# CCOPI: Implementing a Custom Coprocessor Interface for VexRiscv

Jens Nazarenus, Dominik Swierzy

RheinMain University of Applied Sciences

`{jens.nazarenus, dominik.swierzy}@hs-rm.de`

March 27, 2018

**Abstract**

*This paper introduces CCOPI, a minimal-overhead, debug-friendly custom coprocessor interface for the RISC-V implementation VexRiscv. CCOPI, as well as VexRiscv, is written in the hardware description language SpinalHDL. The interface is responsible for the communication between the coprocessor and the core CPU pipeline of VexRiscv and thus helps hardware developers designing a coprocessor with a custom instruction-set extension.*

*CCOPI uses the flexibility of the RISC-V implementation VexRiscv to create the interface. This paper also shows how VexRiscv is designed particularly with regard to modifications and custom extensions.*

## 1 Introduction

Coprocessors in general are used to support the core CPU(s) with the calculation of specific mathematical operations. A coprocessor aims to accelerate this specific task over a software implementation of the same task.

Former coprocessors like the Intel 80387 or Motorolas 68881 were used to execute IEEE 754 compliant floating point operations and could be bought as independent components [1, 2]. To use the coprocessors new instructions were included in the instruction stream of the core CPU.

Nowadays the logic of coprocessors is often included inside the CPU itself as an instruction-set extension. Well-known examples are the Intel® x87 floating point unit (FPU) or the Intel® Advanced Encryption Standard New Instructions (AES-NI) [3, 4]. Both modules introduce new instructions, which can be used by developers. CCOPI is designed to produce coprocessors which resides inside the core RISC-V CPU, named VexRiscv, similar to x87 or AES-NI discussed above.

Before approaching the main topic CCOPI and its implementation, this paper gives an introductional overview of RISC-V and the used hardware description language SpinalHDL.

## 2 RISC-V

RISC-V is the name of an open source instruction-set architecture (ISA), which has been developed at the University of California, Berkeley [5]. The ISA describes the instructions any RISC-V implementation must implement, plus optional extensions. The required minimum is specified in the "Base Integer Instruction Set" of the User Level ISA.

A RISC-V instruction *instr* is always encoded in 32 bits.

| 31 25 | 24 19 | 20 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |

*Figure 1: R-Type instruction format*

The notation $instr[fromBit : toBit]$ is used to extract information from the instruction. Figure 1 $instr[6 : 0]$ represents the opcode, which is the identification code for a machine instruction. Depending on the opcode, $instr[31:7]$ is decoded in different ways. In Figure 1 the "R-Type" instruction format is shown, which is responsible for Integer-Register-Register operations, where the operands are located in the registers specified by $rs1$ and $rs2$. With $instr[11:7]$ the destination register $rd$ is specified. $funct7$ and $funct3$ define the exact sub operation of the instruction.

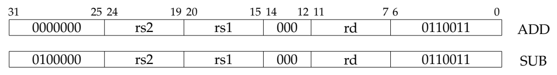


*Figure 2: ADD, SUB operation*

Figure 2 shows that both, the ADD and the SUB operation, share the opcode 0110011. The field $funct7$, which is decoded in $instr[31:25]$, exposes the exact type of operation:

- 0000000 : ADD operation
- 0100000 : SUB operation

$rs1$, $rs2$ and $rd$ are always encoded at the same positions in $instr$, to simplify the decoding process in an implementation of RISC-V [5, p. 11].

The ISA also defines opcodes which will not be used by future extensions and, as a result, available for custom instruction-set extensions [5, p. 103]:

- 0001011 : custom-0
- 0101011 : custom-1
- 1011011 : custom-2
- 1111011 : custom-3

The opcodes of custom-2/3 may not be used in RISC-V implementations where the integer registers are extended to 128 bits (RV128I) [5, p. 103]. With CCOPI it is possible to create instruction-set extensions with one or more of the described custom opcodes.

## 3 SpinalHDL

The CPU implementation VexRiscv and CCOPI are implemented in the language "SpinalHDL", which is a "high level hardware description language" to describe digital hardware as part of register-transfer level design [6]. Output of a compilation process is VHDL or Verilog code. For this reason SpinalHDL is a transcompiler which takes Scala code and transforms it into Verilog or VHDL.

