Improving Equality Saturation for EDA via Semantic E-Graphs

EDA: Electronics Design Automation

Sijie Kong[1,⋆] **Jingtao Xia**[1,⋆] Daniel Ruelas-Petrisko[2]
Zachary D. Sisco[3] Jonathan Balkind[1] Gus Henry Smith[4]

⋆ Co-first authors

[1]UC Santa Barbara [2]U. of Washington [3]CUHK-Shenzhen [4]Southmountain Research

PLDI 2026

Two contributions

1. **Semantic e-graphs**: an e-graph variant that efficiently identifies *semantic* (not just syntactic) equalities.

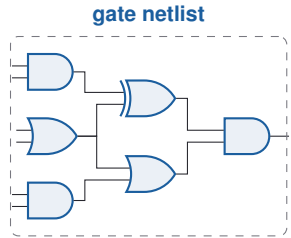
2. **NEXTMAP**: a new eqsat-based EDA framework on semantic e-graphs that bridges word- and bit-level, running more synthesis passes simultaneously than prior eqsat tools.

PART 1

Motivation

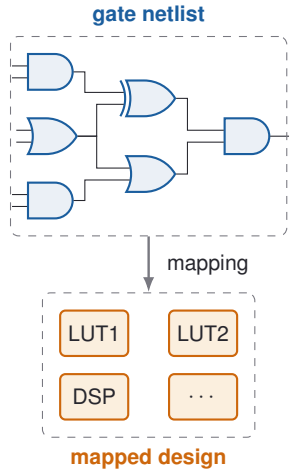
Hardware synthesis: many hand-ordered passes

- **Compiler for hardware**: a long pipeline of **passes**, hand-ordered (*phase ordering*).
  - Yosys runs **50+ passes** for one FPGA backend.



Hardware synthesis: many hand-ordered passes

- **Compiler for hardware**: a long pipeline of **passes**, hand-ordered (*phase ordering*).
  - Yosys runs **50+ passes** for one FPGA backend.
- Two kinds of pass:
  - **Optimization**: backend-independent transforms.
  - **Technology mapping**: lower onto a target's primitives (we target **FPGA**s: `LUT`s, flip-flops, `DSP`s).
- Mapping quality drives **QoR** (area, delay, power).



Today, mapping is *syntactic* pattern matching

- Per block (`DSP`, `LUT`, `BRAM`, ...): a hand-coded **library of syntactic patterns**.
- Mapper matches these patterns against the design; a block is used *only on an exact syntactic match*.

Today, mapping is *syntactic* pattern matching

- Per block (`DSP`, `LUT`, `BRAM`, ...): a hand-coded **library of syntactic patterns**.
- Mapper matches these patterns against the design; a block is used *only on an exact syntactic match*.

- *Example:* a previous pass outputs `a * (b + c)`; the DSP library pattern is `(b + c) * a`.
- Same semantics, wrong syntax $\rightarrow$ **DSP missed**, fall back to `LUT`s.

Today, mapping is *syntactic* pattern matching

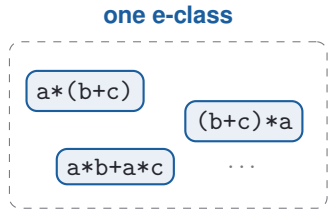
- Per block (`DSP`, `LUT`, `BRAM`, ...): a hand-coded **library of syntactic patterns**.
- Mapper matches these patterns against the design; a block is used *only on an exact syntactic match*.

- *Example:* a previous pass outputs `a * (b + c)`; the DSP library pattern is `(b + c) * a`.
- Same semantics, wrong syntax $\rightarrow$ **DSP missed**, fall back to `LUT`s.

Syntactic matching sees **one** form at a time.

Equality saturation: a principled basis

- **Within a stage,** eqsat removes the single-form limitation:
  - keep **all forms** (`a*(b+c)`, `(b+c)*a`, ...) in one **e-class**;
  - **match against all equivalent forms**; *extract* only once, at the end.

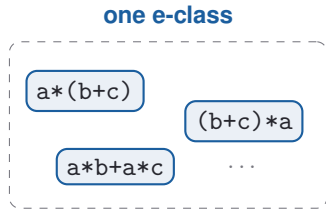


Within a stage, every equivalent form coexists in one e-class.

