Mechanized Finite Domain Solvers

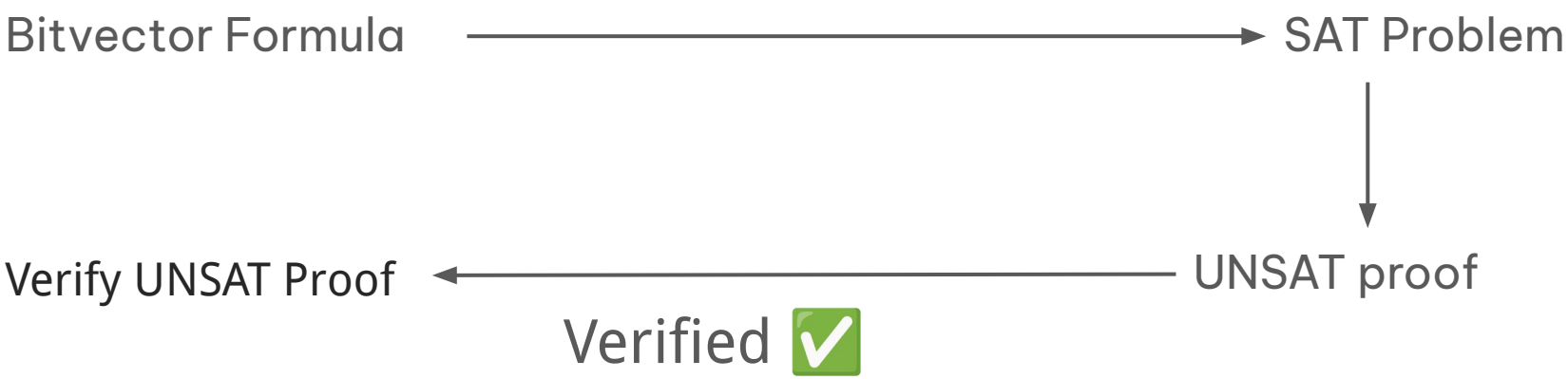
- **Arbitrary-width Bitvector Predicates via Model Checking**:
  `∀ (a b : BitVec w) : (a &&& b = 0#w) → ((a + b) = (a ||| b))`

- **Floating Point Bitblasting**: `∀ (a b c : PackedFloat 5 10) (m : RoundingMode) (hc : c.isNormOrSubnorm = true) : a ≤ b → (add a c m) ≤ (add b c m)`

- **Bitvector Generalization by Inductive Synthesis**:
  `#generalize (x <<< 3) <<< 4 = x <<< 7`
  `((x << y) << z) = (x << (y + z))`

- **Thinking about: Cylindrical Algebraic Decomposition for BV?**

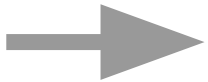
Verified Fixed Width Bitvector Solver: `bv_decide`

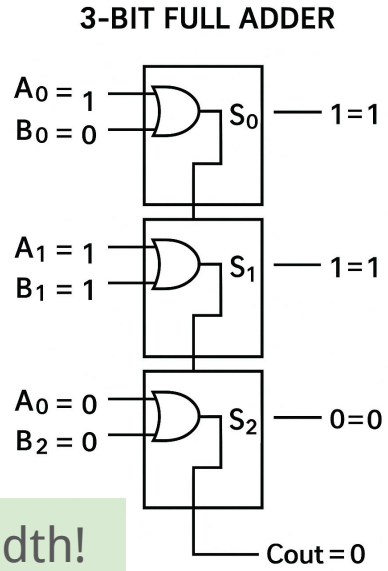


Verified Fixed Width Bitvector Solver: `bv_decide`

Bitvector Formula —— How To Verify? 🤔 ——▶ SAT Problem

```
011b + 100b  (bits / booleans ✅)
= 3 + 4
= 7  (natural numbers ❌)
= 111b
```





In General, We Must Prove This Correct For Arbitrary Width!

`bv_automata`: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
... x5 x4 x3 x2 x1 x0 : x
... x5 x4 x3 x2 x1 x0 : x

: x&x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
... x5 x4 x3 x2 x1 x0 : x
... x5 x4 x3 x2 x1 x0 : x
x0&x0 : x&x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
... x5 x4 x3 x2 x1 x0 : x
... x5 x4 x3 x2 x1 x0 : x
x0&x0 : x&x
1 : x&x=x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
... x5 x4 x3 x2 x1 x0 : x
... x5 x4 x3 x2 x1 x0 : x
                x0&x0 : x&x
                    1 : x&x=x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
...   x5   x4   x3   x2   x1   x0 :    x
...   x5   x4   x3   x2   x1   x0 :    x
                        x1&x1 x0&x0 :  x&x
                                    1 : x&x=x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
...   x5   x4   x3   x2   x1   x0 :    x
...   x5   x4   x3   x2   x1   x0 :    x
                        x1&x1 x0&x0 :  x&x
                           1     1 : x&x=x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
...   x5   x4   x3   x2   x1   x0 :    x
...   x5   x4   x3   x2   x1   x0 :    x
                    x2&x2 x1&x1 x0&x0 :  x&x
                      1     1     1 : x&x=x
```

`bv_automata`: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
...   x5   x4   x3   x2   x1   x0 :    x
...   x5   x4   x3   x2   x1   x0 :    x

                    ....x2&x2 x1&x1 x0&x0 :  x&x
.....1.....1.....1.....1.....1     1    1 : x&x=x
```

