HeRVé: A Pipelined RISC-V CPU with Privileges Extensions in a Formal HDL

Gabriel Desfrene

August 2024

Abstract

Ensuring the correctness of hardware components is vital, as highlighted by historical issues like the *Pentium* FDIV bug. This incident spurred the adoption of formal methods for hardware verification, a practice that remains crucial today. The development of proof assistants has made it possible to reason rigorously about hardware components. Building on this, Thomas Bourgeat and Clément Pit-Claudel formalized the core concepts of the *Bluespec* language in the *Coq* proof assistant, leading to the creation of *Kôika*. This report details the development of a pipelined RISC-V CPU with interrupt management using the *Kôika* framework. The work aims to provide a more complex model for formal verification methods of hardware designs, especially for the framework developed by Pierre Wilke and Matthieu Baty.

1 Introduction

Ensuring the accuracy of hardware components is of paramount importance. The discovery of the *Pentium* FDIV bug, which caused significant errors in floating-point division calculations, prompted chip manufacturers to increasingly adopt formal methods to enhance the reliability of specific aspects of their design. While these techniques are still used today, as evidenced by the formal verification of security mechanisms in *Arm*'s *Morello* architecture at the ISA level, it is important to note that only a fraction of the behaviors in modern designs are subject to formal verification.

The development of proof assistants has significantly facilitated reasoning about complex objects, such as hardware designs. These tools enable the formal verification of complex behaviors, reinforcing confidence in the reliability and correctness of hardware components, which was previously unattainable. With this aim in mind, Thomas Bourgeat and Clément Pit-Claudel decided to formalize the core concepts of the *Bluespec* language [1], a hardware description language (HDL). This formalization, conducted in the *Coq* proof assistant [2], led to the development of a verified compiler for this language subset, called *Kôika*. This compiler targets *Verilog*, thereby enabling the synthesis of *Kôika* hardware designs. A pipelined RISC-V mini-processor serves as an example of *Kôika*'s application.

Using these tools, Pierre Wilke and Matthieu Baty developed a framework for conducting formal proofs on *Kôika* designs [3]. They successfully demonstrated the correctness of a security mechanism on the RISC-V mini-processor.

Building on the mini-processor example, I developed a pipelined RISC-V CPU with interrupt management in *Kôika*, following the RISC-V specification [4]. This development now enables formal proofs of CPU correctness and the verification of security mechanisms over a more sophisticated model. Although this CPU currently lacks additional privilege mechanisms such as execution levels and a memory mapping controller, which are necessary for running a fully-fledged operating system, it is designed with these features in mind for easy future integration.

2 The *Kôika* HDL

2.1 A Concurrent HDL

A pipelined CPU is inherently a concurrent system. For instance, in a classic fetch-execute pipeline, each of the stages will perform computation in the same cycle. As the fetch stage is looking for the instruction at `pc`, the exec stage will actually perform the computation of the previously fetched instruction. Because the different stages are handled simultaneously, this design allows for quicker execution of a set of instructions.

Kôika is a rule-based HDL, it describes the operation performed by hardware components at a high level of abstraction. Rules are atomic operations that will execute concurrently during a cycle. Rule-based HDL provide a simple concurrency model which can be leveraged for describing pipelined systems. *Kôika* programs are written using an embedded language inside the *Coq* proof assistant. This provides a way to manipulate such programs formally. The *Kôika* framework is available on GitHub [5].

2.2 Rule-based HDL

The semantics of *Kôika* is thoroughly exposed in [1]. Only a part of these semantics is required to understand the designs choices that I have made developing the pipelined RISC-V CPU. This presentation is inspired from the previous paper, but it will gloss over some details. Please refer to the article of Thomas Bourgeat et al. for these details.

A *Kôika* program is composed of *rules*, witch represent an atomic unit of computation. At each clock cycle, all rules defined in the *schedule* are executed concurrently. To perform computation, the *rules* manipulate values stored in *registers*. Typically, each *rule* reads some values from a set of input *registers*, performs some combinational (i.e. pure) computations, then writes results to a set of output *registers*. *Kôika* contains standard combinational operations such as:

- Binary operations over buses: addition, equality, signed and unsigned comparisons, etc.
- Multiplexing: conditional or C++ “switch”-like statement.
- Bus manipulation: bits extraction, concatenations, etc.

```
1 rule tick =
2   let count = read(clock) in
3   write(clock, count + 1)
```

Figure 1: A very simple *Kôika* rule, counting the number of cycles in `clock`.

Figure 1 exposes a very simple *Kôika* rule, which counts the number of cycles in a register named `clock`. This example reads the register `clock` and writes the incremented value to the same register. The semantics of reads and writes are not as straightforward as it may seem. Due to the decomposition of *Kôika* programs as a list of concurrent rules, reads return the value of the register at the beginning of the cycle, and all writes performed during a rule are committed at the end of the cycle. So, in out little example in figure 1, the incremented value of register `clock` is only visible at the beginning of the next cycle.

