

Configuring and Building Custom RISC-V SoCs in Chipyard

Jerry Zhao

UC Berkeley

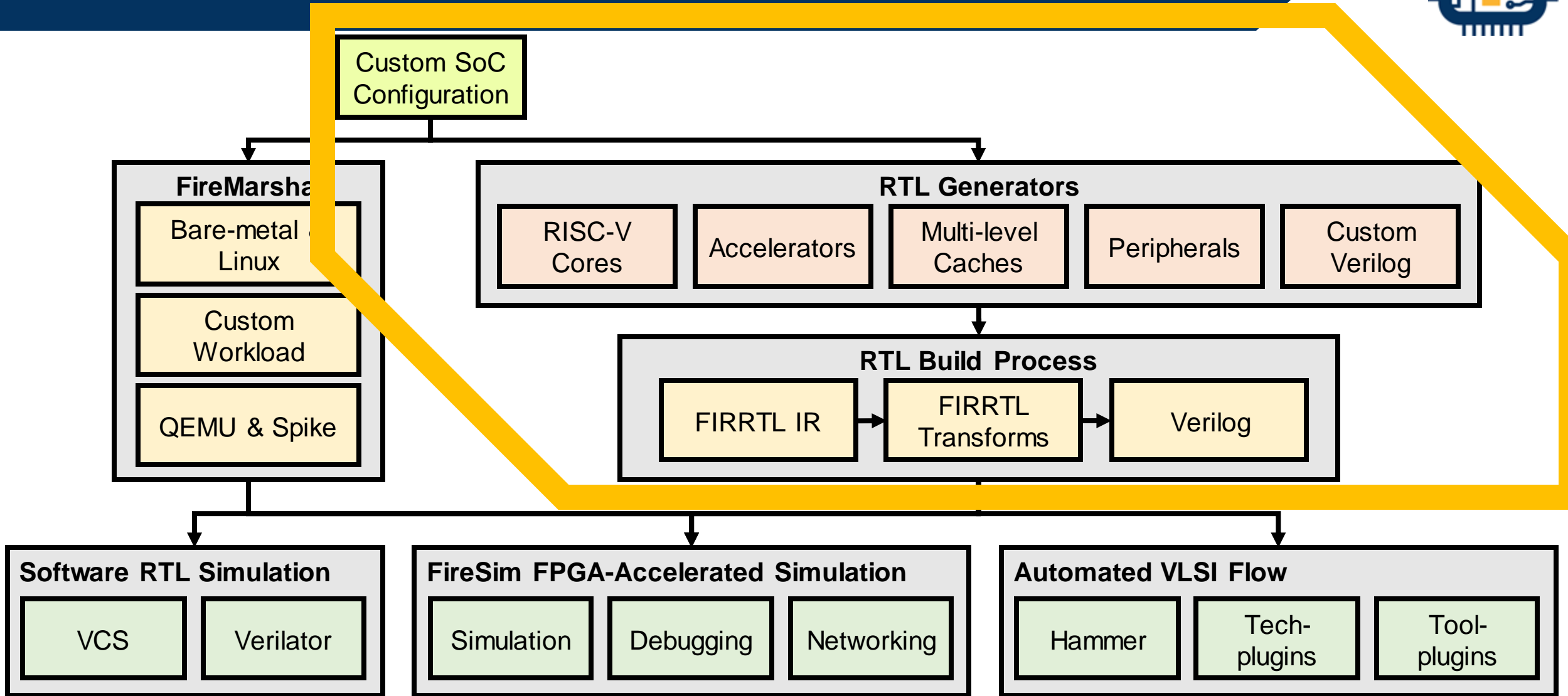
jzh@berkeley.edu



Berkeley
Architecture
Research

CHIPYARD

Tutorial Roadmap



Outline



- **Configure a custom SoC** with heterogeneous cores
- **Generate Verilog RTL** of the complete custom SoC
- **Execute RTL simulations** running RISC-V binaries
- **Integrate a custom SHA3 RoCC accelerator**, and run binaries with SHA3 acceleration



How things will work



```
# command 1
> echo "Chipyard Rules!"

# command 2
> do_this arg1 arg2
```

Terminal Section

```
// SOME COMMENT HERE
class SmallBoomConfig extends Config(
  new WithTop ++
  new WithBootROM ++
  new boom.common.WithSmallBooms ++
  new boom.common.WithNBoomCores(1) ++
  new freechips.rocketchip.system.BaseConfig)
```

Inside-a-File Section



Example



```
# (ALREADY DONE BEFORE) ssh/mosh into your EC2
> ssh -i <PEM file> centos@<IP ADDR>
> mosh -ssh="ssh -i <PEM file>" centos@<IP ADDR>

# (ALREADY DONE BEFORE) ensure RISC-V toolchain is in PATH
> cd ~/chipyard-morning/sims/firesim
> source sourceme-f1-manager.sh

# return to chipyard
> cd ~/chipyard-morning
> ls
```



Directory Structure



chipyard-morning/

generators/ ←

Our library of Chisel generators

chipyard/

sha3/

sims/ ←

Utilities for simulating SoCs

verilator/

firesim/

fpga/ ←

Utilities for FPGA prototyping

software/ ←

Utilities for building RISC-V software

vlsi/ ←

HAMMER VLSI Flow

toolchains/ ←

RISC-V Toolchain





Build and simulate a heterogeneous BOOM + Rocket SoC



Example



```
# open up the heterogeneous configurations file
> cd generators/chipyard/src/main/scala/config
> vim TutorialConfigs.scala
```

```
chipyard-morning/
  generators/
    chipyard/
      src/main/scala/config/
        TutorialConfigs.scala
```



How Configs Work



```
class TutorialStarterConfig extends Config(  
  new boom.common.WithNSmallBooms(1) ++  
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++  
  new freechips.rocketchip.subsystem.WithInclusiveCache() ++  
  new chipyard.config.AbstractConfig)
```

Bottom-to-top (right-to-left) config hierarchy

- `AbstractConfig` sets up “default” system configuration
- `WithInclusiveCache` adds coherent L2
- `WithNBigCores` adds Rocket Tiles
- `WithNSmallCores` adds small BOOM Tiles

Edit this config to customize your SoC

- 1 BOOM + 1 Rocket recommended

```
chipyard-morning/  
  generators/  
    chipyard/  
      src/main/scala/config/  
        TutorialConfigs.scala
```



Example



```
# navigate to the Verilator
> cd ~/chipyard-morning/sims/verilator

# start the Verilator RTL simulator build
> make CONFIG=TutorialStarterConfig

# this will take a few minutes!
# Ignore [error] Picked up JAVA_TOOL_OPTIONS ...
```

```
chipyard-morning/
  generators/
    chipyard/
      src/main/scala/config/
        TutorialConfigs.scala
  sims/
    verilator/
```



How Configs Work



```
class TutorialStarterConfig extends Config(  
  new boom.common.WithNSmallBooms(1) ++  
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++  
  new freechips.rocketchip.subsystem.WithInclusiveCache() ++  
  new chipyard.config.AbstractConfig)
```

- Bottom-to-top (right-to-left) config hierarchy
- Configs can read/override other configs

```
TilesKey      => Nil  
BankedL2Key   => BroadcastManager
```

```
chipyard-morning/  
  generators/  
    chipyard/  
      src/main/scala/config/  
        TutorialConfigs.scala
```