`bv_automata`: Width-Quantified Problems

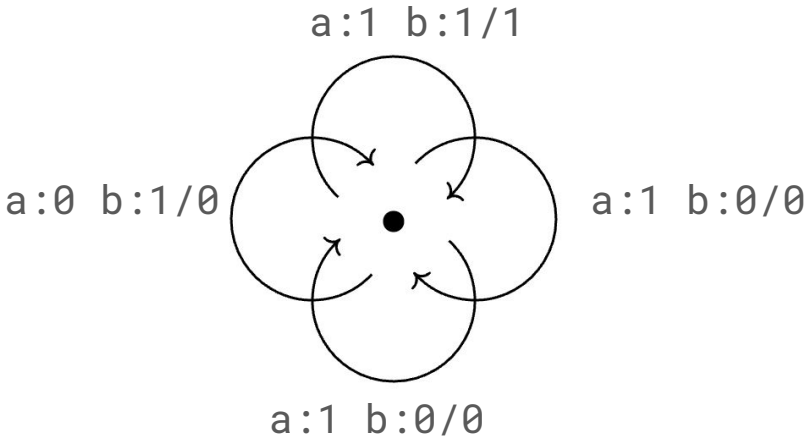
```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
...   x5   x4   x3   x2   x1   x0 :    x
...   x5   x4   x3   x2   x1   x0 :    x
                  ...x2&x2 x1&x1 x0&x0 :  x&x
.....1.....1.....1.....1.....1     1    1 : x&x=x
```

Does **x&x=x** produce an infinite sequence of 1s?

