `st_queue_length`, `ex_queue_length`, `rob_entries`): To implement access-execute decoupling, a Gemmini accelerator has a load instruction queue, a store instruction queue, and an execute instruction queue. The relative sizes of these queue determine the level of access-execute decoupling. Gemmini also implements a reorder buffer (ROB) - the number of entries in the ROB determines possible dependency management limitations.
- DMA parameters (`dma_maxbytes`, `dma_buswidth`, `mem_pipeline`): Gemmini implements a DMA to move data from main memory to the Gemmini scratchpad, and from the Gemmini accumulators to main memory. The size of these DMA transactions is determined by the DMA parameters. These DMA parameters are tightly coupled with Rocket Chip SoC system parameters: in particular `dma_buswidth` is associated with the `SystemBusKey` `beatBytes` parameter, and `dma_maxbytes` is associated with `cacheblockbytes` Rocket Chip parameters.

Gemmini Software

The Gemmini non-standard ISA extension is specified in the Gemmini repository. The ISA includes configuration instructions, data movement instructions (from main memory to the Gemmini scratchpad, and from the Gemmini accumulators to main memory), and matrix multiplication execution instructions.

Since Gemmini instructions are not exposed through the GNU binutils assembler, several C macros are provided in order to construct the instruction encodings to call these instructions.

The Gemmini generator includes a C matrix multiplication library which wraps the calls to the custom Gemmini instructions. The `software` directory of the generator includes the aforementioned library and macros, as well as bare-metal tests, and some FireMarshal workloads to run the tests in a Linux environment. In particular, the matrix multiplication C library can be found in the `software/gemmini-rocc-tests/include/gemmini.h` file.

The Gemmini generator generates a C header file based on the generator parameters. This header files gets compiled together with the matrix multiplication library to tune library performance. The generated header file can be found under `software/gemmini-rocc-tests/include/gemmini_params.h`

Build and Run Gemmini Tests

To build Gemmini tests:

```
cd generators/gemmini/software/gemmini-rocc-tests/
./build.sh
```

Afterwards, the test binaries will be found in `generators/gemmini/software/gemmini-rocc-tests/build`. Binaries whose names end in `-baremetal` are meant to be run in a bare-metal environment, while binaries whose names end in `-linux` are meant to run in a Linux environment. You can run the tests either on a cycle-accurate RTL simulator, or on a (much faster) functional ISA simulator called Spike.

The Gemmini generator implements a custom non-standard version of Spike. This implementation is found within the `esp-tools` Spike implementation, together with the Hwacha vector accelerator non-standard ISA-extension. In order to use this version of Spike, please make sure to build the `esp-tools` software toolchain, as described in *Building a Toolchain*.

In order to run Spike with the gemmini functional model, you will need to use the `--extension=gemmini` flag. For example:

```
spike --extension=gemmini <some/gemmini/baremetal/test>
```

Spike is built by default without a commit log. However, if you would like to add detailed functional log of gemmini operation to the spike model, you can rebuild spike manually (based on the instructions in the `esp-tools/riscv-isa-sim/README` file), with the `--enable-gemminicommitlog` option added to the `configure` step.

