Work in progress: a formally verified shadow stack for RISC-V

Matthieu BATY (matthieu.baty@inria.fr), Guillaume HIET, Pierre WILKE

CentraleSupélec

Inria

Background — 1/4

"A formally verified **shadow stack** for RISC-V"

Shadow stacks are an example of a **security mechanism**. Security mechanisms help with enforcing **security properties**:

- Confidentiality
- **Integrity**
- Availability

In this work, we consider **hardware-based mechanisms** for mitigating **software-based attacks**.

Background — 2/4

"A formally verified shadow stack for **RISC-V**"

RISC-V:

- Emerging technology
- Open standard
- Numerous open source tools and implementations

Background — 3/4

"A **formally verified** shadow stack for RISC-V"

Formal methods:

- Give **strong guarantees**
- **Exhaustive**, unlike test suites

We use proof assistants, which are **expressive** but **not automatic**.

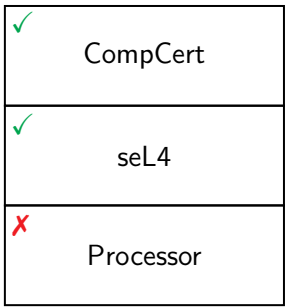
Background — 4/4

An instruction set **architecture** refers to the interface of a processor. Which instructions are available? What is their semantics?

Microarchitecture refers to the organization of an implementation of an architecture. Is it pipelined? How large is the L1 cache?

We consider proofs at the **microarchitectural level**, not at the architectural level.

Motivation



We can't assume the correctness of the hardware

Implementing a processor – 1/3

RISC-V

Processor specification

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if $rs1 < rs2$, 0 otherwise.

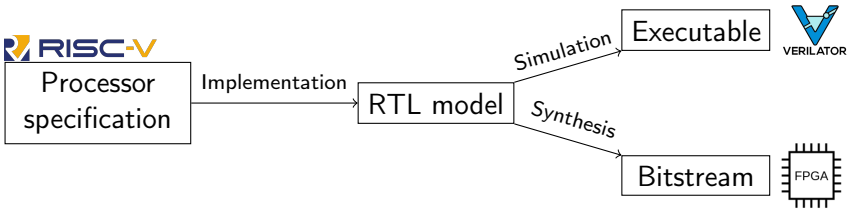
- The specification describes the behavior that an implementation **must** adopt
- Usually not formal

Implementing a processor – 2/3

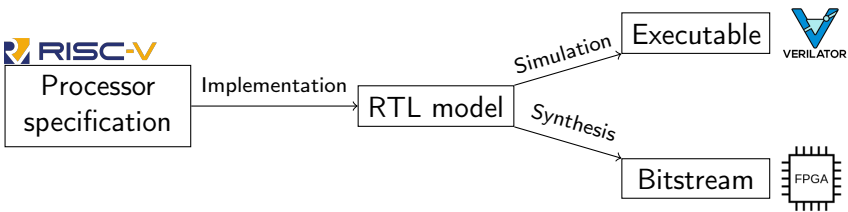

```
//---------------------------------
// Arithmetic
//---------------------------------
`ALU_ADD :
begin
     result_r = (alu_a_i + alu_b_i);
end
```

- An RTL (Register Transfer Level) description of the circuit is used, with:
  - A set of **registers** characterizing the processor
  - **Signals** flowing between registers (can be combined)
- E.g. Verilog, VHDL, Chisel, ...

Implementing a processor – 3/3

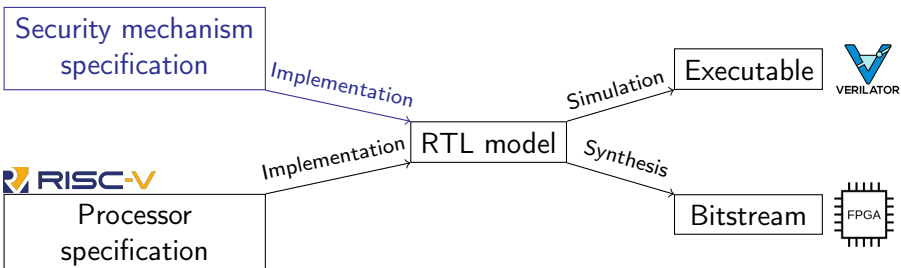


Implementing a processor – 3/3

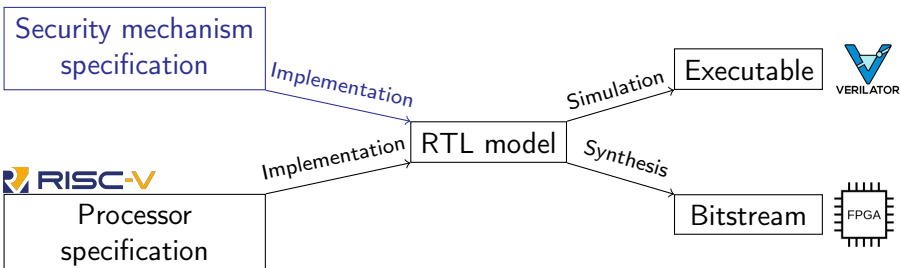


How can we integrate a security mechanism to a processor?

Integration of a security mechanism

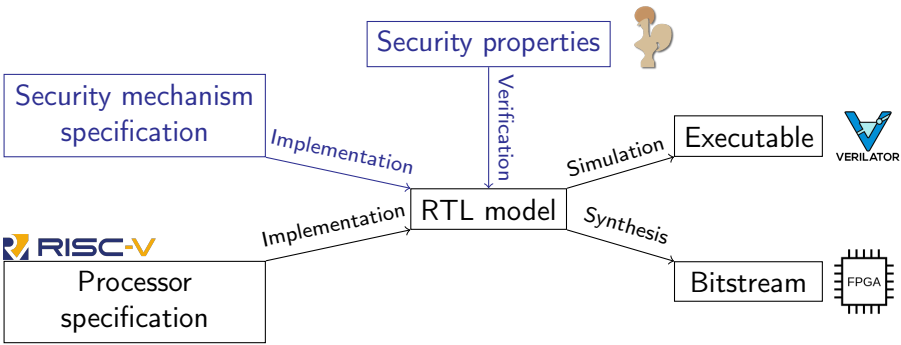


Integration of a security mechanism

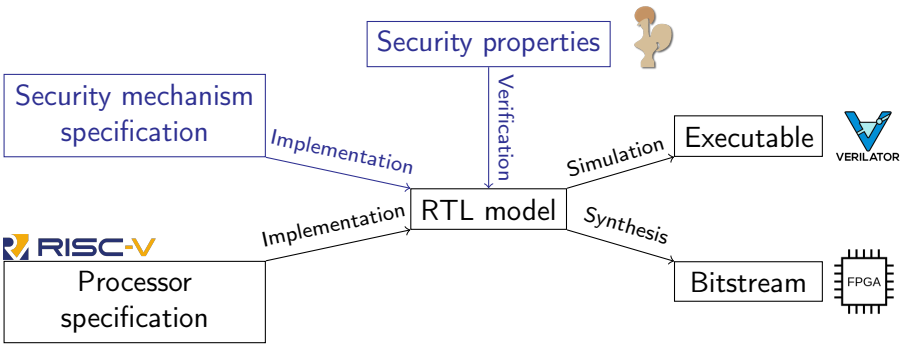


Why should we trust our security mechanism?