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



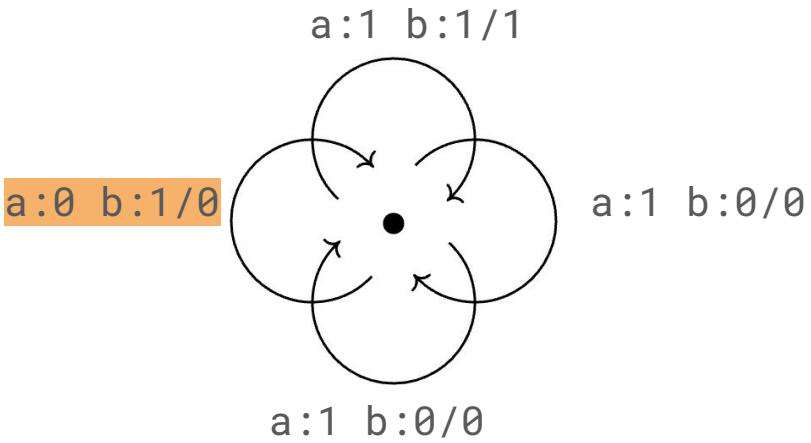
: **a**

: **b**

: **a&b**

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



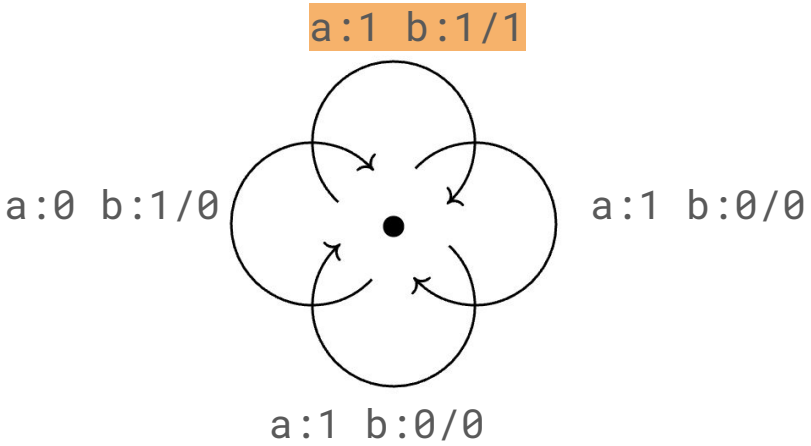
0 : **a**

1 : **b**

0 : **a&b**

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



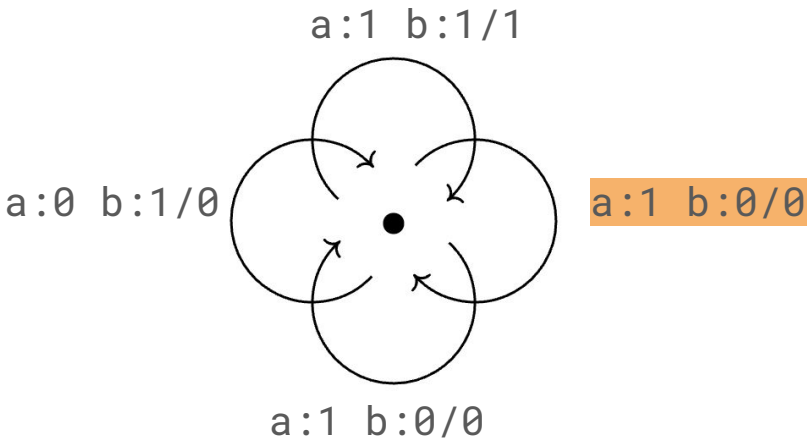
1 0 : a

1 1 : b

1 0 : **a&b**

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



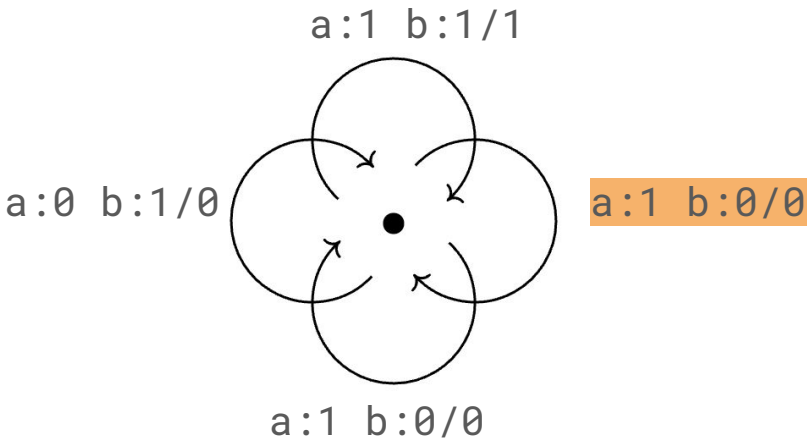
1 1 0 : **a**

0 1 1 : **b**

0 1 0 : **a&b**

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



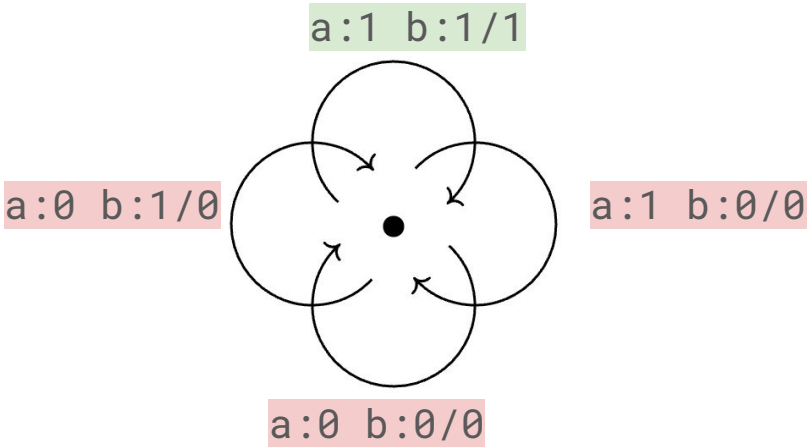
1 1 0 : **a**

0 1 1 : **b**

0 1 0 : **a&b**

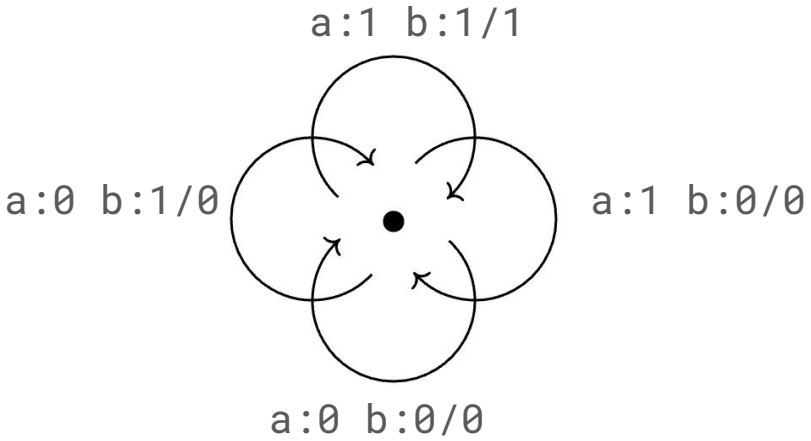
Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**

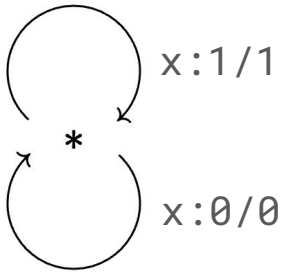


Does x&x=x Produce An Infinite Sequence of 1s?

Automata for **a&b:**

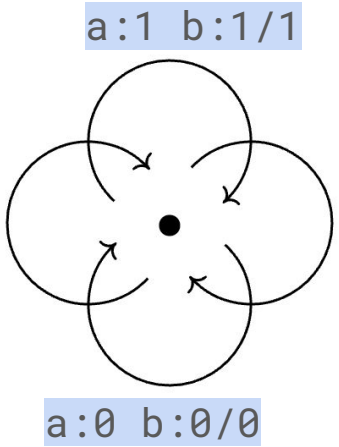


Automata for **x:**

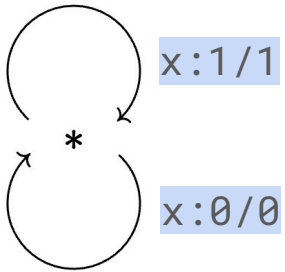


Does x&x=x Produce An Infinite Sequence of 1s?

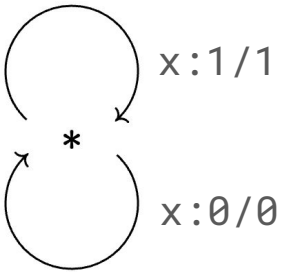
Automata for **a&b:**



Automata for **x:**

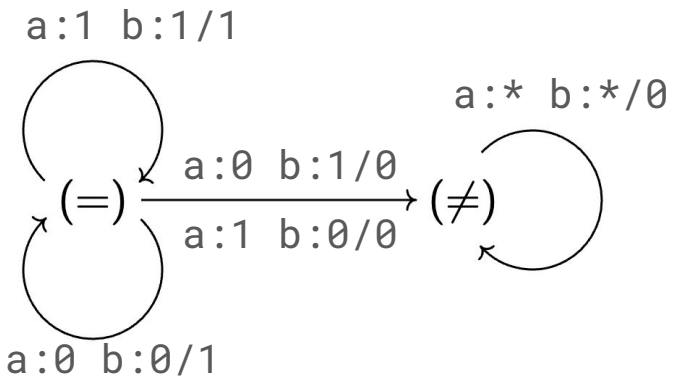


Automata for **x&x:**



Does x&x=x Produce An Infinite Sequence of 1s?