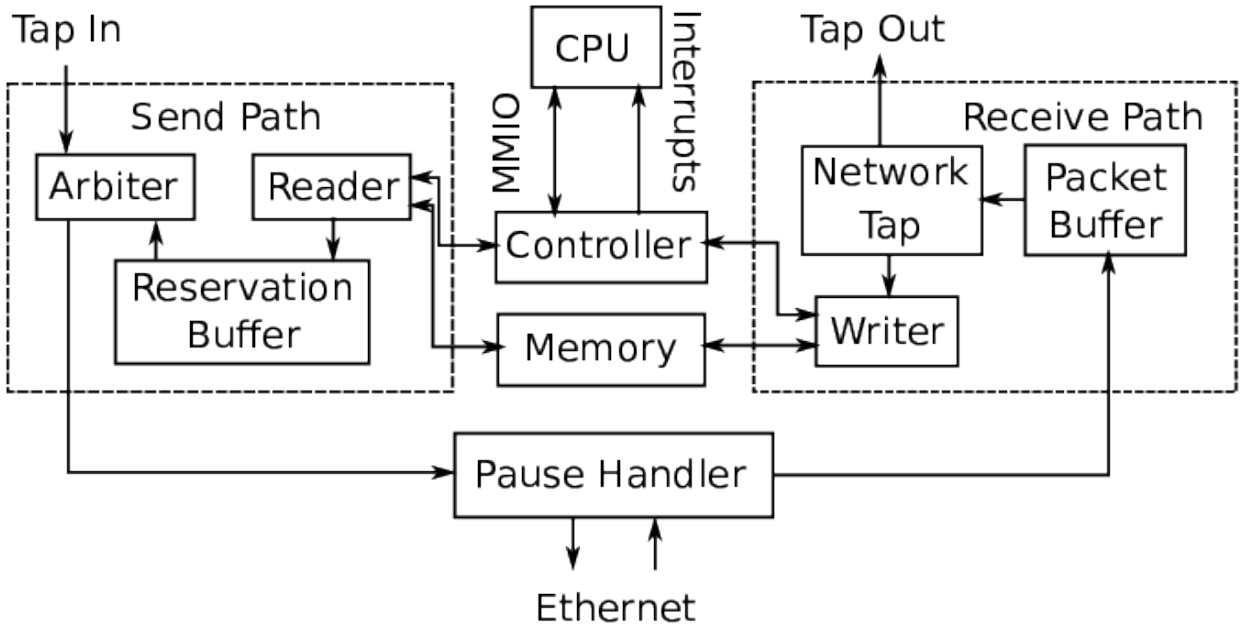
Alternative SoC Configs

The Gemmini generator includes additional alternative SoC configs (configs that are not in the Chipyard example project). If you would like to build one of these alternative SoC configurations which are defined in within the Gemmini project repository, you can run the following commands. These commands are similar to the one required when building a simulation from the example project, but they specify that the location of the configs are in the Gemmini subproject, as opposed to the Chipyard example project:

```
cd sims/verilator # or "cd sims/vcs"
make CONFIG=GemminiAcceleratorConfig CONFIG_PACKAGE=gemmini MODEL_PACKAGE=freechips.
↪rocketchip.system GENERATOR_PACKAGE=freechips.rocketchip.system␣
↪TOP=ExampleRocketSystem
```

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.

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.

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*, 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.

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.

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.

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:

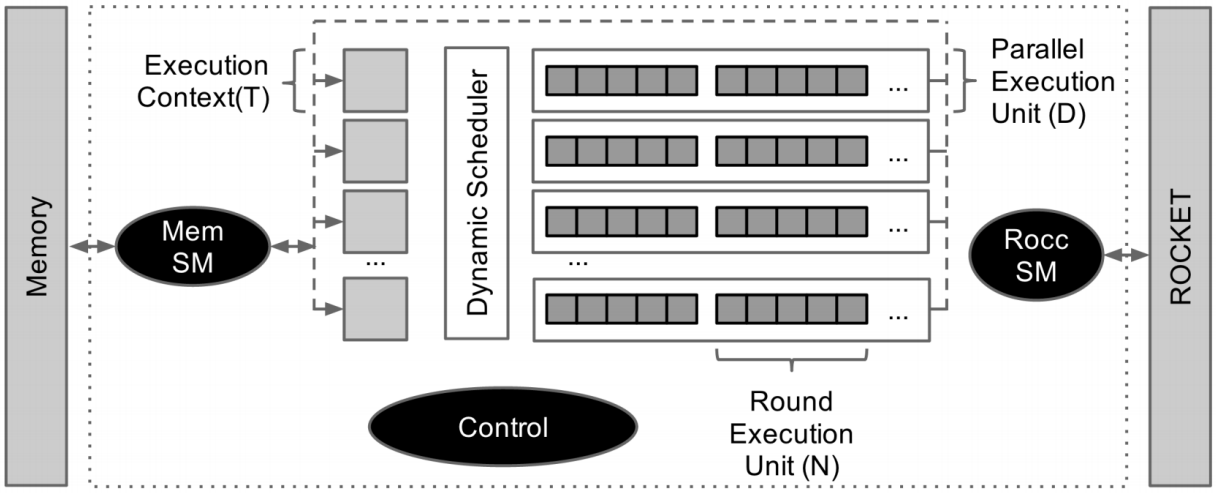
Some of the devices in `sifive-blocks` (such as GPIO) may already have pre-defined config fragments within the Chipyard example project. You may be able to use these config fragments directly, but you should be aware of their addresses within the SoC address map.