Verification of a security mechanism



Verification of a security mechanism



Problem: languages such as Verilog don't have a formal semantics

Formal methods and hardware

- Languages for **formalizing specifications**:
  - Example: Sail[1] for instruction set architectures
  - Microarchitecture out of their scope
- **Formal hardware description languages**:

Name	Active?	Stable?	Verified?	Logic	Inspired by	Output
Kami[2]	✗	✓	✓	Coq	BlueSpec	Verilog
Cava[3]	✓	✗	✗	Coq	Lava	Verilog
CakeML HW[4]	✓	✓	✓	**HOL**	**Verilog**	**Verilog**
Kôika[5]	✓	✓	✓	**Coq**	**BlueSpec**	**Verilog**

[1] "Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics", A. Armstrong et al., ARW 2018

[2] "The Verified IoT Lightbulb: Connecting Hardware and Software in a Simple Embedded System", A. Erbsen et al., PLDI 2020

[3] https://github.com/project-oak/silveroak

[4] "Verified compilation on a verified processor", A. Lööw et al., PLDI 2019

[5] "The Essence of Bluespec", T. Bourgeat et al., PLDI 2020

Formal methods and hardware

- Languages for **formalizing specifications**:
  - Example: Sail[1] for instruction set architectures
  - Microarchitecture out of their scope
- **Formal hardware description languages**:

Name	Active?	Stable?	Verified?	Logic	Inspired by	Output
Kami[2]	✗	✓	✓	Coq	BlueSpec	Verilog
Cava[3]	✓	✗	✗	Coq	Lava	Verilog
CakeML HW[4]	✓	✓	✓	**HOL**	**Verilog**	**Verilog**
Kôika[5]	✓	✓	✓	**Coq**	**BlueSpec**	**Verilog**

[1] "Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics", A. Armstrong et al., ARW 2018

[2] "The Verified IoT Lightbulb: Connecting Hardware and Software in a Simple Embedded System", A. Erbsen et al., PLDI 2020

[3] https://github.com/project-oak/silveroak

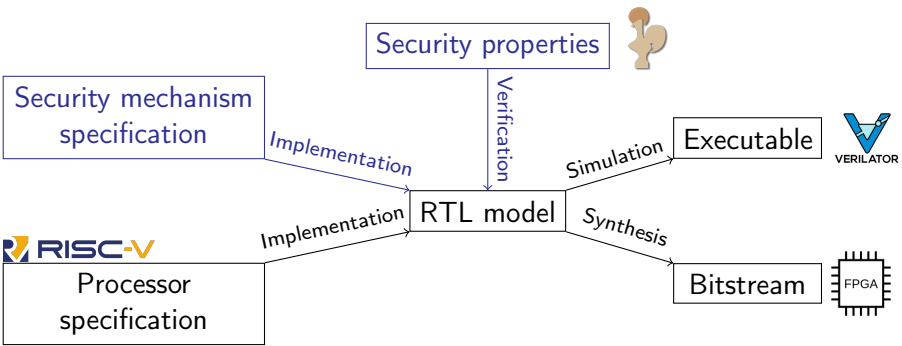
[4] "Verified compilation on a verified processor", A. Lööw et al., PLDI 2019

[5] "The Essence of Bluespec", T. Bourgeat et al., PLDI 2020

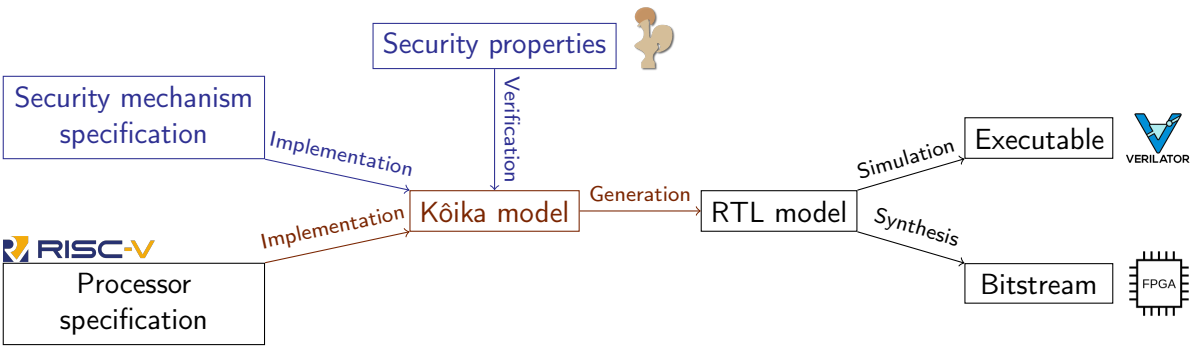
The Kôika language

- "*The Essence of BlueSpec*", PLDI 2020, Thomas Bourgeat et al.
- Open source: `https://github.com/mit-plv/koika`
- Hardware Description Language defined within Coq
- There exists an implementation of a **RISC-V processor** developed in this language:
  - Unprivileged RV32I (typical for IOT)
  - 4 stages
  - No interrupts

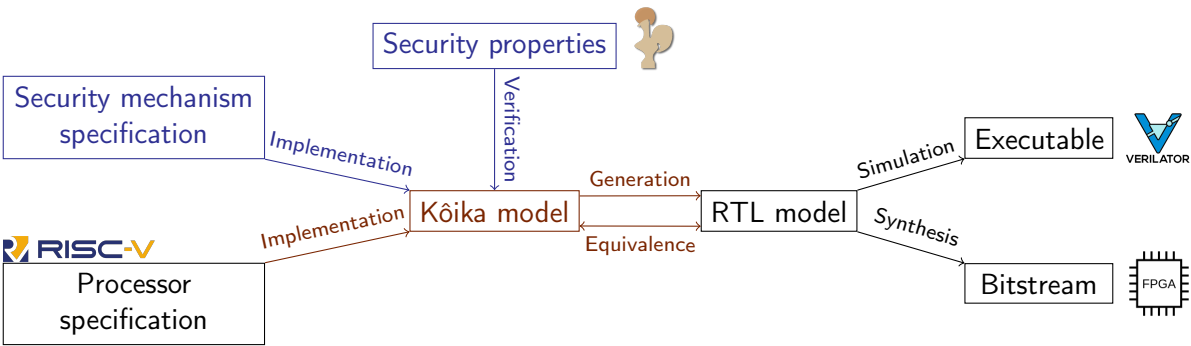
Verification of a security mechanism with Kôika



