VexiiRiscv Documentation

VexiiRiscv contributors

Dec 31, 2023

CONTENTS

Welcome to VexiiRiscv's documentation!

CHAPTER ONE

INTRODUCTION

Miaouuu

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 (29/12/2023) the status is :

- rv 32/64 im supported
- Can run baremetal benchmarks (2.24 dhrystone/mhz, 4.62 coremark/mhz)
- single/dual issue supported
- late-alu supported
- BTB/RAS/GShare branch prediction supported

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 advantages 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

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
- 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

3.5 GSharePlugin

Will :

- Implement a FetchConditionalPrediction (GShare flavor)
- Learn using the LearnPlugin interface
- Will not apply the prediction via flush / pc change, another plugin will do that

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

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"

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

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

- ExecutePipelinePlugin
- ExecuteLanePlugin
- RegFilePlugin
- SrcPlugin
- RsUnsignedPlugin
- IntFormatPlugin
- WriteBackPlugin
- LearnPlugin

Some implement regular instructions

- IntAluPlugin
- BarrelShifterPlugin
- BranchPlugin
- MulPlugin
- DivPlugin
- LsuCachelessPlugin

Some implement CSR, privileges and special instructions

- CsrAccessPlugin
- CsrRamPlugin
- PrivilegedPlugin
- PerformanceCounterPlugin
- EnvPlugin

5.1 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)

5.2 ExecuteLanePlugin

Implement an execution lane in the ExecutePipelinePlugin

5.3 RegFilePlugin

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

5.4 SrcPlugin

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

5.5 RsUnsignedPlugin

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

5.6 IntFormatPlugin

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

5.7 WriteBackPlugin

Used by plugins to provide the RD value to write back to the register file

5.8 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.

5.9 IntAluPlugin

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

5.10 BarrelShifterPlugin

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

5.11 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

5.12 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

5.13 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)

5.14 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

5.15 CsrAccessPlugin

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

5.16 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

5.17 PrivilegedPlugin

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

5.18 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

5.19 EnvPlugin

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