VexiiRiscv Documentation

VexiiRiscv contributors

Jan 22, 2024

CONTENTS

Welcome to VexiiRiscv's documentation!

CHAPTER

ONE

INTRODUCTION

1.1 About VexiiRiscv

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

- RISCV 32/64 bits IMAFDC
- Could start around as small as VexRiscv, but could scale further in performance
- Optional late-alu
- Optional multi issue
- Optional multi threading
- Providing a cleaner implementation, getting ride of the technical debt, especially the frontend
- Proper branch prediction
- …

On this date (22/01/2024) the status is :

- rv 32/64 imacsu supported
- Can run baremetal benchmarks (2.24 dhrystone/mhz, 4.66 coremark/mhz)
- single/dual issue supported
- late-alu supported
- BTB/RAS/GShare branch prediction supported
- MMU SV32/SV39 supported
- can run linux/buildroot in simulation

1.2 Navigating the code

Here are a few key / typical code examples :

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

CHAPTER

TWO

FRAMEWORK

2.1 Dependencies

VexRiscv is based on a few tools / API

- 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
- Fiber : Which allows to define elaboration threads 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 Quine McCluskey to generate logic decoders from the elaboration time specifications

2.2 Scala / SpinalHDL

This combination alows to goes way behond what regular HDL alows in terms of hardware description capabilities. You can find some documentation about SpinalHDL here :

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

2.3 Plugin

One main design aspect of VexiiRiscv is that all its hardware is defined inside plugins. When you want to instanciate 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.

So it is quite different from the regular HDL component/module paradigm. Here are the adventages of this aproache :

- 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 decentralised by nature, you don't have a fat toplevel of doom, software interface between plugins can be used to negociate things durring elaboration time.

The plugins can fork elaboration threads which cover 2 phases :

- setup phase : where plugins can aquire elaboration locks on each others

- build phase : where plugins can negociate between each others and generate hardware

2.3.1 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
```

2.3.2 Negociation example

Here is a example where there a plugin which count the number of hardware event comming 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 lock = Retainer() // Will allow other plugins to block the elaboration of "logic
↪" thread
  val events = ArrayBuffer[Bool]() // Will allow other plugins to add event sources
  val logic = during build new Area{
    lock.await() // Active blocking
```

(continues on next page)

(continued from previous page)

```
    val counter = Reg(UInt(32 bits)) init(0)
    counter := counter + CountOne(events)
  }
}


//For the demo we want to be able to instanciate this plugin multiple times, so we␣
↪add a prefix parameter
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, and so lock some other plugins retainers)
  val logic = during setup new Area{
    val ecp = host[EventCounterPlugin] // Search for the single instance of␣
↪EventCounterPlugin in the plugin pool
    // 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,
  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_
```

(continues on next page)

(continued from previous page)

```
↪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


endmodule
```

2.4 Database

Quite a few things behave kinda like variable specific for each VexiiRiscv instance. For instance XLEN, PC_WIDTH, INSTRUCTION_WIDTH, . . .

So they are end up with things that we would like to share between plugins of a given VexiiRiscv instance with the minimum code possible to keep things slim. For that, a "database" was added. 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.blocking
import vexiiriscv._
import scala.collection.mutable.ArrayBuffer

object Global extends AreaObject{
  val VIRTUAL_WIDTH = blocking[Int] // If accessed while before being set, it will␣
↪actively block (until set by another thread)
}

class LoadStorePlugin extends FiberPlugin{
  val logic = during build new Area{
    val register = Reg(UInt(Global.VIRTUAL_WIDTH bits))
  }
```

(continues on next page)

(continued from previous page)