Verification of a security mechanism with Kôika



Verification of a security mechanism with Kôika



A first example: Collatz sequence

```
registers = [r]

rule divide =
  let v = read0 r in
  if iseven(v) then
    write0 r (v >> 1)

rule multiply =
  let v = read1 r in
  if isodd(v) then
    write1 r (v + v + v + 1)
```

Registers characterize the state of the model.

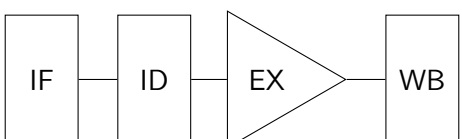
Rules describe how the registers are updated.

For a pipelined processor, you would have one rule per stage.

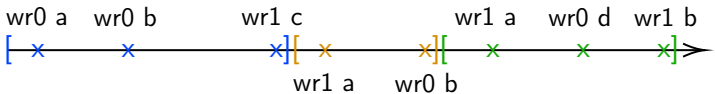
Kôika's design

Parallelism matters in modern hardware and Kôika was built for it:

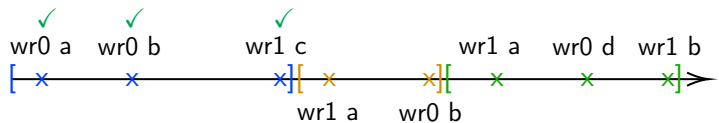
- Rules run **in one cycle**
- They are **atomic**: either they succeed or they are skipped
- They are **parallel**
- All the necessary **control mechanisms** are generated **implicitly** by the compiler: for instance, the stalling behavior is implicit



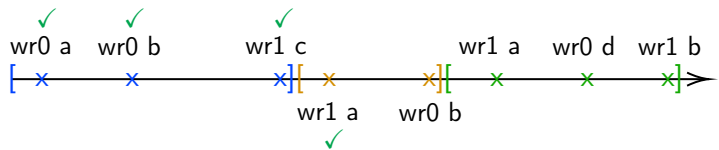
Ports and conflicts



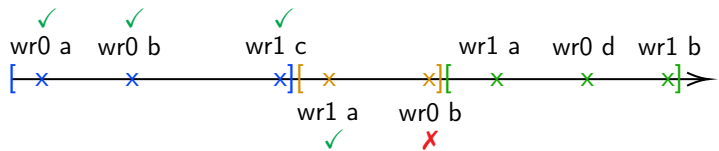
Ports and conflicts



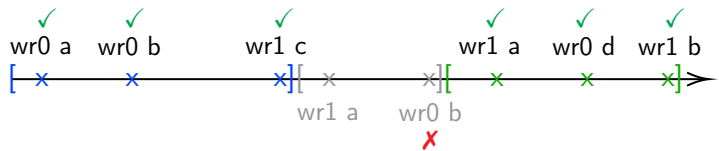
Ports and conflicts



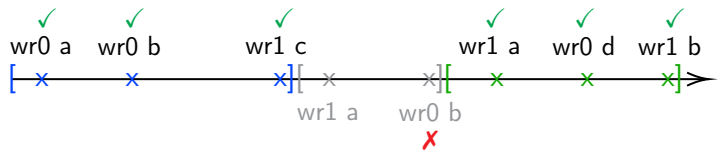
Ports and conflicts



Ports and conflicts

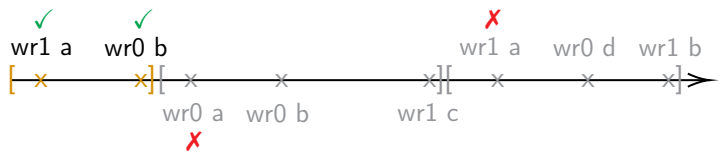


Ports and conflicts



The order in which the effects of the rules are considered is important

Ports and conflicts



The order in which the effects of the rules are considered is important

Summary

We now have what it takes to build a first security mechanism:

- A **formal** hardware description language
- A **RISC-V** processor model

Summary

We now have what it takes to build a first security mechanism:

- A **formal** hardware description language
- A **RISC-V** processor model

How can we integrate our security mechanism?

Choosing a security mechanism

We consider an attacker which can pass **arbitrary input** to a program. Such an attacker can hijack the control-flow by **overwriting the return address** of a procedure through a buffer overflow.

This is an important problem in practice:

- "SoK: Eternal War in Memory", Szekeres et al., 2013 IEEE Symposium on Security and Privacy
- Out-of-bounds writes: first software weakness in the CWE Top 25

Choosing a security mechanism

There are **countermeasures** to control-flow hijacking:

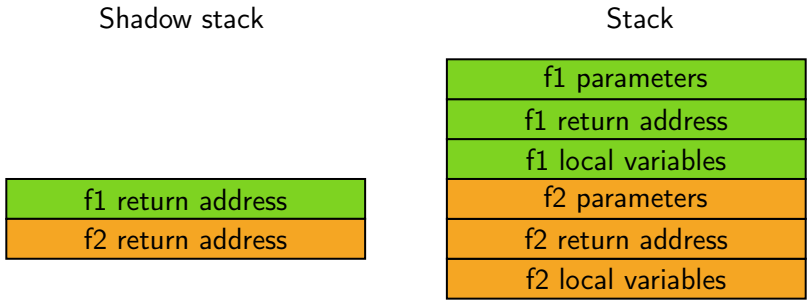
- Stack canaries
- Bounds checking

Most of those are **compiler-based**. They come with some downsides:

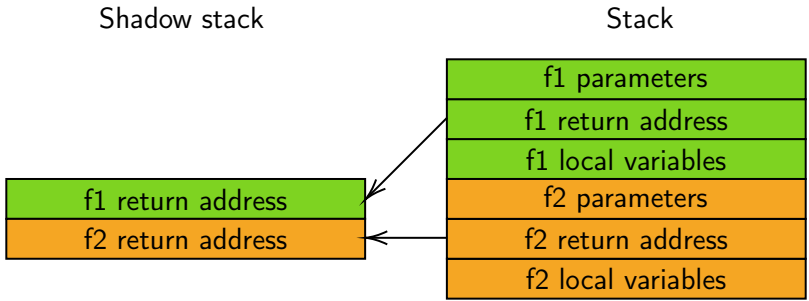
- Performance cost
- Need to be enabled explicitly

A classical **hardware-based** countermeasure is **shadow stacks** (as implemented in Intel CET).