Equality saturation: a principled basis

- **Within a stage,** eqsat removes the single-form limitation:
  - keep **all forms** (`a*(b+c)`, `(b+c)*a`, ...) in one **e-class**;
  - **match against all equivalent forms**; *extract* only once, at the end.

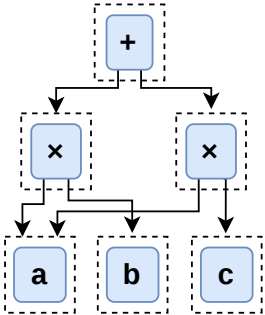
- **But synthesis is many stages.** At each boundary you extract *one* form and hand it off:
  - e.g. optimize in eqsat, then a *separate* mapper;
  - **phase ordering returns** *across* stages.



Within a stage, every equivalent form coexists in one e-class.

Example: within-stage eqsat is not enough

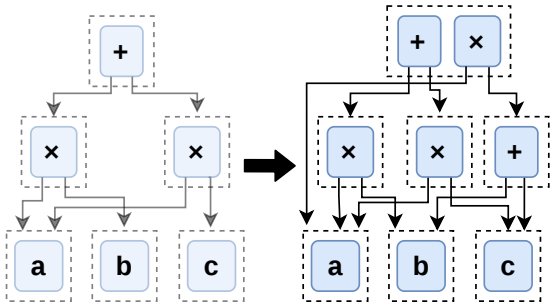
Same `a*(b+c)` example, across the optimize→map boundary.



(a) original egraph

Example: within-stage eqsat is not enough

Same `a*(b+c)` example, across the optimize→map boundary.



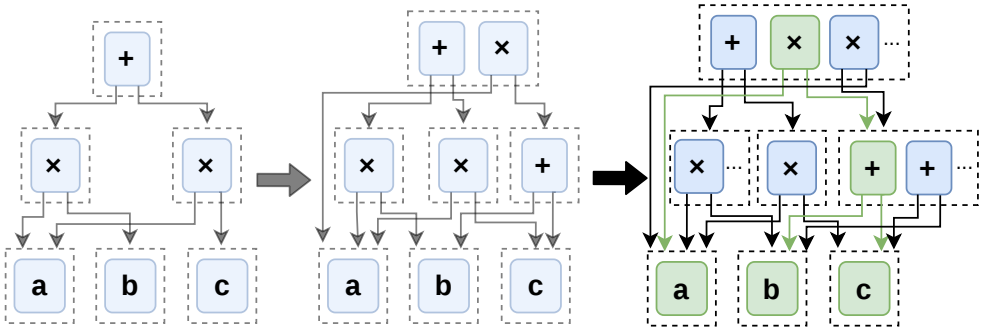
(a) original egraph

(b) egraph after distributivity

`(a×b)+(a×c) => a×(b+c)`

Example: within-stage eqsat is not enough

Same `a*(b+c)` example, across the optimize→map boundary.



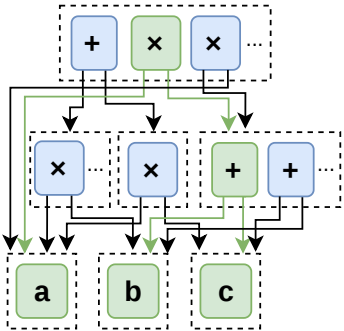
(a) original egraph

(b) egraph after distributivity
`(a×b)+(a×c) => a×(b+c)`

(c) egraph after additional rewrites;
example extraction in green

Example: within-stage eqsat is not enough

Same `a*(b+c)` example, across the optimize→map boundary.