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:

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.

Ariane Core

Ariane is a 6-stage in-order scalar processor core, originally developed at ETH-Zurich by F. Zaruba and L. Benini. The *Ariane core* is wrapped in an *Ariane 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 Ariane 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.

1.6.4 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.

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.

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.

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.

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.

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

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. 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 to 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. 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.

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 you the 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 test harness) consistent after synthesis and place-and-route, which allows the RTL simulation test harness to be reused.

1.6.5 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.

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 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*.

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.

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.

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
- 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.

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.

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
  - 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.v
  - Verilog wrapper around the accelerator and dummy hard macro.
- example.yml
  - Hammer IR for this tutorial.
- extra_libraries
  - Contains collateral for the dummy hard macro.
- 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 and gdspy packages
- Genus, Innovus, and Calibre licenses
- For ASAP7 specifically:
  - Download the ASAP7 PDK tarball to a directory of choice but do not extract it. The tech plugin is configured to extract the PDK into a cache directory for you.
  - If you have additional ASAP7 hard macros, their LEF & GDS need to be 4x upscaled @ 4000 DBU precision. They may live outside `extra_libraries` at your discretion.
  - Innovus version must be >= 15.2 or <= 18.1 (ISRs excluded).

Initial Setup

In the Chipyard root, run:

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

to pull the Hammer & plugin submodules. Note that for technologies other than `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 `Sha3RocketConfig` (Rocket Chip w/ the accelerator) and set up all prerequisites for the build system to push just the accelerator + hard macro through the flow:

```
make buildfile MACROCOMPILER_MODE='--mode synflops' CONFIG=Sha3RocketConfig VLSI_
↪TOP=Sha3AccelwBB
```

The `MACROCOMPILER_MODE='--mode synflops'` is needed because the ASAP7 process does not yet have a memory compiler, so flip-flop arrays are used instead. This will dramatically increase the synthesis runtime if your design has a lot of memory state (e.g. large caches). This change is automatically inferred by the makefile but is included here for completeness.

The `CONFIG=Sha3RocketConfig` selects the target generator config in the same manner as the rest of the Chipyard framework. This elaborates a Rocket Chip with the Sha3Accel module.

The `VLSI_TOP=Sha3AccelwBB` indicates that we are only interested in physical design of the accelerator block. If this variable is not set, the entire SoC will be pushed through physical design. Note that you should not set the `TOP` variable because it is used during Chisel elaboration.

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.

The `scale_final_gds` hook is a particularly powerful hook. It dumps a Python script provided by the ASAP7 tech plugin, an executes it within the Innovus TCL interpreter, and should be inserted after `write_design`. This hook is necessary because the ASAP7 PDK does place-and-route using 4x upscaled LEFs for Innovus licensing reasons, thereby requiring the cells created in the post-P&R GDS to be scaled down by a factor of 4.

example.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 of where the downloaded the ASAP7 PDK tarball lives.

Synthesis

```
make syn
```

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

Place-and-Route

```
make par
```

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):

```
python3 view_gds.py build/par-rundir/Sha3AccelwBB.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
./build/drc-rundir/generated-scripts/view-drc
make lvs
./build/lvs-rundir/generated-scripts/view-lvs
```

Some DRC errors are expected from this PDK, as explained in the ASAP7 plugin readme.