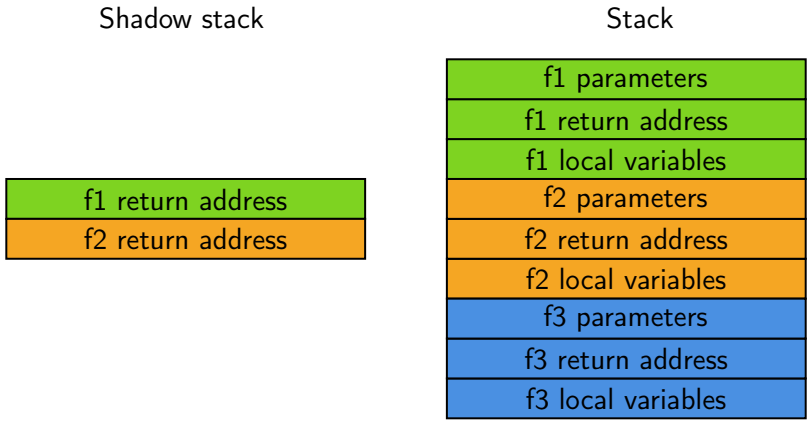
Security mechanism: shadow stack



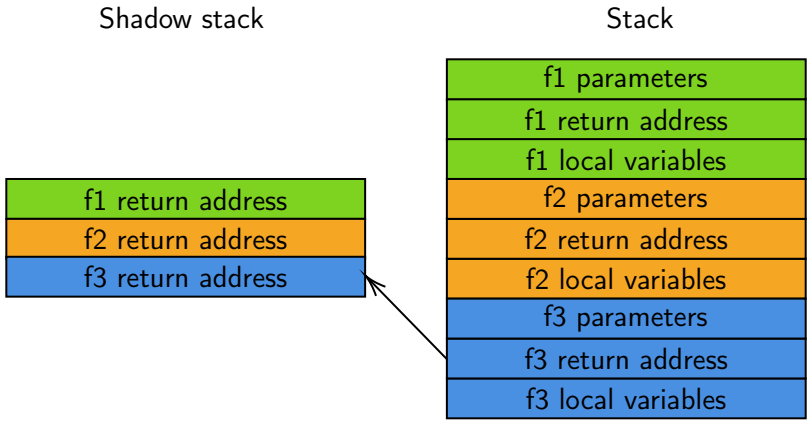
Security mechanism: shadow stack



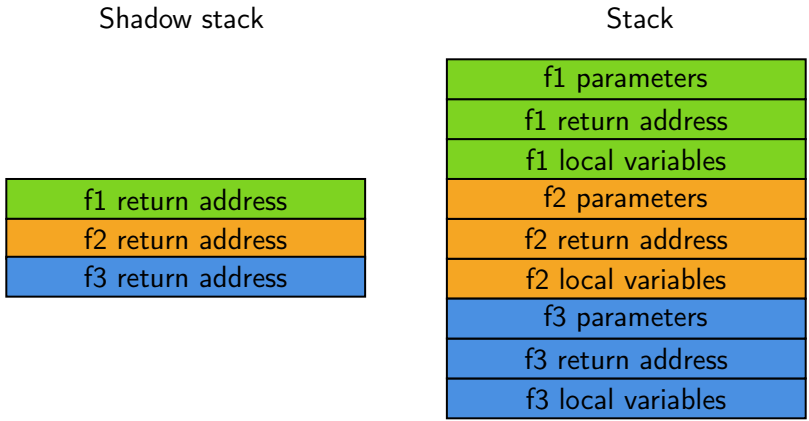
Security mechanism: shadow stack



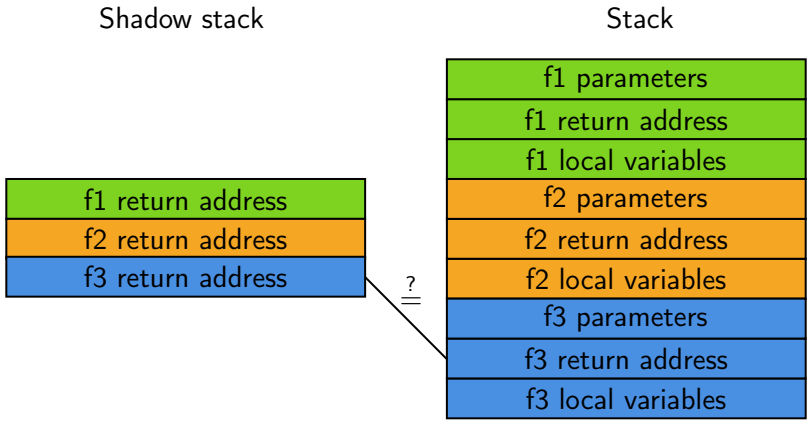
Security mechanism: shadow stack



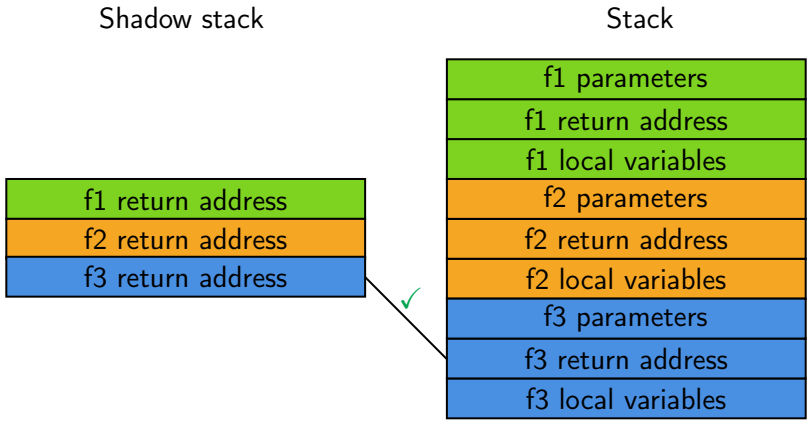
Security mechanism: shadow stack



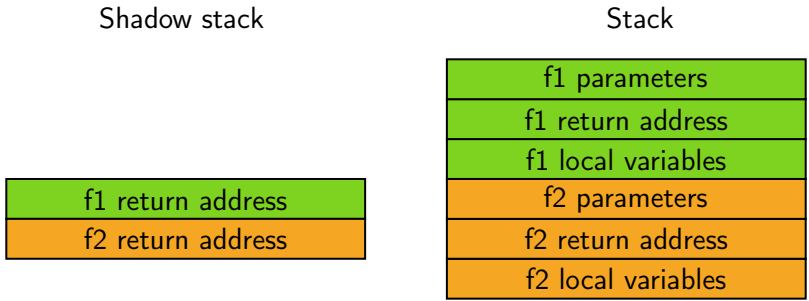
Security mechanism: shadow stack



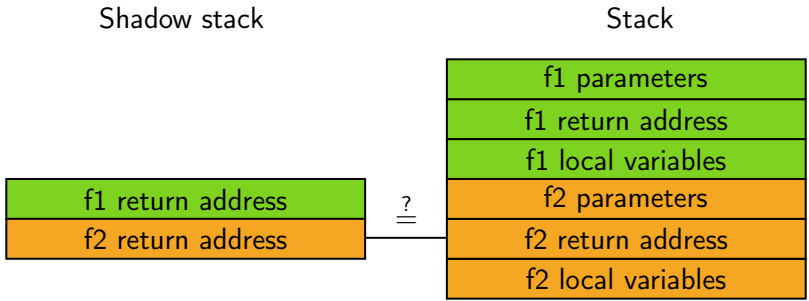
Security mechanism: shadow stack



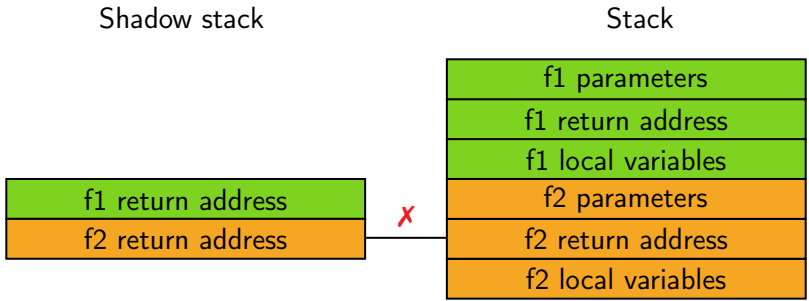
Security mechanism: shadow stack



Security mechanism: shadow stack



Security mechanism: shadow stack



Implementation of the shadow stack

We keep our implementation as **simple** as possible:

- In case of an error, **the processor halts**
