VexiiRiscv Documentation

VexiiRiscv contributors

Nov 04, 2024

CONTENTS

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 :

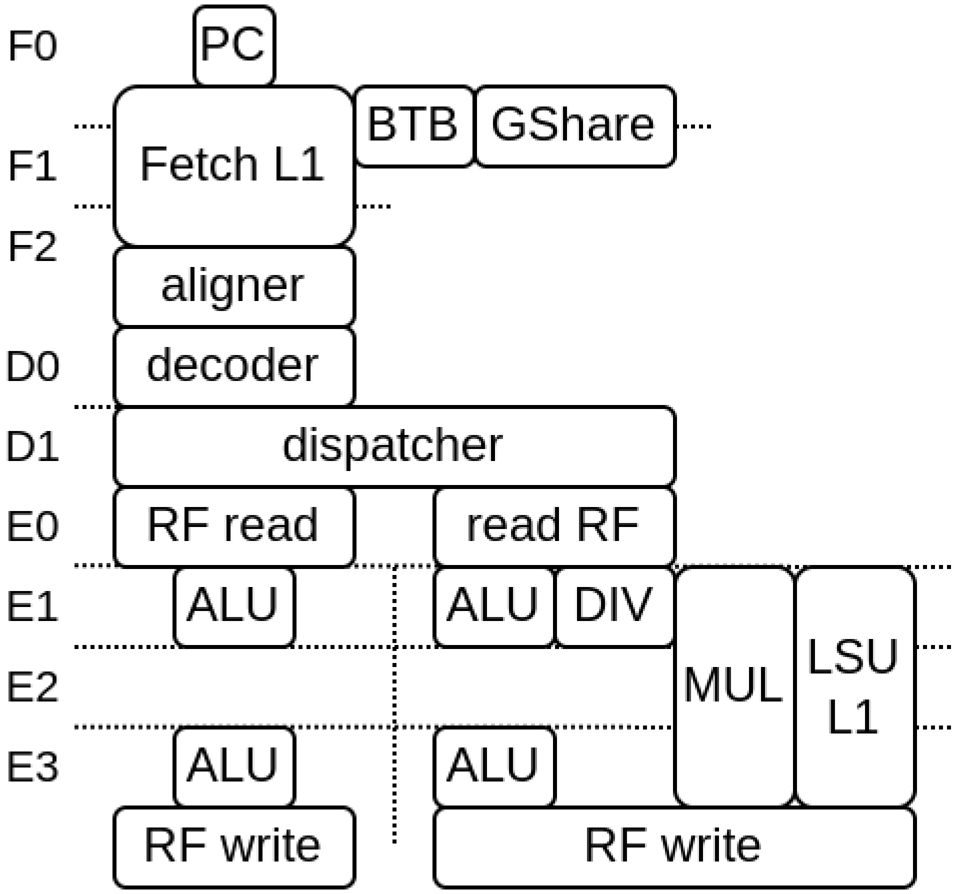
- To implement RISC-V 32/64 bits IMAFDCSU
- Could start around as small as VexRiscv, but could scale further in performance
- Optional late-alu
- Optional multi issue
- Providing a cleaner implementation, getting ride of the technical debt, especially the frontend
- Scale well at higher frequencies via 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 dhrystone/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 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

1.5 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)

CHAPTER

TWO

FRAMEWORK

2.1 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

2.2 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:

```scala
// 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

2.3 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

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 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


endmodule
```

2.4 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 it own dedicated VIRTUAL_WIDTH.get value, while using the same VIRTUAL_WIDTH key to access it.