Automata for **a=b:**



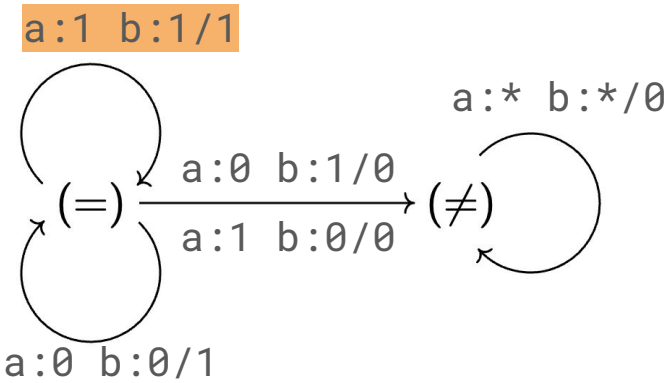
: **a**

: **b**

: **a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

Automata for **a=b:**



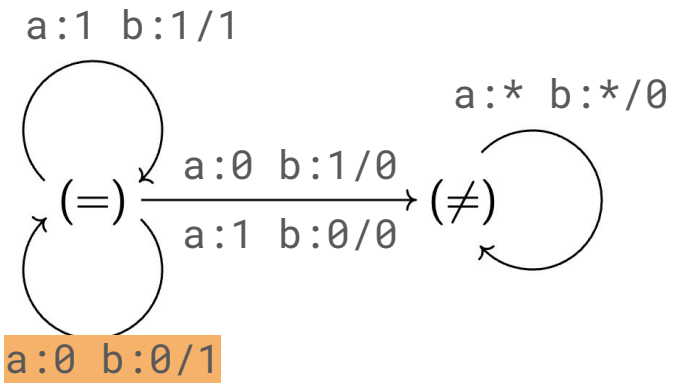
1 : **a**

1 : **b**

1 : **a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

Automata for **a=b:**



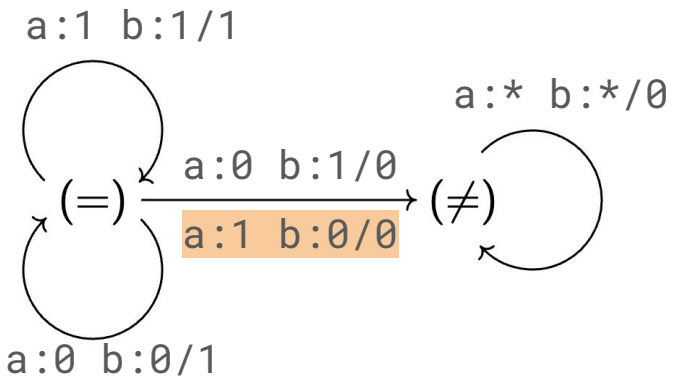
0 1 : **a**

0 1 : **b**

1 1 : **a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

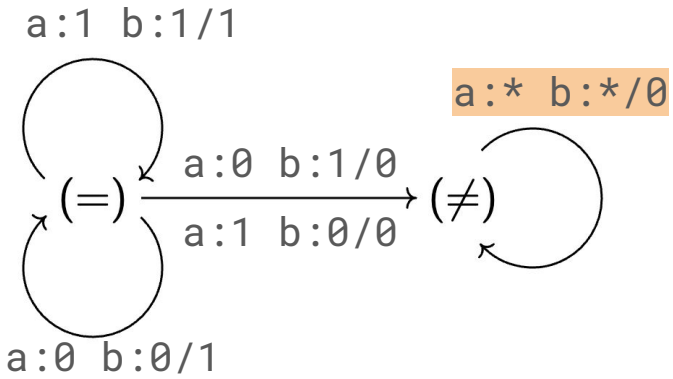
Automata for **a=b:**




```
0 0 1 :    a
1 0 1 :    b
0 1 1 : a=b
```

Does x&x=x Produce An Infinite Sequence of 1s?