In the following example a component named "Comp" is created. The design adds the two input signals "a" and "b" with an additional "x", which is specified as a Scala class constructor parameter:

```
class Comp(x: Int) extends Component {
    val io = new Bundle {
        val a, b = in UInt(8 bits)
        val result = out UInt(8 bits)
    }
    io.result := io.a + io.b + x
}
```

To generate the module, the compiler of SpinalHDL gets called with one of the functions `SpinalVerilog(component)` or `SpinalVhdl(component)`. The variable "x" is set to 15:

```
object MyComponentGen {
  def main(a: Array[String]): Unit = {
    SpinalVerilog(new Comp(15))
  }
}
```

Afterwards the following Verilog code gets emitted in the file "Comp.v", where 15 is resolved to `8'b00001111` by the SpinalHDL compiler:

```
module Comp (
      input  [7:0] io_a,
      input  [7:0] io_b,
      output [7:0] io_result);
  wire [7:0] zz_1;
  assign zz_1 = (io_a + io_b);
  assign io_result = (zz_1 +
                   (8'b00001111));
endmodule
```

With the combination of Scala and SpinalHDL, it is possible to create more abstract hardware designs. Object oriented software patterns like inheritance, polymorphism or type parametrization are built-in features of Scala and can be used. Even higher-order functions like Scalas "map" function will get evaluated during the compilation.

The following SpinalHDL code initializes a ROM with predefined values in a Scala list:

```
def sbox = List(0x63, 0x7C, ...)
val romsbox = Mem(Bits(8 bits),
                  sbox.map(B(_, 8 bits)))
```

The generated code looks as follows:

```
module Comp ();
  reg [7:0] romsbox [0:1];
  initial begin
    romsbox[0] = 'b01100011;
    romsbox[1] = 'b01111100;
    ...
  end
endmodule
```

SpinalHDL also introduces data types which are divided in composite types and base types. The available base types are [7]:

- Bool (True, False)
- Bits (A vector of Bits, like `B"0011"`)
- UInt/SInt (Unsigned/Signed Integer, used for integer arithmetic)

Composite types are supposed to group base types. A `Bundle` for example may be used to group the available input/output signals of a component under a single name:

```
val io = new Bundle {
    val a, b = in UInt(8 bits)
    val result = out UInt(8 bits)
}
```

Since SpinalHDL is based on Scala, build tools like SBT can be used to manage dependencies and compile a SpinalHDL based hardware project. The dependencies `spinalhdl-core` and `spinalhdl-lib` are required to use the language features discussed above:

```
libraryDependencies ++= Seq(
  "com.github.spinalhdl" % "spinalhdl-core_2.11"
    % "0.10.15",
  "com.github.spinalhdl" % "spinalhdl-lib_2.11"
    % "0.10.15",
)
```

Both, VexRiscv and CCOPI, use the object oriented design pattern paired with functional programming elements to describe the hardware in an abstract way. The next chapter gives a more detailed overview of the RISC-V implementation named VexRiscv.

# 4 VexRiscv

CCOPI uses VexRiscv as its underlying RISC-V implementation. VexRiscv implements the "RV32IM" instruction set, which means that it contains the following features [8]:

- 32 Bit registers
- Base Integer instruction set
- Standard Extension for Integer Multiplication and Division

The CPU is pipelined on a five-stage RISC pipeline like the MIPS architecture, whereby the **Instruction Fetch** stage copies the next 32-bit instruction in the register "pc" [9, C-34]. Once the instruction is present, the **Decode** stage analyzes the instruction. Based on the opcode, the Decode stage knows how to interpret $instr[31:7]$ of the instruction, which is stored in "pc". The **Execute** stage computes a result based on the operands which were determined in the Decode stage. One or more Arithmetic Logical Units (ALU) are used to perform arithmetic or logical operations in the **Execute Stage**. The **Memory** stage handles the Load/Store operations of RISC-V. The final pipeline stage, the **Write Back** stage, copies the result of the Execute stage in the destination register "rd", which is specified in $instr[11:7]$ of the instruction.

## 4.1 Plugin mechanism

A CPU in general consists of a large number of modules, for example:

- Arithmetic Logical Units
- Branch prediction module
- Pipeline-hazard management module

In VexRiscv each module is designed as a **plugin**, which describes how the pipeline should behave, when the plugin is active. For this purpose a module (or plugin) can modify every single pipeline stage. The compilation process of SpinalHDL resolves the plugins and creates a pipeline which contains the plugin-specific pipeline modifications.

