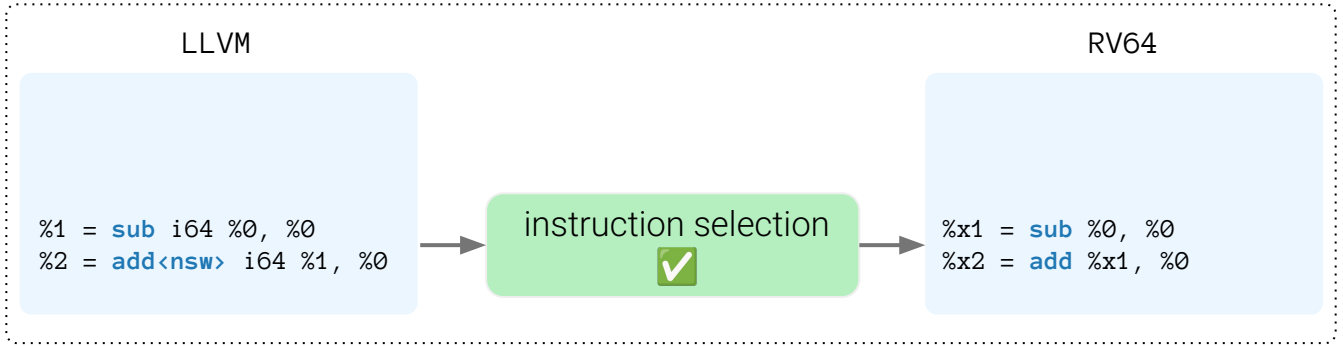
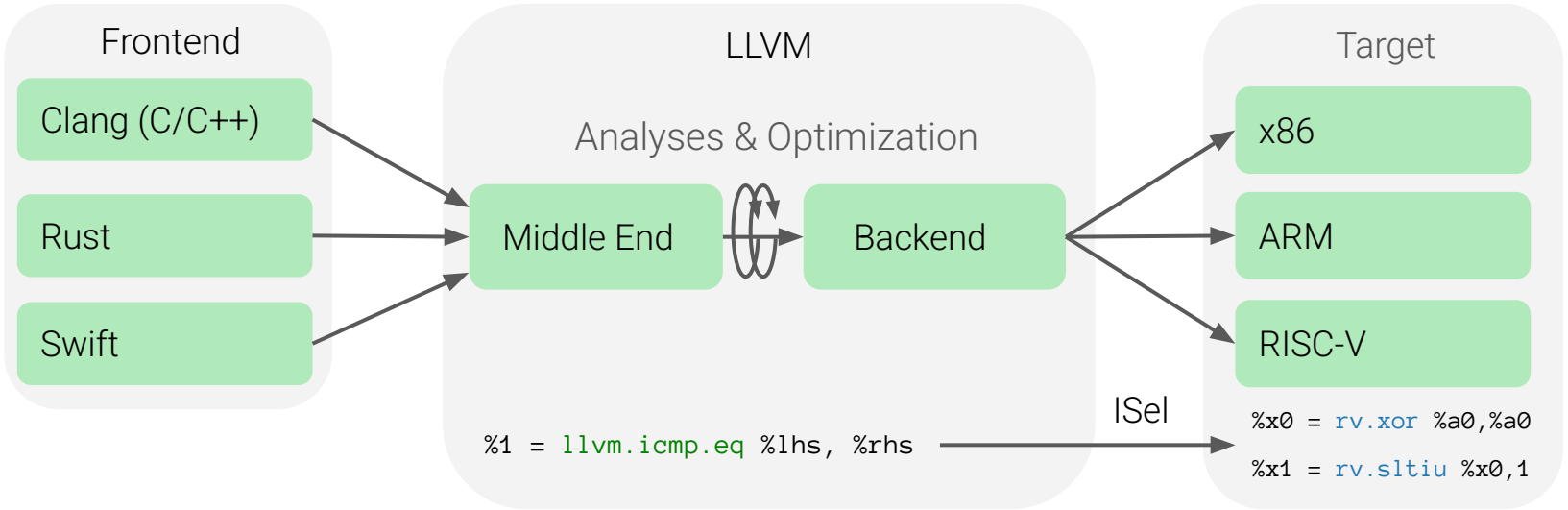
# Scaling Certified Instruction Selection For LLVM IR Through Bitblasting

***Sarah Linh Kuhn***, ***Luisa Cicolini***, *Alex Keizer*, ***Osman Yasar,*** *Tobias Grosser*



# Many Programming Languages Rely On LLVM

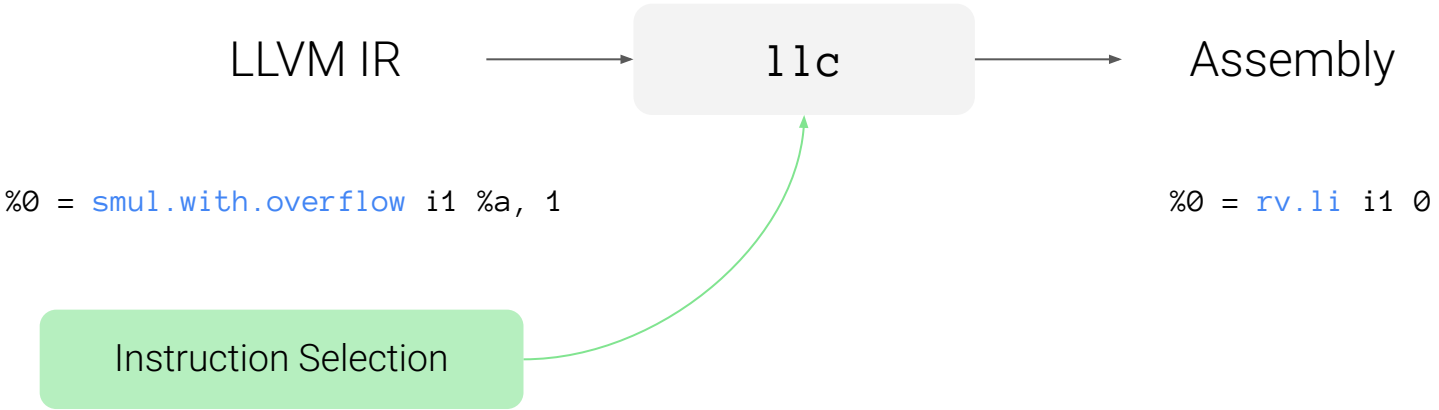


# A look at LLVM's *backend*

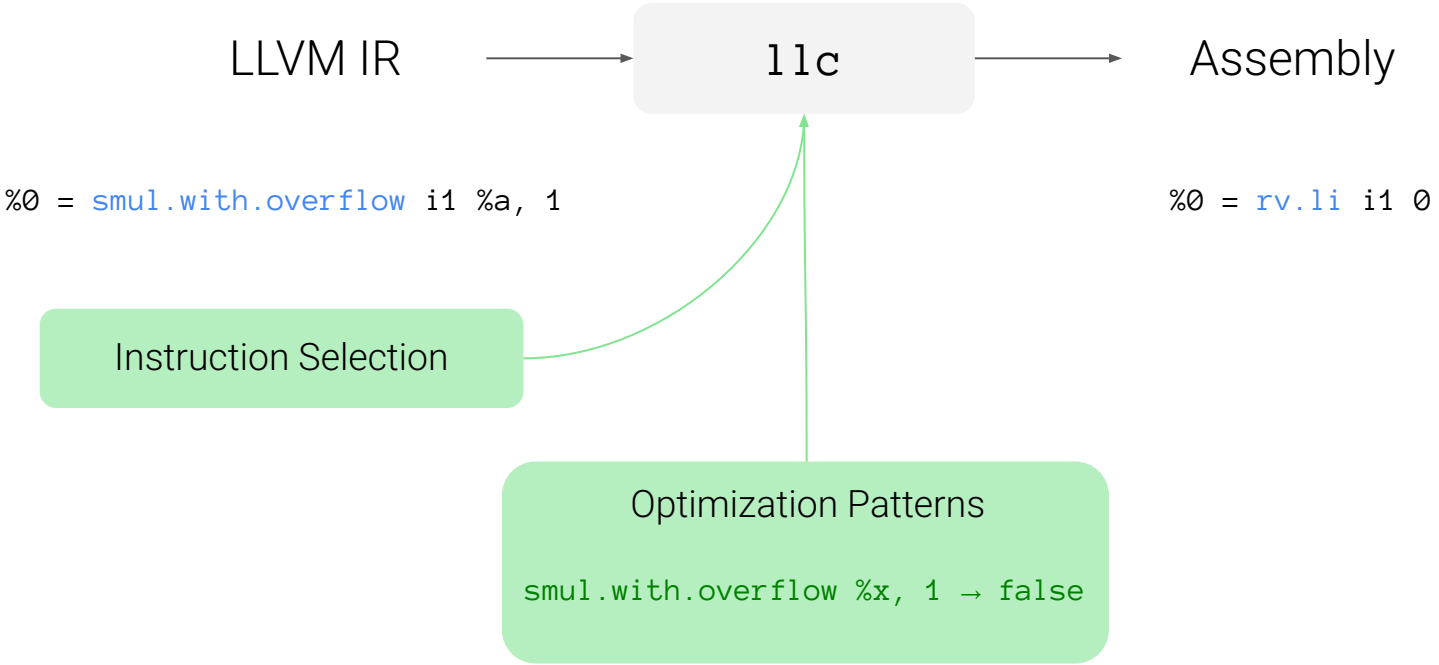

```
%0 = smul.with.overflow i1 %a, 1
```

```
%0 = rv.li i1 0
```

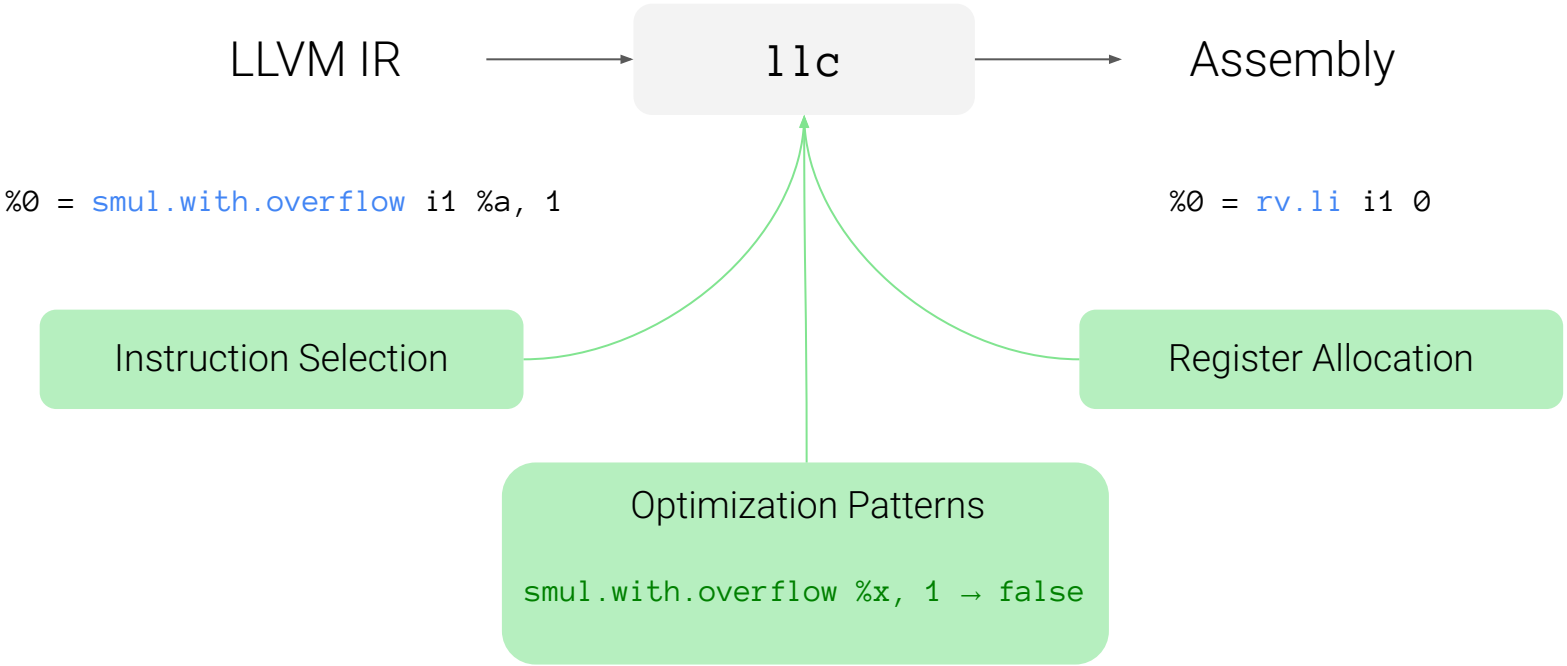
# A look at LLVM's *backend*



# A look at LLVM's *backend*



# A look at LLVM's *backend*



Optimization Patterns

```
smul.with.overflow %x, 1 → false
```

# LLVM has *optimizations*

`x : i2 ⟶ signed integer`

Optimization Patterns

`smul.with.overflow %x, 1 → false`

# LLVM has *optimizations*

`x : i2 ⟶ signed integer`

| * | −2 | −1 | 0 | 1 |
|---|---|---|---|---|
| −2 | 4 | 2 | 0 | −2 |
| −1 | 2 | 1 | 0 | −1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | −2 | −1 | 0 | 1 |

Optimization Patterns

`smul.with.overflow %x, 1 → false`

# LLVM has *optimizations…*

`x : i1 ⟶ signed integer`

| * | −2 | −1 | 0 | 1 |
|---|---|---|---|---|
| −2 | 4 | 2 | 0 | −2 |
| −1 | 2 | 1 | 0 | −1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | −2 | −1 | 0 | 1 |

Optimization Patterns

`smul.with.overflow %x, 1 → false`

# LLVM has *bugs*

`x : i1 ⟶ signed integer`

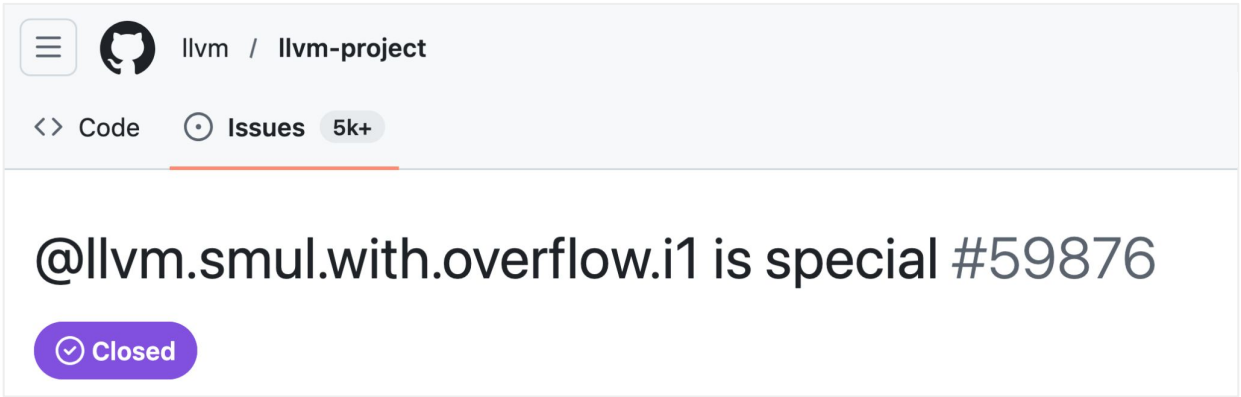
| * | −2 | −1 | 0 | 1 |
|---|---|---|---|---|
| −2 | 4 | 2 | 0 | −2 |
| −1 | 2 | 1 | 0 | −1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | −2 | −1 | 0 | 1 |

1 is interpreted as -1!

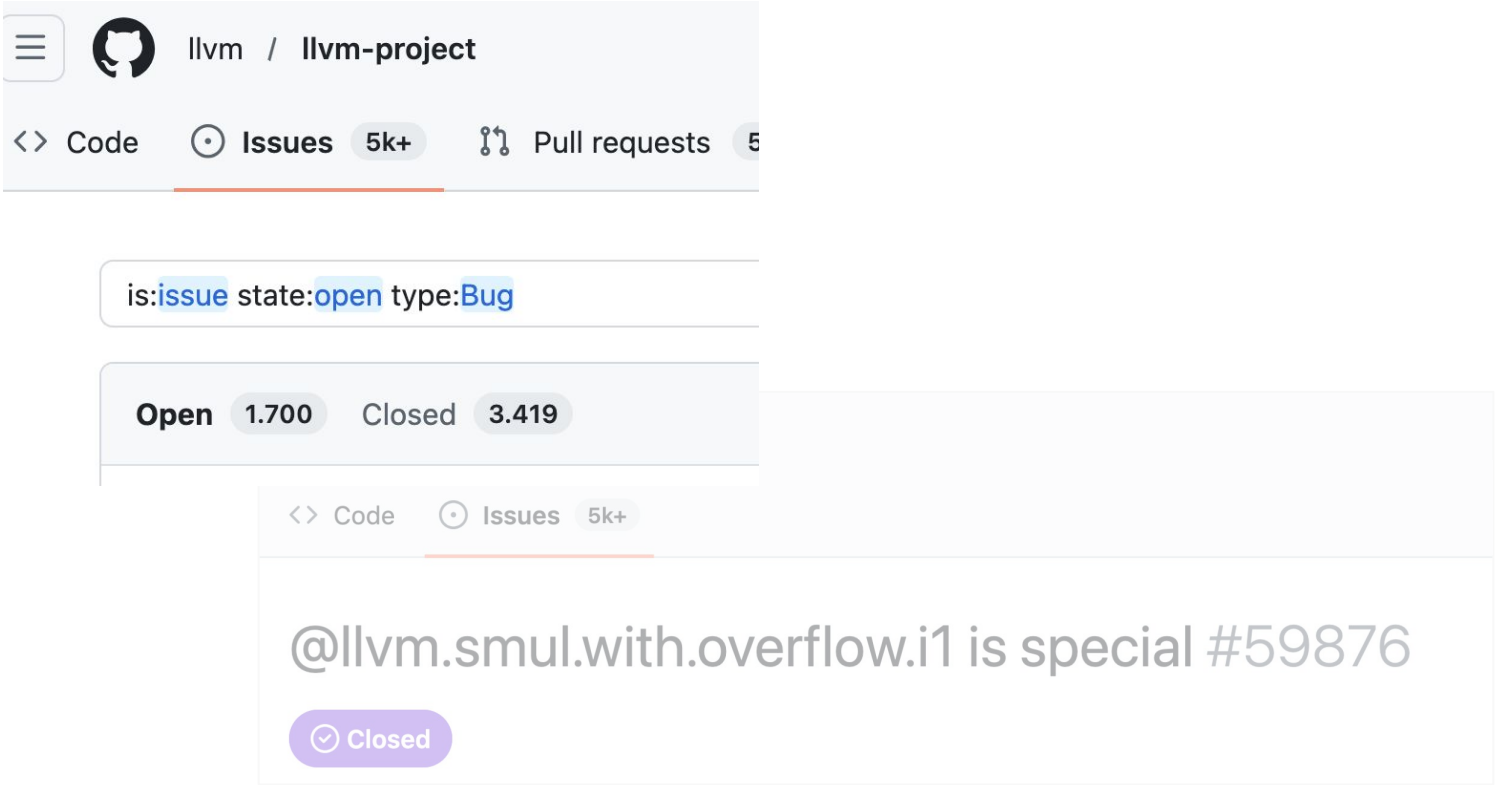
Optimization Patterns

```
smul.with.overflow %x, 1 → false
```