2.5 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

```scala
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 :

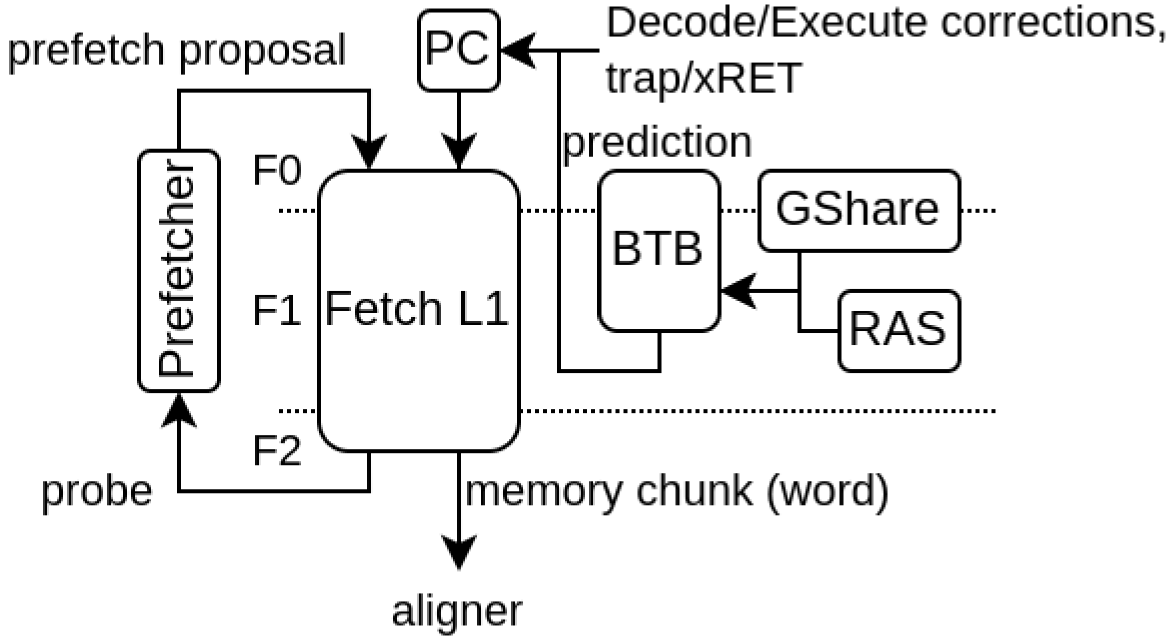
- https://spinalhdl.github.io/SpinalDoc-RTD/master/SpinalHDL/Libraries/Pipeline/index.html

CHAPTER THREE

FETCH

The goal of the fetch pipeline is to provide the CPU with a stream of words in which the instructions to execute are presents. So more precisely, the fetch pipeline doesn't realy 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 aswell as a prefetcher.



A few plugins operate in the fetch stage :

- FetchPipelinePlugin
- PcPlugin
- FetchCachelessPlugin
- FetchL1Plugin
- 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

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

- forkAt = 1
- joinAt = 2

3.4 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

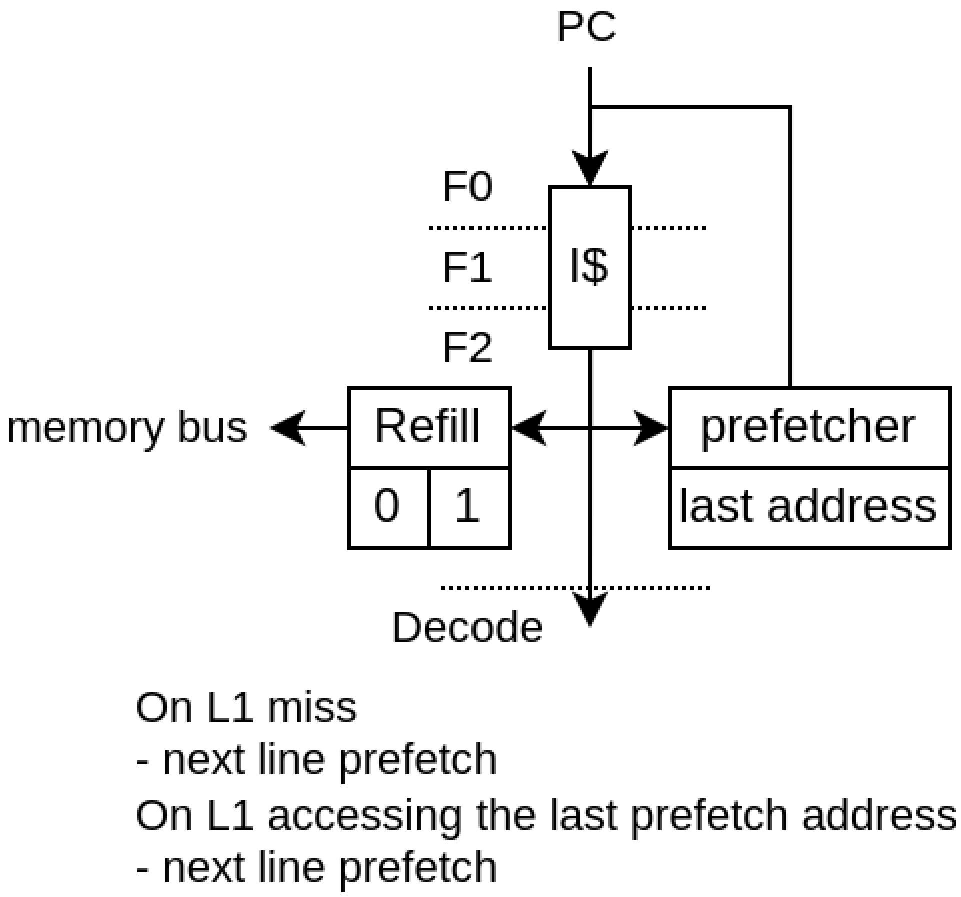
3.5 PrefetcherNextLinePlugin

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.



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.

3.6 BtbPlugin

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