A VexRiscv plugin must be programmed against the Trait `Plugin`:

```
class SomePlugin extends Plugin[VexRiscv] {

  /** Configure plugin */
  override def setup(p: VexRiscv): Unit = {

  }

  /** Change pipeline */
  override def build(p: VexRiscv): Unit = {

  }
}
```

Once "`extends Plugin[VexRiscv]`" is defined, the two functions `setup()` and `build()` must be implemented. The configuration of the plugin takes place in the `setup()` function. In the `build()` function pipeline modifications can be defined. For this purpose it is possible to plug into the different pipeline stanges:

```
override def build(p: VexRiscv): Unit = {
  import pipeline._
  import pipeline.config._

  execute plug new Area {
    // Execute Stage code
  }

  memory plug new Area {
    // Memory Stage code
  }
}
```

In the listing above, pipeline specific code can be defined in the corresponding stage area. With CCOPI, a new instruction-set extension will be defined in the `setup()` function. The communication between VexRiscv and the coprocessor takes place in the execute- and memory-stage specific pipeline areas.

# 5 CCOPI

## 5.1 Communication

CCOPI is the name of the custom coprocessor interface for the RISC-V implementation VexRiscv. It is designed to satisfy the following basic principles:

- A coprocessor is a VexRiscv plugin
- A designed coprocessor can also be used without VexRiscv

A CCOPI based coprocessor contains one or more "instruction functions", which are hardware operations bound to distinct instruction patterns. For example a coprocessor named "SymmetricCryptoCoProcessor" can be created with the following two instruction functions:

- AES, with the pattern:
  `"-----------------000-----0001011"`
- DES, with the pattern:
  `"-----------------001-----0001011"`

Both instruction functions use the custom-0 opcode, but the patterns differ at *instr*[14 : 12] (funct3). The "-" signs represent "Don't-care bits" which are not relevant for the decoding process.

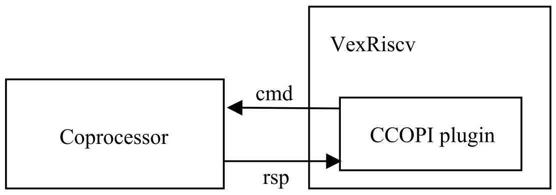


*Figure 3: Component structure*

**Decoder registration.** Since CCOPI is a VexRiscv plugin, it is possible to register the instruction patterns at the decoder service of the CPU implementation. This can be done in the `setup()` function of the CCOPI plugin:

```
val decoder = pipeline.service(
                      classOf[DecoderService])
decoder.add(...)
```

Once the patterns are registered, the decoder of VexRiscv is able to decode the instruction functions of CCOPI.

**Pipeline modifications.** Figure 3 shows that the coprocessor is connected with the CCOPI plugin. The communication between them is controlled by a handshake-based protocol named "Ready/Valid interface" [10]. The corresponding implementation in SpinalHDL is called "Stream bus" [11].

If one of the registered instruction functions gets decoded in the Decode stage, the CCOPI plugin sends a command (cmd) to the coprocessor and waits for its response (rsp). To be more precise, the pipeline gets stalled, until the response is calculated.

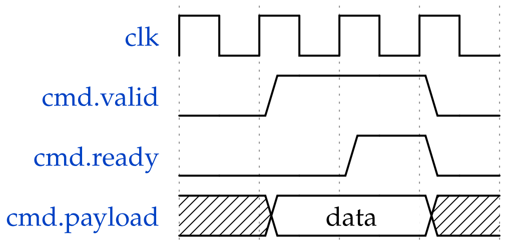


*Figure 4: Ready/Valid interface example communication for cmd.*

Figure 4 shows the communication of a command, which is specified in the Execute stage of the CCOPI plugin:

- cmd.valid – set by the CCOPI plugin to indicate that new data is available
- cmd.payload – holds the content of the command
- cmd.ready – set by the coprocessor to signal that the data transfer is complete

The payload of a command is 96 bits wide and contains the following three 32 bit values:

- The current instruction *instr*[31 : 0]
- Register content of rs1, which is specified in *instr*[20 : 15]
- Register content of rs2, which is specified in *instr*[24 : 19]

CCOPI uses inheritance to define the command as SpinalHDL Bundles:

```
class InputBundle extends Bundle {
  val cpuRS1 = Bits(32 bits)
  val cpuRS2 = Bits(32 bits)
}

case class Cmd() extends InputBundle {
  val opcode = Bits(7 bits)
  val rd = Bits(5 bits)
  val tags = Bits(3 bits)
  val rs1 = Bits(5 bits)
  val rs2 = Bits(5 bits)
  val funct = Bits(7 bits)
}
```

Note that the Bundle `Cmd` represents the introduced R-instruction type from Figure 1. The coprocessor, however, is not limited to the predefined types, which means that other interpretations of *instr*[31 : 0] are possible. The response payload is 32 bits wide and will be written to the register specified in "rd".

