

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

Quentin Ducasse¹ Guillermo Polito² Pablo Tesone²
Pascal Cotret¹ Loïc Lagadec¹

September 15, 2022

(1) ENSTA Bretagne - LabSTICC

(2) INRIA Lille - RMoD

1. Background
2. RISC-V Implementation Details
3. Cogit Internals
4. Clashes
5. Tooling and Port to RISC-V
6. Conclusion and Future Works



September 06, 2022 |

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

[Learn More](#)



September 06, 2022 |

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

[Learn More](#)

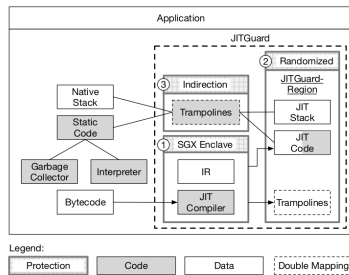
Our main objective with **RISC-V** is to send ~~Pharo~~ to the moon:

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

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

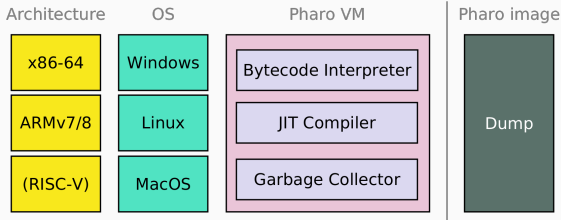
The Pharo language is:

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

```
exampleWithNumber: x
  <aMethodAnnotation>
  | y |
  true & false not & (nil isNil)
  iffFalse: [ 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
```

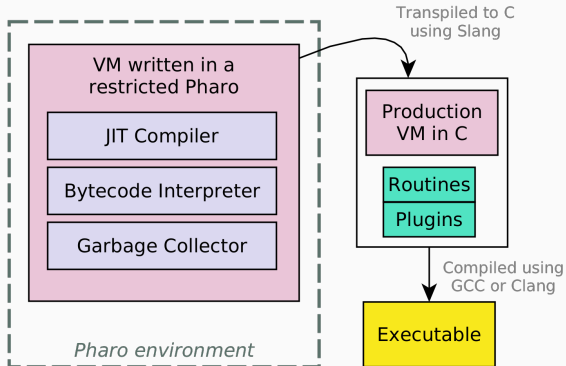
The runtime environment is the Pharo VM, it is composed of:

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



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 was born in Berkeley around **2010**.

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

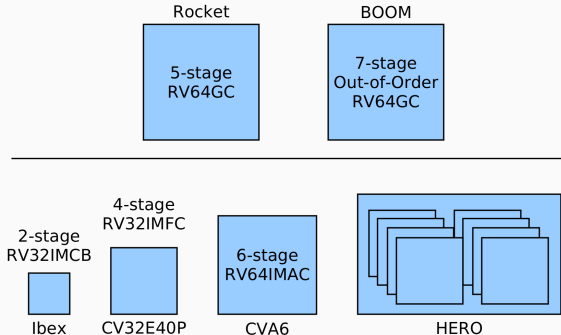
The ISA is:

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

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



Repositories in references!

RISC-V Implementation Details

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 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!
```


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**.

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.

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

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.**

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

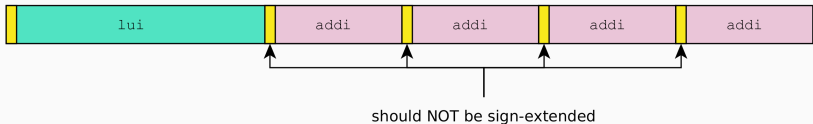
Rationale

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

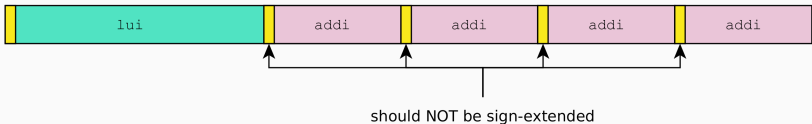
Impact

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

```
li t0, 0x3800800800800800
    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
```



```
li t0, 0x3800800800800800
    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

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`.

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.

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.

