ArchSem: Formalising concurrent CPU architectures to support verification in Rocq

Thibaut Pérami

University of Cambridge

8 June 2026

https://github.com/rems-project/archsem

Context: My trajectory

Started by building a toy OS in 3rd year: Fun but full of bugs

Context: My trajectory

Started by building a toy OS in 3rd year: Fun but full of bugs

Wanted to verify: interned with SeL4 people

Context: My trajectory

Started by building a toy OS in 3rd year: Fun but full of bugs

Wanted to verify: interned with SeL4 people

Realisation: Mostly about OS-user-space interactions, but not OS-hardware interaction

Context: My trajectory

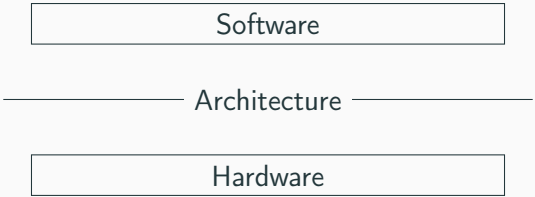
Started by building a toy OS in 3rd year: Fun but full of bugs

Wanted to verify: interned with SeL4 people

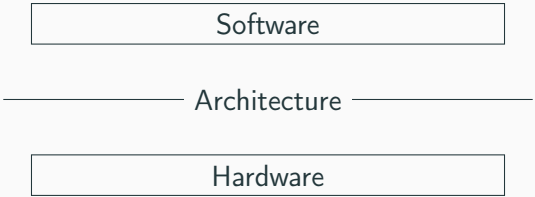
Realisation: Mostly about OS-user-space interactions, but not OS-hardware interaction

Need good architecture models

What is an architectural model



What is an architectural model



More concretely:

Defines an envelope of allowed behaviours:

- Set of final states from an initial state
- Allowed executions traces/witness

Why we need one

Better architecture specifications

Why we need one

Better architecture specifications

Architecture proofs

- Verification Morello (CHERI-Arm)
- Morello-Cerise (PLDI25)
- Virtual memory (VM) refinement

Why we need one

Better architecture specifications

Architecture proofs

- Verification Morello (CHERI-Arm)
- Morello-Cerise (PLDI25)
- Virtual memory (VM) refinement

System software verification

- SeKVM
- NOVA (BlueRock, WIP)
- AWS Nitro
- WIP: pKVM (in Android since 14)

Why we need one

Better architecture specifications

arm intel RISC-V

Architecture proofs

- Verification Morello (CHERI-Arm)
- Morello-Cerise (PLDI25)
- Virtual memory (VM) refinement

Hardware verification

- Verification of an CHERIoT-Ibex Processor (JasperGold, 2025)
- Intel's microcode verification (ACL2)
- ...

System software verification

- SeKVM
- NOVA (BlueRock, WIP)
- AWS Nitro
- WIP: pKVM (in Android since 14)

Why we need one

Better architecture specifications

arm intel RISC-V

Architecture proofs

- Verification Morello (CHERI-Arm)
- Morello-Cerise (PLDI25)
- Virtual memory (VM) refinement

Hardware verification

- Verification of an CHERIoT-Ibex Processor (JasperGold, 2025)
- Intel's microcode verification (ACL2)
- ...

System software verification

seL4 CertiKOS

- SeKVM
- NOVA (BlueRock, WIP)
- AWS Nitro
- WIP: pKVM (in Android since 14)

Compiler verification

- CompCert
- CakeML
- ...

How: Decomposing the architecture

ISA model

Architecture pseudo-code: ASL, Sail, ...

How: Decomposing the architecture

ISA model

Architecture pseudo-code: ASL, Sail, ...

Memory model

ISA model $\rightarrow$ Machine model

How: Decomposing the architecture

ISA model

Architecture pseudo-code: ASL, Sail, ...

Memory model

ISA model $\rightarrow$ Machine model

User

For compiler/application verification:

- Plain arithmetic
- Plain memory
- Optionally mixed-size accesses
- Register dependencies tracking

How: Decomposing the architecture

ISA model

Architecture pseudo-code: ASL, Sail, ...

~~**Memory model**~~ **Concurrency model**

ISA model $\rightarrow$ Machine model

User

For compiler/application verification:

- Plain arithmetic
- Plain memory
- Optionally mixed-size accesses
- Register dependencies tracking

System

For system/runtime verification:

- Relaxed configuration registers
- Virtual memory, MMU, TLB
- Instruction fetching (icaches)
- Exception and Interrupts

ISA models

`0b100011111000011` $\Longrightarrow$ `ADD R1, R2, R3` $\Longrightarrow$ `R1 = R2 + R3`

ISA models (Alasdair Armstrong, Brian Campbell, ...)