How Configs Work



```
class TutorialStarterConfig extends Config(  
  new boom.common.WithNSmallBooms(1) ++  
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++  
  new freechips.rocketchip.subsystem.WithInclusiveCache() ++  
  new chipyard.config.AbstractConfig)
```

- Bottom-to-top (right-to-left) config hierarchy
- Configs can read/override other configs

```
TilesKey      => Nil  
BankedL2Key   => InclusiveCache
```

```
chipyard-morning/  
  generators/  
    chipyard/  
      src/main/scala/config/  
        TutorialConfigs.scala
```



How Configs Work



```
class TutorialStarterConfig extends Config(  
  new boom.common.WithNSmallBooms(1) ++  
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++  
  new freechips.rocketchip.subsystem.WithInclusiveCache() ++  
  new chipyard.config.AbstractConfig)
```

- Bottom-to-top (right-to-left) config hierarchy
- Configs can read/override other configs

```
TilesKey      => Seq(RocketTile)  
BankedL2Key   => InclusiveCache
```

```
chipyard-morning/  
  generators/  
    chipyard/  
      src/main/scala/config/  
        TutorialConfigs.scala
```



How Configs Work



```
class TutorialStarterConfig extends Config(  
  new boom.common.WithNSmallBooms(1) ++  
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++  
  new freechips.rocketchip.subsystem.WithInclusiveCache() ++  
  new chipyard.config.AbstractConfig)
```

- Bottom-to-top (right-to-left) config hierarchy
- Configs can read/override other configs

```
TilesKey      => Seq(RocketTile, BoomTile)  
BankedL2Key   => InclusiveCache
```

```
chipyard-morning/  
  generators/  
    chipyard/  
      src/main/scala/config/  
        TutorialConfigs.scala
```



Diving into AbstractConfig



```
class AbstractConfig extends Config{
```

- AbstractConfig provides “skeleton” base configuration of a SoC
- Adds generally useful features (UART, BootROM, etc.), but no cores
- Standard Chipyard SoCs inherit AbstractConfig settings

```
chipyard-morning/  
  generators/  
    chipyard/  
      src/main/scala/config/  
        AbstractConfig.scala  
        TutorialConfigs.scala
```



Diving into AbstractConfig



```
// The HarnessBinders control generation of hardware in the TestHarness
new chipyard.harness.WithUARTAdapter ++ // add UART adapter to display UART on stdout, if uart is present
new chipyard.harness.WithBlackBoxSimMem ++ // add SimDRAM DRAM model for axi4 backing memory, if axi4 mem is enabled
new chipyard.harness.WithSimSerial ++ // add external serial-adapter and RAM
new chipyard.harness.WithSimDebug ++ // add SimJTAG or SimDTM adapters if debug module is enabled
new chipyard.harness.WithGPIOTieOff ++ // tie-off chiptop GPIOs, if GPIOs are present
new chipyard.harness.WithSimSPIFlashModel ++ // add simulated SPI flash memory, if SPI is enabled
new chipyard.harness.WithSimAXIMMIO ++ // add SimAXIMem for axi4 mmio port, if enabled
new chipyard.harness.WithTieOffInterrupts ++ // tie-off interrupt ports, if present
new chipyard.harness.WithTieOffFL2FBUSAXI ++ // tie-off external AXI4 master, if present
new chipyard.harness.WithTieOffCustomBootPin ++
```

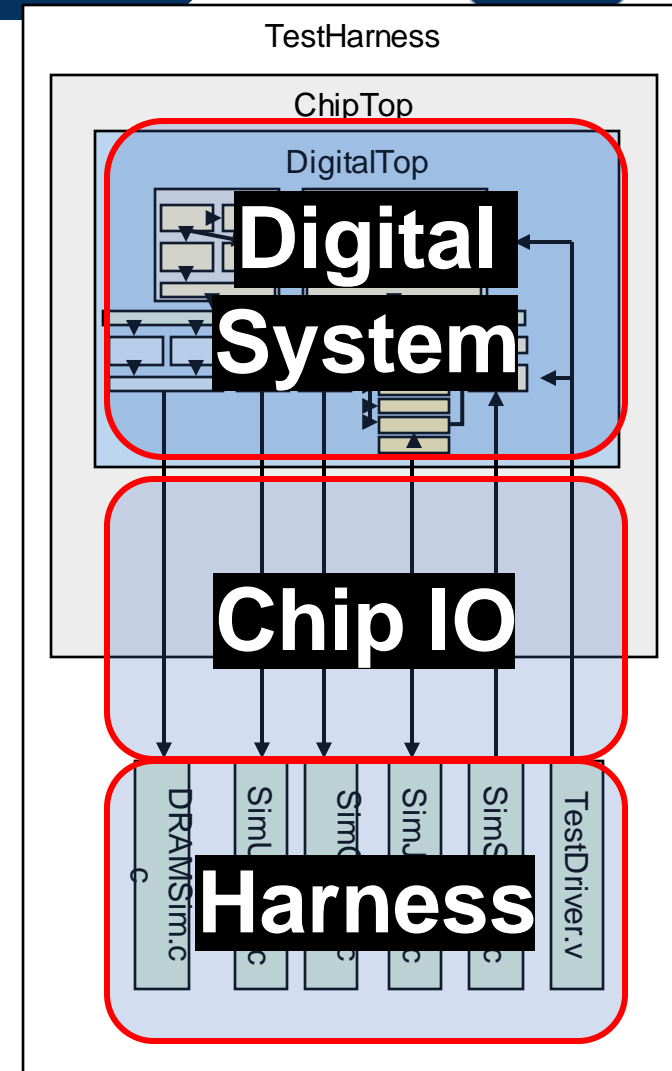
Configuring RTL in the TestHarness

```
// The IOBinders instantiate Chiptop I/Os to match desired digital I/Os
// IOCells are generated for "Chip-like" I/Os, while simulation-only I/Os are directly punched through
new chipyard.iobinders.WithAXI4MMIOpunchthrough ++
new chipyard.iobinders.WithAXI4MMIOpunchthrough ++
new chipyard.iobinders.WithL2FBUSAXI4punchthrough ++
new chipyard.iobinders.WithBlockDeviceIOPunchthrough ++
new chipyard.iobinders.WithNICIOPunchthrough ++
new chipyard.iobinders.WithSerialTLIOCells ++
new chipyard.iobinders.WithDebugIOCells ++
new chipyard.iobinders.WithUARTIOCells ++
new chipyard.iobinders.WithGPIOCells ++
new chipyard.iobinders.WithUARTIOCells ++
new chipyard.iobinders.WithSPIIOCells ++
new chipyard.iobinders.WithTraceIOPunchthrough ++
new chipyard.iobinders.WithExtInterruptIOCells ++
new chipyard.iobinders.WithCustomBootPin ++
```