3.7 GSharePlugin

See more in the Branch prediction chapter

3.8 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.

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.

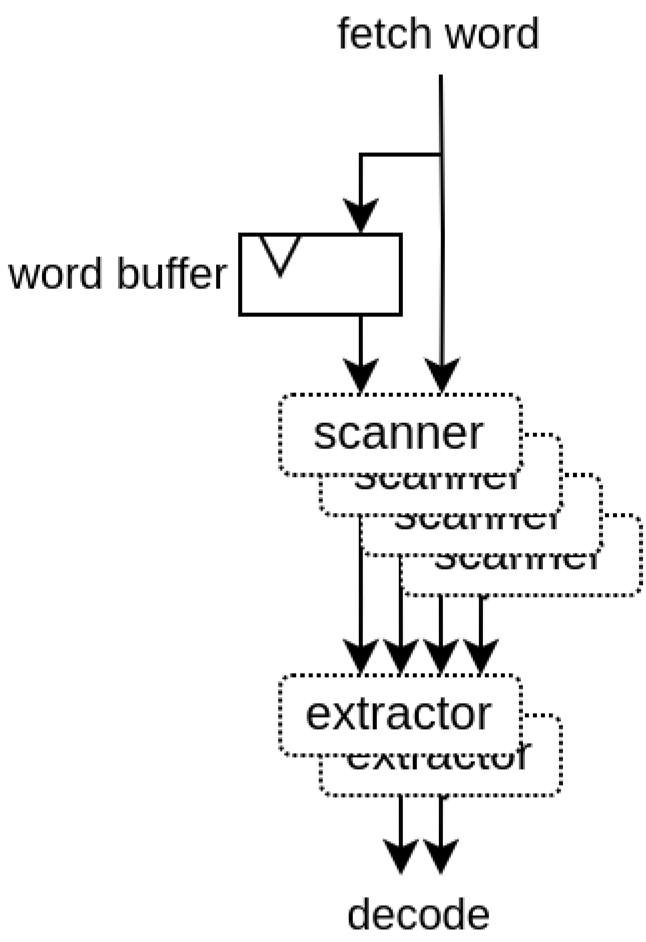
4.2 AlignerPlugin

Decode the words 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.



4.3 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)

4.4 DispatchPlugin

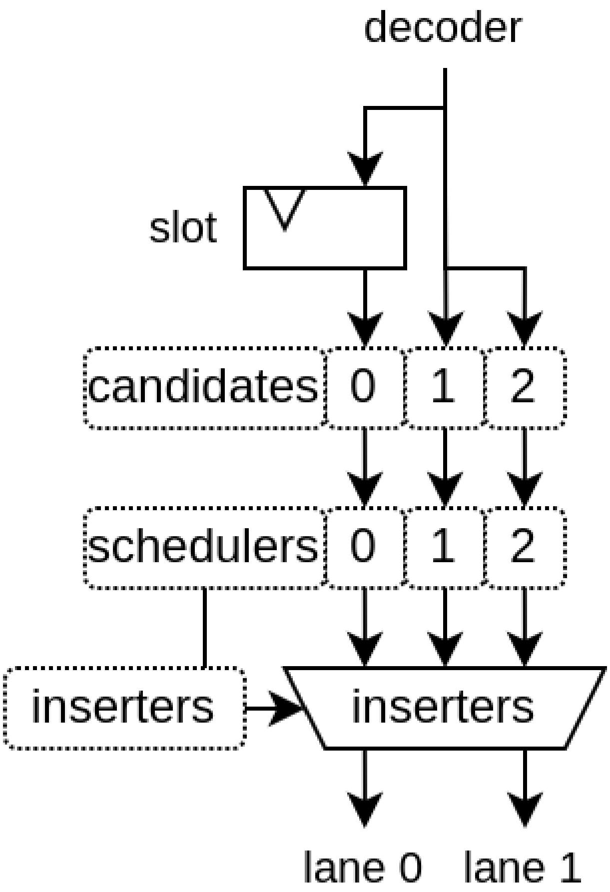
This is probably the hardest part of the VexiiRiscv hardware description to read, as it does a lot of elaboration time computing in order to figure out what hardware need to be generated.

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.

4.4.1 Architecture

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



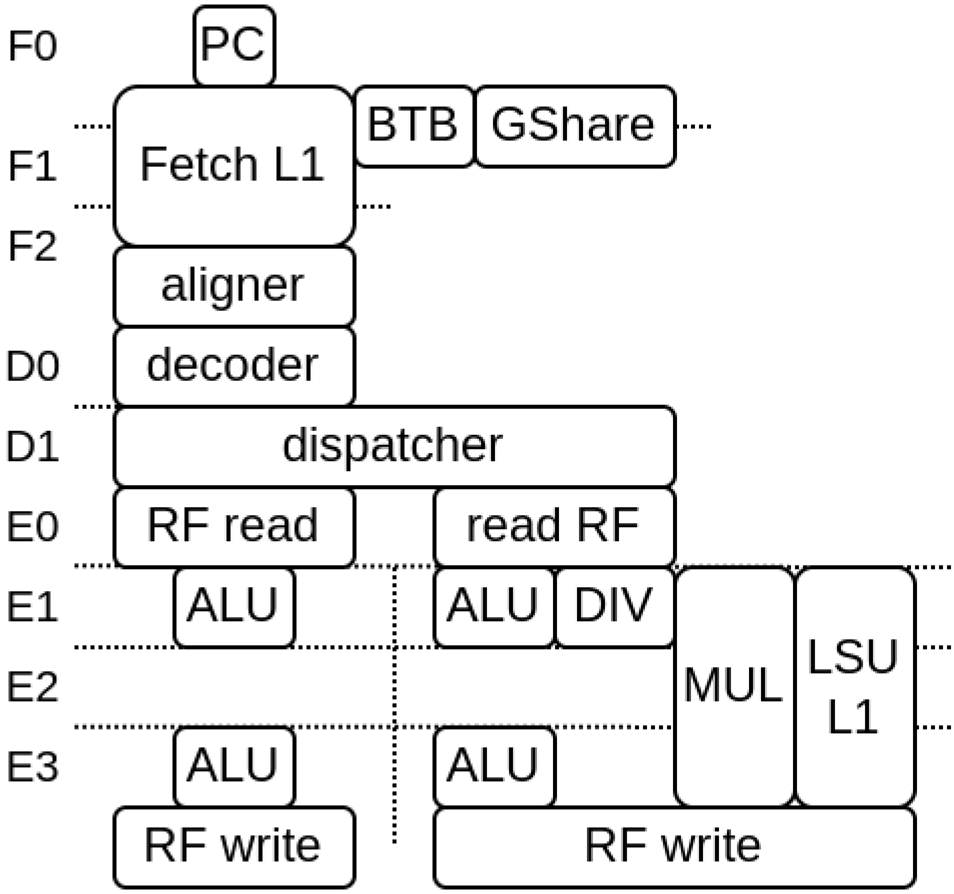
Here is a few explanation about execute lanes and layers :

- 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.



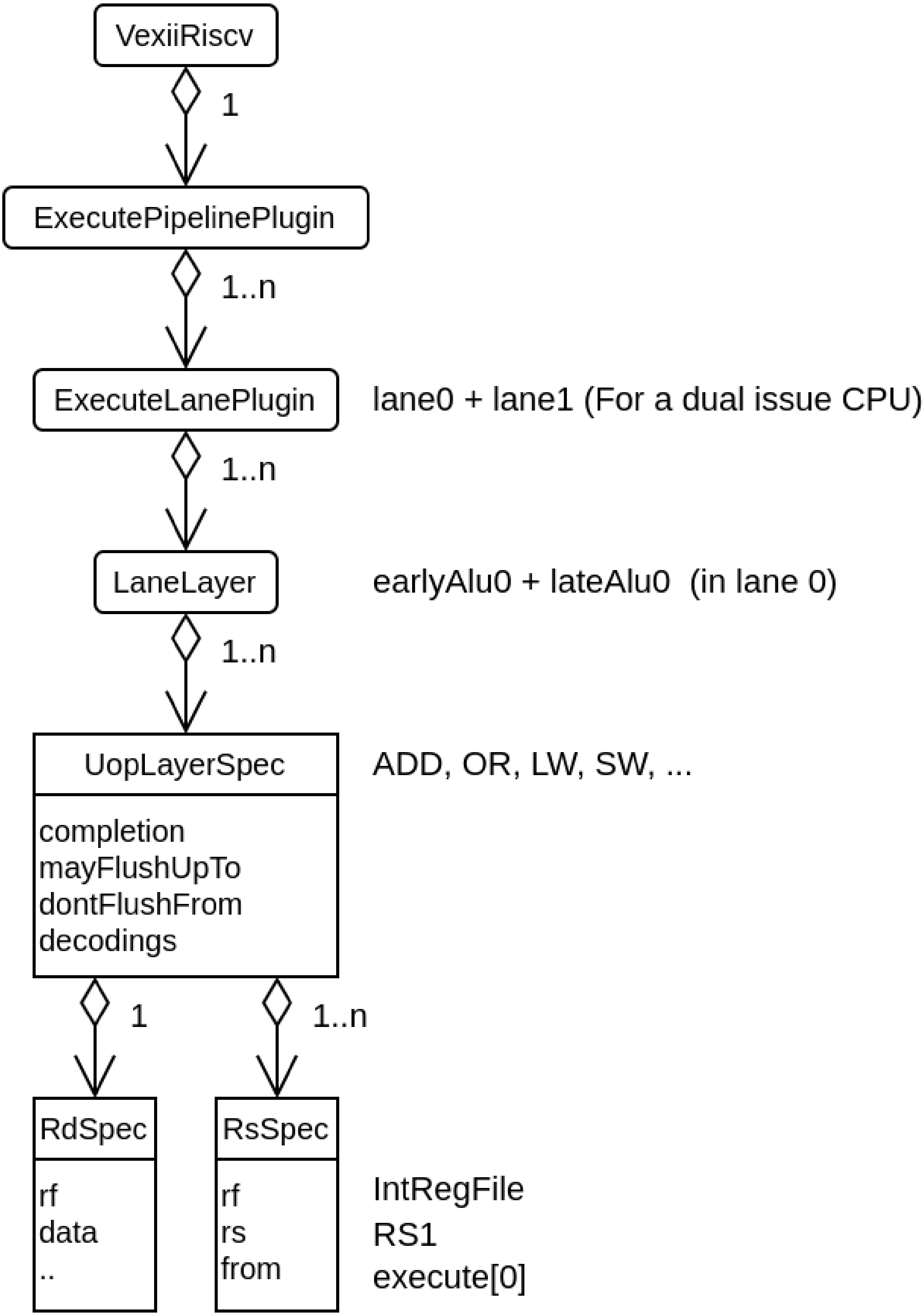
Here are a list of things that the schedulers need to take in account to know on which layer an instruction could be scheduled :

