RISC-V “Rocket Chip” Tutorial

Colin Schmidt
UC Berkeley

colins@eecs.berkeley.edu
What can Rocket Chip do?

How do I change what Rocket Chip generates?
  – What are chisel parameters and how do they help me?

How do I use the C++ emulator?

How do I get a waveform/debug?

How do I add different options?
  – Where do I put my changes?

How do I add new instructions?
  – How do I “drop-in” my accelerator?
  – Where do I put different extensions?

How do I use verilog generation?
  – For an ASIC toolflow
  – For an FPGA target
What can Rocket Chip do?

- Rocket chip allows you to generate different configurations of an SoC, including the software toolchain that would run on this software.
- These configurations are specified through chisel parameters most of which can be freely changed.
- We can then select what to generate:
  - C++ RTL emulator
  - Verilog
    - FPGA
    - ASIC
What are all these submodules in Rocket Chip?

- **Chisel**
  - The HDL we use at Berkeley to develop our RTL.
- **Rocket**
  - Source code for the Rocket core and caches
- **Uncore**
  - Logic outside the core: coherence agent, tile interface, host interface
- **Hardfloat**
  - Parameterized FMAs and converters, see README
- **Dramsim2**
  - Simulates DRAM timing for simulations
- **Fpga-zync**
  - Code that helps get rocket-chip on FPGAs
- **Riscv-tools**
  - Software toolchain used with this version of Rocket Chip
What about the other folders?

- **CSrc**
  - Glue code to be used with the C++ emulator
- **Emulator**
  - Build directory for the C++ emulator, contains generated code and executables
- **Fsim**
  - Build directory for FPGA verilog generation
- **Project**
  - Scala/sbt configuration files
- **Src**
  - Chisel source code for rocket chip
- **Vsrc**
  - Verilog test harness for rocket-chip
- **Vsim**
  - Build directory for ASIC verilog generation
- **Rocc-template** (example rocc used for this tutorial)
Overview of Rocket Chip Parameters

- Located in `src/main/scala/PublicConfigs.scala`
- Easily changed parameters are called Knobs
- Important configuration options fit in a few categories
  - Tile – How many, what types, what accel?
  - Memory – Phys/Virt Address bits, Mem interface params
  - Caches – Sets, ways, width etc. for L1 and L2; TLBs
  - Core – FPU?, fma latency, etc.
  - Uncore – coherence protocol, tilelink params
Parameters can be changed to create different configurations

Knobs require defaults and are parameters we expect to be tunable via Design space exploration

Two examples given at bottom of PublicConfigs.scala
  – DefaultConfig – used when no other configurations are specified
  – SmallConfig – removes FPU and has smaller caches

To generate a different configuration you can simply follow the SmallConfig Example, setting parameters and knobs as you want
Simulating a Configuration

- C++ RTL emulator built from emulator directory
- The default emulator has already been built

`make run-asm-tests`

- We can also build the small config very easily

`make CONFIG=ExampleSmallConfig`

- And test it too!

`make CONFIG=ExampleSmallConfig run-asm-tests`
Making and Simulating a new Configuration

- Lets try making a “medium” sized config
  - Double the number of ways in L1 I and D cache in small config

```scala
class MediumConfig extends SmallConfig{
  override val knobValues:Any=>Any = {
    case "L1D_WAYS" => 2
    case "L1I_WAYS" => 2
  }
}
```

```scala
class ExampleMediumConfig extends ChiselConfig(new MediumConfig ++ new DefaultConfig)
```

- All we need to do is specify it when making the emulator

```bash
make CONFIG=ExampleMediumConfig
```

- We can then test the new config

```bash
make CONFIG=ExampleMediumConfig run-asm-tests
```

- The power of generators!
More Complicated Configurations

- How would I add a new parameter to rocket chip?
  - Widely used parameters for the generator can be added to the DefaultConfig
  - It is then made available via Chisel parameters to the implementation