Configuring I/Os in the ChipTop

```
new testchipip.WithDefaultSerialTL ++ // use serialized tilelink port to external serialadapter/harnessRAM
new chipyard.config.WithBootROM ++ // use default bootrom
new chipyard.config.WithUART ++ // add a UART
new chipyard.config.WithL2TLBs(1024) ++ // use L2 TLBs
new chipyard.config.WithNoSubsystemDrivenClocks ++ // drive the subsystem diplomatic clocks from Chiptop instead of using im1
new chipyard.config.WithInheritBusFrequencyAssignments ++ // Unspecified clocks within a bus will receive the bus frequency if set
new chipyard.config.WithPeripheryBusFrequencyAsDefault ++ // Unspecified frequencies with match the bus frequency (which is always s
new chipyard.config.WithMemoryBusFrequency(100.0) ++ // Default 100 MHz mbus
new chipyard.config.WithPeripheryBusFrequency(100.0) ++ // Default 100 MHz pbus
new freechips.rocketchip.subsystem.WithJtagDTM ++ // set the debug module to expose a JTAG port
new freechips.rocketchip.subsystem.WithNoMMIOPort ++ // no top-level MMIO master port (overrides default set in rocketchip)
new freechips.rocketchip.subsystem.WithNoSlavePort ++ // no top-level MMIO slave port (overrides default set in rocketchip)
new freechips.rocketchip.subsystem.WithInclusiveCache ++ // use Sifive L2 cache
new freechips.rocketchip.subsystem.WithNextTopInterrupts(0) ++ // no external interrupts
new chipyard.config.WithMultiLockCoherentBusTopology ++ // hierarchical buses including mbus+l2
new freechips.rocketchip.system.BaseConfig) // "base" rocketchip system
```

Configuring Digital System components



Behind the Scenes: Make



- `make` is the main system to combine everything
 - Invokes the Scala Build Tool (`sbt`)
 - Invokes all of the simulator builds (`VCS` and `Verilator`)
 - Automatically keeps track of file dependencies for you!
- Useful `make` targets
 - `verilog`: Verilog only
 - `sim`: RTL simulator
 - `run-binary`: Runs RTL simulation after loading `BINARY`
 - `run-bmark-tests`: Runs simple fast RISC-V benchmark tests



Building a SW Simulator

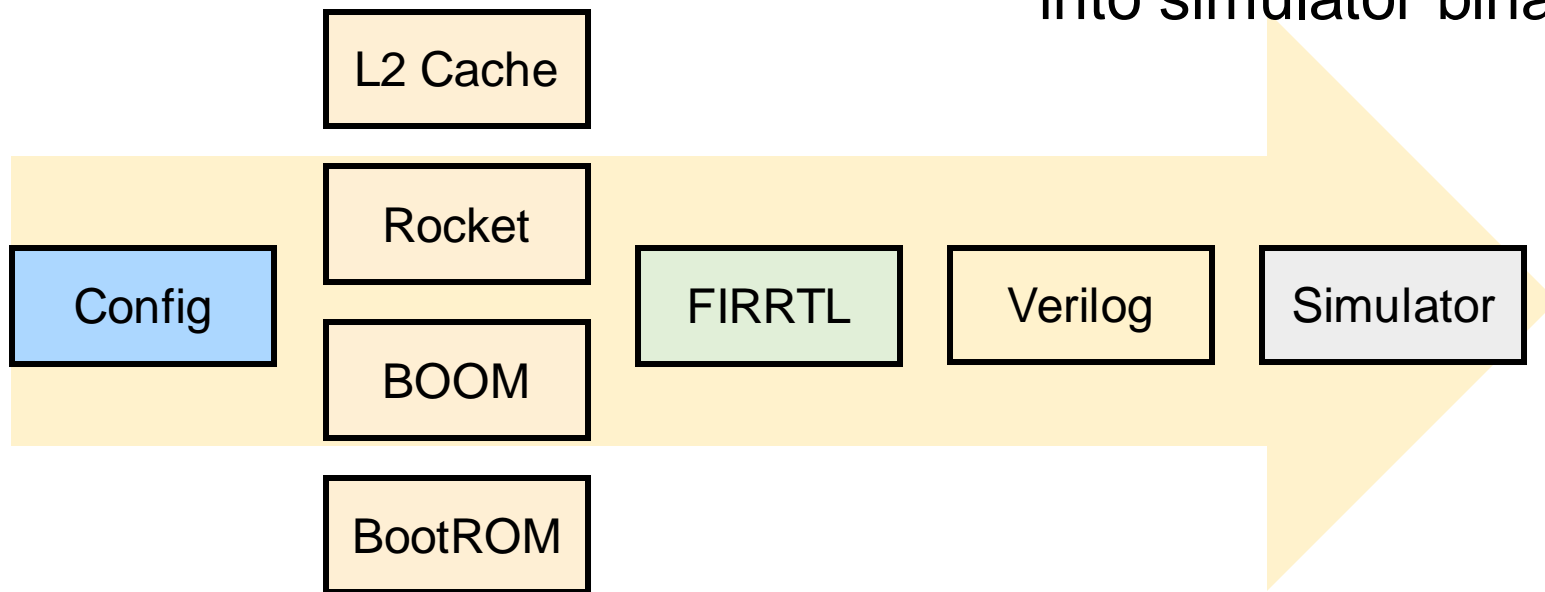


1. Configs parameterize Chisel generators

2. Chisel elaborates into FIRRTL

3. FIRRTL elaborates into Verilog

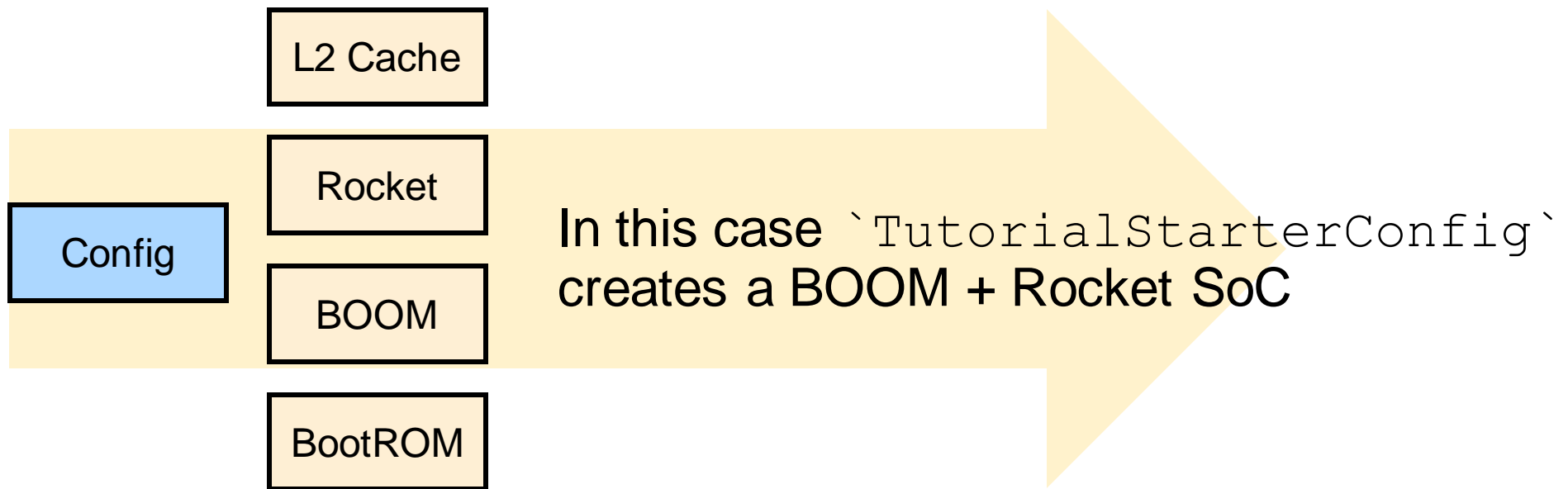
4. Verilator compiles Verilog into simulator binary