(c) egraph after additional rewrites; example extraction in green

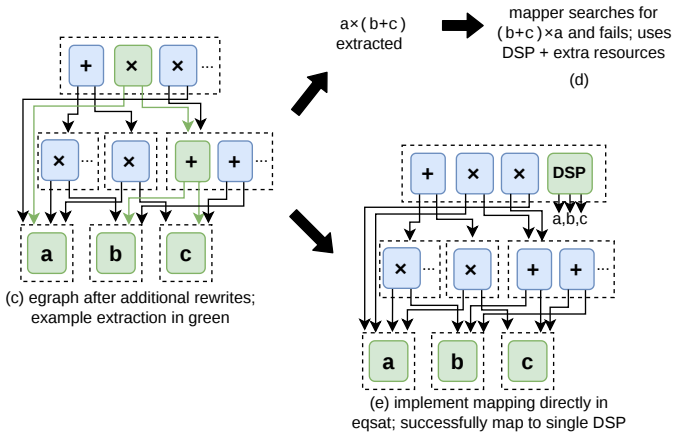
`a×(b+c)` extracted

mapper searches for `(b+c)×a` and fails; uses DSP + extra resources

(d)

Example: joint optimization *and* mapping recovers the DSP

Our fix: optimize and map in the same e-graph; no extraction in between.



PART 2

The Three Problems

Our goal: unify more synthesis stages

Run **more synthesis stages** *simultaneously* in **one e-graph**:
optimization *and* **technology mapping**.

Three problems stand in the way with vanilla eqsat.

Problem 1: prior eqsat for EDA stops at one stage

Tool	What it does with eqsat	Level
ROVER	arithmetic / datapath optimization	word
SEER	high-level synthesis	word
ESFO	FIRRTL circuit optimization	word
E-Syn	logic synthesis	bit
E-Morphic	Boolean / logic synthesis	bit
EqMap	LUT technology mapping	bit
Churchroad	DSP technology mapping	word

Problem 1: prior eqsat for EDA stops at one stage

Tool	What it does with eqsat	Level
ROVER	arithmetic / datapath optimization	word
SEER	high-level synthesis	word
ESFO	FIRRTL circuit optimization	word
E-Syn	logic synthesis	bit
E-Morphic	Boolean / logic synthesis	bit
EqMap	LUT technology mapping	bit
Churchroad	DSP technology mapping	word

Each tool is limited to a *single stage*;
none runs optimization and mapping simultaneously.

Problem 1: prior eqsat for EDA stops at one stage

Tool	What it does with eqsat	Level
ROVER	arithmetic / datapath optimization	word
SEER	high-level synthesis	word
ESFO	FIRRTL circuit optimization	word
E-Syn	logic synthesis	bit
E-Morphic	Boolean / logic synthesis	bit
EqMap	LUT technology mapping	bit
Churchroad	DSP technology mapping	word

Each tool is limited to a *single stage*;
none runs optimization and mapping simultaneously.

Why single-stage?
No prior eqsat representation bridges *word* and *bit* in one e-graph.

Problem 2: a syntactic word–bit bridge explodes

- One e-graph must hold **word-level** bitvectors *and* **bit-level** wires.
- The direct bridge is Verilog **slice** + **concat by syntax**, and it explodes.

Problem 2: a syntactic word–bit bridge explodes

- One e-graph must hold **word-level** bitvectors *and* **bit-level** wires.
- The direct bridge is Verilog **slice** + **concat by syntax**, and it explodes.

The *same* 3-bit signal `s`:

```
{s[2],s[1],s[0]} = {s[2:1],s[0]} = {s[2],s[1:0]} = s[2:0]
```

One `egglog` rewrite enumerates them; n bits $\rightarrow 2^{n-1}$ forms (associativity of concat):

```
(rewrite s (Concat (Extract n-1 i) (Extract i-1 0)))
```

Problem 2: a syntactic word–bit bridge explodes

- One e-graph must hold **word-level** bitvectors *and* **bit-level** wires.
- The direct bridge is Verilog **slice** + **concat by syntax**, and it explodes.

The *same* 3-bit signal `s`:

```
{s[2],s[1],s[0]} = {s[2:1],s[0]} = {s[2],s[1:0]} = s[2:0]
```

One `egglog` rewrite enumerates them; n bits $\rightarrow 2^{n-1}$ forms (associativity of concat):

```
(rewrite s (Concat (Extract n-1 i) (Extract i-1 0)))
```

Our fix: use *semantic* equality, identifying a bitvector by its **underlying bits**:

Invariant: $\mathsf{bits}(e_1) = \mathsf{bits}(e_2) \iff e_1 = e_2$

One semantic invariant replaces 2^{n-1} syntactic forms.