- **Hardware mechanism** with no software-side configuration
- The size of the shadow stack is fixed

We are interested in the following **properties**:

- Return to a **modified return address** $\Rightarrow$ halt processor
- **Underflow or overflow** $\Rightarrow$ halt processor
- Otherwise $\Rightarrow$ **behavior preserved**

Summary

We now have what we need for our first proofs:

- A **formal** hardware description language
- A **RISC-V** processor with a **shadow stack** security mechanism
- A set of **properties** to verify

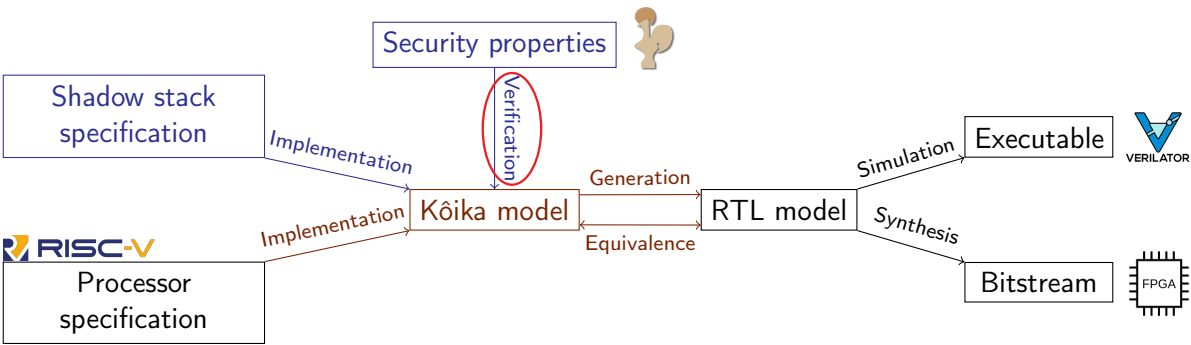
Summary

We now have what we need for our first proofs:

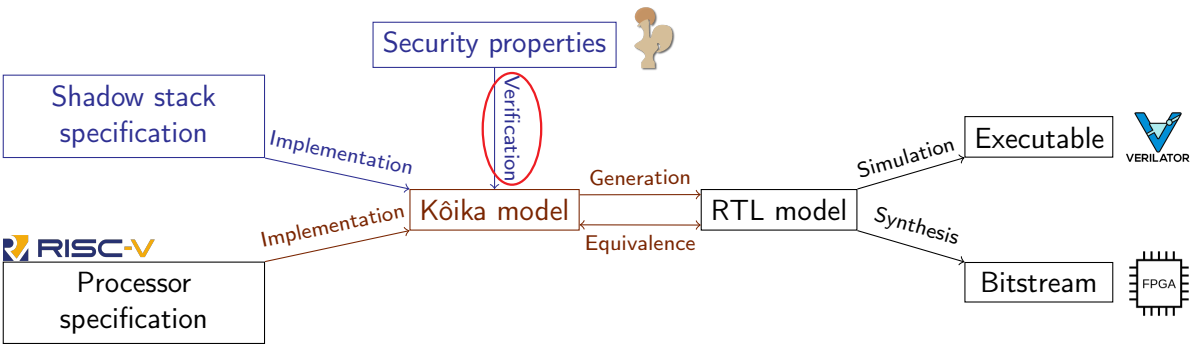
- A **formal** hardware description language
- A **RISC-V** processor with a **shadow stack** security mechanism
- A set of **properties** to verify

How can we prove this mechanism correct?