- 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.

4.4.2 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.


VexiiRiscv
1
ExecutePipelinePlugin
1..n
ExecuteLanePlugin
lane0 + lane1 (For a dual issue CPU)
1..n
LaneLayer
earlyAlu0 + lateAlu0 (in lane 0)
1..n
UopLayerSpec
ADD, OR, LW, SW, ...
completion
mayFlushUpTo
dontFlushFrom
decodings
1
1..n
RdSpec
rf
data
..
RsSpec
rf
rs
from
IntRegFile
RS1
execute[0]

CHAPTER

FIVE

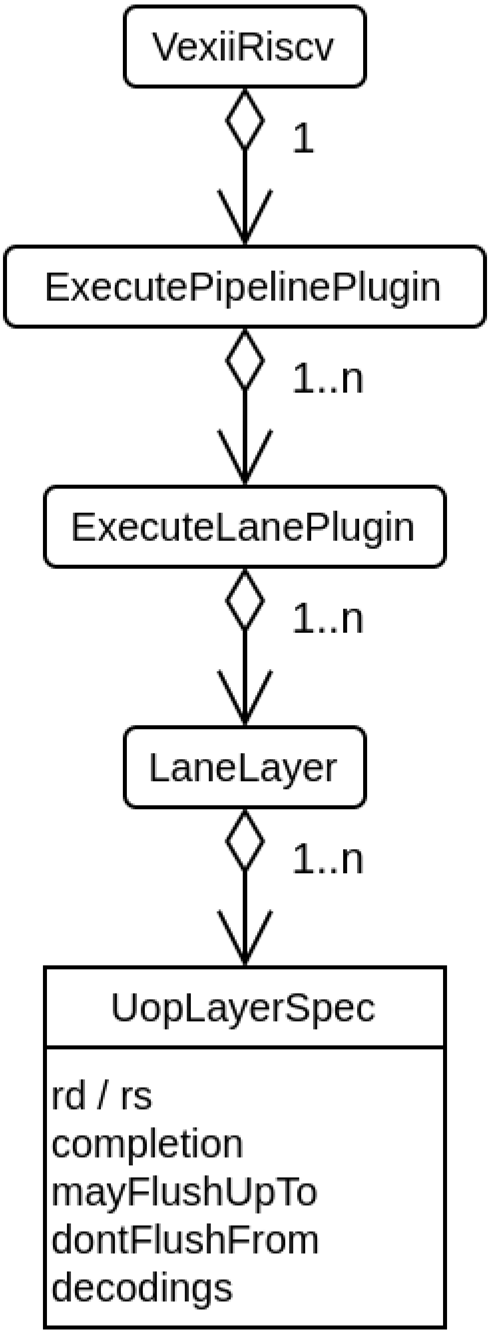
EXECUTE

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 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.

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 demand

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

Allows 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 optionally 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 backpressure 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 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

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)

    // Let's get the hardware interface that we will use to provide the result of our␣
```