Automata for **a=b:**



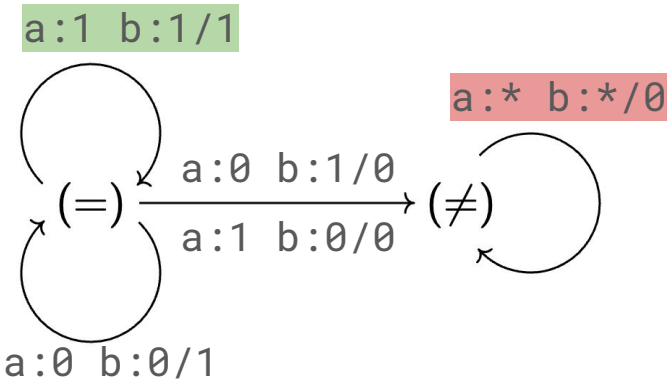
1 0 0 1 : **a**

1 1 0 1 : **b**

0 0 1 1 : **a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

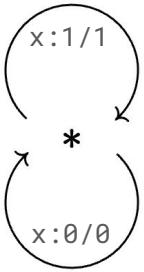
Automata for **a=b:**



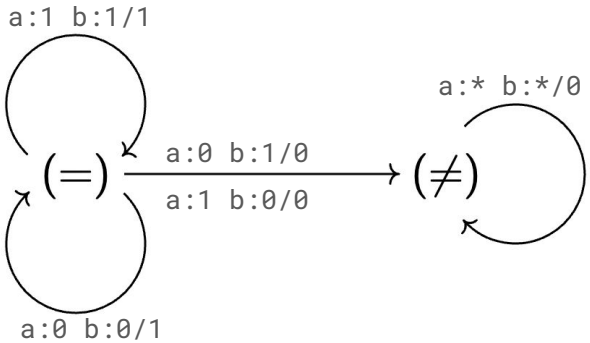
1	0	0	1	:	**a**
1	1	0	1	:	**b**
0	0	1	1	:	**a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

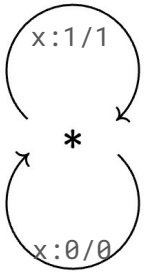
Automata for **x&x**:



Automata for **a=b**:

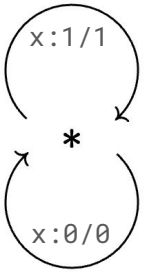


Automata for **x**:

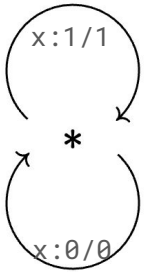


Does x&x=x Produce An Infinite Sequence of 1s?

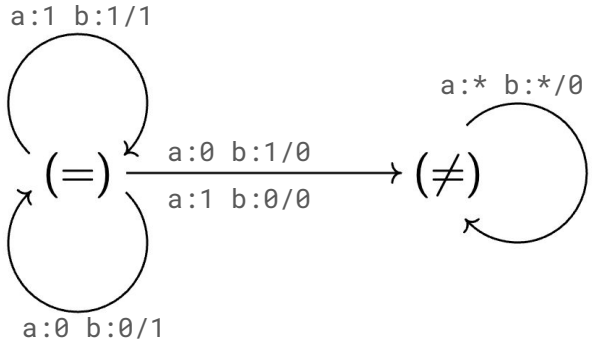
Automata for **x&x:**



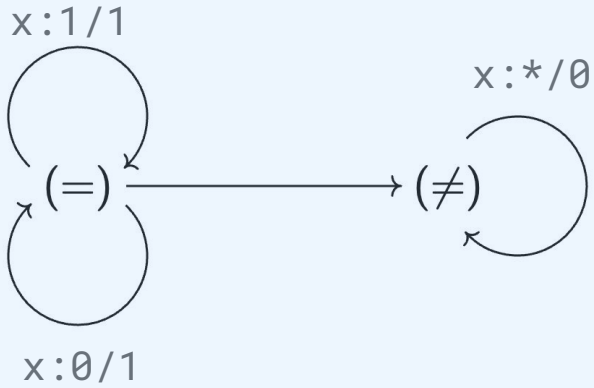
Automata for **x:**



Automata for **a=b:**

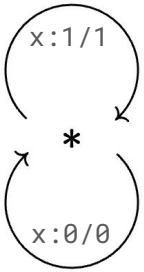


Automata for **x&x=x:**

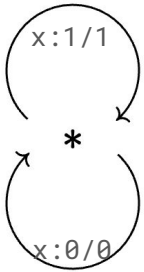


Does x&x=x Produce An Infinite Sequence of 1s?

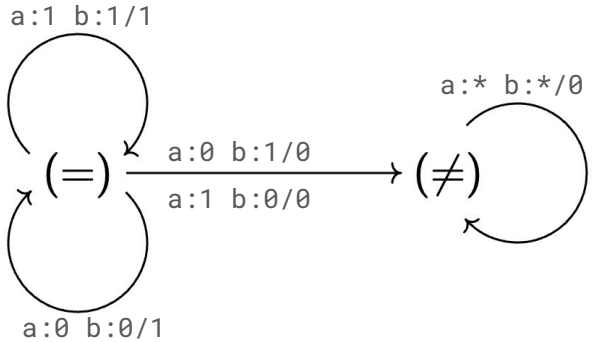
Automata for **x&x:**



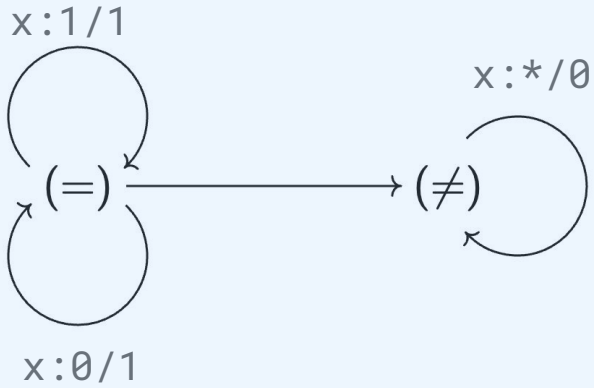
Automata for **x:**



Automata for **a=b:**



Automata for **x&x=x:**



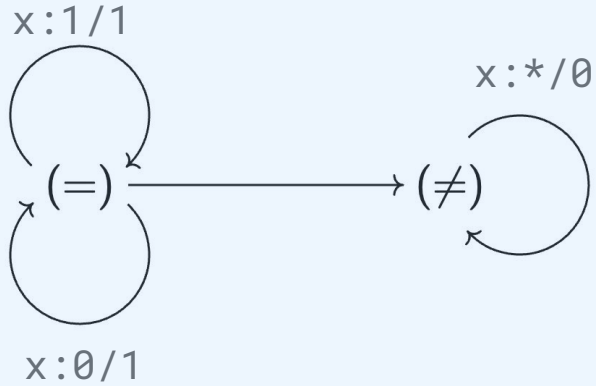
Does Automata for **x&x=x** Always Produce 1s?