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'
```

RTL and Gate-level Simulation

With the Synopsys plugin, RTL and gate-level simulation is supported using VCS. While this example does not implement any simulation, refer to Hammer's documentation for how to set it up for your design.

1.6.6 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 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
- 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!

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 `WithNBoomCores(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.

In this example, the `WithNBoomCores` and `WithNBigCores` config fragments set up the default parameters for the multiple BOOM and Rocket cores, respectively. However, for BOOM, an extra config fragment called `WithLargeBooms` is added to override the default parameters with a different set of more common default parameters. This config fragment applies to all BOOM cores in the system and changes the parameters for each.

Great! Now you have a heterogeneous setup with BOOMs and Rockets. The final thing you need to make this system work is to renumber the `hartId`'s of the cores so that each core has a unique `hartId` (a `hartId` is the hardware thread id of the core). The `WithRenumberHarts` config fragment solves this by assigning a unique `hartId` to all cores in the system (it can label the Rocket cores first or the BOOM cores first). The reason this is needed is because by default the `WithN...Cores(X)` config fragment assumes that there are only BOOM or only Rocket cores in the system. Thus, without the `WithRenumberHarts` config fragment, each set of cores is labeled starting from zero causing multiple cores to be assigned the same `hartId`.

Another alternative option to create a multi heterogeneous core system is to override the parameters yourself so you can specify the core parameters per core. The config fragment to add to your system would look something like the following.

```
// create 6 cores (4 boom and 2 rocket)
class WithHeterCoresSetup extends Config((site, here, up) => {
  case BoomTilesKey => {
    val boomTile0 = BoomTileParams(...) // params for boom core 0
    val boomTile1 = BoomTileParams(...) // params for boom core 1
    val boomTile2 = BoomTileParams(...) // params for boom core 2
    val boomTile3 = BoomTileParams(...) // params for boom core 3
    boomTile0 ++ boomTile1 ++ boomTile2 ++ boomTile3
  }

  case RocketTilesKey => {
    val rocketTile0 = RocketTileParams(...) // params for rocket core 0
    val rocketTile1 = RocketTileParams(...) // params for rocket core 1
    rocketTile0 ++ rocketTile1
  }
})
```

Then you could use this new config fragment like the following.

```
class SixCoreConfig extends Config(
  new WithTSI ++
  new WithNoGPIO ++
  new WithBootROM ++
  new WithUART ++
  new freechips.rocketchip.subsystem.WithNoMMIOPort ++
  new freechips.rocketchip.subsystem.WithNoSlavePort ++
  new WithHeterCoresSetup ++
  new freechips.rocketchip.system.BaseConfig)
```

Note, in this setup you need to specify the `hartId` of each core in the "TileParams", where each `hartId` is unique.

[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).

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.

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/ConfigFragments.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.

In this example, the `WithRenumberHarts` relabels the `hartId`'s of all the BOOM/Rocket cores. Then after that is applied to the parameters, the `WithMultiRoCCHwacha` config fragment assigns a Hwacha accelerator to a particular `hartId` (in this case, the `hartId` of `2` 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,...)`).

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"

version := "1.0"

name := "yourproject"
```

(continues on next page)

(continued from previous page)

```
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)
```

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.

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)
}
```

(continues on next page)

(continued from previous page)

```
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`.

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/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/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`.

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.

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.

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.

Constructing the Top and Config

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

```
class Top(implicit p: Parameters) extends System
  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.CanHavePeripherySerial // Enables optionally adding the TSI serial-
↪adapter and port
  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 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
{
  override lazy val module = new TopModule(this)
}

class TopModule[+L <: Top](l: L) extends SystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with testchipip.CanHavePeripheryBlockDeviceModuleImp
  with testchipip.CanHavePeripherySerialModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with icenet.CanHavePeripheryIceNICModuleImp
  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 `Top` class contains the set of traits which parameterize and define the `Top`. Typically these traits will optionally add IOs or peripherals to the `Top`. The `Top` class includes the pre-elaboration code and also a `lazy val` to produce the module implementation (hence `LazyModule`). The `TopModule` class is the actual RTL that gets synthesized.

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

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
  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
```

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).

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.

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 `Top` 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 `Top`, and enables compartmentalization of code for different custom blocks.

Top-level traits specify that the `Top` 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 `Top` 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 `Top` to instantiate an additional port and concretely connect it to the GCD module.

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

```
class Top(implicit p: Parameters) extends System
  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.CanHavePeripherySerial // Enables optionally adding the TSI serial-
↪adapter and port
  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 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
{
  override lazy val module = new TopModule(this)
}

class TopModule[+L <: Top](l: L) extends SystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with testchipip.CanHavePeripheryBlockDeviceModuleImp
  with testchipip.CanHavePeripherySerialModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with icenet.CanHavePeripheryIceNICModuleImp
  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.

We can use this config fragment when composing our configs.

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

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.