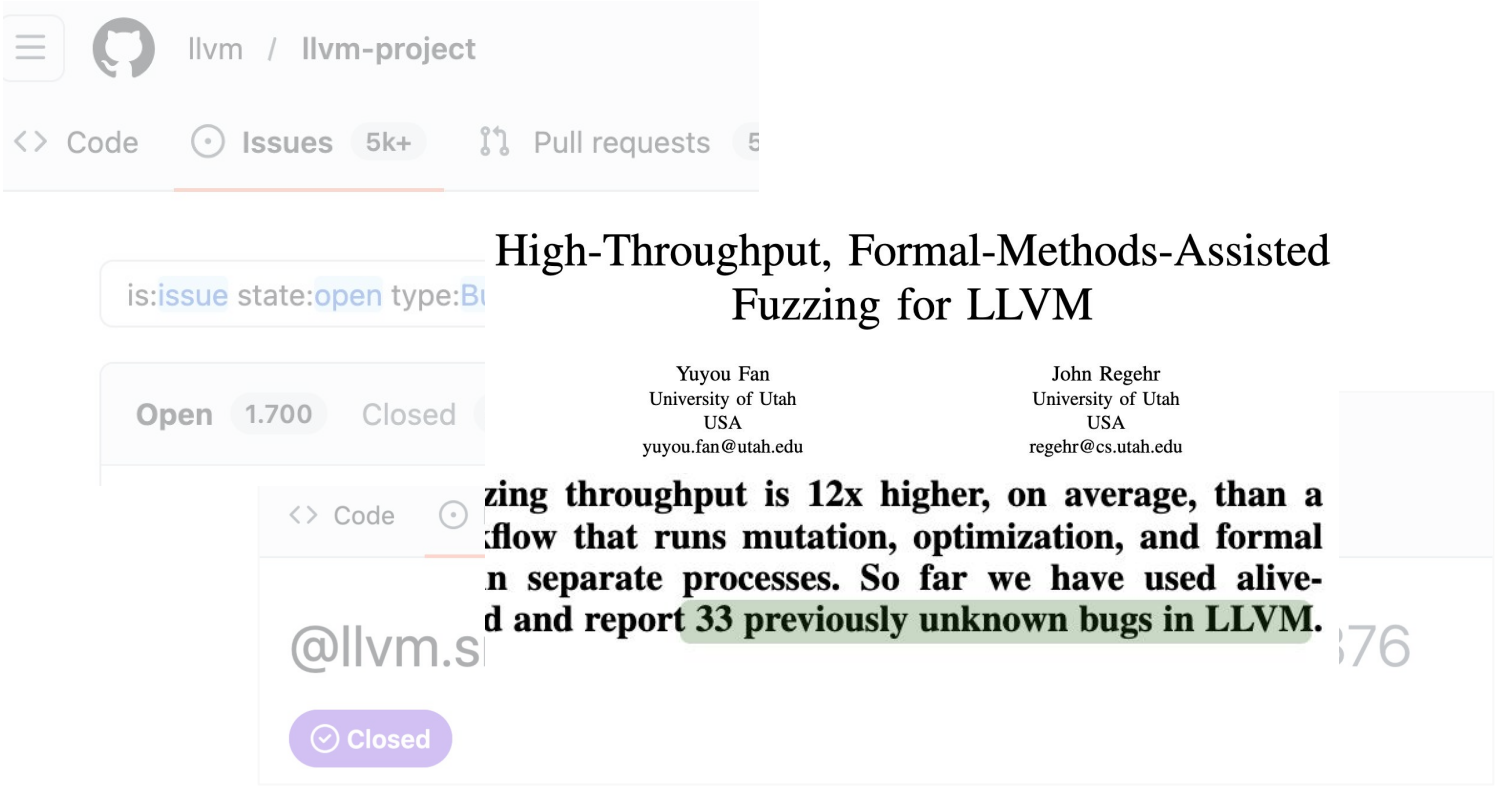
# LLVM has *bugs*



# LLVM has *bugs*



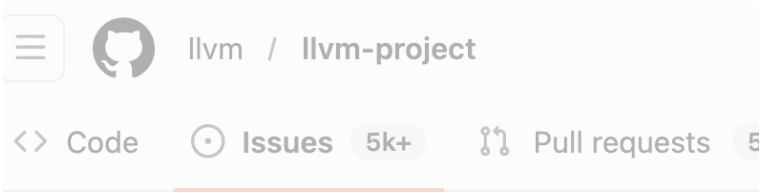
# LLVM has *bugs*

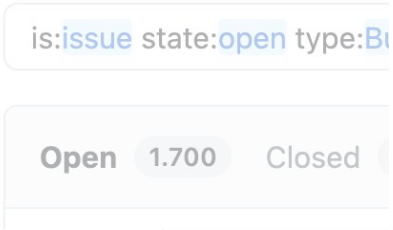


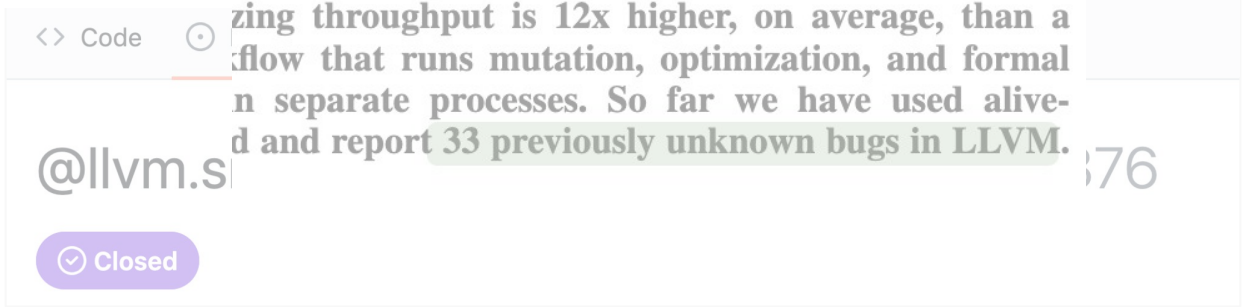



Fan, Yuyou, and John Regehr. "High-Throughput, Formal-Methods-Assisted Fuzzing for LLVM." *CGO'24*.

# LLVM has *bugs*




High-Throughput, Forma
Fuzzing for

Yuyou Fan
University of Utah
USA
yuyou.fan@utah.edu

zing throughput is 12x higher, on average, than a
flow that runs mutation, optimization, and formal
n separate processes. So far we have used alive-
d and report 33 previously unknown bugs in LLVM.

876

**Alive2: Bounded Translation Validation for LLVM**

Nuno P. Lopes
nlopes@microsoft.com
Microsoft Research
UK

Juneyoung Lee
juneyoung.lee@sf.snu.ac.k
Seoul National University
South Korea

Chung-Kil Hur
gil.hur@sf.snu.ac.k
Seoul National University
South Korea

Zhengyang Liu
liuz@cs.utah.edu
University of Utah
USA

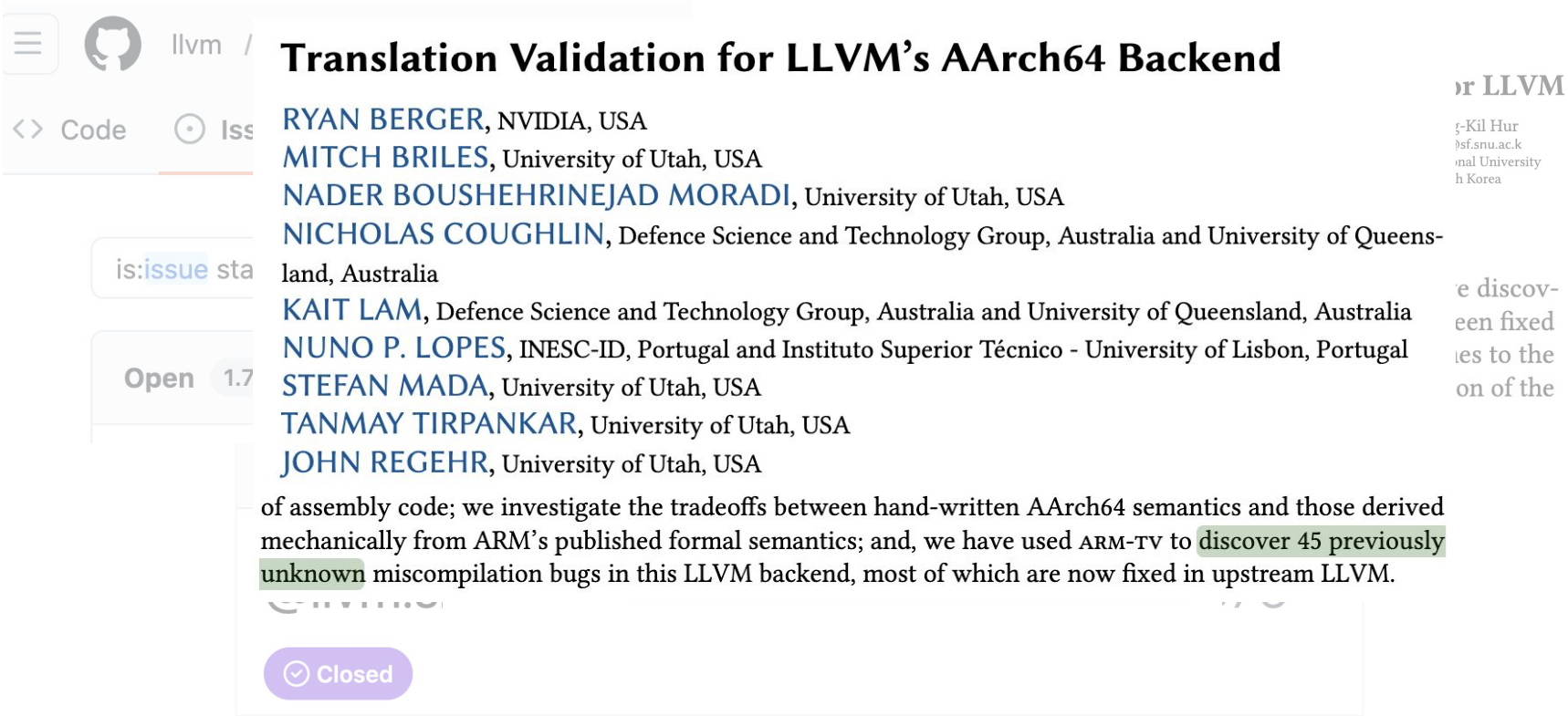
John Regehr
regehr@cs.utah.edu
University of Utah
USA

By running Alive2 over LLVM's unit test suite, we discovered and reported 47 new bugs, 28 of which have been fixed already. Moreover, our work has led to eight patches to the LLVM Language Reference—the definitive description of the



Lopes, Nuno P., et al. "Alive2: bounded translation validation for LLVM." *PLDI'21*.

# LLVM has *bugs*



## Translation Validation for LLVM's AArch64 Backend

RYAN BERGER, NVIDIA, USA
MITCH BRILES, University of Utah, USA
NADER BOUSHEHRINEJAD MORADI, University of Utah, USA
NICHOLAS COUGHLIN, Defence Science and Technology Group, Australia and University of Queensland, Australia
KAIT LAM, Defence Science and Technology Group, Australia and University of Queensland, Australia
NUNO P. LOPES, INESC-ID, Portugal and Instituto Superior Técnico - University of Lisbon, Portugal
STEFAN MADA, University of Utah, USA
TANMAY TIRPANKAR, University of Utah, USA
JOHN REGEHR, University of Utah, USA

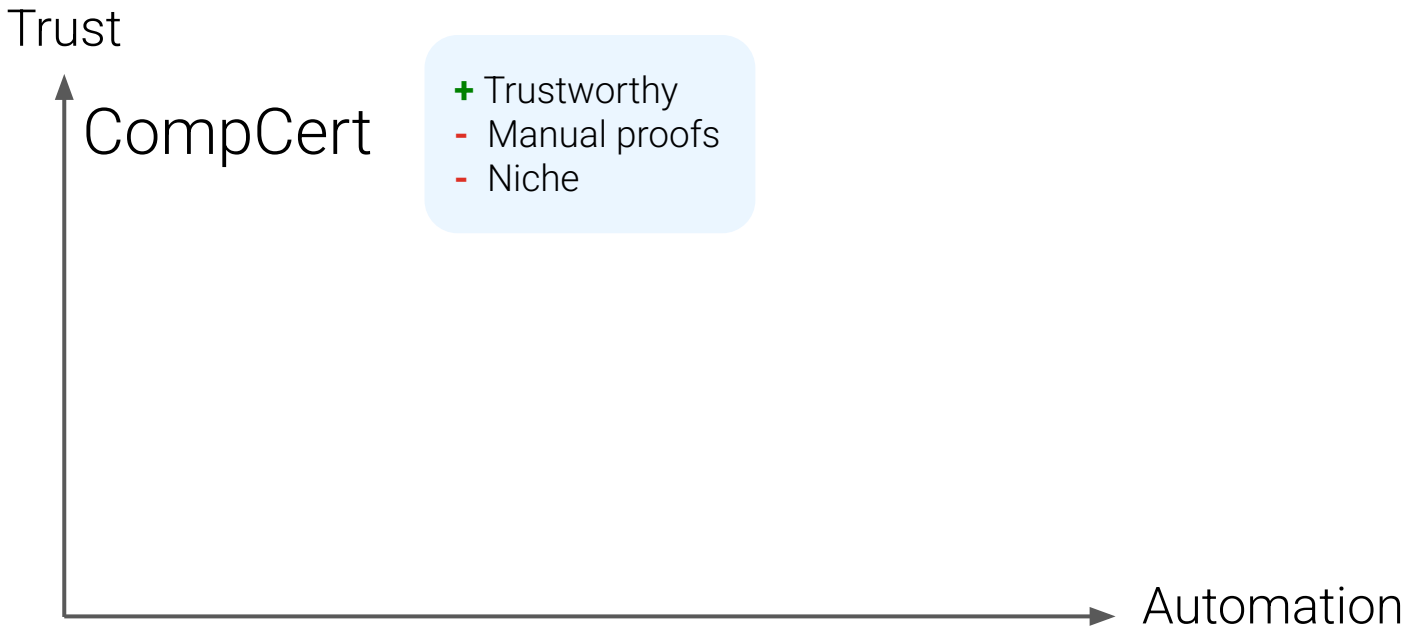
of assembly code; we investigate the tradeoffs between hand-written AArch64 semantics and those derived mechanically from ARM's published formal semantics; and, we have used ARM-TV to discover 45 previously unknown miscompilation bugs in this LLVM backend, most of which are now fixed in upstream LLVM.



Berger, Ryan, et al. "Translation Validation for LLVM's AArch64 Backend." *OOPSLA'25*

# A *Trade-off*: Trust vs. Automation

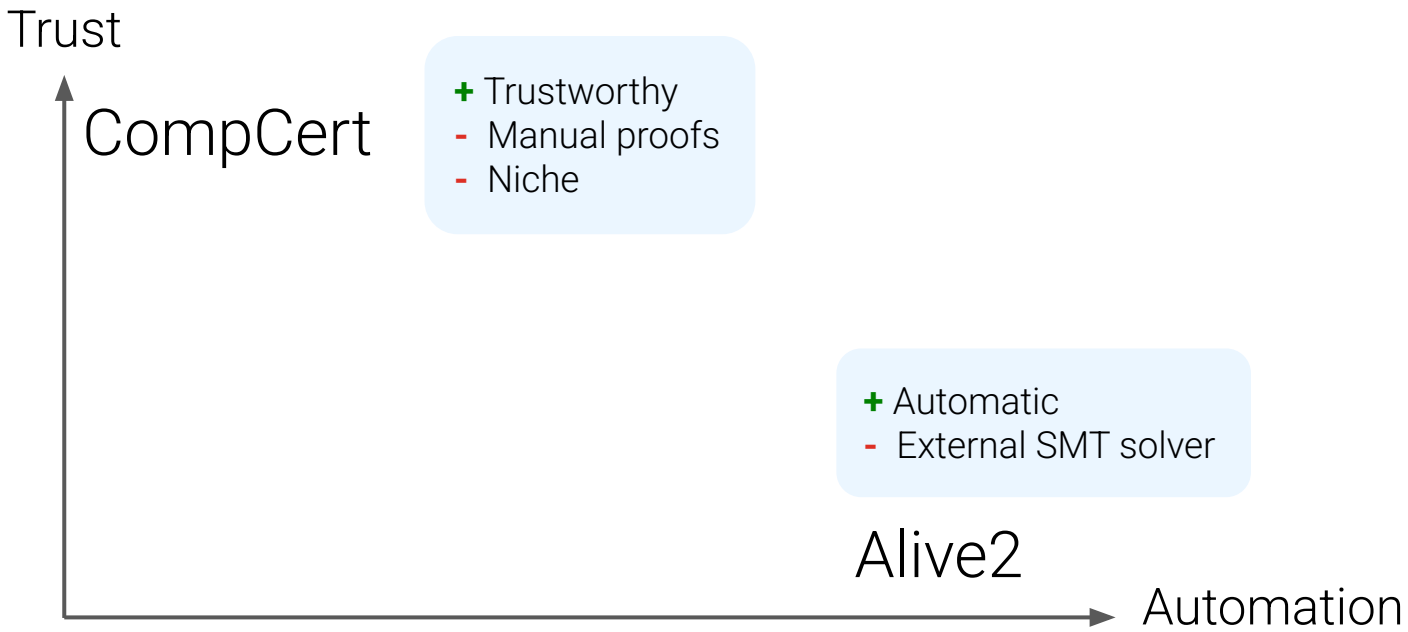
# *Fully-verified* compilation






Leroy, Xavier. "The CompCert verified compiler." Documentation and user's manual. INRIA Paris-Rocquencourt 53 (2012).

