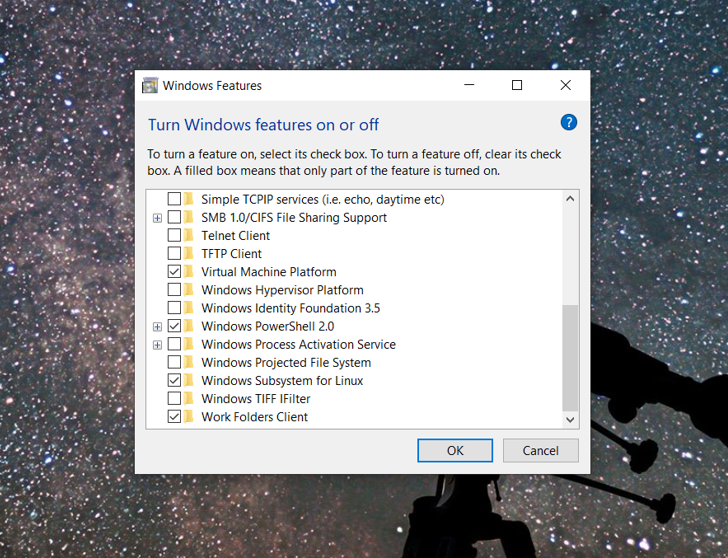
**6.1 Steps to arrive at the Litex console on UBUNTU**

1. Download the UBUNTU 20.04 LTS version from Microsoft store (Windows) and App store (MAC).



1. After the downloading and installation, provide a new username and password on the ubuntu window popped up.
2. Download the verilator from [Verilator](http://www.veripool.org/) or https://www.veripool.org/.
3. Go to the <https://github.com/enjoy-digital/litex> and scroll to the Quick start guide.
4. Copy the links below and go pasting them one by one until a next command prompts after every link you paste.

i)Install Python 3.6+ and FPGA vendor's development tools.

ii)Install Migen/LiteX and the LiteX's cores:

wget https://raw.githubusercontent.com/enjoydigital/litex/master/litex\_setup.py

chmod +x litex\_setup.py

./litex\_setup.py --init --install –user

iii) If you need to update all repositories:

./litex\_setup.py --update

1. Copy and paste the below links

sudo apt-get update

sudo apt-get upgrade

sudo apt install python3-pip

1. Copy and paste the below links

pip3 install meson ninja (if you get any trouble downloading this, repeat the 6th step)

./litex\_setup.py --gcc=riscv

sudo apt install libevent-dev libjson-c-dev verilator

lxsim --cpu-type=vexriscv

1. Now install a RISCV toolchain

wget <https://static.dev.sifive.com/dev-tools/riscv64-unknown-elf-gcc-8.1.0-2019.01.0-x86_64-linux-> ubuntu14.tar.gz