```
class Top(implicit p: Parameters) extends System
  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.CanHavePeripherySerial // Enables optionally adding the TSI serial-
↪adapter and port
  with sifive.blocks.devices.uart.HasPeripheryUART // Enables optionally adding the␣
↪sifive UART
```

(continues on next page)

(continued from previous page)

```
  with sifive.blocks.devices.gpio.HasPeripheryGPIO // Enables optionally adding the␣
↪sifive GPIOs
  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
{
  override lazy val module = new TopModule(this)
}

class TopModule[+L <: Top](l: L) extends SystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with testchipip.CanHavePeripheryBlockDeviceModuleImp
  with testchipip.CanHavePeripherySerialModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
  with icenet.CanHavePeripheryIceNICModuleImp
  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.

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/
```

(continues on next page)

(continued from previous page)

```
            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/
            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

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.

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.

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
  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`

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.

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

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.

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.

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.

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)
        cd $(base_dir) && $(SBT) "project tapeout" "runMain barstools.tapeout.
↪transforms.GenerateTopAndHarness -o $(TOP_FILE) -tho $(HARNESS_FILE) -i $(FIRRTL_
↪FILE) --syn-top $(TOP) --harness-top $(VLOG_MODEL) -faf $(ANNO_FILE) -tsaof $(TOP_
↪ANNO) -tdf $(sim_top_blackboxes) -tsf $(TOP_FIR) -thaof $(HARNESS_ANNO) -hdf $(sim_
↪harness_blackboxes) -thf $(HARNESS_FIR) $(REPL_SEQ_MEM) $(HARNESS_CONF_FLAGS) -td
↪$(build_dir)" && 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)

    // 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)
        cd $(base_dir) && $(SBT) "project tapeout" "runMain barstools.tapeout.
↪transforms.GenerateTopAndHarness -o $(TOP_FILE) -tho $(HARNESS_FILE) -i $(FIRRTL_
↪FILE) --syn-top $(TOP) --harness-top $(VLOG_MODEL) -faf $(ANNO_FILE) -tsaof $(TOP_
↪ANNO) -tdf $(sim_top_blackboxes) -tsf $(TOP_FIR) -thaof $(HARNESS_ANNO) -hdf $(sim_
↪harness_blackboxes) -thf $(HARNESS_FIR) $(REPL_SEQ_MEM) $(HARNESS_CONF_FLAGS) -td
↪$(build_dir)" && 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.

IOBinders

In Chipyard we use a special `Parameters` key, `IOBinders` to determine what modules to bind to the IOs of a `Top` in the `TestHarness`.

```
case object IOBinders extends Field[Map[String, (Clock, Bool, Bool, Any) =>␣
↪Seq[Any]]](
  Map[String, (Clock, Bool, Bool, Any) => Seq[Any]]().withDefaultValue((c: Clock, r:␣
↪Bool, s: Bool, t: Any) => Nil)
)

// This macro overrides previous matches on some Top mixin. This is useful for
// binders which drive IO, since those typically cannot be composed
class OverrideIOBinder[T](fn: => (Clock, Bool, Bool, T) => Seq[Any])(implicit tag:␣
↪ClassTag[T]) extends Config((site, here, up) => {
  case IOBinders => up(IOBinders, site) + (tag.runtimeClass.toString ->
      ((clock: Clock, reset: Bool, success: Bool, t: Any) => {
        t match {
          case top: T => fn(clock, reset, success, top)
          case _ => Nil
        }
      })
  )
})

// This macro composes with previous matches on some Top mixin. This is useful for
// annotation-like binders, since those can typically be composed
class ComposeIOBinder[T](fn: => (Clock, Bool, Bool, T) => Seq[Any])(implicit tag:␣
↪ClassTag[T]) extends Config((site, here, up) => {
  case IOBinders => up(IOBinders, site) + (tag.runtimeClass.toString ->
      ((clock: Clock, reset: Bool, success: Bool, t: Any) => {
        t match {
          case top: T => (up(IOBinders, site)(tag.runtimeClass.toString)(clock, reset,
↪ success, top)
            ++ fn(clock, reset, success, top))
          case _ => Nil
        }
```

(continues on next page)

(continued from previous page)

```
      })
  )
})
```

This special key solves the problem of duplicating test-harnesses for each different `Top` type.

