REBAR Documentation

Release 0.1

Berkeley Architecture Research

Jun 24, 2019

Contents:

REBAR is a 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 REBAR? Jump to the *Getting Started* page for more info.

CHAPTER 1

Getting Started

These guides will walk you through the basics of the REBAR framework:

- First, we will go over the different configurations available.
- Then, we will walk through adding a custom accelerator.

Hit next to get started!

1.1 REBAR Basics

1.1.1 Generators

The REBAR Framework currently consists of the following RTL generators:

Processor Cores

Rocket An in-order RISC-V core. See *Rocket* 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.

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.

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.

1.1.2 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.

1.1.3 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 REBAR 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.)

1.1.4 Sims

verisim (Verilator wrapper) Verilator is an open source Verilog simulator. The `verisim` 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* for more information.

vsim (VCS wrapper) VCS is a proprietary Verilog simulator. Assuming the user has valid VCS licenses and installations, the `vsim` 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 *VCS* 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.

1.1.5 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 *HAMMER* for more information.

1.2 Configs, Parameters, Mix-ins, and Everything In Between

A significant portion of generators in the REBAR 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:

1.2.1 Parameters

TODO: Need to explain up, site, field, and other stuff from Henry's thesis.

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.

1.2.2 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. The naming convention for an additive Config 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*).

`basic-config-example` shows a basic additive Config class that takes in zero arguments and instead uses hard-coded 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 (`complex-config-example`) shows a "higher-level" additive Config 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)
})
```

`top-level-config` shows a non-additive Config that combines the prior two additive Configs using `++`. The additive Configs 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
)
```

1.2.3 Cake Pattern

A cake pattern 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 REBAR framework in merging multiple system components and IO interfaces into a large system component.

`cake-example` shows a Rocket Chip based SoC that merges multiple system components (BootROM, UART, etc) into a single top-level design.

```
class MySoC(implicit p: Parameters) extends RocketSubsystem
  with CanHaveMisalignedMasterAXI4MemPort
  with HasPeripheryBootROM
  with HasNoDebug
  with HasPeripherySerial
  with HasPeripheryUART
  with HasPeripheryIceNIC
{
   //Additional top-level specific instantiations or wiring
}
```

1.2.4 Mix-in

A mix-in is a Scala trait, which sets parameters for specific system components, as well as enabling instantiation and wiring of the relevant system components to system buses. The naming convention for an additive mix-in is `Has<YourMixin>`. This is show in `cake-example` where things such as `HasPeripherySerial` connect a RTL component to a bus and expose signals to the top-level.

1.2.5 Additional References

A brief explanation of some of these topics is given in the following video: https://www.youtube.com/watch?v=Eko86PGEoDY.

1.3 Adding An Accelerator/Device

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.

1.3.1 Integrating into the Generator Build System

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 REBAR 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`.

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

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

```
lazy val yourproject = project.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 `example` project, change the final line in build.sbt to the following.

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

Finally, add `yourproject` to the `PACKAGES` variable in the `common.mk` file in the REBAR top level. This will allow make to detect that your source files have changed when building the Verilog/FIRRTL files.

1.3.2 MMIO Peripheral

The easiest way to create a TileLink peripheral is to use the `TLRegisterRouter`, which abstracts away the details of handling the TileLink protocol and provides a convenient interface for specifying memory-mapped registers. 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.

```
case class PWMParams(address: BigInt, beatBytes: Int)

trait PWMTLBundle extends Bundle {
  val pwmout = Output(Bool())
}

trait PWMTLModule {
  val io: PWMTLBundle
  implicit val p: Parameters
  def params: PWMParams

  val w = params.beatBytes * 8
  val period = Reg(UInt(w.W))
  val duty = Reg(UInt(w.W))
  val enable = RegInit(false.B)

  // ... Use the registers to drive io.pwmout ...

  regmap(
    0x00 -> Seq(
      RegField(w, period)),
    0x04 -> Seq(
      RegField(w, duty)),
    0x08 -> Seq(
      RegField(1, enable)))
}
```

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.