(continues on next page)

(continued from previous page)

```
↪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()
    pa.plugins += new SimdAddPlugin(pa.early0)
```

(continues on next page)

(continued from previous page)

```
    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()

```

(continues on next page)

(continued from previous page)

```
  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.

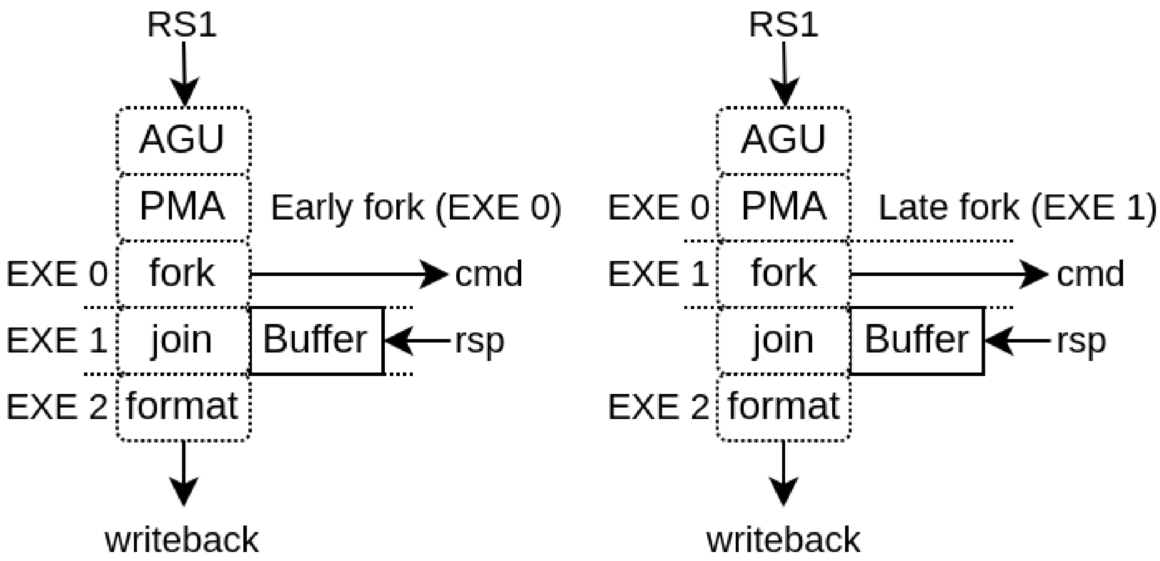
5.4 Load Store Unit (LSU)

VexiiRiscv has 2 implementations of LSU :

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

5.4.1 Without L1

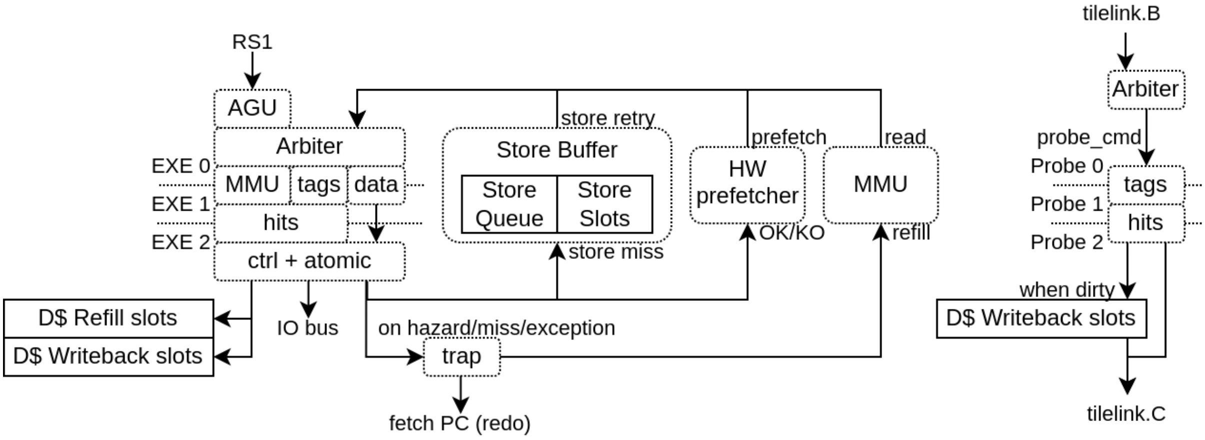
Implemented by the LsuCachelessPlugin, it should be noted that to reach good frequencies on FPGA SoC, forking the memory request at execute stage 1 seems to provide the best results (instead of execute stage 0), as it relax the AGU timings as well as the PMA (Physical Memory Attributes) checks.

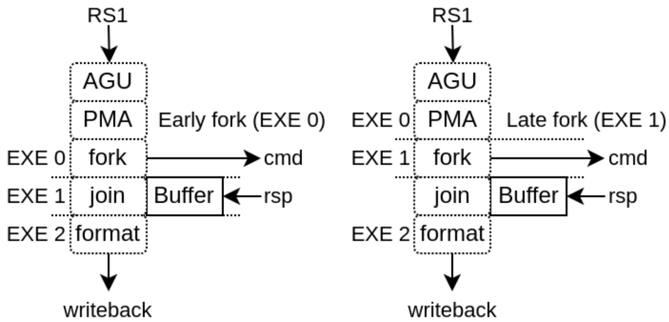


5.4.2 With L1

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.

5.4.3 Memory coherency

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

So, the L1 cache will have the following stream 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 (isInvalid/isShared/isUnique)

PICTURE

5.4.4 Prefetching

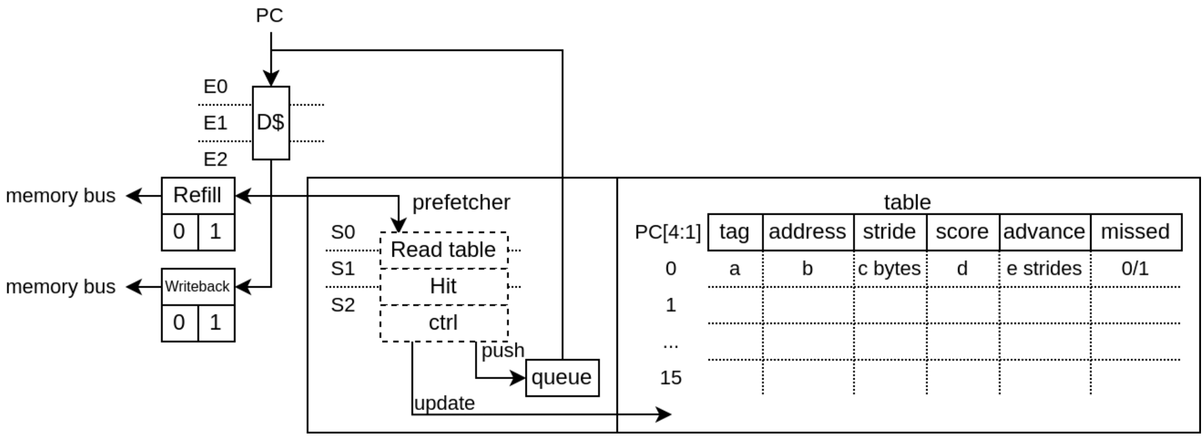
Currently there is two implementation of prefetching

- 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 memcopy 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 1: 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

5.5 FPU

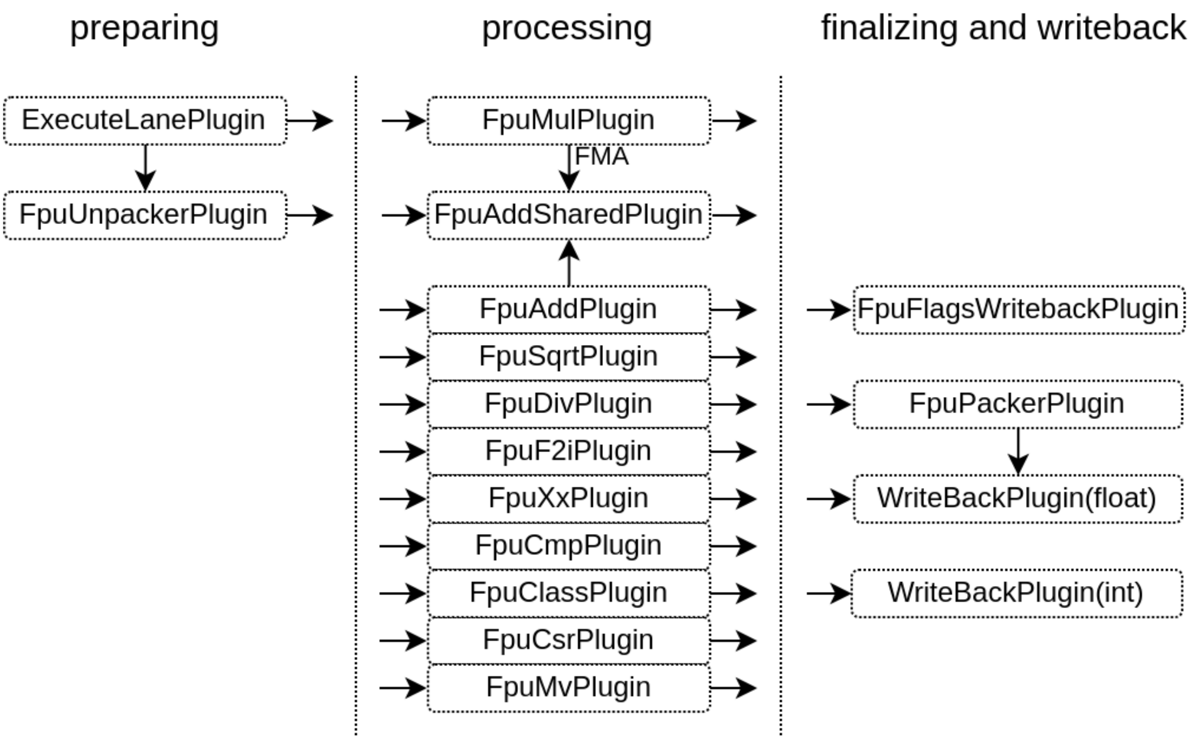
The VexiiRiscv FPU has the following characteristics :

- 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)

5.5.1 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 subnormal) 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



5.5.2 Area / Timings options

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

