
VexiiRiscv Documentation

VexiiRiscv contributors

Dec 27, 2024

CONTENTS

1	Introduction	3
2	How to use	7
3	Self Contained Tutorial	13
4	Ready made Docker environment	29
5	Framework	41
6	Fetch	49
7	Decode	53
8	Execute	59
9	Branch	71
10	Memory (LSU)	75
11	Privileges	83
12	Debug support	87
13	Performance / Area / FMax	89
14	SoC	93

Welcome to VexiiRiscv's documentation!

**CHAPTER
ONE**

INTRODUCTION

In a few words, VexiiRiscv :

- Is an project which implement an hardware CPU as well as a few SoC
- Follows the RISC-V instruction set
- Is free / open-source
- Should fit well on all FPGA families but also be portable to ASIC

1.1 Other doc / media / talks

Here is a list of links to resources which present or document VexiiRiscv :

- FSiC 2024 : https://wiki.f-si.org/index.php?title=Moving_toward_VexiiRiscv
- COSCUP 2024 : <https://coscup.org/2024/en/session/PVAHAS>
- ORConf 2024 : <https://fossi-foundation.org/orconf/2024#vexiiriscv--a-debian-demonstration>

1.2 Technicalities

VexiiRiscv is a from scratch second iteration of VexRiscv, with the following goals :

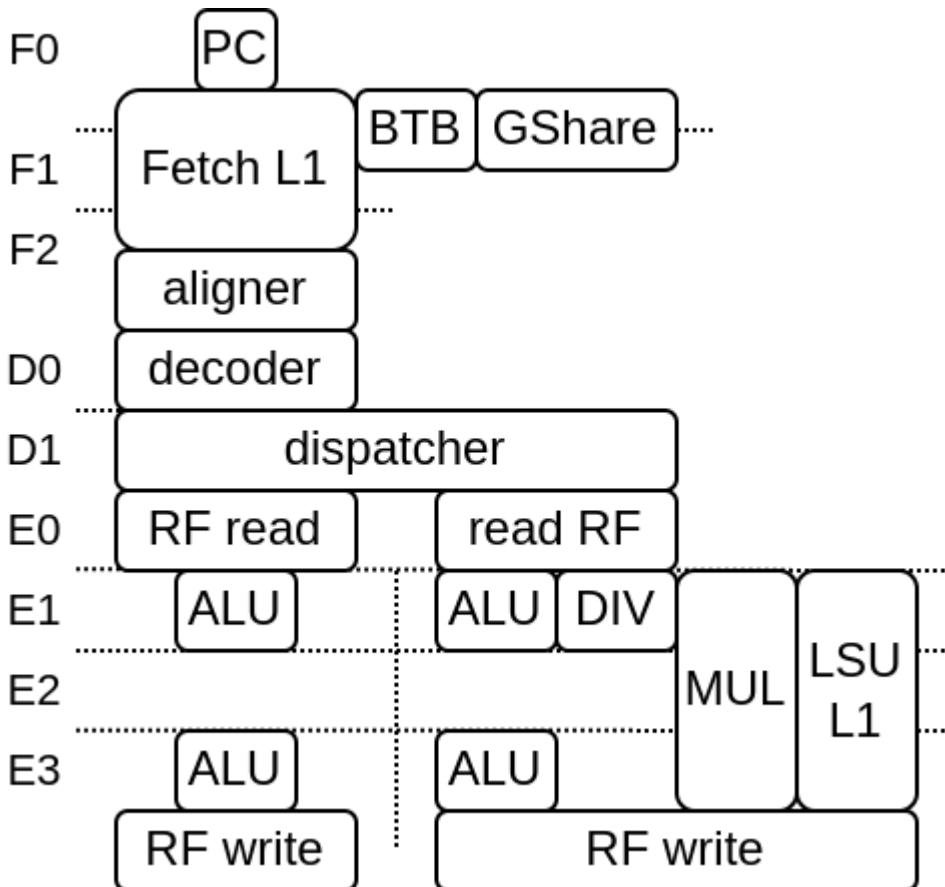
- To implement RISC-V 32/64 bits IMAFDCSU
- Could start around as small as VexRiscv, but could scale further in performance
- Optional late-alu
- Optional multi issue
- Providing a cleaner implementation, getting ride of the technical debt, especially the frontend
- Scale well at higher frequencies via its hardware prefetching and non blocking write-back D\$
- Proper branch prediction
- ...

On this date (09/08/2024) the status is :

- RISC-V 32/64 IMAFDCSU supported (Multiply / Atomic / Float / Double / Supervisor / User)
- Can run baremetal applications (2.50 dhystone/MHz, 5.24 coremark/MHz)
- Can run linux/buildroot/debian on FPGA hardware (via litex)
- single/dual issue supported
- late-alu supported
- BTB/RAS/GShare branch prediction supported

- MMU SV32/SV39 supported
- LSU store buffer supported
- Non-blocking I\$ D\$ supported
- Hardware/Software D\$ prefetch supported
- Hardware I\$ prefetch supported

Here is a diagram with 2 issue / early+late alu / 6 stages configuration (note that the pipeline structure can vary a lot):



1.3 Navigating the code

VexiiRiscv isn't implemented in Verilog nor VHDL. Instead it is written in scala and use the SpinalHDL API to generate hardware. This allows to leverage an advanced elaboration time paradigm in order to generate hardware in a very flexible manner.

Here are a few key / typical code examples :

- Integer ALU plugin ; src/main/scala/vexiiriscv/execute/IntAluPlugin.scala
- A cpu configuration generator : dev/src/main/scala/vexiiriscv/Param.scala
- The CPU toplevel src/main/scala/vexiiriscv/VexiiRiscv.scala
- Some globally shared definitions : src/main/scala/vexiiriscv/Global.scala

Also due to the nested structure of the code base, a text editor / IDE which support curly brace folding can be very usefull.

1.4 About RISC-V

To help onboarding, here is a few thing about RISC-V :

- RISC-V isn't a CPU / CPU architecture
- RISC-V is a Instruction Set Architecture (ISA), which mean that from a CPU perspective, it mostly specify the instructions that need to be implemented, and their behaviour.

RISC-V has 4 main specification :

- *Unprivileged Specification* : Mainly specify the integer, floating point and load / store instructions
- *Privileged Specification* : Mainly specify all the special CPU registers which can be used to handle interruptions, exceptions, traps, virtual memory, memory protections, machine/supervisor/user privilege modes
- *RISC-V calling convention* : Mainly specify how the registers can be used by functions to pass parameters, aswell as providing an alternative name for each of the registers (ex : x2 become the stack pointer, named sp)
- *RISC-V External Debug Support* : Mainly specify how the CPU can support JTAG debug, hardware breakpoints and triggers

To figure out more about those specification, check <https://riscv.org/technical/specifications/>

1.5 About VexRiscv (not VexiiRiscv)

There is few reasons why VexiiRiscv exists instead of doing incremental upgrade on VexRiscv

- Mostly, all the VexRiscv parts could be subject for upgrades
- VexRiscv frontend / branch prediction is quite messy
- The whole VexRiscv pipeline would have need a complete overhaul in oder to support multiple issue / late-alu
- The VexRiscv plugin system has hits some limits
- VexRiscv accumulated quite a bit of technical debt over time (2017)
- The VexRiscv data cache being write though start to create issues the faster the frequency goes (DRAM can't follow)
- The VexRiscv verification infrastructure based on its own golden model isn't great.

So, enough is enough, it was time to start fresh :D

1.6 Check list

Here is a list of important design assumptions and things to know about :

- trap/flush/pc request from the pipeline, once asserted one cycle can not be undone. This also mean that while a given instruction is stuck somewhere, if that instruction did raised on of those request, nothing should change the execution path. For instance, a sudden cache line refill completion should not lift the request from the LSU asking a redo (due to cache refill hazard).
- In the execute pipeline, stage.up(RS1/RS2) is the value to be used, (not stage.down(RS1/RS2) as it implement the bypassing for the next stage)
- Fetch.ctrl(0) isn't persistent (meaning the PC requested can change at any time)

HOW TO USE

For getting started you have two options.

Either you compile it from scratch or you use our Docker container which provides all the dependencies readily installed.

2.1 Environment (Dependencies)

You will need :

- A java JDK
- SBT (Scala build tool)
- Verilator (optional, for simulations)
- RVLS / Spike dependencies (optional, if you want to have lock-step simulations checking)
- GCC for RISC-V (optional, if you want to compile some code)

2.1.1 Docker Container

Probably the easiest way to get started:

Simply run

```
./run_docker.sh
```

Refer to the chapter about the ready made Docker container, in order to get a step by step guide on how to get started with the XFCE4 desktop and the tools provided in the installation.

2.1.2 Setup dependencies

Setting the tools up locally on your machine is a bit more work than just starting a Docker container, but speeds up things a lot because there's no virtual environment anymore.

For Windows Users In order to build RISC-V 64 toolchain you require GCC, so you will have to download Cygwin: <https://www.cygwin.com>

You should be able to install the latest GCC, as well as GIT for cloning repositories following this article: <https://preshing.com/20141108/how-to-install-the-latest-gcc-on-windows>

For Linux and Mac users You can get the RISC-V toolchain directly from

- **Java** * On Linux, simply install the most recent openjdk-<version available>.jdk, using your package manager. * On MacOS you can either install openjdk with brew or download it from the official Oracle Java website * On Windows, you'll have to build <https://www.royanrijn.com/blog/2013/10/building-openjdk-on-windows>) Java yourself within Cygwin

- **SBT** Go to the release page of SBT (<https://github.com/sbt/sbt/releases>) Just download the tar xvf the tar file in your home directory and add \$HOME/sbt/bin to your search path (PATH).

Compiling and installing Verilator

In case the package manager of your platform provides the current Verilator as a precompiled package, you can just install Verilator with your package manager, like apt-get, zypper or brew install.

In case you're on Windows or an old Debian, you'll have to compile it from the sources yourself, however.

```
sudo apt-get install git make autoconf g++ flex bison help2man
git clone https://github.com/verilator/verilator
unsetenv VERILATOR_ROOT # For csh; ignore error if on bash
unset VERILATOR_ROOT # For bash
cd verilator
git pull      # Make sure we're up-to-date
git checkout v4.216 # You don't exactly need that version
autoconf      # Create ./configure script
./configure
make
sudo make install
```

Compiling and installing ELFIO

Some essential headers, you'll have to install yourself on every platform basically.

```
# RVLS / Spike dependencies (optional, for simulations)
sudo apt-get install device-tree-compiler libboost-all-dev
# Install ELFIO, used to load elf file in the sim
git clone https://github.com/serge1/ELFIO.git
cd ELFIO
git checkout d251da09a07dff40af0b63b8f6c8ae71d2d1938d # Avoid C++17
sudo cp -R elfio /usr/include
cd .. && rm -rf ELFIO
```

Compiling and installing the RISC-V toolchain

On MacOS and Windows, at least, you will have to compile the toolchain yourself from scratch in the terminal of your MacOS or the Cygwin shell alternatively, if you're under Windows

(The make command will automatically make sure to check init and update any submodules needed for building the toolchain, so no recursive flag is needed. I didn't forget to add it)

```
git clone https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
./configure --prefix=/opt/riscv --enable-multilib
make
make install
echo 'export PATH=/opt/riscv/bin:$PATH' >> ~/.bashrc
```

On GNU/Linux you can alternatively also download the precompiled bundle

```
# Getting a RISC-V toolchain (optional, if you want to compile RISC-V software)
version=riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ubuntu14
wget -O riscv64-unknown-elf-gcc.tar.gz riscv https://static.dev.sifive.com/dev-tools/
→$version.tar.gz
tar -xzvf riscv64-unknown-elf-gcc.tar.gz
sudo mv $version /opt/riscv
echo 'export PATH=/opt/riscv/bin:$PATH' >> ~/.bashrc
```

2.1.3 Repo setup

After installing the dependencies (see above) :

```
git clone --recursive https://github.com/SpinalHDL/VexiiRiscv.git
cd VexiiRiscv

# (optional) Compile riscv-isa-sim (spike), used as a golden model during the sim to
# check the dut behaviour (lock-step)
cd ext/riscv-isa-sim
mkdir build
cd build
../configure --prefix=$RISCV --enable-commitlog --without-boost --without-boost-asio
--without-boost-regex
make -j$(nproc)
cd ../../

# (optional) Compile RVLS, (need riscv-isa-sim (spike))
cd ext/rvls
make -j$(nproc)
cd ../../
```

2.2 Generate verilog

```
sbt "Test/runMain vexiiriscv.Generate"
```

You can get a list of the supported parameters via :

```
sbt "Test/runMain vexiiriscv.Generate --help"
--help                         prints this usage text
--xlen <value>
--decoders <value>
--lanes <value>
--relaxed-branch
--relaxed-shift
--relaxed-src
--with-mul
--with-div
--with-rva
--with-rvc
--with-supervisor
...
```

Here is a list of the important parameters :

Table 1: Generation parameters

Parameter	Description
--xlen=32/64	Specify the CPU data width (RISC-V XLEN). 32 bits by default, can be set to 64 bits
--with-rvm	Enable RISC-V mul/div instruction
--with-rvc	Enable RISC-V compressed instruction set
--with-rva	Enable atomic instruction support
--with-rvf	Enable 32 bits floating point support
--with-rvd	Enable 32/64 bits floating point support
--with-supervisor	Enable privileged supervisor, user and MMU
--allow-bypass-from=Int	Specify from which execute stage the integer result bypassing is allowed. Default disabled. For performance set it to 0
--with-btb	Enable Branch Target Buffer prediction
--with-gshare	Enable GShare conditional branch prediction. (Require the BTB to be enabled)
--with-ras	Enable Return Address Stack prediction. (Require the BTB to be enabled)
--regfile-async	The register read ports become asynchronous, shaving one stage in the pipeline, but not all FPGA support this kind of memories.
--mmu-sync-read	The MMU TLB memories will be implemented using memories with synchronous read ports. This allows to keep it small on FPGA which doesn't support asynchronous read ports
--fetch-l1	Enable the L1 instruction cache
--fetch-l1-ways=Int	Set the number of instruction cache ways (4KB per way by default)
--lsu-l1	Enable the L1 data cache
--lsu-l1-ways=Int	Set the number of data cache ways (4KB per way by default)
--with-jtag-tap	Enable the RISC-V JTAG debugging.

There is a lot more parameters which can be turned.

2.3 Run a simulation

Important: If you take a VexiiRiscv core and use it with a simulator which does x-prop (not verilator), you will need to add the following option : --with-boot-mem-init. By default this isn't enabled, as it can degrade timings and area while not being necessary for a fully functional hardware.

Here is how you can run a Verilator based simulation, note that Vexiiriscv use mostly an opt-in configuration. So, most performance related configuration are disabled by default.

```
sbt
compile
Test/runMain vexiiriscv.tester.TestBench --with-mul --with-div --load-elf ext/
  ↵NaxSoftware/baremetal/dhystone/build/rv32ima/dhystone.elf --trace-all
```

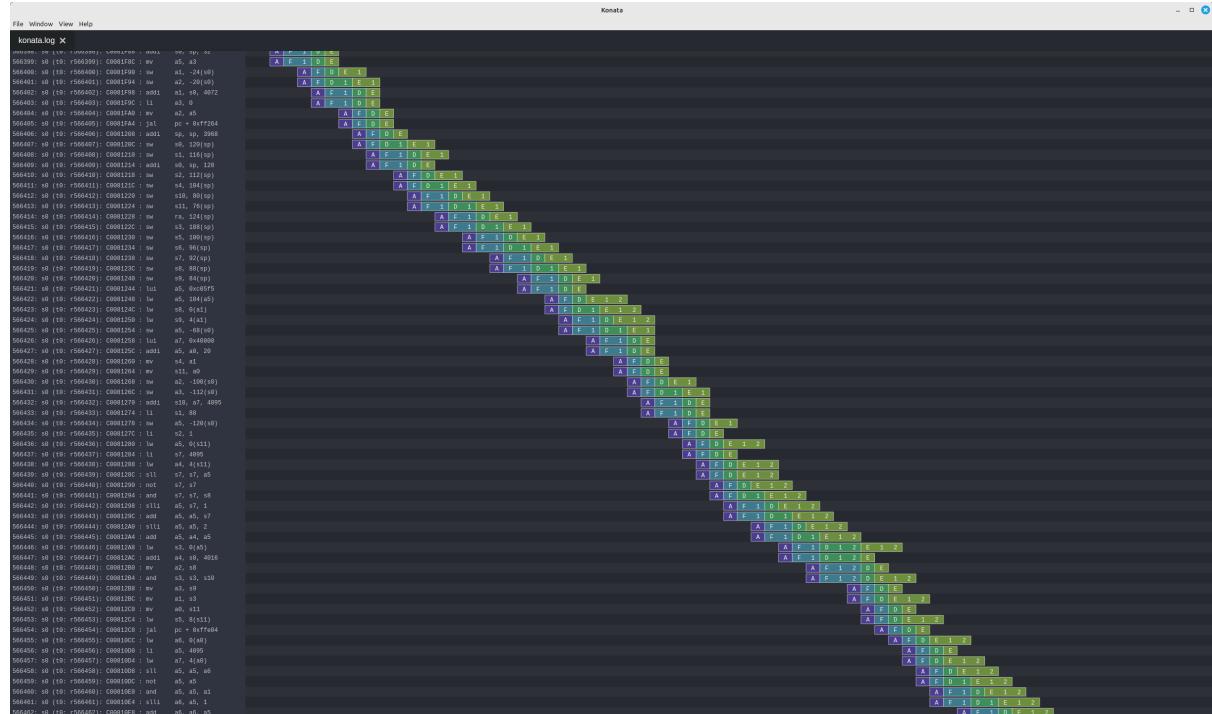
This will generate a simWorkspace/VexiiRiscv/test folder which contains :

- test.fst : A wave file which can be open with gtkwave. It shows all the CPU signals
- konata.log : A wave file which can be open with <https://github.com/shioyadan/Konata>, it shows the pipeline behavior of the CPU
- spike.log : The execution logs of Spike (golden model)
- tracer.log : The execution logs of VexRiscv (Simulation model)

Here is an example of the additional argument you can use to improve the IPC :

```
--with-btb --with-gshare --with-ras --decoders 2 --lanes 2 --with-aligner-buffer --
--with-dispatcher-buffer --with-late-alu --regfile-async --allow-bypass-from 0 --div-
--radix 4
```

Here is a screen shot of a cache-less VexiiRiscv booting linux :



2.4 Synthesis

VexiiRiscv is designed in a way which should make it easy to deploy on all FPGA, including the ones without support for asynchronous memory read (LUT ram / distributed ram / MLAB). The one exception is the MMU, but if configured to only read the memory on cycle 0 (no tag hit), then the synthesis tool should be capable of inferring that asynchronous read into a synchronous one (RAM block, work on Efinix FPGA)

By default SpinalHDL will generate memories in a Verilog/VHDL synthesisable way. Otherwise, for ASIC, you likely want to enable the automatic memory blackboxing, which will instead replace all memories defined in the design by a consistent blackbox module/component, the user having then to provide those blackbox implementation.

Currently all memories used are "simple dual port ram". While this is the best for FPGA usages, on ASIC maybe some of those could be redesigned to be single port rams instead (todo).

2.5 Other resources

There a few other ways to start using VexiiRiscv :

- Trough the MicroSoc reference design, a little microcontroller for FPGA ([MicroSoc](#))
- Through Litex, a tool to build SoC w([Litex](#))

2.6 Using IntelliJ IDEA

IntelliJ IDEA is a Java/Scala IDE which can help a lot navigating the codebase. You can get its community edition for free. Then you just need to install the scala plugin (asked the first time you run the IDE), and open the VexiiRiscv folder with it. (See the screenshots in the Ready To Use Docker guide)

2.6.1 Setup

To download IntelliJ IDEA, got to <https://www.jetbrains.com/idea/download>, select your platform, which is either Mac, Windows or Linux, and make sure to scroll all the way down to the community edition, so that you don't download the 30 days limited trial Ultimate edition instead accidentally.

We have this script for building the Docker image, which does the downloading, unpacking and installing, all by itself: https://github.com/SpinalHDL/VexiiRiscv/blob/dev/docker/setup_intellij.sh

2.6.2 Known issues

The one issue is that it has a bug, and will give you a :

```
object Info is not a member of package spinal.core
```

The workaround is that you need to run the "sbt compile" command in a terminal in the VexiiRiscv folder once.

2.7 Using Konata

Konata is a Node JS application started with Electron, so you will have to install npm with your package manager of your system.

You can setup and start Konata by cloning it and using npm

The make command will execute npm electron ., which will open the Konata window

```
git clone https://github.com/shioyadan/konata.git
cd konata
npm install
make
```

SELF CONTAINED TUTORIAL

In this tutorial you will:

- Write some assembly
- Assemble (compile) it
- Add a bug
- Run your code in a simulator
- Debug the bug
- Learn how to show important signals from the wave
- Fix the bug

3.1 Tooling

You have two options for getting start:

- You can use the Docker image with all the dependencies preinstalled
- You can use the How To Use guide on how to install all the stuff you need locally on your machine

3.2 Assembler

3.2.1 Looking at examples

In case you haven't done so, you should bring your repo up to speed and init+update all the submodules

```
cd VexiiRiscv
git pull
git submodule update --init --recursive
```

After that you can find many test programs in *ext/NaxSoftware/baremetal*, mostly written in assembly. For instance :

- simdAdd : is used to test a custom instruction which implements 4 bytes adder in a single instruction
- pmp : is used to test the RISC-V PMP, which allows the machine mode to restrict memory accesses of the supervisor/user mode to specific ranges (Physical Memory Protection)
- machine_vexii : is used to test most of the RISC-V machine mode privileged spec, as for instance, unaligned memory load exception, ...

Writing tests in assembly is often the only viable way to test low level features for a few reasons :

- It avoid all the noise which would come from C/C++ languages.

- It allows to restrict the features of the CPU being used in the tests, which is very useful for bring up.
- It allows to create very precise sequences of instructions, allowing you to trigger specific corner cases

3.2.2 Write the assembler code

So first of all, create a folder in your repository root ("work" inside the Docker environment, or otherwise simply VexiiRiscv. The folder you cloned the repository into) called "mytest"... basically

```
cd VexiiRiscv
mkdir -p mytest/src
cd mytest
```

or in Docker

```
cd /work
mkdir -p mytest/src
cd mytest
```

then create an assembler file inside the src folder called "crt.S" containing the following code

```
.option arch, +zicsr

.global _start
_start:
    li x1, 42 // Write the literal value 42 in the integer register x1
```

3.2.3 Build the assembler Code

Now, it's time to create a GNU make file, using the NaxSoftware infrastructure, so that we can turn our assembly code.

In the mytest folder, create a Makefile file containing the following

```
PROJ_NAME=mytest
STANDALONE=../ext/NaxSoftware/baremetal
include ../ext/NaxSoftware/baremetal/common/asm.mk
```

After running make in your bash shell respectively Cygwin shell (assuming you have installed everything), you should now be able to find a folder named "build", containing a bin file, and asm file and most importantly the ELF and map file.

```
leviathan@harvey:~/VexiiRiscv/mytest> ls build/
mytest.asm  mytest.bin  mytest.elf  mytest.map
```

In short, here is what those files are for :

- mytest.elf : This is the primary output of the compiler, it contains all the information about our compiled program (instruction, data, symbol locations, ..). If you need to backup a compiled program, backup this file, as all 3 others (bin/asm/map) files are generated from this elf.
- mytest.bin : Raw binary file of your program. In our case, if this binary file was directly loaded in the memory at the reset vector of the CPU (0x80000000), we would be good to go.
- mytest.asm : A text file which tells you the every instructions contained in your compiled program, as well as their location in the memory space, which is quite useful when you debug the CPU itself.
- mytest.map : Specify the memory location of every section/symbol/variable of your program. Not so useful in general, but can allow to track the access to specific memory variables from a waveform.

3.2.4 Initial run (Error)

In order to run the assembly code we just made, we have to tell sbt to load our ELF file into the VexiiRiscv simulator by running the following command in the root source folder (`/work` respectively `VexiiRiscv`)

```
cd ..
sbt "Test/runMain vexiiriscv.tester.TestBench --with-rvm --allow-bypass-from=0 --load-
→elf mytest/build/mytest.elf --trace-all"
```

Here are what the options are for :

- **--with-rvm** : Will turn on the RISC-V RVM extensions, allowing the execution of mul/div instruction.
 - **--allow-bypass-from=0** : Will enable the execute pipeline to forward results from the ALU back to new instruction before they committed, so you can execute ALU instruction back to back, even when they depend on each others.
 - **--load-elf** : This will ask the testbench to load the simulated memory with the content of the elf file before the CPU starts. Additionally, if the testbench detect that the CPU reached the pass/fail symbols of the elf file, it will end the simulation with a success/failure event.
 - **--trace-all** : This will ask the simulation to capture a whole set of simulation traces that you can find in *simWorkspace/VexiiRiscv/test*, including the simulation waveform (wave.fst), a representation of the CPU pipeline status (konata.log).

Keep in mind, by default, VexiiRiscv is configured with most of its features disabled. no branch prediction, no cache, no mmu, no ...

Now it's running

But... ooopsie. It failed.

```

error] Exception in thread "main" spinal.sim.SimFailure: Vexii hasn't committed anything for too long, last uop id 0x475
error]   at spinal.core.sim.package$.simFailure(package.scala:214)
error]   at vexiriscv.test.VexiiRiscvProbe.$anonfun$checkCommits$2(VexiiRiscvProbe.scala:553)
error]   at vexiriscv.test.VexiiRiscvProbe.$anonfun$checkCommits$2$adapted(VexiiRiscvProbe.scala:547)
error]   at scala.collection.TraversableLike$WithFilter.$anonfun$foreach$1(TraversableLike.scala:985)
error]   at scala.collection.IndexedSeqOptimized.foreach(IndexedSeqOptimized.scala:36)
error]   at scala.collection.IndexedSeqOptimized.foreach$(IndexedSeqOptimized.scala:33)
error]   at scala.collection.mutable.ArrayOps$OfRef.foreach(ArrayOps.scala:198)
error]   at scala.collection.TraversableLike$WithFilter.foreach(TraversableLike.scala:984)
error]   at vexiriscv.test.VexiiRiscvProbe.checkCommits(VexiiRiscvProbe.scala:547)
error]   at vexiriscv.test.VexiiRiscvProbe.$anonfun$new$2(VexiiRiscvProbe.scala:630)
error]   at spinal.core.sim.package$SimClockDomainPiper.$anonfun$onSamplings$2(package.scala:1111)
error]   at spinal.core.sim.package$SimClockDomainPiper.$anonfun$onSamplings$2$adapted(package.scala:1111)
error]   at scala.collection.mutable.ResizableArray.foreach(ResizableArray.scala:62)
error]   at scala.collection.mutable.ResizableArray.foreach$(ResizableArray.scala:55)
error]   at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:49)
error]   at spinal.core.sim.package$SimClockDomainPiper.$anonfun$onSamplings$1(package.scala:1111)
error]   at spinal.core.sim.package$$anon$1.update(package.scala:248)
error]   at spinal.sim.SimManager.runWhile(SimManager.scala:340)
error]   at spinal.sim.SimManager.runAll(SimManager.scala:262)
error]   at spinal.core.sim.SimCompiled.doSimApi(SimBootstraps.scala:614)
error]   at spinal.core.sim.SimCompiled.dosimUntilVoid(SimBootstraps.scala:587)
error]   at vexiriscv.tester.TestOptions.test(TestBench.scala:181)
error]   at vexiriscv.tester.TestBench$.doIt(TestBench.scala:686)
error]   at vexiriscv.tester.TestBench$.delayedEndpoint$vexiriscv$tester$TestBench$1(TestBench.scala:612)
error]   at vexiriscv.tester.TestBench$delayedInit$body.apply(TestBench.scala:611)
error]   at scala.Function0.apply$mcV$sp(Function0.scala:39)
error]   at scala.Function0.apply$mcV$sp$1(Function0.scala:39)
error]   at scala.runtime.AbstractFunction0.apply$mcV$sp(AbstractFunction0.scala:17)
error]   at scala.App.$anonfun$main$1$adapted(App.scala:80)
error]   at scala.collection.immutable.List.foreach(List.scala:431)
error]   at scala.App.main(App.scala:80)
error]   at scala.App.main$(App.scala:78)
error]   at vexiriscv.tester.TestBench$.main(TestBench.scala:611)
error]   at vexiriscv.tester.TestBench.main(TestBench.scala)
error] Nonzero exit code returned from runner: 1
error] (Test / runMain) Nonzero exit code returned from runner: 1
error] Total time: 7 s, completed Dec 5, 2024, 2:26:00 PM

```

Question: Why??