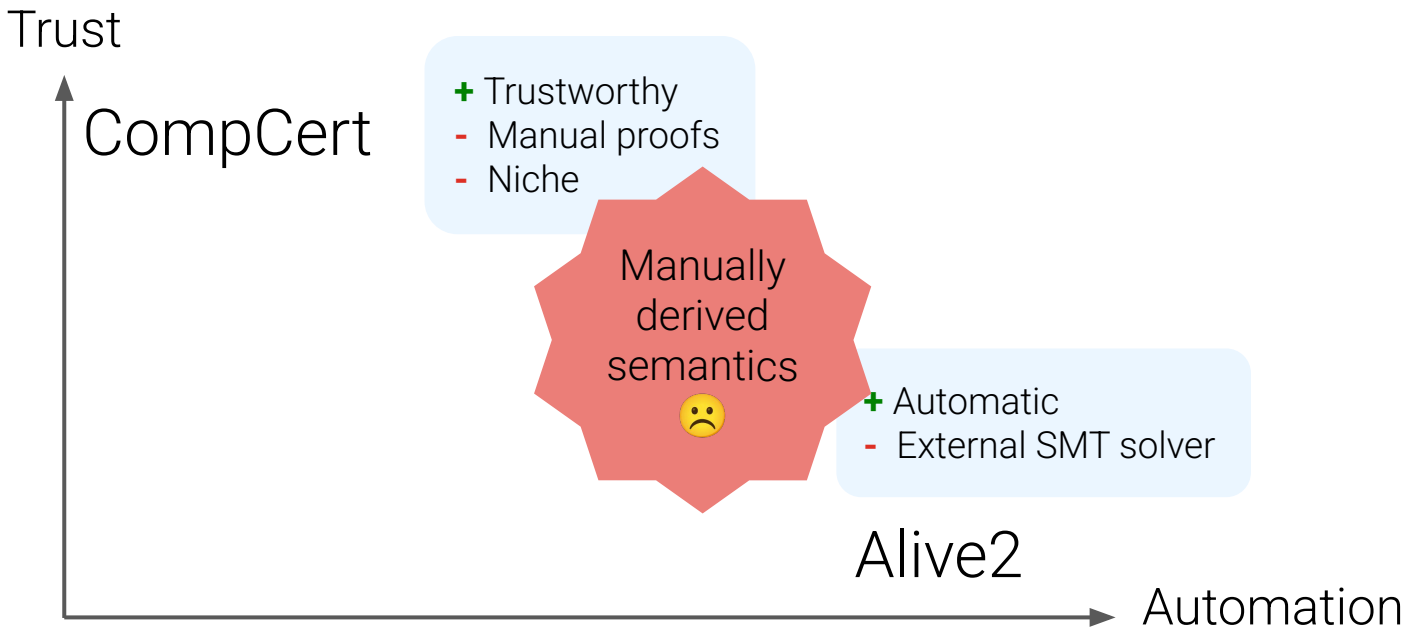
# *Fully-verified* compilation vs. *automated* translation-validation



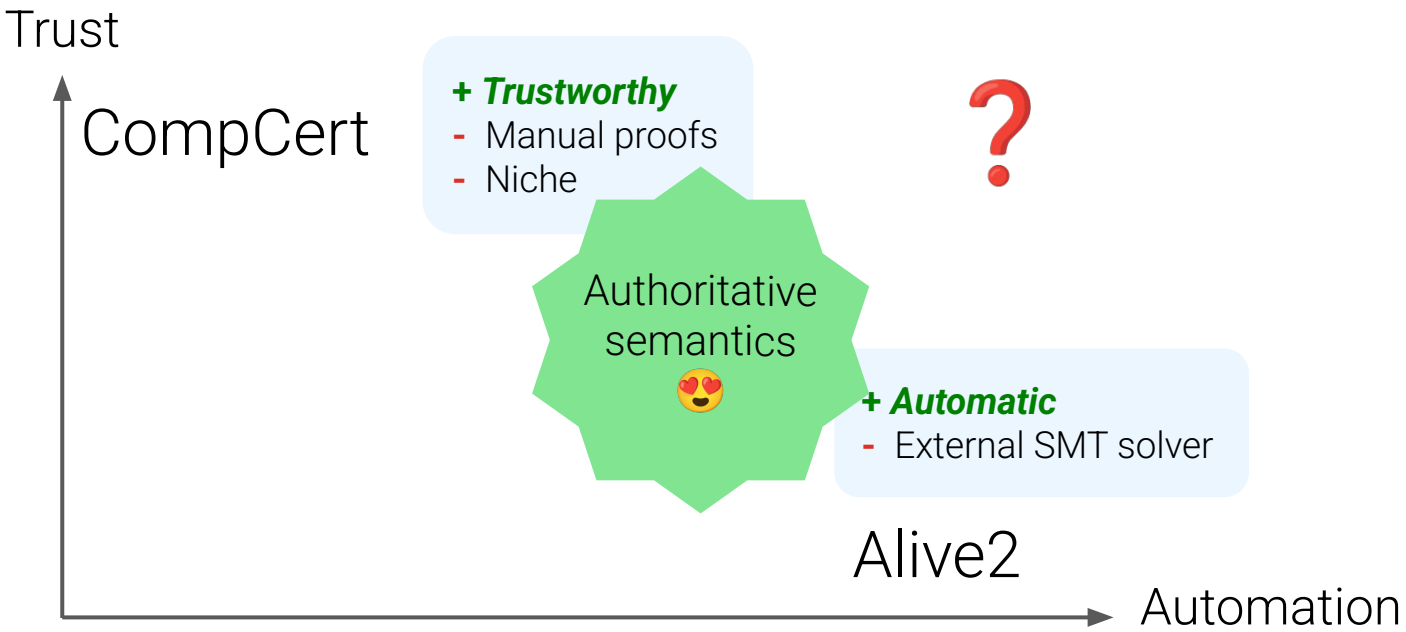



Lopes, Nuno P., et al. "Alive2: bounded translation validation for LLVM." *PLDI'21*.

# What about the *semantics*?

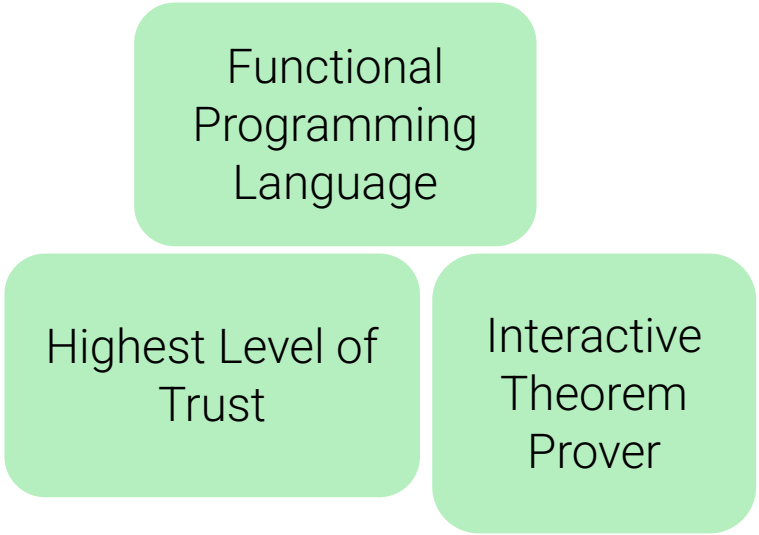


# Can we have it all?

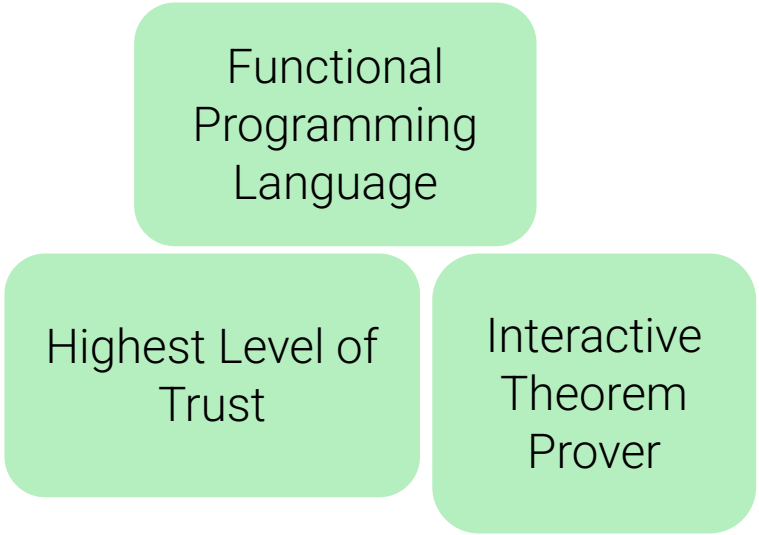


# We use Lean

# We use Lean

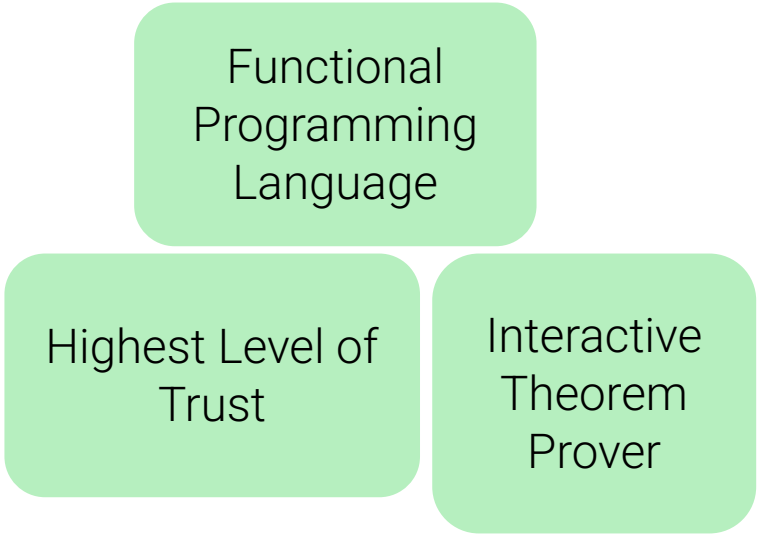

Functional Programming Language
Highest Level of Trust
Interactive Theorem Prover

# We use Lean




```
def main : IO Unit := IO.println "Hello, world!"
```

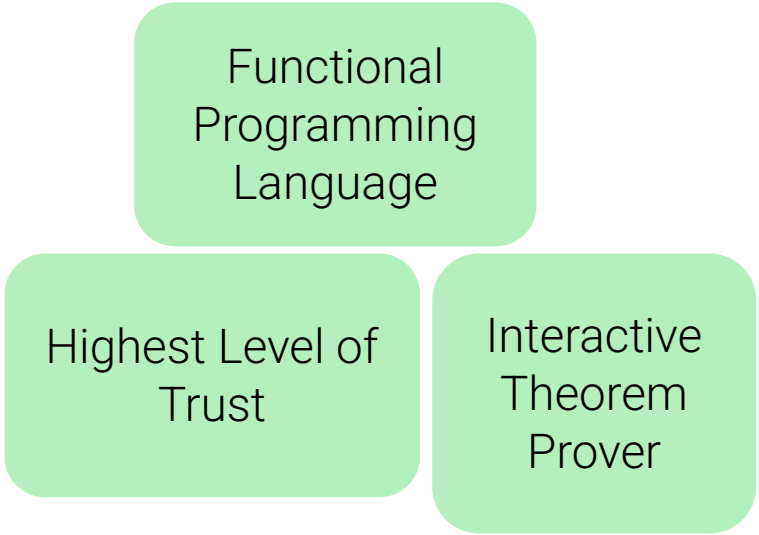
# We use Lean




```
def main : IO Unit := IO.println "Hello, world!"
```

```
def mySum : Nat := Id.run do
 let mut acc := 0
 for i in [0:4] do
   acc := acc + i
 pure acc
```

# We use Lean

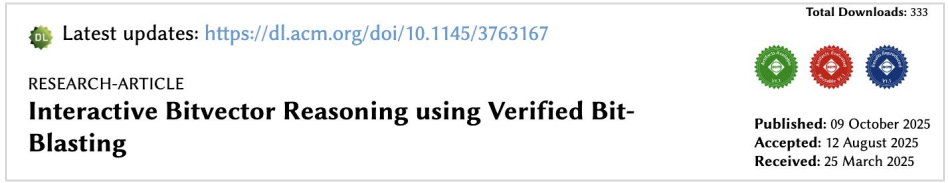



```
def main : IO Unit := IO.println "Hello, world!"
```

```
def mySum : Nat := Id.run do
 let mut acc := 0
 for i in [0:4] do
   acc := acc + i
 pure acc
```

```
theorem thm : mySum = 6 := by
 simp [mySum, List.foldl, List.range'] ✅
```

# Lean and its *bitvector* library and verified bitblaster



Latest updates: https://dl.acm.org/doi/10.1145/3763167

Total Downloads: 333

RESEARCH-ARTICLE

**Interactive Bitvector Reasoning using Verified Bit-Blasting**

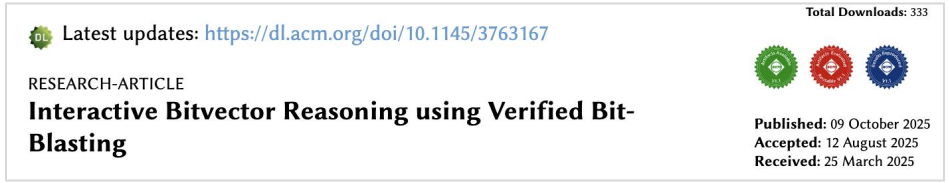
**Published:** 09 October 2025
**Accepted:** 12 August 2025
**Received:** 25 March 2025

```
bv_decide ✅
```



Böving, Henrik, et al. "Interactive bitvector reasoning using verified bit-blasting." *OOPSLA'25*

# Lean and its *bitvector* library and verified bitblaster



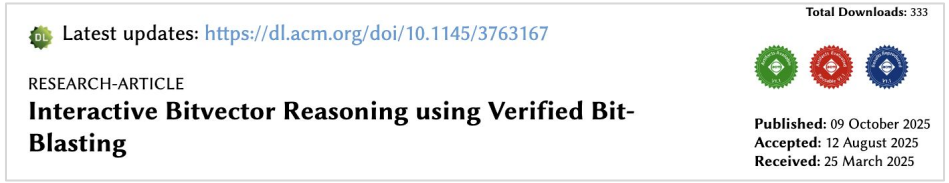

`bv_decide` ✅

```
example (x : BitVec 32) : (y | x) + (y & x) = y + x
```



https://github.com/llvm/llvm-project/tree/main/llvm/test/Transforms/InstCombine

# Lean and its *bitvector* library and verified bitblaster

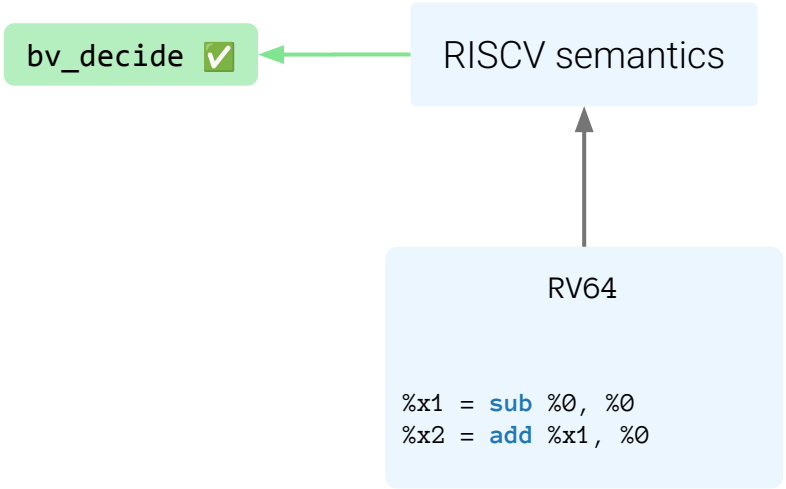



`bv_decide` ✅

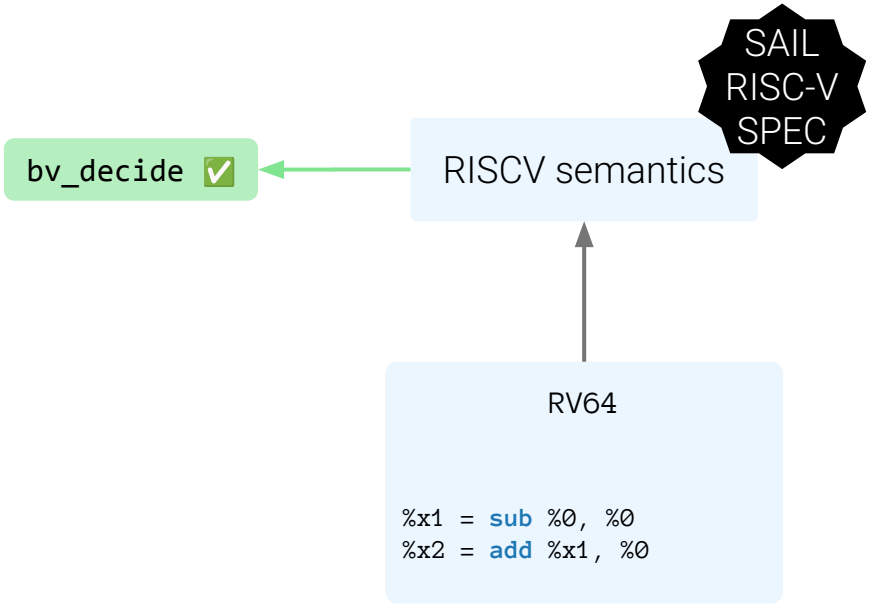
```
example (x : BitVec 32) : (y | x) + (y & x) = y + x
```
✅

https://github.com/llvm/llvm-project/tree/main/llvm/test/Transforms/InstCombine

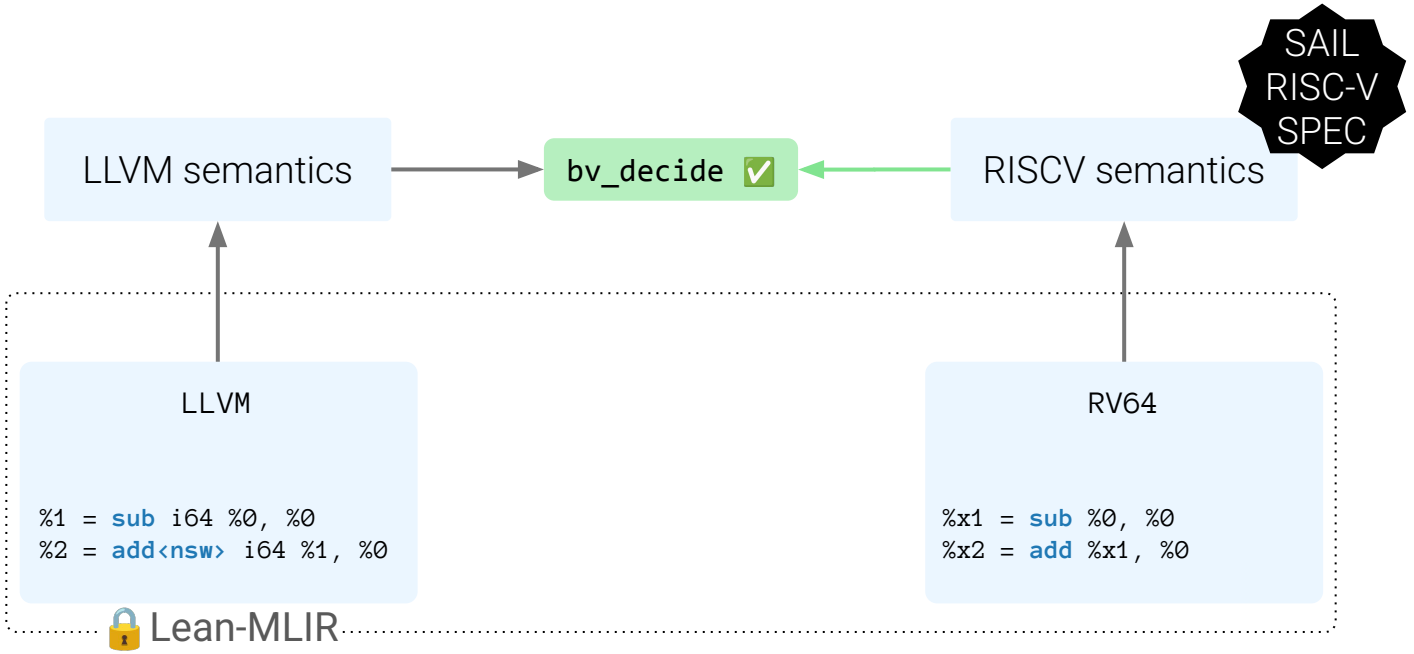
# A *bitblastable* mechanization of the RISC-V ISA