Does x&x=x Produce An Infinite Sequence of 1s?

```
theorem and_idem: ∀ (w : Nat)(x : BitVec w),
    x & x = x := by bv_automata
```

```
          ...x1    x0 :     x
          ...x1    x0 :     x
       ...x1&x1 x0&x0 :   x&x
          ...1     1 :  x&x=x
```

Automata for **x&x=x:**



YES: Automata for **x&x=x** Always Produce 1s!

Does x&x=x Produce An Infinite Sequence of 1s?

YES: Automata for **x&x=x** Always Produce 1s!

Model Checking / **k-induction**

Reachable by path p:

$$(s : S) \xrightarrow{p}{}^{*} (u : S) \equiv \begin{cases} s = u & p = \langle\rangle \\ \delta(s, p_0) = t \wedge t \xrightarrow{q}{}^{*} u & p = \langle p_0; q\rangle \end{cases}$$

Reachable by path p, all intermediate states output true:

$$(s : S) \xrightarrow{p}{}^{*}_{\text{true}} (u : S) \equiv \begin{cases} s = u & p = \langle\rangle \\ \pi(s, p_0) = \text{true} \wedge \delta(s, p_0) = t \wedge t \xrightarrow{q}{}^{*}_{\text{true}} u & p = \langle p_0; q\rangle \end{cases}$$

States reachable in k steps from s_0 are safe:

$$\text{InitPrecond}_k \equiv \forall (t : S)\ (i : \mathbb{B})\ (p : \text{BitVec } k), s_0 \xrightarrow{p}{}^{*} t \implies \pi(t, i) = \text{true}$$

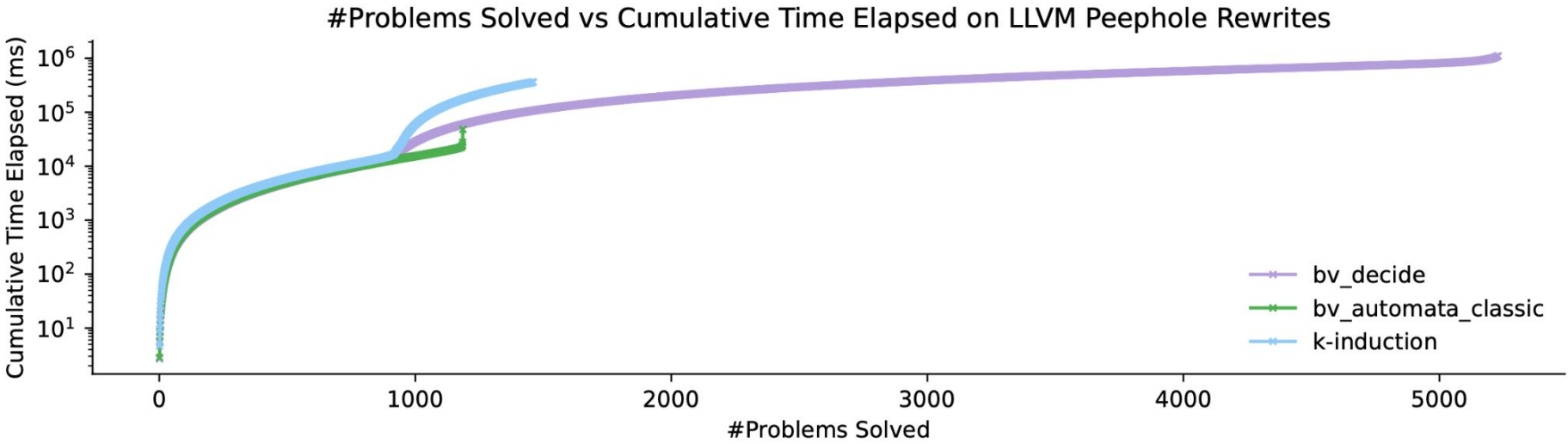
Safe reachability in k steps can be extended to $k + 1$ steps:

$$\text{Ind}_k \equiv \bigvee_{j=1}^{k} \forall (s\ t : S)\ (i : \mathbb{B})\ (p : \text{BitVec } k), s \xrightarrow{p}{}^{*}_{\text{true}} t \implies \pi(t, i) = \text{true}$$