Answer The CPU is locked into a *illegal instruction exception* loop of doom.

Here is the full scénario :

- Once the CPU had executed *li x1, 42*, it then reach a portion of memory which isn't loaded with code but instead has a random value (the testbench is designed that way).
- So it is very likely that the CPU try to execute a portion of memory which isn't recognized as an instruction, which produce a *illegal instruction exception*.
- This results into the CPU jumping to its trap vector (mtvec).
- This trap vector being initialized by the CPU reset to 0, will make the CPU jump/trap to PC=0
- At PC=0 there is as well some random values, which likely will produce another *illegal instruction exception*, again and again, forever.
- Then, the testbench detect that the CPU isn't doing any *commit* anymore (forward progress) and call it a failure.

3.2.5 Fixing the Error

We can fix this error quickly by adding those two additional lines to our assembler file

```

pass:
j pass

```

Which results in the following code

```

.option arch, +zicsr

.global _start
_start:
    li x1, 42 // Write the value 42 in the register x1

```

(continues on next page)

(continued from previous page)

```
pass:
    j pass
```

After that we run the make/sbt command again.

Now the simulation won't fail anymore, and exit gracefully, as the testbench will detect that the CPU reached the *pass* symbol.

However, an endless loop which doesn't anything isn't very useful.

Note, running SBT every time with *sbt "Test/runMain vexiiriscv.tester.TestBench ..."* is slow and painful. What you can do instead is just to run the *sbt* command without arguments, which will bring you in the SBT shell, and there you can run your *Test/runMain vexiiriscv.tester.TestBench ...* with much less overhead.

3.2.6 The assembler "hello world"

Since we can't really print out a "hello world" in this context because we're simulating a CPU and the execution of assembler code on it, we go for the next best thing: a "for" loop

```
uint32_t sum = 0;

for(int i = 0; i < 5; i++) {
    sum = sum + i;
}
```

As RISC-V assembly this looks like that:

```
.option arch, +zicsr

.global _start
_start:

    li a0, 0 # Initialize sum
    li t0, 0 # counter start value
    li t1, 5 # counter end value

sum_loop:
    bge t0, t1, pass # i == 5
    add a0, a0, t0
    addi t0, t0, 1
    j sum_loop

pass:
    j pass
```

Also, note that if you are interested into more C to assembly comparison, you can use the Compiler Explorer tool. Here is an example :

[3.2. Assembler](https://godbolt.org/#g:!((g:!((h:codeEditor,i:(filename:'1',fontScale:14,fontUsePx:'0',j:1,lang:__c,selection:(endColumn:2,endLineNumber:7,positionColumn:2,positionLineNumber:7,selectionStartColumn:2,selectionStartLineNumber:7,startColumn:2,startLineNumber:7),source:'int+miaou()%7B%0A++++int+count+%3D+1000%3B%0A++++while(count+!!%3D+0)%7B%0A++++++asm(%22nop%22)%3B%0A++++++count--%3B%0A++++%7D%0A%7D'),l:'5',n:'0',o:'C+source+%231',t:'0')),k:44.29215489283432,l:'4',n:'0',o:'',s:0,t:'0'),(g:!((h:compiler,i:(compiler:rv32-cgcctrunk,filters:(b:'0',binary:'1',binaryObject:'0',commentOnly:'0',debugCalls:'1',demangle:'0',directives:'0',execute:'1',intel:'0',libraryCode:'0',trim:'1',verboseDemangling:'0'),flagsViewOpen:'1',fontScale:14,fontUsePx:'0',j:2,lang:__c,libs:(),options:'-O3',overrides:(),selection:(endColumn:5,endLineNumber:10,positionColumn:5,positionLineNumber:10,selectionStartColumn:5,selectionStartLineNumber:10,startColumn:5,startLineNumber:10),source:1),l:</p>
</div>
<div data-bbox=)

```
'5',n:'0',o:'+RISC-V+(32-bits)+gcc+(trunk)+(Editor+%231)',t:'0')),k:55.707845107165674,l:'4',n:'0',o:'',s:0,t:'0')),l:'2',n:'0',o:'',t:'0')),version:4
```

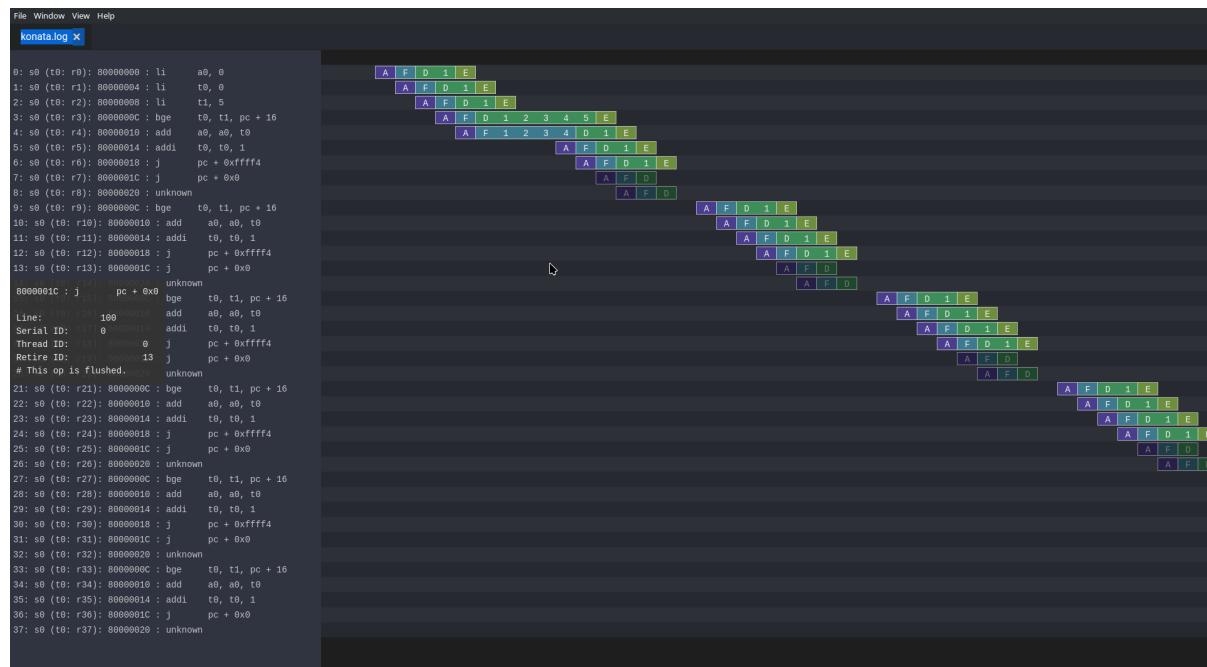
Note, you can see that this assembly example use register names as a0, t1, while the previous example was using x1. RISC-V has two why of naming the registers :

- Via their *raw name* : x0, x1, x2, ..., x31
- Via their *ABI Mnemonic* : zero, ra, sp, gp, tp, t0-t6, s0-s11, a0-a7

All of this is defined the RISC-V ABI register conventions (<https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc#register-convention>), and GCC supports both. So, in general, if you write low level assembly tests, you can go for the *raw name*, else just go with the *ABI Mnemonic* names.

3.2.7 Looking at the pipeline

Opening the pipeline trace (located in simWorkspace/VexiiRiscv/test/konata.log) using Konata , we can see that it goes five times through the loop.



Here are a few explanations how to read those Konata traces :

- Horizontally, you have the time axes
- Vertically, you have every instruction that reached the CPU decode stage (and further).
- If on the left margin, you see some "???", it mean that you need to compile the ext/riscv-isa-sim and ext/rvls. See ext/rvls/README.md
- The reset vector of VexiiRiscv being by default 0x80000000, you can see on the top left it is where it starts.
- the A/F/D/I/E symbols represent when a given instruction is in which part of the FPU
- A : Address generation of the instruction PC
- F : Fetch, when the CPU is reading the instruction from the memory (or its cache)
- D : Decode/dispatch, when the CPU is figuring out what the instruction is about, wait until the time is right to schedule the instruction to the execute pipeline, and read the register file
- E : Execute, when the instruction is being processed.
- Instructions in vivid colors are the one which successfully executed (committed instructions)

- Instructions in dark colors are the one which failed to execute (ex : flushed by an un-predicted/miss-predicted branch/jump)

There you go. Our $i < 5$ condition was successfully executed

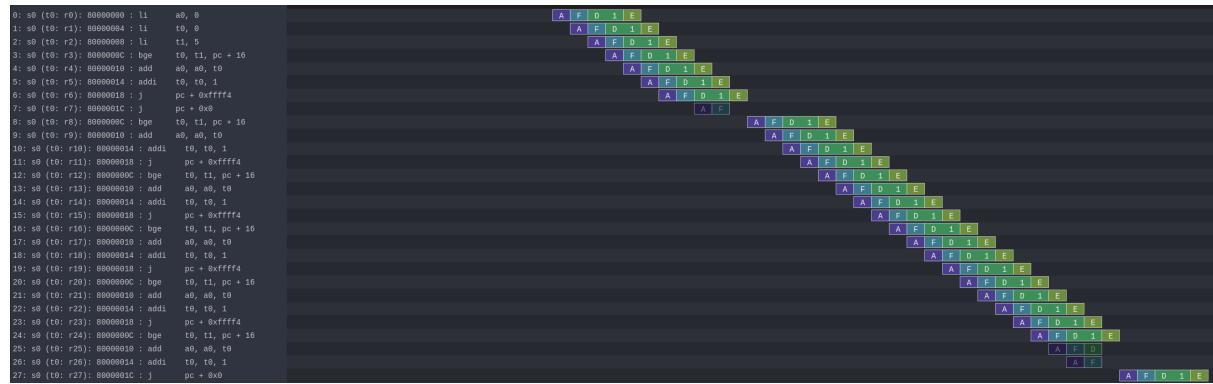
3.2.8 Enabling branch prediction

By default, the VexiiRiscv branch prediction feature is disabled. You can turn on a partial version of it on by adding `--with-btb` argument to your simulation command.

This will enable the Branch Target Buffer (BTB), which allow VexiiRiscv to predict a few things very early in the fetch pipeline, things as :

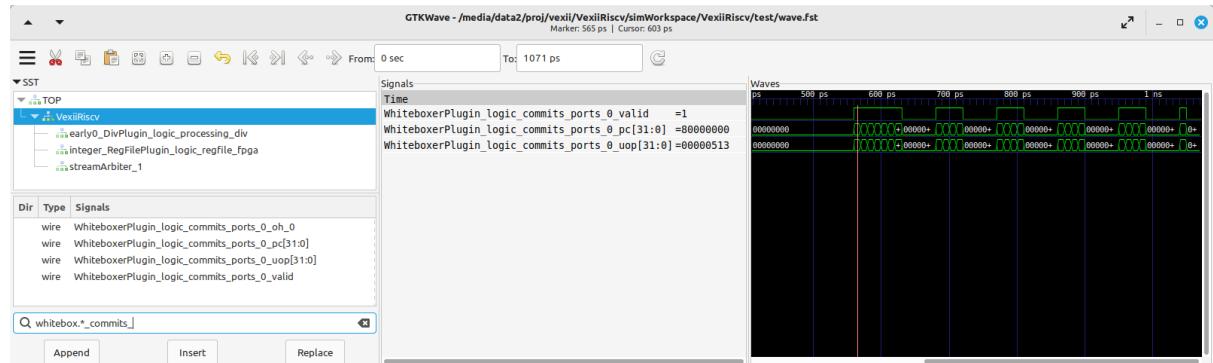
- For a given PC, is the instruction a jump/branch ?
- If it is, what would be its target PC ?
- If it is a branch, is it likely to be taken ?

You can observe the effects of the branch prediction easily via the Konata trace :



3.2.9 Looking at the waveform

Opening the simulation waveform (located in `simWorkspace/VexiiRiscv/test/wave.fst`) using gtkwave you can visualize every signals of the simulated CPU across the whole simulation.



So here the difficulty is to know what to look at in this ocean of wires. Here is a few tips about that.

- The WhiteboxerPlugin collects many key events from the CPU for debug purposes, in particular its `whiteboxerPlugin_logic_commits` signals will tell you when the CPU commits an instruction.
- `DispatchPlugin_logic_candidates` signals will tell you every instruction currently waiting to be dispatched to the execution pipeline, as well as their context.
- There are a few pipeline signals as : `fetch_logic_ctrl`, `decode_ctrl` `execute_ctrl`. Note that to know if there is an actual transaction in a given pipeline varies between the pipelines. For the fetch, you can probe `fetch.*ctrl.*_valid`, for decode it is `decode.*LANE_SEL_.$`, for execute it is `execute.*LANE_SEL_lane.$`.

3.2.10 Introducing a bug

Let's say you want to change the way the integer ALU is implemented, the easiest way to do so would be to modify the IntAluPlugin.scala (<https://github.com/SpinalHDL/VexiiRiscv/blob/977633e2866b0ab0ffbf402b459803e2b6f8a0a/src/main/scala/vexiiriscv/execute/IntAluPlugin.scala#L72>)

Lets corrupt the XOR instruction to behave like a bitwise OR :

```
AluBitwiseCtrlEnum.XOR -> (srcp.SRC1 ^ srcp.SRC2),
//into
AluBitwiseCtrlEnum.XOR -> (srcp.SRC1 | srcp.SRC2),
```

Then lets run this assembly code in the simulation :

```
.option arch, +zicsr

.global _start
_start:
    li x1, 0x0101 // First operand
    li x2, 0x1100 // Second operand
    li x3, 0x0110 // Expected result for a xor
    xor x4, x1, x2
    bne x4, x3, fail
pass:
    j pass
fail:
    j fail
```

Then you compile the test and run it in the simulator, then, if you have ext/riscv-isa-sim and ext/rvls compiled, you should get the following testbench failure (as it should) :

```
[Progress] Start VexiiRiscv test simulation with seed 2
[Error] Simulation failed at time=600
### Stats ###
kind : miss / times      miss   taken
J/B  :    0 /     0  0.0%  0.0%
      B :    0 /     0  0.0%  0.0%
Dispatch 0 :    36 /     44 81.8%
Dispatch 1 :     7 /     44 15.9%
Candidate 0 :    36 /     44 81.8%
Candidate 1 :     7 /     44 15.9%
Dispatch halt :    0 /     44  0.0%
Execute halt :    0 /     44  0.0%
IPC          :     6 /     44 13.6%

Exception in thread "main" java.lang.Exception: INTEGER WRITE MISSMATCH DUT=11100
→REF=110
..
```

So, the interesting thing here, is that the testbench didn't failed because we reached the fail symbol, but because the testbench checks what is happening on every instruction committed by the CPU and detected some bad behavior. It does that by running RVLS as a golden reference, in a lockstep manner with the simulated VexiiRiscv. This way, as soon as any hardware bug appear in VexiiRiscv, it is automatically caught by the testbench, and report it as an error. In our case, it detected that the register file was written with 0x1110 by VexiiRiscv (Device Under Test), instead of 0x0110 by RVLS (Reference).

In other words, you don't need to check that the xor instruction is executing properly by adding assembly code (bne x4, x3, fail), just executing the instruction is enough :D. This is very very useful when you run for instance a simulation of VexiiRiscv booting linux. This take a lot of time (~20mn), and if the CPU is doing bad things, without this lock-step checking, it would be very very hard to figure out when things went bad for a few reasons :

- CPU bugs may not make the software crash instantly, or at all. Symptoms and causes can be very very far apart (in time).
- Long simulation (ex booting linux) are about 400'000'000 cycles long, you can't save all of it in a wave, as that is way too much data.

Note, if you look into simWorkspace/VexiiRiscv/test/spike.log, you can see the riscv-isa-sim logs, which gives you a better insight about what was expected :

```
core  0: 0x80000000 (0x000010b7) lui      ra, 0x1
core  0: 3 0x80000000 (0x000010b7) x 1 0x00001000
core  0: 0x80000004 (0x01008093) addi     ra, ra, 16
core  0: 3 0x80000004 (0x01008093) x 1 0x00001010
core  0: 0x80000008 (0x00001137) lui      sp, 0x1
core  0: 3 0x80000008 (0x00001137) x 2 0x00001000
core  0: 0x8000000c (0x10010113) addi     sp, sp, 256
core  0: 3 0x8000000c (0x10010113) x 2 0x00001100
core  0: 0x80000010 (0x11000193) li      gp, 272
core  0: 3 0x80000010 (0x11000193) x 3 0x00000110
core  0: 0x80000014 (0x0020c233) xor      tp, ra, sp
core  0: 3 0x80000014 (0x0020c233) x 4 0x00000110
```

3.2.11 Experimenting with privilege levels

The RISC-V privileged specification specify 3 levels in which the CPU can be to execute code :

- Machine mode : This is privilege level which can access everything. When the CPU get out of reset, it spawn in machine mode. Typically the machine mode will be used to run bootloaders, bios and baremetal applications.
- Supervisor mode : This is the privileged mode which would be used to run operating systems/kernels which want to take advantage of the RISC-V MMU.
- User mode : Operating systems/kernels will typically use the user mode to run applications. You can see user mode as a sandbox to avoid applications to harm things around.

So, the RISC-V privileged specification is very hard to read if you don't already have some good knowledge about what to expect. What this example aims at doing is to show you how you can navigate your CPU between privilege modes.

```
.option arch, +zicsr

.global _start
_start:
#define MSTATUS_MPP_SUPERVISOR      0x00000800
#define MSTATUS_MPP_USER            0x00000000
#define CAUSE_ILLEGAL_INSTRUCTION 2

    // Specify where the CPU should jump after executing the mret instruction
    la x1, supervisor_entry; csrw mepc, x1
    // Specify where the CPU should jump when it got a interruption/exception for the
    // machine mode
    la x1, supervisor_exit; csrw mtvec, x1
    // Specify that the CPU should go in supervisor mode after executing the mret
    // instruction
    li x1, MSTATUS_MPP_SUPERVISOR; csrw mstatus, x1
    // Engage the privilege transition.
    mret
```

(continues on next page)

(continued from previous page)

```
// The CPU should never reach this point
j fail

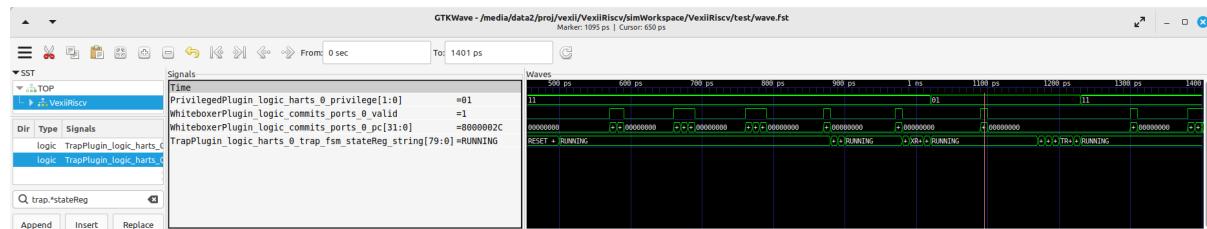
supervisor_entry:
    //Welcome in supervisor mode :D
    li x1, 666
    // Lets run a illegal instruction, we aren't allowed to access machine mode CSR
    //from supervisor mode !
    csrr x1, mepc
    // We should not be able to reach this point, as the previous instruction would
    //have produce a illegal instruction exception
    j fail

supervisor_exit:
    // Welcome back in machine mode :D
    li x1, 42
    // Lets read the CSR which indicate the reason why we back to machine mode, and
    //check it is because of CAUSE_ILLEGAL_INSTRUCTION
    csrr x1, mcause
    li x2, CAUSE_ILLEGAL_INSTRUCTION; bne x1, x2, fail
    // lets read which instruction (PC) caused it
    csrr x1, mcause

pass:
    j pass
fail:
    j fail
```

Then, compile it, but to run it in the simulation, you will need to add the `--with-supervisor`, as the VexiiRiscv only support machine mode by default.

Here is a wave with a few key signals to figure out what the CPU is doing :



Note the `TrapPlugin_logic_harts_0_trap_fsm_stateReg_string` signal, which is a special state machine in VexiiRiscv which is used to handle a few corner case, as interrupts, exceptions, replay of failed instructions, and a few other things.

Also, note that `ext/NaxSoftware/baremetal/driver/privileged.h` contains a bunch of very useful macro to do similar things.

3.2.12 Connecting with openocd to the simulation

Openocd is a tool generally used to connect your PC to a micro-controller and debug/reprogram it through a USB to JTAG dongle.

One interesting thing is that there is ways to simulate that jtag connection between openocd and the VexiiRiscv simulation by using a TCP connection. Here is how you can do it :

First, install openocd (a regular version should be fine)

Then, let a simulation run in one terminal with the following additional arguments `--no-probe --no-rvls-check --debug-privileged --debug-jtag-tap --jtag-remote`. Do not forget to remove the `--trace-all`, as it will create very big log files if you let it run long as well as slowing down the simulation.

- `--no-probe` : Will disable the testbench CPU inactivity watchdog (as we can stop the CPU activity totally using the jtag)
- `--no-rvls-check` : Will disable the RVLS golden model checking, as it isn't supported with the jtag connection yet
- `--debug-privileged` : Will enable the CPU debug interface as well as all the required special CSR (Control Status Register)
- `--debug-jtag-tap` : Will add in the CPU all the required logic to drive the CPU debug interface from a JTAG interface
- `--jtag-remote` : Will ask the testbench to implement the TCP to simulated JTAG bridge

Then you can start openocd via :

```
(cd src/main/tcl/openocd/ && openocd -f vexiiriscv_sim.tcl)
```

This should give you the following message :

```
rawrr@rawrr-pc:/media/data2/proj/vexii/VexiiRiscv$ (cd src/main/tcl/openocd/ && openocd -f vexiiriscv_sim.tcl)
Open On-Chip Debugger 0.11.0
Licensed under GNU GPL v2
For bug reports, read
      http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
Info : set servers polling period to 400ms
Info : Initializing remote_bitbang driver
Info : Connecting to localhost:44853
Info : remote_bitbang driver initialized
Info : This adapter doesn't support configurable speed
Info : JTAG tap: riscv.cpu tap/device found: 0x10002fff (mfg: 0x7ff (<invalid>), part: 0x0002, ver: 0x1)
Info : datacount=1 prodbufsize=2
Info : Disabling abstract command reads from CSRs.
Info : Examined RISC-V core; found 1 harts
Info : hart 0: XLEN=32, misa=0x40000100
Info : starting gdb server for riscv.cpu.0 on 3333
Info : Listening on port 3333 for gdb connections
Ready for Remote Connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

Meaning that the connection is successful !

You can then connect to openocd in a few ways :

- Using GDB, which would allow you to have a fully fledge debugger

- Using telnet, to ask openocd to execute basic commands.

The issue with GDB, for very low level debugging, is that it often has a lot of overhead/noise, even for simple tasks. So in general using telnet is a better first step.

Here is an example of telnet connection to openocd :

```
telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
Open On-Chip Debugger
>
```

Then you can run various commands as :

```
# Read a 32 bits word at the address 0x80000000
mdw 0x80000000

# Write a 32 bits word (0x04200513, which is a "li a0, 0x42" instruction) at the
# address 0x80000000
mww 0x80000000 0x04200513

# Move the CPU PC to the instruction we just wrote at 0x80000000
reg pc 0x80000000

# Ask the CPU to execute a single instruction
step

# Read the CPU PC, it should be 0x80000004
reg pc

# Read the CPU register a0, it should be 0x42 (just written by the instruction we
# step)
reg a0
```

There is plenty other commands available. For instance you could load the opensbi, device tree, linux, buildroot binary files in the memory from the JTAG, and boot linux, all from the JTAG ! (maybe not in simulation, it would take too long to load the images :D)

3.3 C code "hello world" (literally)

Here's a simple example how you can use C and sim_putchar for printing out stuff directly through the simulation environment, allowing you to output debug messages from within the firmware you're developing.

3.3.1 Write the C code

So first of all, create a folder in your repository root ("work" inside the Docker environment, or otherwise simply VexiiRiscv. The folder you cloned the repository into) called "mytest"... basically

```
cd VexiiRiscv
mkdir -p helloworld/src
cd helloworld
```

or in Docker

```
cd /work
mkdir -p helloworld/src
cd helloworld
```

Create a file in src, called main.c

The content of src/main.c should look like that:

```
#include <sim.h>

void main(){
    for(int i=0;i<10;i++) {
        char *str = "hello world";
        while(*str) sim_putchar(*str++);
    }
}
```

3.3.2 Compiling the Code

Now, it's time to create a GNU make file, using the NaxSoftware infrastructure, so that we can turn our C code into an ELF file which we can load in the simulator.

In the same helloworld folder as above create a Makefile file containing the following

```
PROJ_NAME=helloworld
STANDALONE=../ext/NaxSoftware/baremetal
SRCS = $(wildcard src/*.c) \
       $(wildcard src/*.cpp) \
       $(wildcard src/*.S) \
       ${STANDALONE}/common/start.S
include ../ext/NaxSoftware/baremetal/common/app.mk
```

After running make in your bash shell respectively Cygwin shell (assuming you have installed everything), you should now be able to find a folder named "build", containing a bin file, and asm file and most importantly the ELF and map file.