As an example the emitted Verilog code for a coprocessor with an instruction function "AES" looks as follows:

```
module Coprocessor (
  input aes_comm_cmd_valid,
  output reg aes_communication_cmd_ready,
  input [31:0] aes_comm_cmd_payload_cpuRS1,
  input [31:0] aes_comm_cmd_payload_cpuRS2,
  input [6:0] aes_comm_cmd_payload_opcode,
  input [4:0] aes_comm_cmd_payload_rd,
  input [2:0] aes_comm_cmd_payload_tags,
  input [4:0] aes_comm_cmd_payload_rs1,
  input [4:0] aes_comm_cmd_payload_rs2,
  input [6:0] aes_comm_cmd_payload_funct,
  output aes_comm_rsp_valid,
  input aes_comm_rsp_ready,
  output reg [31:0] aes_comm_rsp_payload_data,
  input clk,
  input reset);

  [...]
endmodule
```

The signals, which start with "`aes_communication_cmd`", in the Verilog code above shortened to "`aes_comm_cmd`", control the communication from the CCOPI plugin to the coprocessor (command). The response signals start with the pattern "`aes_communication_rsp`".

In addition, the CCOPI plugin adds the three signals "`{decode, execute,` memory}_cocpu_aes" to the VexRiscv component. These signals are set to "high" when the pipeline executes an instruction for the coprocessor.

**CPU generation.** In SpinalHDL a class, which inherits `Component`, can be used to generate Verilog or VHDL code. In the CPU implementation the class `VexRiscv` must be used for this purpose. The list of plugins must be passed as a parameter:

```
def cpu() = new VexRiscv(
  config = VexRiscvConfig(plugins = List(
      [...]
      new CoProcessorPlugin(new Cocpu1())
      new CoProcessorPlugin(new Cocpu2())
  ))
)
```

The CPU can be generated with the following command and will generate the file `VexRiscv.v`:

```
SpinalVerilog(cpu()) // VexRiscv.v
```

## 5.2 Coprocessors

In the previous chapter the plugin mechanism of VexRiscv and the communication of the CCOPI plugin is described. With CCOPI it is possible to create coprocessors and instruction functions in an object oriented way.

A coprocessor is a class, which inherits the class `CoProcessor`. Inside this class it is possible to create instruction functions which are bound to a specific instruction pattern:

```
def aes = new InstructionFunction[Cmd, Rsp]
          (new Cmd(), new Rsp()) {
  val pattern: String = s"-----------------" +
                         "000-----${custom0}"
  val name: String = "aes"
  val description: String = "AES␣impl."

  def build(c: EventController): Unit = {
    /** AES implementation */
  }
}

def setup(): Unit = {
  activate(aes)
}
```

This code example demonstrates an instruction function named "aes", which is bound to the custom-0 opcode. The function `build()` must be implemented and contains the hardware specific code of the instruction function, for example the implementation of the symmetric cryptosystem AES. Each of them must be activated in the `setup()` method of the coprocessor to indicate SpinalHDL to generate the corresponding Verilog code.

| Signal name | Type | Mode |
|---|---|---|
| command | `Cmd` | Read-Only |
| response | `Rsp` | Read/Write |
| done | `Bool` | Read-Only |
| flush | `Bool` | Read/Write |

*Table 1: Automatically included signals in `build()`*

Table 1 shows four signals which get automatically included in every `build()` function. The signal type of "command" and "response" depends on the definition of `InstructionFunction`. To control the state of the coprocessor, "done" and "flush" can be used:

- done – false, if the instruction function is busy, true otherwise
- flush – User-settable boolean value to signal that the response can be transferred to the CPU

