COA COURSE PROJECT (TEAM 3)

Course And Project Guide : Proff. Satyadhyan Chickerur

January 2022

Contents

1 TEAM MEMBERS:

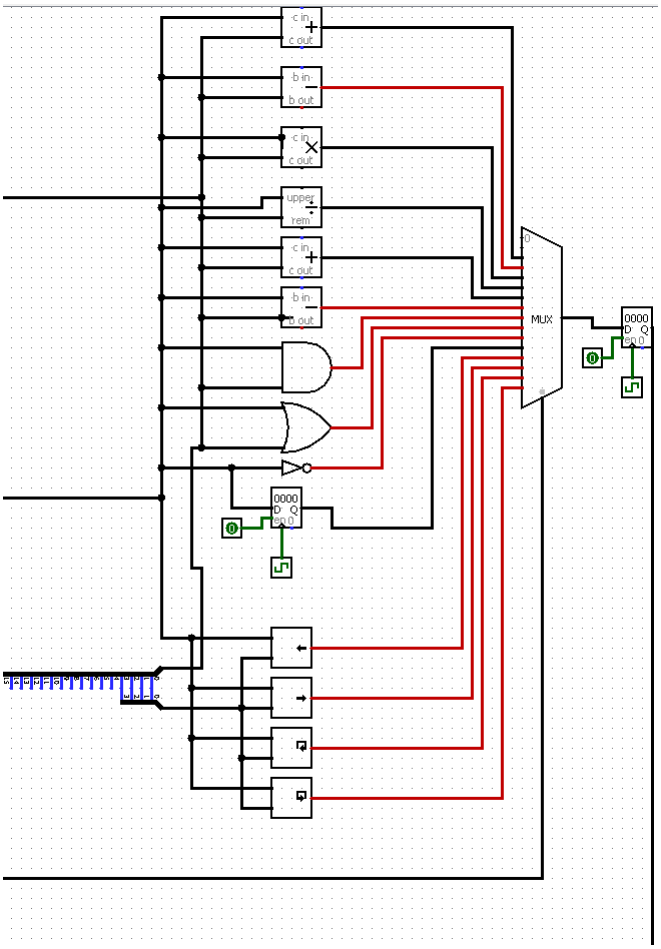
- Mayuri Kalmat - 230 - 01fe20bcs095
- Parag Hegde - 231 - 01fe20bcs096
- Pranav Jadhav - 234 - 01fe20bcs099

2 AIM :

To design and simulate a 16-bit processor.

3 PROCESSOR

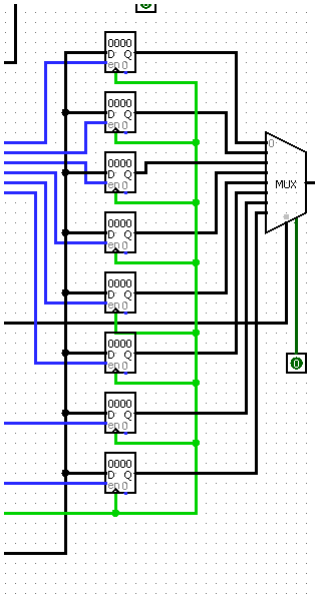
3.1 ARITHEMATIC AND LOGIC UNIT:



An Arithmetic Logic Unit (ALU) in computing is a combinational digital circuit that performs **arithmetic and bitwise operations on integer binary numbers**. This is in contrast to a ***Floating-Point Unit***

(FPU), which operates on **floating point numbers.** ***A multiplexer (MUX)*** is a device that can receive multiple input signals and synthesize a single output signal in a recoverable manner for each input signal. It is also an integrated system that usually contains a certain number of data inputs and a single output.

3.2 REGISTERS:



Registers are a type of computer memory made up of several flip flops used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU.

* The registers used by the ***CPU*** are often termed as ***Processor registers***.