$\rightarrow$ *but how do today's engines maintain it efficiently?*

Computing underlying bits: an e-class analysis

Standard `egg`/`egglog` can attach each expression's bits with an **e-class analysis** `(HasBits e [`$b_{n-1}, \ldots, b_0$`])`, built *compositionally*:

```
(rule ((Var name bw))                                  ; symbolic input
      ((HasBits (Var name bw) (fresh-bits bw))))
(rule ((BV val bw))                                    ; constant
      ((HasBits (BV val bw) (to-bits val bw))))
(rule ((Concat a b) (HasBits a as) (HasBits b bs)) ; combine
      ((HasBits (Concat a b) (append as bs))))
(rule ((Extract h l e) (HasBits e es))                 ; slice
      ((HasBits (Extract h l e) (slice es l h))))
```

Problem 3: standard e-graphs maintain the invariant inefficiently

Computing the bits is cheap; *maintaining equality* on them is not. The invariant $\mathsf{bits}(e_1) = \mathsf{bits}(e_2) \iff e_1 = e_2$ needs two **inefficient** rules:

Problem 3: standard e-graphs maintain the invariant inefficiently

Computing the bits is cheap; *maintaining equality* on them is not. The invariant $\text{bits}(e_1) = \text{bits}(e_2) \iff e_1 = e_2$ needs two **inefficient** rules:

Forward (bits → bitvector)

```
(rule ((HasBits e1 b)
       (HasBits e2 b))
      ((union e1 e2)))
```

$O(n^2)$: pairwise over all **bitvectors** sharing bits.

Backward (bitvector → bits)

```
(rule ((HasBits e b1)
       (HasBits e b2))
      ((union b1[0] b2[0])
       ...
       (union b1[n-1] b2[n-1])))
```

$O(n^2)$: pairwise over a class's **analysis facts** (its bit-vectors).

Problem 3: standard e-graphs maintain the invariant inefficiently

Computing the bits is cheap; *maintaining equality* on them is not. The invariant $\text{bits}(e_1) = \text{bits}(e_2) \iff e_1 = e_2$ needs two **inefficient** rules:

Forward (bits $\to$ bitvector)

```
(rule ((HasBits e1 b)
       (HasBits e2 b))
      ((union e1 e2)))
```

$O(n^2)$: pairwise over all **bitvectors** sharing bits.

Backward (bitvector $\to$ bits)

```
(rule ((HasBits e b1)
       (HasBits e b2))
      ((union b1[0] b2[0])
       ...
       (union b1[n-1] b2[n-1])))
```

$O(n^2)$: pairwise over a class's **analysis facts** (its bit-vectors).

Solution. Introduce semantic e-graphs.

PART 3

Semantic E-Graphs

Three problems, three ideas

Problem

1. Single-stage eqsat: **phase ordering** *between* stages.

SEG idea

1. **Multi-level representation**: multi-level structures in *one* e-graph.

Three problems, three ideas

Problem

1. Single-stage eqsat: **phase ordering** *between* stages.
2. Word–bit bridge *by syntax*: the e-graph **explodes**.

SEG idea

1. **Multi-level representation**: multi-level structures in *one* e-graph.
2. **Semantic eid**: *replaces* the opaque id; the id now *reflects semantics*.

Three problems, three ideas

Problem

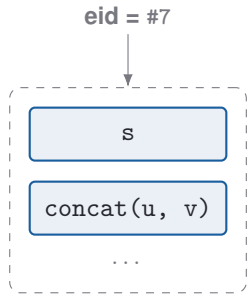
1. Single-stage eqsat: **phase ordering** *between* stages.
2. Word–bit bridge *by syntax*: the e-graph **explodes**.
3. Maintaining the invariant is **inefficient** in standard e-graphs.

SEG idea

1. **Multi-level representation**: multi-level structures in *one* e-graph.
2. **Semantic eid**: *replaces* the opaque id; the id now *reflects semantics*.
3. **Maintain seids efficiently**: a semantic *constructor* + *decomposer* .