## 5.3 Detailed example

In the following detailed example a coprocessor named "ex1" is defined. It adds the register content of rs1 and rs2, plus an additional value of the internal coprocessor ROM. For this purpose five of the seven most significant bits of the 32 bit instruction can be used to define the ROM address:

```
class CoProcessorEx1 extends CoProcessor {

  case class Cmd() extends InputBundle {
    val opcode = Bits(7 bits)
    val rd = Bits(5 bits)
    val funct3 = Bits(3 bits)
    val rs1 = Bits(5 bits)
    val rs2 = Bits(5 bits)
    val intRomAddr = Bits(5 bits) // int. RAM
    val funct2 = Bits(2 bits)
  }
  case class Rsp() extends OutputBundle {
    val data = Bits(32 bits)
  }

  def ex1 = new InstructionFunction[Cmd, Rsp]
              (new Cmd(), new Rsp()) {
    val pattern: String = s"00---------------" +
                           "001-----${custom0}"
    val name: String = "ex1"
    val description: String = "Example␣1"

    def build(c: EventController): Unit = {

      val regs = c.prepare event new Area {
        def init = List (0x20, 0x40, 0x60)
        val internalRom = Mem(Bits(8 bits),
                init.map(B(_ , 8 bits)))
      }

      val exec = c.exec event new Area {
        val counter = Counter(50)

        val romReadAddr = command.intRomAddr
            .asUInt.resize(2)
        val romRead = regs.internalRom
            .readAsync(romReadAddr)
        response.data := (command.cpuRS1.asUInt
            + command.cpuRS2.asUInt
            + romRead.asUInt).asBits

        when(counter.willOverflowIfInc) {
          flush := True
        }

        when(!done) {
          counter.increment()
        }.otherwise {
          counter.clear()
        }
      }
    }
  }

  def setup(): Unit = {
    activate(aes)
  }
}
```

It is recommended to split the hardware code in two areas:

- `val x = c.prepare event new Area`
- `val y = c.exec event new Area`

In the "prepare area" it is possible to initialize ROMs or define additional functions. The actual computation should be executed in the "exec area".

In the example above a ROM with three values is initialized in the "prepare area". A counter is used in the "exec area" to demonstrate the stalling process of the CPU pipeline. The counter is used to delay the response flush for 50 cycles.

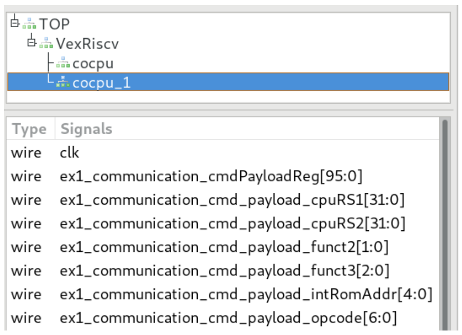
**Usage.** To use CCOPI in an Assembler or C application it is necessary to introduce a new instruction with `#define` statements:

```
#define ccopi_type_insn(_f2, _f5, _rs2, _rs1, \
                        _f3, _rd, _opc) \
.word ( ((_f2)  << 30) | ((_f5)  << 25) | \
        ((_rs2) << 20) | ((_rs1) << 15) | \
        ((_f3)  << 12) | ((_rd)  <<  7) | \
        ((_opc) << 0))

#define ccopicmd1(_rd, _rs1, _rs2, _iaddr) \
ccopi_type_insn(0b00, _iaddr, _rs2, _rs1, \
                0b001, _rd, 0b0001011)
```

The code snippet above defines a new instruction type, where the most significant seven bits (`_f7`) are split into two (`_f2`) and five (`_f5`) bits. In the second `#define` statement the command `ccopicmd1` with the parameters `_rd`, `_rs1`, `_rs2`, and `_iaddr` is introduced. The internal ROM address of the detailed example can be controlled with the `_iaddr` parameter.

**Signal naming.** A lot of effort has been put into giving the signals reasonable names.



| Type | Signals |
|---|---|
| wire | clk |
| wire | ex1_communication_cmdPayloadReg[95:0] |
| wire | ex1_communication_cmd_payload_cpuRS1[31:0] |
| wire | ex1_communication_cmd_payload_cpuRS2[31:0] |
| wire | ex1_communication_cmd_payload_funct2[1:0] |
| wire | ex1_communication_cmd_payload_funct3[2:0] |
| wire | ex1_communication_cmd_payload_intRomAddr[4:0] |
| wire | ex1_communication_cmd_payload_opcode[6:0] |

*Figure 5: GTKWave screenshot*

"ex1" is the name of the instruction function in the detailed example above. For this reason, each signal, which belongs to this specific instruction function, starts with "`ex1_`", followed by the area where the signal is defined:

- `ex1_prepare_<signal-name>`
- `ex1_exec_<signal-name>`

- `ex1_communication_<signal-name>`

The last point in the list is the signal pattern, which controls the internal communication of CCOPI and has already been discussed in 5.1.