You could just as well create a custom harness module that attaches IOs explicitly. Instead, the IOBinders key provides a map from Scala traits to attachment behaviors.

For example, the `WithSimAXIMemTiedOff` IOBinder specifies that any `Top` which matches `CanHaveMasterAXI4MemPortModuleImp` will have a `SimAXIMem` connected.

```
class WithSimAXIMem extends OverrideIOBinder({
  (c, r, s, top: CanHaveMasterAXI4MemPortModuleImp) => top.connectSimAXIMem(); Nil
})
```

These classes are all `Config` objects, which can be mixed into the configs to specify IO connection behaviors.

There are two macros for generating these `Config``s. ``OverrideIOBinder` overrides any existing behaviors set for a particular IO in the `Config` object. This macro is frequently used because typically top-level IOs drive or are driven by only one source, so a composition of `IOBinders` does not make sense. The `ComposeIOBinder` macro provides the functionality of not overriding existing behaviors.

1.6.7 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.

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.

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.

1.6.8 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.

Tops, Test-Harnesses, and the Test-Driver

The three highest levels of hierarchy in a Chipyard SoC are the Top (DUT), `TestHarness`, and the `TestDriver`. The Top and `TestHarness` are both emitted by Chisel generators. The `TestDriver` serves as our testbench, and is a Verilog file in Rocket Chip.

Top/DUT

The top-level module of a Rocket Chip SoC is composed via cake-pattern. Specifically, "Tops" extend 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.

Communicating with the DUT

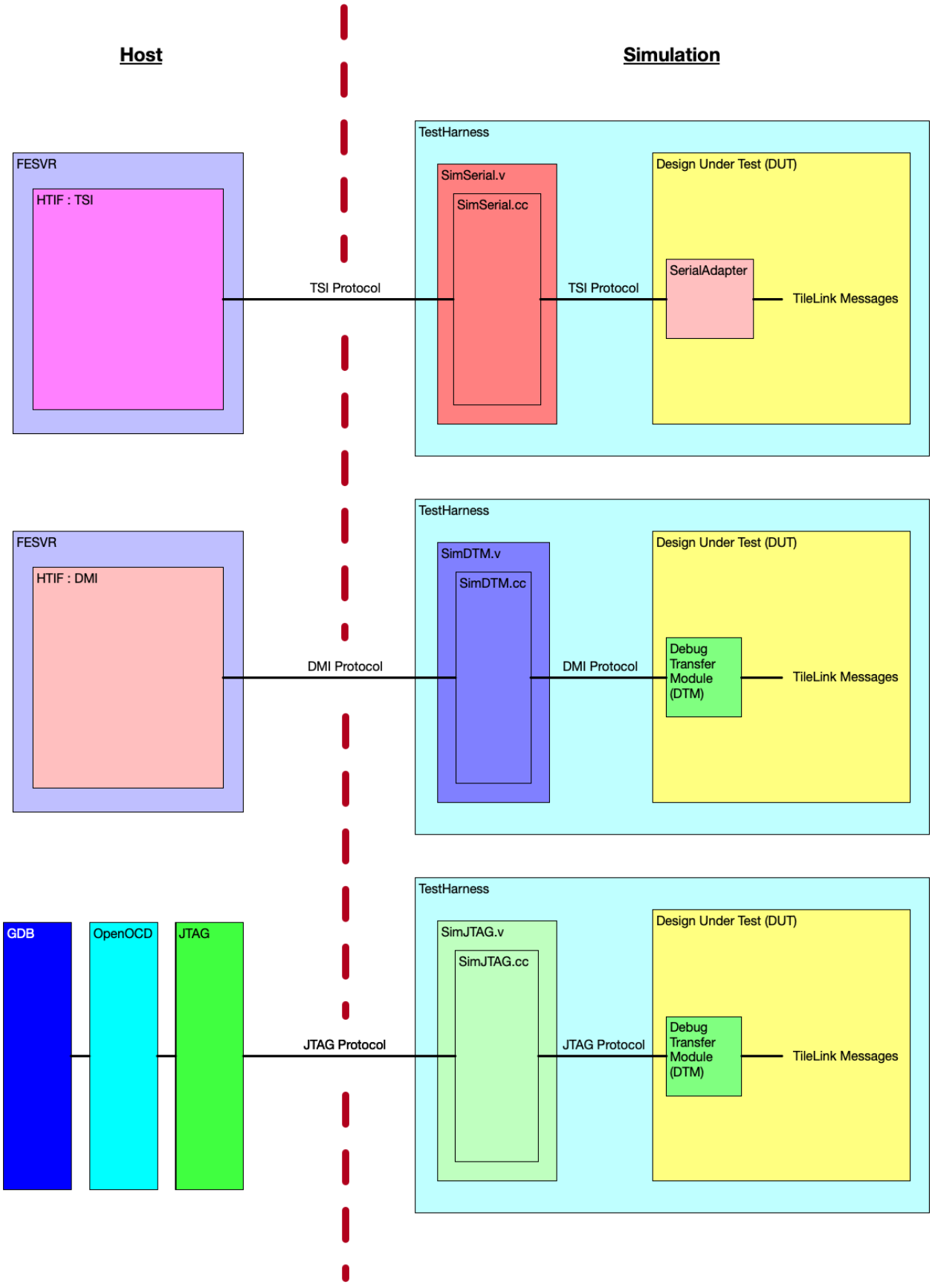
There are two types of DUTs that can be made: *tethered* or *standalone* DUTs. A *tethered* DUT is where a host computer (or just host) must send transactions to the DUT to bringup a program. This differs from a *standalone* DUT