Safety = Preconditions + Inductive Invariant:

$$\text{Safe}_k \equiv \text{InitPrecond}_k \wedge \text{Ind}_k$$

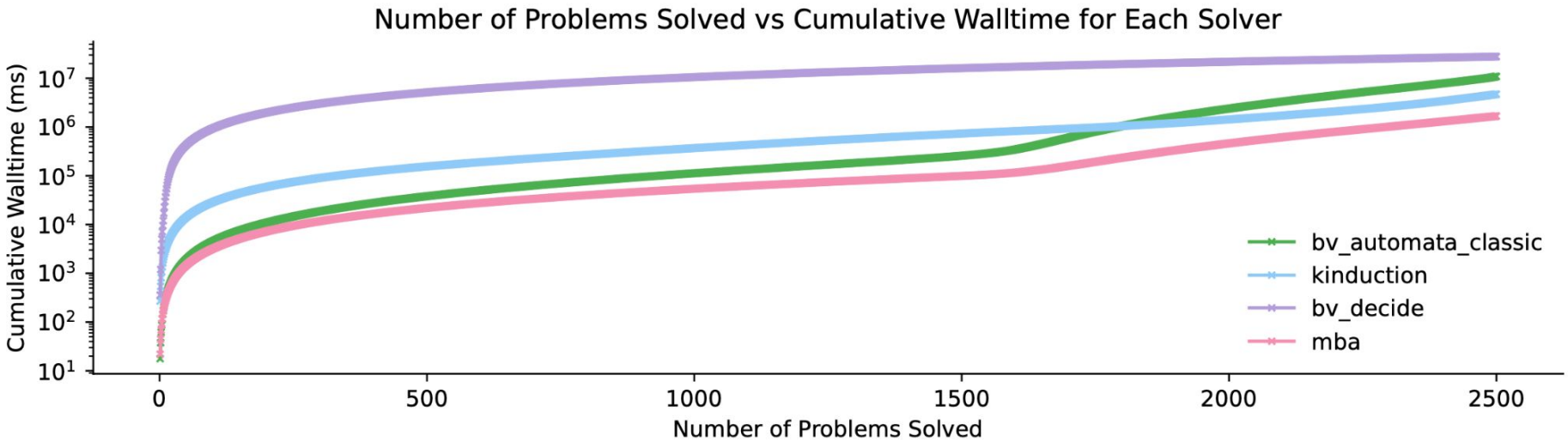
How Well Do These Algorithms Work?



- Rewrites With Constants For Fixed Width: **~500** problems (7 → 2^w-1)
- Rewrites With Multiple Widths: **~2000** problems (v, w, ...)
- Rewrites With Mul / Div / ... : **~500** problems

How Well Do These Algorithms Work?

Problems Solved v/s walltime on MBA-Blast



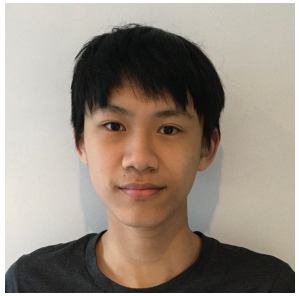

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

Questions + Current & Future Work

- Current Reduction: Based on PA reduction to automata.
  + Extensions for bit-irregular operations: Zero/Sign extension.
- Theoretical question: Is **BV + append decidable**?
  + Note: embeds universally quantified theory of strings!

- Practical Question: Fastest Model Checking Algorithm? (IC3?)
  + Has **IC3 been mechanized**?

Floating Point Bitblasting



CVC5

Verify FP by Reduction to SAT

martin-cs/**symfpu**

A (concrete or symbolic) implementation of IEEE-754 / SMT-LIB floating-point

1 Contributor · 5 Issues · 40 Stars · 20 Forks

symfpu: Library for FP circuits

`Fp.lean`: FP Bitblasting for Lean

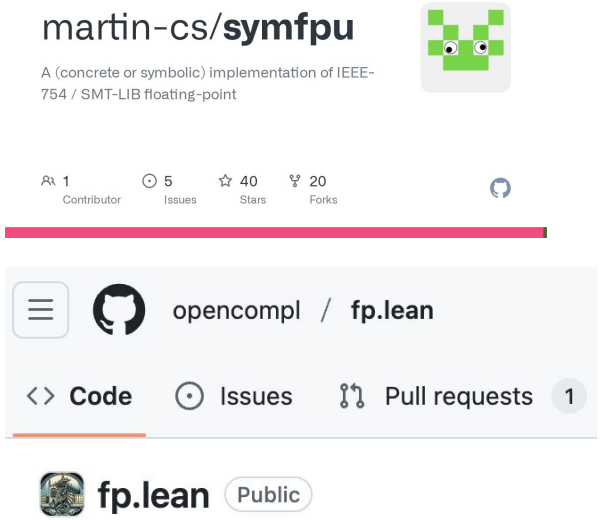
FP number (8 bit): a, b

Unpacked Fixed Width (32 bit)
a′ = a.unpack; b′ = b.unpack

Operation (32 bit) : c′ = a′ + b′

Rounding (32 bit) : cr = c′.round 5 3

Packing (8 bit): out = cr.pack



Written in **fragment of Lean that is bv_decide comprehensible**!

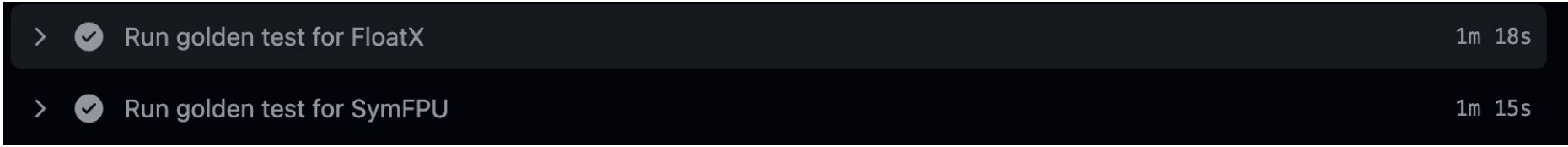
Fp.lean: Theorems!

https://opencompl.github.io/fp.lean/Fp/TheoremsFP16.html

```
∀ (a b c : PackedFloat 5 10)
  (m : RoundingMode) (hc : c.isNormOrSubnorm = true) :
  a ≤ b → (add a c m) ≤ (add b c m)
```

- mul / div time out at width > 10, sometimes in **simp**, sometimes in **bv_decide** (division is brutal).