Building a SW Simulator



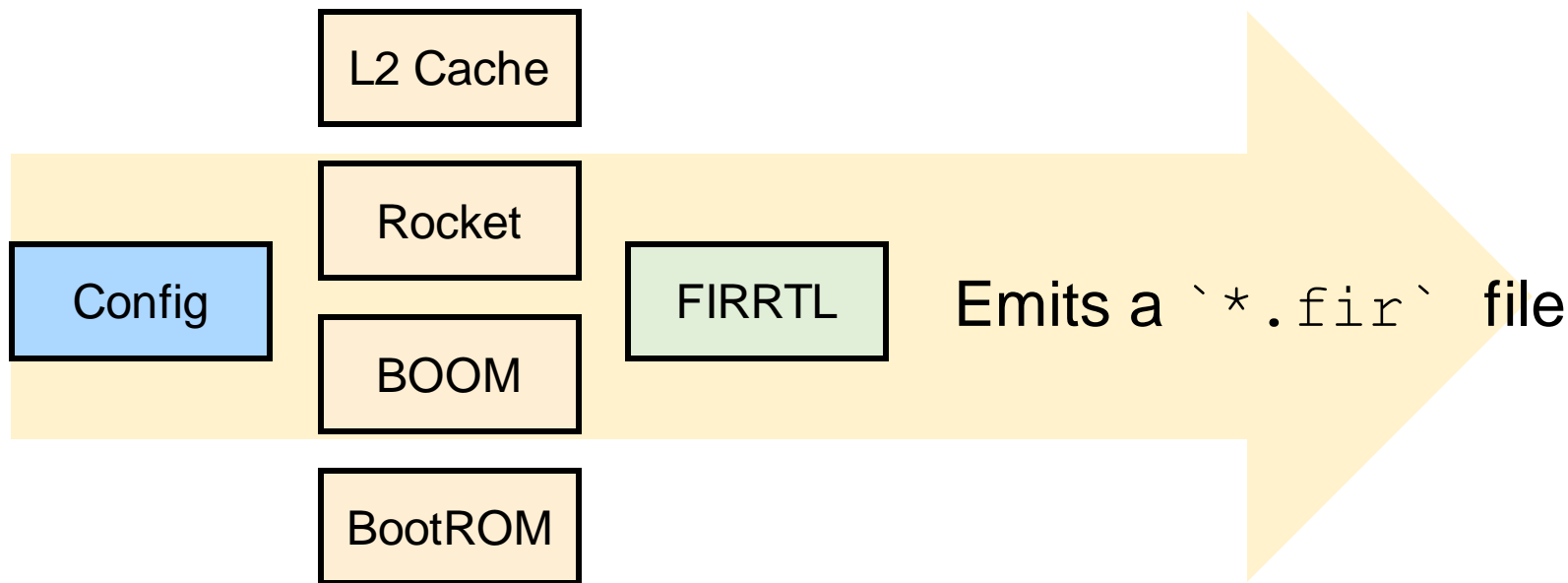
1. Configs parameterize Chisel generators



Building a SW Simulator



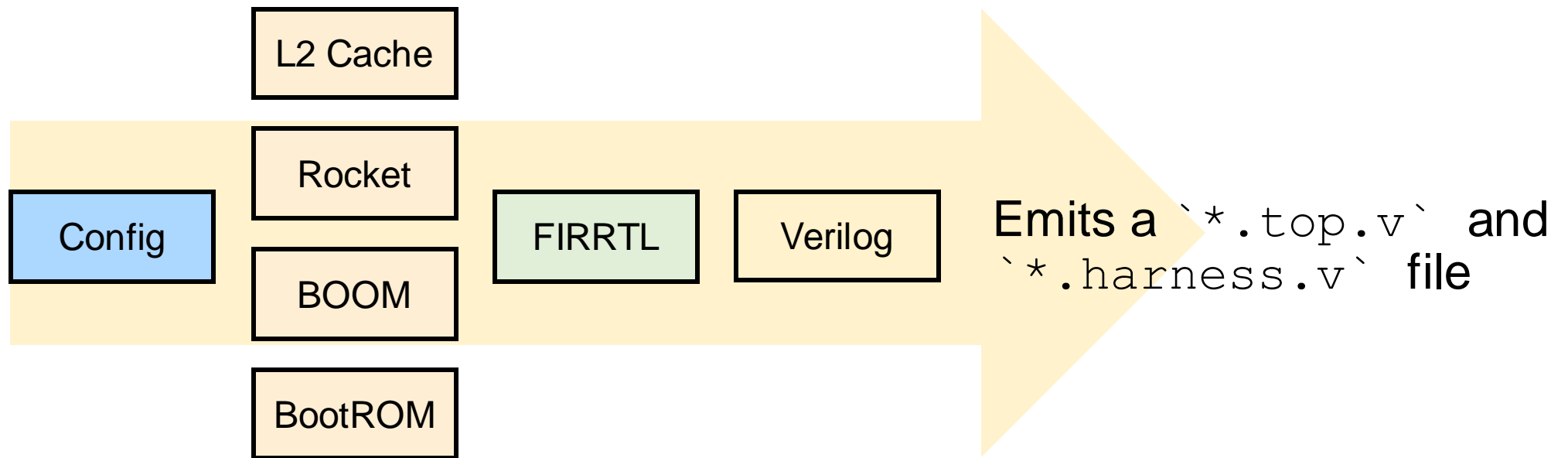
1. Configs parameterize Chisel generators
2. Chisel elaborates into FIRRTL