`0b100011111000011` ⟹ `ADD R1, R2, R3` ⟹ `R1 = R2 + R3`

arm

ASL
(316 kLoC)

⇓


Sail
(433 kLoC)

RISC-V®


Sail
(22 kLoC)

intel

ACL2
(Not official)

⇓


Sail
(30 kLoC)

Concurrency models

MP+dmb.sy+ctrl AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;	
Thread 0	Thread 1

STR X0,[X1]//a
DMB SY//b
STR X0,[X2]//c
	LDR X0,[X1]//d
	CBNZ X0,LC00
	LC00:
	LDR X2,[X3]//e

Allowed: 1:X0=1; 1:X2=0;	

(a) Litmus test source

Concurrency models

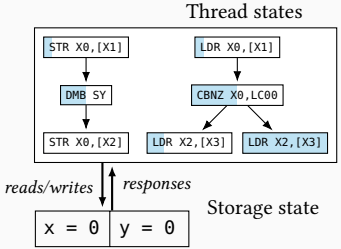
MP+dmb.sy+ctrl AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;	
Thread 0	Thread 1

STR X0,[X1]//a
DMB SY//b
STR X0,[X2]//c
	LDR X0,[X1]//d
	CBNZ X0,LC00
	LC00:
	LDR X2,[X3]//e

Allowed: 1:X0=1; 1:X2=0;	

(a) Litmus test source



(b) “Flat” operational model state

Concurrency models

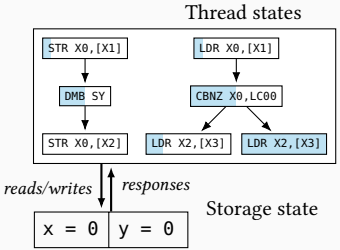
MP+dmb.sy+ctrl AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;	
Thread 0	Thread 1

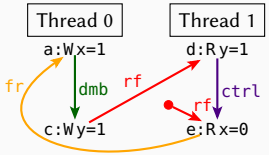
STR X0,[X1]//a
DMB SY//b
STR X0,[X2]//c
	LDR X0,[X1]//d
	CBNZ X0,LC00
	LC00:
	LDR X2,[X3]//e

Allowed: 1:X0=1; 1:X2=0;	

(a) Litmus test source



(b) “Flat” operational model state



(c) Axiomatic candidate

Concurrency models

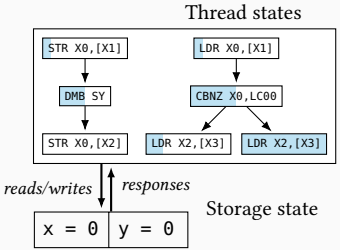
MP+dmb.sy+ctrl AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;	
Thread 0	Thread 1

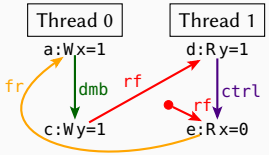
STR X0,[X1]//a
DMB SY//b
STR X0,[X2]//c
	LDR X0,[X1]//d
	CBNZ X0,LC00
	LC00:
	LDR X2,[X3]//e

Allowed: 1:X0=1; 1:X2=0;	

(a) Litmus test source



(b) "Flat" operational model state



(c) Axiomatic candidate

Increasingly complex: Need full formalisation

Previous work

We wanted:

- Architecture models containing full ISA model
- In an ITP, not just a test oracle (e.g. Herd or Isla)
- But still able to execute to compare with state of the art

Previous work

We wanted:

- Architecture models containing full ISA model
- In an ITP, not just a test oracle (e.g. Herd or Isla)
- But still able to execute to compare with state of the art

	Toy or simplified ISA	Real-ISA
Test oracle	Herd(current)	Isla, RMEM, Herd(WIP)
ITP	Promising (POPL 19), IMM's Arm, ...	ArchSem

ArchSem

A reusable formal framework to formally define CPU architecture models

Requirements

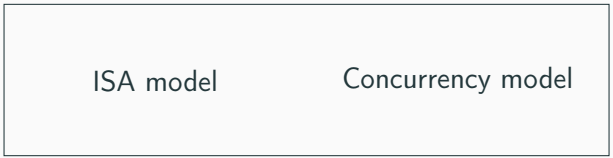
- Architecture generic
- Concurrency model generic (Axiomatic, Promising, micro-architectural, ...)
- ISA model generic
- Executable
- Usable for proof

ArchSem

A reusable formal framework to formally define CPU architecture models

Requirements

- Architecture generic
- Concurrency model generic (Axiomatic, Promising, micro-architectural, ...)
- ISA model generic
- Executable
- Usable for proof



ArchSem

A reusable formal framework to formally define CPU architecture models

Requirements

- Architecture generic
- Concurrency model generic (Axiomatic, Promising, micro-architectural, ...)
- ISA model generic
- Executable
- Usable for proof