2.3 Resolving rule conflicts

When a rule tries to write two times on the same register, it *fails*. The code in figure 2 illustrates this possibility when registers `x` and `y` are both 0. When a rule *fails*, no changes are committed to the

```
rule may_fail =
  if read(x) == 0 then write(z, 1);
  if read(y) == 0 then write(z, 2)
```

Figure 2: A simple *Kôika* rule that can write multiples times on the same register.

written registers in this rule at the end of the cycle. The same issue can occur when two different rules try to write to the same register. Figure 3 demonstrates this issue. At best, the result of only one rule can be committed to the registers. We use the order given by the *schedule* to determine which one should have its results taken into account.

```
rule rule1 =
  if read(x) == 0 then write(z, 1)

rule rule2 =
  if read(y) == 0 then write(z, 2)

schedule prog = [rule1; rule2]
```

Figure 3: A *Kôika* program with two rules that can write to the same register.

The *Kôika* compiler ensures that all the rules are executed in parallel. It generates circuits to resolve conflicts when necessary. This circuitry ensures that if a conflict has occurred, the resulting state will be the same as if the rules were executed one after the other, according to the *schedule* list. In the program of figure 3, when registers `x` and `y` are 0 at the beginning of a cycle, only `rule1` is committed, and the value 1 is written to the register `z` at the cycle's end. The keyword `fail` is present in the *Kôika* language to force a rule to fail.

2.4 Refining `write` and `read`

In the previous description of the *Kôika* language, results of the computation done by a rule, can only be used by another rule at the next cycle. This is due to the semantics of `write` operations, which are only committed at the end of a cycle. This can hurt performance of *Kôika* hardware designs, when the results of a rule could be used immediately by another one.

To address this issue, the *Kôika* language, introduces *ports* for each `read` or `write` operation. Only two *ports* are possible: `0` and `1`. Previous descriptions of `read` and `write` operations are those on *port* `0`. Operations on *port* `1` have the following effects:

- `read1` can read the value written by a `write0`. If no `write0` has occurred, it is equivalent to a `read0`.
- `write1` will overwrite any value previously written by a `write0`. A `write0` occurring after a `write1` on the same register will fail.

The *Kôika* program presented in figure 4, illustrates the use of ports. Indeed, in some cases, two cycles of computation can be proceeded in one:

- If the value in `r` is even but not a multiple of 4, both rules are executed. The circuit will write $3 \times (\mathtt{r}/2) + 1$ in `r`.
- If the value in `r` is a multiple of 4, only the `divide` rule writes data to `r`. The circuit writes $\mathtt{r}/2$ in `r`.

- If the value in `r` is odd, only the `multiply` rule writes data to `r`. The circuit writes $3 \times \texttt{r} + 1$ in `r`.

```
1  rule divide =
2    let v = read0(r) in
3    if v[0] == 0 then (* v is even *)
4      write0(r, v >> 1) (* r := v / 2 *)
5
6  rule multiply =
7    let v = read1(r) in
8    if v[0] == 1 then (* v is odd *)
9      write1(r, (v << 1) + v + 1) (* r := 3 * v + 1 *)
10
11 schedule collatz = [divide; multiply]
```

Figure 4: This simple *Kôika* program computes the terms of the Collatz sequence in the register `r`.

In the next section, I will discuss which privileges extensions have been developed in my pipelined RISC-V CPU, and what designs choices I have made to implement this in *Kôika*.

3 Rewriting the CPU

3.1 *Kôika*'s CPU example

The CPU provided as an example in *Kôika*'s sources, is a pipelined implementation of the RV32I ISA, as described in the RISC-V Unprivileged Specification [6]. It operates through four primary stages: *Fetch*, *Decode*, *Execute*, and *WriteBack*. The last one is responsible for writing the results of the *Execute* stage to the corresponding destination register.

