Porting a JIT Compiler to RISC-V: Challenges and Opportunities

Quentin Ducasse [1] Guillermo Polito [2] Pablo Tesone [2]
Pascal Cotret [1] Loïc Lagadec [1]

September 15, 2022

(1) ENSTA Bretagne - LabSTICC
(2) INRIA Lille - RMoD

Outline

Introduction

September 06, 2022 |

NASA Selects SiFive and Makes RISC-V the Go-to Ecosystem for Future Space Missions

Learn More

Introduction

September 06, 2022 |

NASA Selects SiFive and Makes RISC-V the Go-to Ecosystem for Future Space Missions

Learn More

Our main objective with **RISC-V** is to ~~send **Pharo** to the moon~~:

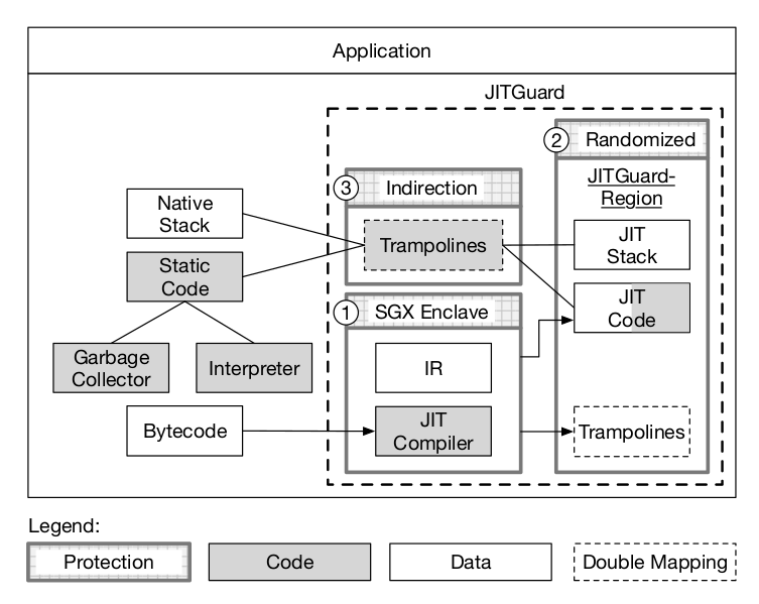
- Experiment with **dedicated VM custom instructions**
- Dedicate hardware to **security or media processing**

Big Picture

Hardware-based security enforcement of **JITed language runtimes**...:

- **Isolate parts** of the VM
- Protect **JIT Compilation** and **JIT code**
- Enforce **strong properties** through **hardware**

... on **RISC-V**!



extracted from JITGuard by Frassetto et al.

Background

Pharo - Language

The Pharo language is:

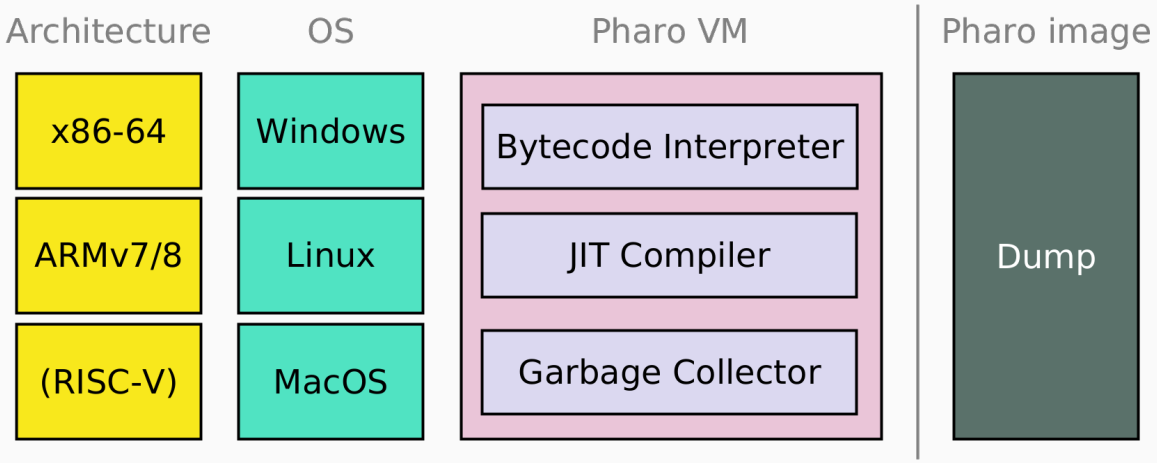
- **Smalltalk-inspired**
- Purely **object-oriented**
- **Dynamically-typed**
- Control flow comes as **message passing**

```
exampleWithNumber: x
  <aMethodAnnotation>
  | y |
  true & false not & (nil isNil)
  ifFalse: [ self halt ].
  y := self size + super size
  #($a #a 'a' 1 1.0)
  do: [ :each | Transcript
    show: (each class name);
    show: (each printString);
    show: ' '].
    ^ x < y
```

Pharo - VM and Images

The runtime environment is the Pharo VM, it is composed of:

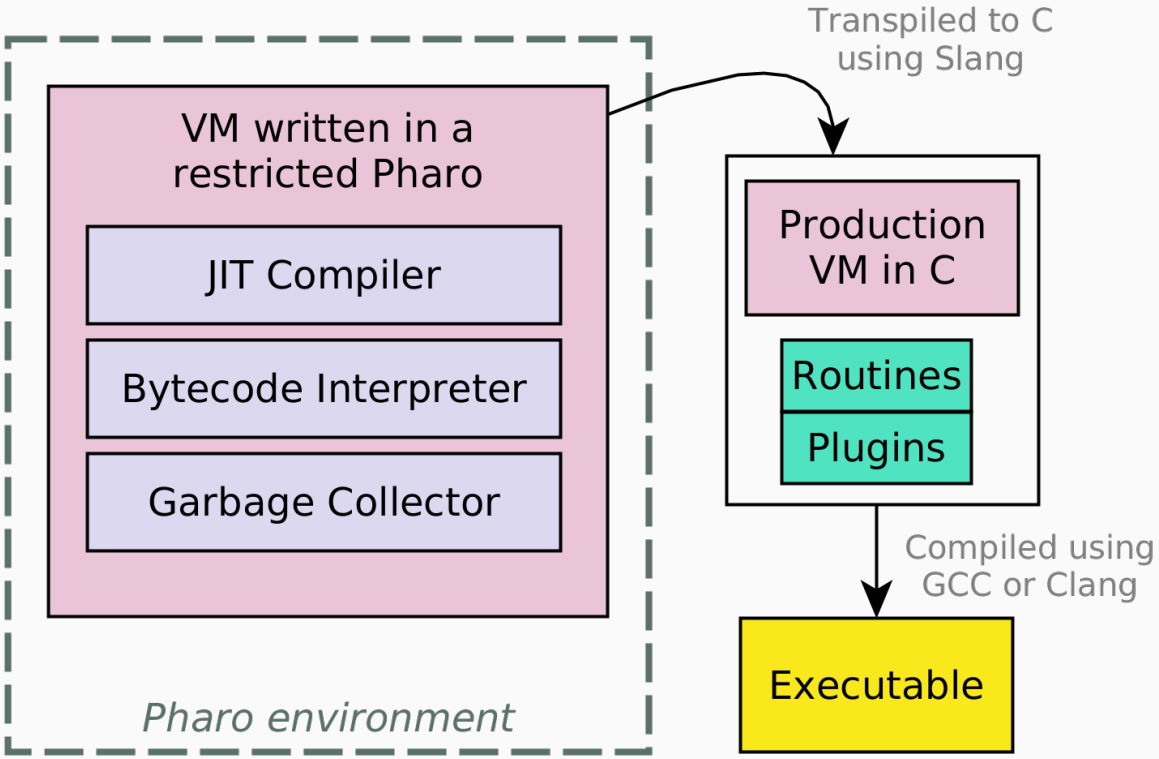
- A threaded bytecode **interpreter**
- A linear non-optimising **JIT compiler**
- A generational scavenger **garbage collector**