Generic Interface

ArchSem

A reusable formal framework to formally define CPU architecture models

Requirements

- Architecture generic
- Concurrency model generic (Axiomatic, Promising, micro-architectural, ...)
- ISA model generic
- Executable
- Usable for proof

Generic Interface

Arch Instantiation

Organisation

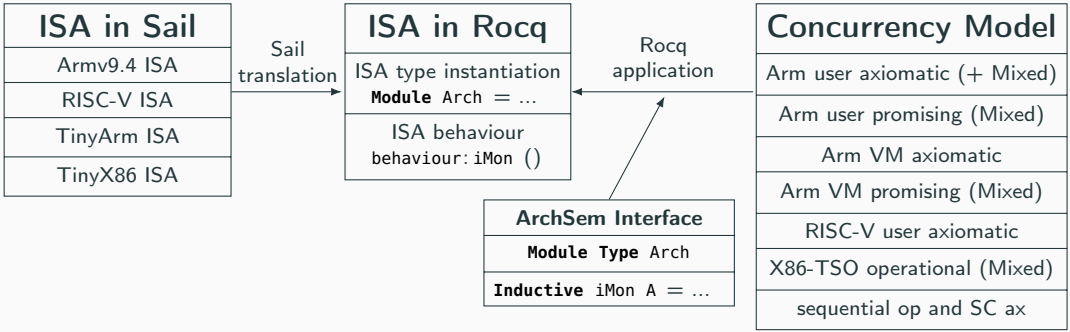
ISA in Sail
Armv9.4 ISA
RISC-V ISA
TinyArm ISA
TinyX86 ISA

Organisation

ISA in Sail
Armv9.4 ISA
RISC-V ISA
TinyArm ISA
TinyX86 ISA

Concurrency Model
Arm user axiomatic (+ Mixed)
Arm user promising (Mixed)
Arm VM axiomatic
Arm VM promising (Mixed)
RISC-V user axiomatic
X86-TSO operational (Mixed)
sequential op and SC ax

Organisation



```
// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
  : bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
```

```
// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
  : bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
```

```
// 2. TinyArm type instantiation
register X3 : bits(64)
union barrier = { Barrier_DMB, ...
instantiation 'barrier = barrier
```

```
// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
  : bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
```

```
// 2. TinyArm type instantiation
register X3 : bits(64)
union barrier = { Barrier_DMB, ...
instantiation 'barrier = barrier
```

```
// 3. TinyArm behaviour: LDR Xt,[Xn,Xm]
function clause execute LDR(t,n,m)={
  let base_addr = X(n);
  let offset = X(m);
  let addr = base_addr+offset;
  let data = MemRead(addr);
  X(t) = data;
```

```
// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
  : bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
```

fixed corresp.

```
(* 4. ArchSem Interface, in Rocq *)
Inductive outcome :=
| RegRead (r: reg)(* eff_ret ... = bv 64*)
| RegWrite (r: reg) (val : bv 64) (* () *)
| MemRead (addr: bv 64) (* bv 64 *)
| MemWrite (addr: bv 64) (val : bv 64)(*()*)
| Barrier (b: barrier) (* () *)
Inductive iMon A :=
| IRet (ret: A)
| INext (o: outcome) (k: eff_ret o → iMon A)
```

```
// 2. TinyArm type instantiation
register X3 : bits(64)
union barrier = { Barrier_DMB, ...
instantiation 'barrier = barrier
```

```
// 3. TinyArm behaviour: LDR Xt,[Xn,Xm]
function clause execute LDR(t,n,m)={
  let base_addr = X(n);
  let offset = X(m);
  let addr = base_addr+offset;
  let data = MemRead(addr);
  X(t) = data;
```

```
// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
  : bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
```

fixed corresp.

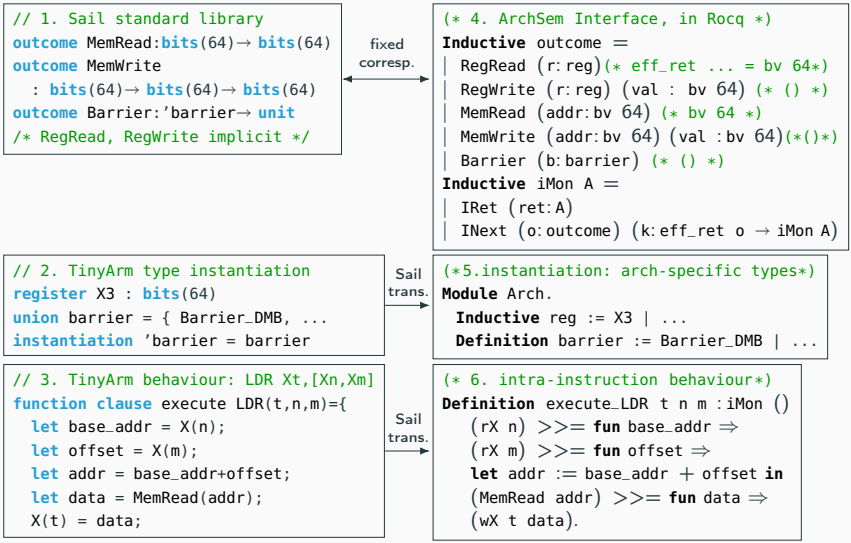
```
(* 4. ArchSem Interface, in Rocq *)
Inductive outcome :=
| RegRead (r: reg)(* eff_ret ... = bv 64*)
| RegWrite (r: reg) (val : bv 64) (* () *)
| MemRead (addr: bv 64) (* bv 64 *)
| MemWrite (addr: bv 64) (val : bv 64)(*()*)
| Barrier (b: barrier) (* () *)
Inductive iMon A :=
| IRet (ret: A)
| INext (o: outcome) (k: eff_ret o → iMon A)
```