$ tar -xvf riscv64-unknown-elf-gcc-8.1.0-2019.01.0-x86\_64-linux-ubuntu14.tar.gz

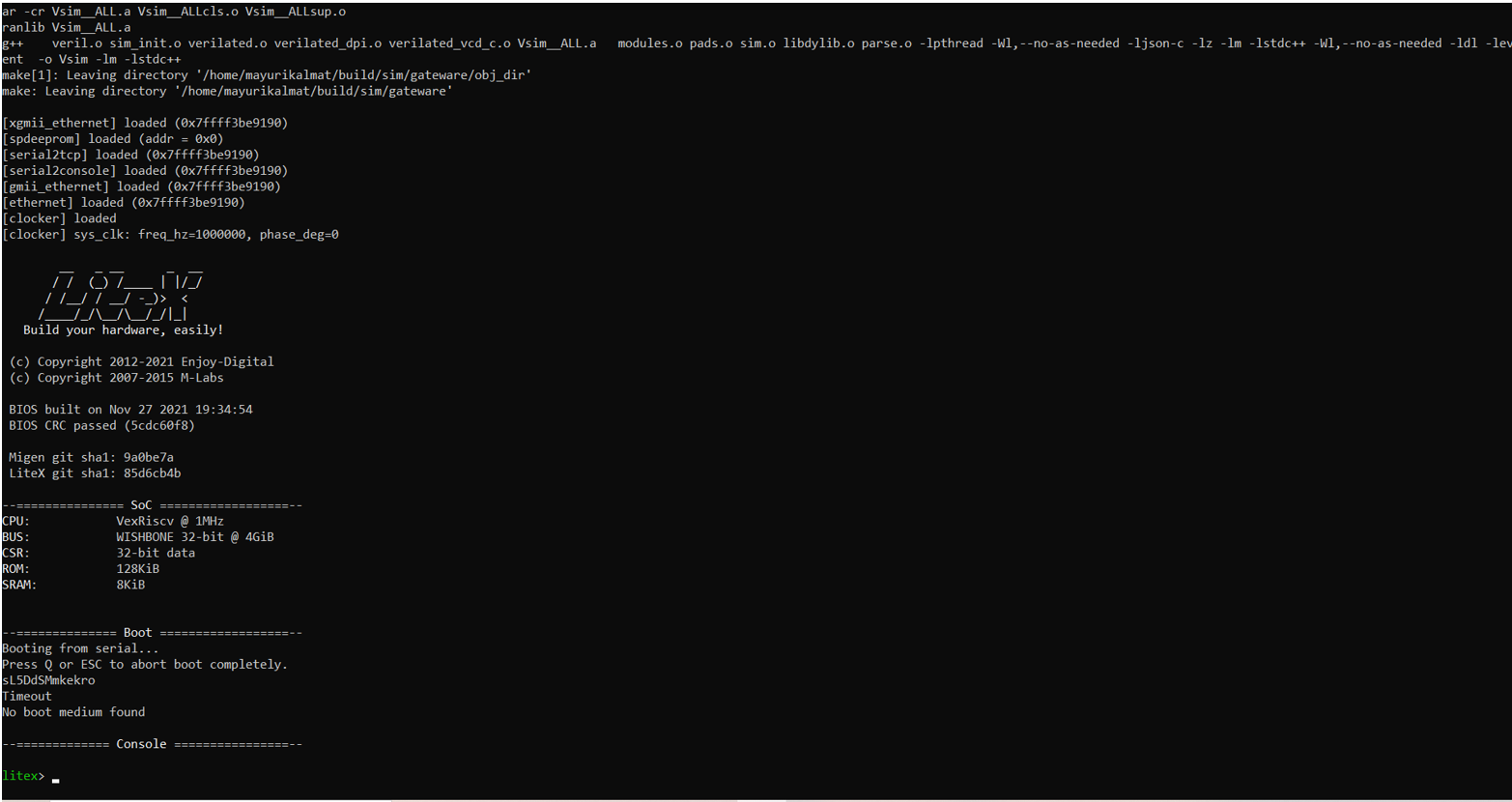
$ export PATH=$PATH:$PWD/riscv64-unknown-elf-gcc-8.1.0-2019.01.0-x86\_64-linux-ubuntu14/bin/

1. Now copy and paste the below command lines and run them on the screen.

./litex\_setup.py init

sudo ./litex\_setup.py install

At first you get a screen showing Litex but there might be no Console word in the end. Then repeat the above steps and the final stage will display the console having Litex and its properties as shown below.



**6.2 ROCKET CHIP**

We have tried developing rocket chip on LiteX. Rocket Chip is an open-source

Sysem-on-Chip design generator that emits synthesizable RTL. It leverages

the Chisel hardware construction language to compose a library of sophisti-

cated generators for cores, caches, and interconnects into an integrated SoC.

Rocket Chip generates general-purpose processor cores that use the open

RISC-V ISA, and provides both an in-order core generator (Rocket) and an

out-of-order core generator (BOOM). For SoC designers interested in uti-

lizing heterogeneous specialization for added efficiency gains, Rocket Chip

supports the integration of custom accelerators in the form of instruction set

extensions, coprocessors, or fully independent novel cores. Rocket Chip has

been taped out (manufactured) eleven times, and yielded functional silicon

prototypes capable of booting Linux.

**Setting up the RISCV environment variable**

To build the rocket-chip repository, you must point the RISCV environment variable to your rocket-tools installation directory.

$ export RISCV=/path/to/riscv/toolchain/installation

The rocket-tools repository known to work with rocket-chip is noted in the file riscv-tools.hash. However, any recent rocket-tools should work. You can build rocket-tools as follows:

$ git clone https://github.com/freechipsproject/rocket-tools

$ cd rocket-tools

$ git submodule update --init --recursive

$ export RISCV=/path/to/install/riscv/toolchain

$ export MAKEFLAGS="$MAKEFLAGS -jN" # Assuming you have N cores on your host system

$ ./build.sh

$ ./build-rv32ima.sh (if you are using RV32).

**Install Necessary Dependencies**

You may need to install some additional packages to use this repository. Rather than list all dependencies here, please see the appropriate section of the READMEs for each of the subprojects:

* [rocket-tools "Ubuntu Packages Needed"](https://github.com/freechipsproject/rocket-tools/blob/master/README.md)
* [chisel3 "Installation"](https://github.com/ucb-bar/chisel3#installation)

**Building The Project**

First, to build the C simulator:

$ cd emulator

$ make

Or to build the VCS simulator:

$ cd vsim

$ make

In either case, you can run a set of assembly tests or simple benchmarks (Assuming you have N cores on your host system):

$ make -jN run-asm-tests

$ make -jN run-bmark-tests

To build a C simulator that is capable of VCD waveform generation:

$ cd emulator

$ make debug

And to run the assembly tests on the C simulator and generate waveforms:

$ make -jN run-asm-tests-debug

$ make -jN run-bmark-tests-debug

To generate FPGA- or VLSI-synthesizable Verilog (output will be in vsim/generated-src):

$ cd vsim

$ make verilog

To run the Scala tests (sbt test) or linter (sbt scalafix):

$ cd regression

# Scala tests

$ make scalatest SUITE=foo

# Scala linter, automatically modifying files to correct issues

$ make scalafix SUITE=foo

# Scala linter, only printing out issues

$ make scalafix-check SUITE=foo

**Keeping Your Repo Up-to-Date**

If you are trying to keep your repo up to date with this GitHub repo, you also need to keep the submodules and tools up to date.

$ # Get the newest versions of the files in this repo

$ git pull origin master

$ # Make sure the submodules have the correct versions

$ git submodule update --init --recursive

If rocket-tools version changes, you should recompile and install rocket-tools according to the directions in the [rocket-tools/README](https://github.com/freechipsproject/rocket-tools/blob/master/README.md).

$ cd rocket-tools

$ ./build.sh

$ ./build-rv32ima.sh (if you are using RV32)

**Git Submodules**

[Git submodules](https://git-scm.com/book/en/v2/Git-Tools-Submodules) allow you to keep a Git repository as a subdirectory of another Git repository. For projects being co-developed with the Rocket Chip Generator, we have often found it expedient to track them as submodules, allowing for rapid exploitation of new features while keeping commit histories separate. As submoduled projects adopt stable public APIs, we transition them to external dependencies. Here are the submodules that are currently being tracked in the rocket-chip repository:

* **chisel3** (<https://github.com/ucb-bar/chisel3>): The Rocket Chip Generator uses [Chisel](http://chisel.eecs.berkeley.edu/) to generate RTL.
* **firrtl** (<https://github.com/ucb-bar/firrtl>): [Firrtl (Flexible Internal Representation for RTL)](http://bar.eecs.berkeley.edu/projects/2015-firrtl.html) is the intermediate representation of RTL constructions used by Chisel3. The Chisel3 compiler generates a Firrtl representation, from which the final product (Verilog code, C code, etc) is generated.
* **hardfloat** (<https://github.com/ucb-bar/berkeley-hardfloat>): Hardfloat holds Chisel code that generates parameterized IEEE 754-2008 compliant floating-point units used for fused multiply-add operations, conversions between integer and floating-point numbers, and conversions between floating-point conversions with different precision.
* **rocket-tools** (<https://github.com/freechipsproject/rocket-tools>): We tag a version of RISC-V software tools that work with the RTL committed in this repository.
* **torture** (<https://github.com/ucb-bar/riscv-torture>): This module is used to generate and execute constrained random instruction streams that can be used to stress-test both the core and uncore portions of the design.

**Scala Packages**

In addition to submodules that track independent Git repositories, the rocket-chip code base is itself factored into a number of Scala packages. These packages are all found within the src/main/scala directory. Some of these packages provide Scala utilities for generator configuration, while other contain the actual Chisel RTL generators themselves. Here is a brief description of what can be found in each package:

* **amba** This RTL package uses diplomacy to generate bus implementations of AMBA protocols, including AXI4, AHB-lite, and APB.
* **config** This utility package provides Scala interfaces for configuring a generator via a dynamically-scoped parameterization library.
* **coreplex** This RTL package generates a complete coreplex by gluing together a variety of components from other packages, including: tiled Rocket cores, a system bus network, coherence agents, debug devices, interrupt handlers, externally-facing peripherals, clock-crossers and converters from TileLink to external bus protocols (e.g. AXI or AHB).
* **devices** This RTL package contains implementations for peripheral devices, including the Debug module and various TL slaves.
* **diplomacy** This utility package extends Chisel by allowing for two-phase hardware elaboration, in which certain parameters are dynamically negotiated between modules. For more information about diplomacy, see [this paper](https://carrv.github.io/2017/papers/cook-diplomacy-carrv2017.pdf).
* **groundtest** This RTL package generates synthesizable hardware testers that emit randomized memory access streams in order to stress-tests the uncore memory hierarchy.
* **jtag** This RTL package provides definitions for generating JTAG bus interfaces.
* **regmapper** This utility package generates slave devices with a standardized interface for accessing their memory-mapped registers.
* **rocket** This RTL package generates the Rocket in-order pipelined core, as well as the L1 instruction and data caches. This library is intended to be used by a chip generator that instantiates the core within a memory system and connects it to the outside world.
* **tile** This RTL package contains components that can be combined with cores to construct tiles, such as FPUs and accelerators.
* **tilelink** This RTL package uses diplomacy to generate bus implementations of the TileLink protocol. It also contains a variety of adapters and protocol converters.
* **system** This top-level utility package invokes Chisel to elaborate a particular configuration of a coreplex, along with the appropriate testing collateral.
* **unittest** This utility package contains a framework for generateing synthesizable hardware testers of individual modules.
* **util** This utility package provides a variety of common Scala and Chisel constructs that are re-used across multiple other packages,

**Other Resources**

Outside of Scala, we also provide a variety of resources to create a complete SoC implementation and test the generated designs.