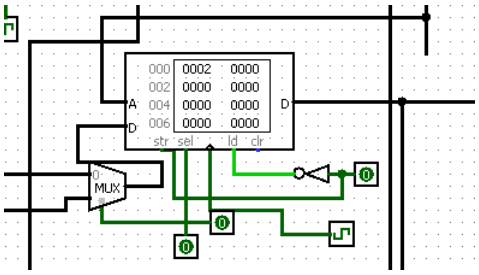
* A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

* The computer needs processor registers for manipulating data and a register for holding a memory address.

* The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

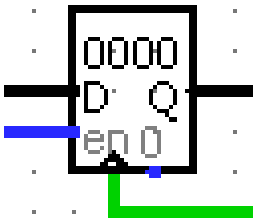
3.3 MAIN MEMORY :

figure:



Main memory: Very closely connected to the processor. The contents are quickly and easily changed. Holds the programs and data that the processor is actively working with. This interacts with the processor ***millions of times per second***.

3.4 ACCUMULATOR :



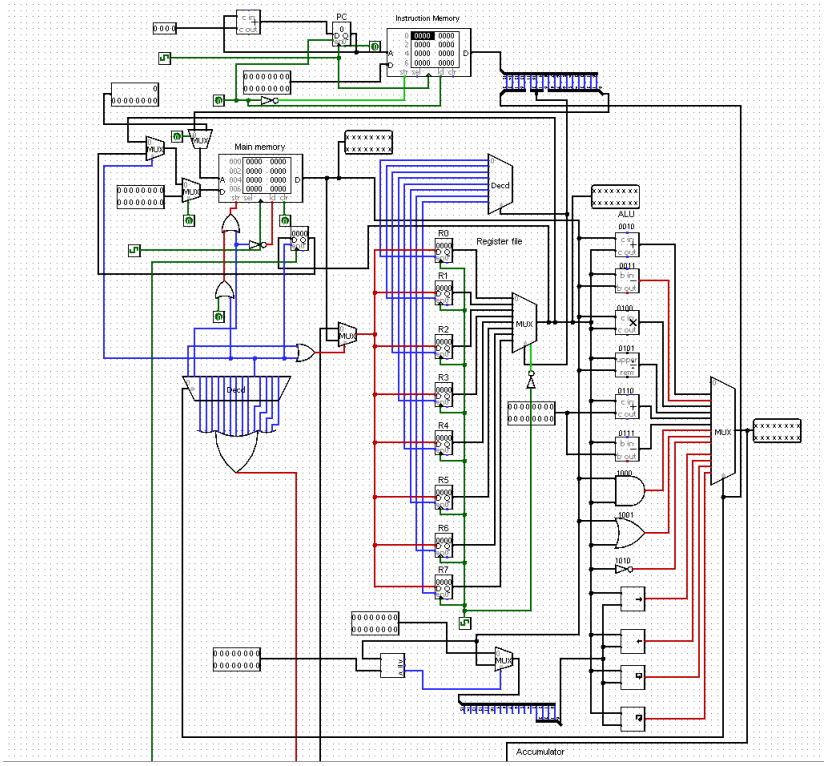
An accumulator is a register for short-term, intermediate storage of arithmetic and logic data in a computer's CPU (central processing unit).

The most elementary use for an accumulator is ***adding a sequence of numbers***.

The numerical value in the accumulator increases as each number is added.

3.5 16 BIT PROCESSOR:

figure:



4 TAKE AWAY FROM THIS PROJECT:

We learnt the ***complete functionality*** of the Processor. We visualized ***how data flows*** from one part to another part of register. We were able to ***appreciate the organization of processor*** and its functionalities.