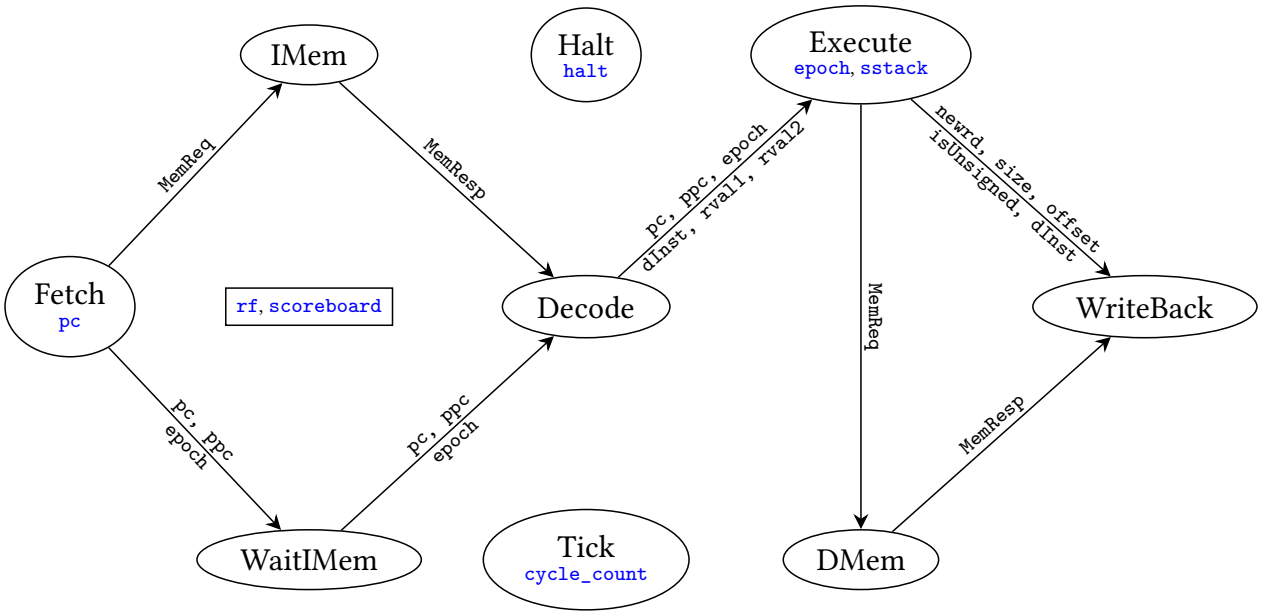

Figure 5: Pipeline of the *Kôika*'s example RISC-V processor.

Figure 5 depicts the detailed pipeline of this processor. Each stage, represented as a node in the diagram, is described as a *Kôika* rule. This ensures that all the stages of the pipeline will execute

in parallel if possible. Edges are FIFO structures that link rules. Data entering a FIFO at one cycle, is available at the FIFO's end at the beginning of the next cycle. Blue variables such as `epoch` or `scoreboard` are internal or global registers.

This CPU does not have any error checking (unaligned *pc*, unaligned memory access, etc.), and because no privilege extensions are implemented, neither does it implement error handling. With the work of Matthieu Baty et al., a shadow stack checks for buffer-overflow attacks. When triggered, this shadow stack halts the CPU, and no other instruction is processed.

Although it can perform calculations, this CPU cannot be used by an operating system. From this example, my work aims to implement a real pipelined processor with error management and signaling. To do so, I followed the RISC-V Privileged Specification [4].

3.2 Control and Status Registers (CSR)

Control and Status Registers (CSRs) are registers introduced in the RISC-V specification that allow the processor flow to be controlled and modified. I decided to implement all the general CSRs that describe the processor capabilities and state, and all the CSRs used for the management of interrupts and synchronous exceptions (errors). Table 1 lists all implemented registers.

Around 450 lines of specification where necessary to describe the circuits needed to read and update these registers. This specification is flexible, and is made for easy addition of new CSRs. The two extensions *Zicsr* and *Zicntr* of the RV32I Unprivileged ISA [6] have also been implemented in the pipelined CPU to perform changes on the CSRs. These extensions introduce six instructions to modify all the implemented CSRs:

- `csrrw` reads the original value of a CSR and writes the value of a register to it.
- `csrrs` reads the original value of a CSR and sets bits to it based on a mask in a register.
- `csrrc` reads the original value of a CSR and clears bits to it based on a mask and set bits to it based on a mask.
- `csrrwi`, `csrrsi`, `csrrci`: They are equivalent to the preceding instructions, but they use immediate values instead of a register value.

The code needed to add a non-privileged mode, called *User* in the RISC-V Specification, is nearly finished in the *HeRVé* processor.

3.3 Pipeline modifications

The majority of the development of the *HeRVé* processor was the conception of its pipeline. With the same convention as the previous figure, the *HeRVé* pipeline is shown in figure 6. Multiple changes were necessary to allow the management of interrupts.

First, checks have been added in the upper stages of the pipeline to detect illegal states, such as misaligned `pc` (Fetch stage), or illegal instructions (Decode stage). These checks have leads to the addition of the `failcode` field in the links between stages. This field is responsible for transmitting the errors that have occurred in processing of the instruction it comes with.

Next, a complete rework of the execution stage was mandatory. This rework has been dictated by a few concepts. First, I wanted the execution of instructions to be modular within the *HeRVé* CPU. It should be possible to add complex components operating on multiple stages, such as an FPU[1] or a shift-and-add multiplier in the ALU. Moreover, adding the interrupt management to this processor should not significantly impact its performance. To ensure maximal clock speeds, I needed to limit the growth of the critical path of the *HeRVé* design. This has led to the separation of the execution

[1] Floating Point Unit.