Pharo - VM Compilation

The VM is compiled by:

- Writing the VM in a **restricted Pharo language**
- **Transpiling** the restricted VM to C (*Slang*)
- **Compiling** it with a C compiler along with **routines and plugins**



RISC-V - Presentation

RISC-V was born in Berkeley around **2010**.

It is the most recent generation of **RISC processors**.

The ISA is:

- **open-source** - multiple cores and implementations are available
- **extensible** - opcode space available for dedicated hardware
- **modular** - wide range of application from IoT to HPC

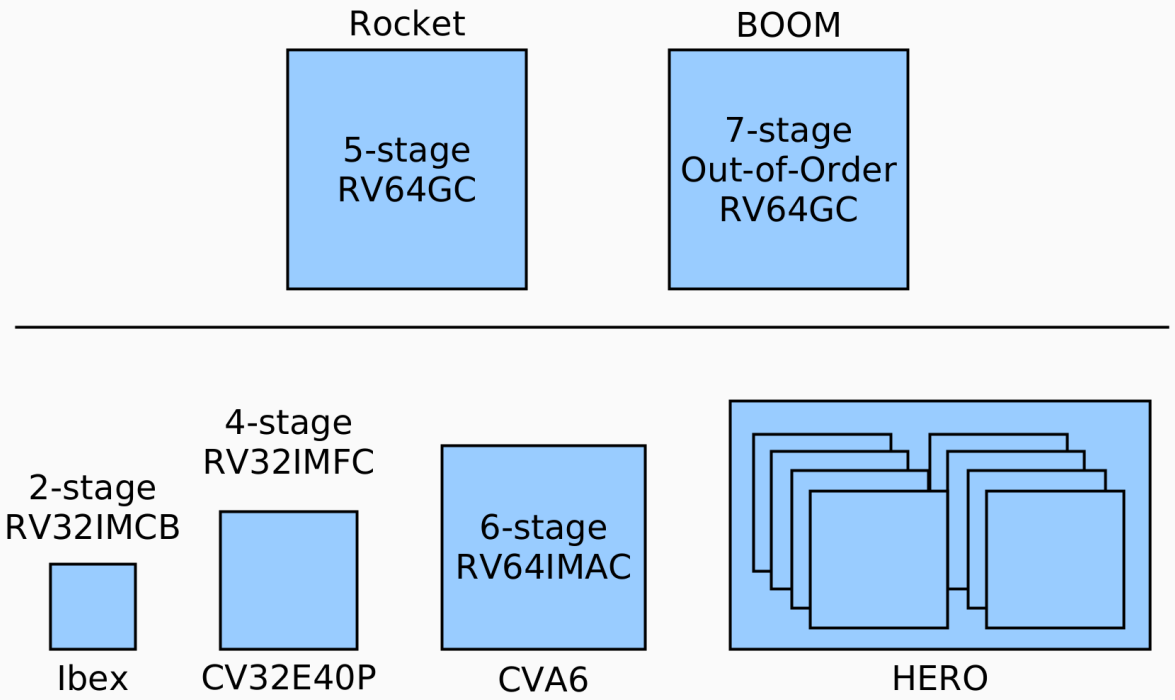
RISC-V - Main Extensions

Name	Description	State	Instructions
RV32I	Base Integer Instruction Set - 32 bits	Frozen	49
RV64I	Base Integer Instruction Set - 64 bits	Frozen	14
M	Integer Multiplication and Division	Frozen	8
A	Atomic Instructions	Frozen	11
F	Single-Precision Floating-Point	Frozen	25
D	Double-Precision Floating-Point	Frozen	25
G	All of the above	-	-
C	Compressed Instructions	Frozen	36
J	Dynamically Translated Languages	Open	undefined
T	Packed-SIMD Instructions	Open	undefined
N	User-Level Interrupts	Open	3
Z*	Cryptographic operations	Open	undefined

Table 1: RISC-V ISA and extensions

RISC-V - Open-source Processors

RISC-V cores come in different sizes and capacities **from IoT to HPC**:



Repositories in references!

RISC-V Implementation Details

RISC-V Features - Simpler Instructions

RISC-V honors the **Reduced** part of the instruction set, choosing simplicity as a main design focus:

- **One** data addressing mode (adding a sign-extended 12-bit immediates to a register)
- **No shifts** in arithmetic-logic operations
- Only **general purpose registers** (with the addition of PC and hardwired 0)
- No complex **call/return** or **stack instructions**

RISC-V Features - Simpler Instructions

RISC-V honors the **Reduced** part of the instruction set, choosing simplicity as a main design focus:

- **One** data addressing mode (adding a sign-extended 12-bit immediates to a register)
- **No shifts** in arithmetic-logic operations
- Only **general purpose registers** (with the addition of PC and hardwired 0)
- No complex **call/return** or **stack instructions**

Rationale

Common operations should be the norm, leaving **complex instructions** at the charge of the developer. Simplification of the **datapath**!

Impact

Redefinition of needed rare instructions. Increase of the number of instructions.

```
# Rotate left with shift amount in register
sll rd, rs1, rshamt      # x[rs1] << rshamt
sub temp, zero, rshamt   # get the negative count
srl temp, rs1, temp      # x[rs1] >> (xlen - rshamt)
or rd, rd, temp          # or between (1) and (2)

# Software overflow check
add  t0, t1, t2          # genuine addition
slti t3, t2, 0           # t3 = t2's sign
slt  t4, t0, t1          # t4 = sum smaller than t1?
bne  t3, t4, overflow    # if t3 != t4, overflow!
```

RISC-V Features - Simpler Instructions

Pseudo-instruction `li`, available in RISC-V assembly is defined as:

li rd, immediate `x[rd] = immediate`

Load Immediate. Pseudoinstruction, RV32I and RV64I.

Loads a constant into x[*rd*], using as few instructions as possible. For RV32I, it expands to **lui** and/or **addi**; for RV64I, it's as long as **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**.

RISC-V Features - Simpler Instructions

Pseudo-instruction `li`, available in RISC-V assembly is defined as:

li rd, immediate `x[rd] = immediate`

Load Immediate. Pseudoinstruction, RV32I and RV64I.