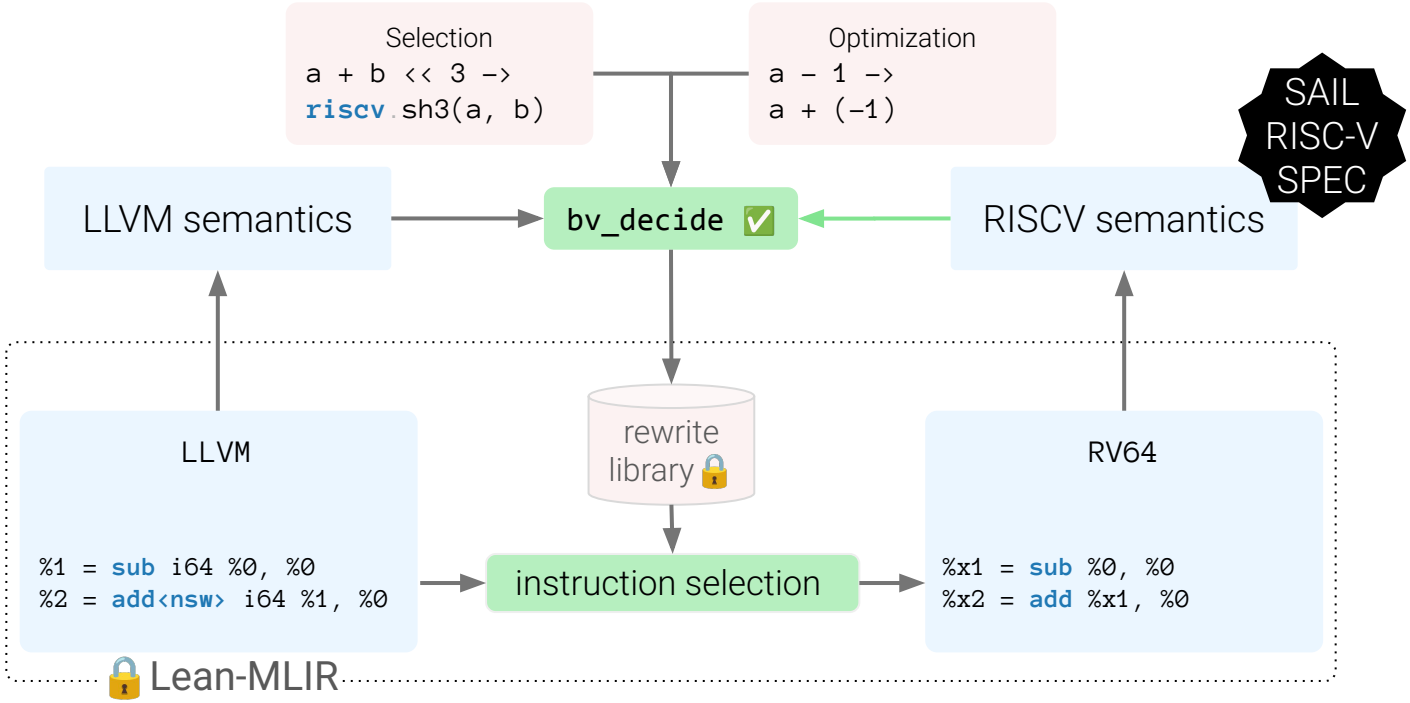
# Equivalent to the *Sail* processor model



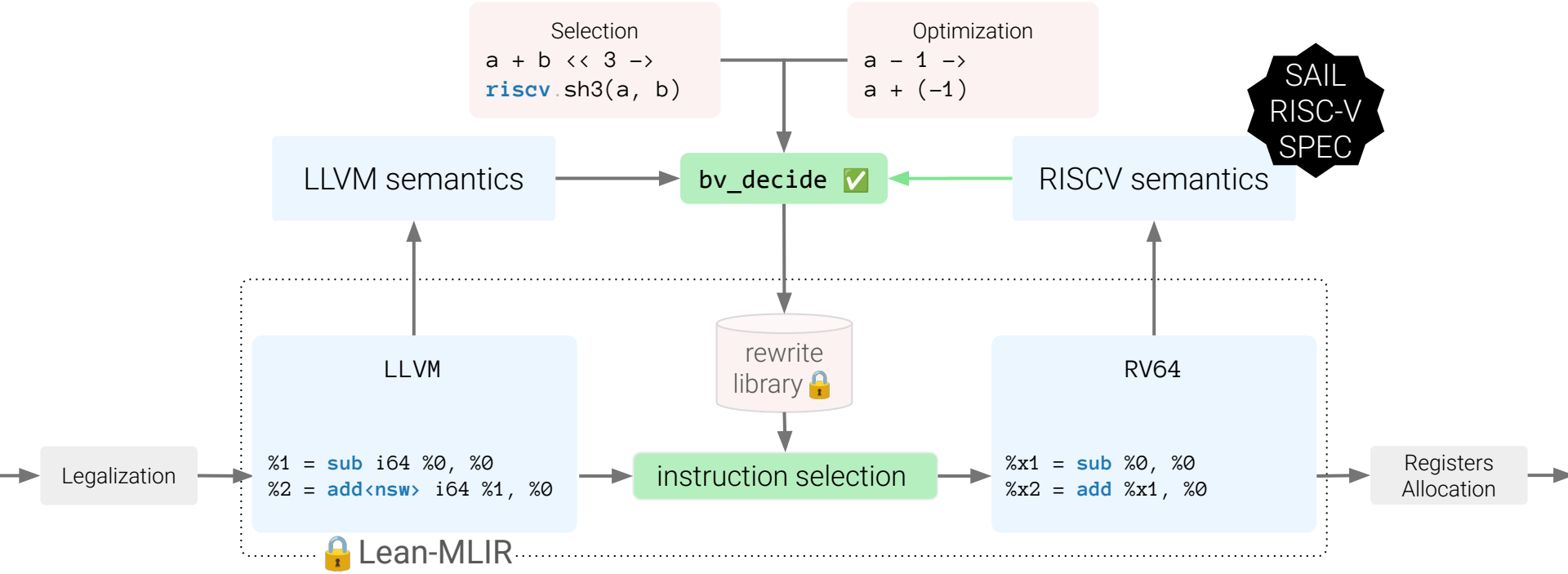
# In the *Lean-MLIR* framework



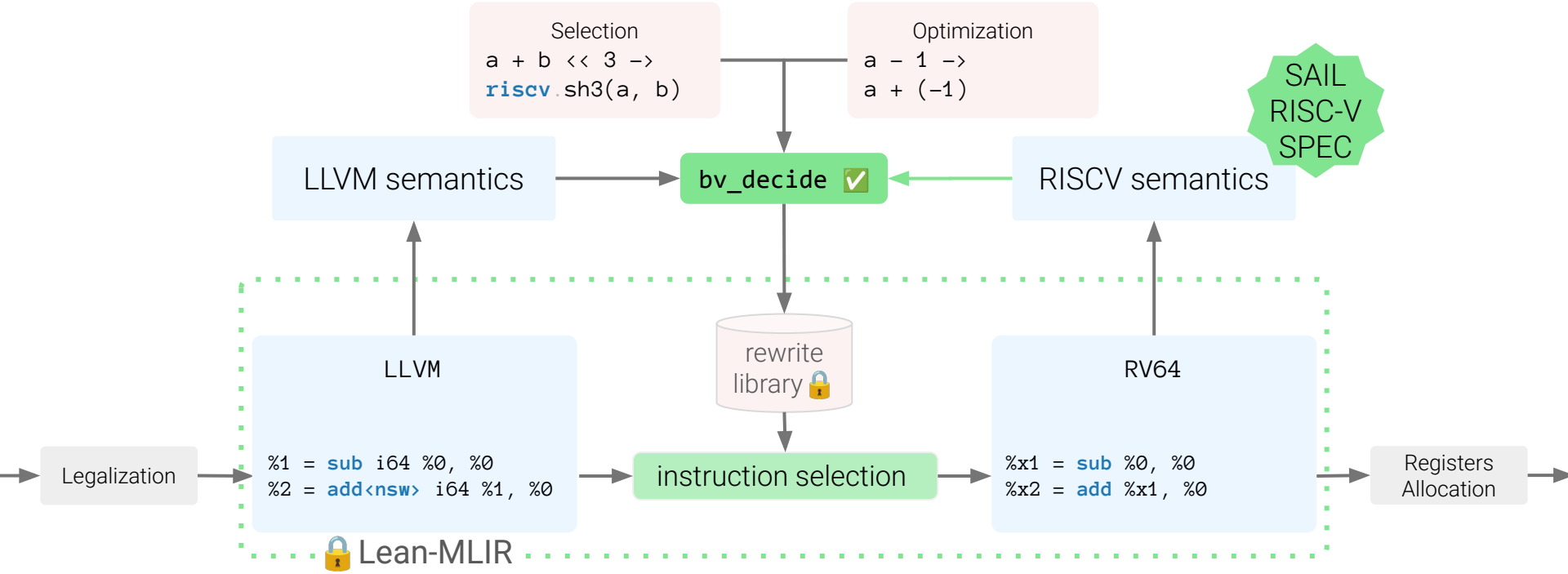
# Verifying LLVM's *instruction selection* and optimizations



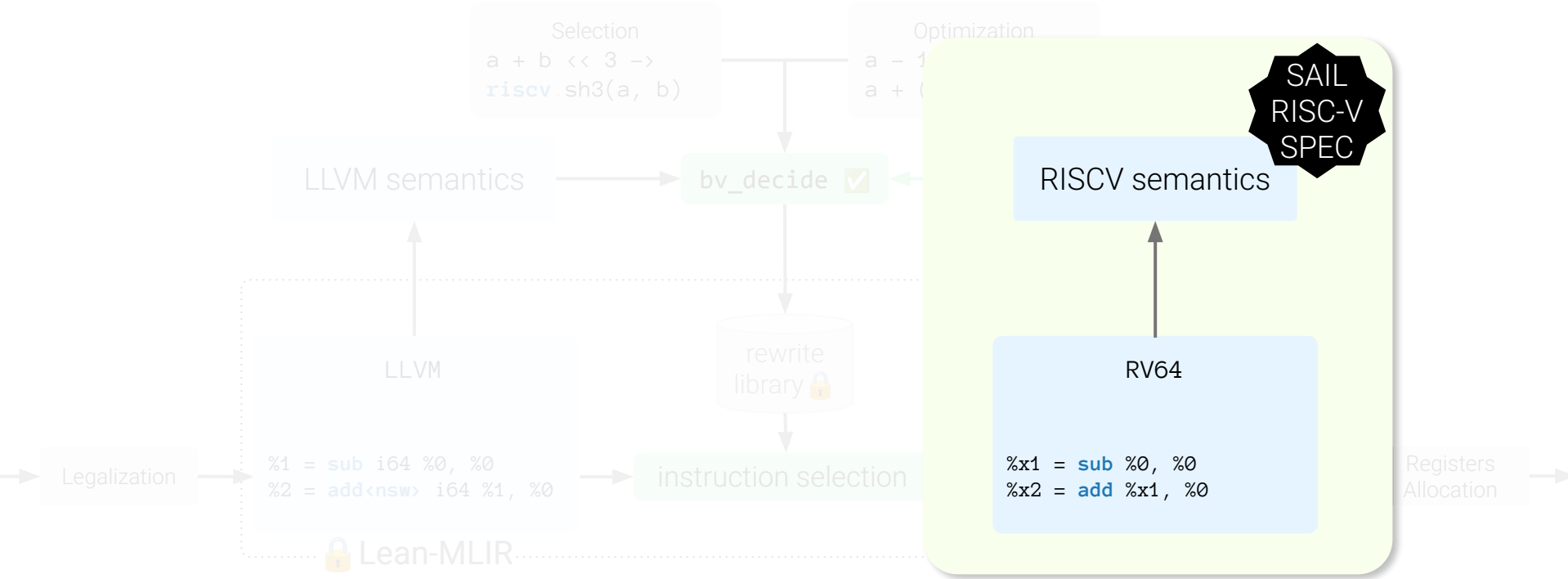
# Certified *assembly* output



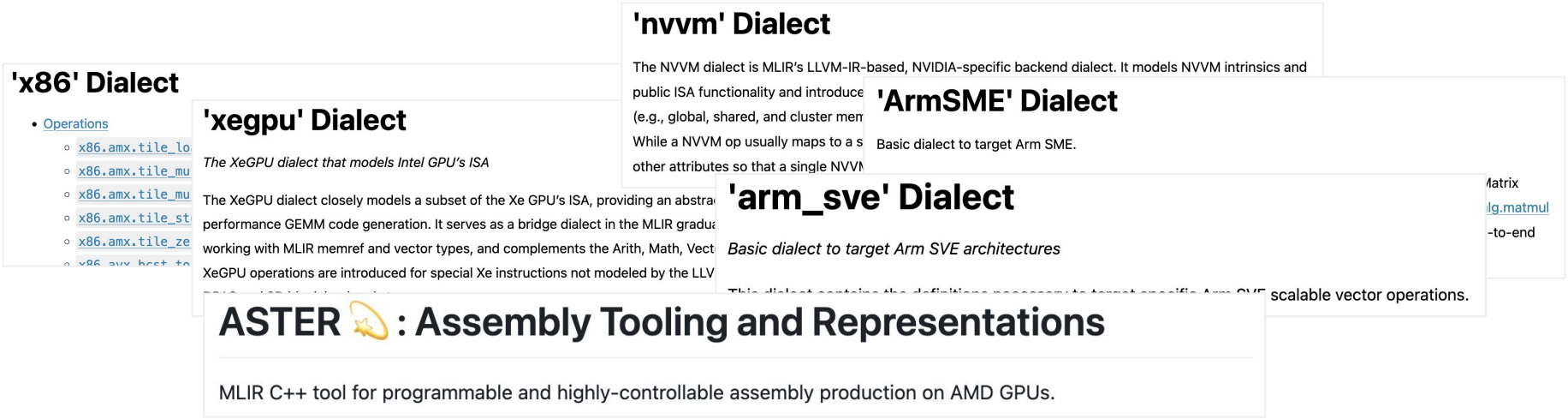
# Obtaining *automation* and *trust*



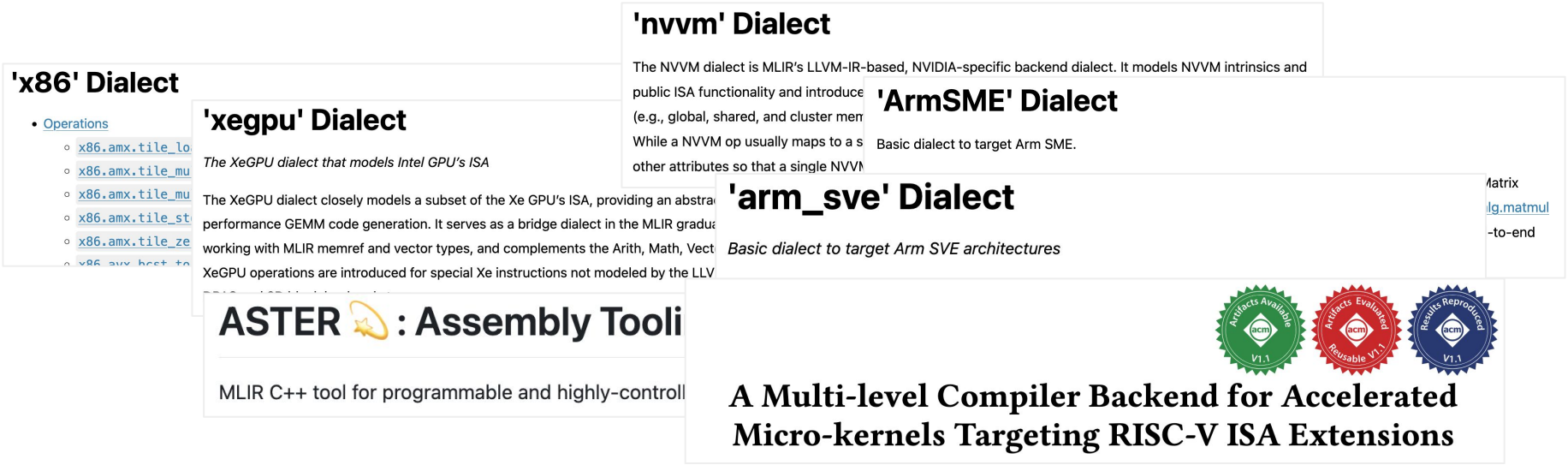
# *RISC-V* semantics at the core of our work