Category	Name	Address	Role
Unprivileged Counter/-Timers	`cycle`	`0xC00`	Cycle counter for `rdcycle` instruction.
	`time`	`0xC01`	Timer for `rdtime` instruction.
	`instret`	`0xC02`	Instructions-retired counter for `rdinstret` instruction.
	`cycleh`	`0xC80`	Upper 32 bits of `cycle`.
	`timeh`	`0xC81`	Upper 32 bits of `time`.
	`instreth`	`0xC82`	Upper 32 bits of `instret`.
Machine Information Registers	`mvendorid`	`0xF11`	Vendor ID.
	`marchid`	`0xF12`	Architecture ID.
	`mimpid`	`0xF13`	Implementation ID.
	`mhartid`	`0xF14`	Hardware thread ID.
	`mconfigptr`	`0xF15`	Pointer to a configuration data structure.
Machine Trap Setup	`mstatus`	`0x300`	Machine status register.
	`misa`	`0x301`	ISA and extensions.
	`mie`	`0x304`	Machine interrupt-enable register.
	`mtvec`	`0x305`	Machine trap-handler base address.
	`mstatush`	`0x310`	Upper 32 bits of `mstatus`.
Machine Trap Handling	`mscratch`	`0x340`	Scratch register for machine trap handlers.
	`mepc`	`0x341`	Machine exception program counter.
	`mcause`	`0x342`	Machine trap cause.
	`mtval`	`0x343`	Machine bad address or instruction.
	`mip`	`0x344`	Machine interrupt pending.
Machine Counter/-Timers	`mcycle`	`0xB00`	Machine cycle counter.
	`mtime`	`0xFC0`	Machine timer counter.
	`minstret`	`0xB02`	Machine instructions-retired counter.
	`mcycleh`	`0xB80`	Upper 32 bits of `mcycle`.
	`mtimeh`	`0xFE0`	Upper 32 bits of `mtime`.
	`minstreth`	`0xB82`	Upper 32 bits of `minstret`.
Timer Registers	`mtimecmp`	`0xFC1`	Value used to generate timer interuptions.
	`mtimecmph`	`0xFE1`	Upper 32 bits of `mtimecmp`.

Table 1: List of all implemented *CSR* in the *HeRVé* processor.

stage into 6 weakly dependent stages: *Flow*, *WaitExec*, *ALU*, *Jump*, *System* and *Control*. Each of these stage is dedicated to a unique task in the pipeline:

- The *Flow* stage ensures that only valid instructions are executed by the pipeline. This is a key stage of this design, section 3.4 will dive in detail into its role.
- The *ALU* stage computes the result of arithmetic and logic instructions. It cannot fail in the RISC-V specification, that's why no `failcode` field is returned by this stage.
- The *Jump* stage is responsible to compute the result of a jump operation[2]. This role also checks for buffer-overflow exploits, with the embedded shadow stack. Errors are reported with the `failcode` field.
- The *System* stage computes and performs any modification to the processor needed by privileged instructions such as those modifying CSRs or system call instructions. Errors are reported with the `failcode` field.
- The *WaitExec* stage's role is to keep the *Control* synchronized with the previous workers. It follows received information from the *Flow* stage to *Control*.
- The *Control* stage is the most important stage of this pipeline, it is the conductor of the whole "execution pipeline", the 6 stages introduced in the *HeRVé* processor. Section 3.5 will detail what it does.

The stages *ALU*, *Jump*, *System* and *DMem* are called *workers*. They compute the result of a class of instruction. This rework of the *Execute* part of the *CPU* is very flexible. Indeed, the *WaitExec* stage ensures that computation can occur on multiples cycle in *workers*. Until the *Control* stage has consumed the structure provided by *WaitExec*, it will not accept any further instructions. The *WaitExec* stage must wait for its output queue to be emptied, in order to push another structure.

3.4 The *Flow* rule

The *Flow* rule marks the beginning of the execution of an instruction. It acts as a barrier, letting through only the instructions that need to be executed. This stage is responsible for the ordering of the executed instruction. To do so, it uses two "internal" registers:

- The register `nextPC` represents the address of the next instruction to be executed. It is updated accordingly when a jump occurs. On an exception or an interrupt, this register is updated by the *Control* stage. All the instructions entering the execution pipeline have their address checked against this register.
- The register `stalled_exec` acts as a flag, controlling when this stage should fail. It is used to prevent more instructions being fed to in the execution pipeline.

Let's unravel what happens during this stage in the *HeRVé* CPU:

1. This stage tries to consume a decoded instruction from the *Decode* stage. If none is present, it fails. If the `stalled_exec` flag is set or if an interrupt is pending, it will also fail. This is to prevent any instructions from entering the execution pipeline. Indeed, when an interrupt is pending, The execution pipeline must be emptied, to then jump to the handler. This is done by the *Control* stage.