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 by default. 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.


Host
Simulation
FESVR
HTIF : TSI
TSI Protocol
TestHarness
SimSerial.v
SimSerial.cc
TSI Protocol
Design Under Test (DUT)
SerialAdapter
TileLink Messages
FESVR
HTIF : DMI
DMI Protocol
TestHarness
SimDTM.v
SimDTM.cc
DMI Protocol
Design Under Test (DUT)
Debug Transfer Module (DTM)
TileLink Messages
GDB
OpenOCD
JTAG
JTAG Protocol
TestHarness
SimJTAG.v
SimJTAG.cc
JTAG Protocol
Design Under Test (DUT)
Debug Transfer Module (DTM)
TileLink Messages

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)

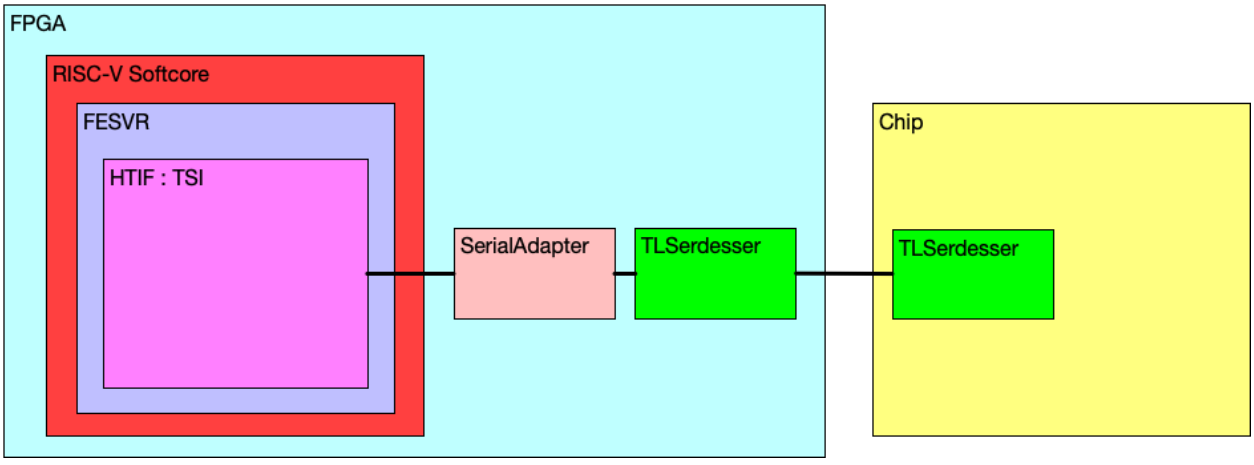
FESVR is a C++ library that manages communication between a host machine and a RISC-V DUT. For debugging, it provides a simple API to reset, send messages, and load/run programs on a DUT. It also emulates peripheral devices. It can be incorporated with simulators (VCS, Verilator, FireSim), or used in 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 probe the DUT's memory space. During simulation, the host sends TSI commands to a simulation stub 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 into the DUT which then converts the TSI command into a TileLink request. This conversion is done by the `SerialAdapter` module (located in the `generators/testchipip` project). 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.