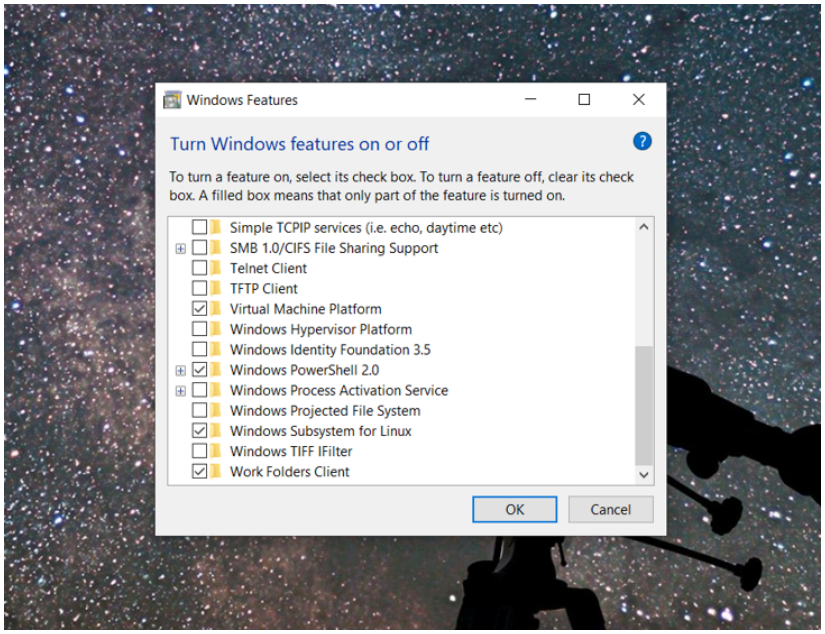
5 RESOURCES :

Wikipedia: https://en.wikipedia.org/wiki/16-bit_computing
Youtube videos
Github documentation

6 LITEX REPORT

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). figure:



2.After the downloading and installation, provide a new username and password on the ubuntu window popped up. 3.Download the verilator from Verilator or https://www.veripool.org/. 4.Go to the https://github.com/enjoy-digital/litex and scroll to the Quick start guide. 5.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 or 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
```

6. Copy and paste the below links :

```
-->sudo apt-get update
-->sudo apt-get upgrade
-->sudo apt install python3-pip
```

7. 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
```

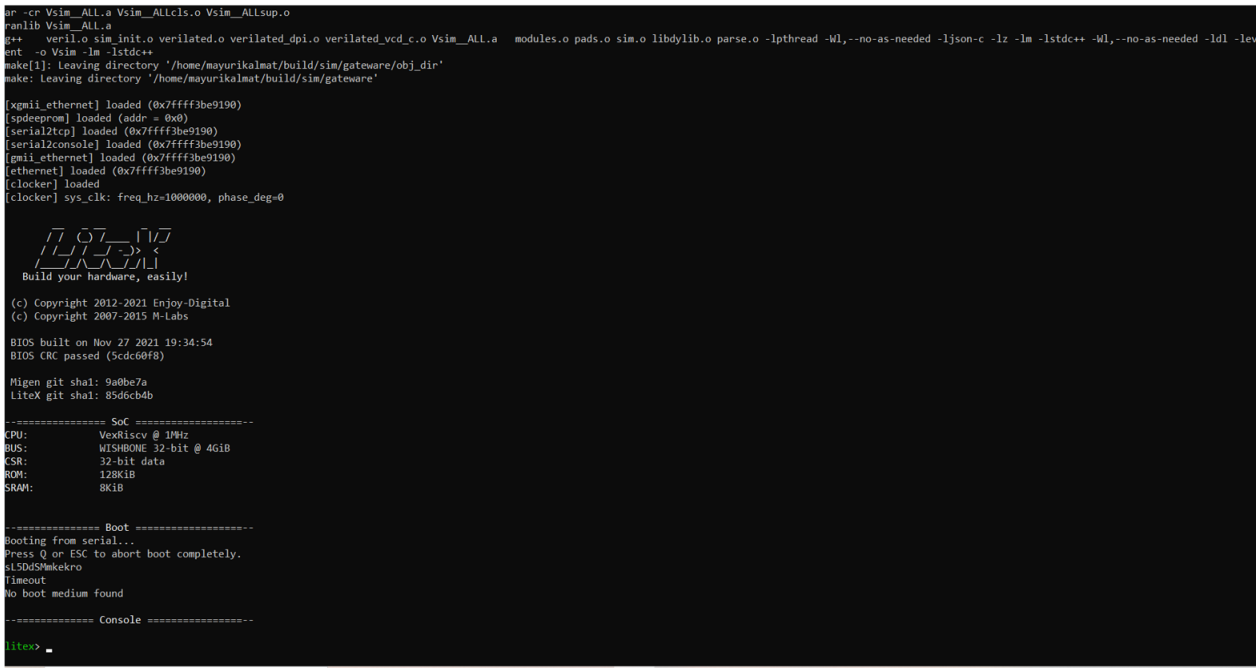
8. Now install a RISCV toolchain

```
-->(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/
```