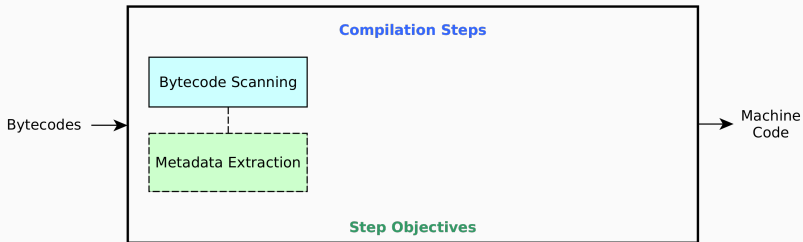
RISC-V presents the results of more than 25 years of RISC architecture development and refinement to emphasize design choices:

- **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

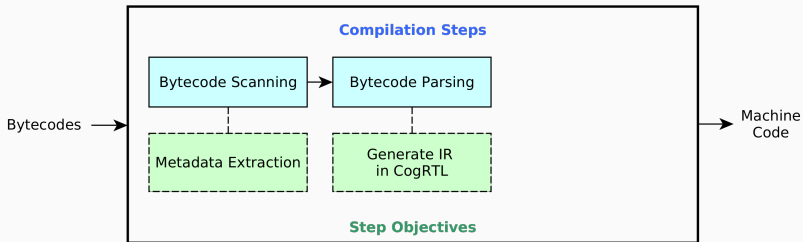
Cogit is Pharo's JIT Compiler is **linear**, **non-optimizing** and uses **registers fixed ahead-of-time**.

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



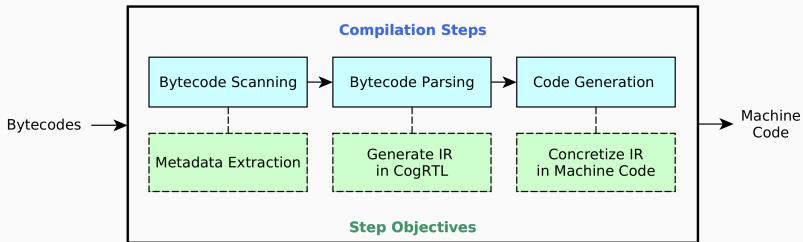
Cogit is Pharo's JIT Compiler is **linear**, **non-optimizing** and uses **registers fixed ahead-of-time**.

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



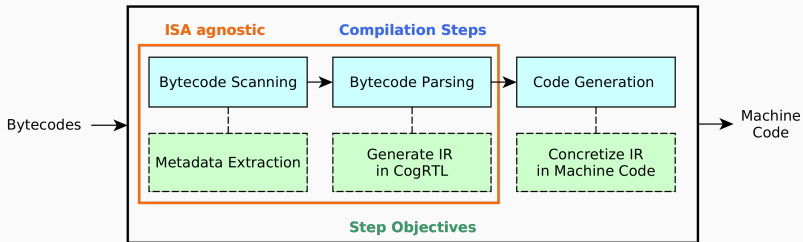
Cogit is Pharo's JIT Compiler is **linear**, **non-optimizing** and uses **registers fixed ahead-of-time**.

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 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 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 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 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

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

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

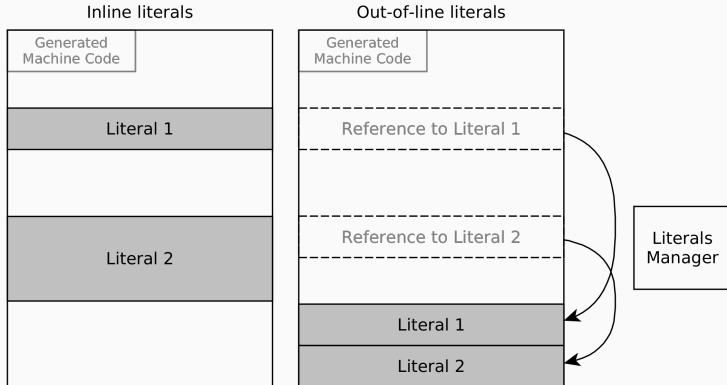
Impact

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

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.



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
```

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.

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
```

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.