Loads a constant into x[*rd*], using as few instructions as possible. For RV32I, it expands to **lui** and/or **addi**; for RV64I, it's as long as **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**.

- **LLVM** defines a complex recursive function to handle all immediate values in the fewest instructions possible.
- **GCC** runs different encoding methods, attributes them a cost and returns the best fitting choice.

RISC-V Features - Sign-extension

As extracted from the RISC-V specifications [5]:

> *Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended [...]. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry*

RISC-V Features - Sign-extension

As extracted from the RISC-V specifications [5]:

> *Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended [...]. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry*

Rationale

A single convention makes manipulating immediates **more reliable**!
Architecture has a **dedicated encoding/decoding circuitry**.

RISC-V Features - Sign-extension

As extracted from the RISC-V specifications [5]:

> *Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended [...]. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry*

Rationale

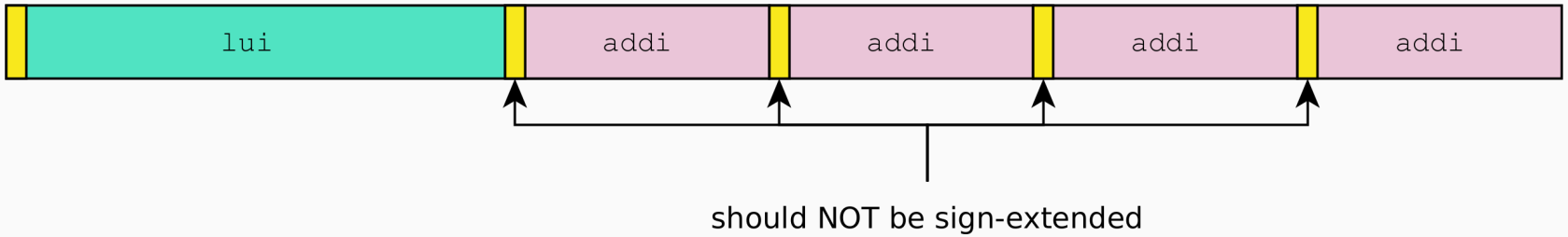
A single convention makes manipulating immediates **more reliable**! Architecture has a **dedicated encoding/decoding circuitry**.

Impact

Large immediates split through multiple instructions will require bit manipulation and a **check at the smallest unit size**.

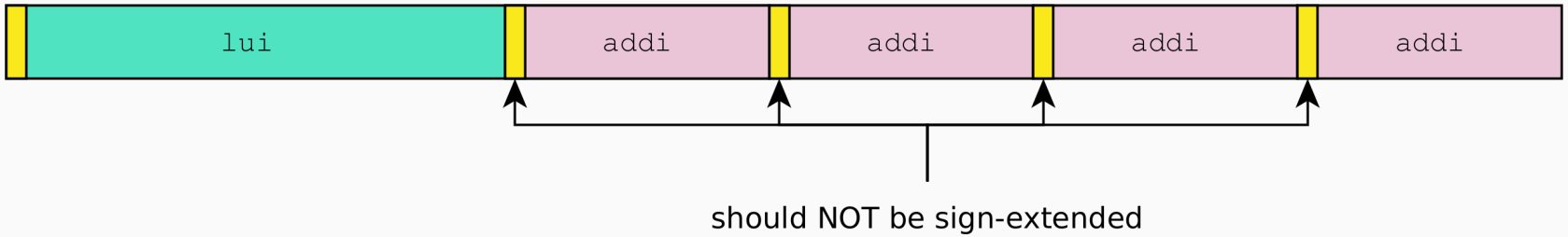
RISC-V Features - Sign-extension

```
li t0, 0x38008008008008 00
    lui     t0, 14337      # 0x3801
    addiw   t0, t0, -2047  # 0x7FF
    slli    t0, t0, 12     #
    addi    t0, t0, -2047  # 0x7FF
    slli    t0, t0, 12     #
    addi    t0, t0, -2047  # 0x7FF
    slli    t0, t0, 12     #
    addi    t0, t0, -2048  # 0x800
```



RISC-V Features - Sign-extension

```
li t0, 0x38008008008008 00
    lui     t0, 14337      # 0x3801
    addiw   t0, t0, -2047  # 0x7FF
    slli    t0, t0, 12     #
    addi    t0, t0, -2047  # 0x7FF
    slli    t0, t0, 12     #
    addi    t0, t0, -2047  # 0x7FF
    slli    t0, t0, 12     #
    addi    t0, t0, -2048  # 0x800
```



Note: Also applies to the `call` pseudo-instruction - `auipc/jalr`

RISC-V Features - Condition Codes

Regarding conditional branches, RISC-V rejects:

- **Condition codes** of ARM/x86
- **Delayed branch** of MIPS
- **Loop instructions** of x86

Instead, it provides a way to compare two registers and branch on the result: `beq`, `bne`, `bge` and `blt`.

RISC-V Features - Condition Codes

Regarding conditional branches, RISC-V rejects:

- **Condition codes** of ARM/x86
- **Delayed branch** of MIPS
- **Loop instructions** of x86

Instead, it provides a way to compare two registers and branch on the result: `beq`, `bne`, `bge` and `blt`.

Rationale

Condition codes added extra state that is implicitly set by most instructions. It complicates out-of-order execution processor design.

RISC-V Features - Condition Codes

Regarding conditional branches, RISC-V rejects:

- **Condition codes** of ARM/x86
- **Delayed branch** of MIPS
- **Loop instructions** of x86

Instead, it provides a way to compare two registers and branch on the result: `beq`, `bne`, `bge` and `blt`.

Rationale

Condition codes added extra state that is implicitly set by most instructions. It complicates out-of-order execution processor design.

Impact

Architectures depending on x86 branching will have to adapt.