The first option is to reduce the FMA (Float Multiply Add instruction A*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 value as normal floating point numbers. This reduce the area by not having to handle subnormals (it removes big barrels shifters) , as well as improving timings. The down side is that the floating point value range is slightly reduced, and if the user provide floating point constants which are subnormals number, they will be considered as 2^exp_subnormal numbers.

In practice those two option do not seems to creates issues (for regular use cases), as it was tested by running debian with various software and graphical environnements.

5.5.3 Optimized software

If you used the default FPU configuration (deeply pipelined), and you want to achieve a high FPU bandwidth, your software need to be careful about dependencies between instruction. 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 necessary do a got job about it, as it may assume assume that the FPU has a much lower latency, or just optimize for code size.

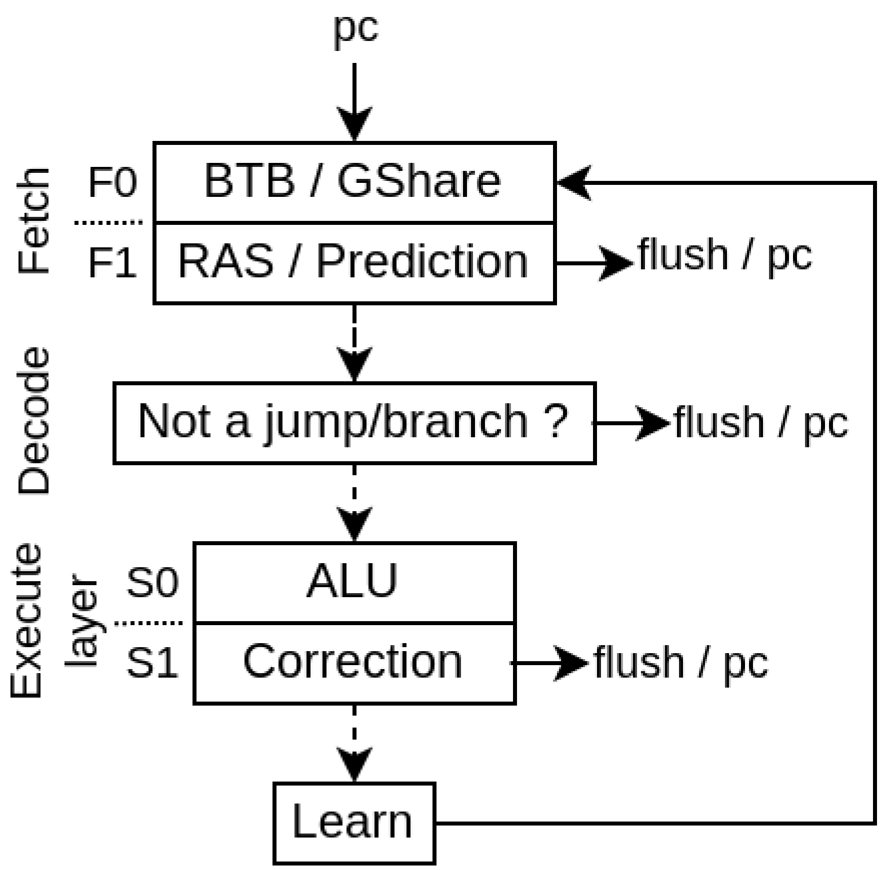
CHAPTER SIX

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 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 :

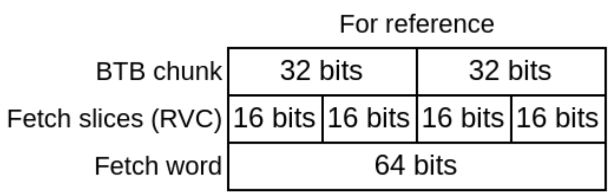


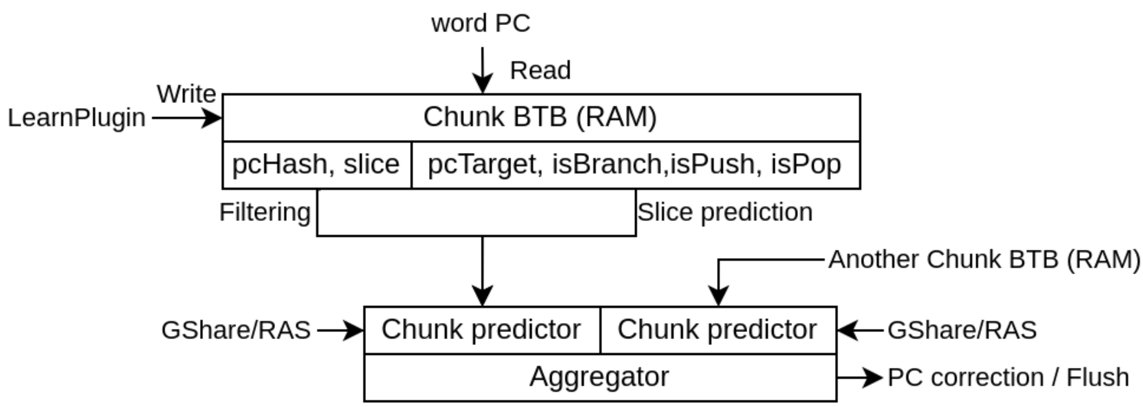
While it would have been possible in the decode stage to correct some miss prediction from the BTB / RAS, it isn’t done to improve timings and reduce Area.

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
- 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 learn.
- 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 usefull
- 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.

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 (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)

6.3 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

6.4 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.

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

DEBUG

7.1 JTAG

VexiiRiscv support 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)

CHAPTER

EIGHT

HOW TO USE

8.1 Dependencies

You will need :

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

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)
```