- How do I add accelerators? What about their parameters?
  - Other modules like accelerators should have their parameters declared in their own source folder
  - Default configuration can be added to a new *Configs.scala in the rocket-chip source
  - More on this later
Chapter 9 in the ISA manual

4 major opcodes set aside for non-standard extensions (Table 8.1)
  - Custom 0-3
  - Custom 2 and 3 are reserved for future RV128

RoCC interface uses this opcode space
  - 2 source operands, 1 destination, 7bit funct field
  - 3 bits(xd,xs1,x2) determine if this instruction uses the register operands, and passes the value in register rs1/2, or writes the response to rd

Depending on the accelerator these could always be repurposed since the instruction is also sent over the RoCC interface

RoCC Accelerators

- Implementing the RoCC interface is probably the simplest way to create a RISC-V extension
- Toolchain already supports custom0-3 assembly
  - No need to modify the toolchain at all if you fit into this interface
- Need to implement the RoCCIO interface
- Located in `rocket/src/main/scala/rocc.scala`
Rocket sends coprocessor instruction via the Cmd interface (including registers)

- Accelerator responds through Resp interface
- Accelerator sends memory requests to L1D$ via CacheIO
- busy bit for fences
- IRQ, S, exception bit used for virtualization
- UncachedTileLinkIO for instruction cache on accelerator
- PTWIO for page-table walker ports on accelerator
We can now start walking through an example accelerator used in teaching CS250 at Berkeley

(checkout sha3 branch of rocc-template)

This branch of rocc-template implements the SHA3 cryptographic hashing algorithm

It includes several things

- C reference code in rocc-template/src/main/c
- Chisel implementation in rocc-template/src/main/scala
- C test cases for both SW and RoCC in rocc-template/tests
- Functional model for Spike in rocc-template/isa-sim
- New Rocket chip configuration in rocc-template/config
- Script to install symlinks to all these files
First step to any architecture project write a simulator

Spike is designed to be extendable

vim rocc-template/isa-sim/sha3/sha3.h

We extend the rocc_t class implementing a subset of the custom opcodes

Describes a functional model of the computation

Adheres to the same interface as the accelerator

Interacting with the simulated memory happens through the processors mmu p->get_mmu()

Now we are ready to test the model
Functional Model of Accelerator

- Rather than moving the files out of the rocc-template directory we just symlink to them
  ./install-symlinks
- Now we can rebuild spike to be able to model our accelerator
  cd ../riscv-tools && ./build-spike-only.sh
- Now spike understands our extension
  spike --extension=sha3
Accelerator Tests

- A few variants of a simple sha3 test
  sha3-sw[−bm].c
  sha3-rocc[−bm].c
- sw versions just uses the reference C implementation
- rocc versions use inline assembly to call the accelerator

```bash
cd ../rocc-template/tests
vim sha3-rocc.c
```

- The operands are xd/rd, xs1/rs1, xs2/rs2, and funct
- Putting 0 for the register operands marks them unused
- Otherwise you can use standard assembly syntax to send values to the accelerator
Functional Model of Accelerator Testing

- Now we are ready to test our model
- First just the software only version
  ```
  spike pk sha3-sw.rv
  ```
- Lets try the accelerator version without the accel
  ```
  spike pk sha3-rocc.rv
  ```
- An expected failure so now we enable our model
  ```
  spike --extension=sha3 sha3-rocc.rv
  ```
- Success!
Chisel Accelerator

- Time to implement our design in chisel and plug it into Rocket chip
- Luckily the implementation is done and rocket chip is smart enough to pick up on folders that look like a chisel project (i.e. have a src/main/scala directory)
- We can look at how the accelerator is parameterized
  ```scala
  vim src/main/scala/sha3.scala
  ```
- Looking at the bottom we see it looks similar to previous configs we have looked at with the addition of a set of constraints
- The constraints help during any design space exploration you want to undertake
Chisel Accelerator Plug-in