Proofs on Kôika models

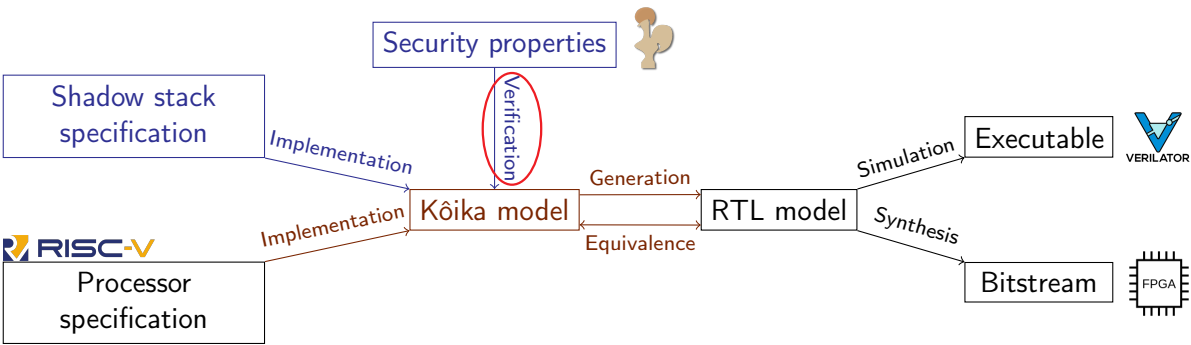


Proofs on Kôika models



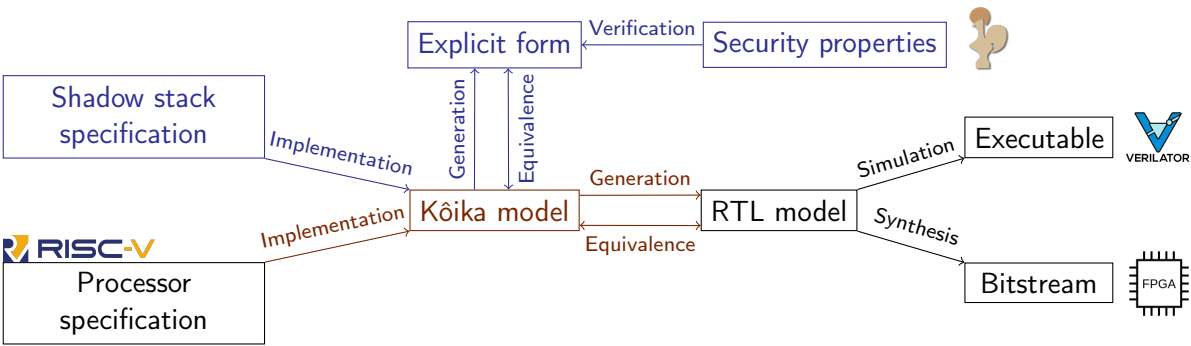
Kôika is a **high-level** language with a **complex semantics**.

Proofs on Kôika models



A **low-level** language with a **simple semantics** is better suited to reasoning.

Proofs on Kôika models



Explicit form

```
registers = [a, b]

rule gcd =
  let v_a = read a in
  let v_b = read b in
  if v_a != 0 then
    if v_a > v_b then
      write0 a v_b;
      write0 b v_a
    else
      write0 a v_a - v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  let v_a = read a in
  let v_b = read b in
  if v_a != 0 then
    if v_a > v_b then
      write0 a v_b;
      write0 b v_a
    else
      write0 a v_a - v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  if v_a != 0 then
    if v_a > v_b then
      write0 a v_b;
      write0 b v_a
    else
      write0 a v_a - v_b
```

```
v_a := read a
v_b := read b
```

Explicit form

```
registers = [a, b]

rule gcd =
  if v_a != 0 then
    if v_a > v_b then
      write0 a v_b;
      write0 b v_a
    else
      write0 a v_a - v_b
```

```
v_a := read a
v_b := read b
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    if comp2 then
      write0 a v_b;
      write0 b v_a
    else
      write0 a sub
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    if comp2 then
      write0 a v_b;
      write0 b v_a
    else
      write0 a sub
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    write0 a
      (if comp2 then v_b else sub)
    if comp2 then
      write0 b v_a
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    write0 a e1
    if comp2 then
      write0 b v_a
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    write0 a e1
    if comp2 then
      write0 b v_a
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    write0 a e1
    write0 b
      (if comp2 then v_a)
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    write0 a e1
    write0 b
      (if comp2 then v_a else v_b)
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    write0 a e1
    write0 b e2
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  if comp1 then
    write0 a e1
    write0 b e2
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  write0 a (if comp1 then e1)
  write0 b (if comp1 then e2)
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  write0 a (if comp1 then e1 else v_a)
  write0 b (if comp1 then e2 else v_b)
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
```

Explicit form

```
registers = [a, b]

rule gcd =
  write0 a w_a
  write0 b w_b
```

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
w_a := if comp1 then e1 else v_a
w_b := if comp1 then e2 else v_b
```

Explicit form

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
w_a := if comp1 then e1 else v_a
w_b := if comp1 then e2 else v_b
```

Explicit form

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
w_a := if comp1 then e1 else v_a <- final value of a
w_b := if comp1 then e2 else v_b <- final value of b
```

Explicit form

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
w_a := if comp1 then e1 else v_a <- final value of a
w_b := if comp1 then e2 else v_b <- final value of b
```

How can we prove properties about models in this form?

Structure of a proof

Let's consider the following property :

When the value of a is 0, the registers are not updated during a cycle.

We can exploit the information that $a = 0$ at the beginning of a cycle.

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_a := read a
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
w_a := if comp1 then e1 else v_a
w_b := if comp1 then e2 else v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_a := 0
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
w_a := if comp1 then e1 else v_a
w_b := if comp1 then e2 else v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_a := 0
v_b := read b
comp1 := v_a != 0
comp2 := v_a > v_b
sub := v_a - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then v_a else v_b
w_a := if comp1 then e1 else v_a
w_b := if comp1 then e2 else v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
--
v_b := read b
comp1 := 0 != 0
comp2 := 0 > v_b
sub := 0 - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then 0 else v_b
w_a := if comp1 then e1 else 0
w_b := if comp1 then e2 else v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_b := read b
comp1 := 0 != 0
comp2 := 0 > v_b
sub := 0 - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then 0 else v_b
w_a := if comp1 then e1 else 0
w_b := if comp1 then e2 else v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_b := read b
comp1 := false
comp2 := 0 > v_b
sub := 0 - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then 0 else v_b
w_a := if comp1 then e1 else 0
w_b := if comp1 then e2 else v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_b := read b
--
comp2 := 0 > v_b
sub := 0 - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then 0 else v_b
w_a := if false then e1 else 0
w_b := if false then e2 else v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_b := read b
comp2 := 0 > v_b
sub := 0 - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then 0 else v_b
w_a := if false then e1 else 0
w_b := if false then e2 else v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_b := read b
comp2 := 0 > v_b
sub := 0 - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then 0 else v_b
w_a := 0
w_b := v_b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_b := read b
comp2 := 0 > v_b
sub := 0 - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then 0 else v_b
w_a := 0   <- final value of a
w_b := v_b <- final value of b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_b := read b
comp2 := 0 > v_b
sub := 0 - v_b
e1 := if comp2 then v_b else sub
e2 := if comp2 then 0 else v_b
w_a := 0   <- final value of a
w_b := v_b <- final value of b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
v_b := read b
w_a := 0   <- final value of a
w_b := v_b <- final value of b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
--
w_a := 0      <- final value of a
w_b := read b <- final value of b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
w_a := 0      <- final value of a
w_b := read b <- final value of b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
w_a := 0      <- final value of a
w_b := read b <- final value of b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
w_a := 0       <- final value of a
w_b := read b <- final value of b
```

Structure of a proof

When the value of a is 0, the registers are not updated during a cycle.