```
class PWMTL(c: PWMParams)(implicit p: Parameters)
  extends TLRegisterRouter(
    c.address, "pwm", Seq("ucbbar,pwm"),
    beatBytes = c.beatBytes)(
      new TLRegBundle(c, _) with PWMTLBundle)(
      new TLRegModule(c, _, _) with PWMTLModule)
```

The full module code can be found in `generators/example/src/main/scala/PWM.scala`.

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

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

```
trait HasPeripheryPWM extends HasSystemNetworks {
  implicit val p: Parameters
```

(continues on next page)

(continued from previous page)

```
  private val address = 0x2000

  val pwm = LazyModule(new PWMTL(
    PWMParams(address, peripheryBusConfig.beatBytes))(p))

  pwm.node := TLFragmenter(
    peripheryBusConfig.beatBytes, cacheBlockBytes)(peripheryBus.node)
}
```

Note that the `PWMTL` 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.

The module implementation trait is where we instantiate our PWM module and connect it to the rest of the SoC. Since this module has an extra *pwmout* output, we declare that in this trait, using Chisel's multi-IO functionality. We then connect the `PWMTL`'s pwmout to the pwmout we declared.

```
trait HasPeripheryPWMModuleImp extends LazyMultiIOModuleImp {
  implicit val p: Parameters
  val outer: HasPeripheryPWM

  val pwmout = IO(Output(Bool()))

  pwmout := outer.pwm.module.io.pwmout
}
```

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

```
class ExampleTopWithPWM(q: Parameters) extends ExampleTop(q)
    with PeripheryPWM {
  override lazy val module = Module(
    new ExampleTopWithPWMModule(p, this))
}

class ExampleTopWithPWMModule(l: ExampleTopWithPWM)
  extends ExampleTopModule(l) with HasPeripheryPWMModuleImp
```

Just as we need separate traits for `LazyModule` and module implementation, we need two classes to build the system. The `ExampleTop` classes already have the basic peripherals included for us, so we will just extend those.

The `ExampleTop` class includes the pre-elaboration code and also a `lazy val` to produce the module implementation (hence `LazyModule`). The `ExampleTopModule` class is the actual RTL that gets synthesized.

Finally, we need to add a configuration class in `generators/example/src/main/scala/Configs.scala` that tells the `TestHarness` to instantiate `ExampleTopWithPWM` instead of the default `ExampleTop`.

```
class WithPWM extends Config((site, here, up) => {
  case BuildTop => (p: Parameters) =>
    Module(LazyModule(new ExampleTopWithPWM()(p)).module)
})

class PWMConfig extends Config(new WithPWM ++ new BaseExampleConfig)
```

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

```
#define PWM_PERIOD 0x2000
#define PWM_DUTY 0x2008
#define PWM_ENABLE 0x2010

static inline void write_reg(unsigned long addr, unsigned long data)
{
        volatile unsigned long *ptr = (volatile unsigned long *) addr;
        *ptr = data;
}

static inline unsigned long read_reg(unsigned long addr)
{
        volatile unsigned long *ptr = (volatile unsigned long *) addr;
        return *ptr;
}

int main(void)
{
        write_reg(PWM_PERIOD, 20);
        write_reg(PWM_DUTY, 5);
        write_reg(PWM_ENABLE, 1);
}
```

This just writes out to the registers we defined earlier. The base of the module's MMIO region is at 0x2000. This will be printed out in the address map portion when you generated the verilog code.

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

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

```
cd verisim
make CONFIG=PWMConfig
./simulator-example-PWMConfig ../tests/pwm.riscv
```

1.3.3 Adding a RoCC Accelerator

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

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

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

(continues on next page)

(continued from previous page)

```
  //   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 DefaultExampleConfig)
```

1.3.4 Adding a DMA port

IO devices or accelerators (like a disk or network driver), we may want to have the device write directly to the coherent memory system instead. To add a device like that, you would do the following.

```
class DMADevice(implicit p: Parameters) extends LazyModule {
  val node = TLClientNode(TLClientParameters(
    name = "dma-device", sourceId = IdRange(0, 1)))

  lazy val module = new DMADeviceModule(this)
}

class DMADeviceModule(outer: DMADevice) extends LazyModuleImp(outer) {
  val io = IO(new Bundle {
    val mem = outer.node.bundleOut
    val ext = new ExtBundle
  })
```

(continues on next page)

(continued from previous page)

```
  // ... rest of the code ...
}

trait HasPeripheryDMA extends HasSystemNetworks {
  implicit val p: Parameters

  val dma = LazyModule(new DMADevice)

  fsb.node := dma.node
}

trait HasPeripheryDMAModuleImp extends LazyMultiIOModuleImp {
  val ext = IO(new ExtBundle)
  ext <> outer.dma.module.io.ext
}
```

The `ExtBundle` contains the signals we connect off-chip that we get data from. The DMADevice also has a Tilelink client port that we connect into the L1-L2 crossbar through the front-side buffer (fsb). The sourceId variable given in the `TLClientNode` instantiation determines the range of ids that can be used in acquire messages from this device. Since we specified [0, 1) as our range, only the ID 0 can be used.

1.4 Initial Repository Setup

1.4.1 Checking out the sources

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

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

1.4.2 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`). There are detailed instructions at https://github.com/riscv/riscv-tools to install the `riscv-tools` toolchain, however, the instructions are similar for the Hwacha `esp-tools` toolchain. But to get a basic installation, just the following steps are necessary.

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

# OR

./scripts/build-toolchains.sh hwacha # for a hwacha modified risc-v toolchain
```

Once the script is run, a `env.sh` file is emitted at 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.