```
leviathan@harvey:~/VexiiRiscv/helloworld> make
CC src/main.c
CC ../ext/NaxSoftware/baremetal/common/start.S
LD helloworld
/opt/riscv/lib/gcc/riscv64-unknown-elf/13.2.0/../../../../riscv64-unknown-elf/bin/ld: warning: build/helloworld.elf has a LOAD segment with RWX permissions
Memory region           Used Size  Region Size %age Used
                  ram:      4848 B      256 KB     1.85%
leviathan@harvey:~/VexiiRiscv/helloworld> ls
build  Makefile  src
leviathan@harvey:~/VexiiRiscv/helloworld> ls build/
helloworld.asm  helloworld.bin  helloworld.elf  helloworld.map  home
```

3.3.3 Compilation error

There might be a compilation error going somewhat like that

```
leviathan@harvey:~/VexiiRiscv/helloworld> make
CC src/fix.S
CC ./ext/NaxSoftware/baremetal/common/start.S
./ext/NaxSoftware/baremetal/common/start.S: Assembler messages:
./ext/NaxSoftware/baremetal/common/start.S:55: Error: unrecognized opcode `csrc_
↳ mstatus,x1', extension `zicsr' required
./ext/NaxSoftware/baremetal/common/start.S:57: Error: unrecognized opcode `csrs_
↳ mstatus,x1', extension `zicsr' required
```

This happens because newer builds of the RISC-V toolchain have this feature disabled by default now and you've got to manually enable it, which can easily be achieved by adding the following on line 1 of ext/NaxSoftware/baremetal/common/start.S

```
.option arch, +zicsr
...
```

3.3.4 Running the code

You can now use SBT in order to run the elf file in your simulation

```
cd ..
sbt "Test/runMain vxiiriscv.tester.TestBench --with-rvm --allow-bypass-from=0 --load-
↳ elf helloworld/build/helloworld.elf --trace-all --no-probe --debug-privileged --no-
↳ rvls-check"
```

This should now print 10 times "hello world" on your terminal

```
leviathan@harvey:~/VexiiRiscv> sbt "Test/runMain vxiiriscv.tester.TestBench --with-
↳ rvm --allow-bypass-from=0 --load-elf helloworld/build/helloworld.elf --trace-all --
↳ no-probe --debug-privileged --no-rvls-check"
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.jline终端.impl.exec.
↳ ExecTerminalProvider$ReflectionRedirectPipeCreator (file:/home/leviathan/.sbt/boot/
↳ scala-2.12.19/org.scala-sbt/sbt/1.10.0/jline-terminal-3.24.1.jar) to constructor
↳ java.lang.ProcessBuilder$RedirectPipeImpl()
WARNING: Please consider reporting this to the maintainers of org.jline终端.impl.
↳ exec.ExecTerminalProvider$ReflectionRedirectPipeCreator
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective_
↳ access operations
WARNING: All illegal access operations will be denied in a future release
[info] welcome to sbt 1.10.0 (Oracle Corporation Java 11.0.25)
[info] loading settings for project vxiiriscv-build from plugins.sbt ...
[info] loading project definition from /home/leviathan/VexiiRiscv/project
[info] loading settings for project ret from build.sbt ...
[info] loading settings for project spinalhdl-build from plugin.sbt ...
[info] loading project definition from /home/leviathan/VexiiRiscv/ext/SpinalHDL/
↳ project
[info] loading settings for project all from build.sbt ...
[info] set current project to VexiiRiscv (in build file:/home/leviathan/VexiiRiscv/)
[info] running (fork) vxiiriscv.tester.TestBench --with-rvm --allow-bypass-from=0 --
↳ load-elf helloworld/build/helloworld.elf --trace-all --no-probe --debug-privileged -
↳ --no-rvls-check
[info] With Vxiiriscv parm :
```

(continues on next page)

(continued from previous page)

```
[info] - rv32im_d1At1_l1_disAt1_rfsDp_fclF0dw32_lsuP0F0dw32_bp0_rsrc_d2Area_pdbg
[info] [Runtime] SpinalHDL dev      git head : 4ea15953aa8a888e636e4ae5d7445770f2e0e73c
[info] [Runtime] JVM max memory : 1826.0MiB
[info] [Runtime] Current date : 2024.12.05 20:01:11
[info] [Progress] at 0.000 : Elaborate components
[info] [Progress] at 1.790 : Checks and transforms
[info] [Progress] at 2.290 : Generate Verilog to ./simWorkspace/tmp/job_1
[info] [Warning] toplevel/FetchCachelessPlugin_logic_buffer_words : Mem[2*33 bits].
  ↳readAsync can only be write first into Verilog
[info] [Warning] 546 signals were pruned. You can call printPruned on the backend.
  ↳report to get more informations.
[info] [Done] at 2.555
[info] [Progress] Simulation workspace in /home/leviathan/VexiiRiscv./simWorkspace/
  ↳VexiiRiscv
[info] [Progress] Verilator compilation started
[info] [info] Found cached verilator binaries
[info] [Progress] Verilator compilation done in 632.813 ms
[info] [Progress] Start VexiiRiscv test simulation with seed 2
[info] hello world
```


READY MADE DOCKER ENVIRONMENT

This self contained tutorial will show you how to pull a Docker container with all the dependencies preinstalled so that you can start right away without having to compile any of the dependencies from scratch.

Simply pull the Docker image from the Docker hub and get started.

The scope of this tutorial is:

- Fetching the Docker image
- Generating the verilog
- Running a simulation
- Opening the traces (gtkwave + konata)

Important

Starting the Docker image might take much longer, when your own user owning the folder where you cloned the repo to doesn't have the same uid as the ubuntu user inside the Docker container! The uid of the ubuntu user is 1000

4.1 Linux and MacOS X

There's a bash script called run_docker.sh which automatically pulls the most recent Docker image, starts it and then launches a VNC viewer.

Just make sure that you have Tiger VNC, bash and of course Docker installed and that the Docker demon is running.

Then you can simply run

```
./run_docker.sh
```

After the image has been fetched and the virtual X server has started you should be greeted with an XFCE4 desktop in a VNC viewer

4.2 Windows

Windows is a bit trickier, but similar as what we do on Linux and Mac

In the Power Shell, first fetch the Docker image, then start it as demon and check with the inspect command what the IP of the container is.

After that, you should be able to connect with a VNC client.

TigerVNC also exists for Windows: <https://sourceforge.net/projects/tigervnc>

```
docker.exe pull leviathanch/vexiiriscv
docker.exe run -v `pwd`:/work --privileged=true -idt leviathanch/vexiiriscv
```

After that, run the inspect command with the container ID docker returns when starting the image as a demonized process.

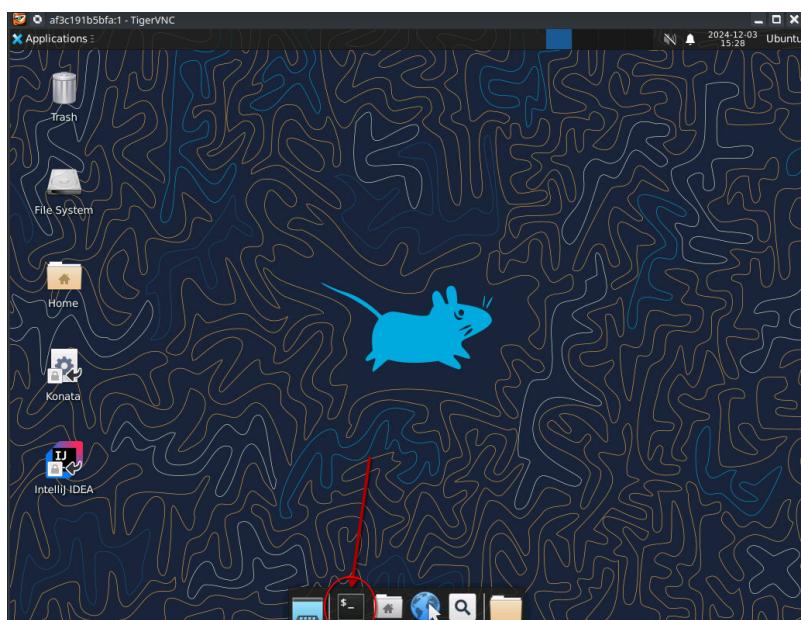
```
docker.ext inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}'  
→$container_id
```

Next run the Tiger VNC vncviewer

```
vncviewer.exe $ip
```

4.3 Generating the verilog

First open the terminal by clicking the terminal icon as shown below

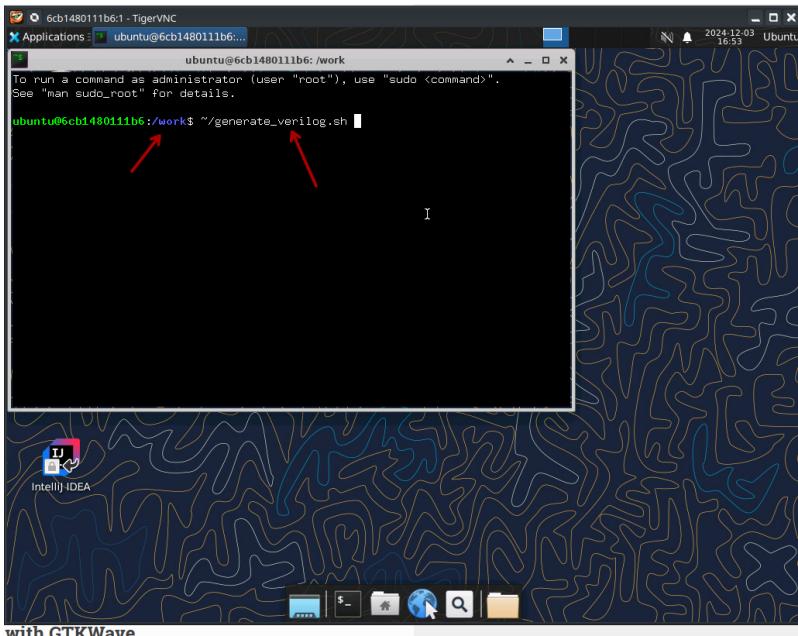


As soon as you've started the Docker container as shown above you can obtain the Verilog code by simply running the following command from within the terminal.

Make sure however that you're in the proper folder

```
~/generate_verilog.sh
```

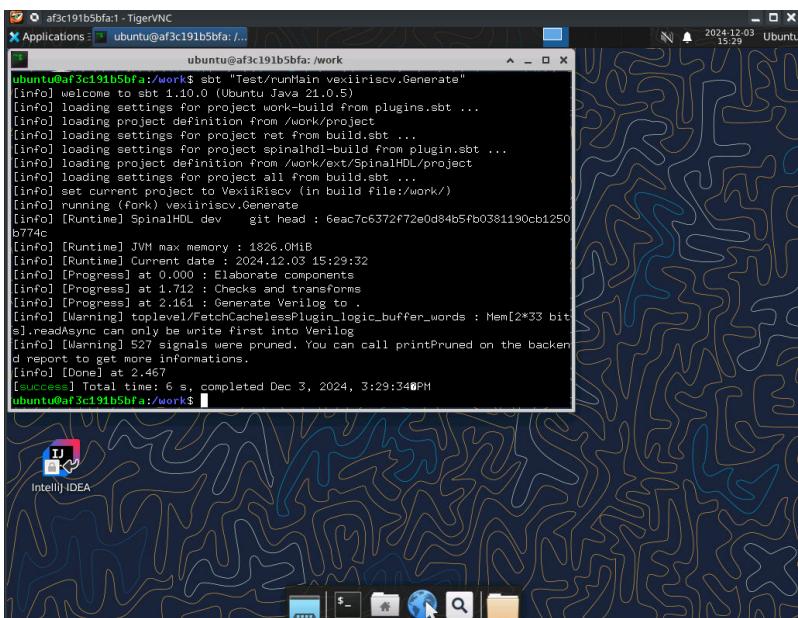
Take care that the path is correct, then press enter



This script simply contains the following command:

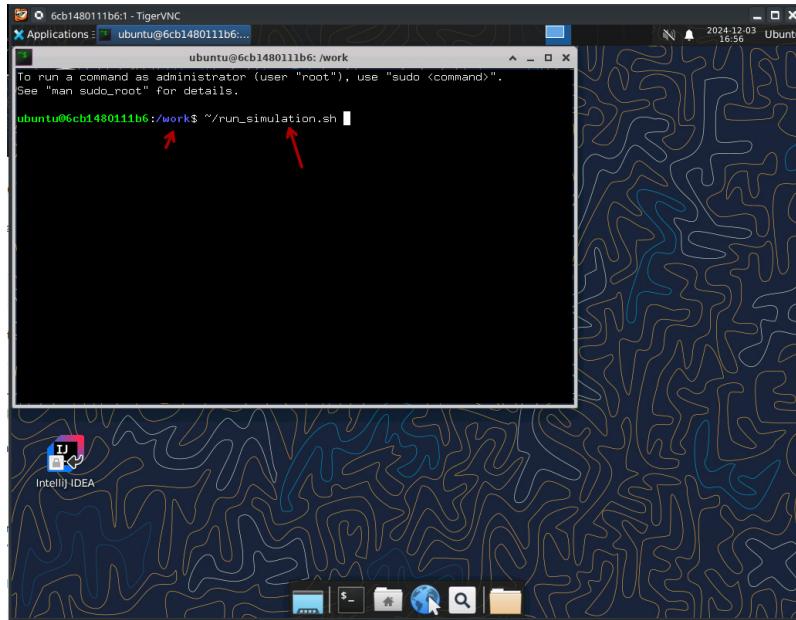
```
#!/bin/bash
sbt "Test/runMain vexiiriscv.Generate"
```

After it has been running through, you should now have a file called "VexiiRiscv.v" right there in your source folder



4.4 Running a simulation

Running a simulation also is straight forward, in the same shell as you used for generating the Verilog code.

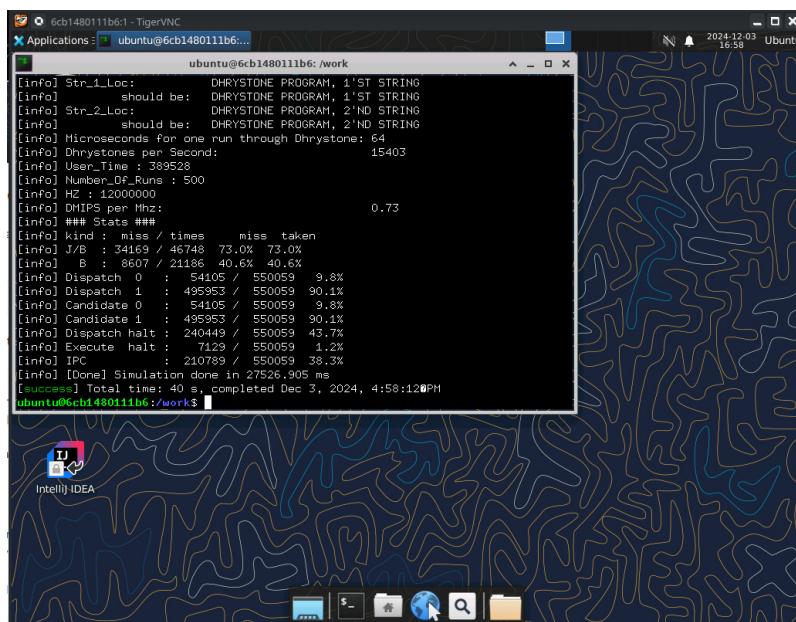


`~/run_simulation.sh`

This readily available script contains the simple command

```
#!/bin/bash
sbt "Test/runMain vexiiriscv.tester.TestBench --with-mul --with-div --load-elf ext/
→NaxSoftware/baremetal/dhystone/build/rv32ima/dhystone.elf --trace-all"
```

This will run through for a moment, and should look like this, finishing without errors



After the simulation has run through, you should now have a wave file in simWorkspace/VexiiRiscv/test/

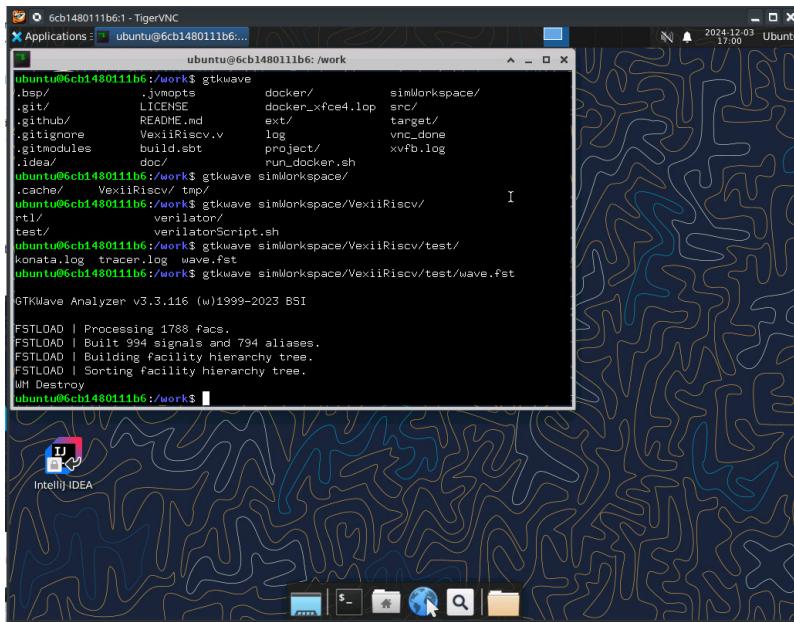
4.5 Opening the traces with GTKWave

You can convert the wave file from the simulation into the VCD format and view it by opening it with GTKWve, which is already installed in the Docker image.

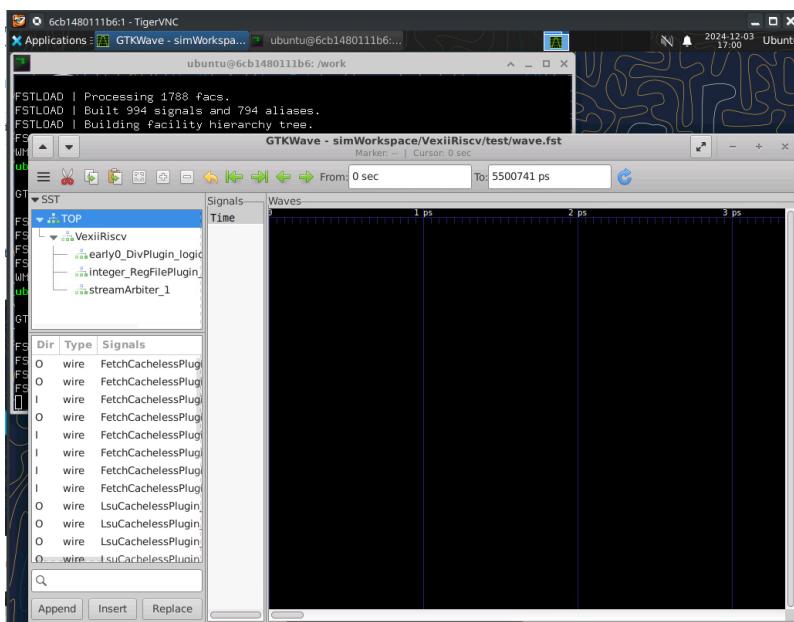
To do so, simply run in the shell

```
gtkwave simWorkspace/VexiiRiscv/test/wave.fst
```

This will start GTKWave.

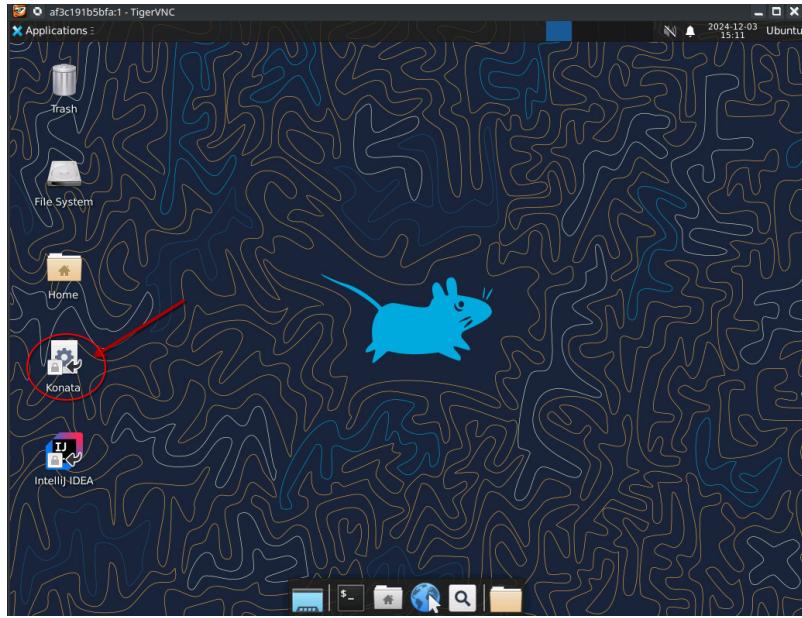


You can now select signal lines and add them to the viewer

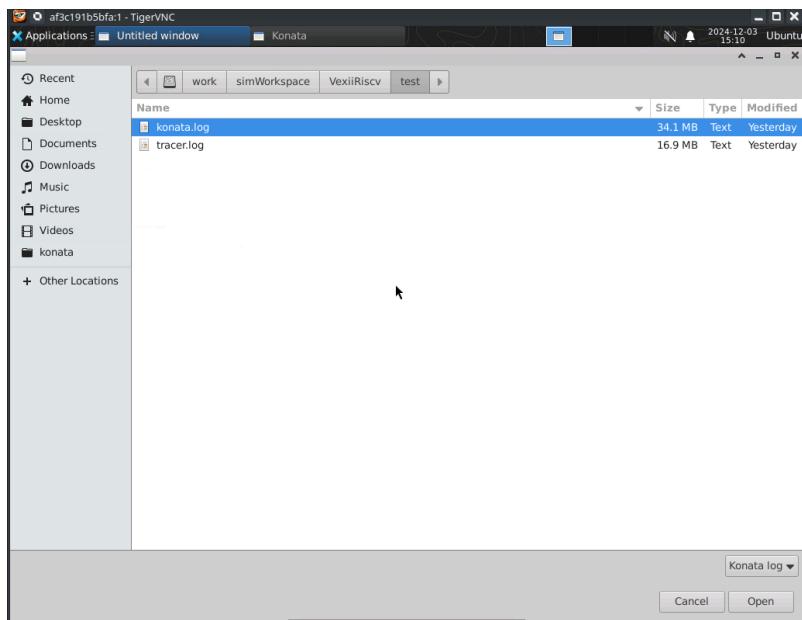


4.6 Opening the traces with Konata

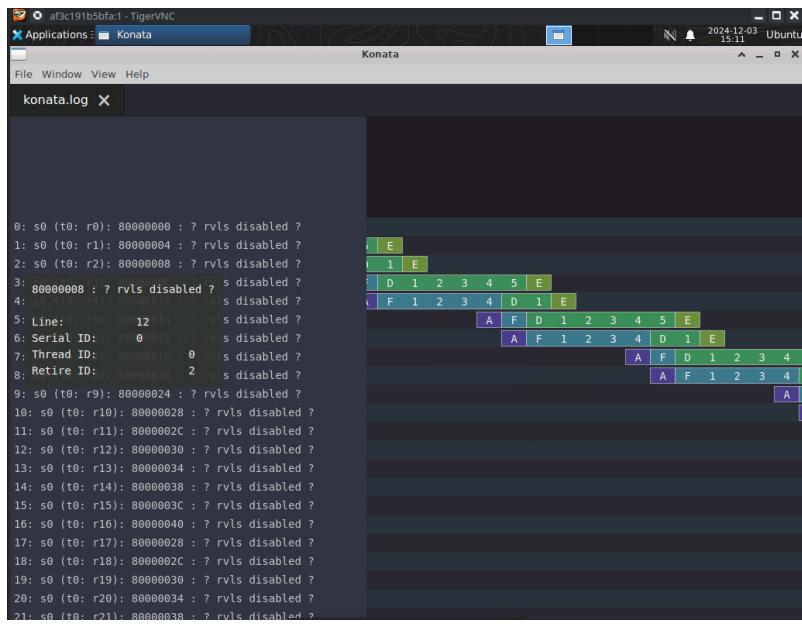
In order to visualize the instruction pipeline, you may wanna open Konata. For doing so, click on the Konata icon



Next load the konata log by going into the folder as shown in the picture



You should be greeted with a colorful representation of the instructions in the RISC-V pipeline during boot up



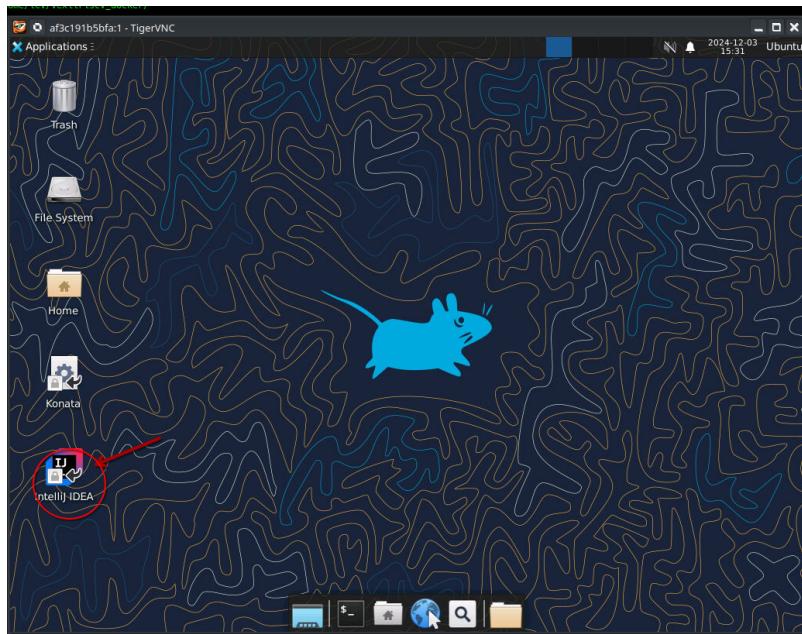
```

0: s0 (t0: r0): 80000000 : ? rvls disabled ?
1: s0 (t0: r1): 80000004 : ? rvls disabled ?
2: s0 (t0: r2): 80000008 : ? rvls disabled ?
3: 80000008 : ? rvls disabled ? s disabled ?
4: 0 (t0: r4): 80000010 : ? rvls disabled ?
5: Line: 0 (5) 80012014 : ? rvls disabled ?
6: Serial ID: 00000000000000000000000000000000
7: Thread ID: 00000000000000000000000000000000
8: Retire ID: 00000000000000000000000000000000
9: s0 (t0: r9): 80000024 : ? rvls disabled ?
10: s0 (t0: r10): 80000028 : ? rvls disabled ?
11: s0 (t0: r11): 8000002C : ? rvls disabled ?
12: s0 (t0: r12): 80000030 : ? rvls disabled ?
13: s0 (t0: r13): 80000034 : ? rvls disabled ?
14: s0 (t0: r14): 80000038 : ? rvls disabled ?
15: s0 (t0: r15): 8000003C : ? rvls disabled ?
16: s0 (t0: r16): 80000040 : ? rvls disabled ?
17: s0 (t0: r17): 80000028 : ? rvls disabled ?
18: s0 (t0: r18): 8000002C : ? rvls disabled ?
19: s0 (t0: r19): 80000030 : ? rvls disabled ?
20: s0 (t0: r20): 80000034 : ? rvls disabled ?
21: s0 (t0: r21): 80000038 : ? rvls disabled ?