RISC-V Features - Overall

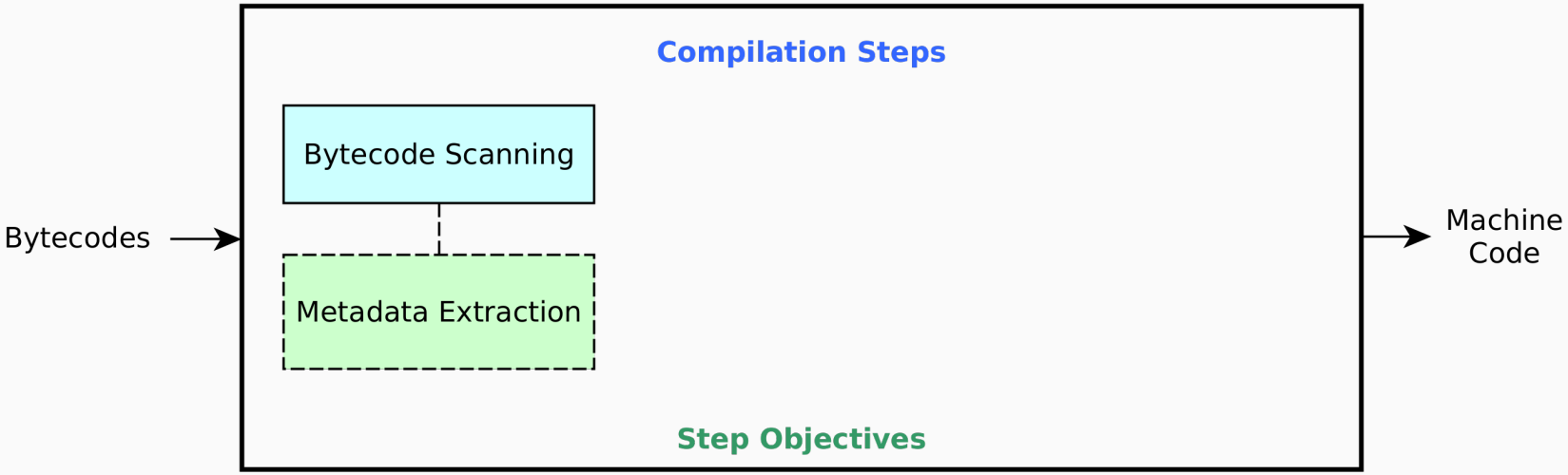
RISC-V presents the results of more than 25 years of RISC architecture development and refinement to emphasize design choices:

- **Simplicity** - common path is the default path
- **Performance** - no implicit state
- **Architecture/Implementation Isolation** - no delayed branch/load
- **Room for Growth** - generous available opcode space (and hints)

Cogit Internals

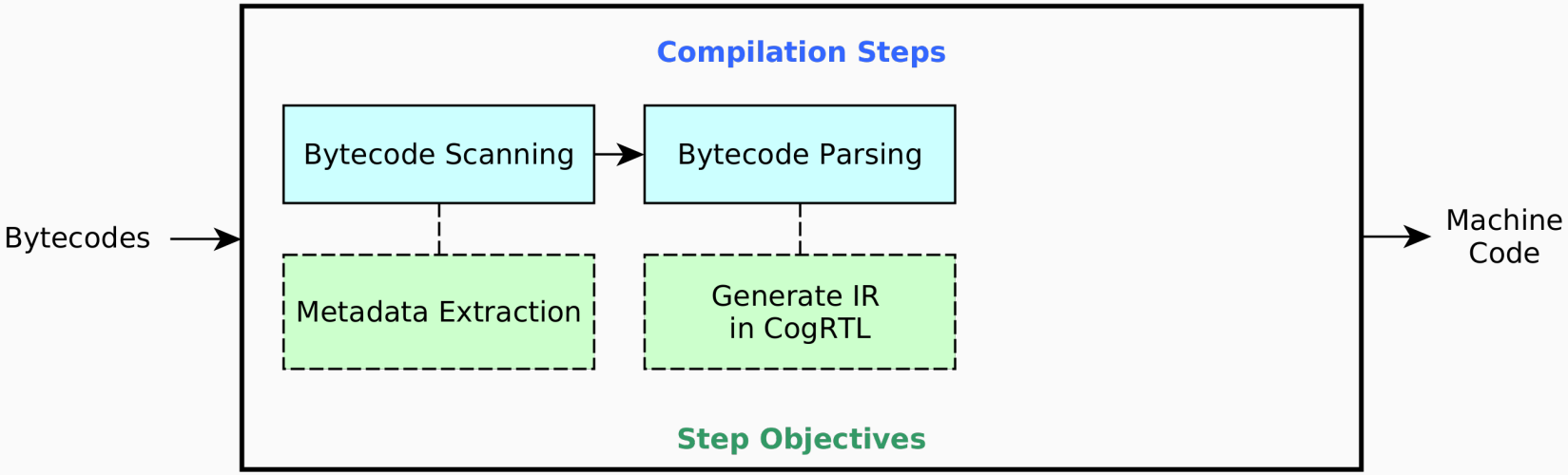
Cogit is Pharo's JIT Compiler is **linear**, **non-optimizing** and uses **registers fixed ahead-of-time**.

It processes **bytecodes** through a three-steps process:



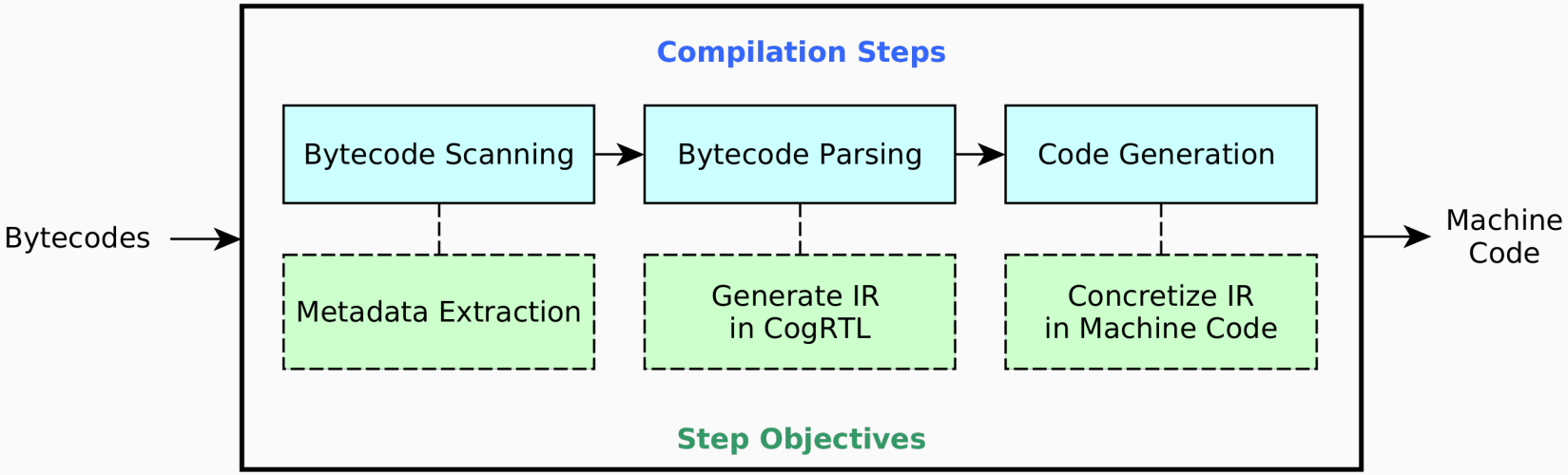
Cogit is Pharo's JIT Compiler is **linear**, **non-optimizing** and uses **registers fixed ahead-of-time**.