* **bootrom** Sources for the first-stage bootloader included in the BootROM.
* **csrc** C sources for use with Verilator simulation.
* **docs** Documentation, tutorials, etc for specific parts of the codebase.
* **emulator** Directory in which Verilator simulations are compiled and run.
* **project** Directory used by SBT for Scala compilation and build.
* **regression** Defines continuous integration and nightly regression suites.
* **scripts** Utilities for parsing the output of simulations or manipulating the contents of source files.
* **vsim** Directory in which Synopsys VCS simulations are compiled and run.
* **vsrc** Verilog sources containing interfaces, harnesses and VPI.

**Extending the Top-Level Design**

See [this description](https://github.com/ucb-bar/project-template) of how to create you own top-level design with custom devices.

**How should I use the Rocket chip generator?**

Chisel can generate code for three targets: a high-performance cycle-accurate Verilator, Verilog optimized for FPGAs, and Verilog for VLSI. The rocket-chip generator can target all three backends. You will need a Java runtime installed on your machine, since Chisel is overlaid on top of [Scala](http://www.scala-lang.org/). Chisel RTL (i.e. rocket-chip source code) is a Scala program executing on top of your Java runtime. To begin, ensure that the ROCKETCHIP environment variable points to the rocket-chip repository.

$ git clone https://github.com/ucb-bar/rocket-chip.git

$ cd rocket-chip

$ export ROCKETCHIP=`pwd`

$ git submodule update --init

Before going any further, you must point the RISCV environment variable to your rocket-tools installation directory. If you do not yet have rocket-tools installed, follow the directions in the [rocket-tools/README](https://github.com/freechipsproject/rocket-tools/blob/master/README.md).

export RISCV=/path/to/install/riscv/toolchain

Otherwise, you will see the following error message while executing any command in the rocket-chip generator:

\*\*\* Please set environment variable RISCV. Please take a look at README.

**1) Using the high-performance cycle-accurate Verilator**

Your next step is to get the Verilator working. Assuming you have N cores on your host system, do the following:

$ cd $ROCKETCHIP/emulator

$ make -jN run

By doing so, the build system will generate C++ code for the cycle-accurate emulator, compile the emulator, compile all RISC-V assembly tests and benchmarks, and run both tests and benchmarks on the emulator. If make finished without any errors, it means that the generated Rocket chip has passed all assembly tests and benchmarks!

You can also run assembly tests and benchmarks separately:

$ make -jN run-asm-tests

$ make -jN run-bmark-tests

To generate vcd waveforms, you can run one of the following commands:

$ make -jN run-debug

$ make -jN run-asm-tests-debug

$ make -jN run-bmark-tests-debug

Or call out individual assembly tests or benchmarks:

$ make output/rv64ui-p-add.out

$ make output/rv64ui-p-add.vcd

Now take a look in the emulator/generated-src directory. You will find Chisel generated Verilog code and its associated C++ code generated by Verilator.

$ ls $ROCKETCHIP/emulator/generated-src

freechips.rocketchip.system.DefaultConfig

freechips.rocketchip.system.DefaultConfig.0x0.0.regmap.json

freechips.rocketchip.system.DefaultConfig.0x0.1.regmap.json

freechips.rocketchip.system.DefaultConfig.0x2000000.0.regmap.json

freechips.rocketchip.system.DefaultConfig.0x40.0.regmap.json

freechips.rocketchip.system.DefaultConfig.0xc000000.0.regmap.json

freechips.rocketchip.system.DefaultConfig.anno.json

freechips.rocketchip.system.DefaultConfig.behav\_srams.v

freechips.rocketchip.system.DefaultConfig.conf

freechips.rocketchip.system.DefaultConfig.d

freechips.rocketchip.system.DefaultConfig.dts

freechips.rocketchip.system.DefaultConfig.fir

freechips.rocketchip.system.DefaultConfig.graphml

freechips.rocketchip.system.DefaultConfig.json

freechips.rocketchip.system.DefaultConfig.memmap.json

freechips.rocketchip.system.DefaultConfig.plusArgs

freechips.rocketchip.system.DefaultConfig.rom.conf

freechips.rocketchip.system.DefaultConfig.v

TestHarness.anno.json

$ ls $ROCKETCHIP/emulator/generated-src/freechips.rocketchip.system.DefaultConfig

VTestHarness\_\_1.cpp

VTestHarness\_\_2.cpp

VTestHarness\_\_3.cpp

...