In the case of a chip tapeout bringup, TSI commands can be sent over a custom communication medium to communicate with the chip. For example, some Berkeley tapeouts have a FPGA with a RISC-V soft-core that runs FESVR. The FESVR on the soft-core sends TSI commands to a TSI-to-TileLink converter living on the FPGA (i.e. `SerialAdapter`). 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 serial-link or serdes). Once the serialized transaction is received on the chip, it is deserialized and masters a bus on the chip. The following image shows this flow:



Note: The `TLSerdesser` can also be used as a slave (client), so it can sink memory requests from the chip and connect to off-chip backing memory. Or in other words, `TLSerdesser` creates a bi-directional TileLink interface.

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. Thus, Chipyard removes the DTM by default in favor of the TSI protocol for DUT communication.

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=LargeBoomConfig run-asm-tests
```

FireSim FPGA-accelerated simulations use TSI by default as well.

If you would like to build and simulate a Chipyard configuration with a DTM configured for DMI communication, then you must tie-off the TSI interface, and instantiate the *SimDTM*. Note that we use *WithTiedOffSerial* ++ *WithSimDebug* instead of *WithTiedOffDebug* ++ *WithSimSerial*.

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

The main way to use JTAG with a Rocket Chip based system is to instantiate the Debug Transfer Module (DTM) and configure it to use a JTAG interface (by default the DTM is setup to use the DMI interface mentioned above).

Creating a DTM+JTAG Config

First, a DTM config must be created for the system that you want to create. This step is similar to the DMI simulation section within the *Starting the TSI or DMI Simulation* section. The configuration is very similar to a DMI-based configuration. The main difference is the addition of the `WithJtagDTM` config fragment that configures the instantiated DTM to use the JTAG protocol as the bringup method.

Building a DTM+JTAG Simulator

After creating the config, call the `make` command like the following to build a simulator for your RTL:

```
cd sims/verilator
# or
cd sims/vcs

make CONFIG=jtagRocketConfig
```

In this example, the simulation will use the config that you previously specified, as well as set the other parameters that are needed to satisfy the build system. After that point, you should have a JTAG enabled simulator that you can attach to using OpenOCD and GDB!

Debugging with JTAG

Please refer to the following resources on how to debug with JTAG.

- https://github.com/chipsalliance/rocket-chip#-debugging-with-gdb
- https://github.com/riscv/riscv-isa-sim#debugging-with-gdb

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 `utilities` directory.

Firesim Debugging

Chisel printfs, asserts, and waveform generation are also available in FireSim FPGA-accelerated simulation. See the FireSim documentation for more detail.

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)
```

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) => {
```

(continues on next page)

(continued from previous page)

```
  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))
```

(continues on next page)

(continued from previous page)

```
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`.

1.6.9 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.

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:

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:

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.

Now we instantiate these two clients in another lazy module and expose their nodes as a single node.

We can also do the same for managers.

If we want to connect the client and manager groups together, we can now do this.

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.

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.

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.

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

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, `smallReg0` will get the value 0xB and `smallReg1` 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.

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.

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.

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.

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 different 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`.

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.

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(
  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.
- `holdFirstDenied: 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*, except it can only break multi-beat AXI4 transactions into single-beat transactions. This effectively serves as an AXI4 to AXI4-Lite converter. The constructor for this widget does not take any arguments.

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:

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

// 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)))
```

(continues on next page)

(continued from previous page)

```
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.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search