```
}

class MmuPlugin extends FiberPlugin{
  val logic = during build new Area{
    Global.VIRTUAL_WIDTH.set(39)
  }
}

object Gen extends App{
  SpinalVerilog{
    val plugins = ArrayBuffer[FiberPlugin]()
    plugins += new LoadStorePlugin()
    plugins += new MmuPlugin()
    VexiiRiscv(plugins)
  }
}
```

2.5 Pipeline API

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

- Allow moving things around with no paine (retiming)
- Reduce boiler plate code

More documentation about it in https://github.com/SpinalHDL/SpinalDoc-RTD/pull/226

CHAPTER

THREE

FETCH

A few plugins operate in the fetch stage :

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

3.1 FetchPipelinePlugin

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

3.2 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.

3.3 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

3.4 BtbPlugin

See more in the Branch prediction chapter

3.5 GSharePlugin

See more in the Branch prediction chapter

3.6 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.

CHAPTER

FOUR

DECODE

A few plugins operate in the fetch stage :

- DecodePipelinePlugin
- AlignerPlugin
- DecoderPlugin
- DispatchPlugin
- DecodePredictionPlugin

4.1 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.

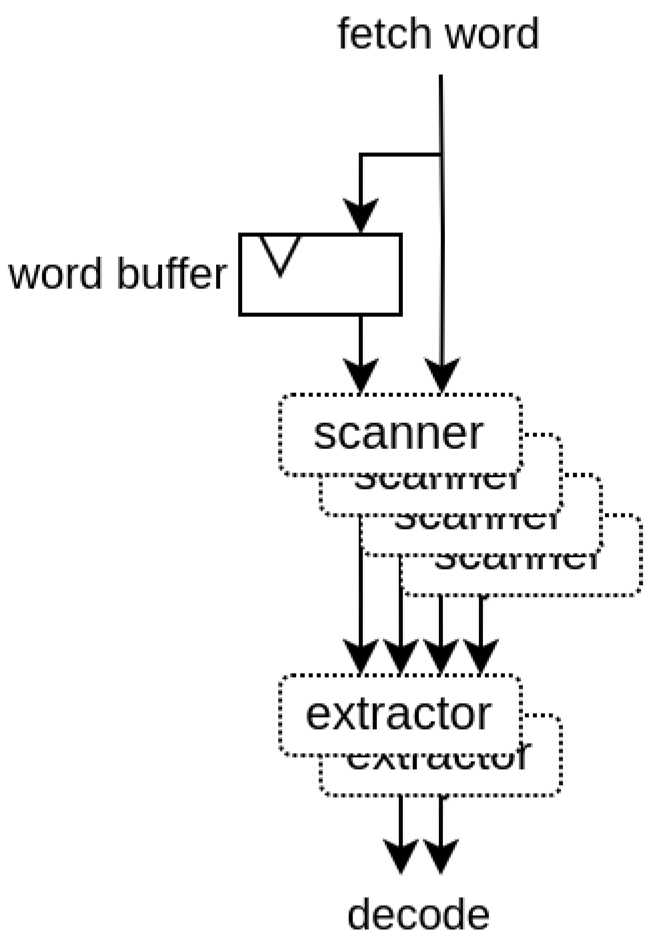
4.2 AlignerPlugin

Decode the words froms 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 (falsly) 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 feeded into the decoder pipeline.



4.3 DecoderPlugin

Will :

- Decode instruction
- Generate ilegal instruction exception
- Generate "interrupt" instruction

4.4 DecodePredictionPlugin

The purpose of this plugin is to ensure that no branch/jump prediction was made for non branch/jump instructions. In case this is detected, the plugin will just flush the pipeline and set the fetch PC to redo everything, but this time with a "first prediction skip"

See more in the Branch prediction chapter

4.5 DispatchPlugin

Will :

- Collect instruction from the end of the decode pipeline
- Try to dispatch them ASAP on the multiple "layers" available

Here is a few explenation 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 static a scheduling priority

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

CHAPTER

FIVE

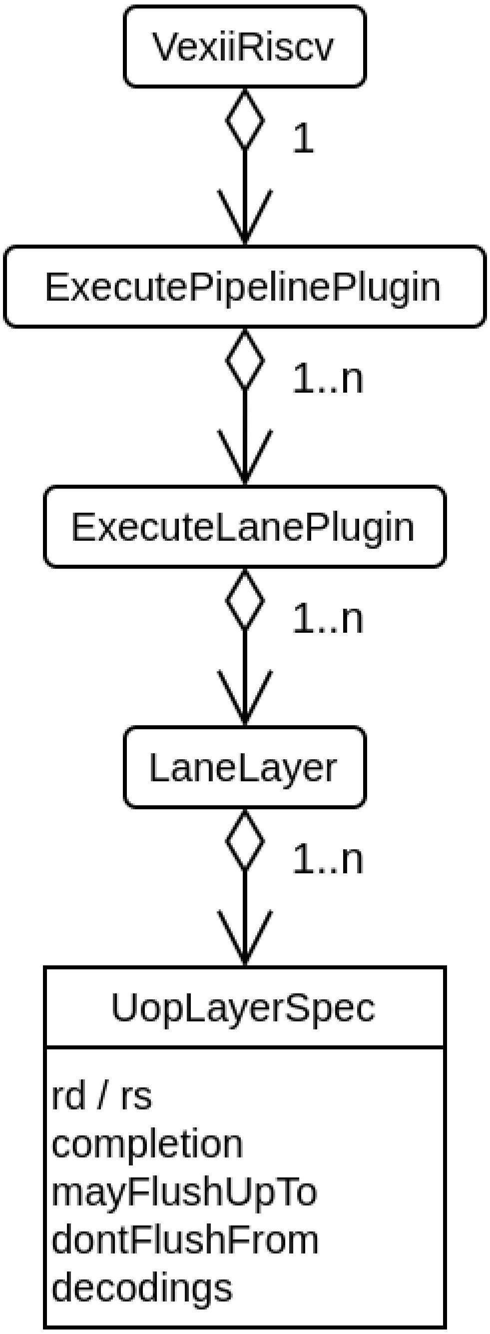
EXECUTE

5.1 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 retreive the result of the instruction (rd)