(continues on next page)

(continued from previous page)

```
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
# 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
```

8.2 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 ../..
```

8.3 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
```

(continues on next page)

(continued from previous page)

```
--with-mul
--with-div
--with-rva
--with-rvc
--with-supervisor
--with-user
--without-mul
--without-div
--with-mul
--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
...
```

8.4 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/dhrystone/build/rv32ima/dhrystone.elf --trace-all
```

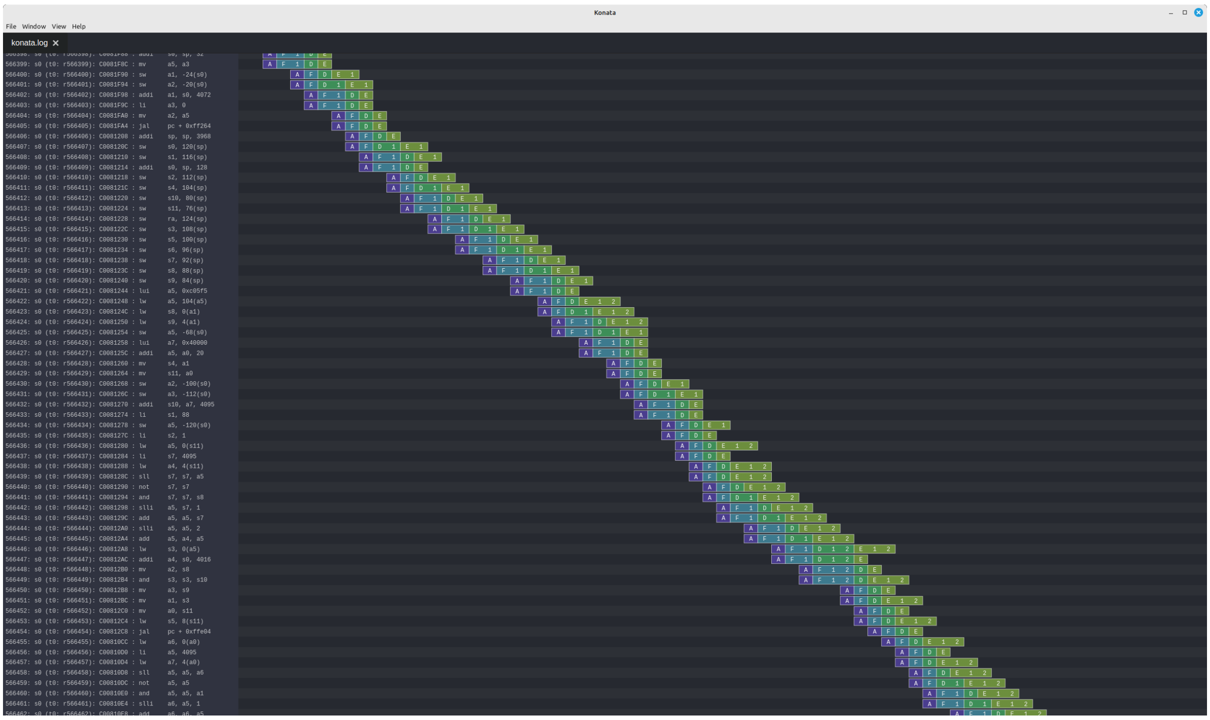
This will generate a simWorkspace/VexiiRiscv/test folder which contains :

- 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 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 :



8.5 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 synthetisable 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).

8.6 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 ((*Litex*))

CHAPTER

NINE

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
Drystone/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␣
↪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␣
↪timings
- The inputs/outputs of the design are serialized/deserialized (ff+logic cost) to␣
↪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 Dhrystone/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 Dhrystone/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 Dhrystone/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 Dhrystone/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 Dhrystone/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
```

9.1 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

CHAPTER

TEN

SOC

This is currently WIP.