It processes **bytecodes** through a three-steps process:



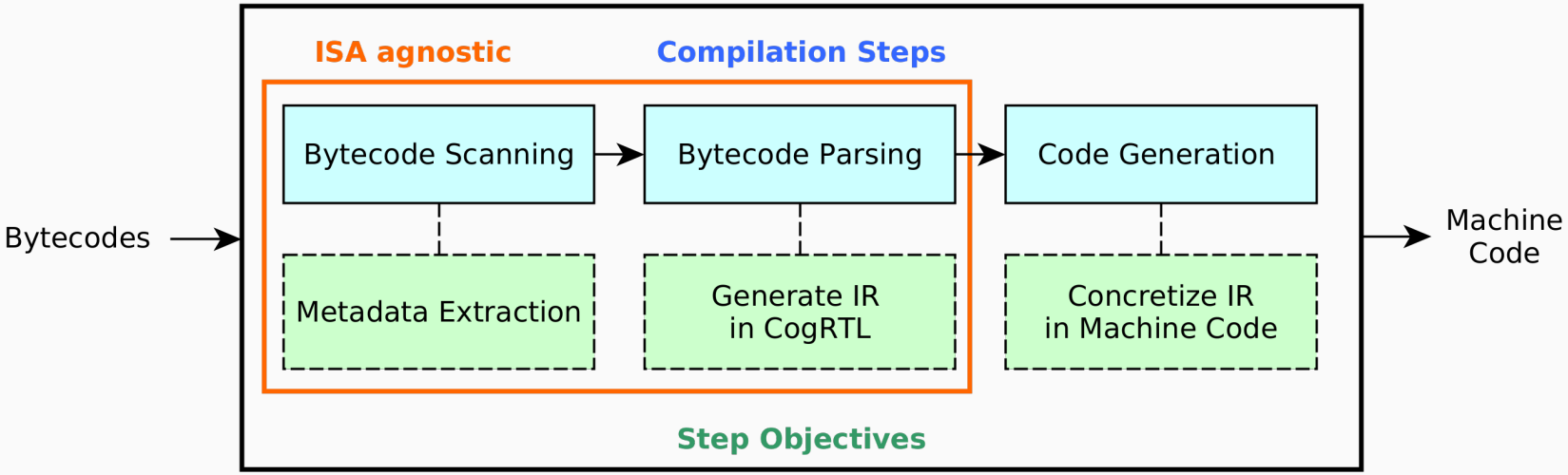
Cogit is Pharo's JIT Compiler is **linear**, **non-optimizing** and uses **registers fixed ahead-of-time**.

It processes **bytecodes** through a three-steps process:



Cogit is Pharo's JIT Compiler is **linear**, **non-optimizing** and uses **registers fixed ahead-of-time**.

It processes **bytecodes** through a three-steps process:



Cogit Internals - IR

Cogit Intermediate Representation called **CogRTL** is derived from x86:

```
Call, CallFull, CallR, ...
MoveRR, MoveMwrR, MoveX32rR, ...
JumpZero, JumpNonNegative, ...
PopR, PushR, ...
AndCqR, OrCqR, TstCqR, ...
AddRR, CmpRR, MulRR, ...
LogicalShiftRightRR, ArithmeticShiftLeftRR, ...
```

Cogit Internals - IR

Cogit Intermediate Representation called **CogRTL** is derived from x86:

```
Call, CallFull, CallR, ...
MoveRR, MoveMwrR, MoveX32rR, ...
JumpZero, JumpNonNegative, ...
PopR, PushR, ...
AndCqR, OrCqR, TstCqR, ...
AddRR, CmpRR, MulRR, ...
LogicalShiftRightRR, ArithmeticShiftLeftRR, ...
```

- Condition codes setter `Tst` or `Cmp`

Cogit Internals - IR

Cogit Intermediate Representation called **CogRTL** is derived from x86:

```
Call, CallFull, CallR, ...
MoveRR, MoveMwrR, MoveX32rR, ...
JumpZero, JumpNonNegative, ...
PopR, PushR, ...
AndCqR, OrCqR, TstCqR, ...
AddRR, CmpRR, MulRR, ...
LogicalShiftRightRR, ArithmeticShiftLeftRR, ...
```

- Condition codes setter `Tst` or `Cmp`
- Conditional Jumps

Cogit Internals - IR

Cogit Intermediate Representation called **CogRTL** is derived from x86:

```
Call, CallFull, CallR, ...
MoveRR, MoveMwrR, MoveX32rR, ...
JumpZero, JumpNonNegative, ...
PopR, PushR, ...
AndCqR, OrCqR, TstCqR, ...
AddRR, CmpRR, MulRR, ...
LogicalShiftRightRR, ArithmeticShiftLeftRR, ...
```

- Condition codes setter `Tst` or `Cmp`
- Conditional Jumps
- Different addressing modes

Rationale

Decisions on CogRTL design date from when x86 was the main architecture.

Applications to **ARMv7** or **ARMv8** remained feasible as both provided **x86-compatible capabilities** such as:

- Branching on flags
- Many addressing modes
- Bit manipulation operations

Rationale

Decisions on CogRTL design date from when x86 was the main architecture.

Applications to **ARMv7** or **ARMv8** remained feasible as both provided **x86-compatible capabilities** such as:

- Branching on flags
- Many addressing modes
- Bit manipulation operations

Unfortunately, it is a different story with RISC-V...

Clashes

Literals and Inline Caching

Cogit needs to patch generated machine codes whether for (1) garbage collection or (2) **inline caches** (*mono-, poly- and megamorphic*).

Literals and Inline Caching

Cogit needs to patch generated machine codes whether for (1) garbage collection or (2) **inline caches** (*mono-, poly- and megamorphic*).