- From which point in the pipeline it use which register file (rs)
- From which point in the pipleine 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.

5.2 Plugins

5.2.1 infrastructures

Many plugins operate in the fetch stage. Some provide infrastructures :

ExecutePipelinePlugin

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

- It is based on the spinal.lib.misc.pipeline API and can host multiple "lanes" in it.
- For flow control, the lanes can only freeze the whole pipeline
- The pipeline do not collapse bubbles (empty stages)

ExecuteLanePlugin

Implement an execution lane in the ExecutePipelinePlugin

RegFilePlugin

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

SrcPlugin

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

RsUnsignedPlugin

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

IntFormatPlugin

Alows plugins to write integer values back to the register file through a optional sign extender. It uses WriteBack-Plugin as value backend.

WriteBackPlugin

Used by plugins to provide the RD value to write back to 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.

5.2.2 Instructions

Some implement regular 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 optionaly extends the last stage for one cycle in order to buffer the MULH bits

DivPlugin

- Implement the division/remain
- 2 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 backpresure by using a little fifo on the response data

5.2.3 Special

Some implement CSR, privileges and special instructions

CsrAccessPlugin

- Implement the CSR instruction
- Provide an API for other plugins to specify its hardware mapping

CsrRamPlugin

- Implement a shared on chip ram
- Provide an API which allows to staticaly 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

PrivilegedPlugin

- Implement the RISCV privileged spec
- Implement the trap buffer / FSM
- Use the CsrRamPlugin to implement various CSR as MTVAL, MTVEC, MEPC, MSCRATCH, …

PerformanceCounterPlugin

- Implement the privileged performance counters in a very FPGA way
- Use the CsrRamPlugin to store most of the counter bits
- 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

EnvPlugin

- Implement a few instructions as MRET, SRET, ECALL, EBREAK

5.3 Custom instruction

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

5.3.1 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