Also, output of the executed assembly tests and benchmarks can be found at emulator/output/\*.out. Each file has a cycle-by-cycle dump of write-back stage of the pipeline. Here's an excerpt of emulator/output/rv64ui-p-add.out:

C0: 483 [1] pc=[00000002138] W[r 3=000000007fff7fff][1] R[r 1=000000007fffffff] R[r 2=ffffffffffff8000] inst=[002081b3] add s1, ra, s0

C0: 484 [1] pc=[0000000213c] W[r29=000000007fff8000][1] R[r31=ffffffff80007ffe] R[r31=0000000000000005] inst=[7fff8eb7] lui t3, 0x7fff8

C0: 485 [0] pc=[00000002140] W[r 0=0000000000000000][0] R[r 0=0000000000000000] R[r 0=0000000000000000] inst=[00000000] unknown

The first [1] at cycle 483, core 0, shows that there's a valid instruction at PC 0x2138 in the writeback stage, which is 0x002081b3 (add s1, ra, s0). The second [1] tells us that the register file is writing r3 with the corresponding value 0x7fff7fff. When the add instruction was in the decode stage, the pipeline had read r1 and r2 with the corresponding values next to it. Similarly at cycle 484, there's a valid instruction (lui instruction) at PC 0x213c in the writeback stage. At cycle 485, there isn't a valid instruction in the writeback stage, perhaps, because of a instruction cache miss at PC 0x2140.

**2) Mapping a Rocket core to an FPGA**

You can generate synthesizable Verilog with the following commands:

$ cd $ROCKETCHIP/vsim

$ make verilog CONFIG=freechips.rocketchip.system.DefaultFPGAConfig

The Verilog used for the FPGA tools will be generated in vsim/generated-src. Please proceed further with the directions shown in the [README](https://github.com/sifive/freedom/blob/master/README.md) of the freedom repository. You can also run Rocket Chip on Amazon EC2 F1 with [FireSim](https://github.com/firesim/firesim).

If you have access to VCS, you will be able to run assembly tests and benchmarks in simulation with the following commands (again assuming you have N cores on your host machine):

$ cd $ROCKETCHIP/vsim

$ make -jN run CONFIG=freechips.rocketchip.system.DefaultFPGAConfig

The generated output looks similar to those generated from the emulator. Look into vsim/output/\*.out for the output of the executed assembly tests and benchmarks.

**3) Pushing a Rocket core through the VLSI tools**

You can generate Verilog for your VLSI flow with the following commands:

$ cd $ROCKETCHIP/vsim

$ make verilog

Now take a look at vsim/generated-src, and the contents of the Top.DefaultConfig.conf file:

$ cd $ROCKETCHIP/vsim/generated-src

freechips.rocketchip.system.DefaultConfig

freechips.rocketchip.system.DefaultConfig.0x0.0.regmap.json

freechips.rocketchip.system.DefaultConfig.0x0.1.regmap.json

freechips.rocketchip.system.DefaultConfig.0x2000000.0.regmap.json

freechips.rocketchip.system.DefaultConfig.0x40.0.regmap.json

freechips.rocketchip.system.DefaultConfig.0xc000000.0.regmap.json

freechips.rocketchip.system.DefaultConfig.anno.json

freechips.rocketchip.system.DefaultConfig.behav\_srams.v

freechips.rocketchip.system.DefaultConfig.conf

freechips.rocketchip.system.DefaultConfig.d

freechips.rocketchip.system.DefaultConfig.dts

freechips.rocketchip.system.DefaultConfig.fir

freechips.rocketchip.system.DefaultConfig.graphml

freechips.rocketchip.system.DefaultConfig.json

freechips.rocketchip.system.DefaultConfig.memmap.json

freechips.rocketchip.system.DefaultConfig.plusArgs

freechips.rocketchip.system.DefaultConfig.rom.conf

freechips.rocketchip.system.DefaultConfig.v

TestHarness.anno.json

$ cat $ROCKETCHIP/vsim/generated-src/\*.conf

name data\_arrays\_0\_ext depth 512 width 256 ports mrw mask\_gran 8

name tag\_array\_ext depth 64 width 88 ports mrw mask\_gran 22

name tag\_array\_0\_ext depth 64 width 84 ports mrw mask\_gran 21

name data\_arrays\_0\_1\_ext depth 512 width 128 ports mrw mask\_gran 32

name mem\_ext depth 33554432 width 64 ports mwrite,read mask\_gran 8

name mem\_2\_ext depth 512 width 64 ports mwrite,read mask\_gran 8

The conf file contains information for all SRAMs instantiated in the flow. If you take a close look at the $ROCKETCHIP/Makefrag, you will see that during Verilog generation, the build system calls a $(mem\_gen) script with the generated configuration file as an argument, which will fill in the Verilog for the SRAMs. Currently, the $(mem\_gen) script points to vsim/vlsi\_mem\_gen, which simply instantiates behavioral SRAMs. You will see those SRAMs being appended at the end of vsim/generated-src/Top.DefaultConfig.v. To target vendor-specific SRAMs, you will need to make necessary changes to vsim/vlsi\_mem\_gen.