[2] In RISC-V, jump operations produce a value equal to $\texttt{pc} + 4$.

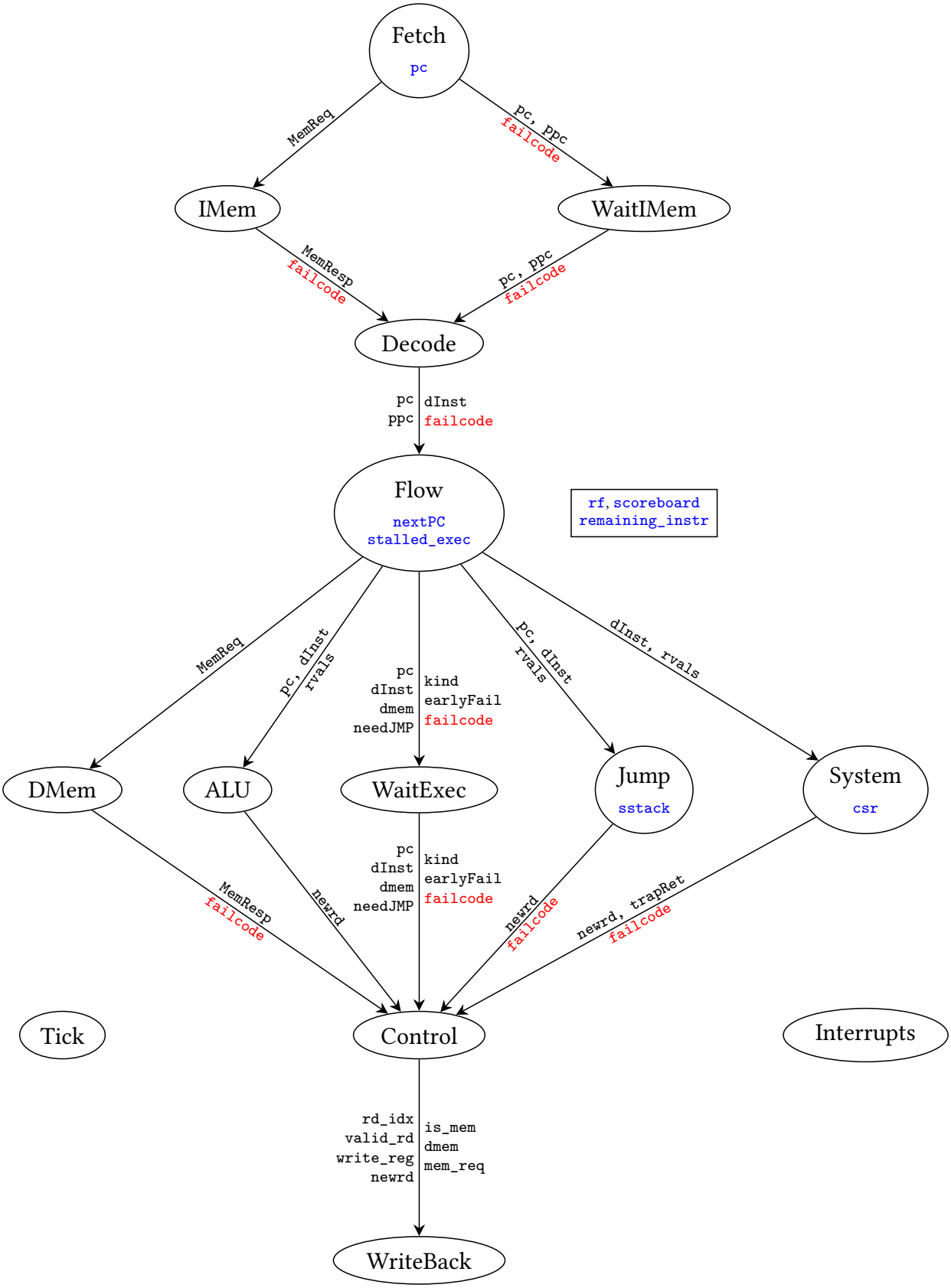

Figure 6: Pipeline of the *HeRVé* processor.

2. Data retrieved from the *Decode* stage must be processed, but it can be invalid. When a jump occurs in the pipeline, whether caused by an instruction or an interrupt, previous stages (*Fetch*, *Decode*, etc.) have worked with invalid values. Their results must be discarded. This is done when `nextPC` is different from the incoming one.