# Bringing ISA dialects in MLIR



# Bringing ISA dialects in MLIR



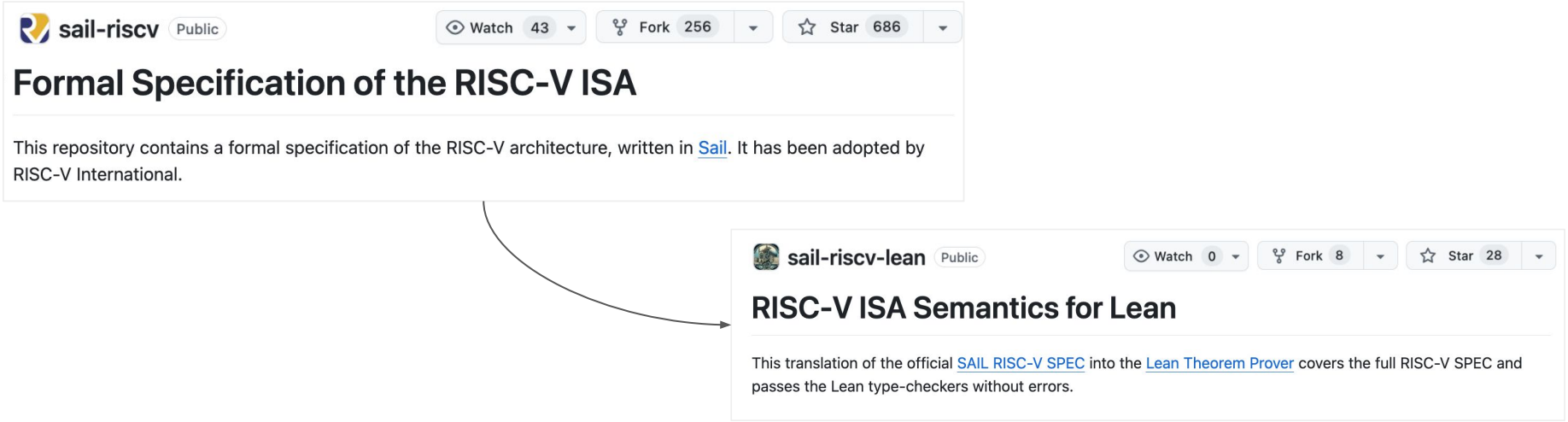



Lopoukhine, A., et al. "A multi-level compiler backend for accelerated micro-kernels targeting risc-v isa extensions." *CGO'25*

# *Authoritative* ISA semantics for our RISC-V dialect

**RISC-V ISA**

# *Authoritative* ISA semantics for our RISC-V dialect



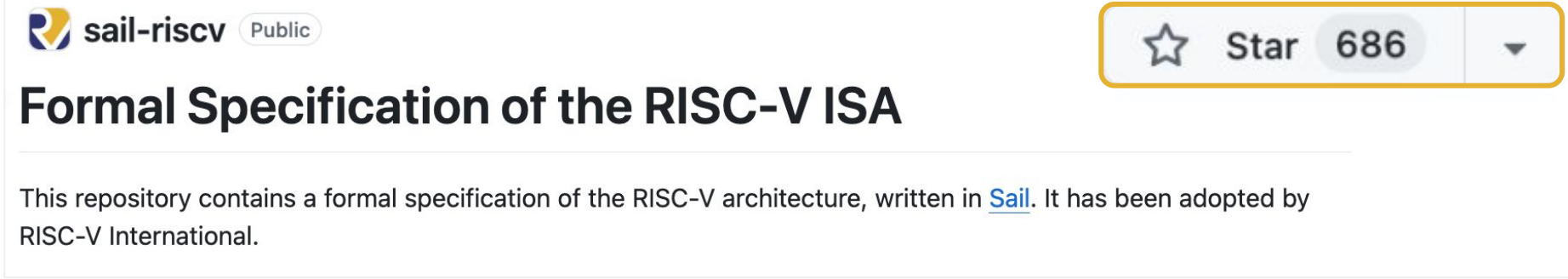
sail-riscv Public

Star 686

# Formal Specification of the RISC-V ISA

This repository contains a formal specification of the RISC-V architecture, written in Sail. It has been adopted by RISC-V International.

```
function clause execute REM (rs2, rs1, rd, is_unsigned) = {

  let rs1_bits = X(rs1);
  let rs2_bits = X(rs2);

  let rs1_int = if is_unsigned then unsigned(rs1_bits) else signed(rs1_bits);

  let rs2_int = if is_unsigned then unsigned(rs2_bits) else signed(rs2_bits);

  let remainder = if rs2_int == 0 then rs1_int else rem_round_zero(rs1_int, rs2_int);

  X(rd) = to_bits_truncate (remainder);

        RETIRE_SUCCESS
}
```




```
function clause execute REM (rs2, rs1, rd, is_unsigned) = {

  let rs1_bits =           X(rs1);
  let rs2_bits =           X(rs2);

  let rs1_int  = if is_unsigned          then unsigned(rs1_bits)         else signed(rs1_bits);

  let rs2_int  = if is_unsigned          then unsigned(rs2_bits)         else signed(rs2_bits);

  let remainder  = if rs2_int == 0            then rs1_int else rem_round_zero(rs1_int, rs2_int);

  X(rd) =      to_bits_truncate          (remainder);

        RETIRE_SUCCESS
}
```

sail-riscv-lean Public

Star 28

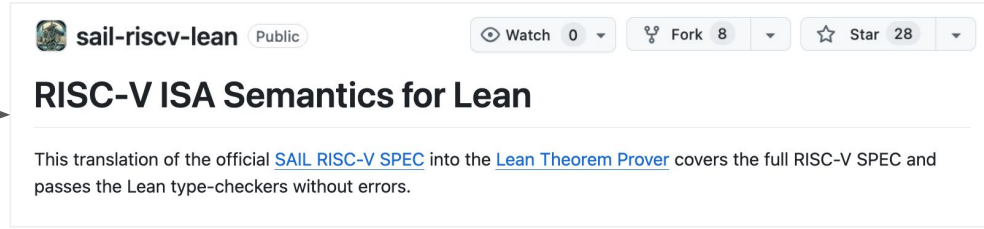
# RISC-V ISA Semantics for Lean

This translation of the official SAIL RISC-V SPEC into the Lean Theorem Prover covers the full RISC-V SPEC and passes the Lean type-checkers without errors.

```
def execute_REM (rs2 : regidx) (rs1 : regidx) (rd : regidx) (is_unsigned : Bool) :=

  let rs1_bits ← do (rX_bits rs1)
  let rs2_bits ← do (rX_bits rs2)

  let rs1_int := if (is_unsigned : Bool) then (BitVec.toNatInt rs1_bits) else (BitVec.toInt rs1_bits)

  let rs2_int := if (is_unsigned : Bool) then (BitVec.toNatInt rs2_bits) else (BitVec.toInt rs2_bits)

  let remainder := if ((rs2_int == 0) : Bool) then rs1_int else (Int.tmod rs1_int rs2_int)

  (wX_bits rd (to_bits_truncate (l := 64) remainder))

  (pure RETIRE_SUCCESS)
```

# *Authoritative* ISA semantics for our RISC-V dialect


```
function clause execute REM
  (rs2, rs1, rd, is_unsigned) = {
    let rs1_bits = X(rs1);
    let rs2_bits = X(rs2);
    let rs1_int = if is_unsigned then unsigned(rs1_bits)
                  else signed(rs1_bits);
    let rs2_int = if is_unsigned then unsigned(rs2_bits)
                  else signed(rs2_bits);
    let remainder = if rs2_int == 0 then rs1_int
                    else rem_round_zero(rs1_int, rs2_int);
    X(rd) = to_bits_truncate(remainder);
    RETIRE_SUCCESS
}
```




```
def execute_REM (rs2 : regidx) (rs1 : regidx) (rd : regidx) (is_unsigned : Bool) :=
  let rs1_bits ← do (rX_bits rs1)
  let rs2_bits ← do (rX_bits rs2)
  let rs1_int := if (is_unsigned : Bool) then (BitVec.toNatInt rs1_bits)
                 else (BitVec.toInt rs1_bits)
  let rs2_int := if (is_unsigned : Bool) then (BitVec.toNatInt rs2_bits)
                 else (BitVec.toInt rs2_bits)
  let remainder := if ((rs2_int == 0) : Bool) then rs1_int
                   else (Int.tmod rs1_int rs2_int)
  (wX_bits rd (to_bits_truncate (l := 64) remainder))
  (pure RETIRE_SUCCESS)
```

# Bitblastable, *authoritative* ISA semantics


```
def execute_REM (rs2 : regidx)
   (rs1 : regidx) (rd : regidx)
   (is_unsigned : Bool) :
      SailM ExecutionResult :=
  let rs1_bits ← do (rX_bits rs1)
  let rs2_bits ← do (rX_bits rs2)
  let rs1_int := if (is_unsigned : Bool)
      then (BitVec.toNatInt rs1_bits) else (BitVec.toInt rs1_bits)
  let rs2_int := if
      (is_unsigned : Bool)
      then (BitVec.toNatInt rs2_bits) else (BitVec.toInt rs2_bits)
  let remainder := if ((rs2_int == 0) : Bool) then rs1_int else (Int.tmod rs1_int rs2_int)
  (wX_bits rd (to_bits_truncate (l := 64) remainder)) (pure RETIRE_SUCCESS)
```

# Bitblastable, *authoritative* ISA semantics


```
def execute_REM (rs2 : regidx)
   (rs1 : regidx) (rd : regidx)
   (is_unsigned : Bool) :
      SailM ExecutionResult :=
  let rs1_bits ← do (rX_bits rs1)
  let rs2_bits ← do (rX_bits rs2)
  let rs1_int := if (is_unsigned : Bool)
      then (BitVec.toNatInt rs1_bits) else (BitVec.toInt rs1_bits)
  let rs2_int := if
      (is_unsigned : Bool)
      then (BitVec.toNatInt rs2_bits) else (BitVec.toInt rs2_bits)
  let remainder := if ((rs2_int == 0) : Bool) then rs1_int else (Int.tmod rs1_int rs2_int)
  (wX_bits rd (to_bits_truncate (l := 64) remainder)) (pure RETIRE_SUCCESS)
```

# Bitblastable, *authoritative* ISA semantics

SAIL model

```
def execute_REM (rs2 : regidx)
   (rs1 : regidx) (rd : regidx)
   (is_unsigned : Bool) :
      SailM ExecutionResult :=
  let rs1_bits ← do (rX_bits rs1)
  let rs2_bits ← do (rX_bits rs2)
  let rs1_int := if (is_unsigned : Bool)
      then (BitVec.toNatInt rs1_bits) else (BitVec.toInt rs1_bits)
  let rs2_int := if
      (is_unsigned : Bool)
      then (BitVec.toNatInt rs2_bits) else (BitVec.toInt rs2_bits)
  let remainder := if ((rs2_int == 0) : Bool) then rs1_int else (Int.tmod rs1_int rs2_int)
  (wX_bits rd (to_bits_truncate (l := 64) remainder)) (pure RETIRE_SUCCESS)
```

Bitblastable semantics

```
def rem (rs2_val : BitVec 64)
  (rs1_val : BitVec 64) : BitVec 64 :=
    rs1_val.srem rs2_val
```

# Bitblastable, *authoritative* ISA semantics

SAIL model

```
def execute_REM (rs2 : regidx)
   (rs1 : regidx) (rd : regidx)
   (is_unsigned : Bool) :
      SailM ExecutionResult :=
  let rs1_bits ← do (rX_bits rs1)
  let rs2_bits ← do (rX_bits rs2)
  let rs1_int := if (is_unsigned : Bool)
      then (BitVec.toNatInt rs1_bits) else (BitVec.toInt rs1_bits)
  let rs2_int := if
      (is_unsigned : Bool)
      then (BitVec.toNatInt rs2_bits) else (BitVec.toInt rs2_bits)
  let remainder := if ((rs2_int == 0) : Bool) then rs1_int else (Int.tmod rs1_int rs2_int)
  (wX_bits rd (to_bits_truncate (l := 64) remainder)) (pure RETIRE_SUCCESS)
```

Bitblastable semantics

```
def rem (rs2_val : BitVec 64)
  (rs1_val : BitVec 64) : BitVec 64 :=
    rs1_val.srem rs2_val
```



bv_decide ✅

# Bitblastable, *authoritative* ISA semantics

SAIL model

✅

Bitblastable semantics

```
def execute_REM (rs2 : regidx)
   (rs1 : regidx) (rd : regidx)
   (is_unsigned : Bool) :
      SailM ExecutionResult :=
  let rs1_bits ← do (rX_bits rs1)
  let rs2_bits ← do (rX_bits rs2)
  let rs1_int := if (is_unsigned : Bool)
      then (BitVec.toNatInt rs1_bits) else (BitVec.toInt rs1_bits)
  let rs2_int := if
      (is_unsigned : Bool)
      then (BitVec.toNatInt rs2_bits) else (BitVec.toInt rs2_bits)
  let remainder := if ((rs2_int == 0) : Bool) then rs1_int else (Int.tmod rs1_int rs2_int)
  (wX_bits rd (to_bits_truncate (l := 64) remainder)) (pure RETIRE_SUCCESS)
```

```
def rem (rs2_val : BitVec 64)
  (rs1_val : BitVec 64) : BitVec 64 :=
    rs1_val.srem rs2_val
```

↓

`bv_decide` ✅

UNIVERSITY OF CAMBRIDGE

# ISA: lowering LLVM IR's arithmetic fragment

| | | | | |
|---|---|---|---|---|
| lui | srai | sraw | srl | srai |
| auipc | slliw | slli | sra | sub |
| addi | srliw | slti | addw | and |
| andi | sraiw | sltiu | subw | or |
| ori | slt | slli | sllw | xor |
| xori | sltu | srli | srlw | sll |
| addiw | add | fence | load | store |

RISC-V **Base**

| | | | | |
|---|---|---|---|---|
| mul | mulw | mulh | mulhu | mulhsu |
| divuw | div | divu | rem | remu |
| remuw | divw | remw | | |

RISC-V **M**

| | | | | |
|---|---|---|---|---|
| adduw | sh1add | sh2add | sh3add | sh1adduw |
| sh3adduw | slliuw | rol | ror | rolw |
| sextb | sexth | zexth | rori | roriw |
| orn | xnor | max | maxu | min |
| bclr | bclri | bext | bexti | binvi |
| bseti | czeroeqz | czeronez | orcb | rev8 |
| clzw | ctz | ctzw | cpop | cpopw |
| sh2adduw | rorw | andn | minu | bset |
| clz | clmul | clmulh | clmulr | brev8 |
| pack | packw | zip | unzip | andn |
| xperm4 | xperm8 | | | |

RISC-V **B**

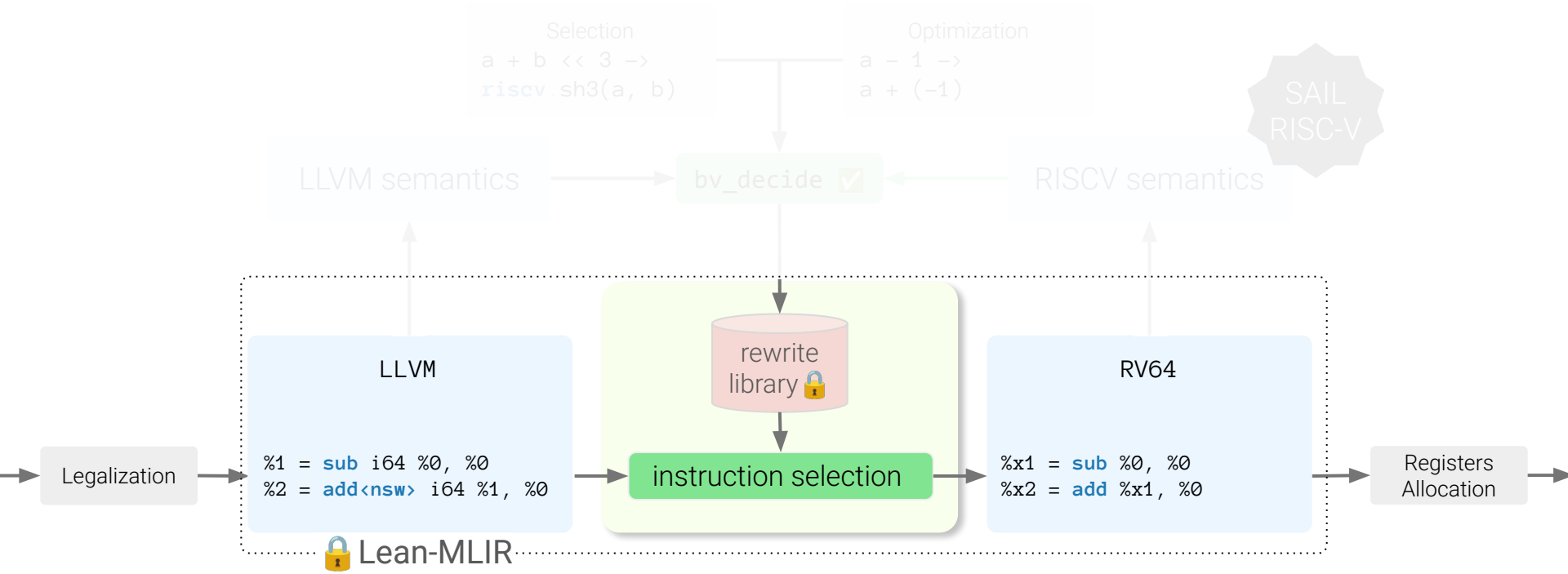
| | | | | |
|---|---|---|---|---|
| li | mv | not | neg | negw |
| zextb | seqz | snez | sltz | sgtz |
| sextw | | | | |