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

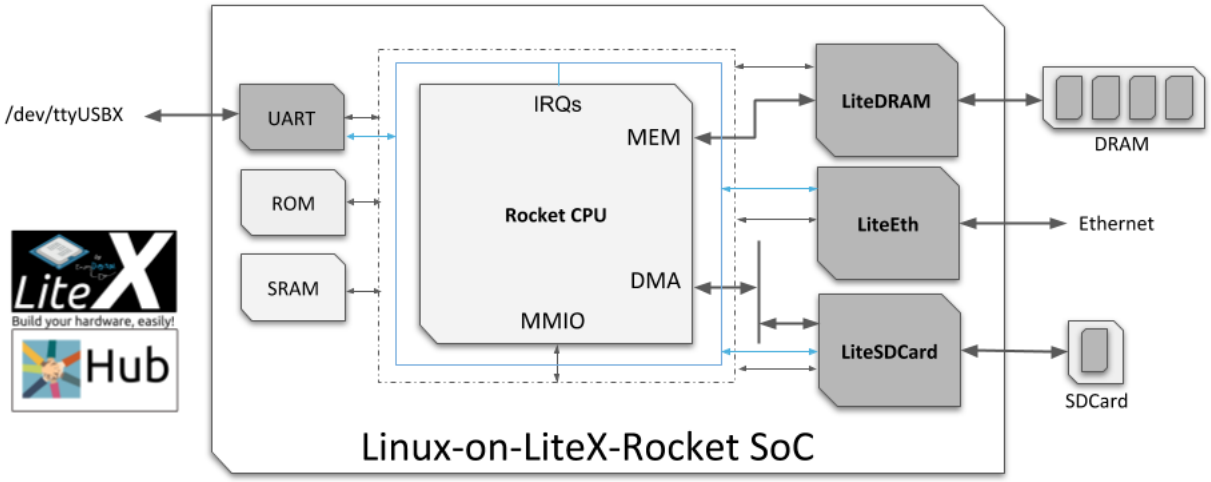
```
-->./litex_setup.py init
-->sudo ./litex_setup.py install
```

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 sophisticated 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 utilizing 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.



Source :https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

6.3 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
$./build.sh$
$./build - rv32ima.sh(if you are using RV32)$.

6.4 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"
• 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

6.5 Scala tests

make scalatest SUITE=foo

6.6 Scala linter, automatically modifying files to correct issues

make scalafix SUITE=foo

6.7 Scala linter, only printing out issues

make scalafix-check SUITE=foo

6.8 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
$gitpulloriginmaster$
Make sure the submodules have the correct versions
$gitsubmoduleupdate--init--recursive$
If rocket-tools version changes, you should recompile and install rocket-tools accordin
cd rocket-tools
cd rocket-tools
./build.sh
./build-rv32ima.sh (if you are using RV32)

6.9 What's in the Rocket chip generator repository?

The rocket-chip repository is a meta-repository that points to several sub-repositories using Git submodules. Those repositories contain tools needed to generate and test SoC designs. This respository also contains code that is used to generate RTL. Hardware generation is done using Chisel, a hardware construction language embedded in Scala. The rocket-chip generator is a Scala program that invokes the Chisel compiler in order to emit RTL describing a complete SoC. The following sections describe the components of this repository.