Similarly, if you have access to VCS, you can run assembly tests and benchmarks with the following commands (again assuming you have N cores on your host machine):

$ cd $ROCKETCHIP/vsim

$ make -jN run

The generated output looks similar to those generated from the emulator. Look into vsim/output/\*.out for the output of the executed assembly tests and benchmarks.

**How can I parameterize my Rocket chip?**

By now, you probably figured out that all generated files have a configuration name attached, e.g. freechips.rocketchip.system.DefaultConfig. Take a look at src/main/scala/system/Configs.scala. Search for NSets and NWays defined in BaseConfig. You can change those numbers to get a Rocket core with different cache parameters. For example, by changing L1I, NWays to 4, you will get a 32KB 4-way set-associative L1 instruction cache rather than a 16KB 2-way set-associative L1 instruction cache. Towards the end, you can also find that DefaultSmallConfig inherits all parameters from BaseConfig but overrides the same parameters of WithNSmallCores.

Now take a look at vsim/Makefile. Search for the CONFIG variable. By default, it is set to freechips.rocketchip.system.DefaultConfig. You can also change the CONFIG variable on the make command line:

$ cd $ROCKETCHIP/vsim

$ make -jN CONFIG=freechips.rocketchip.system.DefaultSmallConfig run-asm-tests

Or, even by defining CONFIG as an environment variable:

$ export CONFIG=freechips.rocketchip.system.DefaultSmallConfig

$ make -jN run-asm-tests

This parameterization is one of the many strengths of processor generators written in Chisel, and will be more detailed in a future blog post, so please stay tuned.

To override specific configuration items, such as the number of external interrupts, you can create your own Configuration(s) and compose them with Config's ++ operator

class WithNExtInterrupts(nExt: Int) extends Config {

(site, here, up) => {

case NExtInterrupts => nExt

}

}

class MyConfig extends Config (new WithNExtInterrupts(16) ++ new DefaultSmallConfig)

Then you can build as usual with CONFIG=<MyConfigPackage>.MyConfig.

**Debugging with GDB**

**1) Generating the Remote Bit-Bang (RBB) Emulator**

The objective of this section is to use GNU debugger to debug RISC-V programs running on the emulator in the same fashion as in [Spike](https://github.com/riscv/riscv-isa-sim#debugging-with-gdb).

For that we need to add a Remote Bit-Bang client to the emulator. We can do so by extending our Config with JtagDTMSystem, which will add a DebugTransportModuleJTAG to the DUT and connect a SimJTAG module in the Test Harness. This will allow OpenOCD to interface with the emulator, and GDB can interface with OpenOCD. In the following example we add this Config alteration to src/main/scala/system/Configs.scala:

class DefaultConfigRBB extends Config(

new WithJtagDTMSystem ++ new WithNBigCores(1) ++ new WithCoherentBusTopology ++ new BaseConfig)

class QuadCoreConfigRBB extends Config(

new WithJtagDTMSystem ++ new WithNBigCores(4) ++ new WithCoherentBusTopology ++ new BaseConfig)

To build the emulator with DefaultConfigRBB configuration we use the command:

rocket-chip$ cd emulator

emulator$ CONFIG=freechips.rocketchip.system.DefaultConfigRBB make

We can also build a debug version capable of generating VCD waveforms using the command:

emulator$ CONFIG=freechips.rocketchip.system.DefaultConfigRBB make debug

By default the emulator is generated under the name emulator-freechips.rocketchip.system-DefaultConfigRBB in the first case and emulator-freechips.rocketchip.system-DefaultConfigRBB-debug in the second.

**2) Compiling and executing a custom program using the emulator**

We suppose that helloworld is our program, you can use crt.S, syscalls.c and the linker script test.ld to construct your own program, check examples stated in [riscv-tests](https://github.com/riscv/riscv-tests). Note that test.ld loads the program at 0x80000000 so you will need to use -mcmodel=medany otherwise you will get relocation errors. See [All Aboard, Part 4: The RISC-V Code Models](https://www.sifive.com/blog/2017/09/11/all-aboard-part-4-risc-v-code-models/) for more details.