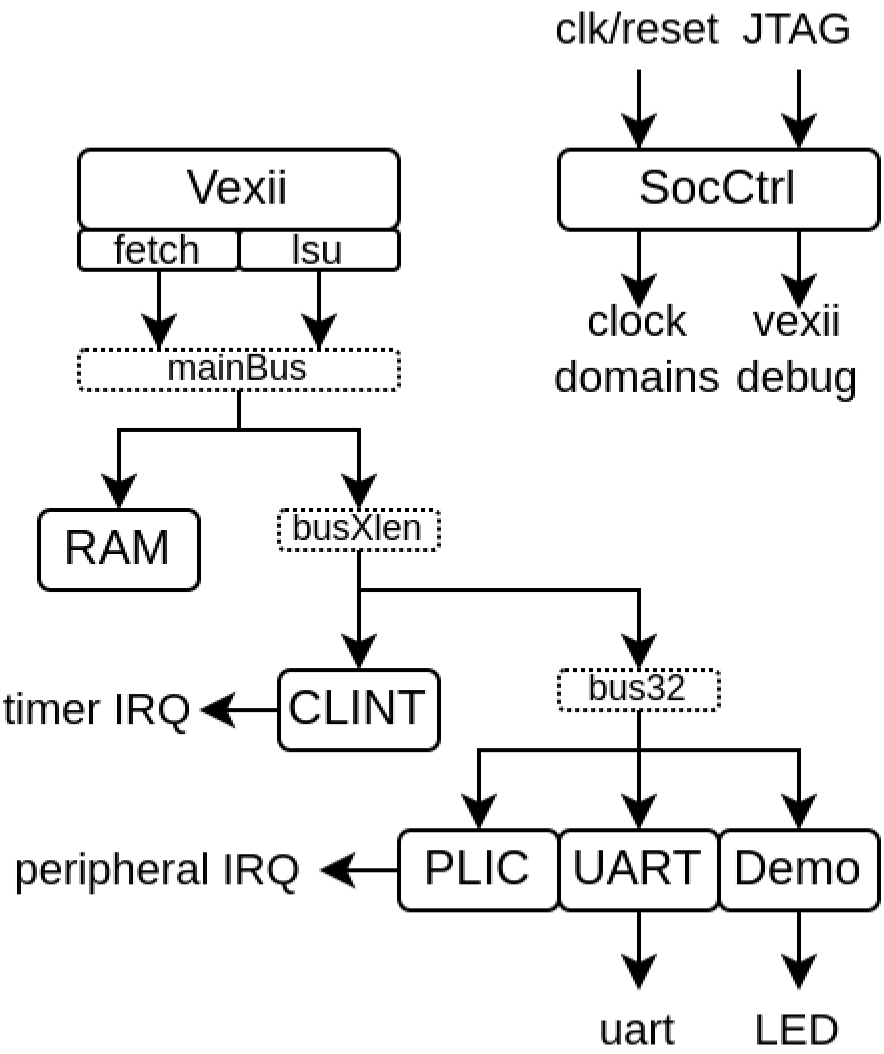
10.1 MicroSoc

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

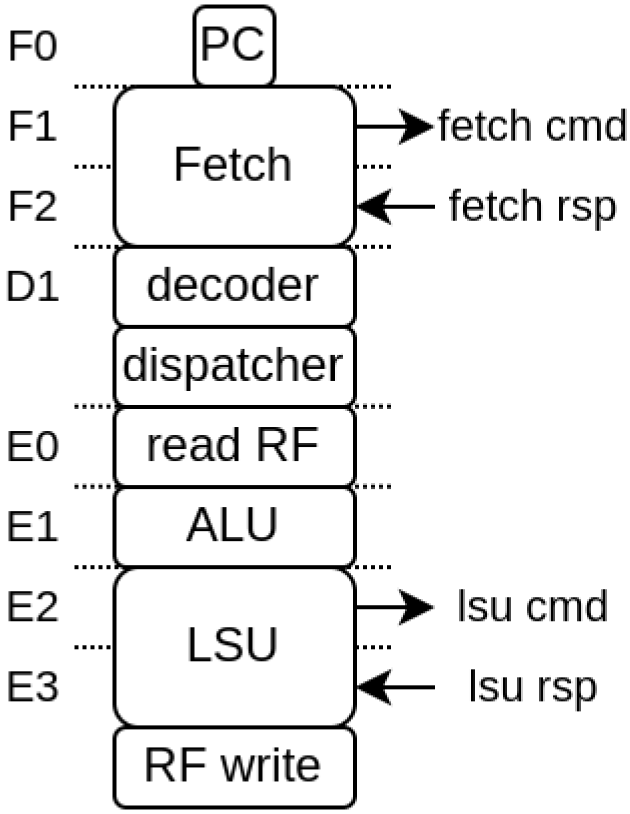
Its goals are :

- 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 ^^)

10.1.1 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"
```

10.1.2 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 simWorkspace/MicroSocSim/test (you can open it using GTKwave)
–trace-konata	A konata trace of all the executed instruction will be stored in simWorkspace/MicroSocSim/test (you can open it using https://github.com/shioyadan/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-btb –with-ras –with-gshare*

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

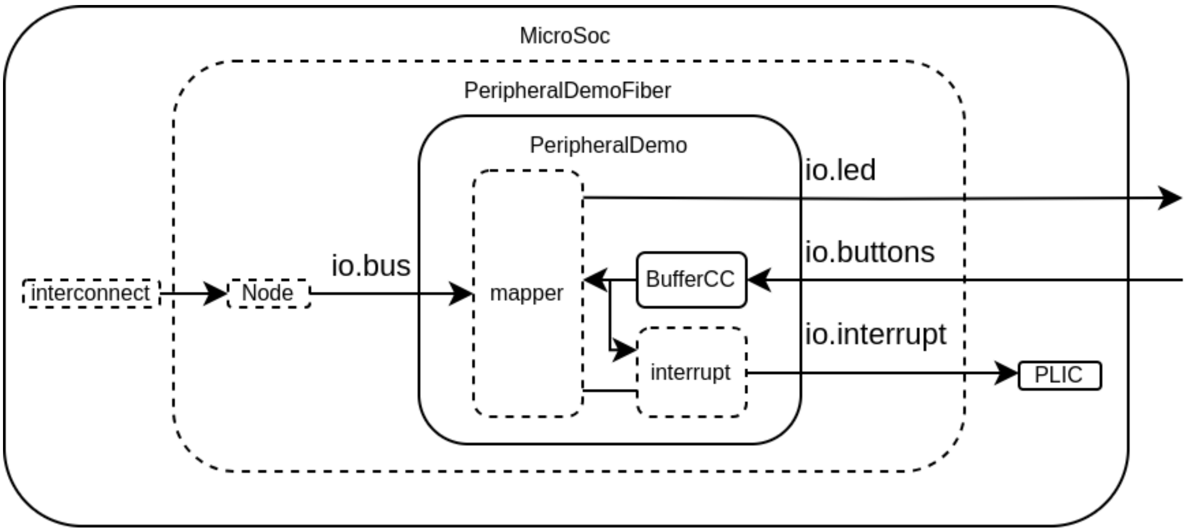
```
openocd -f src/main/tcl/openocd/vexiiriscv_sim.tcl
```

10.1.3 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 :

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, aswell 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"

10.2 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=vexiiriscv --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
```