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

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!**

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!**

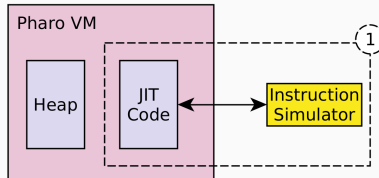
Impact

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

Tooling and Port to RISC-V

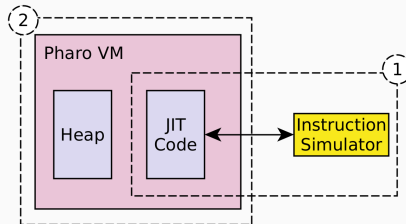
Pharo VM development uses several simulation and testing levels:

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



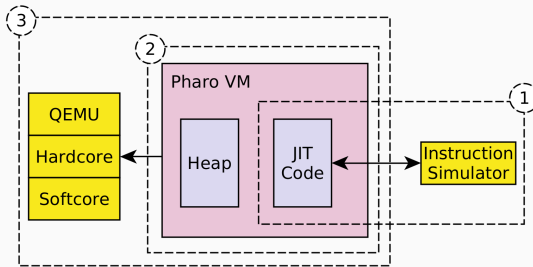
Pharo VM development uses several simulation and testing levels:

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



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

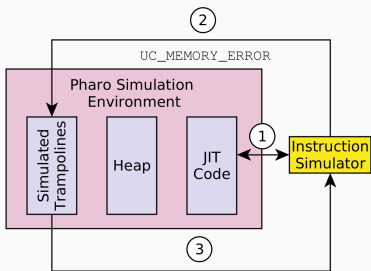


VM Debugger

Address	Name	Name	Op1	Op2	Op3	Address	ASM	Bytes	Name	Machine Alias	Smalltalk Alias	Value	Pointer	Address	Value
16r100000	ccCaptureCStackPointers	MoveCqR	0		ReceiverRes	16r1000980	sub t3, s6, s7	#"16r33" 16	x4	tp		"16r0"		16r143EF88	16r0
16r1000070	ceEnterCqCodePopReceiv	PushR	LinkReg			16r1000984	sbt t1, s7, 1	#"16r13" 16	x5	t0	ip1	"16r0"		16r143EF90	16r0
16r10000A0	ceCallCqCodePopReceiv	Call		16r1000438/		16r1000988	sbt t2, t3, s6	#"16rB3" 16	x6	t1	ip2	"16r1"		16r143EF9A	16r11
16r10000D8	ceCallCqCodePopReceiv	AlignmentN8				16r100099C	xor t5, t1, t2	#"16r33" 16	x7	t2	ip3	"16r0"		16r143EF9A	16r19
16r1000118	cePrimReturnEnterCqCo	Label	1	37		16r1000990	seqz t5, t5	#"16r13" 16	x8	fp	fp	"16r143E"	SP	16r143EFA8	16r11
16r10001B8	cePrimReturnEnterCqCo	AndCqRR	7		ReceiverRes TempReg	16r1000994	shu t6, s6, t3	#"16rB3" 16	x9	s1		"16r0"		16r143EF80	16r9
16r10002E8	ceCallCqCodePopReceiv	JumpNonZe			Label 2 37/;	16r1000998	sbt t4, t3, 0	#"16r93" 16	x10	a0	carg0	"16rFO0E"		16r143EF88	16r1238000
16r1000328	ceCallCqCodePopReceiv	MoveMwR	0		ReceiverRes TempReg	16r100099C	seqz t3, t3	#"16r13" 16	x11	a1	carg1	"16r800C"		16r143EFC0	16r0
16r1000330	ceCallPIC0Args	AndCqR			16r3FFFFF/A TempReg	16r10009A0	beqz t3, -116	#"16rE3" 16	x12	a2	carg2	"16r800C"		16r143EFC3	16r10001
16r1000370	ceCallPIC1Args	Nop				16r10009A4	addi sp, sp, -8	#"16r13" 16	x13	a3	carg3/arg0	"16r800C"		16r143EF00	16r1080CB8
16r1000388	ceCallPIC2Args	Nop				16r10009A8	sd s8, 0(sp)	#"16r23" 16	x14	a4	arg1	"16r800C"		16r143EF08	16r1249388
16r1000408	sendJargsTrampoline	Label	2	37		16r10009AC	addi sp, sp, -8	#"16r13" 16	x15	a5		"16r800C"	FP	16r143EF00	16r0
16r1000410	sendJargsTrampoline	CmpRR	ClassReg	TempReg		16r10009B0	sd a3, 0(sp)	#"16r23" 16	x16	a6		"16r800C"		16r143EF8	16r4A8BCDD
16r1000418	sendJargsTrampoline	JumpNonZe			(PushR 1 FF)	16r10009B4	sd s9, 376(s10)	#"16r23" 16	x17	a7		"16r800C"		16r143EF70	16r1238000