1.5 Running A Simulation

REBAR provides support and integration for multiple simulation flows, for various user levels and requirements. In the majority of cases during a digital design development process, simple software RTL simulation is needed. When more advanced full-system evaluation is required, with long running workloads, FPGA-accelerated simulation will then become a preferable solution.

1.5.1 Software RTL Simulation

The REBAR framework provides wrappers for two common software RTL simulators: the open-source Verilator simulator and the proprietary VCS simulator. For more information on either of these simulators, please refer to *Verilator* or *VCS*. The following instructions assume at least one of these simulators is installed.

Verilator/VCS Flows

Verilator is an open-source RTL simulator. We run Verilator simulations from within the `sims/verisim` directory which provides the necessary `Makefile` to both install and run Verilator simulations. On the other hand, VCS is a proprietary RTL simulator. We run VCS simulations from within the `sims/vsim` directory. Assuming VCS is already installed on the machine running simulations (and is found on our `PATH`), then this guide is the same for both Verilator and VCS.

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

```
# Enter Verilator directory
cd sims/verisim

# OR

# Enter VCS directory
cd sims/vsim
```

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=example
```

Alternatively, if we would like to simulate a simple BOOM-based example system we can use:

```
make SUB_PROJECT=exampleboom
```

Once the simulator has been constructed, we would like to run RISC-V programs on it. In the simulation directory, we will find an executable file called `<...>-<package>-<config>`. We run this executable with our target RISC-V program as a command line argument in one of two ways. One, we can directly call the simulator binary or use make to run the binary for us with extra simulation flags. For example:

```
# directly calling the simulation binary
./<...>-<package>-<config> my_program_binary

# using make to do it
make SUB_PROJECT=example BINARY=my_program_binary run-binary
```

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 SUB_PROJECT=example run-asm-tests
make SUB_PROJECT=example run-bmark-tests
```

Note: You need to specify all the make variables once again to match what the build gave to run the assembly tests or the benchmarks or the binaries if you are using the make option.

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`.

1.5.2 FPGA Accelerated Simulation

FireSim enables simulations at 1000x-100000x the speed of standard software simulation. This is enabled using FPGA-acceleration on F1 instances of the AWS (Amazon Web Services) public cloud. Therefore FireSim simulation requires to be set-up on the AWS public cloud rather than on our local development machine.

To run an FPGA-accelerated simulation using FireSim, a we need to clone the REBAR repository (or our fork of the REBAR repository) to an AWS EC2, and follow the setup instructions specified in the FireSim Initial Setup documentation page.

After setting up the FireSim environment, we now need to generate a FireSim simulation around our selected digital design. We will work from within the `sims/firesim` directory.

TODO: Continue from here

1.6 SoC Generator Config Mix-ins:

1.6.1 Rocket Chip

- **System-on-Chip**
  - HasTiles
  - HasClockDomainCrossing

  - HasResetVectorWire
  - HasNoiseMakerIO
- **Basic Core**
  - HasRocketTiles
  - HasRocketCoreParameters
  - HasCoreIO
- **Branch Prediction**
  - HasBtbParameters
- **Additional Compute**
  - HasFPUCtrlSigs
  - HasFPUParameters
  - HasLazyRoCC
  - HasFpuOpt
- **Memory System**
  - HasRegMap
  - HasCoreMemOp
  - HasHellaCache
  - HasL1ICacheParameters
  - HasICacheFrontendModule
  - HasAXI4ControlRegMap
  - HasTLControlRegMap
  - HasTLBusParams
  - HasTLXbarPhy
- **Interrupts**
  - HasInterruptSources
  - HasExtInterrupts
  - HasAsyncExtInterrupts
  - HasSyncExtInterrupts
- **Periphery**
  - HasPeripheryDebug
  - HasPeripheryBootROM
  - HasBuiltInDeviceParams