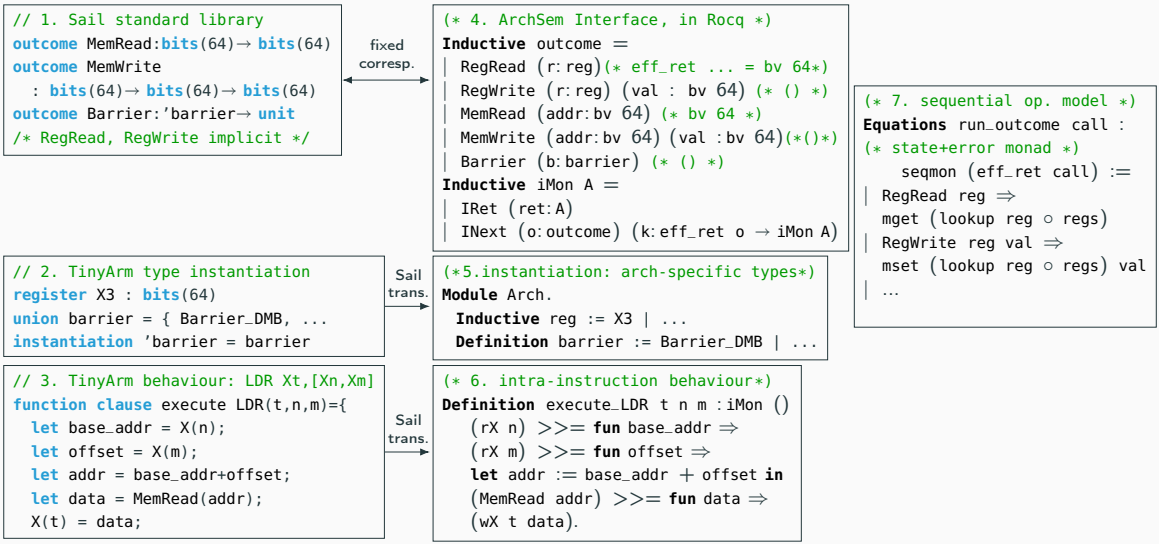
```
// 2. TinyArm type instantiation
register X3 : bits(64)
union barrier = { Barrier_DMB, ...
instantiation 'barrier = barrier
```

Sail trans.

```
(*5.instantiation: arch-specific types*)
Module Arch.
  Inductive reg := X3 | ...
  Definition barrier := Barrier_DMB | ...
```

```
// 3. TinyArm behaviour: LDR Xt,[Xn,Xm]
function clause execute LDR(t,n,m)={
  let base_addr = X(n);
  let offset = X(m);
  let addr = base_addr+offset;
  let data = MemRead(addr);
  X(t) = data;
```


// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
: bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
fixed corresp.
(* 4. ArchSem Interface, in Rocq *)
Inductive outcome =
| RegRead (r: reg)(* eff_ret ... = bv 64*)
| RegWrite (r: reg) (val : bv 64) (* () *)
| MemRead (addr: bv 64) (* bv 64 *)
| MemWrite (addr: bv 64) (val : bv 64)(*()*)
| Barrier (b: barrier) (* () *)
Inductive iMon A =
| IRet (ret: A)
| INext (o: outcome) (k: eff_ret o → iMon A)
// 2. TinyArm type instantiation
register X3 : bits(64)
union barrier = { Barrier_DMB, ...
instantiation 'barrier = barrier
Sail trans.
(*5.instantiation: arch-specific types*)
Module Arch.
Inductive reg := X3 | ...
Definition barrier := Barrier_DMB | ...
// 3. TinyArm behaviour: LDR Xt,[Xn,Xm]
function clause execute LDR(t,n,m)={
let base_addr = X(n);
let offset = X(m);
let addr = base_addr+offset;
let data = MemRead(addr);
X(t) = data;
Sail trans.
(* 6. intra-instruction behaviour*)
Definition execute_LDR t n m : iMon ()
(rX n) >>= fun base_addr ⇒
(rX m) >>= fun offset ⇒
let addr := base_addr + offset in
(MemRead addr) >>= fun data ⇒
(wX t data).


// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
: bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
fixed corresp.
(* 4. ArchSem Interface, in Rocq *)
Inductive outcome =
| RegRead (r: reg)(* eff_ret ... = bv 64*)
| RegWrite (r: reg) (val : bv 64) (* () *)
| MemRead (addr: bv 64) (* bv 64 *)
| MemWrite (addr: bv 64) (val : bv 64)(*()*)
| Barrier (b: barrier) (* () *)
Inductive iMon A =
| IRet (ret: A)
| INext (o: outcome) (k: eff_ret o → iMon A)
(* 7. sequential op. model *)
Equations run_outcome call :
(* state+error monad *)
seqmon (eff_ret call) :=
| RegRead reg ⇒
mget (lookup reg ○ regs)
| RegWrite reg val ⇒
mset (lookup reg ○ regs) val
| …
// 2. TinyArm type instantiation
register X3 : bits(64)
union barrier = { Barrier_DMB, ...
instantiation 'barrier = barrier
Sail trans.
(*5.instantiation: arch-specific types*)
Module Arch.
Inductive reg := X3 | ...
Definition barrier := Barrier_DMB | ...
// 3. TinyArm behaviour: LDR Xt,[Xn,Xm]
function clause execute LDR(t,n,m)={
let base_addr = X(n);
let offset = X(m);
let addr = base_addr+offset;
let data = MemRead(addr);
X(t) = data;
Sail trans.
(* 6. intra-instruction behaviour*)
Definition execute_LDR t n m : iMon ()
(rX n) >>= fun base_addr ⇒
(rX m) >>= fun offset ⇒
let addr := base_addr + offset in
(MemRead addr) >>= fun data ⇒
(wX t data).

```
// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
  : bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
```

fixed corresp.

```
(* 4. ArchSem Interface, in Rocq *)
Inductive outcome =
| RegRead (r: reg)(* eff_ret ... = bv 64*)
| RegWrite (r: reg) (val : bv 64) (* () *)
| MemRead (addr: bv 64) (* bv 64 *)
| MemWrite (addr: bv 64) (val : bv 64)(*()*)
| Barrier (b: barrier) (* () *)
Inductive iMon A =
| IRet (ret: A)
| INext (o: outcome) (k: eff_ret o → iMon A)
```

```
(* 7. sequential op. model *)
Equations run_outcome call :
(* state+error monad *)
    seqmon (eff_ret call) :=
| RegRead reg ⇒
  mget (lookup reg ○ regs)
| RegWrite reg val ⇒
  mset (lookup reg ○ regs) val
| …
```

```
// 2. TinyArm type instantiation
register X3 : bits(64)
union barrier = { Barrier_DMB, ...
instantiation 'barrier = barrier
```

Sail trans.

```
(*5.instantiation: arch-specific types*)
Module Arch.
  Inductive reg := X3 | ...
  Definition barrier := Barrier_DMB | ...
```

```
// 3. TinyArm behaviour: LDR Xt,[Xn,Xm]
function clause execute LDR(t,n,m)={
  let base_addr = X(n);
  let offset = X(m);
  let addr = base_addr+offset;
  let data = MemRead(addr);
  X(t) = data;
```

Sail trans.

```
(* 6. intra-instruction behaviour*)
Definition execute_LDR t n m : iMon ()
    (rX n) >>= fun base_addr ⇒
    (rX m) >>= fun offset ⇒
    let addr := base_addr + offset in
    (MemRead addr) >>= fun data ⇒
    (wX t data).
```

```
(* 8. LDR concrete trace*)
(*as in axiomatic models*)
[RegRead X3 &↦ 0x100;
 RegRead X4 &↦ 0x1000;
 MemRead 0x1100 &↦ 0x2a;
 RegWrite X7 0x2a &↦ ()]
```

Practical example

Let's run a simple instruction on a sequential model

Practical example

Let's run a simple instruction on a sequential model

```
LDR X1, [X0]
```

Practical example

Let's run a simple instruction on a sequential model

```
                addr = R0;
LDR X1, [X0]    val = MemRead(addr);
                R1 = val;
```

Practical example

Let's run a simple instruction on a sequential model

	`addr = R0;`	`addr ← RegRead R0;`
`LDR X1, [X0]`	`val = MemRead(addr);`	`val ← MemRead 8 addr Exp;`
	`R1 = val;`	`RegWrite R1 val`

Practical example

Let's run a simple instruction on a sequential model

```
LDR X1, [X0]
```

```
addr = R0;
val = MemRead(addr);
R1 = val;
```