3. Valid incoming data from *Decode* is now known correct. If this data refers to an error (unaligned pc, etc.), it is passed along to *WaitExec*. The flag `stalled_exec` is set, the *Control* stage will deal with the error and restore the pipeline state. Otherwise, the execution of the decoded instruction begins:

   (a) Because the execution of an instruction occurs over multiple cycle, a mechanism should ensure that the values of registers used for the computation are correct. This specificity is represented in figure 7. The result of the second instruction depends on the result of the former. To prevent issues, values of registers are frozen, and the destination registers are marked as dirty. This prevents freezing its value. When the result of this instruction reaches the *WriteBack* stage, the destination register will be marked as clean.

```
1 add t0, a0, a1 // t0 := a0 + a1
2 sub a0, a2, t0 // a0 := a2 - t0
```

   Figure 7: Very simple RISC-V program illustrating dependencies between instructions.

   (b) Next, according to the instruction class, data is given to its corresponding worker.

   (c) At the end of this stage, internal registers are updated. The `remaining_instr` register is incremented. This register counts the number of instructions that have entered the execution pipeline but which have not yet been retired.

The *Flow* rule is also responsible for blocking instructions being executed when a worker reports an error. To do so, read and write *ports* are used. Workers report errors by enabling the `stalled_exec` flag. Because they are "executed" before in the *Kôika* schedule they use the *port* `0`. The *Flow* rule reads this register on port `1` thus seeing any modification made by the workers. This allows for the *Flow* stage to fail in the same cycle that the error is reported by a worker. That's why, to ensure the consistency of the execution, workers must absolutely report errors at the cycle they received the data.

3.5 The *Control* rule

This stage is the one that supervises the whole pipeline. Only this rule can prepare the processor for a PC jump. This is why every jump induced by an instruction (`jal`, `jalr`, `mret`, ...), an exception (unaligned PC, invalid instruction, ...) or an interrupt (timer tick, ...) leads to actions here. This stage is organized in two main sections:

- Retiring instructions from the execution pipeline and committing their results to the registers. This occurs in several steps:
  - The *Control* rule checks if it can consume data from the *WaitExec* stage. If so, it does and thanks to the information passed along, it tries to retrieve the result of the instruction from the corresponding worker. If the worker has not yet finished, this rule fails, giving another cycle for the worker to finish.
  - When no error has occured in processing the instruction, it is counted as retired and the input structure of *WriteBack* is filled to update the value of the destination register. This stage will restore the state of the register as clean.

- The `remaining_instr` register is decremented.
- If the instruction results in a jump or in an error, the registers of the pipeline are changed to perform a change of the instruction address. The PC address is the address of the trap-handler when an exception occurs, or the one specified by the program on a jump. The `pc` register controlling the fetch address is updated. The `nextPC` variable is updated to mark the future instruction as valid in *Flow*. The `stalled_exec` flag is unset to allow the future instruction to enter the execution pipeline.

- Performing a jump when an interrupt is fired. According to the previous section, when an interrupt fires, instructions stop entering the execution pipeline because *Flow* fails at each cycle. The execution pipeline will therefore empty over the next few cycles. When the register `remaining_instr` equals 0, no instructions are being executed, and it is now safe to set up the processor state for a jump at the handler. The jump is done is the same way as a traditional one, a few CSRs are updated to mark entry into a trap. This system avoids jumping immediately when an instruction disabling interrupts was being executed.

4 Experiments and Synthesis

In this section we address the design performances and raw performances of the *HeRVé* processor. The *HeRVé* processor is described in roughly 6760 lines with the *Kôika* HDL. To put it in perspective, the original processor provided as a *Kôika* example is described in roughly 5860 lines. The processor code is available at [7] with the used version of *Kôika* at [8].

Figures 8 and 9 compare the two CPUs. Its content comes from cycle accurate simulations of the two processors on general test programs. Overall, the *HeRVé* processor is a little slower than the *Kôika* one, by about 4 %. This is due to the split of the *Execute* stage by a whole pipeline, with instructions taking slightly longer to execute. To put these processors in perspective, they execute about the same number of instruction per second as an *Intel i486* (1990) or an *ARM3* processor (1990).