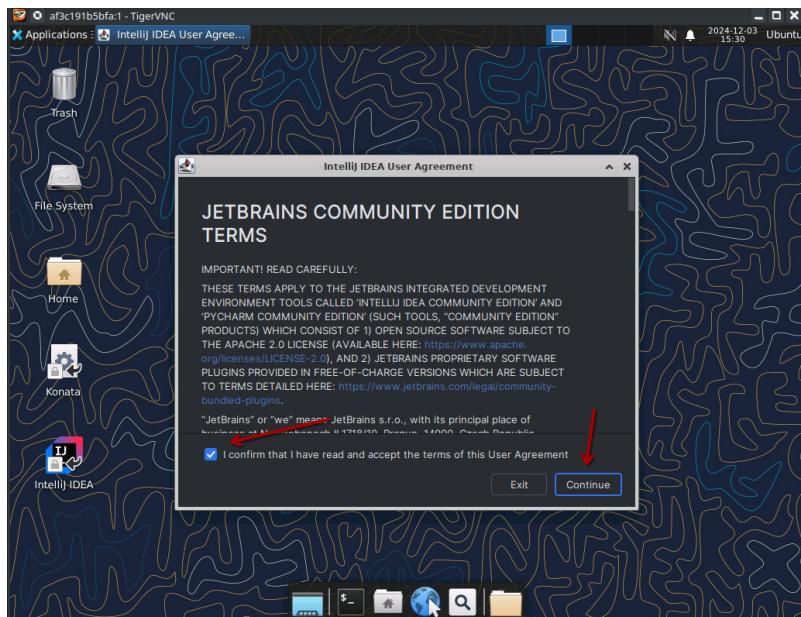
```

4.7 Opening IntelliJ IDEA

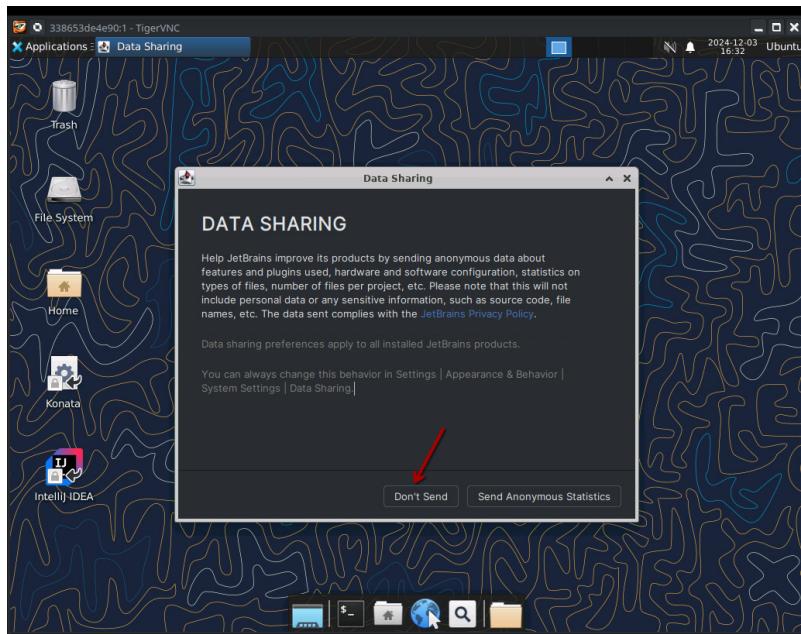
First click onto the IntelliJ IDEA icon



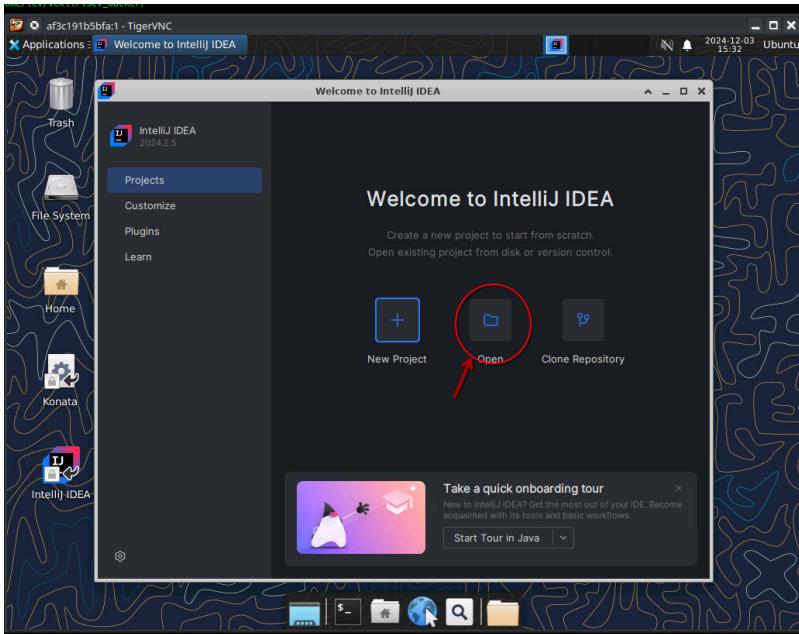
Accept the terms and conditions



We don't send data

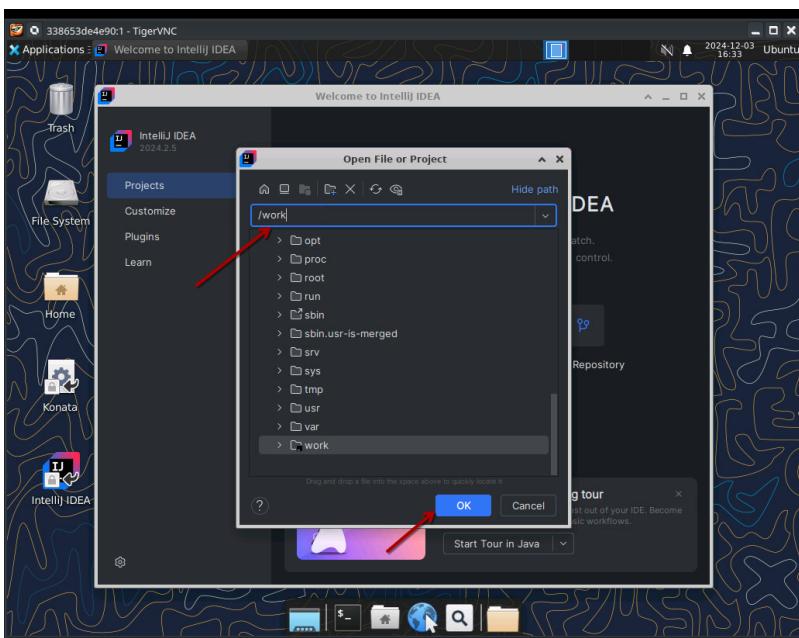


Load the VexiiRiscv project

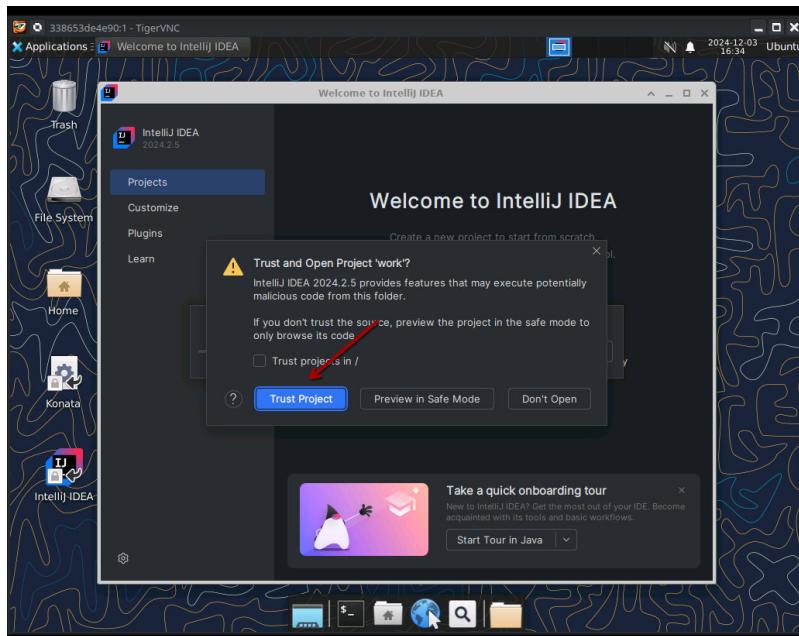


Enter the folder where your cloned repo is mounted to from outside, which is configured to be /work.

Then press OK



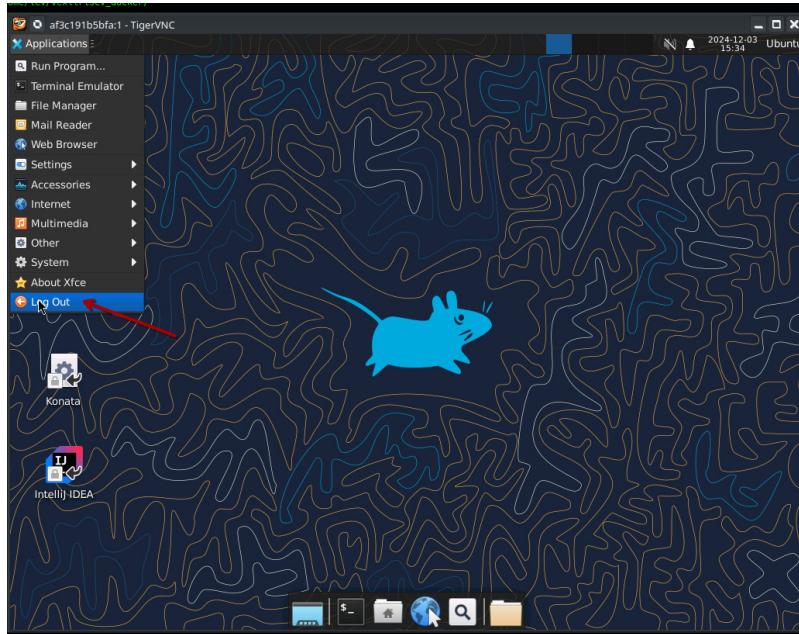
Confirm that you trust the project



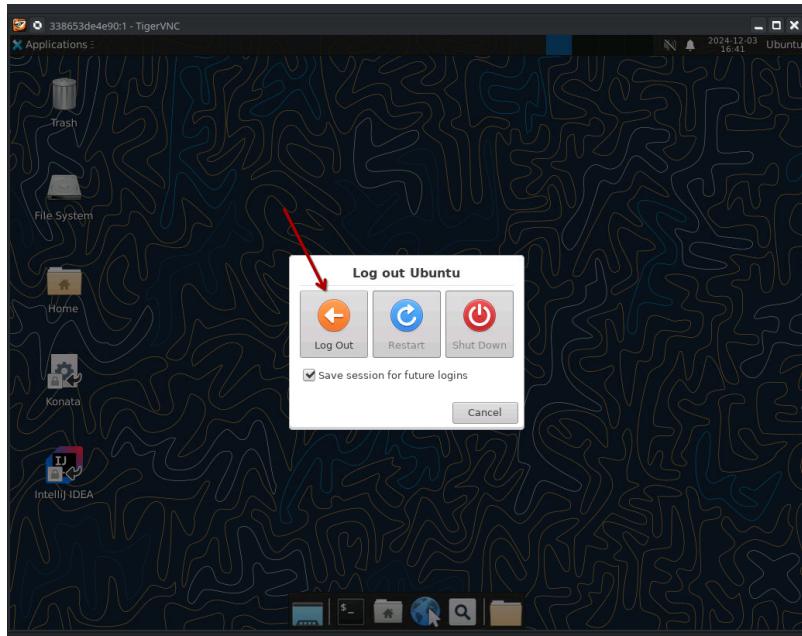
After that it will take a while until the entire project has been loaded and indexed. Make a cup of coffee or tea in the meanwhile.

4.8 Shutting down the Container

In order to shut down the container, simply logout from XFCE4 which will make the process stop and the container terminate



Confirm that you wanna log out



4.9 Using the build environment

Now that your build environment is up and running and you've got IntelliJ running as well as are familiar with the shell, you can now take your first dive into modifying the configurations and generating and testing your own modified version of the VexiiRiscv

Here are some ideas of things to try:

- How to add a custom instruction and how to test it: <https://spinalhdl.github.io/VexiiRiscv-RTD/master/VexiiRiscv/Execute/custom.html>
- How to add that custom instruction the MicroSoc: <https://spinalhdl.github.io/VexiiRiscv-RTD/master/VexiiRiscv/Soc/microsoc.html#adding-a-custom-instruction>
- How to export an APB3 bus from the MicroSoc toplevel: <https://spinalhdl.github.io/VexiiRiscv-RTD/master/VexiiRiscv/Soc/microsoc.html#exporting-an-apb3-bus-to-the-toplevel>

FRAMEWORK

5.1 Tools and API

Overall VexiiRiscv is based on a few tools and API which aim at describing hardware in more productive/flexible ways than with Verilog/VHDL.

- Scala : Which will take care of the elaboration
- SpinalHDL : Which provide a hardware description API
- Plugin : Which are used to inject hardware in the CPU. Plugins can discover each others.
- Fiber : Which allows to define elaboration threads (used in the plugins)
- Retainer : Which allows to block the execution of the elaboration threads waiting on it
- Database : Which specify a shared scope for all the plugins to share elaboration time stuff
- spinal.lib.misc.pipeline : Which allow to pipeline things in a very dynamic manner.
- spinal.lib.logic : Which provide the Quine McCluskey algorithm to generate logic decoders from the elaboration time specifications

5.2 Scala / SpinalHDL

VexiiRiscv is implemented in Scala and the SpinalHDL API to generate hardware in a explicit manner.

Scala is a general purpose programming language (like C/C++/Java/Rust/...). Statically typed, with a garbage collector. This combination allows to goes way beyond what regular HDL allows in terms of hardware elaboration time capabilities.

Here is a simple example of scala/SpinalHDL:

```
// Lets define a Counter Component/Module, with a "width" parameter
class Counter(width: Int) extends Component {
    // Lets define all its inputs/outputs in a io Bundle (Kinda similar to a
    //SystemVerilog interface)
    val io = new Bundle {
        val clear = in Bool()
        val value = out UInt(width bits)
    }

    val accumulator = Reg(UInt(width bits)) init(0) // In SpinalHDL registers/flipflop
    //are defined explicitly. Not inferred.
    accumulator := accumulator + 1 //Each cycle we increment the accumulator
    when(io.clear) {
        accumulator := 0 //But be override its value if io.clear is set (last assignment
    //win)
```

(continues on next page)

(continued from previous page)

```

}

// We connect the accumulator to the io.value.
io.value := accumulator
}

```

Here is another simple example, but which use an JtagTap tool built on the top of Scala/SpinalHDL :

```

// Lets define a component which will provide access to a few input/outputs through
// JTAG
class SimpleJtagTap extends Component {
  val io = new Bundle {
    val jtag      = slave(Jtag())
    val switches = in Bits(8 bits)
    val keys     = in Bits(4 bits)
    val leds     = out Bits(8 bits)
  }

  //The JtagTap tool allows to create the mapping between the JTAG bus and the
  //hardware
  val tap = new JtagTap(io.jtag, 8)

  //JTAG taps need an idcode, lets add it !
  val idcodeArea  = tap.idcode(B"x87654321") (instructionId=4)

  // For instance here we specify that the jtag instruction id 5 will allow it to
  //read the io.switches value
  val switchesArea = tap.read(io.switches)      (instructionId=5)

  //Lets add a few other jtag instructions to access the keys and leds hardware
  val keysArea    = tap.read(io.keys)           (instructionId=6)
  val ledsArea    = tap.write(io.leds)          (instructionId=7)
}

```

The key thing about the example above is that the JtagTap tool itself is defined in regular Scala / SpinalHDL. In other words, you can easily layer abstraction and tool on the top of the ecosystem. Use feature like Scala classes, inheritance, function overloading, collections, ..., during the hardware elaboration time.

You can find more documentation about SpinalHDL here :

- <https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html>

5.3 Plugin / Fiber / Retainer

One of the main aspect of VexiiRiscv is that all its hardware is defined inside plugins instead of a big toplevel. When you want to instantiate a VexiiRiscv CPU, you "only" need to provide a list of plugins as parameters. So, plugins can be seen as both parameters and hardware definition from a VexiiRiscv perspective.

It is quite different from the regular HDL component/module paradigm. Here are the advantages of this approach :

- The CPU can be extended without modifying its core source code, just add a new plugin in the parameters
- You can swap a specific implementation for another just by swapping plugin in the parameter list. (ex branch prediction, mul/div, ...)
- It is decentralized by nature, you don't have a endless toplevel of doom, software interface between plugins can be used to negotiate and connect things during elaboration time.

The plugins can fork elaboration threads which cover 2 phases :

- setup phase : where plugins can acquire elaboration locks on each others
- build phase : where plugins can negotiate between each others and generate hardware

5.3.1 Simple all-in-one example

Here is a simple example :

```
import spinal.core._
import spinal.lib.misc.plugin._
import vexiiriscv._
import scala.collection.mutable.ArrayBuffer

// Define a new plugin kind
class FixedOutputPlugin extends FiberPlugin{
  // Define a build phase elaboration thread
  val logic = during build new Area{
    val port = out UInt(8 bits)
    port := 42
  }
}

object Gen extends App{
  // Generate the verilog
  SpinalVerilog{
    val plugins = ArrayBuffer[FiberPlugin]()
    plugins += new FixedOutputPlugin()
    VexiiRiscv(plugins)
  }
}
```

Will generate

```
module VexiiRiscv (
  output wire [7:0]  FixedOutputPlugin_logic_port
);

  assign FixedOutputPlugin_logic_port = 8'h42;

endmodule
```

5.3.2 Negotiation example

Here is a example where there a plugin which count the number of hardware event coming from other plugins :

```
import spinal.core._
import spinal.core.fiber.Retainer
import spinal.lib.misc.plugin._
import spinal.lib.CountOne
import vexiiriscv._
import scala.collection.mutable.ArrayBuffer

class EventCounterPlugin extends FiberPlugin{
  val retainer = Retainer() // Will allow other plugins to block the elaboration of
  // "logic" thread
  val events = ArrayBuffer[Bool]() // Will allow other plugins to add event sources
```

(continues on next page)

(continued from previous page)

```

val logic = during build new Area {
    // Prevent executing this thread until the retainer is locked by other plugins
    retainer.await()

    // Now that all the other plugins are done adding event sources, we can generate
    // the actual hardware
    val counter = Reg(UInt(32 bits)) init(0)
    counter := counter + CountOne(events) // CountOne will take each bits of events,
    // add sum all them all. ex : 0b1011 => 3
}
}

// For the demo we want to be able to instantiate this plugin multiple times, so we
// add a prefix parameter to name the specific instance
class EventSourcePlugin(prefix : String) extends FiberPlugin{
    withPrefix(prefix)

    // Create a thread starting from the setup phase (this allow to run some code
    // before the build phase,
    // this allows to lock some other plugins retainers before their build phase
    val logic = during setup new Area {
        // Search for the single instance of EventCounterPlugin in the plugin pool
        val ecp = host[EventCounterPlugin]

        // Generate a lock to prevent the EventCounterPlugin elaboration (until we
        // release it).
        // This will allow us to add our localEvent to the ecp.events list
        val ecpLocker = ecp.lock()

        // Wait for the build phase before generating any hardware
        awaitBuild()

        // Here the local event is a input of the VexiiRiscv toplevel (just for the demo)
        val localEvent = in Bool()
        ecp.events += localEvent

        // As everything is done, we now allow the ecp to elaborate itself
        ecpLocker.release()
    }
}

object Gen extends App {
    SpinalVerilog {
        val plugins = ArrayBuffer[FiberPlugin]()
        plugins += new EventCounterPlugin()
        plugins += new EventSourcePlugin("lane0")
        plugins += new EventSourcePlugin("lane1")
        VexiiRiscv(plugins)
    }
}

```

```

module VexiiRiscv (
    input wire lane0_EventSourcePlugin_logic_localEvent,
    input wire lane1_EventSourcePlugin_logic_localEvent,
    input wire clk,

```

(continues on next page)

(continued from previous page)

```

    input  wire      reset
);

    wire      [31:0]  _zz_EventCounterPlugin_logic_counter;
    reg       [1:0]   _zz_EventCounterPlugin_logic_counter_1;
    wire      [1:0]   _zz_EventCounterPlugin_logic_counter_2;
    reg       [31:0]  EventCounterPlugin_logic_counter;

    assign _zz_EventCounterPlugin_logic_counter = {30'd0, _zz_EventCounterPlugin_logic_
    ↪counter_1};
    assign _zz_EventCounterPlugin_logic_counter_2 = {lane1_EventSourcePlugin_logic_
    ↪localEvent, lane0_EventSourcePlugin_logic_localEvent};
    always @(*) begin
        case(_zz_EventCounterPlugin_logic_counter_2)
            2'b00 : _zz_EventCounterPlugin_logic_counter_1 = 2'b00;
            2'b01 : _zz_EventCounterPlugin_logic_counter_1 = 2'b01;
            2'b10 : _zz_EventCounterPlugin_logic_counter_1 = 2'b01;
            default : _zz_EventCounterPlugin_logic_counter_1 = 2'b10;
        endcase
    end

    always @(posedge clk or posedge reset) begin
        if(reset) begin
            EventCounterPlugin_logic_counter <= 32'h00000000;
        end else begin
            EventCounterPlugin_logic_counter <= (EventCounterPlugin_logic_counter + _zz_
            ↪EventCounterPlugin_logic_counter);
        end
    end
end

endmodule

```

5.4 Database

In VexiiRiscv, there is the possibility to define elaboration time variable which are unique to each VexiiRiscv instance while being easily accessible as if they were global variable. For instance XLEN, PC_WIDTH, INSTRUCTION_WIDTH, ...

Those variable are handled through the VexiiRiscv "database". You can see it in the VexRiscv toplevel :

```

class VexiiRiscv extends Component{
    val database = new Database
    val host = database on (new PluginHost)
}

```

What it does is that all the plugin thread will run in the context of that database. Allowing the following patterns :

```

import spinal.core._
import spinal.lib.misc.plugin._
import spinal.lib.misc.database.Database
import vexiiriscv._
import scala.collection.mutable.ArrayBuffer

// In Scala, an object define a singleton / static thing.

```

(continues on next page)

(continued from previous page)

```

object Global extends AreaObject{
    // Lets define VIRTUAL_WIDTH as a variable in the data base.
    // VIRTUAL_WIDTH will act as the "key" to access the variable value in the current
    // context.
    // If accessed before being set, it will block the current thread execution (until
    // it is set by another thread)
    val VIRTUAL_WIDTH = Database.blocking[Int]
}

// Lets define a plugin which will use the VIRTUAL_WIDTH value.
class LoadStorePlugin extends FiberPlugin{
    val logic = during build new Area{
        val address = Reg(UInt(Global.VIRTUAL_WIDTH.get bits))
    }
}

// Lets define a plugin which will set the VIRTUAL_WIDTH value
class MmuPlugin extends FiberPlugin{
    val logic = during build new Area{
        Global.VIRTUAL_WIDTH.set(39)
    }
}

// Lets define the scala application which can generate the VexiiRiscv hardware using
// those two plugins.
object Gen extends App{
    SpinalVerilog{
        val plugins = ArrayBuffer[FiberPlugin]()
        plugins += new LoadStorePlugin()
        plugins += new MmuPlugin()
        VexiiRiscv(plugins)
    }
}

```

This will generate the following hardware :

```

module VexiiRiscv (
    input wire          clk,
    input wire          reset
);

    reg      [38:0]  LoadStorePlugin_logic_address;
endmodule

```

Keep in mind that if our toplevel had to instantiate two VexiiRiscv, each of them would have its own dedicated VIRTUAL_WIDTH.get value, while using the same VIRTUAL_WIDTH key to access it.

5.5 Pipeline API

In short, the design use a pipeline API in order to :

- Propagate data into the pipeline automatically
- Allow design space exploration with less paine (retiming, moving around the architecture)
- Handle the valid/ready arbitration
- Reduce boiler plate code

This is one of the main pillar on which VexiiRiscv is based, as it allows to define pipelines in a very distributed manner, meaning that each Plugin can very easily add and extract things on pipeline.

For instance, the plugin A can insert a given value into the pipeline at stage 1, and another plugin can ask that given value at stage 4, and that's it, it just work.

Here is an example which expose a simple usage of the pipelining API (not related to VexiiRiscv):

- Take the input at stage 0
- Sum the input at stage 1
- Square the sum at stage 2
- Provide the result at stage 3

```
import spinal.core._
import spinal.lib.misc.pipeline._

class PipelineExample extends Component{
  // Lets define a few inputs/outputs
  val a,b = in UInt(8 bits)
  val result = out(UInt(16 bits))

  // Lets create the pipelining tool.
  val pip = new StagePipeline

  // Lets insert a and b into the pipeline at stage 0
  val A = pip(0).insert(a)
  val B = pip(0).insert(b)

  // Lets insert the sum of A and B into the stage 1 of our pipeline
  val SUM = pip(1).insert(pip(1)(A) + pip(1)(B))

  // Clearly, i don't want to say pip(x)(y) on every pipelined thing.
  // So instead we can create a pip.Area(x) which will provide a scope which work in
  // stage "x"
  val onSquare = new pip.Area(2){
    val VALUE = insert(SUM * SUM)
  }

  // Lets assign our output result from stage 3
  result := pip(3)(onSquare.VALUE)

  // Now that everything is specified, we can build the pipeline
  pip.build()
}

object PipelineExampleGen extends App{
  SpinalVerilog(new PipelineExample)
}
```

This will generate the following verilog :