- Now let's setup rocket chip to include our accelerator
  `vim config/PrivateConfigs.scala`
- The important parameter is the BuildRoCC parameter which gives the constructor for the Sha3 accelerator
- Rocket chip uses this parameter to instantiate the accelerator in its datapath
- The clean interface allows this to happen seamlessly
- Now we can build the accelerated version
  `make CONFIG=Sha3CPPPConfig`
Chisel Accelerator Performance

- Time to test this new emulator
- We can even measure performance (pk “s” flag)
  
  ./emulator-DefaultCPPConfig pk -s sha3-sw-bm.rv
  ./emulator-Sha3CPPConfig pk -s sha3-sw-bm.rv
  ./emulator-Sha3CPPConfig pk -s sha3-rocc-bm.rv

- Even on a very short test with a single hash we see a good speed up
What if I had a bug?

- Chisel has support for “printf” in your code but you might want to just see a waveform
- C++ emulator supports this too

```
make debug
./emulator-DefaultCPPConfig-debug -vtest.vcd +loadmem=output/median.riscv.hex
```

- This creates a standard vcd that a program like `gtkwave` can open
  `gtkwave test.vcd`
- This same setup works for the accelerator just takes longer because of the pk and test length
Non-RoCC extensions

- What if I want to extend the ISA in a different way, not RoCC
- This will be more work but could give you more freedom and a tighter integration
- Updates need to be made in several locations
  - riscv-opcodes (define your new encodings)
  - riscv-gnu-toolchain (add new instructions to assembler)
  - riscv-isa-sim (update/add instruction definition)
  - rocket (datapath and front-end updates)
Non-RoCC extension riscv-opcodes

- Repository for all encodings
- Generates
  - Header files gnu-toolchain
  - Header files for isa-sim
  - ISA manual tables
  - Chisel code to include in rocket
- Add the instruction to one of the opcodes files

make install
  - Generates all the different files and installs them in the correct folders
Non-RoCC extension riscv-gnu-toolchain

- Contains binutils, gcc, newlib and gcc ports
- Add instruction definition to
  `binutils/opcodes/riscv-opc.c`
- This is all that’s needed for simple instructions
- Rebuild the toolchain and you can assemble your new instruction
Non-RoCC extension riscv-isa-sim

- Already looked at this earlier for RoCC extensions
- Standard riscv instructions are defined in `riscv/insns`
- Adding the instruction to riscv-opcodes will cause spike to look for a header file in this folder with the instructions name
- The header file describes how the instruction behaves
- Many examples of different instructions to start with
Non-RoCC extension rocket

- Modifications to this code will greatly depend on the instruction.
- Simply adding a new ALU op would require very few changes.
- The complexity of the changes will depend greatly on the instruction.
- Happy to work through this with you in lab time but we’ll skip over it for now.
Rocket Chip Verilog

- The vsim directory contains build scripts to generate verilog with an ASIC backend in mind
  ```
  cd ../vsim && make
  ```
- The generated-src directory contains
  - Verilog source (`Top.$CONFIG.v`)
  - Set of exported parameters (`Top.$CONFIG.prm`)
  - Memory parameters (`Top.$CONFIG.conf`)
- Memory parameters are used in our flow to figure out which SRAMs to generate or request
- `vlsi_mem_gen` script is used by Berkeley to automate this process
- After this processing the verilog is ready for CAD tools
Rocket Chip Verilog for FPGA

- The fsim directory contains build scripts to generate verilog with an FPGA backend in mind

```bash
cd ../fsim && make
```

- The generated-src directory contains
  - Verilog source (`Top.$CONFIG.v`)
  - Set of exported parameters (`Top.$CONFIG.prm`)
  - Memory parameters (`Top.$CONFIG.conf`)

- `fpga_mem_gen` handles the memory configurations
- `fpga-zynq` repo has build scripts after this point but requires the fpga tools to run
- Well documented repo so refer to its README for more instructions
Lab Time!