RISC-V **pseudo**

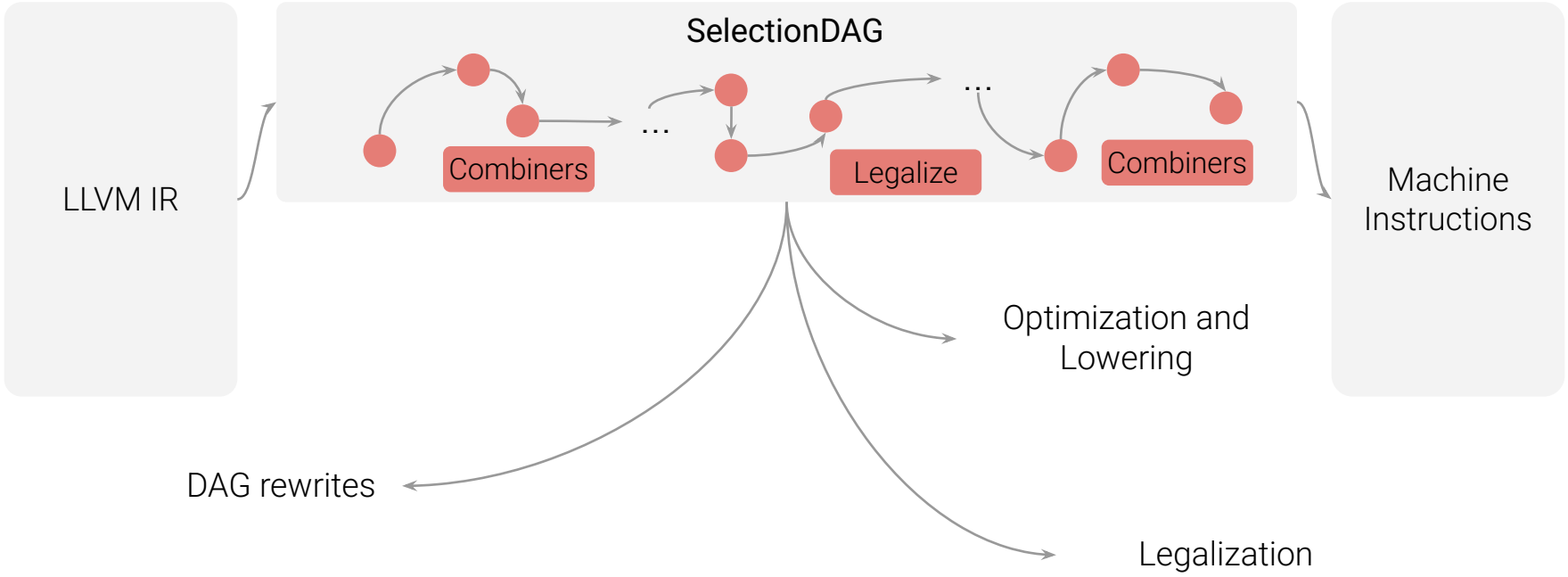


https://github.com/opencompl/riscv-lean

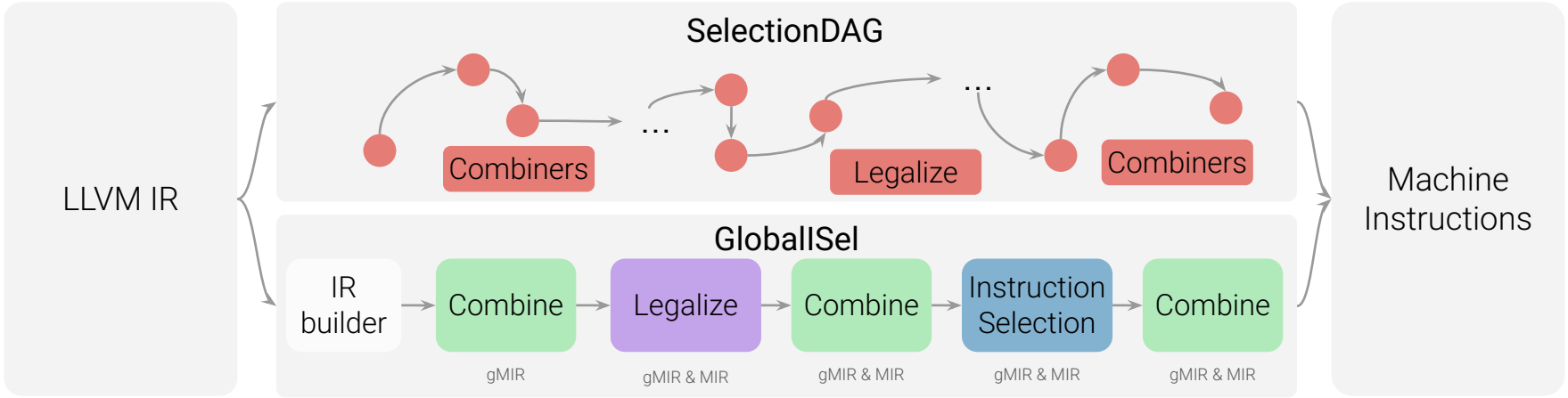
# A progressive *lowering* that mirrors LLVM



# LLVM's *instruction selectors*

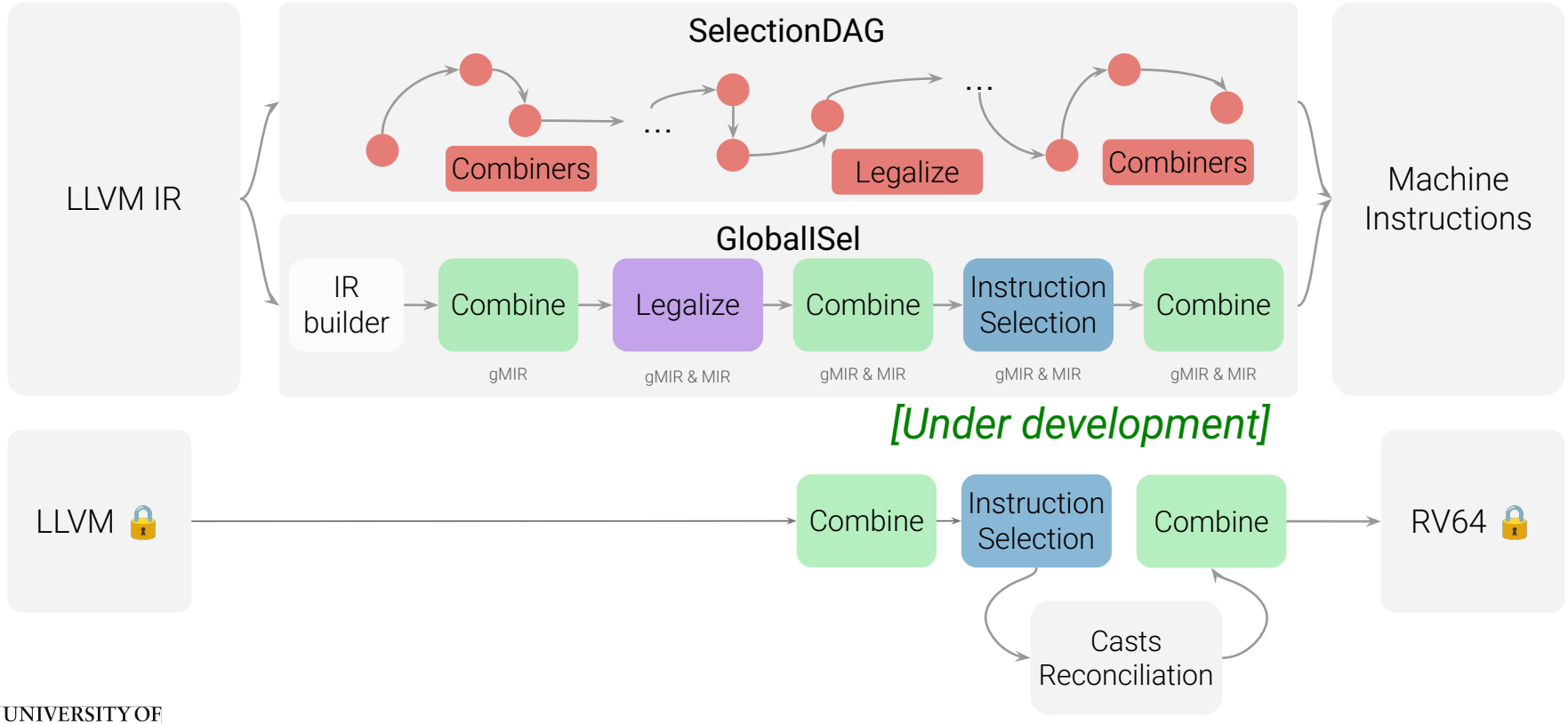


# LLVM's *instruction selectors*

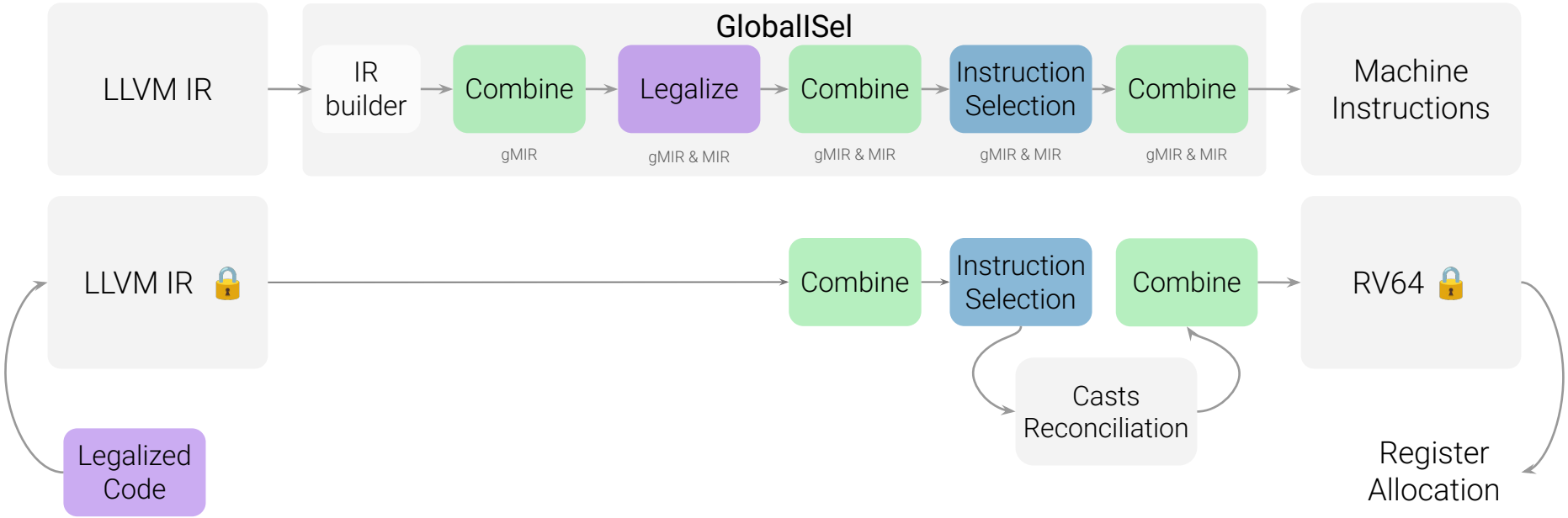



*[Under development]*

# LLVM's *instruction selectors* and our Lean implementation



# *GlobalISel* vs. our Lean implementation



# *LLVM* has poison semantics


LLVM 🔒

```
def and_llvm := [LV| {
  ^entry (%lhs: i1, %rhs: i1 ):
  %1 = llvm.and %lhs, %rhs : i1
  llvm.return %1 : i1
  }]
```

Poison or concrete

# *RISC-V* semantics are concrete


LLVM 🔒

```
def and_llvm := [LV| {
  ^entry (%lhs: i1, %rhs: i1 ):
  %1 = llvm.and %lhs, %rhs : i1
  llvm.return %1 : i1
  }]
```

RV64 🔒

```
def and_riscv := [LV| {
  ^entry (%lhs: i1, %rhs: i1):
  %lhsr = cast (%lhs) : (i1) -> (reg)
  %rhsr = cast (%rhs) : (i1) -> (reg)
  %0 = RV64.and %lhsr, %rhsr : reg
  %1 = cast (%0) : (reg) -> (i1)
  llvm.return %1 : i1
  }]
```

# We need to prove *refinement*


LLVM 🔒

```
def and_llvm := [LV| {
  ^entry (%lhs: i1, %rhs: i1 ):
  %1 = llvm.and %lhs, %rhs : i1
  llvm.return %1 : i1
  }]
```

RV64 🔒

```
def and_riscv := [LV| {
  ^entry (%lhs: i1, %rhs: i1):
  %lhsr = cast (%lhs) : (i1) -> (reg)
  %rhsr = cast (%rhs) : (i1) -> (reg)
  %0 = RV64.and %lhsr, %rhsr : reg
  %1 = cast (%0) : (reg) -> (i1)
  llvm.return %1 : i1
  }]
```

```
def llvm_and_lower_riscv : LLVMRewriteRefine 1 [Ty.llvm (.bv 1), Ty.llvm (.bv 1)] where
  lhs:= and_llvm
  rhs:= and_riscv
  correct := by simp_lowering <;>; bv_decide
```

# *Refinement* verification


```
∀ (x y : ⟦Ty.llvm (Mty.bv 1)⟧),
    LLVM.and x y ⊑ castriscvToLLVM (RV64.and (castLLVMToriscv y) (castLLVMToriscv x))
```

```
∀ (x y : ⟦Ty.llvm (MTy.bv 1)⟧), (do let x' ← x; let y' ← y
  PoisonOr.value (x' &&& y')) ⊑
  PoisonOr.value (BitVec.signExtend 1
     (BitVec.signExtend 64 (x.toOption.getD 0#1) &&&
      BitVec.signExtend 64 (y.toOption.getD 0#1)))
```

```
∀ (x y : BitVec 1),
  x &&& y = BitVec.signExtend 1 (BitVec.signExtend 64 x &&& BitVec.signExtend 64 y)
```

→ **bv_decide** ✅

# *158* lowering patterns


```
^bb0(%x : i64, %y : i64):
 %3 = llvm.add %x, %y
      <{overflow<none>}> : i64
llvm.return(%3) : i64
```



```
^bb0(%x : i64, %y : i64):
 %0 = cast(%reg1) : i64 -> reg
 %1 = cast(%reg2) : i64 -> reg
 %2 = RV64.add %0, %1 : reg
 %3 = cast(%2) : reg -> i64
llvm.return(%3) : i64
```

# With *pseudoinstructions*


```
^bb0(%x : i64, %y : i64):
 %3 = llvm.add %x, %y
      <{overflow<none>}> : i64
llvm.return(%3) : i64
```

→

```
^bb0(%x : i64, %y : i64):
 %0 = cast(%reg1) : i64 -> reg
 %1 = cast(%reg2) : i64 -> reg
 %2 = RV64.add %0, %1 : reg
 %3 = cast(%2) : reg -> i64
llvm.return(%3) : i64
```

```
^bb0(%x : i64, %y : i64):
 %1 = llvm.icmp.eq %x, %y : i1
llvm.return(%1) : i1
```

→

```
xor a0, a0, a1
seqz a0, a0
ret
```

# With *pseudoinstructions*


```
^bb0(%x : i64, %y : i64):
 %3 = llvm.add %x, %y
      <{overflow<none>}> : i64
llvm.return(%3) : i64
```

```
^bb0(%x : i64, %y : i64):
 %0 = cast(%reg1) : i64 -> reg
 %1 = cast(%reg2) : i64 -> reg
 %2 = RV64.add %0, %1 : reg
 %3 = cast(%2) : reg -> i64
llvm.return(%3) : i64
```

```
^bb0(%x : i64, %y : i64):
 %1 = llvm.icmp.eq %x, %y : i1
llvm.return(%1) : i1
```

```
xor a0, a0, a1
seqz a0, a0
ret
```

```
def seqz_pseudo (rs1_val : BitVec 64) : BitVec 64 :=
  RV64.sltiu 1 rs1_val
```

# With *pseudoinstructions*

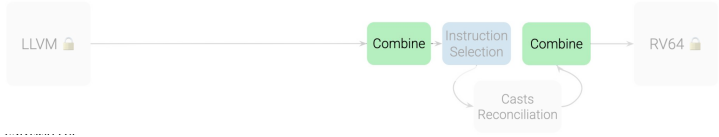

```
^bb0(%x : i64, %y : i64):
 %3 = llvm.add %x, %y
      <{overflow<none>}> : i64
llvm.return(%3) : i64
```

```
^bb0(%x : i64, %y : i64):
 %0 = cast(%reg1) : i64 -> reg
 %1 = cast(%reg2) : i64 -> reg
 %2 = RV64.add %0, %1 : reg
 %3 = cast(%2) : reg -> i64
llvm.return(%3) : i64
```

```
^bb0(%x : i64, %y : i64):
 %1 = llvm.icmp.eq %x, %y : i1
llvm.return(%1) : i1
```

```
^bb0(%x : i64, %y : i64):
 %0 = cast(%x) : i64 -> reg
 %1 = cast(%y) : i64 -> reg
 %2 = RV64.xor %0, %1 : reg
 %3 = RV64.sltiu %0, 1 : reg
 %4 = cast(%3) : reg -> i64
llvm.return(%4) : i64
```

# Verifying *optimizations*



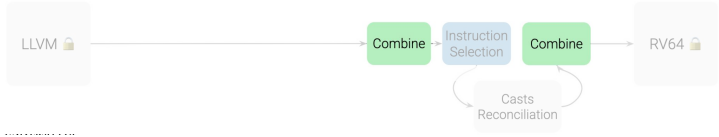

```
// Fold (x * 0) -> 0
def binop_right_to_zero: GICombineRule<
  (defs root:$dst),
  (match (G_MUL $dst, $lhs, 0:$zero)),
  (apply (GIReplaceReg $dst, $zero))
>;
```

# Verifying *optimizations*


```
// Fold (x * 0) -> 0
def binop_right_to_zero: GICombineRule<
  (defs root:$dst),
  (match (G_MUL $dst, $lhs, 0:$zero)),
  (apply (GIReplaceReg $dst, $zero))
>;
```

```
def binop_right_to_zero :
     LLVMRewriteRefine 64 [Ty.llvm (.bv 64)] where
  lhs := [LV| {
    ^entry (%0: i64):
      %1 = llvm.mlir.constant (0) : i64
      %2 = llvm.mul %0, %1 : i64
      llvm.return %2 : i64
  }]
  rhs := [LV| {
    ^entry (%0: i64):
      %1 = llvm.mlir.constant (0) : i64
      llvm.return %1 : i64
  }]
```

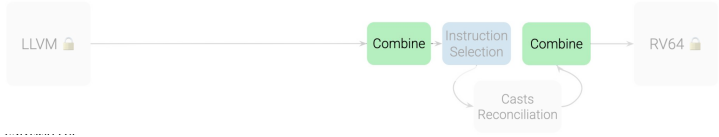
# Verifying *optimizations*




```
// Fold (x * 0) -> 0
def binop_right_to_zero: GICombineRule<
  (defs root:$dst),
  (match (G_MUL $dst, $lhs, 0:$zero)),
  (apply (GIReplaceReg $dst, $zero))
>;
```

```
def binop_right_to_zero :
     LLVMRewriteRefine 64 [Ty.llvm (.bv 64)] where
  lhs := [LV| {
    ^entry (%0: i64):
      %1 = llvm.mlir.constant (0) : i64
      %2 = llvm.mul %0, %1 : i64
      llvm.return %2 : i64
  }]
  rhs := [LV| {
    ^entry (%0: i64):
      %1 = llvm.mlir.constant (0) : i64
      llvm.return %1 : i64
  }]
```