From opaque eids to *semantic* eids

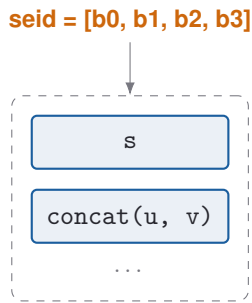
- **Opaque eid** (standard): an *opaque identifier* (e.g. an integer) that represents the e-class but carries *no semantics*.



From opaque eids to *semantic* eids

- **Opaque eid** (standard): an *opaque identifier* (e.g. an integer) that represents the e-class but carries *no semantics*.

- **Semantic eid** (*seid*): a *semantic identifier* from a user-defined domain; it *is* the e-class's **semantic value**.
  - a **concrete** value → equal seids **auto-merge**;

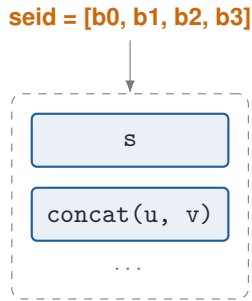


From opaque eids to *semantic* eids

- **Opaque eid** (standard): an *opaque identifier* (e.g. an integer) that represents the e-class but carries *no semantics*.

- **Semantic eid** (*seid*): a *semantic identifier* from a user-defined domain; it *is* the e-class's **semantic value**.
  - a **concrete** value → equal seids **auto-merge**;
  - else **symbolic**: same as an opaque eid as fallback.

Semantics are **first-class**



.

Maintaining seids: three operations

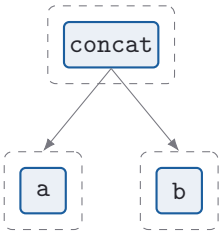
egg's operations: `add`,`merge`,`rebuild`, each with *one* small semantic extension:

operations	**egg does**	**+ SEG adds**
`sadd`	canonicalize children; reuse or make a new e-class	**constructor** $[\![f]\!]$
`smerge`	union the two e-classes	**decompose** Δ
`rebuild`	re-canonicalize parents; upward-merge congruent	**propagate** to fixpoint

egg is lazy: `merge` appends to a worklist that `rebuild` flushes in batches.

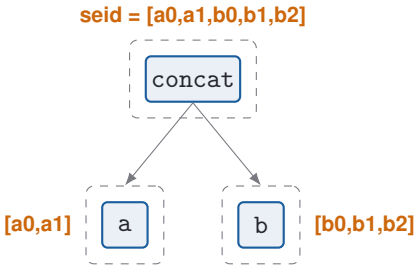
Standard rewriting and e-matching unchanged.

Insertion (`sadd`): the semantic constructor



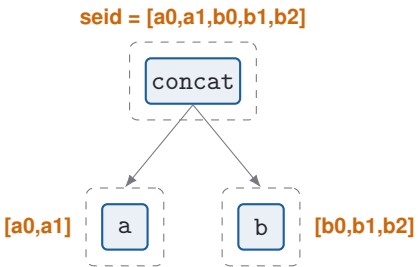
- **Standard `add`.**

Insertion (`sadd`): the semantic constructor



- **Standard `add`.**
- $+$ compute the seid via the **semantic constructor** $[\![f]\!](\delta_1, \ldots, \delta_m) = \delta$, and annotate the new e-class with it.
  - $[\![\text{concat}]\!]([a0, a1], [b0, b1, b2]) = [a0, a1, b0, b1, b2]$

Insertion (sadd): the semantic constructor



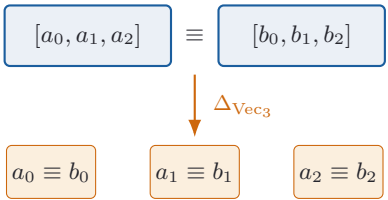
- **Standard add.**
- $+$ compute the seid via the **semantic constructor** $[\![f]\!](\delta_1, \ldots, \delta_m) = \delta$, and annotate the new e-class with it.
  - $[\![\text{concat}]\!]([a0, a1], [b0, b1, b2]) = [a0, a1, b0, b1, b2]$
- $+$ that δ may match another e-class's $\rightarrow$ a free equality.

The constructor establishes the equality at insertion, with no rewrite rule.