In our case we will use the following example:

char text[] = "Vafgehpgvba frgf jnag gb or serr!";

// Don't use the stack, because sp isn't set up.

volatile int wait = 1;

int main()

{

while (wait)

;

// Doesn't actually go on the stack, because there are lots of GPRs.

int i = 0;

while (text[i]) {

char lower = text[i] | 32;

if (lower >= 'a' && lower <= 'm')

text[i] += 13;

else if (lower > 'm' && lower <= 'z')

text[i] -= 13;

i++;

}

while (!wait)

;

}

First we can test if your program executes well in the simple version of emulator before moving to debugging in step 3 :

$ ./emulator-freechips.rocketchip.system-DefaultConfig helloworld

Additional verbose information (clock cycle, pc, instruction being executed) can be printed using the following command:

$ ./emulator-freechips.rocketchip.system-DefaultConfig +verbose helloworld 2>&1 | spike-dasm

VCD output files can be obtained using the -debug version of the emulator and are specified using -v or --vcd=FILE arguments. A detailed log file of all executed instructions can also be obtained from the emulator, this is an example:

$ ./emulator-freechips.rocketchip.system-DefaultConfig-debug +verbose -v output.vcd helloworld 2>&1 | spike-dasm > output.log

Please note that generated VCD waveforms and execution log files can be very voluminous depending on the size of the .elf file (i.e. code size + debugging symbols).

Please note also that the time it takes the emulator to load your program depends on executable size. Stripping the .elf executable will unsurprisingly make it run faster. For this you can use $RISCV/bin/riscv64-unknown-elf-strip tool to reduce the size. This is good for accelerating your simulation but not for debugging. Keep in mind that the HTIF communication interface between our system and the emulator relies on tohost and fromhost symbols to communicate. This is why you may get the following error when you try to run a totally stripped executable on the emulator:

$ ./emulator-freechips.rocketchip.system-DefaultConfig totally-stripped-helloworld

This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag\_rbb\_enable=1.

Listening on port 46529

warning: tohost and fromhost symbols not in ELF; can't communicate with target

To resolve this, we need to strip all the .elf executable but keep tohost and fromhost symbols using the following command:

$riscv64-unknown-elf-strip -s -Kfromhost -Ktohost helloworld

More details on the GNU strip tool can be found [here](https://www.thegeekstuff.com/2012/09/strip-command-examples/).

The interest of this step is to make sure your program executes well. To perform debugging you need the original unstripped version, as explained in step 3.

**3) Launch the emulator**

First, do not forget to compile your program with -g -Og flags to provide debugging support as explained [here](https://github.com/riscv/riscv-isa-sim#debugging-with-gdb).

We can then launch the Remote Bit-Bang enabled emulator with:

./emulator-freechips.rocketchip.system-DefaultConfigRBB +jtag\_rbb\_enable=1 --rbb-port=9823 helloworld

This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag\_rbb\_enable=1.

Listening on port 9823

Attempting to accept client socket

You can also use the emulator-freechips.rocketchip.system-DefaultConfigRBB-debug version instead if you would like to generate VCD waveforms.

Please note that if the argument --rbb-port is not passed, a default free TCP port on your computer will be chosen randomly.

Please note also that when debugging with GDB, the .elf file is not actually loaded by the FESVR. In contrast with Spike, it must be loaded from GDB as explained in step 5. So the helloworld argument may be replaced by any dummy name.

**4) Launch OpenOCD**

You will need a RISC-V Enabled OpenOCD binary. This is installed with rocket-tools in $(RISCV)/bin/openocd, or can be compiled manually from riscv-openocd. OpenOCD requires a configuration file, in which we define the RBB port we will use, which is in our case 9823.

$ cat cemulator.cfg

interface remote\_bitbang

remote\_bitbang\_host localhost

remote\_bitbang\_port 9823

set \_CHIPNAME riscv

jtag newtap $\_CHIPNAME cpu -irlen 5

set \_TARGETNAME $\_CHIPNAME.cpu

target create $\_TARGETNAME riscv -chain-position $\_TARGETNAME

gdb\_report\_data\_abort enable

init

halt

Then we launch OpenOCD in another terminal using the command

$(RISCV)/bin/openocd -f ./cemulator.cfg

Open On-Chip Debugger 0.10.0+dev-00112-g3c1c6e0 (2018-04-12-10:40)

Licensed under GNU GPL v2

For bug reports, read

http://openocd.org/doc/doxygen/bugs.html

Warn : Adapter driver 'remote\_bitbang' did not declare which transports it allows; assuming legacy JTAG-only

Info : only one transport option; autoselect 'jtag'

Info : Initializing remote\_bitbang driver

Info : Connecting to localhost:9823

Info : remote\_bitbang driver initialized

Info : This adapter doesn't support configurable speed

Info : JTAG tap: riscv.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000, ver: 0x0)

Info : datacount=2 progbufsize=16

Info : Disabling abstract command reads from CSRs.

Info : Disabling abstract command writes to CSRs.