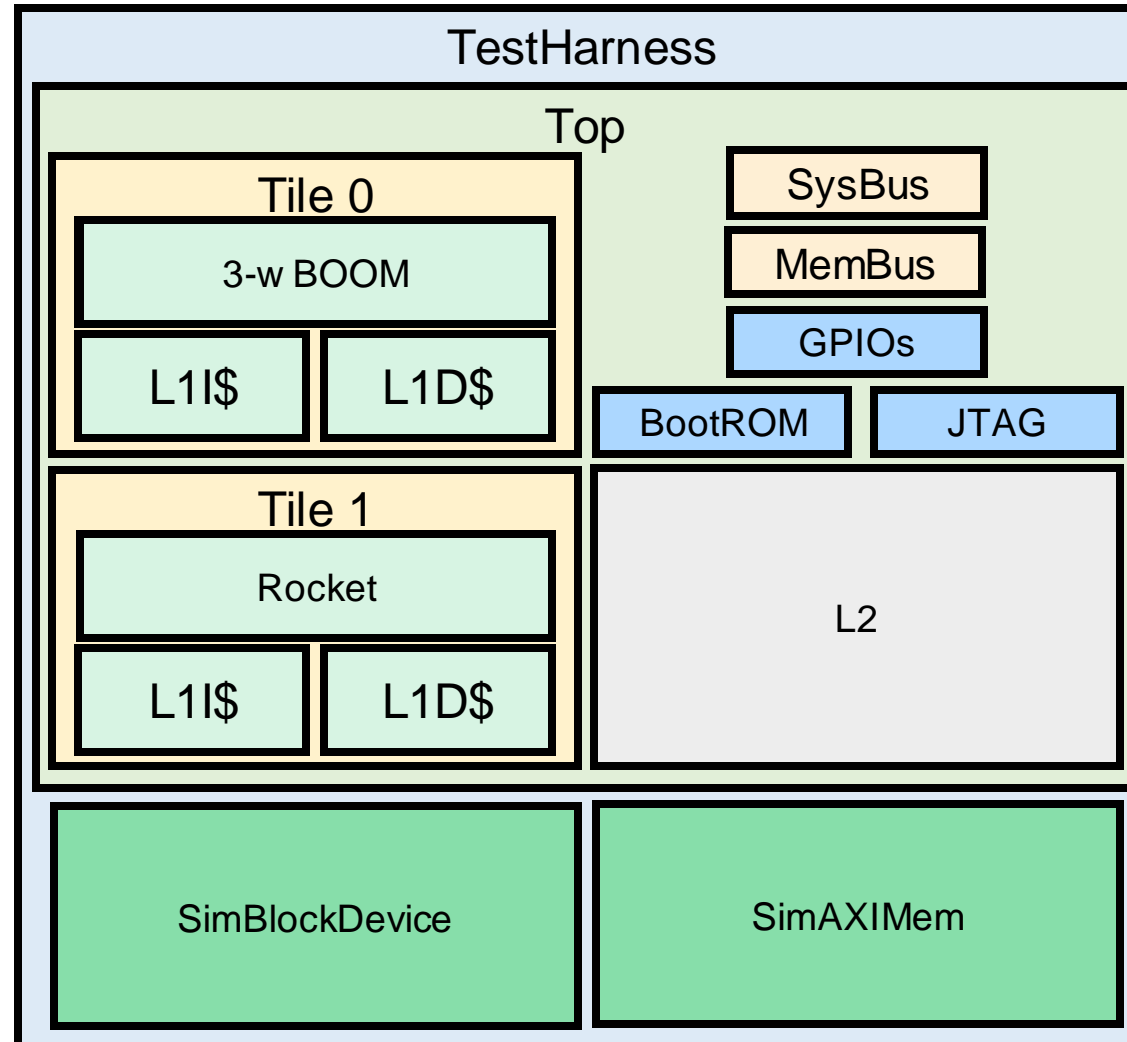
Building a SW Simulator



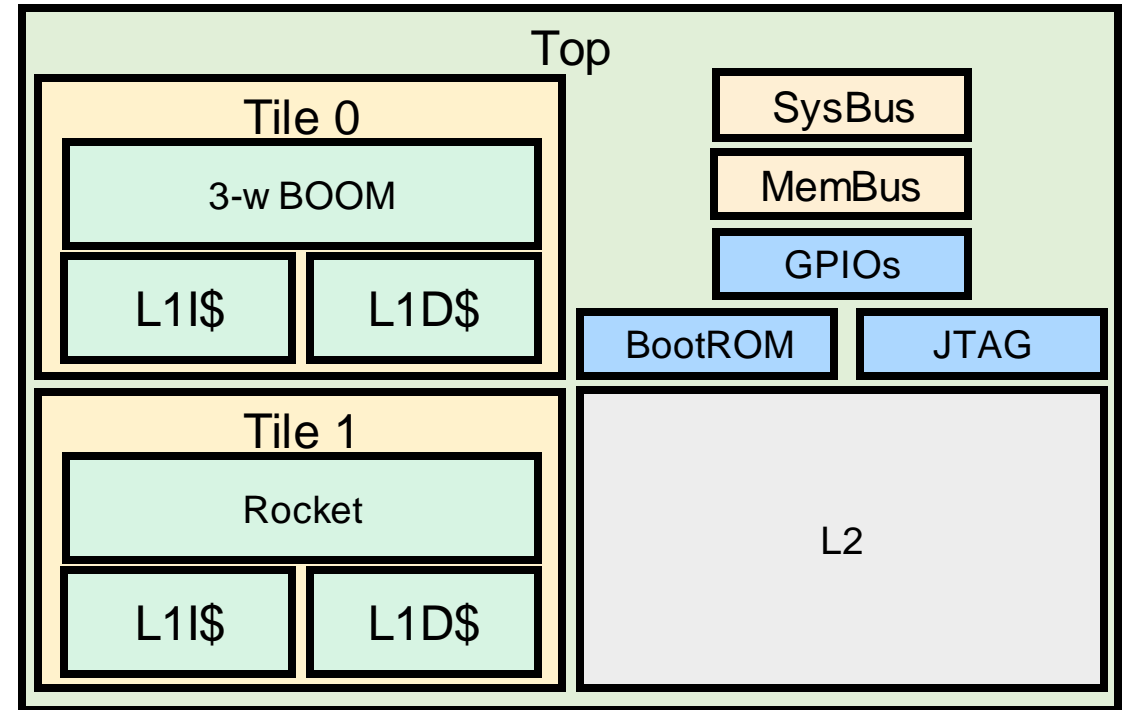
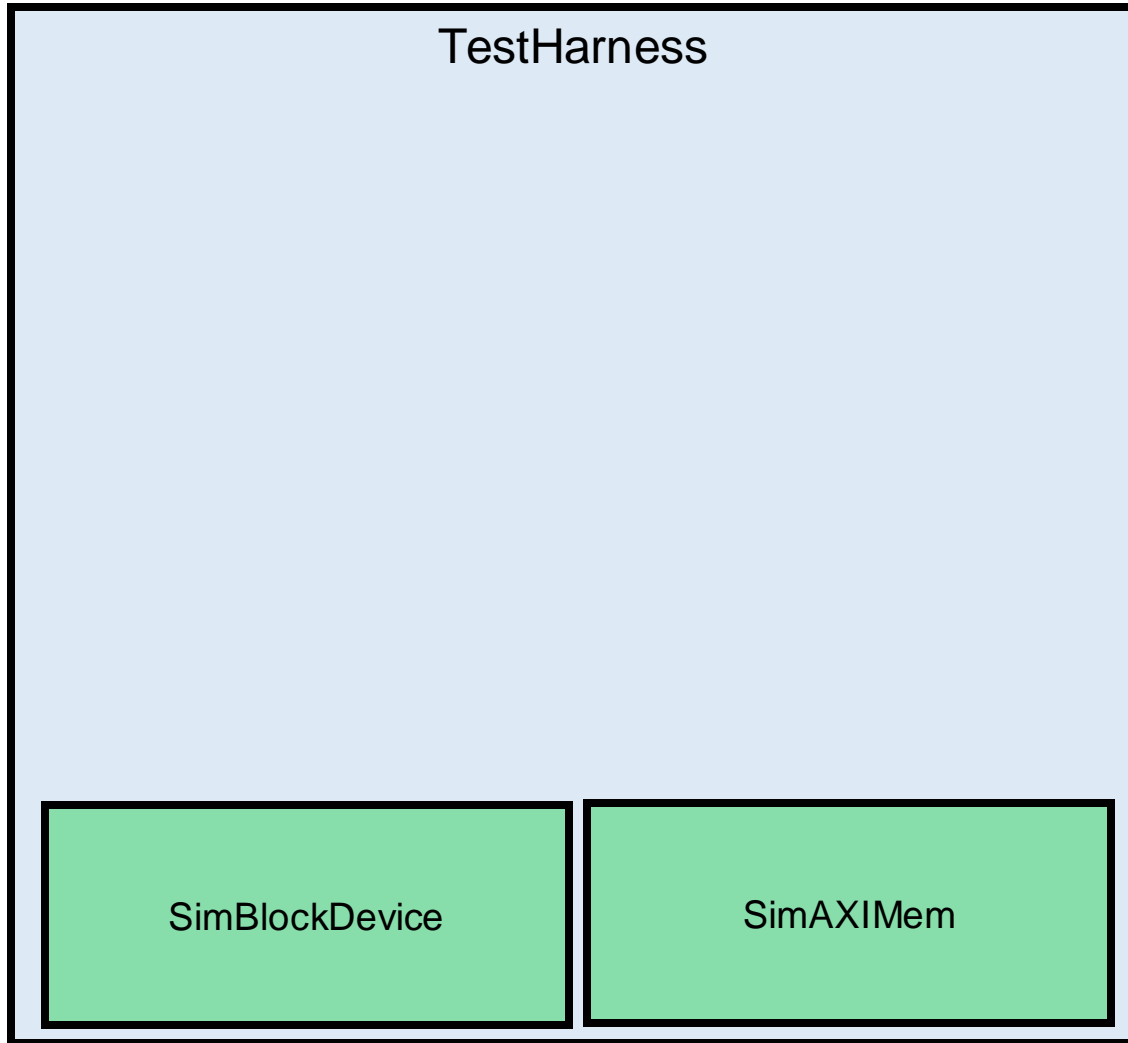
1. Configs parameterize Chisel generators
2. Chisel elaborates into FIRRTL
3. FIRRTL elaborates into Verilog