Merging (`smerge`): the semantic decomposer

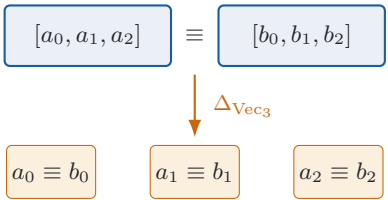
- **Standard `merge`.**

Merging (`smerge`): the semantic decomposer



- **Standard `merge`.**
- $+$ if both seids are composite, the **semantic decomposer** Δ splits the merge into **component merges**.

Merging (`smerge`): the semantic decomposer



- **Standard `merge`.**
- $+$ if both seids are composite, the **semantic decomposer** Δ splits the merge into **component merges**.
- $+$ Each component merge may **recursively invoke** `smerge`.

Merging a *composite* seid **propagates equality downward to its components** .

`rebuild`: propagate seids to a fixpoint

- **Standard `rebuild`.**

`rebuild`: propagate seids to a fixpoint

- **Standard `rebuild`.**
- $+$ SEG **propagates seids in two directions**, recursively:
  - **Upward (constructor):** recompute $[\![f]\!]$ on affected parents; a recomputed seid that coincides with another e-class's **merges the two e-classes**.
  - **Downward:** merging *composite* seids invokes `smerge`'s **decomposer** $(\Delta) \to$ **component merges**.

`rebuild`: propagate seids to a fixpoint

- **Standard `rebuild`.**
- \+ SEG **propagates seids in two directions**, recursively:
  - **Upward (constructor):** recompute $[\![f]\!]$ on affected parents; a recomputed seid that coincides with another e-class's **merges the two e-classes**.
  - **Downward:** merging *composite* seids invokes `smerge`'s **decomposer** $(\Delta) \to$ **component merges**.
- Iterate to a **fixpoint**.

Properties

- **Soundness.** SEG maintains exactly the *least congruence* of its two equality sources:

$$\sim_{\mathsf{SEG}} = \mathrm{Cong}\big(R_{\mathsf{syntactic}} \cup R_{\mathsf{semantic}}\big)$$

  - $R_{\mathsf{syntactic}}$: rewrite rules; R_{semantic}: constructor $[\![f]\!]$ + decomposer Δ.
  - both sound w.r.t. ground truth $\equiv$ $\Rightarrow$ $\sim_{\mathsf{SEG}} \subseteq \equiv$.

- **Termination.** `rebuild` always reaches a *fixpoint*:
  - each merge strictly shrinks the e-class count;
  - seids only *refine* ($\delta_1, \delta_2 \sqsubseteq_\tau \delta^*$), along chains of **finite height**.

PART 4

Nextmap

Instantiating the semantic e-graph for hardware

Back to hardware: we build NEXTMAP based on the semantic e-graph.

- A bitvector's **seid** is the vector of its underlying wires.

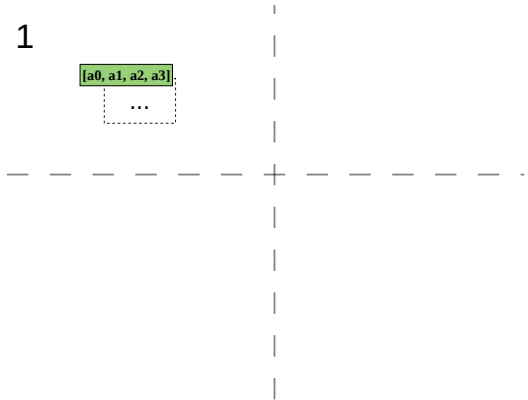
Instantiating the semantic e-graph for hardware

Back to hardware: we build NEXTMAP based on the semantic e-graph.

- A bitvector's **seid** is the vector of its underlying wires.

- **Hardware e-class types:**
  - `Wire`: a 1-bit signal.
  - `WireVec`: a bitvector; its *seid* is a `Vec[Wire]`.
  - `Register`: a stateful element (D flip-flop).
- **Cells**: the primitive operations over these types (logic gates, add, mux, slice/concat, DSP, . . .), selected at extraction.