6.10 Git Submodules

Git 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 to generate RTL.
• firrtl (https://github.com/ucb-bar/firrtl): Firrtl (Flexible Internal Representation for RTL) 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.

6.11 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.
• 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,

6.12 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.

6.13 Extending the Top-Level Design

See this description of how to create you own top-level design with custom devices.

6.14 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. 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.
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 $ROCKETCHIP/emulator$
$make - jNrun$
$Bydoingso, thebuildsystemwillgenerateC++codeforthecycle-accurateemulator, compiletheemul$
$Vassemblytestsandbenchmarks, andrunbothtestsandbenchmarksontheemulator.IfMakefinishe$
$Youcanalsorunassemblytestsandbenchmarksseparately$:
$make - jNrun - asm - tests$
$make - jNrun - bmark - tests$

$Togeneratevcdwaveforms, youcanrunoneofthefollowingcommands$:
$make - jNrun - debug$
$make - jNrun - asm - tests - debug$
$make - jNrun - bmark - tests - debug$
$Orcalloutindividualassemblytestsorbenchmarks$:
$makeoutput/rv64ui - p - add.out$
$makeoutput/rv64ui - p - add.vcd$
$Nowtakealookintheemulator/generated-srcdirectory.Youwillfind ChiselgeneratedVerilogcode$
$+codegeneratedbyVerilator.$
lsROCKETCHIP/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$_srams.v$
$freechips.rocketchip.system.DefaultConfig.conf$
$freechips.rocketchip.system.DefaultConfig.d$
$freechips.rocketchip.system.DefaultConfig.dts$
$freechips.rocketchip.system.DefaultConfig.fir$
$freechips.rocketchip.system.DefaultConfig.graphml$
$freechips.rocketchip.system.DefaultConfig.json$

$freechips.rocketchip.system.DefaultConfig.memmap.json$
$freechips.rocketchip.system.DefaultConfig.plusArgs$
$freechips.rocketchip.system.DefaultConfig.rom.conf$
$freechips.rocketchip.system.DefaultConfig.v$
$TestHarness.anno.json$
lsROCKETCHIP/emulator/generated-
src/freechips.rocketchip.system.DefaultConfig
VTestHarness$_{1.cppVTestHarness_{2.cppVTestHarness_{3.cppAlso,outputoftheexecutedassemblytestsandbenchmarkscanbefoundatemulator/}}}$

6.15 2) Mapping a Rocket core to an FPGA

You can generate synthesizable Verilog with the following commands:
cd $ROCKETCHIP/vsim$
$makeverilogCONFIG = freechips.rocketchip.system.DefaultFPGAConfig$
$TheVerilogusedfortheFPGAtoolswillbegeneratedinvsim/generated-src.Pleaseproceedfurtheru$

cdROCKETCHIP/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.

6.16 3) Pushing a Rocket core through the VLSI tools

You can generate Verilog for your VLSI flow with the following commands:
cd $ROCKETCHIP/vsim$
$makeverilog$
$Nowtakealookatvsim/generated-src, andthecontentsoftheTop.DefaultConfig.conffile$:

cdROCKETCHIP/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$_srams.v$
$freechips.rocketchip.system.DefaultConfig.conf$
$freechips.rocketchip.system.DefaultConfig.d$
$freechips.rocketchip.system.DefaultConfig.dts$
$freechips.rocketchip.system.DefaultConfig.fir$
$freechips.rocketchip.system.DefaultConfig.graphml$
$freechips.rocketchip.system.DefaultConfig.json$
$freechips.rocketchip.system.DefaultConfig.memmap.json$
$freechips.rocketchip.system.DefaultConfig.plusArgs$
$freechips.rocketchip.system.DefaultConfig.rom.conf$
$freechips.rocketchip.system.DefaultConfig.v$
$TestHarness.anno.json$
catROCKETCHIP/vsim/generated-src/*.conf
name data$_arrays_{0e}xtdepth512width256portsmrwmask_gran8$
$nametag_array_extdepth64width88portsmrwmask_gran22$
$nametag_arrray_{0e}xtdepth64width84portsmrwmask_gran21$
$namedata_arrays_{01e}xtdepth512width128portsmrwmask_gran32$
$namemem_extdepth33554432width64portsmwrite, readmask_gran8$
$namemem_{2e}xtdepth512width64portsmwrite, readmask_gran8$
$Theconffilecontainsinformationforall SRAMsinstantiatedintheflow.Ifyoutakeacloselookatth$
you will see that during Verilog generation, the build system calls a $(mem_gen)scriptwiththegenerate$
$src/Top.DefaultConfig.v.Totargetvendor-specificSRAMs, youwillneedtomakenecessarychan$
$Similarly, ifyouhaveaccesstoVCS, youcanrunassemblytestsandbenchmarkswiththefollowingco$

cdROCKETCHIP/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.

6.17 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

6.18 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.
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 De-

bugTransportModuleJTAG 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*cdemulator*
emulator CONFIG=freechips.rocketchip.system.DefaultConfigRBB make We can also build a debug version capable of generating VCD waveforms using the command:
emulator$CONFIG = freechips.rocketchip.system.DefaultConfigRBBmakedebugBydefaultth$
$freechips.rocketchip.system-DefaultConfigRBBinthefirstcaseandemulator-$
$freechips.rocketchip.system - DefaultConfigRBB - debuginthesecond.$

6.19 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. 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 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. Kee

$./emulator-freechips.rocketchip.system-DefaultConfig totally-stripped-helloworld This emulator compiled with JTAG Remote Bitbang client. To enable, use + jtag_rbb_enable = 1.$
$Listening on port 46529$

$warning : tohostandfromhostsymbolsnotinELF; can'tcommunicatewithtargetToresolvethis, wer$

$riscv64-unknown-elf-strip-s-Kfromhost-Ktohosthelloworld More details on the GNU stript$
$The interest of this step is to make sure your program executes well. To perform debugging you need the o$

6.20 3) Launch the emulator

First, do not forget to compile your program with -g -Og flags to provide debugging support as explained here.
We can then launch the Remote Bit-Bang enabled emulator with:
./emulator-freechips.rocketchip.system-DefaultConfigRBB +jtag$_r bb_e nable =$
$1 - -rbb - port = 9823 helloworld$
$This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_r bb_e nable =$
1.
$Listening on port 9823$
$Attempting to accept client socket$
$You can also use the emulator - freechips.rocketchip.system - DefaultConfigRBB -$
$debug version instead if you would like to generate VCD waveforms.$
$Please note that if the argument - -rbb - port is not passed, a default free TCP port on your computer w$
$Please note also that when debugging with GDB, the .elf file is not actually loaded by the FESVR. Incon$

6.21 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$_b itbang$
$remote_b itbang_h ostlocalhost$
$remote_b itbang_p ort 9823$