16r1000420	sendJargsTrampoline	Label	3	37		16r10009B8	sd sp, 384(s10C)	#"16r23" 16	x18	t2	extra0	"16r0"		16r143EF78	16r1080CB6
16r1000428	ceCPICMissTrampoline	PushR	ReceiverRes			16r10009BC	sd ra, 360(s10)	#"16r23" 16	x19	t3	extra1	"16r0"		16r143F000	16r1238000
16r1000430	ceCPICAbortTrampoline	PushR	ArgReg			16r10009C0	auspct t0, 0	#"16r97" 16	x20	s4	extra2	"16r0"		16r143F008	16r0
16r1000438	ceMethodAbortTrampoline	MoveRAw	FPReg		16r7FFFFFFF	16r10009C4	ld t0, 368(t0)	#"16rB3" 16	x21	s5		"16r0"		16r143F010	16r0
16r1000440	ceStoreCheckTrampoline	MoveRAw	SPReg		16r7FFFFFFF	16r10009C8	ld sp, 0(t0)	#"16r3" 16r1	x22	s6	temp	"16r0"		16r143F018	16r0
16r1000448	ceStoreTrampoline	MoveRAw	LinkReg		16r7FFFFFFF	16r10009CC	auspct t0, 0	#"16r97" 16	x23	s7	class	"16r100C"		16r143F020	16r0
16r10006A0	methodZoneBase	MoveAwR	16r143BFF/	FPReg		16r10009D0	ld t0, 364(t0)	#"16rB3" 16	x24	s8	receiver	"16r13"		16r143F028	16r0
		MoveAwR	16r143BFF/	FPReg		16r10009D4	ld s0, 0(t0)	#"16r3" 16r1	x25	s9	argnum	"16r27"		16r143F030	16r0
		MoveABR	16r7FFFFFFF	SndNumArJ		16r10009D8	lbu s9, 304(s1)	#"16rB3" 16	x26	s10	varbase	"16r7FFF"		16r143F038	16r0
		MoveRR	SndNumArJ	ClassReg		16r10009DC	mv s7, s9	#"16r93" 16	x27	s11		"16r0"		16r143F040	16r0
		AddCqR	1	ClassReg		16r10009E0	addi t3, s7, 1	#"16r13" 16	x28	t3	zero	"16r1"		16r143F048	16r0
		MoveRAb	ClassReg		16r7FFFFFFF	16r10009E4	sbt t1, s7, 0	#"16r13" 16	x29	t4	sign	"16r0"		16r143F050	16r0
		MoveCwR	16r1000900/	ClassReg		16r10009E8	sbt t2, t3, 1	#"16r93" 16	x30	t5	overflow	"16r0"		16r143F058	16r0
		MoveMwR	16r20/32	ClassReg	TempReg	16r10009EC	xor t5, t1, t2	#"16r33" 16	x31	t6	carry	"16r0"		16r143F060	16r0
		MoveCwR	16r10126A0/	ClassReg		16r10009F0	shu t6, t3, s7	#"16rB3" 16	t0	ft0		"16r0"		16r143F068	16r0
		MoveRXwR	TempReg	SndNumArJ	ClassReg	16r10009F4	sbt t4, t3, 0	#"16r93" 16	t1	ft1		"16r0"		16r143F070	16r0
		MoveCqR	0	TempReg		16r10009F8	mv s7, t3	#"16r93" 16	t2	ft2		"16r0"			
		MoveRAw	TempReg		16r7FFFFFFF	16r10009FC	seqz t3, t3	#"16r13" 16	t3	ft3		"16r0"			

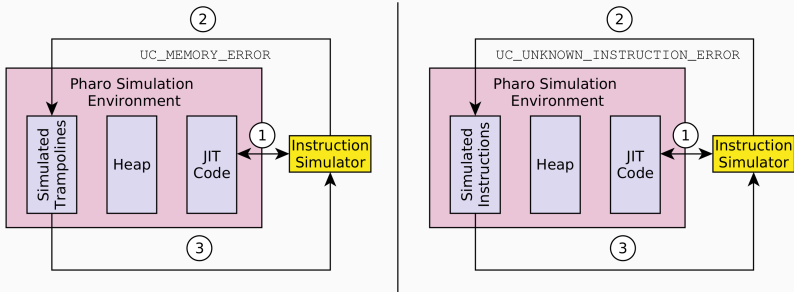
Disassemble Trampoline
Step
Jump to
Disassemble at PC
Set SP to
Refresh Stack

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



Simulated Trampolines —

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



Simulated Trampolines — Simulated Instructions

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

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**

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?*

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

-  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>.



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.