`Fp.lean`: Trustworthiness, Completeness



https://github.com/opencompl/fp.lean/actions/runs/14981167344/job/42085394430

- Parity to SymFPU: `+`, `-`, `*` `/`
- WIP from SymFPU: `sqrt`, `FMA`
- Extended: **all rounding modes** (SymFPU has only RNE)
- Caveat: No FP ≃ $\mathbb{R}$ theory. Need Flocq-like library for this.

`Fp.lean`: **(FUTURE)** Bitblasting ≃ Real Semantics

FP number (8 bit): a, b	≃ closest representative of $\mathbb{R}$
Unpacked Fixed Width (32 bit) a′ = a.unpack; b′ = b.unpack	≃ $\mathbb{Q}$ with finite precision
Operation (32 bit) : c′ = a′ + b′	≃ operation on $\mathbb{Q}$ with finite precision
Rounding (32 bit) : cr = c′.round 5 3	≃ Find closest representative of $\mathbb{R}$ from $\mathbb{Q}$
Packing (8 bit): out = cr.pack	≃ closest representative of $\mathbb{R}$

Written in **fragment of Lean that is bv_decide comprehensible**!

`Fp.lean`: **(FUTURE)** Bitblasting ≃ Real Semantics

FP number (8 bit): a, b	≃ closest representative of $\mathbb{R}$
Unpacked Fixed Width (32 bit) a′ = a.unpack; b′ = b.unpack	≃ $\mathbb{Q}$ with finite precision
Rounding (32 bit): cr = c′.round 5 3	Find closest representative of $\mathbb{R}$ from $\mathbb{Q}$
Packing (8 bit): out = cr.pack	≃ closest representative of $\mathbb{R}$

Q: What's most impactful?

Written in **fragment of Lean that is bv_decide comprehensible**!

Bitvector Generalization (with Timi!)

[SSA/Experimental/Bits/Fast/Generalize.lean]

Hydra: Generalizing Peephole Optimizations with Program Synthesis

MANASIJ MUKHERJEE, University of Utah, USA
JOHN REGEHR, University of Utah, USA

Optimizing compilers rely on peephole optimizations to simplify combinations of instructions and remove redundant instructions. Typically, a new peephole optimization is added when a compiler developer notices an optimization opportunity—a collection of dependent instructions that can be improved—and manually derives a more general rewrite rule that optimizes not only the original code, but also other, similar collections of instructions. In this paper, we present Hydra, a tool that automates the process of generalizing peephole optimizations using a collection of techniques centered on program synthesis. One of the most important problems we have solved is finding a version of each optimization that is independent of the bitwidths of the optimization's inputs (when this version exists). We show that Hydra can generalize 75% of the ungeneralized missed peephole optimizations that LLVM developers have posted to the LLVM project's issue tracker. All of Hydra's generalized peephole optimizations have been formally verified, and furthermore we can automatically turn them into C++ code that is suitable for inclusion in an LLVM pass.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**; *Source code generation.*

Additional Key Words and Phrases: program synthesis, generalization, superoptimization, llvm, alive2, souper, hydra, peephole optimization

ACM Reference Format:
Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 120 (April 2024), 29 pages. https://doi.org/10.1145/3649837

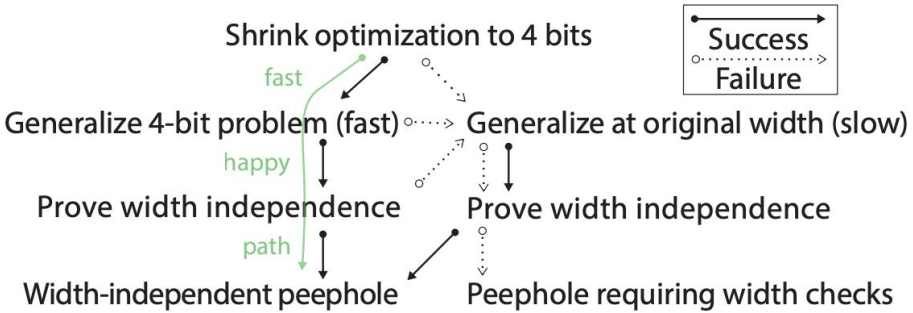


Fig. 3. Generalization pipeline with width reduction

Bitvector Generalization

SSA/Experimental/Bits/Fast/Generalize.lean

```
#generalize (x <<< 3) <<< 4 = x <<< 7
 ((x << y) << z) = (x << (y + z))
```

Reusable building blocks for BV algorithms:
- ∃ ∀ solver for BV
- Model counting for BV
- **Q: Other useful building blocks?**
- **Q: Generalization to Tensors?**
  What's a canonical set of operations?

CAD for BV algorithms

SSA/Experimental/Bits/Fast/Generalize.lean

- Reals (& p-adics) support CAD for deciding FO theory.
- Width-generic BV "looks a lot" like a **2adic integer**. Exploration of the connection?
- I implemented Real CAD, but relies **heavily** on arbitrary real & complex number library (Arb).
- Pointers to **implementations of p-adic CAD?**
- **Mechanized CAD?** (Assia, also other?)