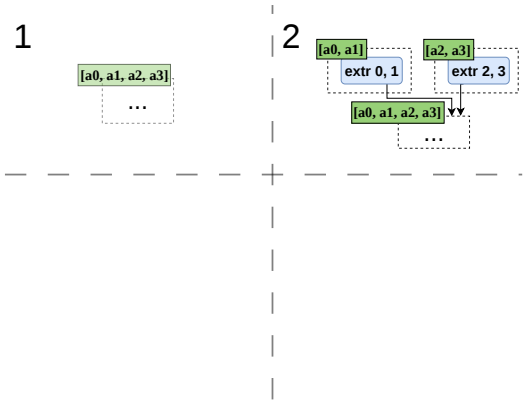
Example: slice–concat round-trips



Goal: recover `concat(v[0:1], v[2:3]) = v`.

1. 4-bit `v`, seid `[a0,a1,a2,a3]`.

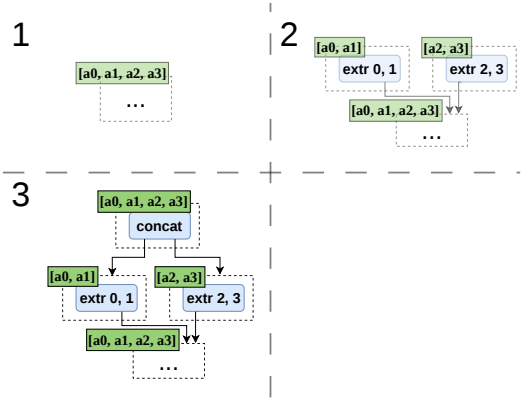
Example: slice–concat round-trips



Goal: recover `concat(v[0:1], v[2:3]) = v`.

1. 4-bit `v`, seid `[a0,a1,a2,a3]`.
2. Slice: `extr(v,0,1)` $\rightarrow$ `[a0,a1]`, `extr(v,2,3)` $\rightarrow$ `[a2,a3]`.

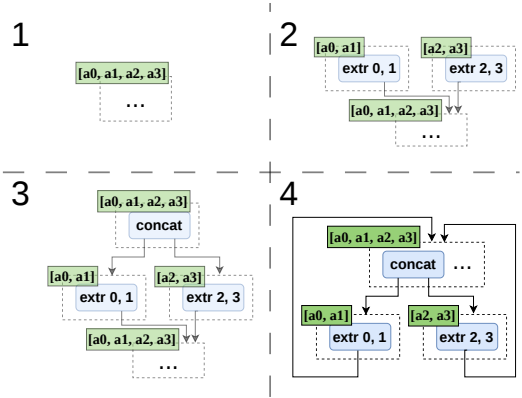
Example: slice–concat round-trips



Goal: recover `concat(v[0:1], v[2:3]) = v`.

1. 4-bit `v`, seid `[a0,a1,a2,a3]`.
2. Slice: `extr(v,0,1)` $\rightarrow$ `[a0,a1]`, `extr(v,2,3)` $\rightarrow$ `[a2,a3]`.
3. Concat: `concat([a0,a1], [a2,a3]) = [a0,a1,a2,a3]`.

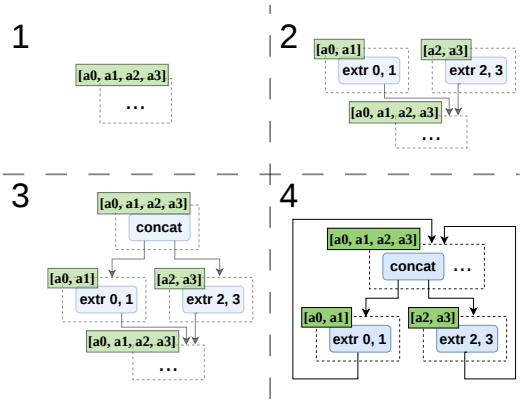
Example: slice–concat round-trips



Goal: recover `concat(v[0:1], v[2:3]) = v`.

1. 4-bit `v`, seid `[a0,a1,a2,a3]`.
2. Slice: `extr(v,0,1)` $\rightarrow$ `[a0,a1]`, `extr(v,2,3)` $\rightarrow$ `[a2,a3]`.
3. Concat: `concat([a0,a1], [a2,a3]) = [a0,a1,a2,a3]`.
4. Same seid as `v` $\rightarrow$ **auto-merge**.