```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
//       |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 wll 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)
```

(continues on next page)

(continued from previous page)

```
    //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)
      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 MultiPortWritesSymplifier)
  val report = sc.generateVerilog {
    val pa = param.pluginsArea()
```

(continues on next page)

(continued from previous page)

```
    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)

    //Print result value
    li x4, PUT_HEX
    sw x3, 0(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 parm :\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.

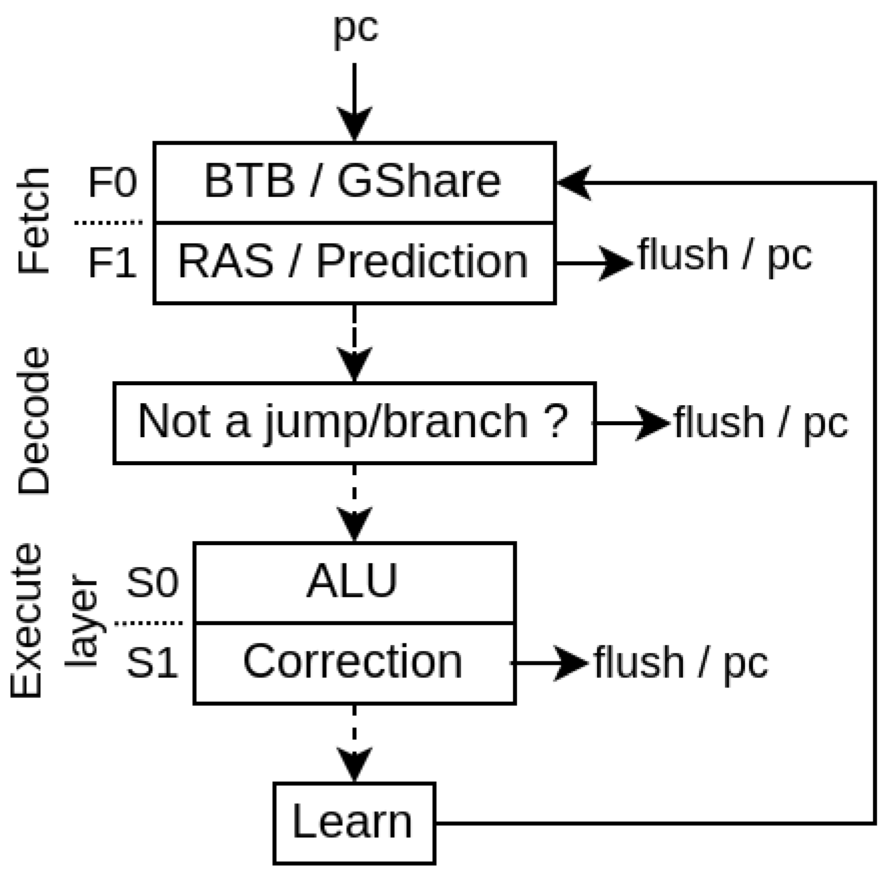
CHAPTER SIX

BRANCH PREDICTION

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 eventualy 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.

6.1 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
- 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 missprediction. To avoid write to read hazard, the fetch stage is blocked when it learn.
- Implement "ways" named chunks which are staticaly assigned to groups of word's slices, allowing to predict multiple branch/jump present in the same word

6.2 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

6.3 DecodePredictionPlugin

The purpose of this plugin is to ensure that no branch/jump prediction was made for non branch/jump instructions. In case this is detected, the plugin will just flush the pipeline and set the fetch PC to redo everything, but this time with a "first prediction skip"

6.4 BranchPlugin

Placed in the execute pipeline, it will ensure that the branch prediction was correct, else it correct it. It also generate a learn interface.

6.5 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.

CHAPTER

SEVEN

HOW TO USE

7.1 Dependencies

On debian :

```
# JAVA JDK
sudo add-apt-repository -y ppa:openjdk-r/ppa
sudo apt-get update
sudo apt-get install openjdk-19-jdk -y # You don't exactly need that version
sudo update-alternatives --config java
sudo update-alternatives --config javac

# Install SBT - https://www.scala-sbt.org/
echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" | sudo tee /etc/apt/
↪sources.list.d/sbt.list
echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee /etc/apt/sources.
↪list.d/sbt_old.list
curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&
↪search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add
sudo apt-get update
sudo apt-get install sbt

# Verilator (optional, for simulations)
sudo apt-get install git make autoconf g++ flex bison
git clone http://git.veripool.org/git/verilator   # Only first time
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

# Getting a RISC-V toolchain (optional)
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

# RVLS / Spike dependencies
sudo apt-get install device-tree-compiler libboost-all-dev
```

(continues on next page)

(continued from previous page)

```
# 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
```

7.2 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 ../..
```

7.3 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
 --with-user
 --without-mul
 --without-div
 --with-mul
```