1.6.2 BOOM

- **Basic Core**
  - HasBoomTiles
  - HasBoomCoreParameters
  - HasBoomCoreIO
  - HasBoomUOP
  - HasRegisterFileIO
- **Branch Prediction**
  - HasGShareParameters
  - HasBoomBTBParameters
- **Memory System**
  - HasL1ICacheBankedParameters
  - HasBoomICacheFrontend
  - HasBoomHellaCache

1.6.3 SiFive Blocks

- **Peripherals**
  - HasPeripheryGPIO
  - HasPeripheryI2C
  - HasPeripheryMockAON
  - HasPeripheryPWM
  - HasPeripherySPI
  - **HasSPIProtocol**
    - HasSPIEndian
    - HasSPILength
    - HasSPICSMode
  - HasPeripherySPIFlash
  - HasPeripheryUART

1.6.4 testchipip

- **Peripherals**
  - HasPeripheryBlockDevice
  - HasPeripherySerial
  - HasNoDebug

1.6.5 Icenet

- **Periphery Network Interface Controller**
  - HasPeripheryIceNIC

1.6.6 AWL

- **IO**
  - HasEncoding8b10b
  - HasTLBidirectionalPacketizer
  - HasTLController
  - HasGenericTransceiverSubsystem
- **Debug/Testing**
  - HasBertDebug
  - HasPatternMemDebug
  - HasBitStufferDebug4Modes
  - HasBitReversalDebug

CHAPTER 2

Simulators

REBAR provides support and integration for multiple simulation flows, for various user levels and requirements. In the majority of cases during a digital design development process, a simple software RTL simulation will do. When more advanced full-system evaluation is required, with long running workloads, FPGA-accelerated simulation will then become a preferable solution. The following pages provide detailed information about the simulation possibilities within the REBAR framework.

2.1 Open Source Software RTL Simulators

2.1.1 Verilator

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

To run a simulation using Verilator, perform the following steps:

To compile the example design, run `make` in the `sims/verisim` directory. This will elaborate the `DefaultRocketConfig` in the example project.

An executable called `simulator-example-DefaultRocketConfig` 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-example-DefaultRocketConfig $RISCV/riscv64-unknown-elf/share/riscv-tests/
↪isa/rv64ui-p-simple
```

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

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

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

Please refer to *Running A Simulation* for a step by step tutorial on how to get a simulator up and running.

2.2 Commercial Software RTL Simulators

2.2.1 VCS

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

To run a simulation using VCS, perform the following steps:

Make sure that the VCS simulator is on your `PATH`.

To compile the example design, run make in the `sims/vsim` directory. This will elaborate the `DefaultRocketConfig` in the example project.

An executable called `simulator-example-DefaultRocketConfig` 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-example-DefaultRocketConfig $RISCV/riscv64-unknown-elf/share/riscv-tests/
↪isa/rv64ui-p-simple
```

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

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

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

Please refer to *Running A Simulation* for a step by step tutorial on how to get a simulator up and running.

2.3 FPGA-Based Simulators

2.3.1 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 on the public cloud. In order to simulate your REBAR design using FireSim, you should follow the following steps:

Follow the initial EC2 setup instructions as detailed in the FireSim documentation. Then clone your full REBAR repository onto your Amazon EC2 FireSim manager instance.

Enter the `sims/FireSim` directory, and follow the FireSim instructions for running a simulation.

CHAPTER 3

Generators

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 REBAR framework.

3.1 Rocket

Rocket is a 5-stage in-order scalar core generator that is supported by SiFive. It 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.

3.2 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. 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.

3.3 Hwacha

The Hwacha project is developing a new vector architecture for future computer systems that are constrained in their power and energy consumption. Inspired by traditional vector machines from the 70s and 80s, and lessons learned from our previous vector-thread architectures Scale and Maven, we are bringing back elegant, performant, and energy-efficient aspects of vector processing to modern data-parallel architectures. We propose a new vector-fetch architectural paradigm, which focuses on the following aspects for higher performance, better energy efficiency, and lower complexity.

For more information, please visit the Hwacha website.

CHAPTER 4

Tools

The REBAR 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.

4.1 Chisel

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

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

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

4.2 FIRRTL

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

For more information on please visit their website.

4.3 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.

CHAPTER 5

VLSI Production

The REBAR framework aim to provide wrappers to a general VLSI flow. In particular, we aim to support the HAMMER flow.

5.1 HAMMER

HAMMER is a physical design generator that wraps around vendor specific technologies and tools to provide a single API to create ASICs. 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.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search