Why top.v and harness.v



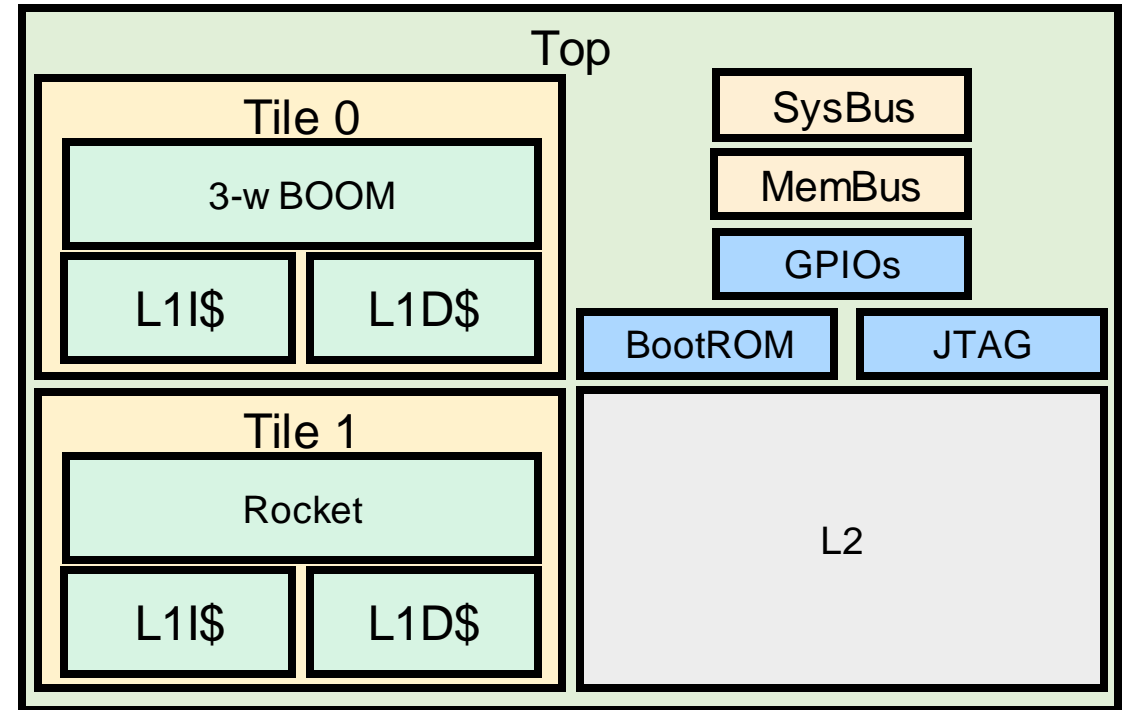
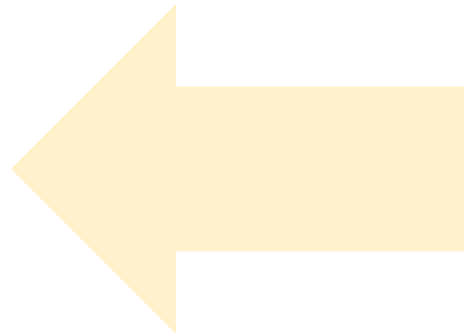
Why top.v and harness.v



Why top.v and harness.v



Passed to the VLSI flow



Building a SW Simulator

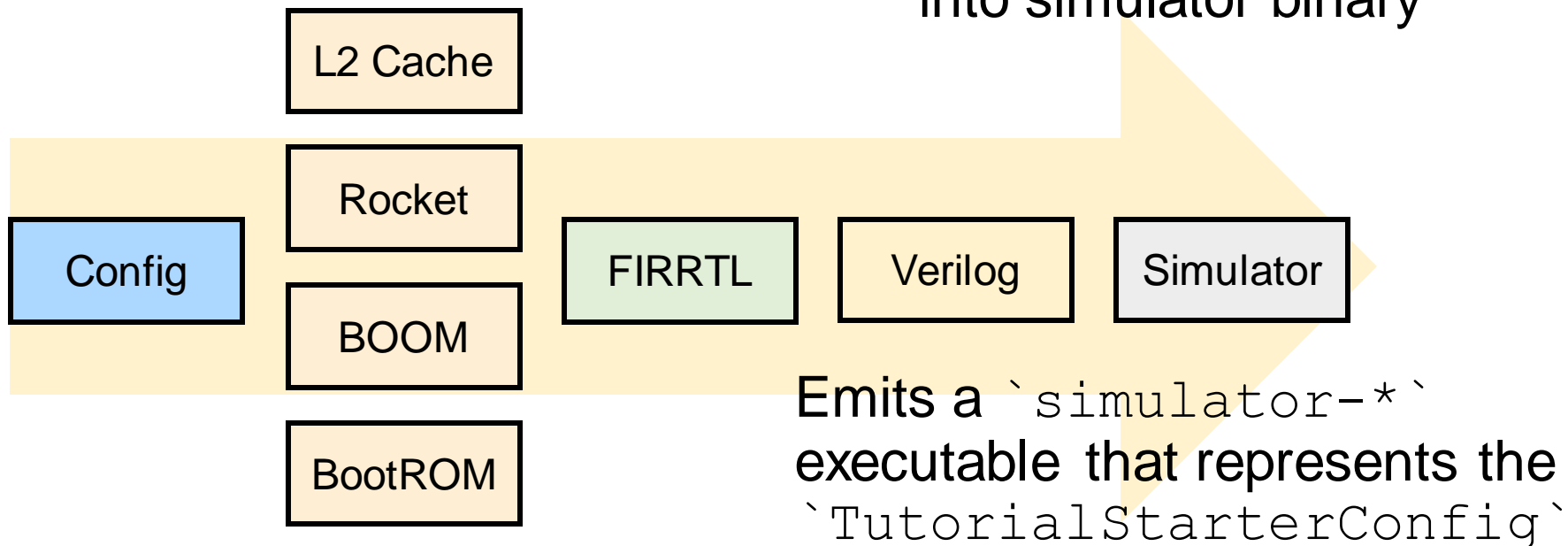


1. Configs parameterize Chisel generators

2. Chisel elaborates into FIRRTL

3. FIRRTL elaborates into Verilog

4. Verilator compiles Verilog into simulator binary



Where is my Verilog!