Consistent signal names are very helpful for hardware simulation and debug purposes. Tools like "Verilator" use the signal names to produce a VCD-Trace, which afterwards can be analyzed with a Waveform viewer like "GTK-Wave". Figure 5 shows a GTKWave screenshot with CCOPI signals.

## 6 Parameters

Another advantage of CCOPI is the small overhead in terms of hardware size and communication latency. Since the size of a result computed by a coprocessor is known before the actual hardware is generated, the result can be transferred within one clock cycle via the Ready/Valid Interface.

The required number of Lookup tables (LUT) and Flip-Flops for VexRiscv depends on the active plugins. For test purposes a minimal configuration was chosen, which needs a total of 500 LUTs and 592 Flip-Flops on an Artix-7 FPGA. After adding a trivial coprocessor with CCOPI, which fills the destination register with zeros, only 34 additional LUTs and two more Flip-Flops are needed. This is a rather low price for the huge amount of flexibility CCOPI offers.

## 7 Future Work

At the moment CCOPI handles the communication for a synchronous execution, which means that the CPU pipeline is stalled while the coprocessor is busy. In the future it should be possible to use interrupts with the Platform Level Interrupt Controller (PLIC) to signal that the result of the coprocessor has been determined [12]. With this feature an asynchronous execution of one or more coprocessors would be possible.

## 8 Conclusion

This paper described the implementation of the custom coprocessor interface CCOPI and its integration in the RISC-V implementation VexRiscv. SpinalHDL was used as the hardware description language, which offers the possibility to define high level hardware designs.

CCOPI manages the communication between VexRiscv and a specified coprocessor automatically. A developer of a coprocessor no longer needs to worry about the Ready/Valid interface or pipeline stalls. Even in the leading RISC-V implementation "Rocket chip" the communication must be controlled by the developer, which makes CCOPI a unique coprocessor interface with a built-in communication abstraction layer [13].

With this in mind it is possible to create a unique RISC-V CPU with a set of additional functions. As a result, the integration of existing hardware implementations, like AES, can be achieved in a time-efficient way.

## References

[1] Motorola Inc. Motorola m68000 family programmer's reference manual. `https://www.nxp.com/docs/en/reference-manual/M68000PRM.pdf`, 1992. [Last accessed on: 2018/03/19].

[2] Intel Corporation. *80387 Programmer's Reference Manual*. Intel Corporation, 1987.

[3] Intel Corporation. Intel® 64 and ia-32 architectures developer's manual: Vol. 1, order no 253667. `https://www.intel.de/content/www/de/de/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html`. [Last accessed on: 2018/03/20].

[4] Intel Architecture Group Shay Gueron. Intel® advanced encryption stan-

dard (aes) new instructions set. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf. [Last accessed on: 2018/03/19].

[5] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. *The RISC-V Instruction Set Manual Volume I*. CS Division, EECS Department, University of Berkeley, California, v. 2.2 edition, 2017.

[6] SpinalHDL. Spinalhdl user guide. https://spinalhdl.github.io/SpinalDoc/. [Last accessed on: 2018/03/19].

[7] SpinalHDL. Spinalhdl user guide - types introduction. https://spinalhdl.github.io/SpinalDoc/spinal/core/types/TypeIntroduction. [Last accessed on: 2018/03/20].

[8] SpinalHDL/VexRiscv. Vexriscv - readme.md. https://github.com/SpinalHDL/VexRiscv/blob/master/README.md. [Last accessed on: 2018/03/20].

[9] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[10] Department of Electrical Engineering UC Berkeley College of Engineering and Computer Science. Interfaces: "fifo" (a.k.a. ready/valid). https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf. [Last accessed on: 2018/03/20].

[11] SpinalHDL. Spinalhdl user guide - stream bus. https://spinalhdl.github.io/SpinalDoc/spinal/lib/stream/. [Last accessed on: 2018/03/20].

[12] SiFive Inc. Andrew Waterman, Krste Asanovic. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. CS Division, EECS Department, University of Berkeley, California, v 1.10 edition, 2017.

[13] Github: freechipsproject/rocket chip. rocket-chip/src/main/scala/tile/lazyrocc.scala. https://github.com/freechipsproject/rocket-chip/blob/98b4e6223ed28b689646339 5ec56ca6e19b0d6e8/src/main/scala/tile/LazyRoCC.scala#L185-L196.