```
w_a := 0 <- final value of a
```

Structure of a proof

We have to **prove** that all the simplifications we apply (value replacement, expression simplification, ...) are **correct**.

Most of the tactics that we developed are not specific to our model and **can be reused**.

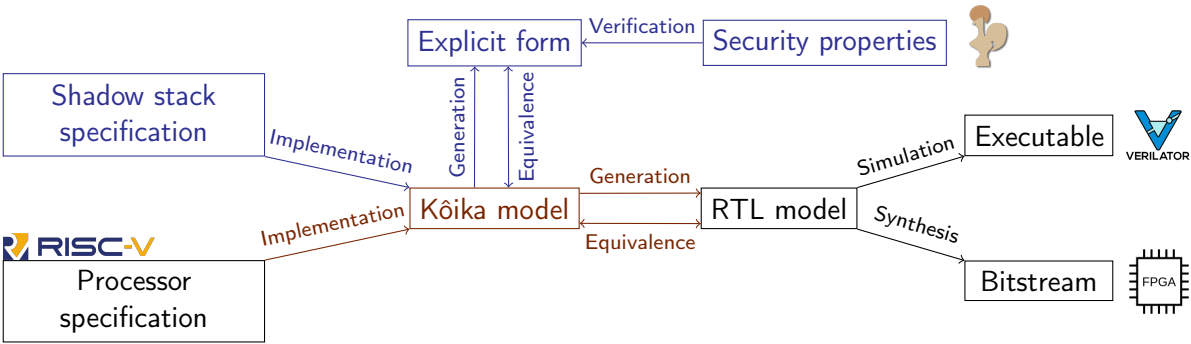
Current state of our work

We have some **intermediate proofs**:

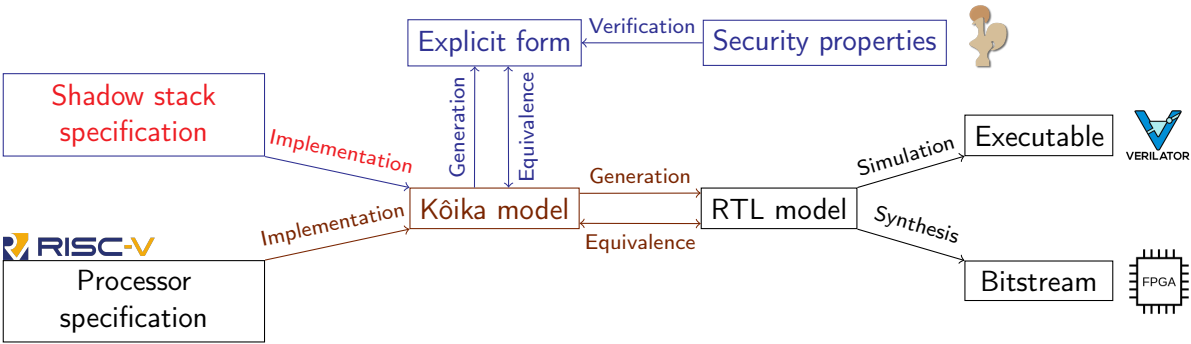
- The halt state is a sink state
- Overflows $\Rightarrow$ halt

The main element we are missing is a way of **exploiting partial information** about the value of registers (some simplifications only depend on the value of some bits in a word, and we don't always know the value of each bit).

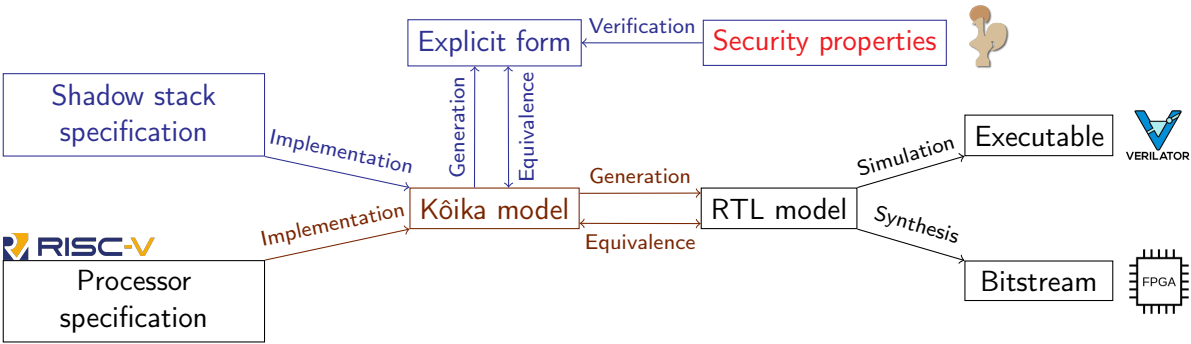
Recap


Explicit form
Verification
Security properties
Shadow stack specification
Implementation
Generation
Equivalence
Kôika model
Generation
Equivalence
RTL model
Simulation
Executable
VERILATOR
Synthesis
Bitstream
FPGA
Implementation
RISC-V
Processor specification

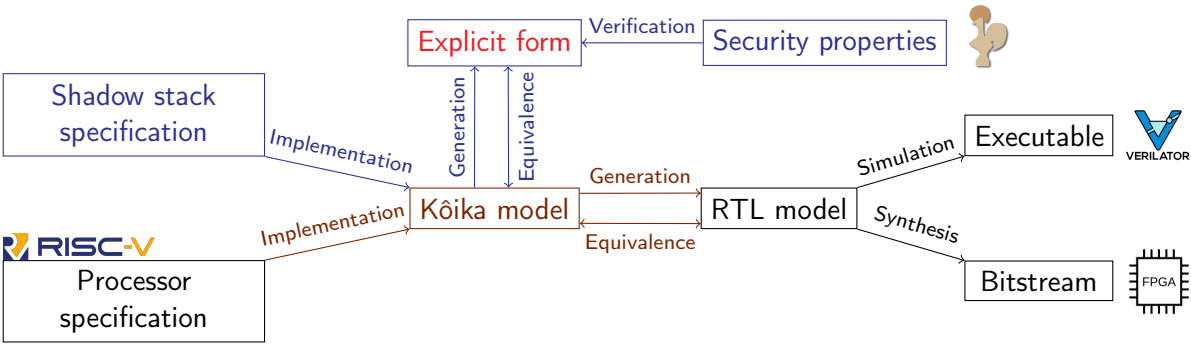
Recap


Explicit form
Verification
Security properties
Shadow stack specification
Implementation
Generation
Equivalence
Kôika model
Generation
Equivalence
RTL model
Simulation
Executable
VERILATOR
Synthesis
Bitstream
FPGA
Implementation
RISC-V
Processor specification

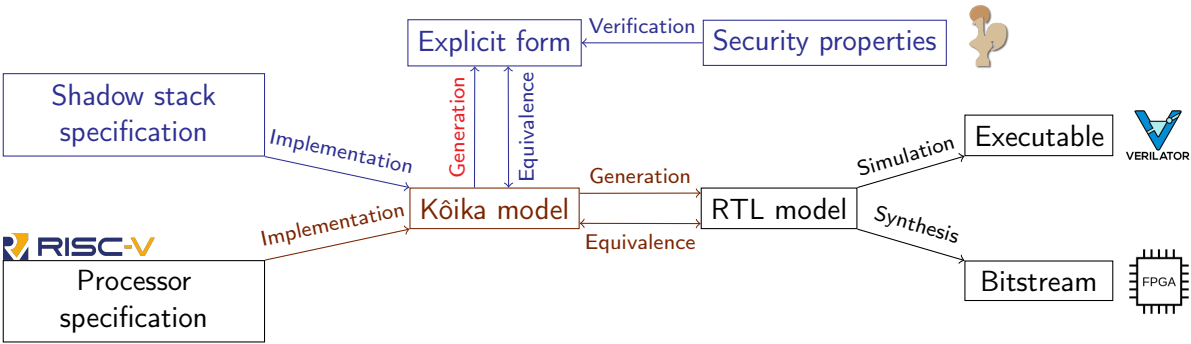
Recap


Explicit form
Verification
Security properties
Shadow stack specification
Implementation
Generation
Equivalence
Kôika model
Generation
Equivalence
RTL model
Simulation
Executable