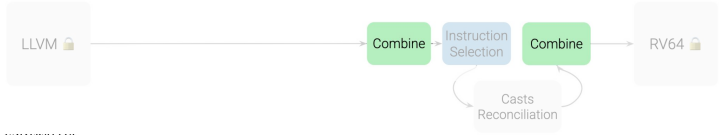
# Verifying *optimizations*




```
// Fold (x * 0) -> 0
def binop_right_to_zero: GICombineRule<
  (defs root:$dst),
  (match (G_MUL $dst, $lhs, 0:$zero)),
  (apply (GIReplaceReg $dst, $zero))
>;
```

```
def binop_right_to_zero :
     LLVMRewriteRefine 64 [Ty.llvm (.bv 64)] where
  lhs := [LV| {
    ^entry (%0: i64):
      %1 = llvm.mlir.constant (0) : i64
      %2 = llvm.mul %0, %1 : i64
      llvm.return %2 : i64
  }]
  rhs := [LV| {
    ^entry (%0: i64):
      %1 = llvm.mlir.constant (0) : i64
      llvm.return %1 : i64
  }]
  correct := by simp_lowering <;>; bv_decide
```

# Extracting *optimizations*

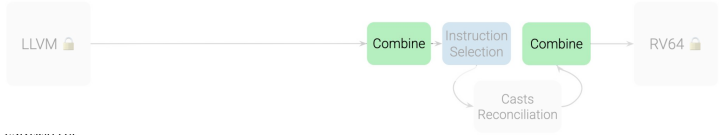



```
// Fold (x * 0) -> 0
def binop_right_to_zero: GICombineRule<
  (defs root:$dst),
  (match (G_MUL $dst, $lhs, 0:$zero)),
  (apply (GIReplaceReg $dst, $zero))
>;
```

```
def binop_right_to_zero :
     LLVMRewriteRefine 64 [Ty.llvm (.bv 64)] where
  lhs := [LV| {
    ^entry (%0: i64):
      %1 = llvm.mlir.constant (0) : i64
      %2 = llvm.mul %0, %1 : i64
      llvm.return %2 : i64
  }]
  rhs := [LV| {
    ^entry (%0: i64):
      %1 = llvm.mlir.constant (0) : i64
      llvm.return %1 : i64
  }]
  correct := by simp_lowering <;>; bv_decide
```

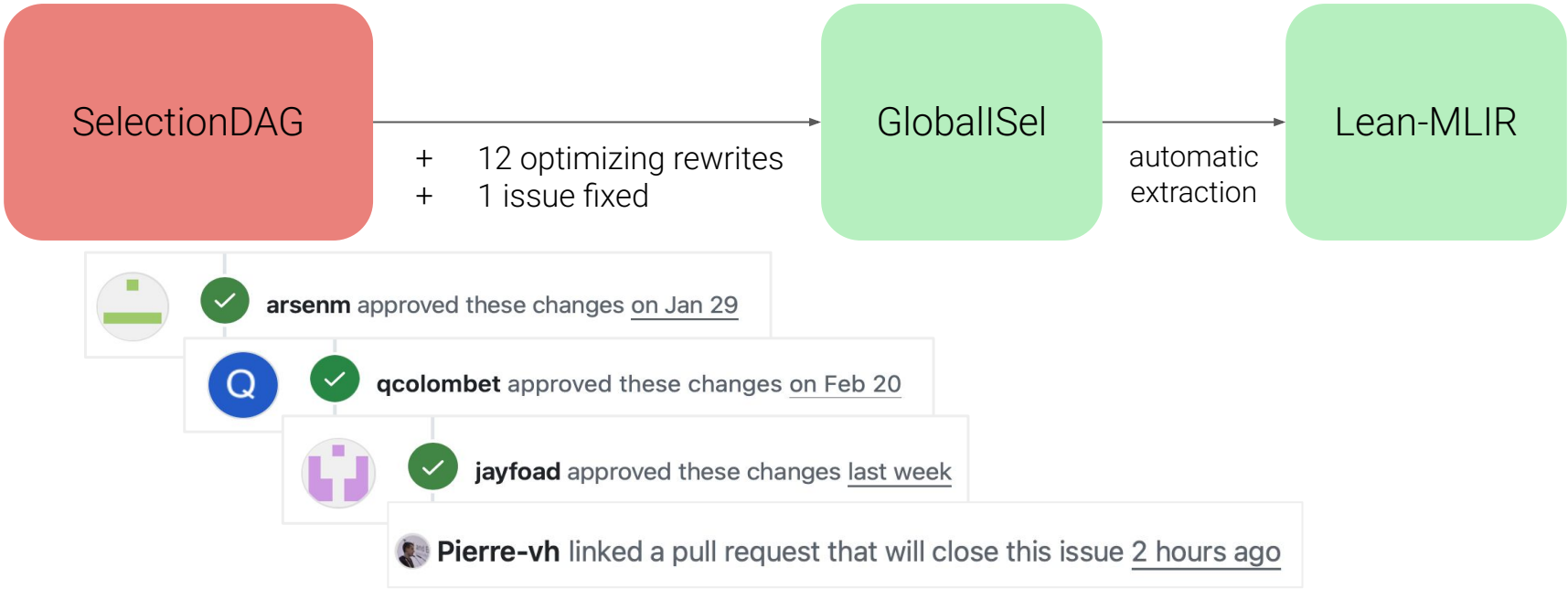
automatic extraction of MIR patterns

# Porting LLVM *optimizations*




| Group | O0, PreLeg | PreLeg. | PostLeg. | #Rules | #Rules In Frag. | #Mechanized |
|---|---|---|---|---|---|---|
| `trivial` | ✅ | ✅ | | 6 | 4 | 4 |
| `optnone` | ✅ | | | 12 | 5 | 5 |
| `identity` | | ✅ | ✅ | 20 | 8 | 8 |
| `cmp` | | ✅ | | 7 | 4 | 4 |
| `integer_reassoc` | | ✅ | | 13 | 13 | 13 |
| `cast` | | ✅ | | 11 | 9 | 9 |
| `cast_of_cast` | | ✅ | | 11 | 10 | 10 |
| others | | ✅ | ✅ | 148 | 27 | 13 |
| Total | | | | 228 | 80 | 66 |
| Mechanized [%] | | | | | | **82.5%** |

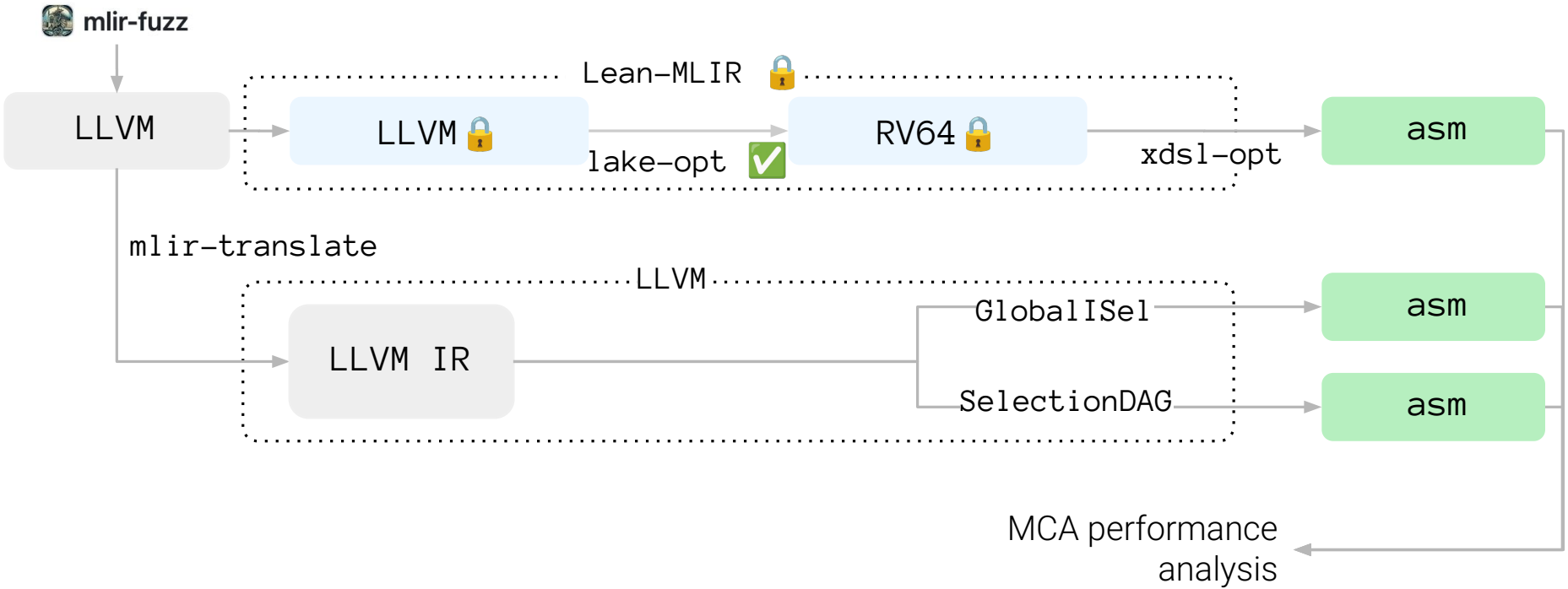
# Improving *GlobalISel* improves verified instruction selection



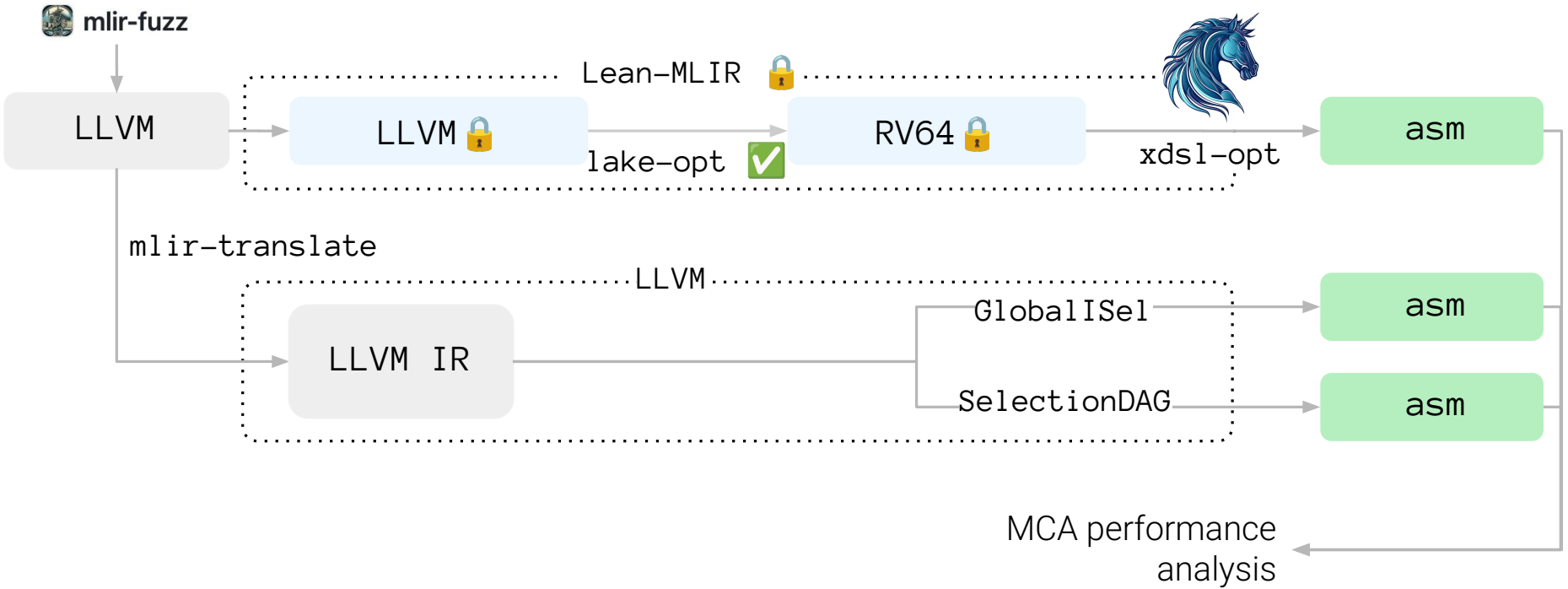
# Improving *GlobalISel* improves verified instruction selection