$set_C HIPNAME riscv$
$jtag newtap_C HIPNAME cpu - irlen 5$

$set_T ARGETNAME_C HIPNAME.cpu$

$target create_{T}ARGETNAME riscv -chain-position_{T}ARGETNAME$

$gdb_{r}eport_{d}ata_{a}bort enable$

$init$
$halt$
$Then we launch OpenOCD in another terminal using the command$
$(RISCV)/bin/openocd -f ./cemulator.cfg$
$OpenOn-ChipDebugger 0.10.0+dev-00112-g3c1c6e0 (2018-04-12-10:$
$40)$
$Licensed under GNU GPL v2$
$For bug reports, read$
$http://openocd.org/doc/doxygen/bugs.html$
$Warn : Adapter driver 'remote_bitbang' did not declare which transports it allows; assuming legacy JT$
$only$
$Info : only one transport option; autoselect 'jtag'$
$Info : Initializing remote_bitbang driver$
$Info : Connecting to localhost : 9823$
$Info : remote_bitbang driver initialized$
$Info : This adapter doesn't support configurable speed$
$Info : JTAG tap : riscv.cputap/device found : 0x00000001 (mfg : 0x000 (<$
$invalid >),$
$part : 0x0000, ver : 0x0)$
$Info : data count = 2 progbuf size = 16$
$Info : Disabling abstract command reads from CSRs.$
$Info : Disabling abstract command writes to CSRs.$
$Info : [0] Found 1 triggers$
$Info : Examined RISC-V core; found 1 harts$
$Info : hart 0 : XLEN = 64, 1 triggers$
$Info : Listening on port 3333 for gdb connections$
$Info : Listening on port 6666 for tcl connections$
$Info : Listening on port 4444 for telnet connections$
$A-d flag can be added to the command to show further debug information.$

6.22 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$_6$4 − $pc - linux - gnu - -target =$
$riscv64 - unknown - elf$".
$Type"showconfiguration"forconfigurationdetails.$
$Forbugreportinginstructions, pleasesee :< http : //www.gnu.org/software/gdb/bugs/ >$
.
$FindtheGDBmanualandotherdocumentationresourcesonlineat :< http : //www.gnu.org/softwa$
.
$Forhelp, type"help".Type"aproposword"tosearchforcommandsrelatedto"word"...Readingsymbol$
$ComparedtoSpike, theCEmulatorisveryslow, soseveralproblemsmaybeencounteredduetotimeouts$
$Afterthatweloadourprogrambyperformingaloadcommand.Thisautomaticallysetsthe$PC
to the $_s tartsymbolinour.elf file.$
$(gdb)setremotetimeout2000$
$(gdb)targetremotelocalhost : 3333$
$Remotedebuggingusinglocalhost : 3333$
$0x0000000000010050in??()$
$(gdb)load$
$Loadingsection.text.init, size0x2cclma0x80000000$
$Loadingsection.tohost, size0x48lma0x80001000$
$Loadingsection.text, size0x98clma0x80001048$
$Loadingsection.rodata, size0x158lma0x800019d4$
$Loadingsection.rodata.str1.8, size0x20lma0x80001b30$
$Loadingsection.data, size0x22lma0x80001b50$
$Loadingsection.sdata, size0x4lma0x80001b74$
$Startaddress0x80000000, loadsize3646$
$Transferrate : 40bytes/sec, 520bytes/write.$
(gdb)
$NowwecanproceedaswithSpike, debuggingworksinasimilarway :$

$(gdb) printwait$
$1 = 1$
$(gdb) printwait = 0$
$2 = 0$
$(gdb) printtext$
$3 = "Vafgehpgvbafrgfjnaggborserr!"$
$(gdb) c$
$Continuing.$
^{C}
$Programreceivedsignal SIGINT, Interrupt.$
$main(argc = 0, argv =< optimizedout >) at src/main.c : 33$
$33 while(!wait)$
$(gdb) printwait$
$4 = 0$
$(gdb) printtext$
5 = "Instruction sets want to be free!" (gdb)
Further information about GDB debugging is available here and here. Building Rocket Chip with an IDE
The Rocket Chip Scala build uses the standard Scala build tool SBT. IDEs like IntelliJ and VSCode 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 $PWD variable inside of the file to an absolute path pointing to the location of your Rocket Chip clone. You$
$sed - i"s|\$PWD|$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 $PWD variable inside of it, you can import Rocket C$

THANK YOU