VERILATOR
Synthesis
Bitstream
FPGA
Implementation
RISC-V
Processor specification

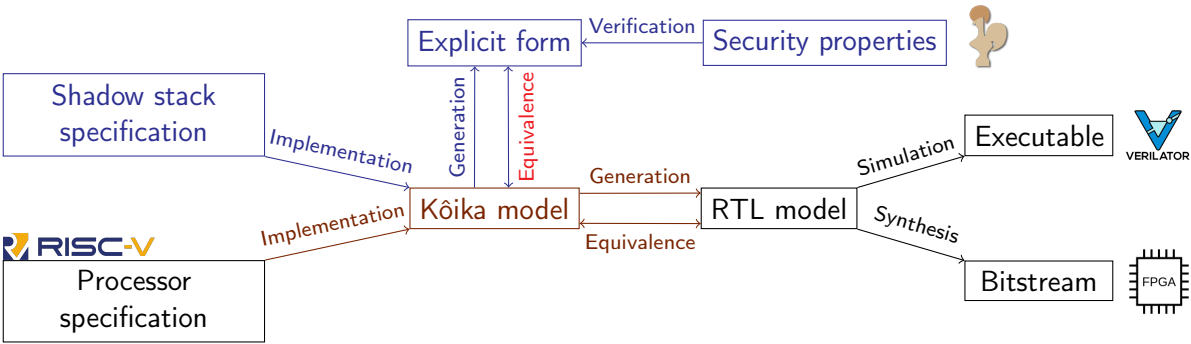
Recap


Explicit form
Verification
Security properties
Shadow stack specification
Implementation
Generation
Equivalence
Kôika model
Generation
Equivalence
RTL model
Simulation
Executable
VERILATOR
Synthesis
Bitstream
FPGA
Implementation
RISC-V
Processor specification

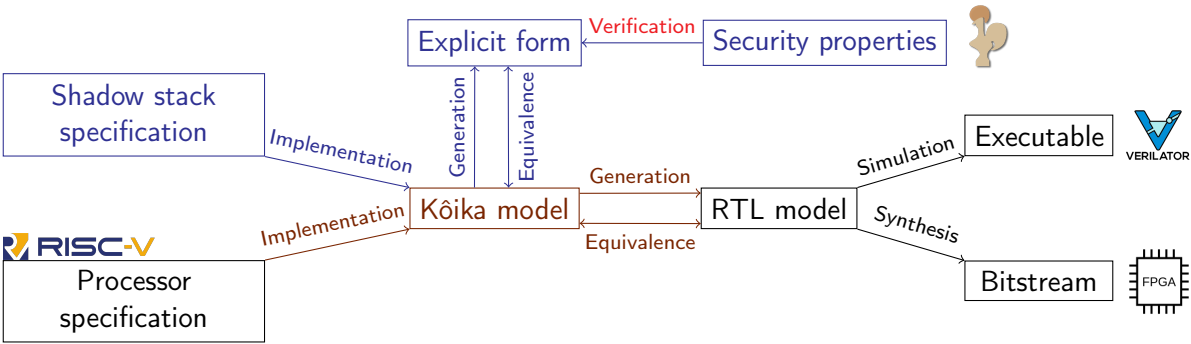
Recap


Explicit form
Verification
Security properties
Shadow stack specification
Implementation
Generation
Equivalence
Generation
Simulation
Executable
VERILATOR
Implementation
Kôika model
RTL model
Equivalence
Synthesis
RISC-V
Processor specification
Bitstream
FPGA

Recap


Explicit form
Verification
Security properties
Shadow stack specification
Implementation
Generation
Equivalence
Kôika model
Generation
Equivalence
RTL model
Simulation
Executable
VERILATOR
Synthesis
Bitstream
FPGA
Implementation
RISC-V
Processor specification

Recap


Explicit form
Verification
Security properties
Shadow stack specification
Implementation
Generation
Equivalence
Kôika model
Generation
Equivalence
RTL model
Simulation
Executable
VERILATOR
Synthesis
Bitstream
FPGA
Implementation
RISC-V
Processor specification

Future works

We want to reuse the tooling we developed on **more complex examples**. In particular, we want to consider **interactions with the software**.

We are considering the following mechanisms:

- More realistic shadow stacks (context switching)
- Entirely different mechanisms traditionally implemented in software