```

module PipelineExample (
    input  wire [7:0]    a,
    input  wire [7:0]    b,
    output wire [15:0]   result,
    input  wire          clk,
    input  wire          reset
);

    reg      [15:0]  pip_node_3_onSquare_VALUE;
    wire     [15:0]  pip_node_2_onSquare_VALUE;
    reg      [7:0]   pip_node_2_SUM;
    wire     [7:0]   pip_node_1_SUM;
    reg      [7:0]   pip_node_1_B;
    reg      [7:0]   pip_node_1_A;
    wire     [7:0]   pip_node_0_B;
    wire     [7:0]   pip_node_0_A;

    assign pip_node_0_A = a;
    assign pip_node_0_B = b;
    assign pip_node_1_SUM = (pip_node_1_A + pip_node_1_B);
    assign pip_node_2_onSquare_VALUE = (pip_node_2_SUM * pip_node_2_SUM);
    assign result = pip_node_3_onSquare_VALUE;
    always @(*posedge clk) begin
        pip_node_1_A <= pip_node_0_A;
        pip_node_1_B <= pip_node_0_B;
        pip_node_2_SUM <= pip_node_1_SUM;
        pip_node_3_onSquare_VALUE <= pip_node_2_onSquare_VALUE;
    end
endmodule

```

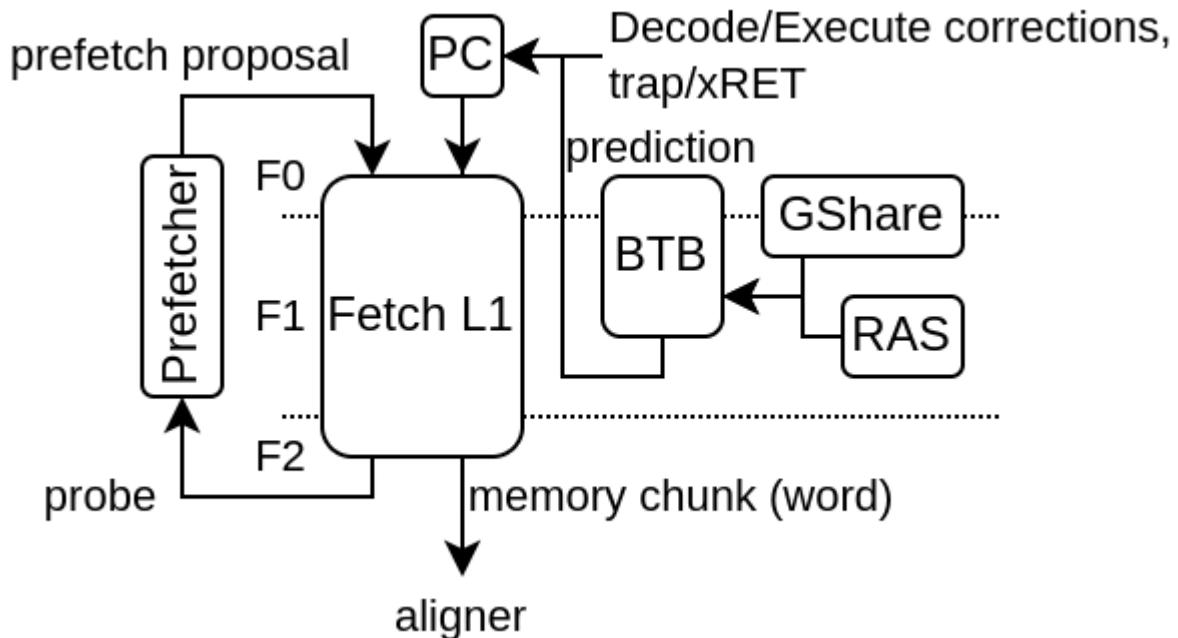
More documentation about it in :

- <https://spinalhdl.github.io/SpinalDoc-RTD/master/SpinalHDL/Libraries/Pipeline/index.html>

FETCH

The goal of the fetch pipeline is to provide the CPU with a stream of words in which the instructions to execute are present. So more precisely, the fetch pipeline doesn't really have the notion of instruction, but instead, just provide memory aligned chunks of memory block (ex 64 bits). Those chunks of memory (word) will later be handled by the "AlignerPlugin" to extract the instruction to be executed (and also handle the decompression in the case of RVC).

Here is an example of fetch architecture with an instruction cache, branch predictor as well as a prefetcher.



A few plugins operate in the fetch stage :

- FetchPipelinePlugin
- PcPlugin
- FetchCachelessPlugin
- FetchL1Plugin
- BtbPlugin
- GSharePlugin
- HistoryPlugin

6.1 FetchPipelinePlugin

Provide the pipeline framework for all the fetch related hardware. It use the native spinal.lib.misc.pipeline API without any restriction.

6.2 PcPlugin

Will :

- implement the fetch program counter register
- inject the program counter in the first fetch stage
- allow other plugin to create "jump" interface allowing to override the PC value

Jump interfaces will impact the PC value injected in the fetch stage in a combinatorial manner to reduce latency.

6.3 FetchCachelessPlugin

Will :

- Generate a fetch memory bus
- Connect that memory bus to the fetch pipeline with a response buffer
- Allow out of order memory bus responses (for maximal compatibility)
- Always generate aligned memory accesses

Note that in order to get good performance on FPGA, you may want to set it with the following config in order to relax timings :

- forkAt = 1
- joinAt = 2

6.4 FetchL1Plugin

Will :

- Implement a L1 fetch cache (non-blocking)
- Generate a fetch memory bus for the SoC interconnect
- Check for the presence of a fetch.PrefetcherPlugin to bind it to the L1

Table 1: Generation parameters

Parameter	Description
--fetch-l1	Enable the L1 D\$
--fetch-l1-ways=X	Specify the number of ways for the L1 I\$ (how many direct mapped caches in parallel), default=1
--fetch-l1-sets=X	Specify the number of sets for the L1 I\$ (how many lines of cache per way), default=64
--fetch-l1-mem-data-width-min=X	Set a lower bound for the L1 I\$ data width
--fetch-l1-hardware-prefetch=n	Enable the L1 I\$ hardware prefetcher (prefetch the next line)
--fetch-l1-refill-count=X	Specify how many cache line refill the L1 I\$ can handle at the same time, default=1

To improve the performances, consider first increasing the number of cache ways to 4. The hardware prefetcher can help, but it is very variable in function of the workload. If you enable it, then consider increasing the number of refill slots to at least 2, ideally 3.

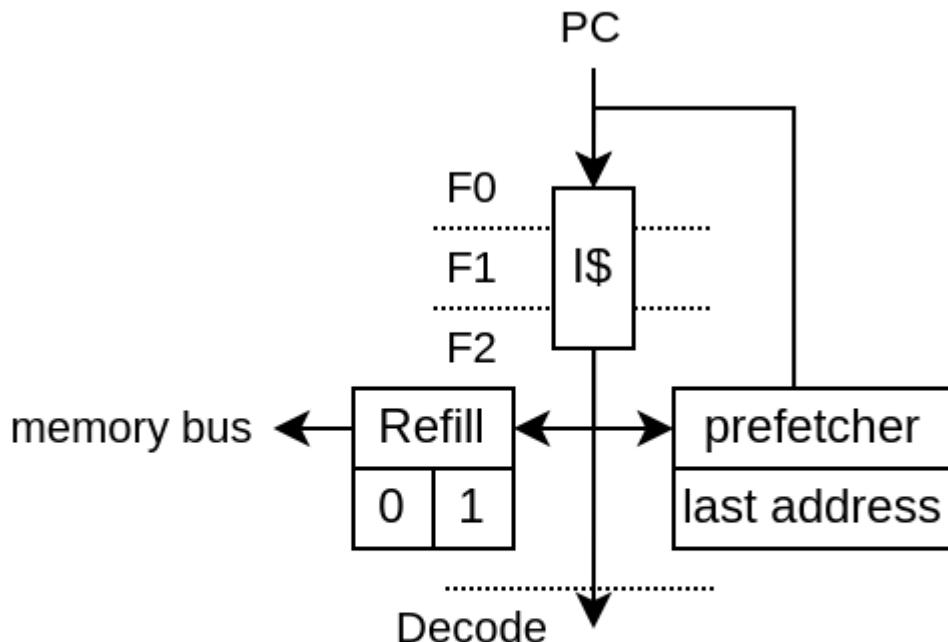
6.5 PrefetcherNextLinePlugin

Currently, there is one instruction L1 prefetcher implementation (PrefetchNextLinePlugin).

It is a very simple implementation :

- On L1 access miss, it trigger the prefetching of the next cache line
- On L1 access hit, if the cache line accessed is the same than the last prefetch, is trigger the prefetching of the next cache line

In short it can only prefetch one cache block ahead and assume that if there was a cache miss on a block, then the following blocks are likely worth prefetching as well.



On L1 miss

- next line prefetch

On L1 accessing the last prefetch address

- next line prefetch

Note, for the best results, the FetchL1Plugin need to have 2 hardware refill slots instead of 1 (default).

The prefetcher can be turned off by setting the CSR 0x7FF bit 0.

6.6 BtbPlugin

This plugin implement most of the branch prediction logic. See more in the *Branch* chapter

6.7 GSharePlugin

See more in the *Branch* chapter

6.8 HistoryPlugin

Will :

- implement the branch history register
- inject the branch history in the first fetch stage
- allow other plugin to create interface to override the branch history value (on branch prediction / execution)

branch history interfaces will impact the branch history value injected in the fetch stage in a combinatorial manner to reduce latency.

DECODE

A few plugins operate in the fetch stage :

- DecodePipelinePlugin
- AlignerPlugin
- DecoderPlugin
- DispatchPlugin
- DecodePredictionPlugin

7.1 DecodePipelinePlugin

Provide the pipeline framework for all the decode related hardware. It use the spinal.lib.misc.pipeline API but implement multiple "lanes" in it.

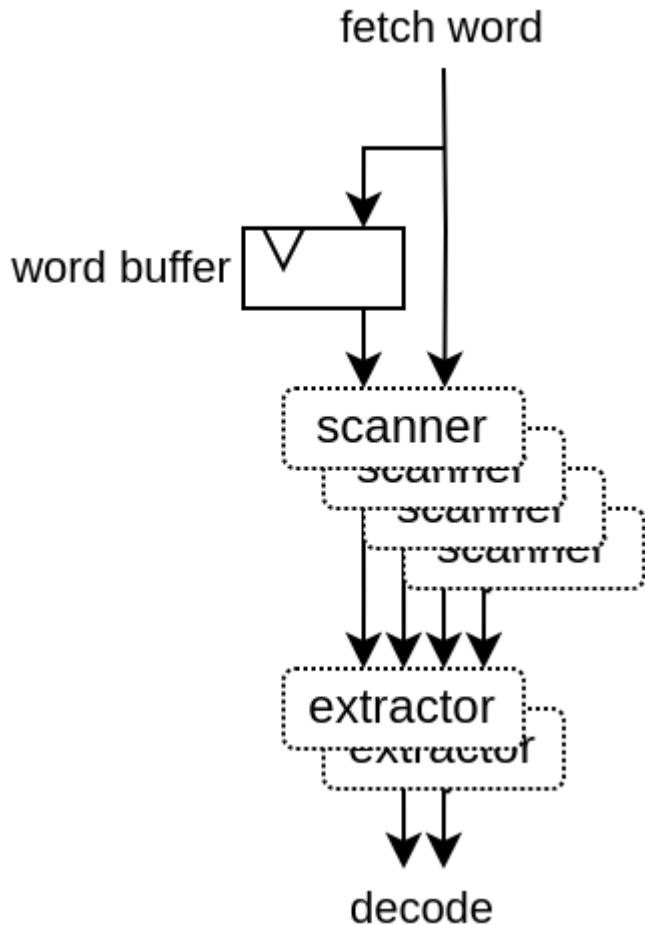
7.2 AlignerPlugin

Decode the words from the fetch pipeline into aligned instructions in the decode pipeline. Its complexity mostly come from the necessity to support having RVC [and BTB], mostly by adding additional cases to handle.

- 1) RVC allows 32 bits instruction to be unaligned, meaning they can cross between 2 fetched words, so it need to have some internal buffer / states to work.
- 2) The BTB may have predicted (falsely) a jump instruction where there is none, which may cut the fetch of an 32 bits instruction in the middle.

The AlignerPlugin is designed as following :

- Has a internal fetch word buffer in oder to support 32 bits instruction with RVC
- First it scan at every possible instruction position, ex : RVC with 64 bits fetch words => 2x64/16 scanners. Extracting the instruction length, presence of all the instruction data (slices) and necessity to redo the fetch because of a bad BTB prediction.
- Then it has one extractor per decoding lane. They will check the scanner for the firsts valid instructions.
- Then each extractor is fed into the decoder pipeline.



7.3 DecoderPlugin

Will :

- Decode instruction
- Generate illegal instruction exception
- Generate "interrupt" instruction
- Ensure that no instruction predicted as a branch/jump by the BTB (but isn't a branch/jump) doesn't goes any further. (See more in the Branch prediction chapter)

7.4 DispatchPlugin

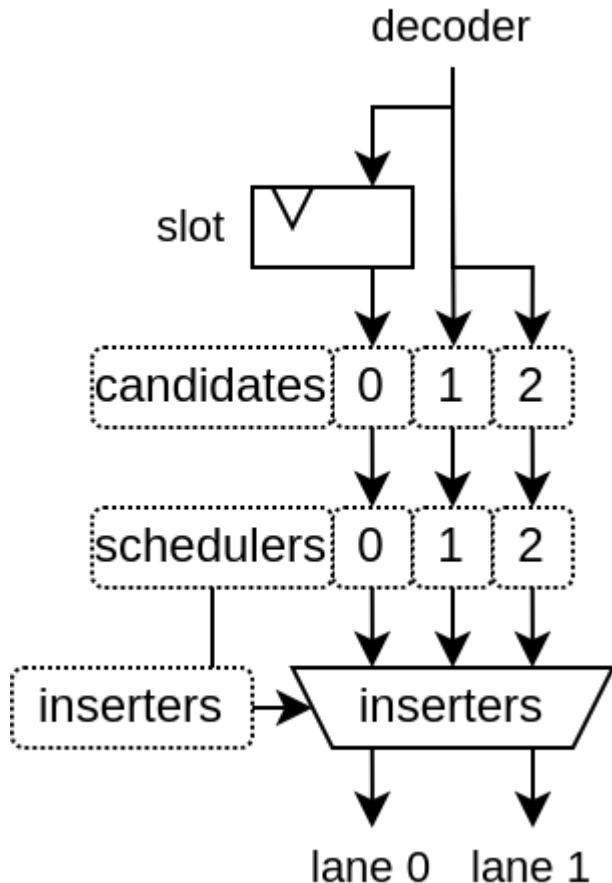
This is probably the hardest part of the VexiiRiscv hardware description to read, as it does a lot of elaboration time computing in order to figure out what hardware need to be generated.

The function of the plugin is to :

- Collect instruction from the end of the decode pipeline
- Dispatch them on the multiple "execution layers" (Execution lanes's ALUs) available when all dependencies are done.

7.4.1 Architecture

Here is a diagram of the DispatchPlugin hardware for a dual issue VexiiRiscv :



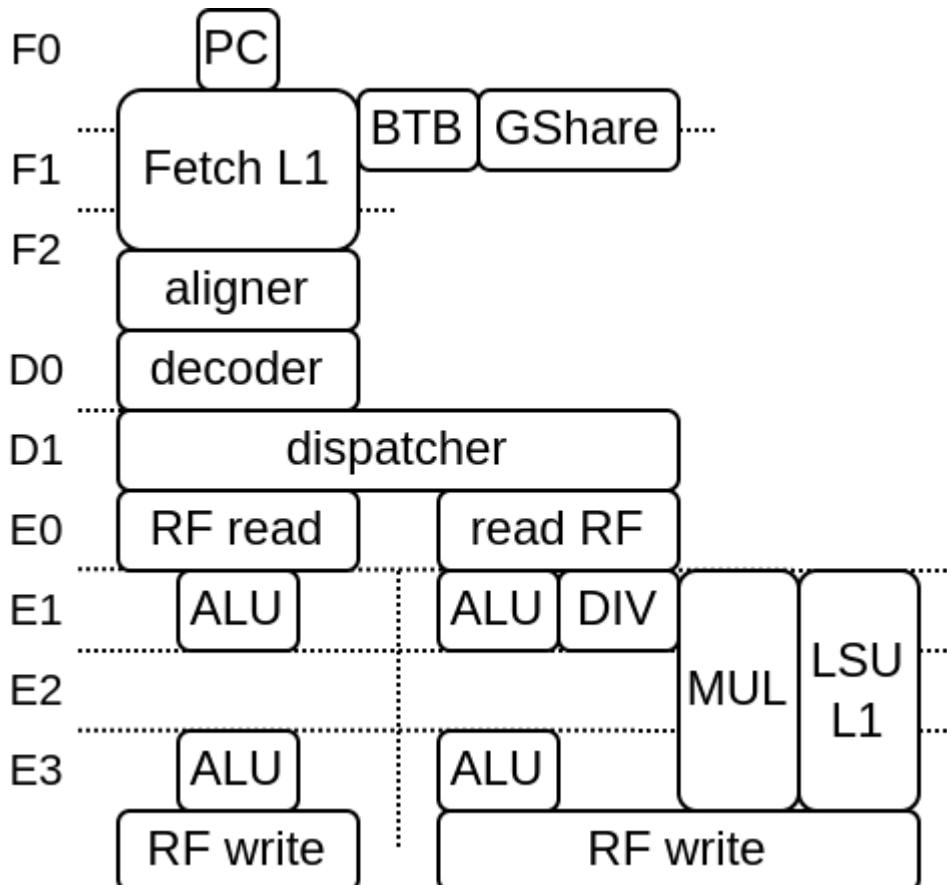
Here is a few explanation about execute lanes and layers :

- A execute lane represent a path toward which an instruction can be executed.
- A execute lane can have one or many layers, which can be used to implement things as early ALU / late ALU
- Each layer will have a static scheduling priority

The DispatchPlugin doesn't require lanes or layers to be symmetric in any way.

Here is an picture example of VexiiRiscv with 2 execution lanes and 2 layer per execution lane. the 2 execution lanes are separated left and right in stages E1-E2-E3.

- Left E1 ALU is one layer, with highest priority, as it provide the best timings and keep the LSU/MUL/DIV path free
- Right E1 ALU/DIV/MUL/LSU is one layer, with high priority, as it provide the best timings but it does allocate the MUL/LSU path as well (even if the instruction doesn't need it)
- Left E3 ALU is one layer, with low priority, as it provide a late ALU result (bad for dependencies).
- Right E3 ALU is one layer, with lowest priority, as it provide a late ALU result and also allocate the MUL/LSU path as well.



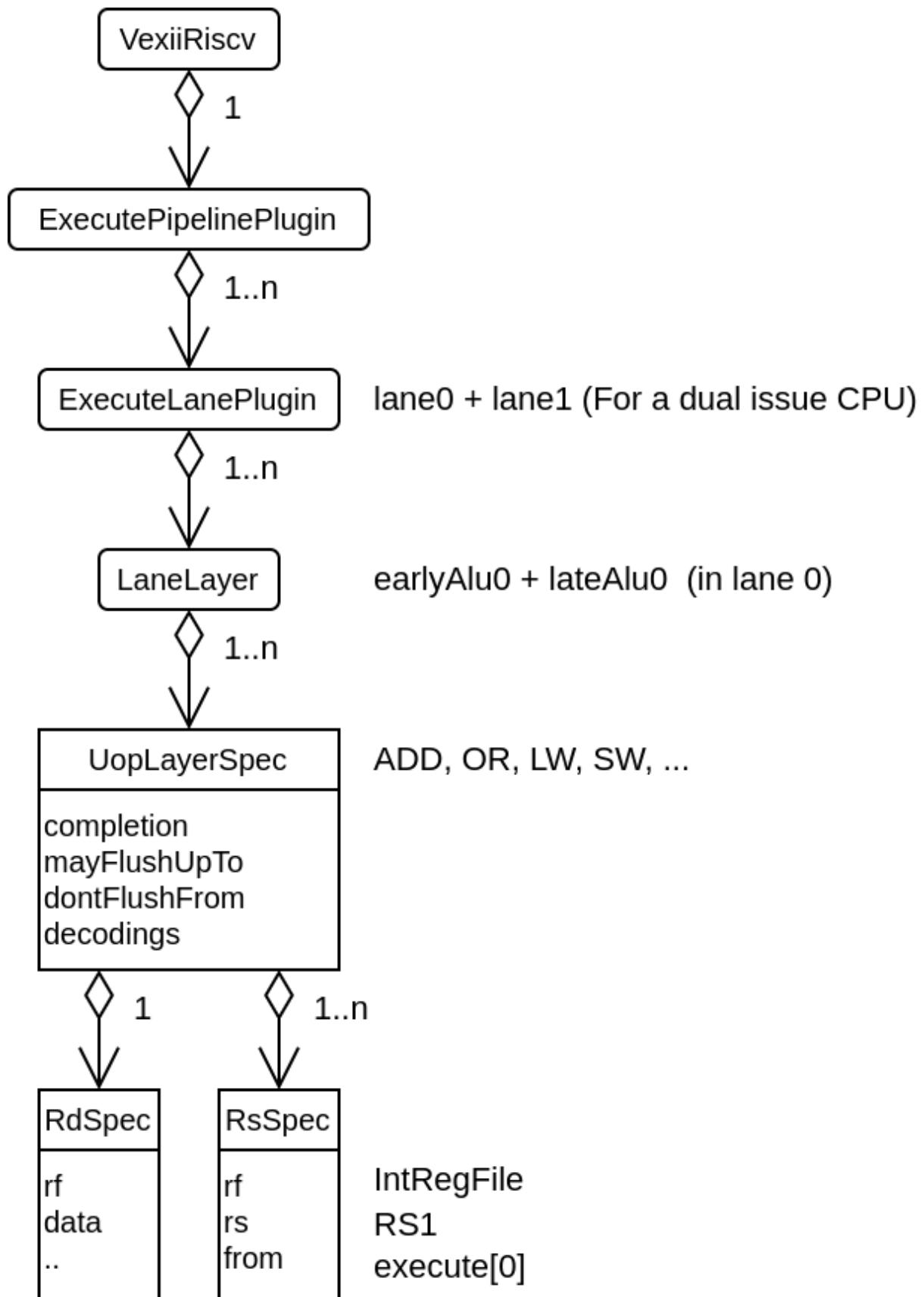
Here are a list of things that the schedulers need to take in account to know on which layer an instruction could be scheduled :

- Check if, in the future (after the instruction side-effects timing), the instruction could be flushed by an already scheduled instruction
- Check at which stage of the execute pipeline the instruction need its RS (operands) to be readable (this is the main feature allowing late-alu)
- Check if the timing at which the instruction would use shared resources would conflict with something already scheduled
- Check if a instruction fence is pending
- And a few other minor things

The inserter will then select which candidates instruction can be executed in which execution lane / layer depending the instruction order and layer priorities.

7.4.2 Elaboration

This is what make the DispatcherPlugin quite special. During elaboration time, it look at the specification of every execution lane's layers, to figure out which instruction it supports and what are its dependencies / limitations, and then try to generate a scheduler for it.



**CHAPTER
EIGHT**

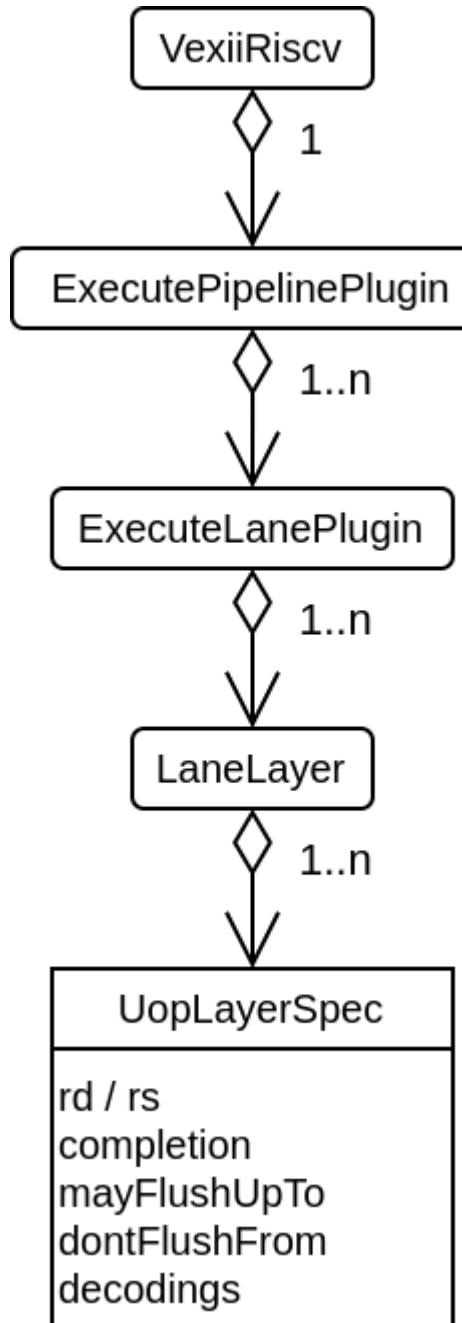
EXECUTE

8.1 Introduction

The execute pipeline has the following properties :

- Support multiple lane of execution.
- Support multiple implementation of the same instruction on the same lane (late-alu) via the concept of "layer"
- each layer is owned by a given lane
- each layer can implement multiple instructions and store a data model of their requirements.
- The whole pipeline never collapse bubbles, all lanes of every stage move forward together as one.
- Elements of the pipeline are allowed to stop the whole pipeline via a shared freeze interface.

Here is a class diagram :



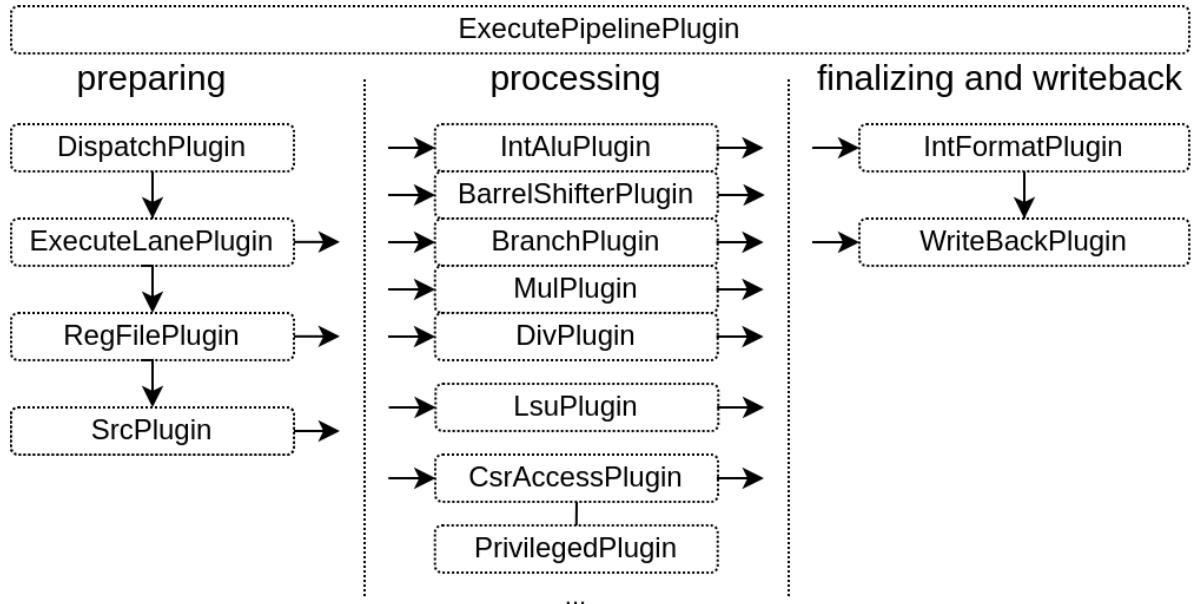
The main thing about it is that for every uop implementation in the pipeline, there is the elaboration time information for :

- How/where to retrieve the result of the instruction (rd)
- From which point in the pipeline it use which register file (rs)
- From which point in the pipeline the instruction can be considered as done (completion)
- Until which point in the pipeline the instruction may flush younger instructions (mayFlushUpTo)
- From which point in the pipeline the instruction should not be flushed anymore because it already had produced side effects (dontFlushFrom)
- The list of decoded signals/values that the instruction is using (decodings)

The idea is that with all those information, the **ExecuteLanePlugin** and **DispatchPlugin** **DecodePlugin** are able to generate the proper logics to generate a functional pipeline / dispatch / decoder with no hand written hardcoded hardware.

8.2 Plugins

The execute pipeline is composed by many plugins, here is a diagram to illustrate the flow of instructions through them :



8.2.1 Infrastructures

Many of the plugins operating in the execute stage aren't directly implementing instructions, but instead provide some infrastructure which will be used to do so.

ExecutePipelinePlugin

Provide the pipeline framework for all the execute related hardware with the following specificities :

- For flow control, the lanes can only freeze the whole pipeline
- The pipeline do not collapse bubbles (a bubble is a stage with no instruction at a given cycle)

ExecuteLanePlugin

Implement an execution lane in the ExecutePipelinePlugin :

- Read the register files
- Implement the register files write to read bypasses networks
- Provide a pipelining API built on the top ExecutePipelinePlugin. That API allows to operate in the given lane.

RegFilePlugin

Implement one register file, with the possibility to create new read / write port on demands.

SrcPlugin

Provide some integer values to instruction which can mux between RS1/RS2 and multiple RISC-V instruction's literal values :

- SRC1 can be : RS1 or U literal
- SRC2 can be : RS1 or PC or I or S literal

It also provide the hardware for a :

- SRC1 + SRC2
- SRC1 - SRC2
- SRC1 < SRC2

RsUnsignedPlugin

Used by mul/div in order to get an unsigned RS1/RS2 value early in the pipeline

IntFormatPlugin

Allows plugins to sign extends their result values using a shared hardware. It uses the WriteBackPlugin to write its results back to the register file.

WriteBackPlugin

Used by plugins to inject results into the pipeline, which will then be written into the register file.

LearnPlugin

Will collect all interface which provide jump/branch learning interfaces to aggregate them into a single one, which will then be used by branch prediction plugins to learn.

8.2.2 Instructions

Some plugins just focus on implementing the CPU instructions.

IntAluPlugin

Implement the arithmetic, binary and literal instructions (ADD, SUB, AND, OR, LUI, ...)

BarrelShifterPlugin

Implement the shift instructions in a non-blocking way (no iterations). Fast but "heavy".

BranchPlugin

Will :

- Implement branch/jump instruction
- Correct the PC / History in the case the branch prediction was wrong
- Provide a learn interface to the LearnPlugin

MulPlugin

- Implement multiplication operation using partial multiplications and then summing their result
- Done over multiple stage
- Can optionally extends the last stage for one cycle in order to buffer the MULH bits

DivPlugin

- Implement the division/remain instructions
- Can be configured in Radix 2/4 (1/ bits per cycle are solved)
- When it start, it scan for the numerator leading bits for 0, and can skip dividing them (can skip blocks of XLEN/4)

LsuCachelessPlugin

- Implement load / store through a cacheless memory bus
- Will fork the cmd as soon as fork stage is valid (with no flush)
- Handle backpressure by using a little fifo on the response data

More information in the *Memory (LSU)* chapter

LsuPlugin

Implement load / store through a 11 cache.

More information in the *Memory (LSU)* chapter

CsrAccessPlugin

- Implement the CSR read and write instruction in the execute pipeline
- Provide an API for other plugins to specify the mapping between the CSR registers and the CSR instruction

See the *Privileges* chapter for more information.

EnvPlugin

See the [Privileges](#) chapter for more information.

- Implement a few instructions as MRET, SRET, ECALL, EBREAK, FENCE.I, WFI by producing hardware traps

8.3 Custom instruction

There are multiple ways you can add custom instructions into VexiiRiscv. The following chapter will provide some demo.

8.3.1 SIMD add

Let's define a plugin which will implement a SIMD add (4x8bits adder), working on the integer register file.

The plugin will be based on the ExecutionUnitElementSimple which makes implementing ALU plugins simpler. Such a plugin can then be used to compose a given execution lane layer

For instance the Plugin configuration could be :