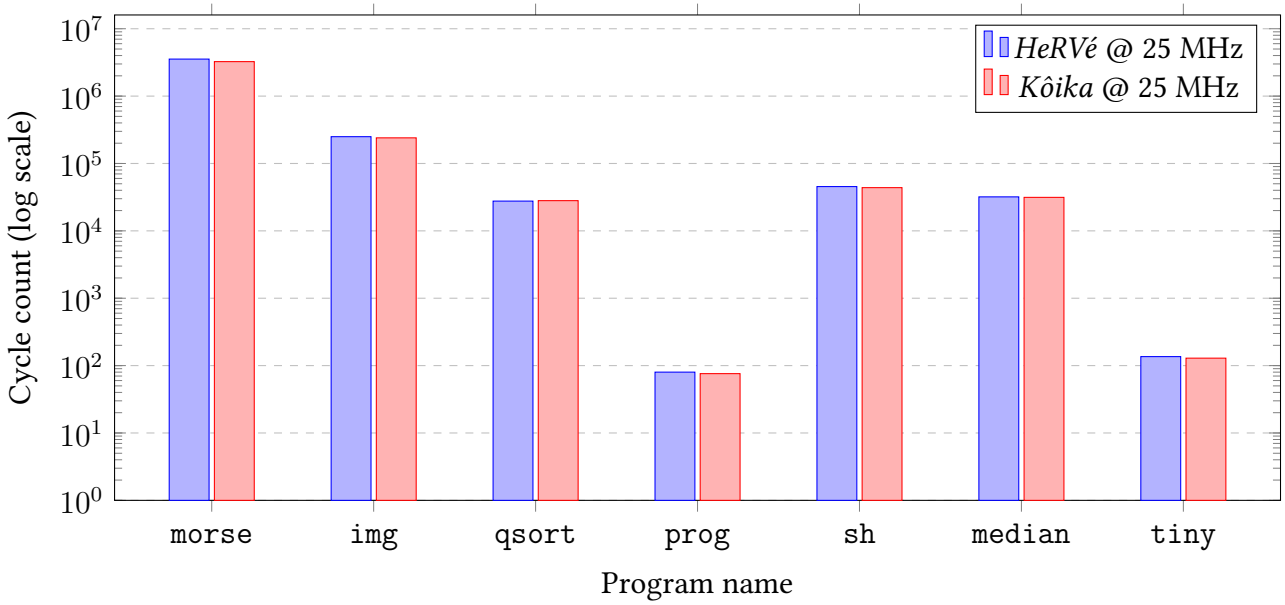

Figure 8: Comparison of the performance of the *HeRVé* and *Kôika* processors in terms of cycle count at a fixed clock frequency of 25 MHz.

These two design have not only been simulated but also synthesized on a FPGA board. I have used the ECPIX5 platform to test the two CPUs and compare them. The synthesis has been done

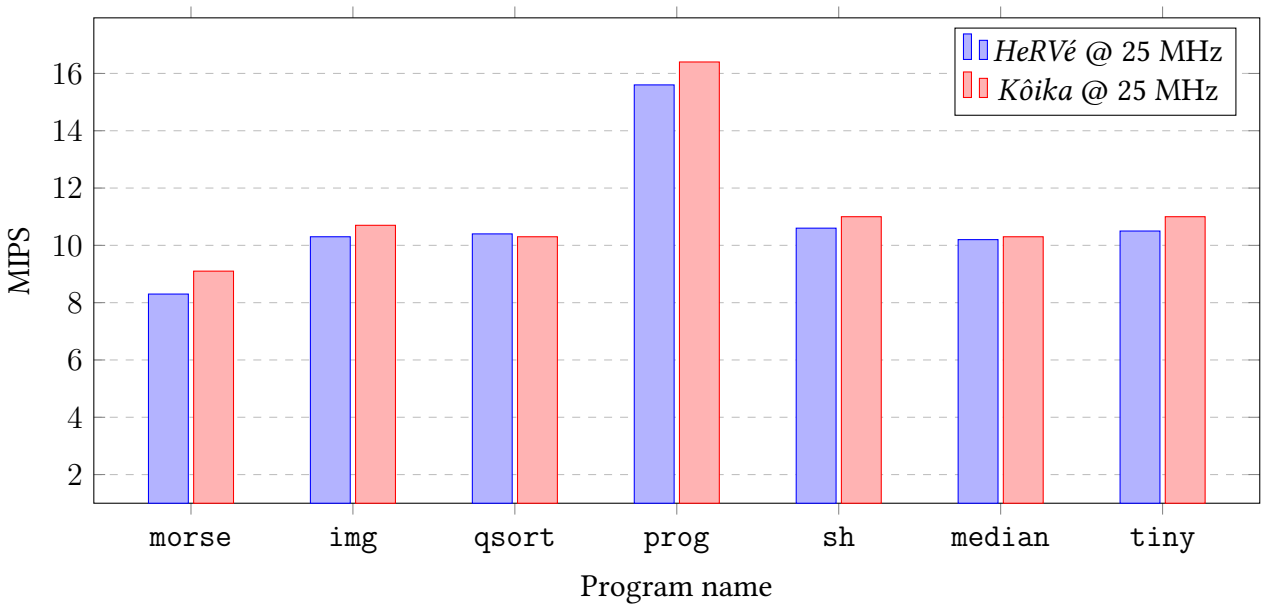

Figure 9: Performance evaluation of the *HeRVé* and *Kôika* processors based on MIPS (Millions of Instructions Per Second) at a constant clock speed of 25 MHz.

with *Yosys* [9] and the placement with *nextpnr* [10]. The *HeRVé* processor performs as expected and simulated on the FPGA. The table 2 resumes the differences of the synthesis of the two processors. A *LUT*, acronym for "Look-Up Table", is the component of FPGAs that enables them to encode any combinatorial logic function. A *D Flip-Flop* is an electronic component used to store data from one cycle to another.