Sequential model

$$
\begin{aligned}
\texttt{R0} &\mapsto \texttt{0x42} \\
\texttt{R1} &\mapsto 0 \\
\texttt{0x42} &\mapsto \texttt{0x2a}
\end{aligned}
$$

Practical example

Let's run a simple instruction on a sequential model

```
                addr = R0;
LDR X1, [X0]    val = MemRead(addr);
                R1 = val;
```

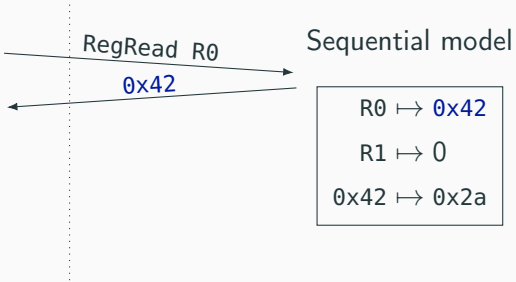
Sequential model

$$\begin{aligned} R0 &\mapsto 0x42 \\ R1 &\mapsto 0 \\ 0x42 &\mapsto 0x2a \end{aligned}$$

Practical example

Let's run a simple instruction on a sequential model

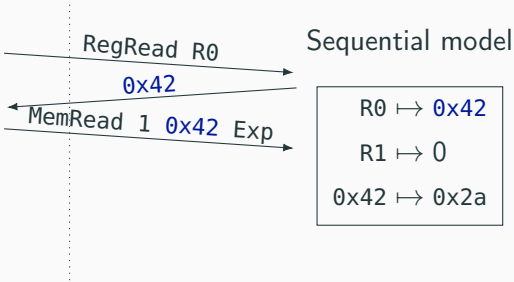
```
                addr = R0;
LDR X1, [X0]    val = MemRead(addr);
                R1 = val;
```



Practical example

Let's run a simple instruction on a sequential model

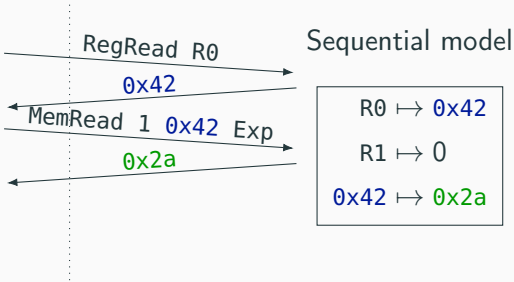
```
                addr = R0;
LDR X1, [X0]    val = MemRead(addr);
                R1 = val;
```



Practical example

Let's run a simple instruction on a sequential model

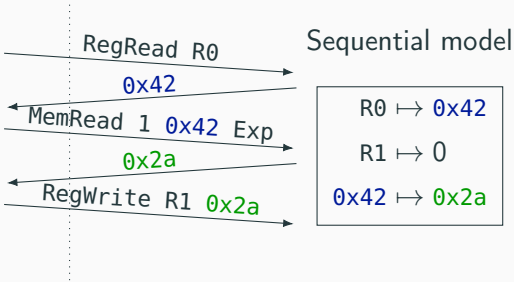
```
                addr = R0;
LDR X1, [X0]    val = MemRead(addr);
                R1 = val;
```



Practical example

Let's run a simple instruction on a sequential model

```
                addr = R0;
LDR X1, [X0]    val = MemRead(addr);
                R1 = val;
```

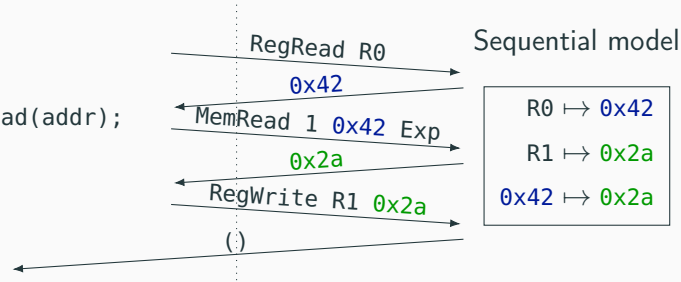


Practical example

Let's run a simple instruction on a sequential model

```
LDR X1, [X0]
```

```
addr = R0;
val = MemRead(addr);
R1 = val;
```



ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

However:

- Non-determinism

ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

However:

- Non-determinism (Execution limitations)

ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

However:

- Non-determinism (Execution limitations)
- Failure and UB

ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

However:

- Non-determinism (Execution limitations)
- Failure and UB (Not supported by concurency models)

ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

However:

- Non-determinism (Execution limitations)
- Failure and UB (Not supported by concurency models)
- Non-linear ordering

ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

However:

- Non-determinism (Execution limitations)
- Failure and UB (Not supported by concurency models)
- Non-linear ordering (Solved on paper but not implemented)

ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

However:

- Non-determinism (Execution limitations)
- Failure and UB (Not supported by concurency models)
- Non-linear ordering (Solved on paper but not implemented)

  Unsupported:CAS,CSEL,SWP + multiple memory accesses

ISA models as free monads

- ISA model can be represented as an effectful program over a precise list of effects.
- ISA model must always be (per-instruction) terminating
- Therefore: free monad (not need for itrees)

However:

- Non-determinism (Execution limitations)
- Failure and UB (Not supported by concurency models)
- Non-linear ordering (Solved on paper but not implemented)
  Unsupported:CAS,CSEL,SWP + multiple memory accesses

Allows decomposing reasoning between ISA and concurrency

Concurrency model implementation

Axiomatic models

- Use intra-instruction trace
- Stitch into candidate executions
- Define usual relations
- Define well-formedness
- Define your axioms

Concurrency model implementation

Axiomatic models

- Use intra-instruction trace
- Stitch into candidate executions
- Define usual relations
- Define well-formedness
- Define your axioms

Operational models

- Define your state
- Interpret the free monad
- Use that to define your transitions

Concurrency model implementation

Axiomatic models

- Use intra-instruction trace
- Stitch into candidate executions
- Define usual relations
- Define well-formedness
- Define your axioms

Operational models

- Define your state
- Interpret the free monad
- Use that to define your transitions

VM Promising model (with Yeji Han)

Update/improvement of the ESOP 2022 model by Ben Simner et al.

- Each memory access preceded by a translation (with some corresponding views)
- Maximal operational TLB model: Cache everything that is allowed to be cached.
- Add TLB Invalidate (TLBI) instructions to the main memory trace
- Relaxed system register history per-thread.
- Managed translation failure faults and returns

We found many behaviours implicitly allowed by axiomatic model that shouldn't have been

What can we do with this?

Proving architecture properties

Model equivalence

Sanity checks e.g.
seq $\Longleftrightarrow$ SC ax on single thread
(Nils Lauermann)

Model refinement

e.g. VM to UM abstraction theorem

Reasoning about programs

AxSL

Reasoning about axiomatic weak memory model with load buffering

Open: Systems program logic

What kind of ressources to represent CPU components:

- Page tables
- Incoherent caches
- Interrupts

VM Theorem – cartoon (with Thomas Baureiss)

Theorem (VM abstraction – very cartoon version)

The armv9-A architecture can be configured to provide a sound illusion of virtual memory.

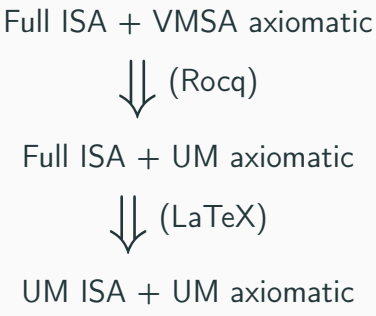
Theorem (VM abstraction – cartoon version)

In any armv9-A machine state at EL0 with a simple Stage 1 page-table configuration that doesn't map the page-tables themselves, any EL0 execution allowed by the VMSA model (full ISA + VMSA axiomatic) is also allowed by the User model (ISA without translation + User axiomatic).

VM Theorem – sketch

First: Isabelle ISA-only proof

Then:



AxSL

An Axiomatic Basis for Computer Programming on Relaxed Hardware Architectures: The AxSL Logics

Zongyuan Liu, A Hammond, T. Pérami, P. Sewell, L. Birkedal, and J. Pichon-Pharabod

TOPLAS (extension of POPL24 paper)

An Axiomatic Basis for Computer Programming on Relaxed Hardware Architectures: The AxSL Logics

Zongyuan Liu, A Hammond, T. Pérami, P. Sewell, L. Birkedal, and J. Pichon-Pharabod

TOPLAS (extension of POPL24 paper)

HT-MICRO-MEMREAD-RDEP-EXT

$$\left\{ \begin{array}{l} (1)\mathrm{NoLocalWrites}(x) * (2)d = (\mathrm{dom}(regs), \emptyset) * \\ (3)\mathrm{PoPred}(e_{\mathrm{po}}) * (4)\mathrm{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\ (5) \mathop{\Large *}_{(r \mapsto v@E) \in regs} r \overset{\mathrm{r}}{\mapsto} v@E * (6) \mathop{\Large *}_{(e_{\mathrm{lob}} \mapsto P_{\mathrm{lob}}) \in m} (e \rightsquigarrow P_{\mathrm{lob}}) * \\ \forall e, v, e_w. \left(\begin{array}{l} (7)\mathrm{GraphFacts}(e, os, vr, x, v, e_w, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) \mathrel{-\!\!*} \\ ((8)\mathrm{Lob}(\mathrm{dom}(m), e) * (9)\mathrm{Flow}_{\Phi}(e, x, v, e_w, m, P)) \end{array} \right) \end{array} \right\}$$

MemRead *os vr x d*

$$\left\{ v, E. \begin{array}{l} \exists e, e_w.\, E = \{e\} * (10)\mathrm{NoLocalWrites}(x) * (11)\mathrm{PoPred}(e) * (12)\mathrm{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\ (13) \mathop{\Large *}_{(r \mapsto v@E) \in regs} r \overset{\mathrm{r}}{\mapsto} v@E * (14)\mathrm{GraphFacts}(e, os, vr, x, v, e_w, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) * \\ (15) e \rightsquigarrow P(x, v, e_w) \end{array} \right\}_{tid,\Phi}$$

How to use this to prove systems code?