Select

# Comparing with LLVM's *instruction selection*

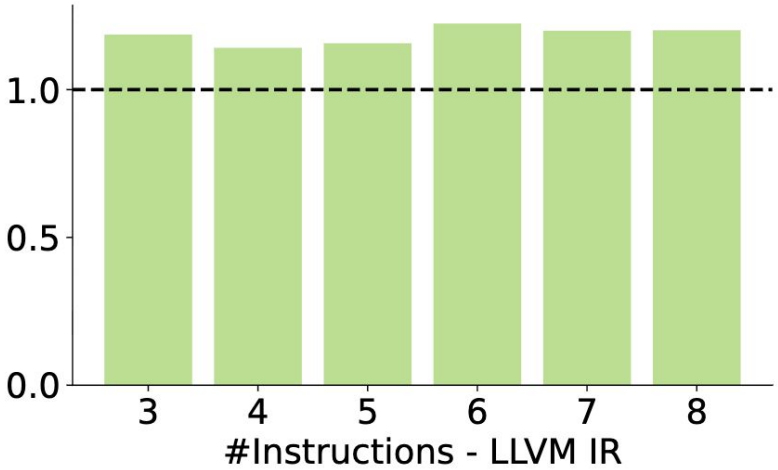


# Comparing with LLVM's *instruction selection*

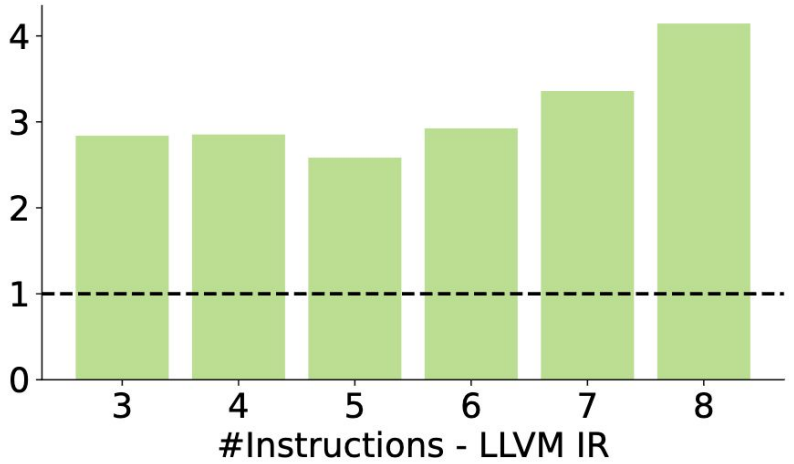


# MCA Estimate: #Cycles

$$\frac{\text{LeanMLIR}}{\text{GlobalISel}}$$

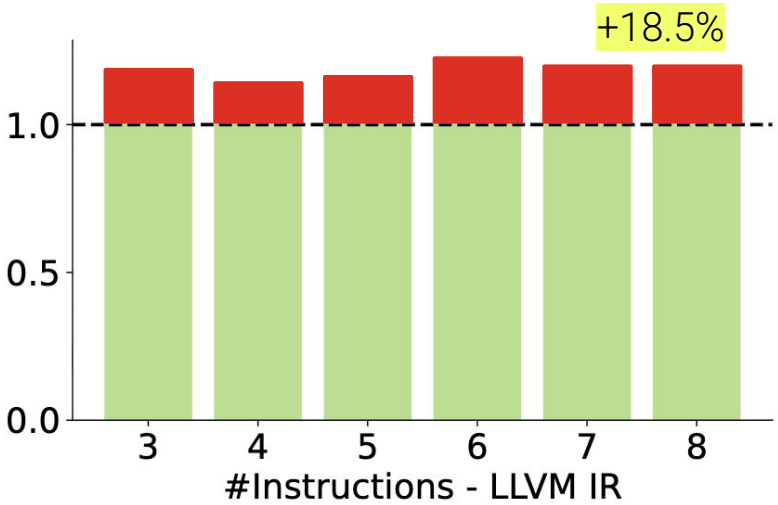



$$\frac{\text{LeanMLIR}}{\text{SelDAG}}$$

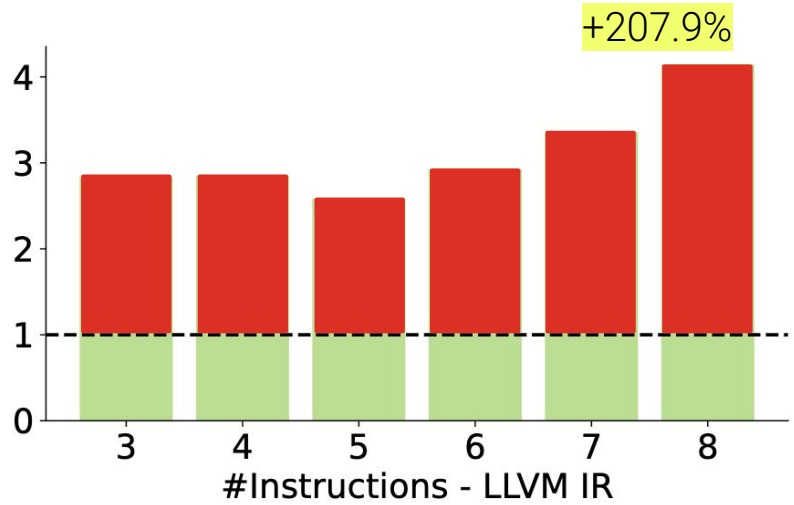


# MCA Estimate: #Cycles

LeanMLIR / GlobalISel

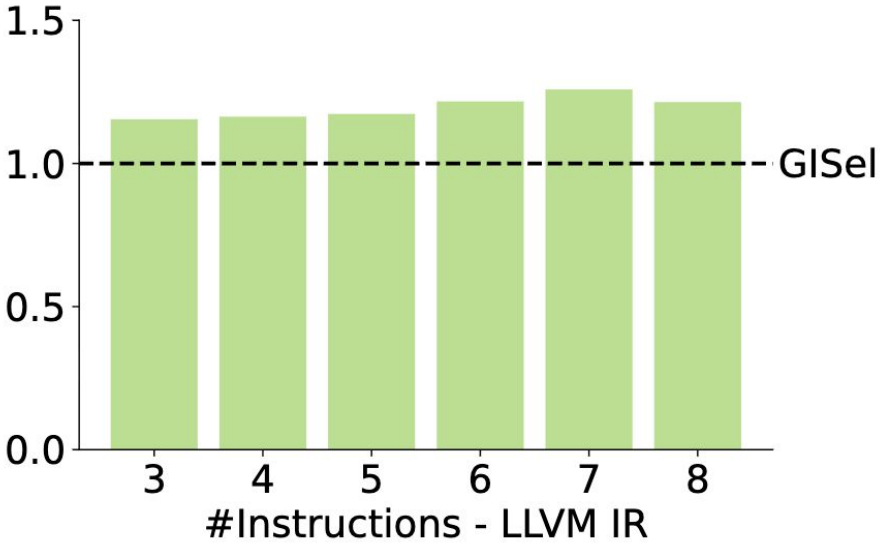



LeanMLIR / SelDAG

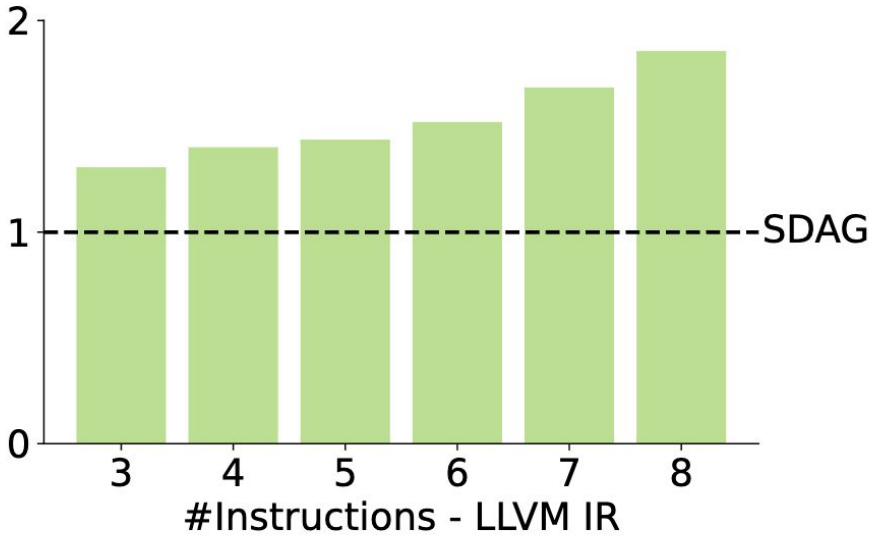


# MCA Estimate: #Instructions

LeanMLIR / GlobalISel

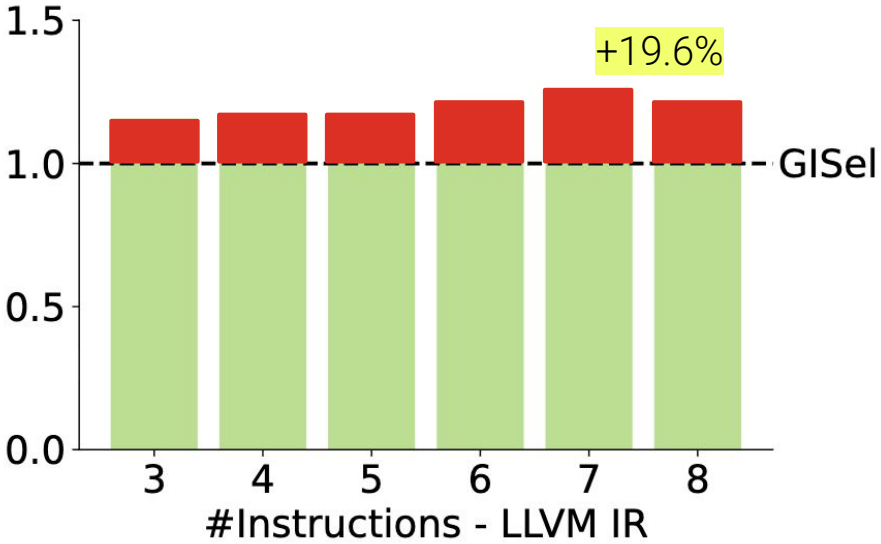



LeanMLIR / SelDAG

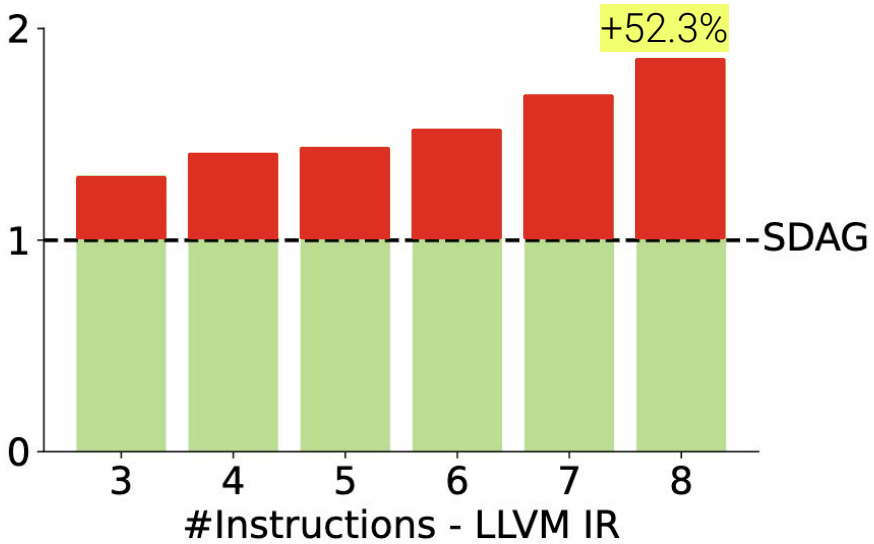


# MCA Estimate: #Instructions

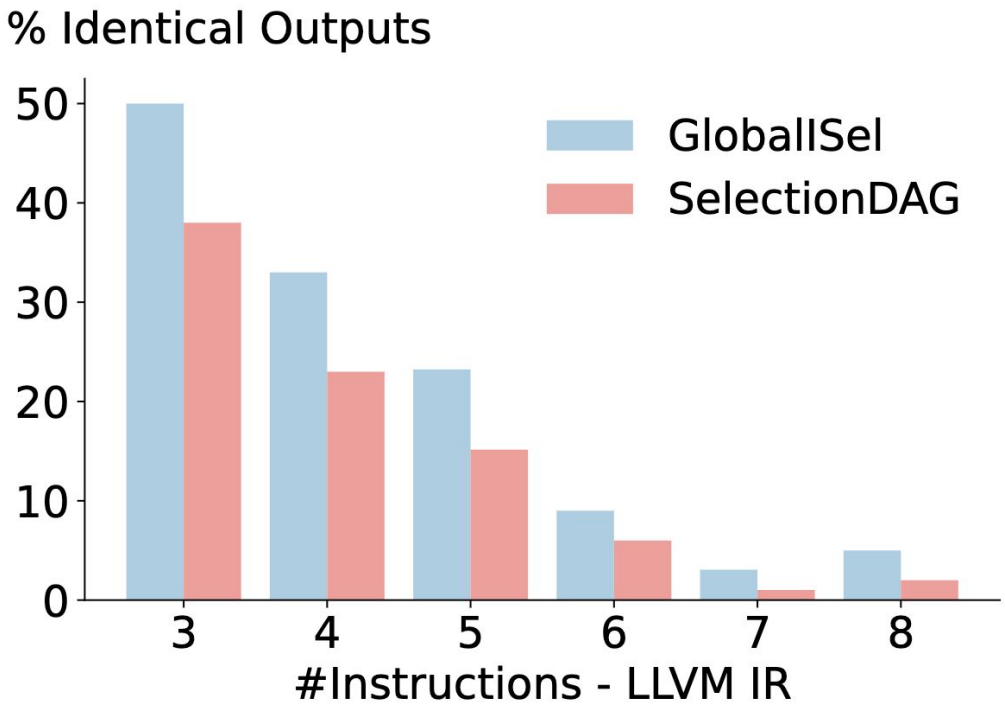
LeanMLIR / GlobalISel



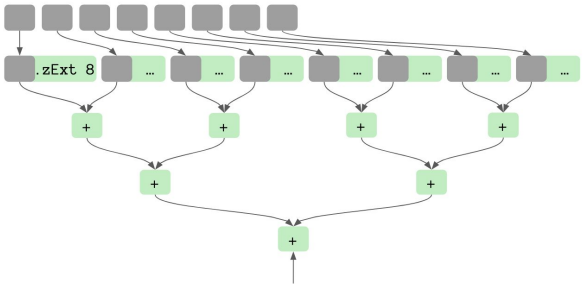

LeanMLIR / SelDAG

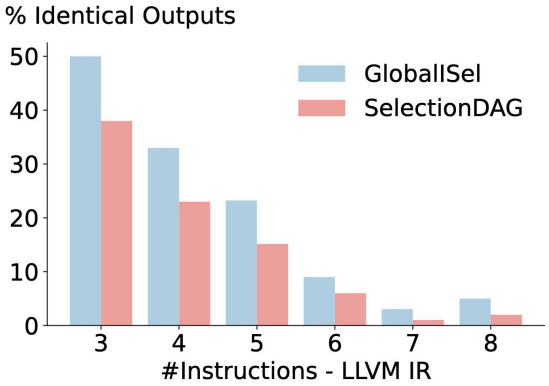


# The same `.asm` output!



# What's next




**Verified Intermediate Representation**

A verified implementation of the MLIR SSA-based datastructures.

https://github.com/opencompl/veir/

Bitblastable
Vector Semantics

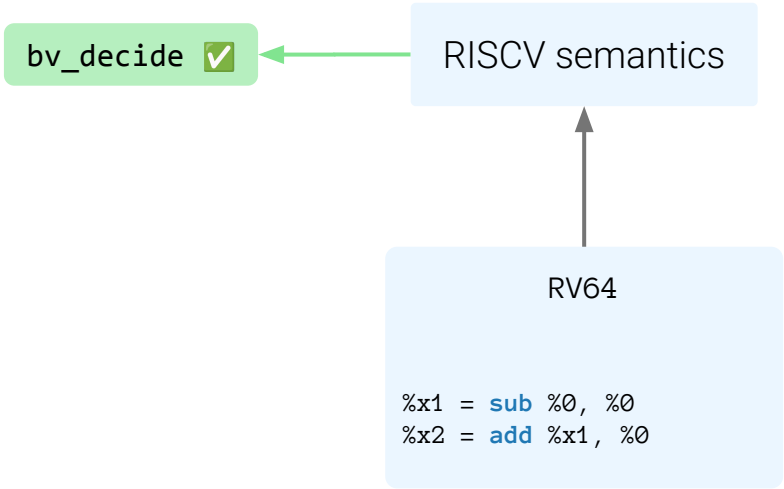
Close the gap
SelectionDAG → GlobalISel

Realistic use case
and performance

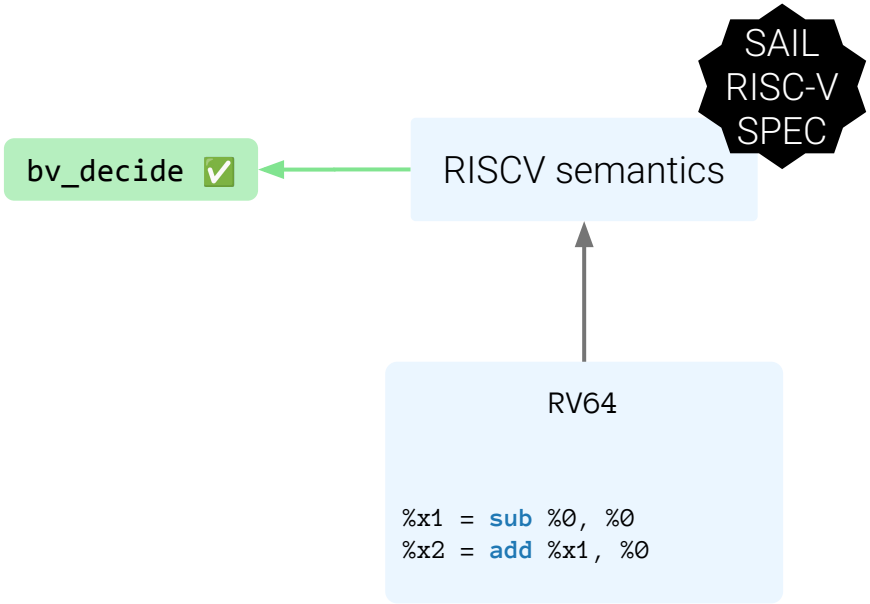
# *Questions?*

`bv_decide` ✅

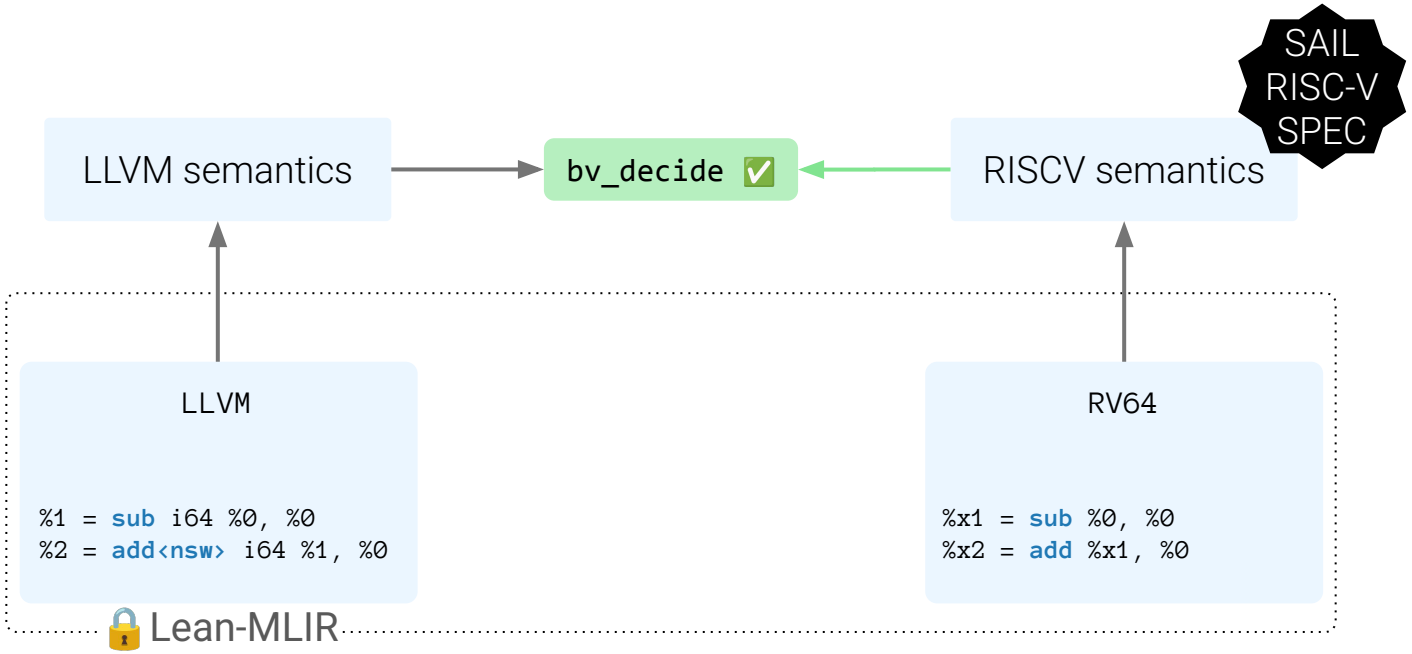
# *Questions?*



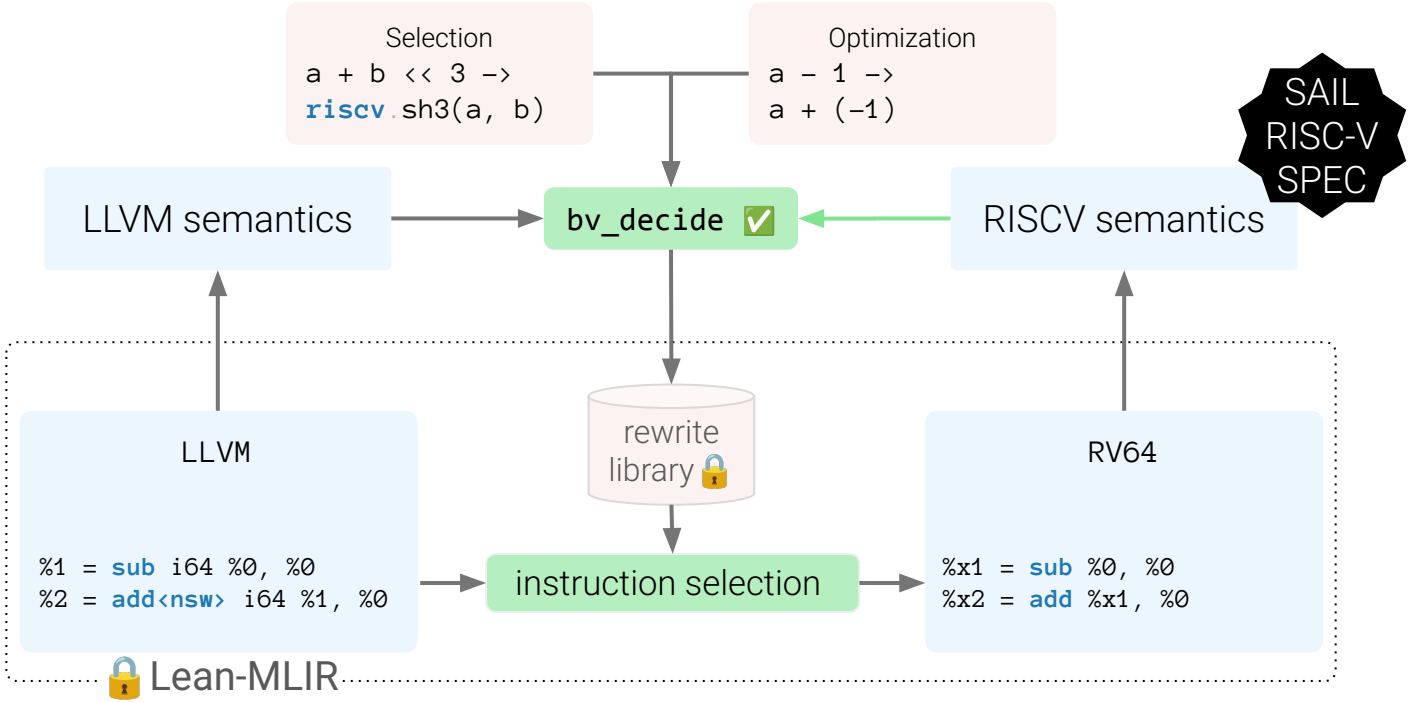
# *Questions?*


SAIL
RISC-V
SPEC
bv_decide ✅
RISCV semantics
RV64
%x1 = sub %0, %0
%x2 = add %x1, %0

# *Questions?*


SAIL
RISC-V
SPEC
LLVM semantics
bv_decide ✅
RISCV semantics
LLVM
%1 = sub i64 %0, %0
%2 = add<nsw> i64 %1, %0
RV64
%x1 = sub %0, %0
%x2 = add %x1, %0
🔒Lean-MLIR

# Questions?


Selection
a + b << 3 ->
riscv.sh3(a, b)
Optimization
a - 1 ->
a + (-1)
SAIL
RISC-V
SPEC
LLVM semantics
bv_decide ✅
RISCV semantics
LLVM
%1 = sub i64 %0, %0
%2 = add<nsw> i64 %1, %0
rewrite
library 🔒
instruction selection
RV64
%x1 = sub %0, %0
%x2 = add %x1, %0
🔒 Lean-MLIR