- Once completed the build outputs are located here

```
> ls generated-src/*/
```

```
generated-src/*/
```

```
*.harness.v
```

Verilog used for testing

```
*.top.v
```

Main Verilog of the design

```
*.dts
```

Device Tree used for SW

```
*.json
```

MMIO files (where and what is connected)

```
...
```

```
chipyard-morning/  
  sims/  
    verilator/  
      generated-src/*/
```



Open the Verilog



```
# navigate to the output sources
> cd generated-src/*/
> ls

# take a look at the following files
> vim chipyard.TestHarness.*.top.v
> vim chipyard.TestHarness.*.harness.v
```

```
chipyard-morning/
  sims/
    verilator/
      generated-src/*/
```



How does the Verilog look?



chipyard.TestHarness.TutorialStarterConfig.top.v

```
// Rocket Tile (includes a Rocket Core + L1's)
module RocketTile(
    input    clock,
    input    reset,
    . . .
// BOOM Tile (includes a BOOM Core + L1's)
module BoomTile(
    input    clock,
    input    reset,
    . . .
// Top-level DUT that includes the BOOM and Rocket Tiles
module ChipTop(
    input    clock,
    input    reset,
```

```
chipyard-morning/
  sims/
    verilator/
      generated-src/*/
```



Run a Binary Test



```
# navigate to the output sources
> cd ~/chipyard-morning/sims/verilator
> ls

# run basic RISC-V assembly tests
> export BINARY=$RISCV/riscv64-unknown-elf/share/tests/isa/rv64ui-p-simple
> make CONFIG=TutorialStarterConfig run-binary

# take a look at an output file
> vim output/chipyard.TestHarness.TutorialStarterConfig/rv64ui-p-simple.out
```

```
chipyard-morning/
sims/
  verilator/
```



Output file



```
using random seed 1570590524
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 38117
C1:      0 [0] pc=[0000004c0c7cadf1] W[r 0=0000000000000080][0] R[r18=eb4b96208b54f69a] R[r28=eb4b96208b54f69a] inst=[000 0c34d] c.beqz a4, pc + 162
C1:      1 [0] pc=[0000004c0c7cadf1] W[r 0=0000000000000080][0] R[r18=eb4b96208b54f69a] R[r28=eb4b96208b54f69a] inst=[000 0c34d] c.beqz a4, pc + 162
C1:      2 [0] pc=[0000004c0c7cadf1] W[r 0=0000000000000080][0] R[r18=eb4b96208b54f69a] R[r28=eb4b96208b54f69a] inst=[000 0c34d] c.beqz a4, pc + 162
C1:      3 [0] pc=[0000004c0c7cadf1] W[r 0=0000000000000080][0] R[r18=eb4b96208b54f69a] R[r28=eb4b96208b54f69a] inst=[000 0c34d] c.beqz a4, pc + 162
C1:      4 [0] pc=[0000004c0c7cadf1] W[r 0=0000000000000080][0] R[r18=eb4b96208b54f69a] R[r28=eb4b96208b54f69a] inst=[000 0c34d] c.beqz a4, pc + 162
C1:      5 [0] pc=[0000004c0c7cadf1] W[r 0=0000000000000080][0] R[r18=eb4b96208b54f69a] R[r28=eb4b96208b54f69a] inst=[000 0c34d] c.beqz a4, pc + 162
. . .
C1:    63210 [1] pc=[00000008000010e] W[r 0=0000000000000000][0] R[r10=0000000000000001] R[r11=0000000000000001] inst=[00b 57063] bgeu a0, a1, pc + 0
C1:    63211 [0] pc=[00000008000010e] W[r 0=0000000000000000][0] R[r10=0000000000000001] R[r11=0000000000000001] inst=[00b 57063] bgeu a0, a1, pc + 0
*** PASSED *** Completed after 107267 cycles
```

Test passed

Commit log lines emitted from the Rocket Core





Case Study: Configure and Test SHA3-accelerated SoC

SHA3 accelerator?



- SHA3 accelerator is a pre-implemented Chisel accelerator
- Implements the Secure Hash Algorithm 3 (SHA3)
 - Rough specification in 2012, released in late 2015
 - Uses variable length messages with a sponge function
- Costly when running on general purpose CPU
 - Want to improve the hashes/sec and hashes/Watt
- Drastically reduced runtimes with the accelerator

Integrating the SHA3 Accelerator



```
# clone SHA3-accelerator
> cd ~/chipyard-morning/generators
> git clone https://github.com/ucb-bar/sha3.git
> cd sha3
> git checkout tutorial
```

```
chipyard-morning/
  generators/
    chipyard/
      sha3/
      build.sbt
```



Integrating the SHA3 Accelerator



```
# Add sha3 to build.sbt
> cd ~/chipyard-morning
> vim build.sbt
```

```
chipyard-morning/
  generators/
    chipyard/
      sha3/
      build.sbt
```

```
lazy val chipyard = (project in file("generators/chipyard"))
  .dependsOn(testchipip, rocketchip, boom, hwacha, sifive_blocks, sifive_cache, iocell,
  sha3, // On separate line to allow for cleaner tutorial-setup patches
  dsptools, `rocket-dsp-utils`,
  gemmini, icenet, tracegen, cva6, nvdla, sodor, ibex, fft_generator)
```



Integrating the SHA3 Accelerator



```
# Add sha3 to build.sbt
> cd ~/chipyard-morning
> vim build.sbt
```

```
chipyard-morning/
  generators/
    chipyard/
      sha3/
      build.sbt
```

```
lazy val sha3 = (project in file("generators/sha3"))
  .dependsOn(rocketchip, midasTargetUtils)
  .settings(libraryDependencies += rocketLibDeps.value)
  .settings(chiselTestSettings)
  .settings(commonSettings)
```



Understanding the SHA3 Accelerator



- SHA3 is a minimal example of a RoCC-based accelerator
 - Executes custom “sha3” instructions sent by the Rocket or BOOM core
- sha3.scala
 - Note the ``WithSha3Accel`` mixin, which plugs into the Rocket Chip config system
 - ``Sha3AccelImp`` implements the Chisel-based accelerator

```
chipyard-morning/  
  generators/  
    sha3/  
      src/main/scala/  
        sha3.scala
```



The SHA3 Accelerator



- The SHA3 mixin

```
class WithSha3Accel extends Config ((site, here, up) => {  
  case Sha3WidthP => 64  
  case Sha3Stages => 1  
  case Sha3FastMem => true  
  case Sha3BufferSram => false  
  case BuildRoCC => Seq(  
    (p: Parameters) => {  
      val sha3 = LazyModule.apply(  
        new Sha3Accel(OpcodeSet.custom2)(p)  
      )  
      sha3  
    }  
  )  
})
```

SHA3 Parameters

Rocket Chip uses the “BuildRoCC” key to figure out which accelerator to build

```
chipyard-morning/  
  generators/  
    sha3/  
      src/main/scala/  
        sha3.scala
```



Example



```
# open rocket configs file
> cd generators/chipyard/src/main/scala/config
> vim TutorialConfigs.scala
```

```
class TutorialSha3Config extends Config(
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

Uncomment

```
class TutorialSha3Config extends Config(
  new sha3.WithSha3Accel ++
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

```
chipyard-morning/
  generators/
    chipyard/
      src/main/scala/config
        TutorialConfigs.scala
```



Running customized software on the SHA3 Accelerated SoC



Example



```
# navigate to firemarshal
> cd ~/chipyard-morning/generators/sha3/software/

# view SHA3 accelerated program
> vim tests/src/sha3-rocc.c
```

```
chipyard-morning/
  generators/
    sha3/
      software/
        tests/
          src/
            sha3-rocc.c
```