```
plugins += new SrcPlugin(early0, executeAt = 0, relaxedRs = relaxedSrc)
plugins += new IntAluPlugin(early0, formatAt = 0)
plugins += new BarrelShifterPlugin(early0, formatAt = relaxedShift.toInt)
plugins += new IntFormatPlugin("lane0")
plugins += new BranchPlugin(early0, aluAt = 0, jumpAt = relaxedBranch.toInt, wbAt = 0)
plugins += new SimdAddPlugin(early0) // <- We will implement this plugin
```

Plugin implementation

Here is a example how this plugin could be implemented :

- <https://github.com/SpinalHDL/VexiiRiscv/blob/dev/src/main/scala/vexiiriscv/execute/SimdAddPlugin.scala>

```
package vexiiriscv.execute

import spinal.core._
import spinal.lib._
import spinal.lib.pipeline.Stageable
import vexiiriscv.Generate.args
import vexiiriscv.{Global, ParamSimple, VexiiRiscv}
import vexiiriscv.compat.MultiPortWritesSymplifier
import vexiiriscv.riscv.{IntRegFile, RS1, RS2, Riscv}

// This plugin example will add a new instruction named SIMD_ADD which do the
// following :
//
// RD : Regfile Destination, RS : Regfile Source
// RD( 7 downto 0) = RS1( 7 downto 0) + RS2( 7 downto 0)
// RD(16 downto 8) = RS1(16 downto 8) + RS2(16 downto 8)
// RD(23 downto 16) = RS1(23 downto 16) + RS2(23 downto 16)
// RD(31 downto 24) = RS1(31 downto 24) + RS2(31 downto 24)
//
// Instruction encoding :
// 0000000-----000-----0001011    <- Custom0 func3=0 func7=0
```

(continues on next page)

(continued from previous page)

```

//          |RS2| |RS1|   |RD |
//
// Note : RS1, RS2, RD positions follow the RISC-V spec and are common for all
// instruction of the ISA

object SimdAddPlugin{
    // Define the instruction type and encoding that we will use
    val ADD4 = IntRegFile.TypeR(M"0000000-----000----0001011")
}

// ExecutionUnitElementSimple is a plugin base class which will integrate itself in a
// execute lane layer
// It provide quite a few utilities to ease the implementation of custom instruction.
// Here we will implement a plugin which provide SIMD add on the register file.
class SimdAddPlugin(val layer : LaneLayer) extends ExecutionUnitElementSimple(layer) {

    // Here we create an elaboration thread. The Logic class is provided by
    // ExecutionUnitElementSimple to provide functionalities
    val logic = during setup new Logic {
        // Here we could have lock the elaboration of some other plugins (ex CSR), but
        // here we don't need any of that
        // as all is already sorted out in the Logic base class.
        // So we just wait for the build phase
        awaitBuild()

        // Let's assume we only support RV32 for now
        assert(Riscv.XLEN.get == 32)

        // Let's get the hardware interface that we will use to provide the result of our
        // custom instruction
        val wb = newWriteback(ifp, 0)

        // Specify that the current plugin will implement the ADD4 instruction
        val add4 = add(SimdAddPlugin.ADD4).spec

        // We need to specify on which stage we start using the register file values
        add4.addRsSpec(RS1, executeAt = 0)
        add4.addRsSpec(RS2, executeAt = 0)

        // Now that we are done specifying everything about the instructions, we can
        // release the Logic.uopRetainer
        // This will allow a few other plugins to continue their elaboration (ex :
        // decoder, dispatcher, ...)
        uopRetainer.release()

        // Let's define some logic in the execute lane [0]
        val process = new el.Execute(id = 0) {
            // Get the RISC-V RS1/RS2 values from the register file
            val rs1 = el(IntRegFile, RS1).asUInt
            val rs2 = el(IntRegFile, RS2).asUInt

            // Do some computation
            val rd = UInt(32 bits)
            rd(7 downto 0) := rs1(7 downto 0) + rs2(7 downto 0)
            rd(16 downto 8) := rs1(16 downto 8) + rs2(16 downto 8)
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

rd(23 downto 16) := rs1(23 downto 16) + rs2(23 downto 16)
rd(31 downto 24) := rs1(31 downto 24) + rs2(31 downto 24)

// Provide the computation value for the writeback
wb.valid := SEL
wb.payload := rd.asBits
}
}
}

```

VexiiRiscv generation

Then, to generate a VexiiRiscv with this new plugin, we could run the following App :

- Bottom of <https://github.com/SpinalHDL/VexiiRiscv/blob/dev/src/main/scala/vexiiriscv/execute/SimdAddPlugin.scala>

```

object VexiiSimdAddGen extends App {
  val param = new ParamSimple()
  val sc = SpinalConfig()

  assert(new scopt.OptionParser[Unit]("VexiiRiscv") {
    help("help").text("prints this usage text")
    param.addOptions(this)
  }.parse(args, Unit).nonEmpty)

  sc.addTransformationPhase(new MultiPortWritesSymplicifier)
  val report = sc.generateVerilog {
    val pa = param.pluginsArea()
    pa.plugins += new SimdAddPlugin(pa.early0)
    VexiiRiscv(pa.plugins)
  }
}

```

To run this App, you can go to the NaxRiscv directory and run :

```
sbt "runMain vexiiriscv.execute.VexiiSimdAddGen"
```

Software test

Then let's write some assembly test code : (<https://github.com/SpinalHDL/NaxSoftware/tree/849679c70b238ceee021bdfd18eb2e9809e7bdd0/baremetal/simdAdd>)

```

.globl _start
_start:

#include "../../driver/riscv_asm.h"
#include "../../driver/sim_asm.h"
#include "../../driver/custom_asm.h"

// Test 1
li x1, 0x01234567
li x2, 0x01FF01FF
opcode_R(CUSTOM0, 0x0, 0x00, x3, x1, x2) // x3 = ADD4(x1, x2)

```

(continues on next page)

(continued from previous page)

```
// Print result value
li x4, PUT_HEX
sw x3, @x4

// Check result
li x5, 0x02224666
bne x3, x5, fail

j pass

pass:
    j pass
fail:
    j fail
```

Compile it with

```
make clean rv32im
```

Simulation

You could run a simulation using this testbench :

- Bottom of <https://github.com/SpinalHDL/VexiiRiscv/blob/dev/src/main/scala/vexiiriscv/execute/SimdAddPlugin.scala>

```
object VexiiSimdAddSim extends App {
    val param = new ParamSimple()
    val testOpt = new TestOptions()

    val genConfig = SpinalConfig()
    genConfig.includeSimulation

    val simConfig = SpinalSimConfig()
    simConfig.withFstWave
    simConfig.withTestFolder
    simConfig.withConfig(genConfig)

    assert(new scopt.OptionParser[Unit]("VexiiRiscv") {
        help("help").text("prints this usage text")
        testOpt.addOptions(this)
        param.addOptions(this)
    }.parse(args, Unit).nonEmpty)

    println(s"With Vexiiriscv param :\n - ${param.getName()}")
    val compiled = simConfig.compile {
        val pa = param.pluginsArea()
        pa.plugins += new SimdAddPlugin(pa.early0)
        VexiiRiscv(pa.plugins)
    }
    testOpt.test(compiled)
}
```

Which can be run with :