Info : [0] Found 1 triggers

Info : Examined RISC-V core; found 1 harts

Info : hart 0: XLEN=64, 1 triggers

Info : Listening on port 3333 for gdb connections

Info : Listening on port 6666 for tcl connections

Info : Listening on port 4444 for telnet connections

A -d flag can be added to the command to show further debug information.

**5) Launch GDB**

In another terminal launch GDB and point to the elf file you would like to load then run it with the debugger (in this example, helloworld):

$ riscv64-unknown-elf-gdb helloworld

GNU gdb (GDB) 8.0.50.20170724-git

Copyright (C) 2017 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law. Type "show copying"

and "show warranty" for details.

This GDB was configured as "--host=x86\_64-pc-linux-gnu --target=riscv64-unknown-elf".

Type "show configuration" for configuration details.

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>.

Find the GDB manual and other documentation resources online at:

<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from ./proj1.out...done.

(gdb)

Compared to Spike, the C Emulator is very slow, so several problems may be encountered due to timeouts between issuing commands and response from the emulator. To solve this problem, we increase the timeout with the GDB set remotetimeout command.

After that we load our program by performing a load command. This automatically sets the $PC to the \_start symbol in our .elf file.

(gdb) set remotetimeout 2000

(gdb) target remote localhost:3333

Remote debugging using localhost:3333

0x0000000000010050 in ?? ()

(gdb) load

Loading section .text.init, size 0x2cc lma 0x80000000

Loading section .tohost, size 0x48 lma 0x80001000

Loading section .text, size 0x98c lma 0x80001048

Loading section .rodata, size 0x158 lma 0x800019d4

Loading section .rodata.str1.8, size 0x20 lma 0x80001b30

Loading section .data, size 0x22 lma 0x80001b50

Loading section .sdata, size 0x4 lma 0x80001b74

Start address 0x80000000, load size 3646

Transfer rate: 40 bytes/sec, 520 bytes/write.

(gdb)

Now we can proceed as with Spike, debugging works in a similar way:

(gdb) print wait

$1 = 1

(gdb) print wait=0

$2 = 0

(gdb) print text

$3 = "Vafgehpgvba frgf jnag gb or serr!"

(gdb) c

Continuing.

^C

Program received signal SIGINT, Interrupt.

main (argc=0, argv=<optimized out>) at src/main.c:33

33 while (!wait)

(gdb) print wait

$4 = 0

(gdb) print text

$5 = "Instruction sets want to be free!"

(gdb)

Further information about GDB debugging is available [here](https://sourceware.org/gdb/onlinedocs/gdb/) and [here](https://sourceware.org/gdb/onlinedocs/gdb/Remote-Debugging.html#Remote-Debugging).

**Building Rocket Chip with an IDE**

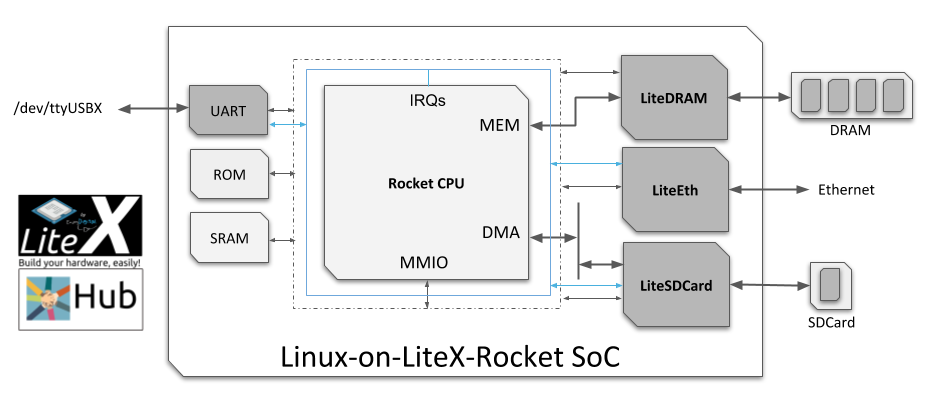
The Rocket Chip Scala build uses the standard Scala build tool SBT. IDEs like [IntelliJ](https://www.jetbrains.com/idea/) and [VSCode](https://code.visualstudio.com/) are popular in the Scala community and work with Rocket Chip. To use one of these IDEs, there is one minor peculiarity of the Rocket Chip build that must be addressed.

If the file .sbtopts exists in the root of the repository, you need to expand the $PWD variable inside of the file to an absolute path pointing to the location of your Rocket Chip clone. You can do this in bash with:

sed -i "s|\$PWD|$PWD|" .sbtopts

*If the file .sbtopts does not exist, you do not need to do anything special.*

If .sbtopts does not exist or if you have expanded the $PWD variable inside of it, you can import Rocket Chip into your IDE of choice.



**THANK YOU**