- 95% of the code is standard imperative sequential code
  - We sort of know how to deal with that (maybe not to full scale)

How to use this to prove systems code?

- 95% of the code is standard imperative sequential code
  - We sort of know how to deal with that (maybe not to full scale)
- 5% interact with hardware features and weak memory concurrency
  - That the part we don't know how to do well

Future work and Conclusion

- Proof structure questions:
  - For the 95%: we can use a simplified but sound model then prove refinement

Future work and Conclusion

- Proof structure questions:
  - For the 95%: we can use a simplified but sound model then prove refinement
  - How do we drop down to the level needed by the 5%? A mix of:
    - Building stacks of model refinement
    - Building a logic on low-level layer and stacks of logical abstractions

Future work and Conclusion

- Proof structure questions:
  - For the 95%: we can use a simplified but sound model then prove refinement
  - How do we drop down to the level needed by the 5%? A mix of:
    - Building stacks of model refinement
    - Building a logic on low-level layer and stacks of logical abstractions
- Partiality (Handling UB)
- Intra-instruction parallelism
- Full test campaign against the state of the art
- More models
- More proofs relating models
- Lean?

Thank you for your attention!

Why multiple styles of models

Axiomatic models

Naturally too relaxed:

- Need explicit constraints
- Easy to allow unwanted behaviours

Operational models

Naturally too strict:

- More relaxed $\Longrightarrow$ More complexity

Writing in both styles helps identify unconsidered cases,.

Why multiple styles of models

Axiomatic models

Naturally too relaxed:

- Need explicit constraints
- Easy to allow unwanted behaviours

Operational models

Naturally too strict:

- More relaxed $\implies$ More complexity

Writing in both styles helps identify unconsidered cases,.

"Bugs" we found by trying to operationalize models

- Read commit problems for mixed-size axiomatic Arm vs Flat
- TLB ordering questions, discovered when making Promising Arm for VMSA

Partiality

Fact1: Model inclusion/refinement requires correct handling of UB

Partiality

Fact1: Model inclusion/refinement requires correct handling of UB

Fact2: Proper handling of UB in axiomatic model requires partial executions

Partiality

Fact1: Model inclusion/refinement requires correct handling of UB

Fact2: Proper handling of UB in axiomatic model requires partial executions

Partial executions

- "Easy" with acyclicity of `(p)po` $\cup$ `rf` (c.f. GenMC)
- For Arm with Load-Buffering: hard (WIP)

Partiality

Fact1: Model inclusion/refinement requires correct handling of UB

Fact2: Proper handling of UB in axiomatic model requires partial executions

Partial executions

- "Easy" with acyclicity of `(p)po` $\cup$ `rf` (c.f. GenMC)
- For Arm with Load-Buffering: hard (WIP)

Speculation vs Completion

Partiality

Fact1: Model inclusion/refinement requires correct handling of UB

Fact2: Proper handling of UB in axiomatic model requires partial executions

Partial executions

- "Easy" with acyclicity of `(p)po` $\cup$ `rf` (c.f. GenMC)
- For Arm with Load-Buffering: hard (WIP)

Speculation vs Completion

Plan and paper math but no implementation

Intra instruction parallelism

So far: Internally sequenced instructions and free/choice monads

Intra instruction parallelism

So far: Internally sequenced instructions and free/choice monads

```
| Async : iMon A → (unit → iMon (future A))
| Await : future A → (A → iMon (future A))
```

Intra instruction parallelism

So far: Internally sequenced instructions and free/choice monads

```
| Async : iMon A → (unit → iMon (future A))
| Await : future A → (A → iMon (future A))
```

Internal control dependencies

`iio_data` vs `iio_ctrl`

```
| Speculate : future A → (A → iMon (future A))
```

Intra instruction parallelism

So far: Internally sequenced instructions and free/choice monads

```
| Async : iMon A → (unit → iMon (future A))
| Await : future A → (A → iMon (future A))
```

Internal control dependencies

`iio_data` vs `iio_ctrl`

```
| Speculate : future A → (A → iMon (future A))
```

Affected instructions

CAS, CSEL, LDP, STP, any unaligned access, 95% of SIMD load and stores