```
sbt "runMain vexiiriscv.execute.VexiiSimdAddSim --load-elf ext/NaxSoftware/baremetal/
→simdAdd/build/rv32ima/simdAdd.elf --trace-all --no-rvls-check"
```

Which will output the value 02224666 in the shell and show traces in simWorkspace/VexiiRiscv/test :D

Note that --no-rvls-check is required as spike do not implement that custom simdAdd.

Conclusion

So overall this example didn't introduce how to specify some additional decoding, nor how to define multi-cycle ALU. (TODO). But you can take a look in the IntAluPlugin, ShiftPlugin, DivPlugin, MulPlugin and BranchPlugin which are doing those things using the same ExecutionUnitElementSimple base class.

8.4 FPU

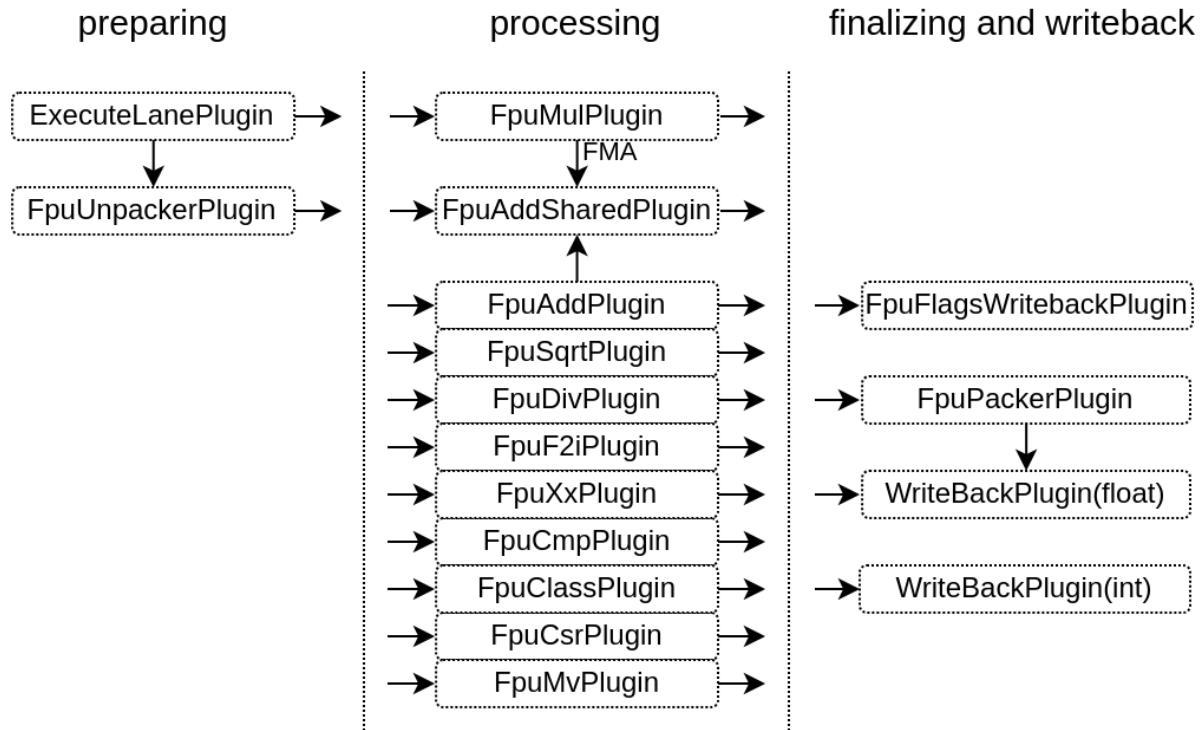
The VexiiRiscv FPU has the following characteristics :

- By default, It is fully compliant with the IEEE-754 spec (subnormal, rounding, exception flags, ..)
- There is options to reduce its footprint at the cost of compliance (reduced FMA accuracy, and drop subnormal support)
- It isn't a single chunky module, instead it is composed of many plugins in the same ways than the rest of the CPU.
- It is tightly coupled to the execute pipeline
- All operations can be issued at the rate of 1 instruction per cycle, excepted for FDIV/FSQRT/Subnormals
- By default, it is deeply pipelined to help with FPGA timings (10 stages FMA)
- Multiple hardware resources are shared between multiple instruction (ex rounding, adder (FMA+FADD))
- The VexiiRiscv scheduler take care to not schedule an instruction which would use the same resource than an older instruction
- FDIV and FMUL reuse the integer pipeline DIV and MUL hardware
- Subnormal numbers are handled by recoding/encoding them on operands and results of math instructions. This will trigger some little state machines which will halt the CPU a few cycles (2-3 cycles)

8.4.1 Plugins architecture

There is a few foundation plugins that compose the FPU :

- FpuUnpackPlugin : Will decode the RS1/2/3 operands (isZero, isInfinity, ..) as well as recode them in a floating point format which simplify subnormals into regular floating point values
- FpuPackPlugin : Will apply rounding to floating point results, recode them into IEEE-754 (including sub-normal) before sending those to the WriteBackPlugin(float)
- WriteBackPlugin(float) : Allows to write values back to the register file (it is the same implementation as the WriteBackPlugin(integer))
- FpuFlagsWriteback ; Allows instruction to set FPU exception flags



8.4.2 Area / Timings options

To improve the FPU area and timings (especially on FPGA), there are currently two main options implemented.

The first option is to reduce the FMA (Float Multiply Add instruction $A \cdot B + C$) accuracy. The reason is that the mantissa result of the multiply operation (for 64 bits float) is $2x(52+1)=106$ bits, then we need to take those bits and implement the floating point adder against the third operand. So, instead of having to do a 52 bits + 52 bits floating point adder, we need to do a 106 bits + 52 bits floating point adder, which is quite heavy, increase the timings and latencies while being (very likely) overkilled. So this option throw away about half of the multiplication mantissa result.

The second option is to disable subnormal support, and instead consider those values as normal floating point numbers. This reduces the area by not having to handle subnormals (it removes big barrel shifters), as well as improving timings. The downside is that the floating point value range is slightly reduced, and if the user provides floating point constants which are subnormals, they will be considered as $2^{\text{exp_subnormal}}$ numbers.

In practice, these two options do not seem to cause issues (for regular use cases), as it was tested by running Debian with various software and graphical environments.

8.4.3 Optimized software

If you used the default FPU configuration (deeply pipelined), and you want to achieve a high FPU bandwidth, your software needs to be careful about dependencies between instructions. For instance, a FMA instruction will have around 10 cycle latency before providing its results, so if you want for instance to multiply 1000 values against some constants and accumulate the results together, you will need to accumulate things using multiple accumulators and then, only at the end, aggregate the accumulators together.

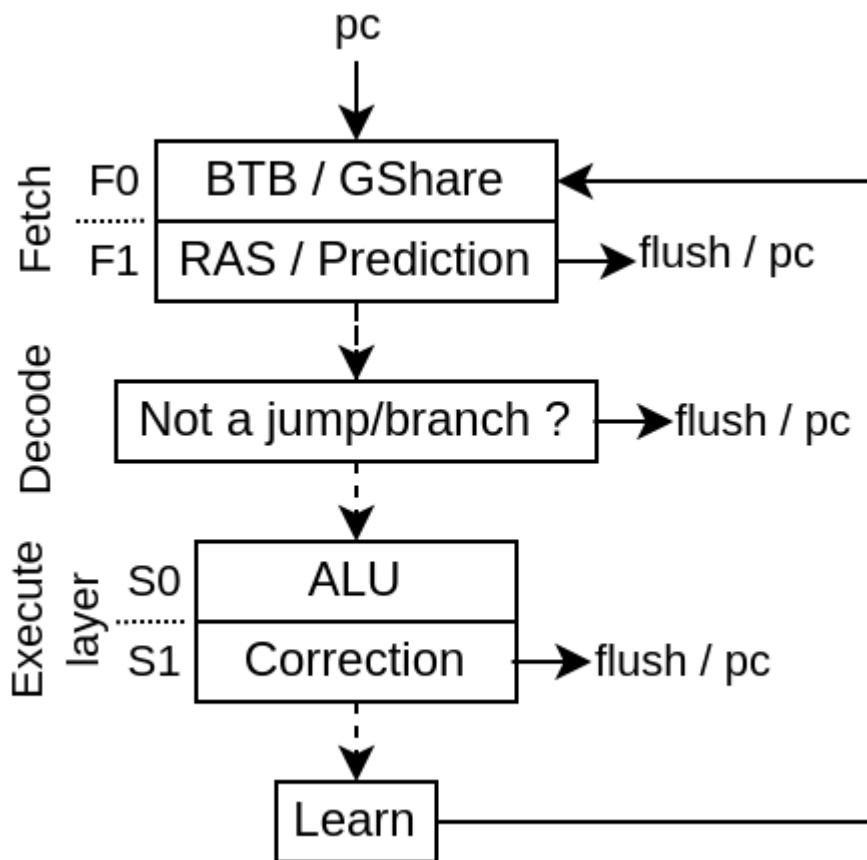
So think about code pipelining. GCC will not necessarily do a good job about it, as it may assume that the FPU has a much lower latency, or just optimize for code size.

BRANCH

The branch prediction is implemented as follow :

- During fetch, a BTB, GShare, RAS memory is used to provide an early branch prediction (BtbPlugin / GSharePlugin)
- In Decode, the DecodePredictionPlugin will ensure that no "none jump/branch instruction"" predicted as a jump/branch continues down the pipeline.
- In Execute, the prediction made is checked and eventually corrected. Also a stream of data is generated to feed the BTB / GShare memories with good data to learn.

Here is a diagram of the whole architecture :

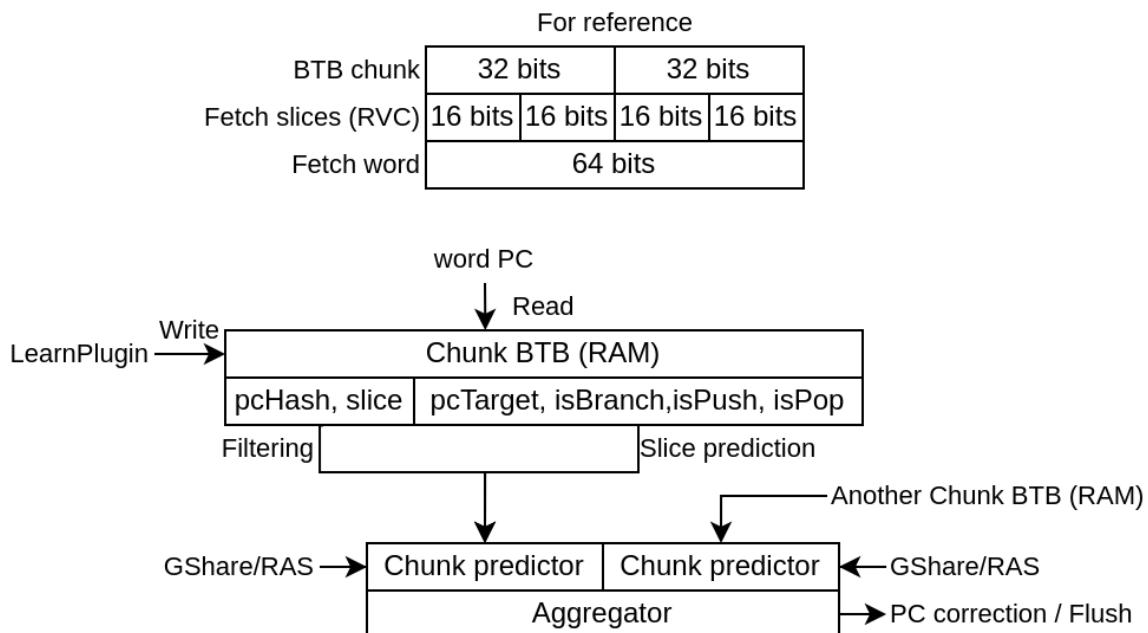


While it would have been possible in the decode stage to correct some miss prediction from the BTB / RAS, it isn't done to improve timings and reduce Area.

9.1 BtbPlugin

Will :

- Implement a branch target buffer in the fetch pipeline
- Implement a return address stack buffer
- Predict which slices of the fetched word are the last slice of a branch/jump
- Predict the branch/jump target
- Predict if the given instruction is a branch, a jump or something else
- Predict if the given instruction should push or pop the RAS (Return Address Stack)
- Use the FetchConditionalPrediction plugin (GSharePlugin) to know if branch should be taken
- Apply the prediction (flush + pc update + history update)
- Learn using the LearnPlugin interface. Only learn on misprediction. To avoid write to read hazard, the fetch stage is blocked when it learns.
- Implement "ways" named chunks which are statically assigned to groups of word's slices, allowing to predict multiple branch/jump present in the same word



Note that it may help to not make the BTB learn when there has been a non-taken branch.

- The BTB don't need to predict non-taken branch
- Keep the BTB entry for something more useful
- For configs in which multiple instruction can reside in a single fetch word (ex dual issue with RVC), multiple branch/jump instruction can reside in a single fetch word => need for compromises, and hope that some of the branch/jump in the chunk are rarely taken.

9.2 GSharePlugin

Will :

- Implement a FetchConditionalPrediction (GShare flavor)
- Learn using the LearnPlugin interface. Write to read hazard are handled via a bypass
- Will not apply the prediction via flush / pc change, another plugin will do that (ex : BtbPlugin)

Note that one of the current issue with GShare, is that it take quite a few iterations to learn (depending the branch history)

9.3 DecodePlugin

The DecodePlugin, in addition of just decoding the incoming instructions, will also ensure that no branch/jump prediction was made for non branch/jump instructions. In case this is detected, the plugin will :

- schedule a "REDO trap" which will flush everything and make the CPU jump to the failed instruction
- Make the predictor skip the first incoming prediction
- Make the predictor unlearn the prediction entry which failed

9.4 BranchPlugin

Placed in the execute pipeline, it will ensure that the branch predictions were correct, else it correct them. It also generate a learn interface to feed the LearnPlugin.

9.5 LearnPlugin

This plugin will collect all the learn interface (generated by the BranchPlugin) and produce a single stream of learn interface for the BtbPlugin / GShare plugin to use.

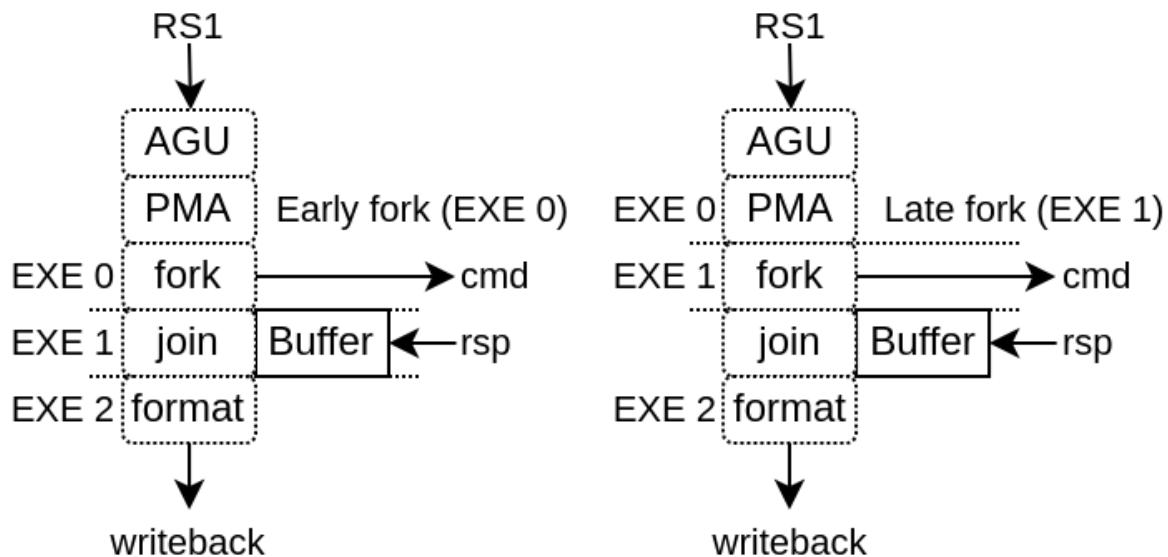
MEMORY (LSU)

LSU stand for Load Store Unit, VexiiRiscv has currently 2 implementations for it:

- LsuCachelessPlugin for microcontrollers, which doesn't implement any cache
- LsuPlugin / LsuL1Plugin which can work together to implement load and store through an L1 cache

10.1 Without L1

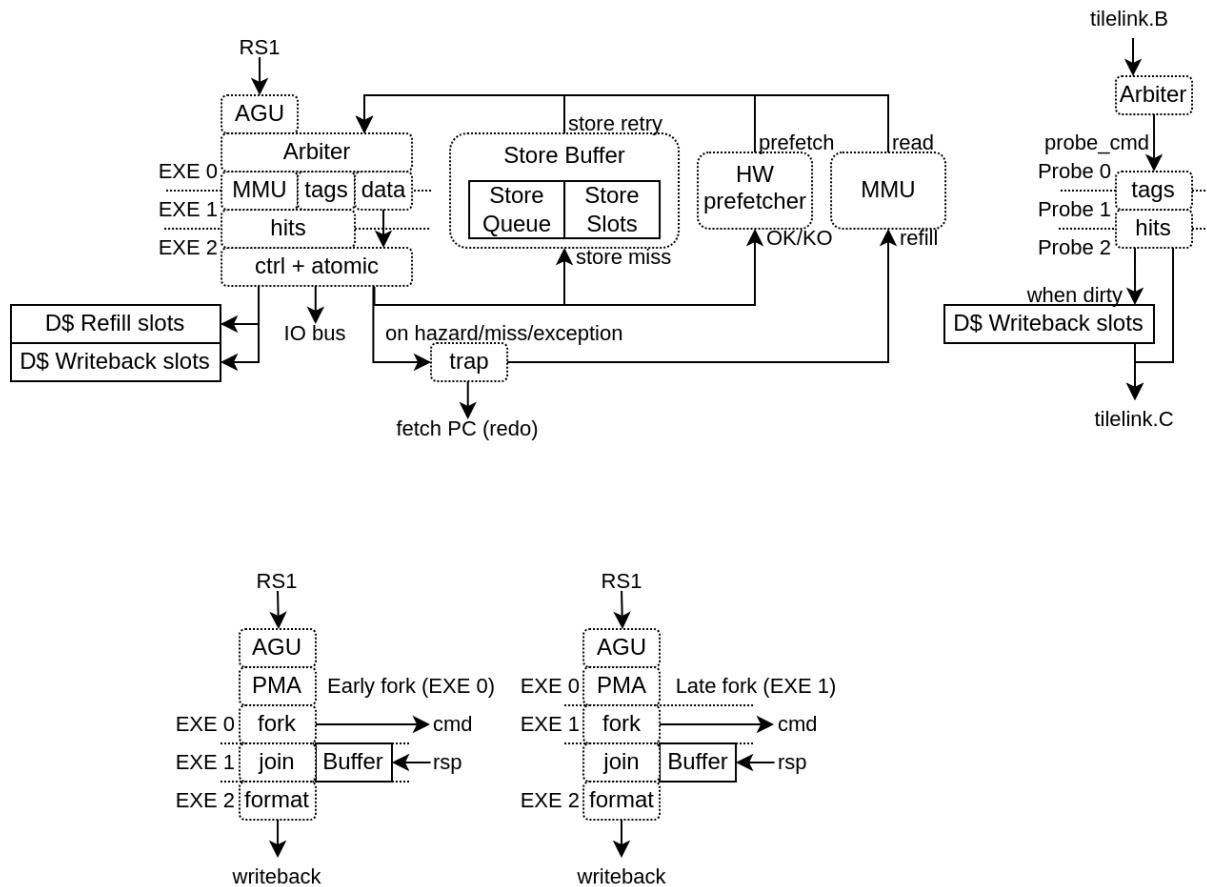
Implemented by the LsuCachelessPlugin, it should be noted that to reach good frequencies on FPGA SoC, forking the memory request at execute stage 1 seems to provide the best results (instead of execute stage 0), as it relax the AGU timings as well as the PMA (Physical Memory Attributes) checks.



10.2 With L1

This configuration supports :

- N ways (limited to 4 KB per way if the MMU is enabled)
- Non-blocking design, able to handle multiple cache line refill and writeback
- Hardware and software prefetching (RPT design)



This LSU implementation is partitioned between 2 plugins :

The LsuPlugin :

- Implement AGU (Address Generation Unit)
- Arbitrate all the different sources of memory request (AGU, store queue, prefetch, MMU refill)
- Provide the memory request to the LsuL1Plugin
- Bind the MMU translation port
- Handle the exceptions and hazard recovery
- Handle the atomic operations (ALU + locking of the given cache line)
- Handle IO memory accesses
- Implement the store queue to handle store misses in a non-blocking way
- Feed the hardware prefetcher with load/store execution traces

The LsuL1Plugin :

- Implement the L1 tags and data storage
- Implement the cache line refill and writeback slots (non-blocking)
- Implement the store to load bypasses
- Implement the memory coherency interface
- Is integrated in the execute pipeline (to save area and improve timings)

For multiple reasons (ease of implementation, FMax, hardware usage), VexiiRiscv LSU can hit hazards situations :

- Cache miss, MMU miss

- Refill / Writeback aliasing (4KB)
- Unread data bank during load (ex : load during data bank refill)
- Load which hit the store queue
- Store miss while the store queue is full
- ...

In those situation, the LsuPlugin will trigger an "hardware trap" which will flush the pipeline and reschedule the failed instruction to the fetch unit.

Here is a set of options which can be used :

Table 1: Generation parameters

Parameter	Description
--lsu-l1	Enable the L1 D\$
--lsu-l1-ways=X	Specify the number of ways for the L1 D\$ (how many direct mapped caches in parallel), default=1
--lsu-l1-sets=X	Specify the number of sets for the L1 D\$ (how many line of cache per way), default=64
--lsu-l1-mem-data-width-min=X	Set a lower bound for the L1 D\$ data width
--lsu-software-prefetch	Enable RISC-V CMO for software prefetching in the D\$
--lsu-hardware-prefetch rpt	Enable the L1 D\$ hardware prefetcher (based on RPT)
--lsu-l1-store-buffer-ops=X	Specify how many store miss can be pushed in the store buffer (disabled/0 by default)
--lsu-l1-store-buffer-slots=X	Specify how many block of memory can be targeted by the store buffer (disabled/0 by default)
--lsu-l1-refill-count=X	Specify how many cache line refill the L1 D\$ can handle at the same time, default=1
--lsu-l1-writeback-count=X	Specify how many cache line writeback the L1 D\$ can handle at the same time, default=1

To improve the performances, consider first increasing the number of cache ways to 4.

The store buffer will help a lot with the store bandwidth by allowing the CPU to not be blocked by every store miss. The hardware prefetcher will help with both store/load bandwidth (but if the store buffer is already enabled, it will not really increase the store bandwidth).

For the hardware prefetcher to stretch its leg, consider using 4 refill/writeback slots. This will also help the store buffer.

10.2.1 Prefetching

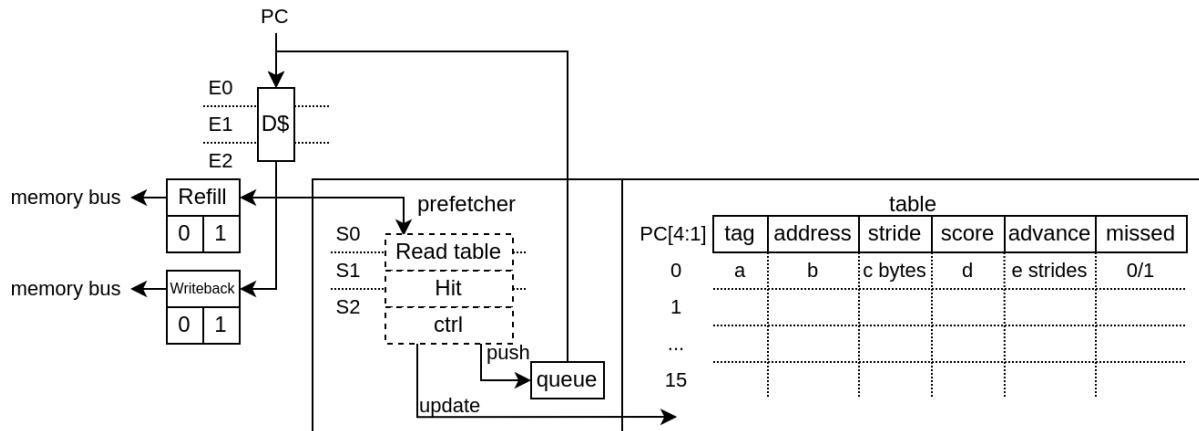
Currently there is two implementation of prefetching

- PrefetchNextLinePlugin : As its name indicates, on each cache miss it will prefetch the next cache line
- PrefetchRptPlugin : Enable prefetching for instruction which have a constant stride between accesses

PrefetchRptPlugin

This prefetcher is capable of recognizing instructions which have a constant stride between their own previous accesses in order to prefetch multiple strides ahead.

- Will learn memory accesses patterns from the LsuPlugin traces
- Patterns need to have a constant stride in order to be recognized
- By default, it can keep track of up to 128 instructions access pattern (1 way * 128 sets, pc indexed)



This can improve performance dramatically (for some use cases). For instance, on a 100 MHz SoC in a FPGA, equipped of a 16x800 MT/s DDR3, the load bandwidth went from 112 MB/s to 449 MB/s. (sequential load)

Here is a description of the table fields :

"Tag" : Allows to get a better idea if the given instruction (PC) is the one owning the table entry by comparing more PC's MSB bits. An entry is "owned" by an instruction if its tag match the given instruction PC's msb bits.

"Address" : Previous virtual address generated by the instruction

"stride" : Number of bytes expected between memory accesses

"Score" : Allows to know if the given entry is useful or not. Each time the instruction is keeping the same stride, the score increase, else it decrease. If another instruction (with another tag) want to use an entry, the score field has to be low enough.

"Advance" : Allows to keep track how far the prefetching for the given instruction already went. This field is cleared when a entry switch to a new instruction

"Missed" : This field was added in order to reduce the spam of redundant prefetch request which were happening for load/store intensive code. For instance, for a deeply unrolled memory clear loop will generate (x16), as each store instruction PC will be tracked individually, and as each execution of a given instruction will stride over a full cache line, this will generate one hardware prefetch request on each store instruction every time, spamming the LSU pipeline with redundant requests and reducing overall performances.

This "missed" field works as following :

- It is cleared when a stride disruption happens (ex new memcpy execution)
- It is set on cache miss (set win over clear)
- An instruction will only trigger a prefetch if it miss or if its "missed" field is already set.

For example, in a hardware simulation test (RV64, 20 cycles memory latency, 16xload loop), this addition increased the memory read memory bandwidth from 3.6 bytes/cycle to 6.8 bytes per cycle.

Note that if you want to take full advantage of this prefetcher, you need to have enough hardware refill/writeback slots in the LsuL1Plugin.

Also, prefetch which fail (ex : because of hazards in L1) aren't replayed.

The prefetcher can be turned off by setting the CSR 0x7FF bit 1.

performance measurements

Here are a few performance gain measurements done on litex with a :

- quad-core RV64GC running at 200 MHz
- 16 KB L1 cache for each core
- 512 KB of L2 cache shared (128 bits data bus)
- 4 refill slots + 4 writeback slots + 32 entry store queue + 4 slots store queue

Table 2: Prefetch performance

Test	No prefetch	RPT prefetch
Litex bios read speed	204.2MiB/s	790.9MiB/s
Litex bios write speed	559.2MiB/s	576.8MiB/s
iperf3 RX	617 Mbits/sec	766 Mbits/sec
iperf3 TX	623 Mbits/sec	623 Mbits/sec
chocolate-doom -1 demo1.lmp	43.1 fps	50.2 fps

10.2.2 Hardware Memory coherency

Hardware memory coherency, is the feature which allows multiple memory agents (ex : CPU, DMA, ...) to work on the same memory locations and notify each others when they change their contents. Without it, the CPU software would have to manually flush/invalidate their L1 caches to keep things in sync.

There is mostly 2 kinds of hardware memory coherency architecture :

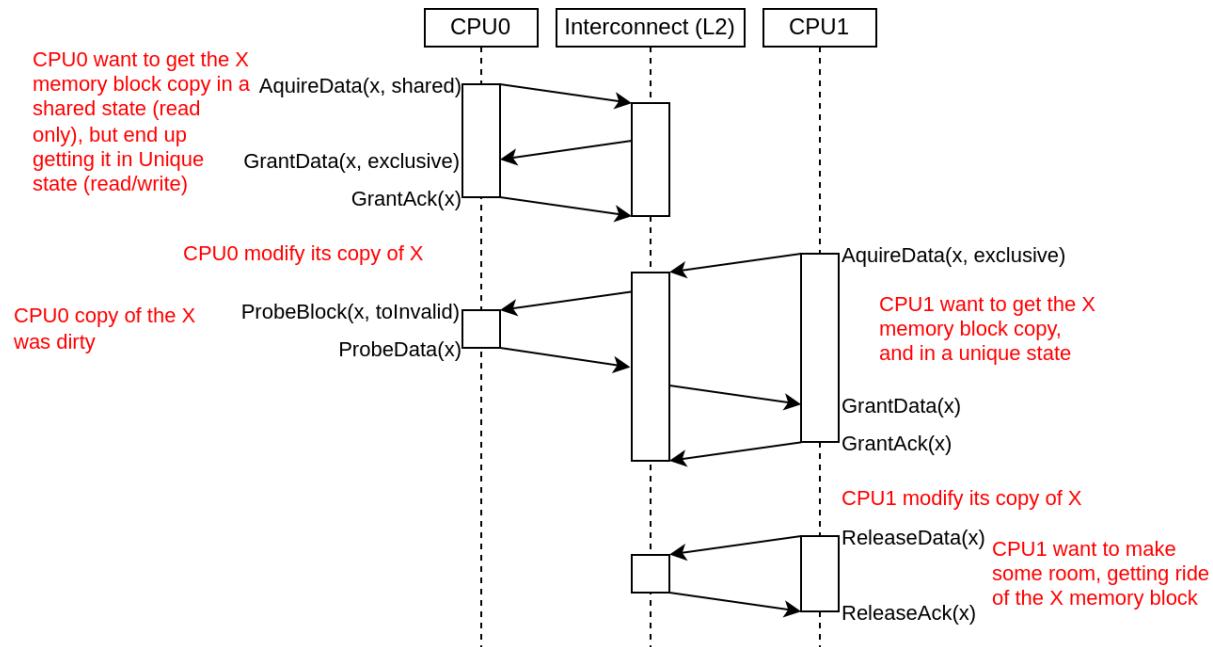
- By invalidation : When a CPU/DMA write some memory, it notifies the other CPU caches that they should invalidate any old copy that they have of the written memory locations. This is generally used for write-through L1 caches. This isn't what VexiiRiscv implements.
- By permission : Memory blocks copies (typically 64 aligned bytes blocks which resides in L1 cache lines) can have multiple states. Some of which provide read only accesses, while others provide read/write accesses. This is generally used in write-back L1 caches, and this is what VexiiRiscv uses.

In VexiiRiscv, the hardware memory coherency (L1) with other memory agents (CPU, DMA, L2, ..) is supported though a MESI implementation which can be bridged to a tilelink memory bus.

MESI is an standard acronym for every possible state that a copy of a memory block can have in the caches :

- I : Invalid, meaning that there is no copy of that memory block
- S : Shared, meaning that the cache has a read only copy of the memory block, and that other caches may also have a copy. This state is sometime named : Shared/Clean
- E : Exclusive, meaning that the cache has a read/writable copy of the memory block which is still in a clean state (unmodified, no writeback required), and that no other cache has a copy of the block. This state is sometime named : Unique/Clean
- M : Modified, meaning that the cache line exclusive, but has been modified, and so, require a writeback later on. This state is sometime named : Unique/Dirty

Here is a diagram which shows an example of memory block copy exchanges between 2 CPUs :



The VexiiRiscv L1 cache interconnect interface is kinda close to what Tilelink specifies and can easily be bridged to Tilelink. The main difference is that probe requests can fail (need to be replayed), and that probes which hit will then go through the writeback interface. Here is the hardware interfaces :

- `read_cmd` : To send memory block acquire requests (invalid/shared -> shared/exclusive)
- `read_rsp` : For responses of the above requests
- `read_ack` : To send acquire requests completion
- `write_cmd` : To send release a memory block permission (shared/exclusive -> invalid)
- `write_rsp` : For responses of the above requests
- `probe_cmd` : To receive probe requests (toInvalid/toShared/toUnique)
- `probe_rsp` : to send responses from the above requests (isValid/isShared/isUnique). When data need to be written back, it will be done through the `write_cmd` channel.

10.2.3 Memory system

Currently, VexiiRiscv can be used with the Tilelink memory interconnect from SpinalHDL and Chipyard (<https://chipyard.readthedocs.io/en/latest/Generators/VexiiRiscv.html>).

Why Tilelink

So, why using Tilelink, while most of the FPGA industry is using AXI4 ? Here are some issues / complexities that AXI4 bring with it. (Dolu1990 opinions, with the perspective of using it in FPGA, with limited manpower, don't see this as an absolute truth)

- The AXI4 memory ordering, while allowing CPU/DMA to get preserved ordering between transactions with the same ID, is creating complexities and bottlenecks in the memory system. Typically in the interconnect decoders to avoid dead-locks, but even more in L2 caches and DRAM controllers which ideally would handle every request out of order. Tilelink instead specify that the CPU/DMA shouldn't assume any memory ordering between inflight transactions.
- AXI4 specifies that memory read response channel can interleave between multiple ongoing bursts. While this can be use full for very large burst (which in itself is a bad idea, see next chapter), this can lead to big area overhead for memory bridges, especially with width adapters. Tilelink doesn't allows this behavior.

- AXI4 splits write address from write data, which add additional synchronizations points in the interconnect decoders/arbiters and peripherals (bad for timings) as well as potentially decrease performances when integrating multiple AXI4 modules which do not use similar address/data timings.
- AXI4 isn't great for low latency memory interconnects, mostly because of the previous point.
- AXI4 splits read and write channels (ar r / aw w b), which mostly double the area cost of address decoding/routing for DMA and non-coherent CPUs.
- AXI4 specifies a few "low values" features which increase complexity and area (ex: WRAP/FIXED bursts, unaligned memory accesses).

Efficiency cookbook

Here are a set of design guideline to keep a memory system lean and efficient (don't see this as an absolute truth) :

- Memory blocks are 64 aligned bytes long : DDR3/4/5 modules are tuned to provides native 64 bytes burst accesses (not less, not more). In particular, with DDR5 modules, they doubled the module burst size (to 16 beats), but in order to preserve 64 bytes burst accesses, they divided the 64 bits physical data width between two independent channels. CPU cache lines, L2 and L3 designs follow that 64 bytes block "rule" as well. Their coherency dictionary will be designed to handle 64 bytes memory blocks too. AMBA 5 CHI enforce 64 bytes cache lines, and doesn't support memory transfers with more than 64 bytes.
- DMA should not reuse the same transaction ID (axi/tilelink) between multiple inflight transactions and should not expect any ordering between inflight transactions. That keep them highly portable and relax the memory system.
- DMA should access up to 64 aligned bytes per burst, this should be enough to reach peak bandwidth. No need for 4KB Rambo bursts. Asking a system to support bursts bigger than 64 aligned bytes can lead to extra cost, as it create new ordering constraints between the memory block of the burst. For instance in a L2 cache it can lead to implementation of a reorder buffer to deal between transaction which hit/miss the cache. Adding extra complexity/area/timings to deal with. Additionally, big burst can create high latency spike for other agents (CPU/DMA).
- DMA should only do burst aligned memory accesses (to keep them easily portable to Tilelink)
- It is fine for DMA to over fetch (let's say you need 48 bytes, but access aligned 64 bytes instead), as long as the bulk of the memory bandwidth is not doing it.
- DMA should avoid doing multiple accesses in a 64 byte block if possible, and instead use a single access. This can preserve the DRAM controller bandwidth (see DDR3/4/5 comments above), but also, L2/L3 cache designs may block any additional memory request targeting a memory block which is already under operation.
- When a DMA start a write burst, it has to complete as fast as possible. The reason is that the interconnect can lock itself on your burst until you finish it.
- When a DMA start a read burst, it should avoid putting backpressure on the read responses. The reason is that the interconnect can lock itself on your burst until you finish it.

PRIVILEGES

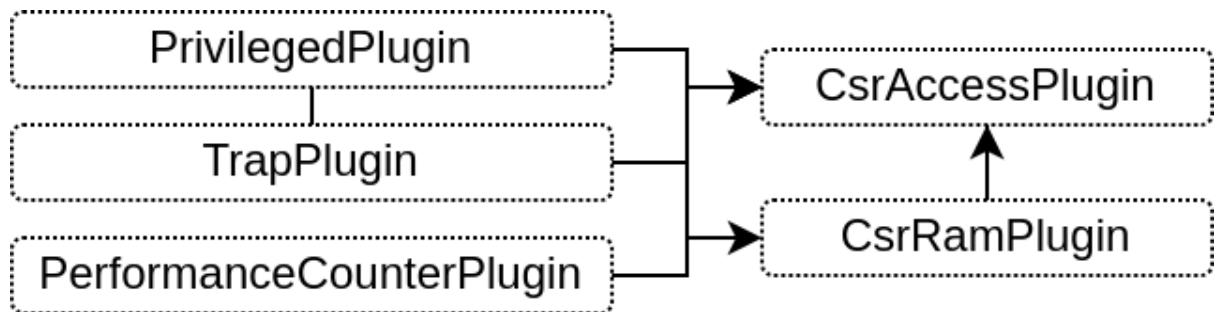
RISC-V specifies in "Volume 2, Privileged Specification" most of its special registers (CSR) which allows to handle things as :

- Traps (interrupts + exceptions)
- Memory protections (MMU, PMP)
- Privilege modes (Machine, Supervisor, User)

A microcontroller will typically only need Machine mode, maybe User mode, while a Linux capable CPU will normally need them all.

- Machine mode : Baremetal / Bootloader / BIOS / OpenSBI / RTOS
- Supervisor mode : Kernel / Linux
- User mode : Applications running on the top of linux

Those are handled in VexiiRiscv via a sets of plugins.



11.1 CsrAccessPlugin

- Implement the CSR read and write instruction in the execute pipeline
- Provide an API for other plugins to specify the mapping between the CSR registers and the CSR instruction

For instance, when another plugin want to add a custom CSR, it can do as follow :

```

class CustomPlugin() extends FiberPlugin {
    val logic = during setup new Area{
        val cp = host[CsrService] // CsrAccessPlugin is an implementation of CsrService
        val buildBefore = retains(cp.csrLock) // This ensure that the CsrService hold one lock
        →until we are finished with the API usages
        awaitBuild()

        // Define a few registers
        val regX = Reg(UInt(8 bits)) init(0)
        val regY = Reg(UInt(8 bits)) init(0)
    }
}
  
```

(continues on next page)

(continued from previous page)

```

// Map those registers in the RISC-V CSRs at address 0xFF0.
// - Bits 17:10 will be regX
// - Bits 27:20 will be regY
cp.readWrite(0xFF0, 10 -> regX, 20 -> regY)

// Now that we are with the csr API, we allows it to elaborate
buildBefore.release()
}
}

```

11.1.1 PrivilegedPlugin

- Implement the RISC-V privileged spec, mostly by using the CsrAccessPlugin API
- Use the CsrRamPlugin to implement various CSR as MVAL, MTVEC, MEPC, MSCRATCH, ...
- By default only the machine mode is enabled.
- You can use `--with-supervisor` and `--with-user`` to enable the corresponding privileged modes

11.1.2 CsrRamPlugin

- Provide an API which allows to statically allocate space on it
- Provide an API to create read / write ports on it
- Used by various plugins to store the CSR contents in a FPGA efficient way

11.1.3 TrapPlugin

- Implement the trap buffer / FSM
- The FSM implement the core logic of many special instructions (MRET, SRET, ECALL, EBREAK, FENCE.I, WFI, ...)
- Also allows the CPU pipeline to emit hardware traps to re-execute (REDO) the current instruction or to jump to the next one after a full pipeline flush (NEXT).
- the REDO hardware trap is used by I\$ D\$ miss, the DecodePlugin when it detect a illegal branch prediction
- the NEXT hardware trap is used by the CsrAccessPlugin when a state change require a full CPU flush

11.1.4 PerformanceCounterPlugin

Implement the privileged performance counters in a FPGA friendly way :

- Use the CsrRamPlugin to store 57 bits for each performance counter
- Use a dedicated 7 bits hardware register per counter
- Once that 7 bits register MSB is set, a FSM will flush it into the CsrRamPlugin
- By default, this plugin is disabled, to enable it, you can use, for instance, `--performance-counters 9`

11.1.5 EnvPlugin

- Implement a few instructions as MRET, SRET, ECALL, EBREAK, FENCE.I, WFI by producing hardware traps
- Those hardware trap are then handled in the TrapPlugin FSM

11.1.6 MmuPlugin

- Implements supervisor mode memory protections
- Include a hardware page walker
- Has a TLB to store the page walker results
- TLB are stored in multiples directly mapped memories. Typically 2 way x 32 TLB for 4KB pages + 1 way x 32 TLB for mega pages
- Map very well with FPGA which supports asynchronous read memory (LUT based RAM)
- Can be configured to work with synchronous read memory (block ram), but will likely be your critical path for timings

This plugin is enabled via `--with-mmu` or `--with-supervisor`

11.1.7 PmpPlugin

- Allows the machine mode to restrict memory accesses of the supervisor/user mode to specific ranges (Physical Memory Protection)
- Quite expensive in resources and timings.
- Support NAPOT (aligned power of 2 sized regions)
- Support TOR (unrestricted region address/size)
- You can disable the RISC-V TOR support to help with area and timings via `--pmp-tor-disable`
- You can set the granularity of the memory regions (to improve timings and area usage). This throw away some of the address LSB bits. By default, the granularity is 4KB (`--pmp-granularity=4096`). Minimum allowed is 4.
- By default, the PmpPlugin is disabled, but you can enable it via, for instance, `--pmp-size=4`, which will enable 4 hardware PMP registers, allowing you to set up to 4 memory regions.

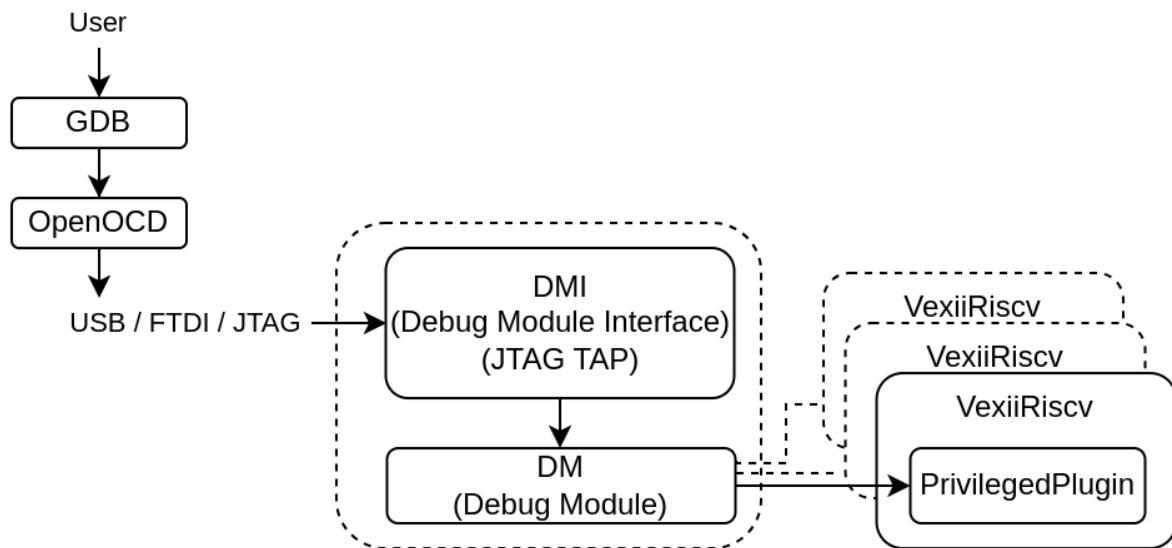
DEBUG SUPPORT

12.1 Architecture

VexiiRiscv support hardware debugging by implementing the official RISC-V debug spec.

- Compatible with OpenOCD (and maybe some other closed vendor, but untested)
- Can be used through a regular JTAG interface
- Can be used via tunneling through a single JTAG TAP instruction (FPGA native jtag interface)
- Support for some hardware trigger (PC, Load/Store address)

Here is a diagram of a typical debug setup :



The current implementation tends to provide the minimum required by the debug spec in order to reduce its area usage and complexity. It mostly work the following :

- The RISC-V debug module can push RISC-V instructions for the VexiiRiscv to execute.
- VexiiRiscv implement a custom CSR used by the debug module to read/write data of the CPU. This CSR doesn't behave like a regular register.
- The Debug Module can update the value of that special CSR, which can then be read by the CPU via *csrr* instructions
- When the CPU write into the CSR via a *csrw*, it will sent the written value to the Debug Module.

So let's say the debug module want to read some memory, here is what it will do :

- Push instructions to set one register in the register file (let's say x1) to the address whe want to read: *li x1, 0x12345678*
- Push a memory load instruction: *lw x1, 0(x1)*

- Push a instruction to write the readed value into the special CSR (0x7B4): *csrw 0x7B4, x1*. Writing this CSR will automatically push the value to the debug module
- Provide that value to the JTAG

If you run a simulation (for instance : *Run a simulation* with the --debug-jtag-tap argument), then you can connect to the simulated JTAG via openocd and its TCP remote_bitbang bridge as if it was real hardware:

```
openocd -f src/main/tcl/openocd/vexiiriscv_sim.tcl
```

But note that the speed will be quite low (as it is a hardware simulation)

12.2 EmbeddedRiscvJtag

EmbeddedRiscvJtag is a plugin which can be used to integrate the RISC-V debug module and its JTAG TAP directly inside the VexiiRiscv. This simplify its deployment, but can only be used in single core configs.

It is the plugin being used to implement the simulation jtag described in the previous chapter (--debug-jtag-tap)

CHAPTER
THIRTEEN

PERFORMANCE / AREA / FMAX

It is still very early in the development, but here are some metrics :

Name	Max IPC
Issue	2
Late ALU	2
BTB / RAS	512 / 4
GShare	4KB
Dhrystone/MHz	2.50
Coremark/MHz	5.24
EmBench	1.62

It is too early for area / fmax metric, there is a lot of design space exploration to do which will trade IPC against FMax / Area.

Here are a few synthesis results :

! Note !
Those results are with the best speed grade of each family
In practice, depending what board/FPGA you use, it is common for them to have worst red speed grade.
Also, concerning the area usage, those numbers are a bit inflated because :
- The SDC constraint stress the timings => Synthesis use more logic to improve the red timings
- The inputs/outputs of the design are serialized/deserialized (ff+logic cost) to red reduce the pin count

rv32i_noBypass ->
- 0.78 Dhrystone/MHz 0.60 Coremark/MHz
- Artix 7 -> 210 MHz 1182 LUT 1759 FF
- Cyclone V -> 159 MHz 1,015 ALMs
- Cyclone IV -> 130 MHz 1,987 LUT 2,017 FF
- Trion -> 94 MHz LUT 1847 FF 1990
- Titanium -> 320 MHz LUT 2005 FF 2030

rv32i ->
- 1.12 Dhrystone/MHz 0.87 Coremark/MHz
- Artix 7 -> 206 MHz 1413 LUT 1761 FF
- Cyclone V -> 138 MHz 1,244 ALMs
- Cyclone IV -> 124 MHz 2,188 LUT 2,019 FF
- Trion -> 78 MHz LUT 2252 FF 1962
- Titanium -> 300 MHz LUT 2347 FF 2000

rv64i ->
- 1.18 Dhrystone/MHz 0.77 Coremark/MHz

(continues on next page)

(continued from previous page)

```
- Artix 7    -> 186 MHz 2157 LUT 2332 FF
- Cyclone V  -> 117 MHz 1,760 ALMs
- Cyclone IV -> 113 MHz 3,432 LUT 2,770 FF
- Trion       -> 83 MHz LUT 3883   FF 2681
- Titanium    -> 278 MHz LUT 3909   FF 2783
```

`rv32im` ->

```
- 1.20 Dhystone/MHz 2.70 Coremark/MHz
- Artix 7    -> 190 MHz 1815 LUT 2078 FF
- Cyclone V  -> 131 MHz 1,474 ALMs
- Cyclone IV -> 125 MHz 2,781 LUT 2,266 FF
- Trion       -> 83 MHz LUT 2643   FF 2209
- Titanium    -> 324 MHz LUT 2685   FF 2279
```

`rv32im_branchPredict` ->

```
- 1.45 Dhystone/MHz 2.99 Coremark/MHz
- Artix 7    -> 195 MHz 2066 LUT 2438 FF
- Cyclone V  -> 136 MHz 1,648 ALMs
- Cyclone IV -> 117 MHz 3,093 LUT 2,597 FF
- Trion       -> 86 MHz LUT 2963   FF 2568
- Titanium    -> 327 MHz LUT 3015   FF 2636
```

`rv32im_branchPredict_cached8k8k` ->

```
- 1.45 Dhystone/MHz 2.97 Coremark/MHz
- Artix 7    -> 210 MHz 2721 LUT 3477 FF
- Cyclone V  -> 137 MHz 1,953 ALMs
- Cyclone IV -> 127 MHz 3,648 LUT 3,153 FF
- Trion       -> 93 MHz LUT 3388   FF 3204
- Titanium    -> 314 MHz LUT 3432   FF 3274
```

`rv32imasu_cached_branchPredict_cached8k8k_linux` ->

```
- 1.45 Dhystone/MHz 2.96 Coremark/MHz
- Artix 7    -> 199 MHz 3351 LUT 3833 FF
- Cyclone V  -> 131 MHz 2,612 ALMs
- Cyclone IV -> 109 MHz 4,909 LUT 3,897 FF
- Trion       -> 73 MHz LUT 4367   FF 3613
- Titanium    -> 270 MHz LUT 4409   FF 3724
```

`rv32im_branchPredictStressed_cached8k8k_ipcMax_lateAlu` ->

```
- 1.74 Dhystone/MHz 3.41 Coremark/MHz
- Artix 7    -> 140 MHz 3247 LUT 3755 FF
- Cyclone V  -> 99 MHz 2,477 ALMs
- Cyclone IV -> 85 MHz 4,835 LUT 3,765 FF
- Trion       -> 60 MHz LUT 4438   FF 3832
- Titanium    -> 228 MHz LUT 4459   FF 3963
```

13.1 Tuning

VexiiRiscv can scale a lot in function of its plugins/parameters. It can scale from simple microcontroller (ex M0) up to an application processor (A53).

On FPGA there is a few options which can be key in order to scale up the IPC while preserving the FMax :

- `--relaxed-btb` : When the BTB is enabled, by default it is implemented as a single cycle predictor, This can be easily be the first critical path to appear. This option make the BTB implementation spread over 2 cycles, which relax the timings at the cost of 1 cycle penalty on every successful branch predictions.
- `--relaxed-branch` : By default, the BranchPlugin will flush/setPc in the same stage than its own ALU. This is good for IPC but can easily be a critical path. This option will add one cycle latency between the ALU and the side effects (flush/setPc) in order to improve timings. If you enabled the branch prediction, then the impact on the IPC should be quite low.
- `--fma-reduced-accuracy` and `--fpu-ignore-subnormal` both reduce and can improve the fmax at the cost of accuracy
- `--fetch-fork-at=1` : If you don't have a instruction cache, this option will significantly relax the timings of the instruction fetch bus. With this option, instead of connecting the fetch bus to the first stage of the fetch pipeline (address generation stage), it will connect it to the stage after.
- `--lsu-fork-at=1 --lsu-pma-at` : If you don't have a data cache, this option will relax the LSU memory bus command channel by pushing everything down one stage before forking requests to the memory system.

13.2 Critical paths tool

At the end of your synthesis/place/route tools, you get a critical path report where hopefully, the source and destination registers are well named. The issue is that in between, all the combinatorial logic and signals names become unrecognizable or misleading most of the time. Also, in CPU design, it can quite easily happen that some combinatorial path "leak" through the pipeline, degrading the FMax quite a bit !

So there is a tool you can use in SpinalHDL to provide you a "clean" combinatorial path report between 2 signals of a design. Here is an example how you can use it in VexiiRiscv :

```
sbt "Test/runMain vexiiriscv.Generate --stressed-src --allow-bypass-from=0 --analyse-
  ↵path from=execute_ctrl2_up_integer_RS1_lane0,to=execute_ctrl1_down_integer_RS1_lane0
  ↵"
```

This will report you the various paths from `execute_ctrl2_up_integer_RS1_lane0` to `execute_ctrl1_down_integer_RS1_lane0`.

Here is one of the path reported, which goes through RS1 -> SrcPlugin -> BarrelShifterPlugin -> IntFormatPlugin -> WriteBackPlugin -> RS1 bypass -> RS1 :

```
- (toplevel/execute_ctrl2_up_integer_RS1_lane0 : Bits[32 bits])
- (Bits -> SInt of 32 bits)
- (toplevel/_zz_execute_ctrl2_down_early0_SrcPlugin_SRC1_lane0 : SInt[32 bits])
- (toplevel/execute_ctrl2_down_early0_SrcPlugin_SRC1_lane0 : SInt[32 bits])
- (Bool ? Bits | Bits)[32 bits]
- (toplevel/early0_BarrelShifterPlugin_logic_shift_reversed : SInt[32 bits])
- (SInt -> Bits of 32 bits)
- Bits ## Bits
- (Bits -> SInt of 33 bits)
- (SInt >> UInt)[33 bits]
- resize(SInt,32 bits)
- (toplevel/early0_BarrelShifterPlugin_logic_shift_shifted : SInt[32 bits])
- (Bool ? Bits | Bits)[32 bits]
```

(continues on next page)

(continued from previous page)

```

- (toplevel/early0_BarrelShifterPlugin_logic_shift_patched : SInt[32 bits])
- (SInt -> Bits of 32 bits)
- (toplevel/execute_ctrl2_down_early0_BarrelShifterPlugin_SHIFT_RESULT_lane0 : ↳
↳ Bits[32 bits])
- (toplevel/early0_BarrelShifterPlugin_logic_wb_payload : Bits[32 bits])
- (Bool ? Bits | Bits)[32 bits]
- (Bits | Bits)[32 bits]
- (toplevel/lane0_IntFormatPlugin_logic_stages_0_raw : Bits[32 bits])
- (toplevel/lane0_IntFormatPlugin_logic_stages_0_wb_payload : Bits[32 bits])
- (Bool ? Bits | Bits)[32 bits]
- (Bits | Bits)[32 bits]
- (toplevel/lane0_integer_WriteBackPlugin_logic_stages_0_muxed : Bits[32 bits])
- (toplevel/execute_ctrl2_lane0_integer_WriteBackPlugin_logic_DATA_lane0_bypass : ↳
↳ Bits[32 bits])
- (toplevel/execute_ctrl2_down_lane0_integer_WriteBackPlugin_logic_DATA_lane0 : ↳
↳ Bits[32 bits])
- (toplevel/_zz_execute_ctrl1_down_integer_RS1_lane0_1 : Bits[32 bits])
- (toplevel/execute_ctrl1_down_integer_RS1_lane0 : Bits[32 bits])

```

And there is the reported list of all named signal used by any of the paths :

```

- (toplevel/_zz_execute_ctrl1_down_integer_RS1_lane0_1 : Bits[32 bits])
- (toplevel/_zz_execute_ctrl2_down_early0_SrcPlugin_SRC1_lane0 : SInt[32 bits])
- (toplevel/early0_BarrelShifterPlugin_logic_shift_patched : SInt[32 bits])
- (toplevel/early0_BarrelShifterPlugin_logic_shift_reversed : SInt[32 bits])
- (toplevel/early0_BarrelShifterPlugin_logic_shift_shifted : SInt[32 bits])
- (toplevel/early0_BarrelShifterPlugin_logic_wb_payload : Bits[32 bits])
- (toplevel/early0_IntAluPlugin_logic_alu_bitwise : SInt[32 bits])
- (toplevel/early0_IntAluPlugin_logic_alu_result : SInt[32 bits])
- (toplevel/early0_IntAluPlugin_logic_wb_payload : Bits[32 bits])
- (toplevel/execute_ctrl1_down_integer_RS1_lane0 : Bits[32 bits])
- (toplevel/execute_ctrl2_down_early0_BarrelShifterPlugin_SHIFT_RESULT_lane0 : ↳
↳ Bits[32 bits])
- (toplevel/execute_ctrl2_down_early0_IntAluPlugin_ALU_RESULT_lane0 : Bits[32 bits])
- (toplevel/execute_ctrl2_down_early0_SrcPlugin_ADD_SUB_lane0 : SInt[32 bits])
- (toplevel/execute_ctrl2_down_early0_SrcPlugin_LESS_lane0 : Bool)
- (toplevel/execute_ctrl2_down_early0_SrcPlugin_SRC1_lane0 : SInt[32 bits])
- (toplevel/execute_ctrl2_down_lane0_integer_WriteBackPlugin_logic_DATA_lane0 : ↳
↳ Bits[32 bits])
- (toplevel/execute_ctrl2_lane0_integer_WriteBackPlugin_logic_DATA_lane0_bypass : ↳
↳ Bits[32 bits])
- (toplevel/execute_ctrl2_up_integer_RS1_lane0 : Bits[32 bits])
- (toplevel/lane0_IntFormatPlugin_logic_stages_0_raw : Bits[32 bits])
- (toplevel/lane0_IntFormatPlugin_logic_stages_0_wb_payload : Bits[32 bits])
- (toplevel/lane0_integer_WriteBackPlugin_logic_stages_0_muxed : Bits[32 bits])

```

CHAPTER
FOURTEEN

SOC

This is currently WIP.

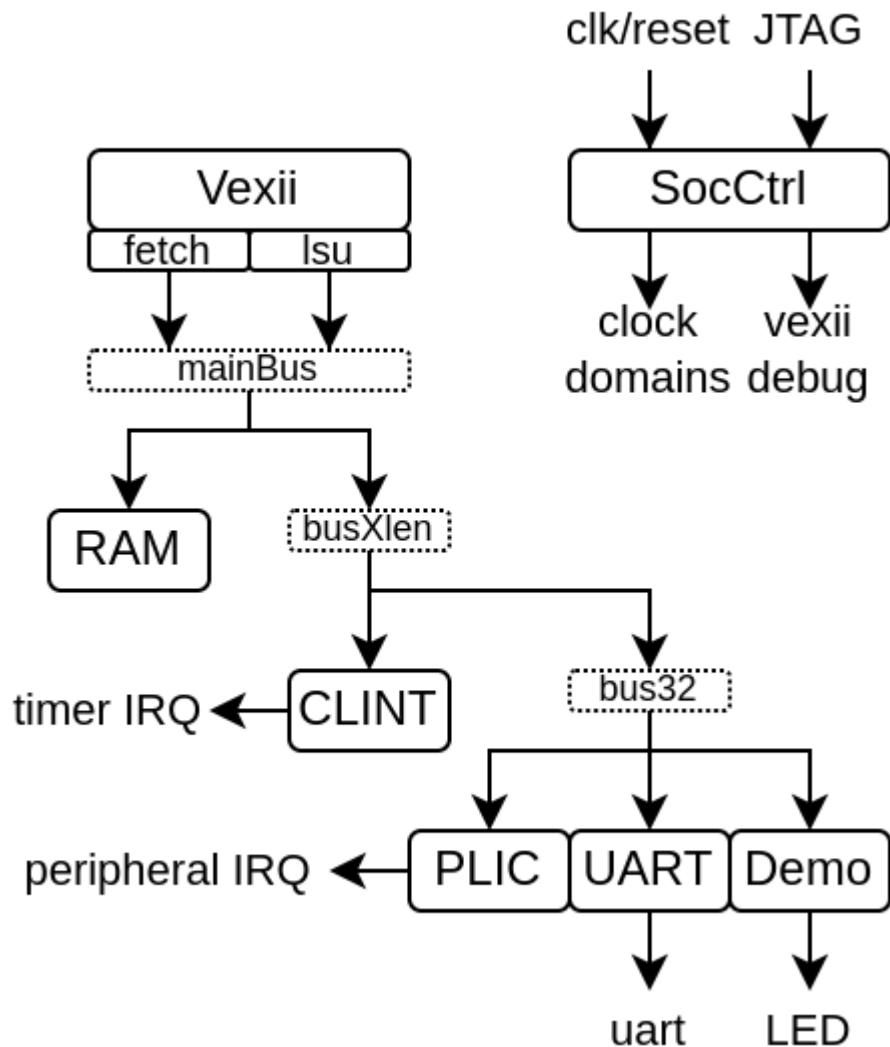
14.1 MicroSoc

MicroSoC is a little SoC based on VexiiRiscv and a tilelink interconnect.

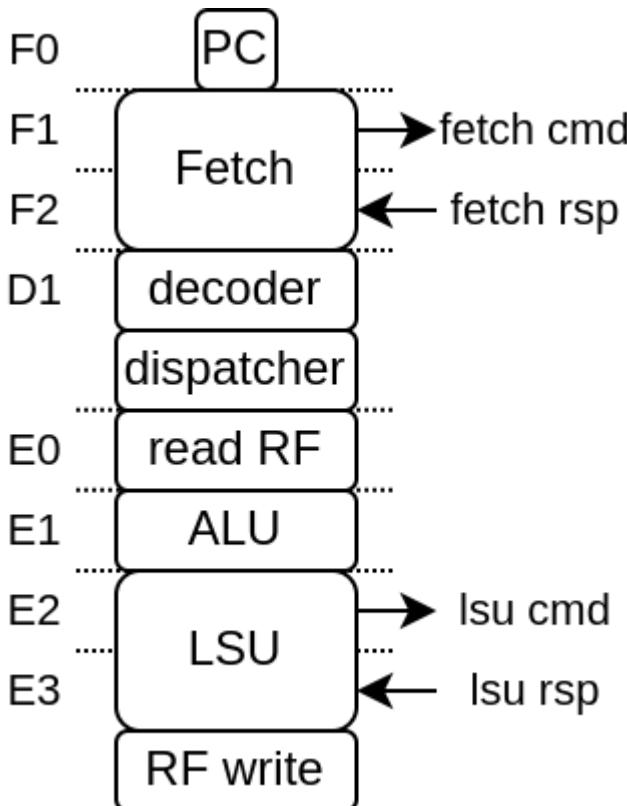
Its goals are :

- To provide a simple reference design
- To be a simple and light FPGA SoC
- Target a high frequency of operation, but not a high IPC (by default)

Here is a architecture diagram :



Here you can see the default vexiiriscv architecture for this SoC :



You can find its implementation here <https://github.com/SpinalHDL/VexiiRiscv/blob/dev/src/main/scala/vexiiriscv/soc/micro>

- MicroSoc.scala : Contains the SoC toplevel
- MicroSocGen.scala : Contains the scala main which can be used to generate the SoC verilog
- MicroSocSim.scala : Contains a simple SpinalSim testbench for the SoC

The MicroSoC code is commented in a way which should help non-initiated to understand what is happening. (this is an invitation to read the code ^^)

14.1.1 Verilog generation

To generate the SoC verilog, you can run :

```
# Default configuration
sbt "runMain vexiiriscv.soc.micro.MicroSocGen"
# SoC with 32 KB + RV32IMC running at 50 MHz:
sbt "runMain vexiiriscv.soc.micro.MicroSocGen --ram-bytes=32768 --with-rvm --with-rvc
    --system-frequency=50000000"
# List all the parameters available
sbt "runMain vexiiriscv.soc.micro.MicroSocGen --help"
```

14.1.2 Simulation (SpinalSim / Verilator)

If you have Verilator installed, you can run a simulation by doing :

```
# Default configuration
sbt "runMain vexiiriscv.soc.micro.MicroSocSim"
# List all the parameters available
sbt "runMain vexiiriscv.soc.micro.MicroSocSim --help"
```

Here is a set of important command line arguments :

Table 1: Arguments

Command	Description
--load-elf ELF_FILE	Will load elf file into the ram/rom/flash of the SoC
--trace-fst	A FST wave of all the DUT signals will be stored in sim-Workspace/MicroSocSim/test (you can open it using GTKwave)
--trace-konata	A konata trace of all the executed instruction will be stored in sim-Workspace/MicroSocSim/test (you can open it using https://github.com/shiroyadan/Konata)

Note that the default VexiiRiscv configuration is RV32I, with a relatively low area/performance. You can for instance get more performance by adding `--allow-bypass-from=0 --with-rvm --with-bitb --with-ras --with-gshare`

While the simulation is running you can connect to it using openocd as if it was real hardware :

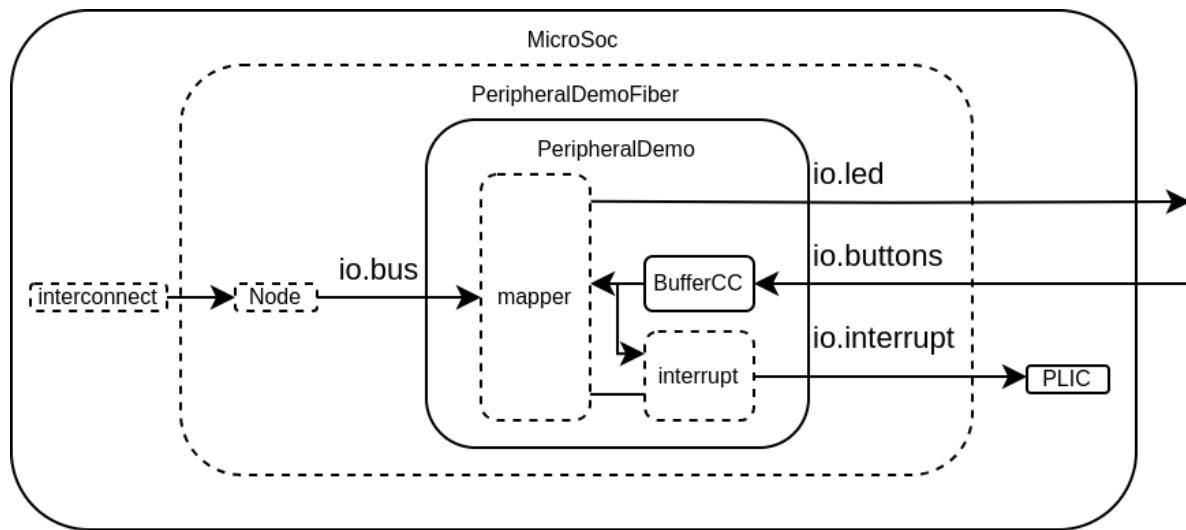
```
openocd -f src/main/tcl/openocd/vexiiriscv_sim.tcl
```

14.1.3 Adding a custom peripheral

Let's say you want to design a peripheral and then add it to the SoC, the MicroSoc contains one example of that via PeripheralDemo.scala. Take a look at it, its code is extensively commented :

<https://github.com/SpinalHDL/VexiiRiscv/blob/dev/src/main/scala/vexiiriscv/soc/micro/PeripheralDemo.scala>

This peripheral example is a very simple one which provide the CPU access to leds, buttons and an interrupt function of the buttons value.



You can see in the diagram above :

- **PeripheralDemo** : Which is our custom peripheral in its traditional sense (a hardware Component / Module). It use regular SpinalHDL stuff.

- mapper : This is a tool which ease the creation of peripherals register file. Instead of having stuff like big switch case on the bus address, you just need to say "Create a RW register at this address" in a more natural language.
- BufferCC : Used to avoid metastability when we use the buttons value in our hardware (this is a chain of 2 flip-flop)
- PeripheralDemoFiber : This is sort of the integration layer for our PeripheralDemo into a SoC. This serve a few purposes. It handle the Tilelink parameters negotiation / propagation, as well as exporting the leds and buttons directly to the MicroSoc io.
- Node : This is an instance of the tilelink bus in our SoC. It is used for parameter negotiation/propagation as well as to get the hardware bus instance.

You can then add that peripheral in the toplevel around the other peripherals by :

```
val demo = new PeripheralDemoFiber(new PeripheralDemoParam(12, 16))
demo.node at 0x10003000 of bus32
PLIC.mapUpInterrupt(3, demo.interrupt)
```

This peripheral is already integrated into MicroSoC as a demo but disabled by default. To enable it, will need to provide a specific command line parameter. For instance :

```
sbt "runMain vexiiriscv.soc.micro.MicroSocSim --demo-peripheral leds=16,buttons=12"
```

14.1.4 Exporting an APB3 bus to the toplevel

Let's say you want to allow the CPU to access a APB3 peripheral which stand outside the SoC toplevel. Here is how you can do so by adding code to the MicroSoc.system.peripheral area :

```
class MicroSoc(p : MicroSocParam) extends Component {
  ...
  val system = new ClockingArea(socCtrl.system.cd) {
    ...
    val peripheral = new Area {
      ...
      // Let's define a namespace to contains all our logic
      val exported = new Area {
        ...
        // Let's define tl as our Tilelink peripheral endpoint (before the APB3
        // bridge)
        val tl = tilelink.fabric.Node.slave()
        tl at 0x10006000 of bus32 // Lets map our tilelink bus in the memory space

        // Let's define our APB3 bus which will be exposed to the IO of the SoC
        val bus = master(APB3(addressWidth = 12, dataWidth = 32))

        ...
        // Let's define a Fiber thread which will
        // - Handle the tilelink parameter negotiation
        // - Instantiate the APB3 bridge and connect the buses
        val fiber = Fiber build new Area{
          ...
          // Here we go with the tilelink negotiation
          tl.m2s.supported.load(
            M2sSupport(
              addressWidth = bus.config.addressWidth,
              dataWidth = bus.config.dataWidth,
              transfers = M2sTransfers(
                get = tilelink.SizeTypeRange(4),
                putFull = tilelink.SizeTypeRange(4)
              )
            )
          )
        }
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

If you want the CPU to be able to execute code located in the APB3 peripheral, then you will need to tag the tl bus with :

```
val tl  = tilelink.fabric.Node.slave()
tl at 0x10006000 of bus32 // Lets map our tilelink bus in the memory space
tl.addTag(spinal.lib.system.tag.PMA.EXECUTABLE)
```

14.1.5 Adding a custom instruction

Let's say you want to add a custom instruction to the MicroSoc. Let's use the *Plugin implementation* which does SIMD add.

In the MicroSoc, you can find :

```
val cpu = new TilelinkVexiiRiscvFiber(p.vexii.plugins())
```

We need to edit this into :

```
// Instantiate all the plugins from the command line arguments
val pluginsArea = p.vexii.pluginsArea()
// Add our custom plugin, pluginsArea.early0 refer to the default execute lane of the
// CPU
pluginsArea.plugins += new vexiiriscv.execute.SimdAddPlugin(pluginsArea.early0)
// Build the CPU
val cpu = new TilelinkVexiiRiscvFiber(pluginsArea.plugins)
```

TODO add software example

14.2 Litex

VexiiRiscv can also be deployed using Litex.

You can find some fully self contained example about how to generate the software and hardware files to run buildroot and debian here :

- <https://github.com/SpinalHDL/VexiiRiscv/tree/dev/doc/litex>

For instance, you can run the following litex command to generate a linux capable SoC on the digilent_nexys_video dev kit (RV32IMA):

```
python3 -m litex_boards.targets.digilent_nexys_video --cpu-type=vxiiriscv --cpu-variant=linux --cpu-count=1 --build --load
```

Here is an example for a dual core, debian capable (RV64GC) with L2 cache and a few other peripherals :

```
python3 -m litex_boards.targets.digilent_nexys_video --cpu-type=vexiiriscv --cpu-
→variant=debian --cpu-count=2 --with-video-framebuffer --with-sdcard --with-
→ethernet --with-coherent-dma --l2-byte=262144 --build --load
```

Additional arguments can be provided to customize the VexiiRiscv configuration, for instance the following will enable the PMU, 0 cycle latency register file, multiple outstanding D\$ refill/writeback and store buffer:

```
--vexii-args="--performance-counters 9 --regfile-async --lsu-l1-refill-count 2 --lsu-
→l1-writeback-count 2 --lsu-l1-store-buffer-ops=32 --lsu-l1-store-buffer-slots=2"
```

To generate a DTS, I recommend adding `--soc-json build/csr.json` to the command line, and then running :

```
python3 -m litex.tools.litex_json2dts_linux build/csr.json > build/linux.dts
```

That linux.dts will miss the CLINT definition (used by opensbi), so you need to patch in (in the soc region, for instance for a quad core) :

```
clint@f0010000 {
    compatible = "riscv,clint0";
    interrupts-extended = <
        &L0 3 &L0 7
        &L1 3 &L1 7
        &L2 3 &L2 7
        &L3 3 &L3 7>;
    reg = <0xf0010000 0x10000>;
};
```

Then you can convert the linux.dts into linux.dtb via :

```
dtc -O dtb -o build/linux.dtb build/linux.dts
```

To run debian, you would need to change the dts boot device to your block device, as well as removing the initrd from the dts. You can find more information about how to setup the debian images on https://github.com/SpinalHDL/NaxSoftware/tree/main/debian_litex

But note that for opensbi, use instead the following (official upstream opensbi using the generic platform, which will also contains the dtb):

```
git clone https://github.com/riscv-software-src/opensbi.git
cd opensbi
make CROSS_COMPILE=riscv-none-embed- \
PLATFORM=generic \
FW_FDT_PATH=../build/linux.dtb \
FW_JUMP_ADDR=0x41000000 \
FW_JUMP_FDT_ADDR=0x46000000
```