(continues on next page)

(continued from previous page)

```
--with-div
--with-gshare
--with-btb
--with-ras
--with-late-alu
--regfile-async
--regfile-sync
--allow-bypass-from <value>
--performance-counters <value>
--with-fetch-l1
...
```

7.4 Run a simulation

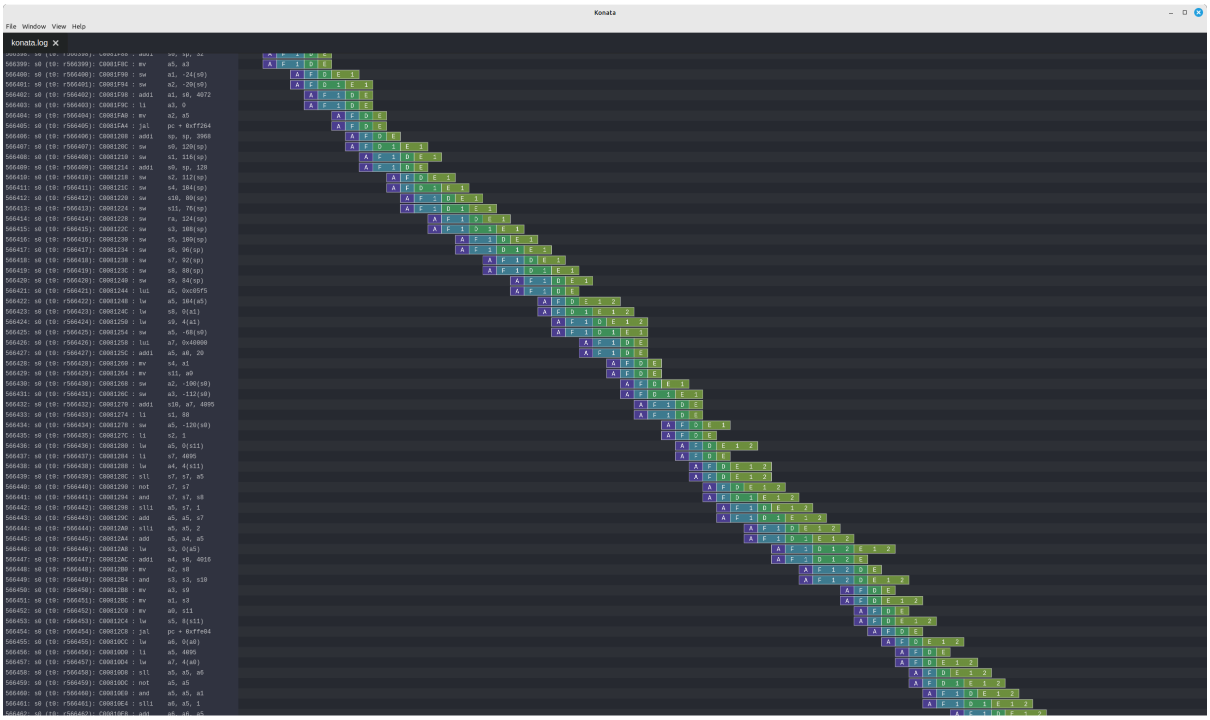
Note that Vexiiriscv use mostly an opt-in configuration. So, most performance related configuration are disabled by default.

```
sbt
compile
Test/runMain vexiiriscv.tester.TestBench --load-elf ext/NaxSoftware/baremetal/
↪dhrystone/build/rv32ima/dhrystone.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 behaviour of the CPU
- spike.log : The execution logs of Spike (golden model)
- tracer.log : The execution logs of VexRiscv (Simulation model)

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



CHAPTER

EIGHT

PERFORMANCE / AREA / FMAX

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

Note those are done without data cache. A data cache would likely improve the performance quite a bit by allowing more speculative load/store (not talking about data miss / hit, but realy execution hazard / interlock)

Name	Max IPC
Issue	2
Late ALU	2
BTB / RAS	512 / 4
GShare	4KB
Dhrystone/MHz	2.24
Coremark/MHz	4.66
EmBench	1.47

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.