Accelerator Software



sha3-rocc.c

```
printf("Start basic test 1.\n");  
// BASIC TEST 1 - 150 zero bytes  
  
start = rdcycle();  
  
asm volatile ("fence");  
  
// setup accelerator with addresses of input and output  
ROCC_INSTRUCTION_SS(2, &input, &output, 0);  
  
// Set length and compute hash  
ROCC_INSTRUCTION_S(2, sizeof(input), 1);  
  
asm volatile ("fence" ::: "memory");  
  
end = rdcycle();
```

```
chipyard-morning/  
  generators/  
    sha3/  
      software/  
        tests/  
          src/  
            sha3-rocc.c
```



Accelerator Software



sha3-rocc.c

```
printf("Start basic test 1.\n");  
// BASIC TEST 1 - 150 zero bytes
```

```
start = rdcycle();
```

```
asm volatile ("fence");
```

```
// setup accelerator with addresses of input and output  
ROCC_INSTRUCTION_SS(2, &input, &output, 0);
```

```
// Set length and compute hash  
ROCC_INSTRUCTION_S(2, sizeof(input), 1);
```

```
asm volatile ("fence" ::: "memory");
```

```
end = rdcycle();
```

Expanded C macro

```
asm volatile ("opcode rd rs1 rs2 funct  
:: [msg_addr] "r" (&input), [hash_addr] "r" (&output));
```



Accelerator Software



sha3-rocc.c

```
printf("Start basic test 1.\n");  
// BASIC TEST 1 - 150 zero bytes
```

```
start = rdcycle();
```

```
asm volatile ("fence");
```

```
// setup accelerator with addresses of input and output  
ROCC_INSTRUCTION_SS(2, &input, &output, 0);
```

```
// Set length and compute hash  
ROCC_INSTRUCTION_S(2, sizeof(input), 1);
```

```
asm volatile ("fence" ::: "memory");
```

```
end = rdcycle();
```

Expanded C macro

```
asm volatile ("custom2 x0, %[msg_addr], %[hash_addr], 0"  
             :: [msg_addr] "r" (&input), [hash_addr] "r" (&output));
```

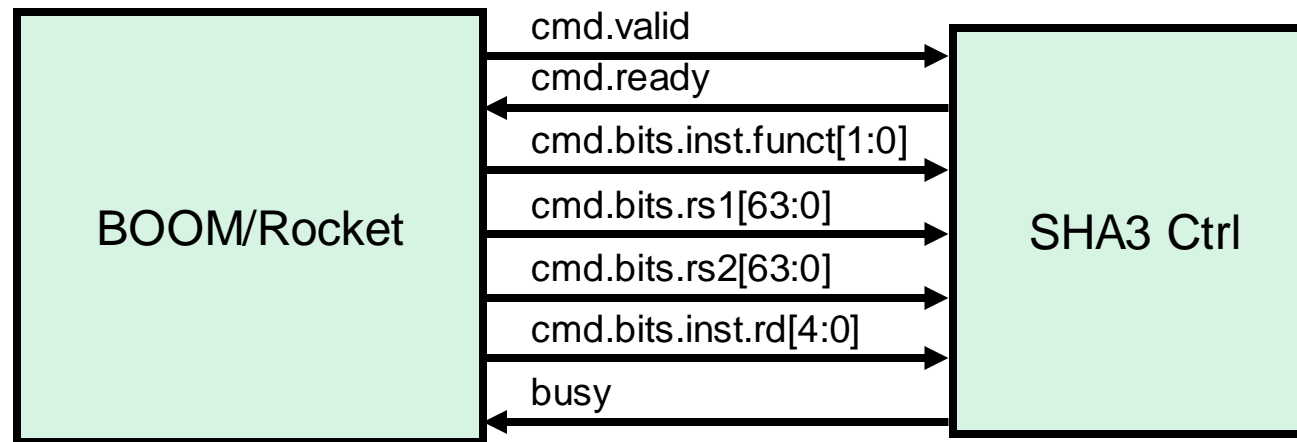
Setup and run the
accelerator



Understanding SHA3 Software



- Send necessary information to the accelerator
 - Source(s), Destination, and what function to run
- Accelerator accesses memory, performs SHA3



```
asm volatile ("custom2 x0, %[msg_addr], %[hash_addr], 0"
              :: [msg_addr] "r" (&input), [hash_addr] "r" (&output));
```



Example



```
# view SW implementation of SHA3 program  
> vim tests/src/sha3-sw.c
```

```
chipyard-morning/  
  generators/  
    sha3/  
      software/
```



Software SHA3



sha3-sw.c

```
printf("Start basic test 1.\n");  
// BASIC TEST 1 - 150 zero bytes  
  
start = rdcycle();  
  
// run sw to compute the SHA3 hash  
sha3ONE(input, sizeof(input), output);  
  
end = rdcycle();
```

```
chipyard-morning/  
  generators/  
    sha3/  
      software/  
        tests/  
          src/  
            sha3-sw.c
```



Example



```
# build both binaries
> marshal build marshal-configs/sha3-bare-rocc.yaml
> marshal build marshal-configs/sha3-bare-sw.yaml
```

```
chipyard-morning/
  generators/
    sha3/
      software/
        marshal-configs/
```



Building SHA3 Software



- Used the FireMarshal utility to build the binaries
 - Tool that takes in ``.yaml`` description of build and emits the ``.riscv`` binary
 - More in-depth view after lunch
- What was done... built two binaries
 - ``.sha3-sw.riscv`` - software version of SHA3 computation
 - ``.sha3-rocc.riscv`` - sends SHA3 computation to the accelerator
- Both binaries created in
 - ``.sha3/software/tests/bare/sha3-*.riscv``



Example



```
# navigate to the Verilator directory
> cd ~/chipyard-morning/sims/verilator

# run accelerated program
> make CONFIG=TutorialSha3Config run-binary
BINARY=../../generators/sha3/software/tests/bare/sha3-
rocc.riscv

# run non-accelerated program
> make CONFIG=TutorialSha3Config run-binary
BINARY=../../generators/sha3/software/tests/bare/sha
3-sw.riscv
```

```
chipyard-morning/
sims/
  verilator/
```

The non-accelerated version takes longer



Try Other Configs



- Lots of other demonstration configs available to try
 - Heterogeneous configs
 - Customized memory systems
 - Customized cores
 - Different accelerators
- `make CONFIG=<chosen_config> run-asm-tests -j4``

Questions?

```
chipyard-morning/  
  generators/  
    chipyard/  
      src/main/scala/config  
        RocketConfigs.scala  
        BoomConfigs.scala  
        HeteroConfigs.scala
```



More Information



- chipyard.readthedocs.io
 - Talks about heterogeneous SoCs
 - Talks about adding Verilog IP
 - Talks about adding accelerators
 - . . .

Covers what we did in
this talk and more!

Docs » Welcome to Chipyard's documentation! [Edit on GitHub](#)

Welcome to Chipyard's documentation!



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 intergration 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.

Quick Start

Requirements

Chipyard is developed and tested on Linux-based systems.

Warning

- 1. Chipyard Basics
- 2. Simulation
- 3. Generators
- 4. Tools
- 5. VLSI Flow
- 6. Customization
- 7. Target Software
- 8. Advanced Concepts
- 9. TileLink and Diplomacy Reference