Impact

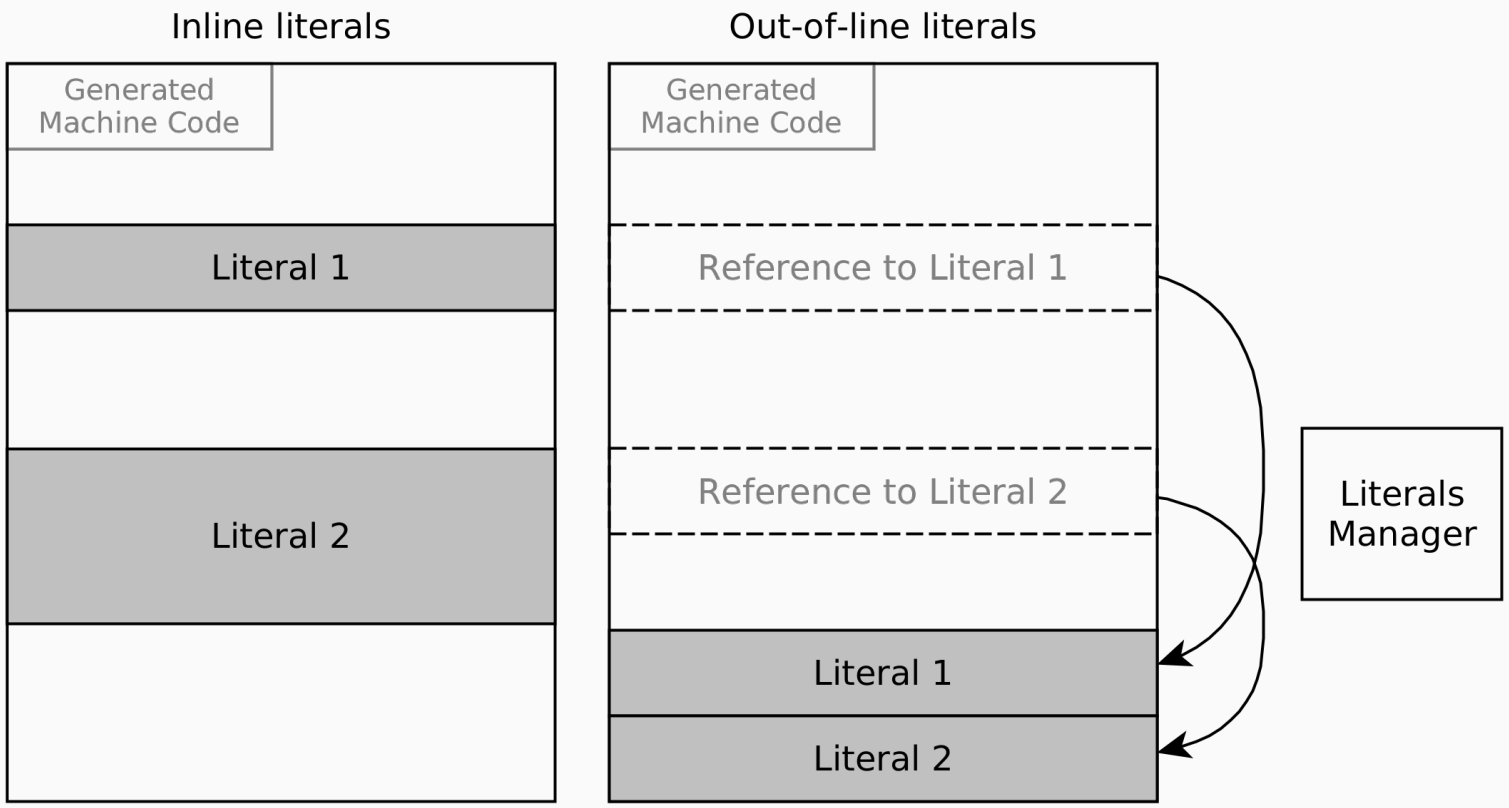
Patching literals requires to leave room for the biggest immediate value.

Literals and Inline Caching

Cogit needs to patch generated machine codes whether for (1) garbage collection or (2) **inline caches** (*mono-, poly- and megamorphic*).

Impact

Patching literals requires to leave room for the biggest immediate value.



Intermediate Representation Mismatch

The close link between CogRTL and x86/ARMv8 expects a *1-1* mapping:

```
# CogRTL instructions
cogit CmpR: ClassReg R: TempReg
cogit JumpNonZero: (Label 2)

# ARMv8 output
cmp r1, r22
b.ne 48

# RISC-V wanted output
bne r1, r22, 48
```

Intermediate Representation Mismatch

The close link between CogRTL and x86/ARMv8 expects a *1-1* mapping:

```
# CogRTL instructions
cogit CmpR: ClassReg R: TempReg
cogit JumpNonZero: (Label 2)

# ARMv8 output
cmp r1, r22
b.ne 48

# RISC-V wanted output
bne r1, r22, 48
```

Impact

Mismatch between **RISC-V and CogRTL** when mapping IR and machine code.

Reintroducing Condition Codes

One way to patch the issue is to **reintroduce condition codes**:

```
# CogRTL instruction
cogit CmpR: Arg0Reg R: ReceiverReg

# RISC-V output
sub  t3, s8, a3      seqz t5, t5
slti t1, a3, 1       sltu t6, s8, t3
slt  t2, t3, s8      slti t4, t3, 0
xor  t5, t1, t2      seqz t3, t3
```

Reintroducing Condition Codes

One way to patch the issue is to **reintroduce condition codes**:

```
# CogRTL instruction
cogit CmpR: Arg0Reg R: ReceiverReg

# RISC-V output
sub  t3, s8, a3      seqz t5, t5
slti t1, a3, 1       sltu t6, s8, t3
slt  t2, t3, s8      slti t4, t3, 0
xor  t5, t1, t2      seqz t3, t3
```

Impact

Reintroduction of a motivated ban from RISC-V. Increase of the number of instructions.

Peephole Optimization

Patching the IR to resolve the *1-1* mapping into a *2-1*:

```
newBranchOpcode := nextInstruction opcode caseOf: {
  [JumpZero] -> [BrEqualRR].
  [JumpNonZero] -> [BrNotEqualRR].
...}.
opcode caseOf: {
  ...
  [CmpRR] -> [newBranchLeft := operands at: 1.
    newBranchRight := operands at: 0.
    opcode := Label].
  [CmpCqR] -> [newBranchLeft := operands at: 1.
    newBranchRight := TempReg.
    opcode := MoveCqR.
    operands at: 1 put: TempReg].
...}.
```

`CmpRR op1 op2 / JumpZero op3` becomes `BrEqualRR op1 op2 op3`

Reworking the IR

Is this the time to rework the IR? We could get:

- Higher level **abstraction**
- Complex **optimizations**
- Data/control **flow analysis**

This could take the form of **V8's sea of nodes** or **LuaJIT SSA**!

Reworking the IR

Is this the time to rework the IR? We could get:

- Higher level **abstraction**
- Complex **optimizations**
- Data/control **flow analysis**

This could take the form of **V8's sea of nodes** or **LuaJIT SSA**!

Impact

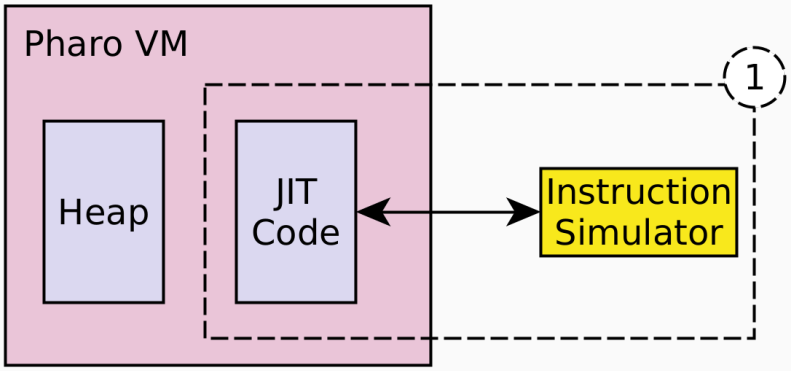
Rewriting the IR is a **consequent workload** but should be **beneficial long-term**!