The main information in this table is:

- The *HeRVé* CPU is about two times the size of the *Kôika* one. It uses twice as many as *LUTs* and about a third as many *D Flip-Flops*.
- This increase in size does not have a linear impact on the critical path. In fact, the critical path is only increased by 12 %, which translates into a drop in maximum frequency of the same order. More over, the increase of the critical path is almost completely due to additional routing.

		HeRVé	*Kôika*
Used *LUTs*		12 189 (27 %)	6312 (14 %)
Used *D Flip-Flops*		3090 (7 %)	1961 (4 %)
Critical path	Logic	9.0 ns	8.9 ns
	Routing	29.5 ns	25.3 ns
	Total	38.5 ns	34.2 ns
Max frequency		25.96 MHz	29.19 MHz

Table 2: Statistics on the FPGA synthesis of the two processor with *Yosys* and *nextpnr*.

5 Towards proofs

Being written in *Kôika*, proofs proofs can be made on the *HeRVé* CPU design. For the moment, these proofs can only be made with the SAT with the *Z3* SMT solver [11].

Reasoning on the *HeRVé* processor is a little more challenging than on the previous *Kôika* processor because of its execution pipeline. With the appearance of this pipeline, the conditions ensuring conflict-free execution of the rules are more complex. Moreover, because of the breakdown of the old *Execute* rule, instruction execution now occurs on multiple cycles, that need to be simulated to conduct proofs. This has not yet been done.

The majority of the proofs about the behavior of shadow-stack on the *Kôika* CPU [3] have been reintegrated for the *HeRVé* design. However, these proofs are based on the stronger assumption that the *Jump* rule is executed. This rule is responsible for the checks performed by the shadow stack. But assuring that this rule will run is not obvious. The establishment of formal conditions determining whether the execution of a rule is likely to lead to a conflict has been done for some stages of the *HeRVé* CPU. However, work remains to be done in order to obtain such conditions for all the rules.

6 Conclusion

Learning and designing a RISC-V processor from the example given in *Kôika* was a very interesting project, not without its share of challenges. It took time to design a general-purpose pipeline that was robust and flexible enough to handle interrupts. The result, *HeRVé*, although still simplistic for use as a general-purpose processor, is quite a success. Its performance, given the increase in processor size, is satisfactory. The complexity introduced by all the changes made disturbs the proofs a little, but they can still be carried out automatically using the *Z3* solver.

This project could be continued by adding new security mechanisms to the processor, such as privilege levels or an MMU[3]. Working towards an in-depth formal verification of this design is also possible, and would increase confidence in this processor.

A digital version of this document is available at the following address:
`https://gabriel.desfrene.fr/herve/report.pdf`

[3]Memory Mapping Unit, this is the part of the processor that isolates processes in memory.

References

[1] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. "The essence of Bluespec: a core language for rule-based hardware design". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery (ACM), 2020, pp. 243–257. DOI: 10.1145/3385412.3385965.

[2] *The Coq Proof Assistant*. Version 8.14.1. The Coq development team, Dec. 2021. URL: https://coq.inria.fr/.

[3] Matthieu Baty, Pierre Wilke, Guillaume Hiet, Arnaud Fontaine, and Alix Trieu. "A Generic Framework to Develop and Verify Security Mechanisms at the Microarchitectural Level: Application to Control-Flow Integrity". In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. Institute of Electrical and Electronics Engineers (IEEE), 2023, pp. 372–387. DOI: 10.1109/CSF57540.2023.00029.

[4] *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture*. 20240411th ed. RISC-V International. Apr. 2024.

[5] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. *Koîka*. URL: https://github.com/mit-plv/koika.

[6] *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*. 20240411th ed. RISC-V International. Apr. 2024.

[7] Gabriel Desfrene, Pierre Wilke, and Matthieu Baty. *HeRVé Processor*. Aug. 2024. URL: https://gitlab.inria.fr/SUSHI-public/FMH/herve.

[8] Pierre Wilke and Matthieu Baty. *Koîka mirror of the SUSHI Inria Team*. Aug. 2024. URL: https://gitlab.inria.fr/SUSHI-public/FMH/koika.

[9] Claire Wolf. *Yosys Open SYnthesis Suite*. Version 0.43. July 2024. URL: https://yosyshq.net/yosys/.

[10] *nextpnr – Next Generation Place and Route*. Version 0.7. Jan. 2024. URL: https://github.com/YosysHQ/nextpnr.

[11] Microsoft Research. *Z3 Theorem Prover*. Version 4.13.0. Mar. 2024. URL: https://github.com/Z3Prover/z3.