Tooling and Port to RISC-V

Testing Ecosystem - Test Harness

Pharo VM development uses several simulation and testing levels:

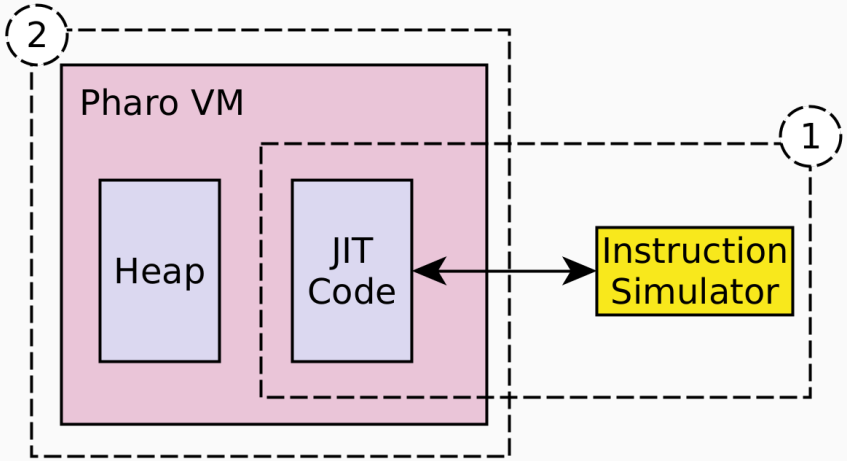
1. Unit-testing with an instruction simulator, **Unicorn**



Testing Ecosystem - Test Harness

Pharo VM development uses several simulation and testing levels:

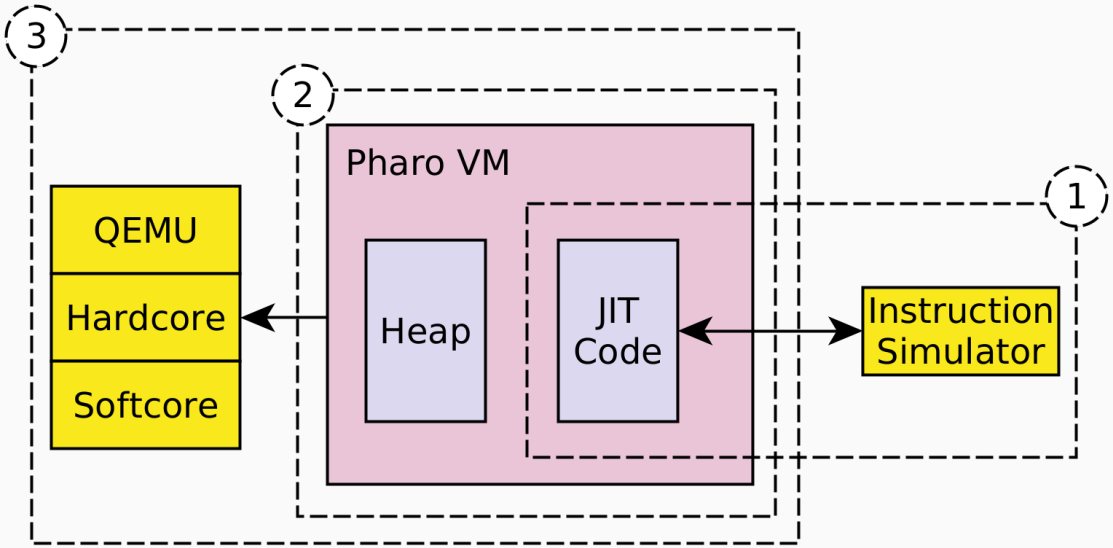
1. Unit-testing with an instruction simulator, **Unicorn**
2. Simulating the whole VM in Pharo, **CogVMSimulator**



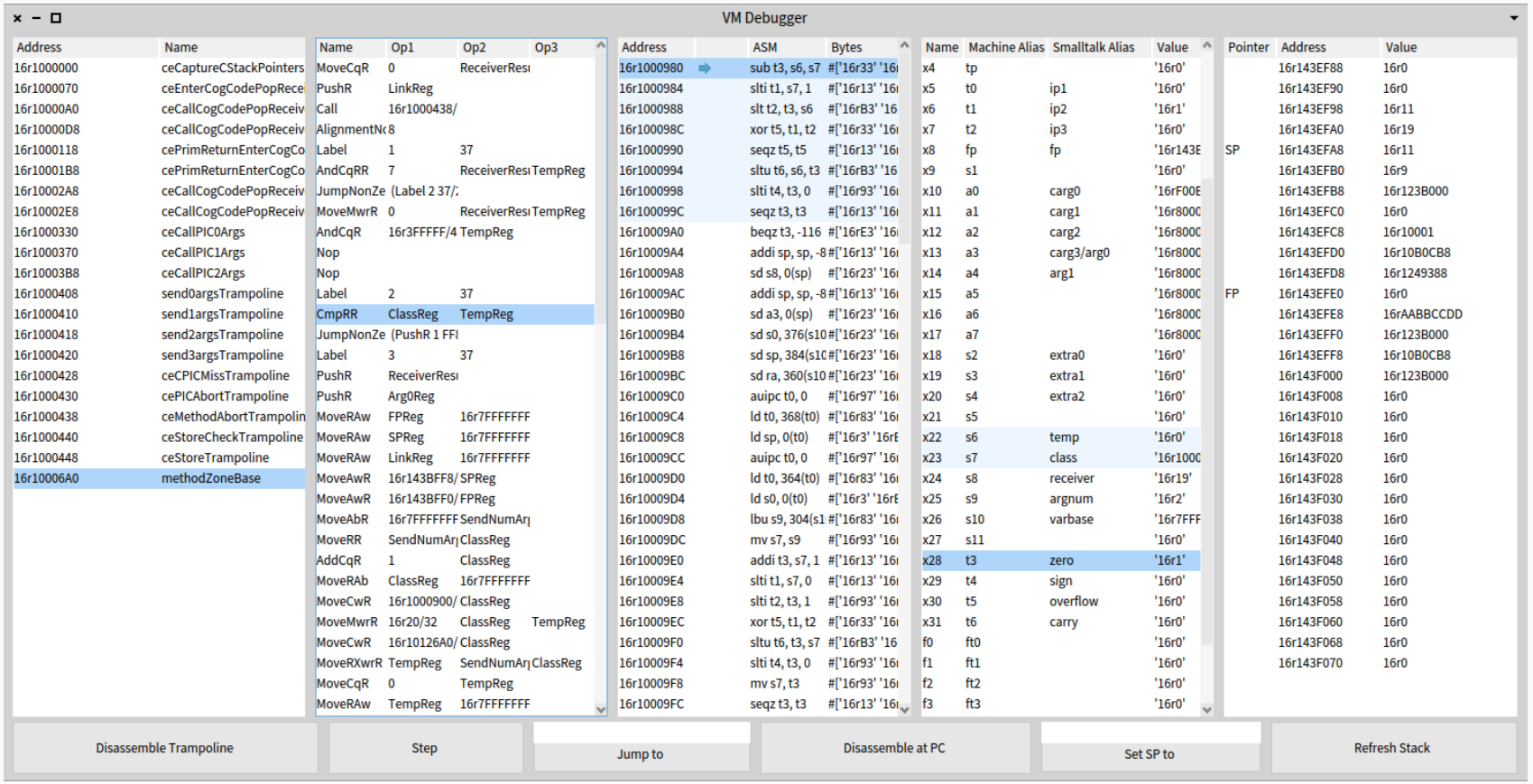
Testing Ecosystem - Test Harness

Pharo VM development uses several simulation and testing levels:

1. Unit-testing with an instruction simulator, **Unicorn**
2. Simulating the whole VM in Pharo, **CogVMSimulator**
3. Running on the architecture:
   - **Fedora Rawhide** in QEMU
   - **BeagleV** as the *hardcore* RISC-V SoC
   - **Rocket** as the *softcore* RISC-V CPU

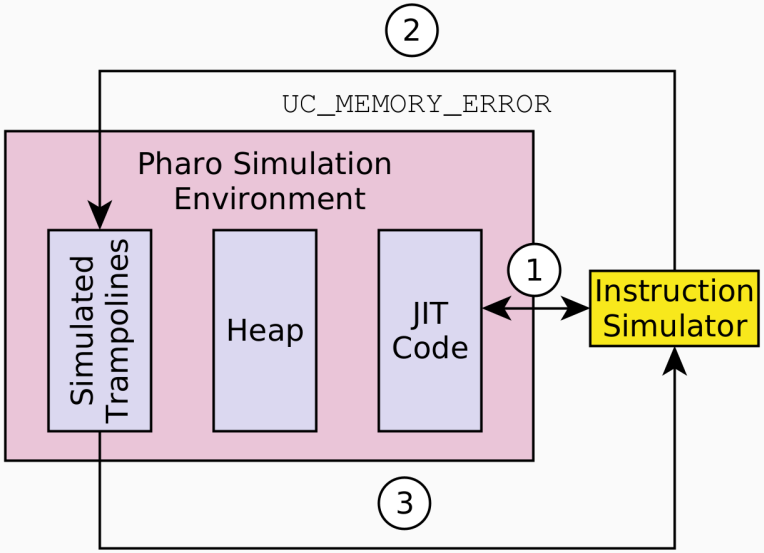


Tools - Machine Code Debugger



Tools - Instruction Simulator

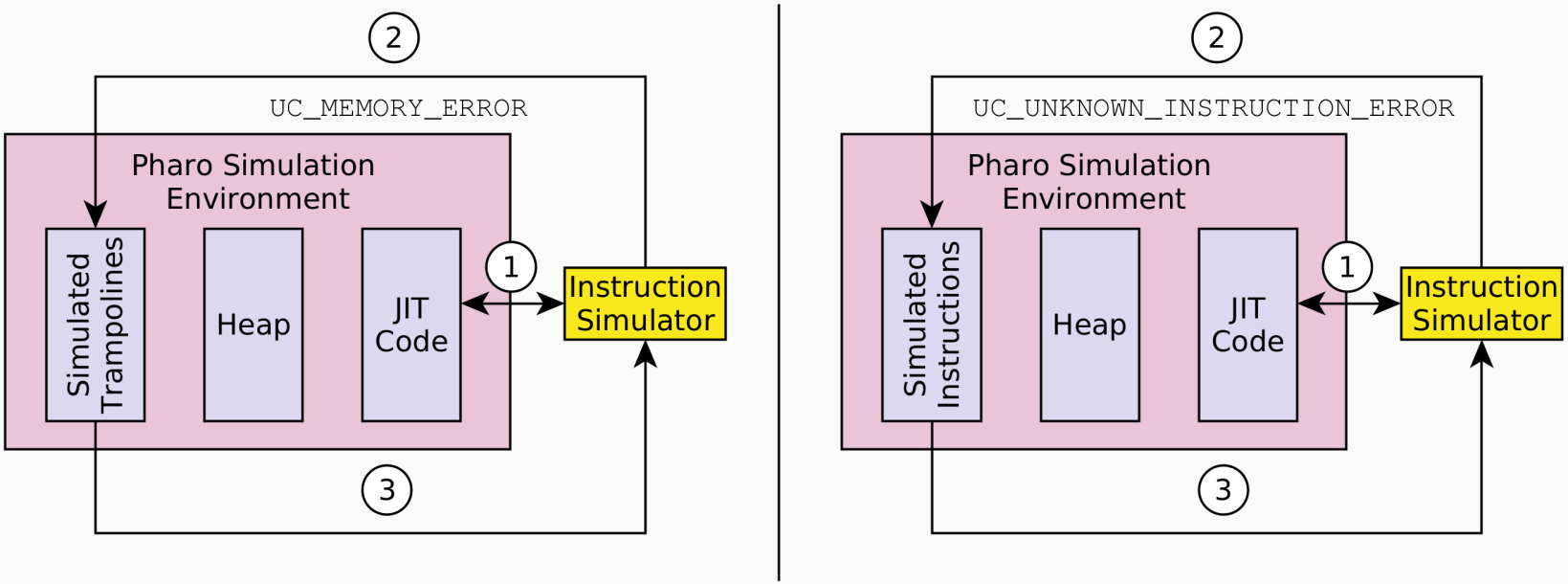
The **rich simulation environment** coupled with the **various hooks** Unicorn provide makes it very flexible:



Simulated Trampolines —

Tools - Instruction Simulator

The **rich simulation environment** coupled with the **various hooks** Unicorn provide makes it very flexible:



Simulated Trampolines — Simulated Instructions

Port to RISC-V

As for now:

- Cogit is **compliant with unit tests (1)**!
- Rocket has been extended to **support custom instructions**!
- We still need to work our way through **simulation (2)** and **hardware execution (3)**

However, regarding the *toolchain*:

- Every item remains in *early/stable-ish* development
- ISA being open-source also means *various implementations*
- Having access to *reliable hardware* is not easy

Conclusion and Future Works

Conclusion and Future Works

Takeaways:

- RISC-V is an **open-source, modular ISA** with bold design decisions
- Are **clashes with CogRTL** significant enough to suggest a **rewriting**?
- **Testing and tooling** help dealing with issues at **high level**

Conclusion and Future Works

Takeaways:

- RISC-V is an **open-source, modular ISA** with bold design decisions
- Are **clashes with CogRTL** significant enough to suggest a **rewriting**?
- **Testing and tooling** help dealing with issues at **high level**

Future works:

- *What dedicated instruction would the VM benefit from?*
- *How to secure the VM using RISC-V?*
- *How to use a dedicated co-processor along the VM?*

Conclusion and Future Works

Takeaways:

- RISC-V is an **open-source, modular ISA** with bold design decisions
- Are **clashes with CogRTL** significant enough to suggest a **rewriting**?
- **Testing and tooling** help dealing with issues at **high level**

Future works:

- *What dedicated instruction would the VM benefit from?*
- *How to secure the VM using RISC-V?*
- *How to use a dedicated co-processor along the VM?*

Thank you!

Contact: **quentin.ducasse@ensta-bretagne.org**

References i

- Boom organization.
  https://github.com/riscv-boom.
- Cv32 chip.
  https://github.com/openhwgroup/cv32e40p.
- Cva6 chip.
  https://github.com/openhwgroup/cva6.
- Ibex chip.
  https://github.com/lowRISC/ibex.
- The RISC-V instruction set manual - volume I: Unprivileged ISA. Document version 20191213.
  https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

References ii

Rocket chip generator.
https://github.com/chipsalliance/rocket-chip.

D. Patterson and A. Waterman.
The RISC-V Reader: An Open Architecture Atlas.
Technical report, 2015.