Example: slice–concat round-trips



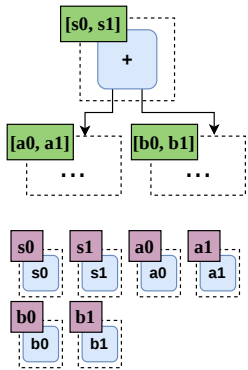
Goal: recover `concat(v[0:1], v[2:3]) = v`.

1. 4-bit `v`, seid `[a0,a1,a2,a3]`.
2. Slice: `extr(v,0,1)` → `[a0,a1]`, `extr(v,2,3)` → `[a2,a3]`.
3. Concat: `concat([a0,a1], [a2,a3]) = [a0,a1,a2,a3]`.
4. Same seid as `v` → **auto-merge**.

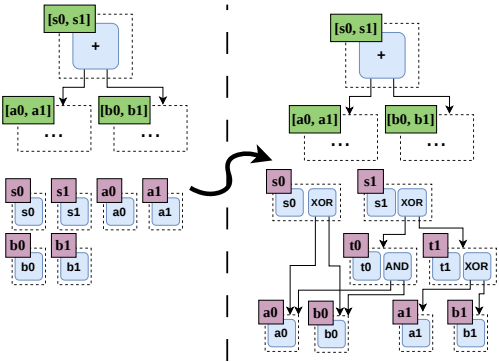
Slice/concat equivalence emerges from the seid alone; no rewrite rule.

Example: cross-level synthesis of an adder

Bit-blasting rule. For a 2-bit adder:



Example: cross-level synthesis of an adder



Bit-blasting rule. For a 2-bit adder:

```
adder(a,b) -> [xor(a0,b0),
               xor(and(a0,b0),
                   xor(a1,b1))]
```

Example: cross-level synthesis of an adder

2-bit: `s[0:1] = a[0:1] + b[0:1]`

3-bit: `t[0:2] = a[0:2] + b[0:2]`

Bit-blasting rule. For a 2-bit adder:

```
adder(a,b) -> [xor(a0,b0),
               xor(and(a0,b0),
                   xor(a1,b1))]
```

Now a 3-bit adder over the same inputs (same rule applies):

Example: cross-level synthesis of an adder

2-bit: `s[0:1] = a[0:1] + b[0:1]`

3-bit: `t[0:2] = a[0:2] + b[0:2]`

Auto-shared: `s[0:1] = t[0:1]`

Bit-blasting rule. For a 2-bit adder:

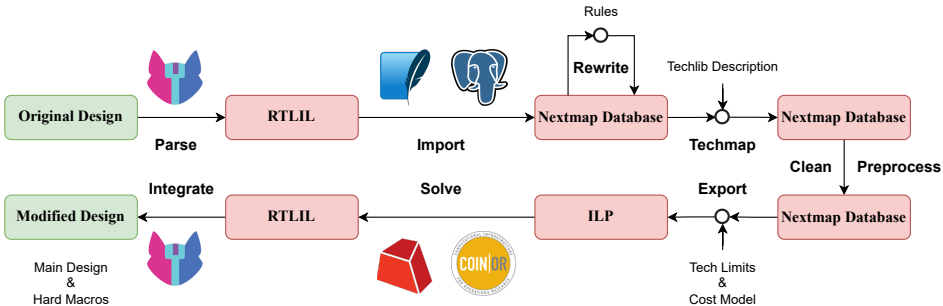
```
adder(a,b) -> [xor(a0,b0),
               xor(and(a0,b0),
                   xor(a1,b1))]
```

Now a 3-bit adder over the same inputs (same rule applies):

- The word-level slice sharing is *propagated up* from the **bit-level netlist**.

Keeping word- and bit-level together exposes more sharing, more optimization opportunities.

NEXTMAP: one e-graph, every pass



All synthesis passes share the e-graph

Technology-independent rewrites

- *Arithmetic*: commutativity, associativity, width reduction, Karatsuba
- *Bitblast*: adder / comparator expansion, word-mux
- *Logic*: De Morgan, absorption, idempotence
- *Retiming*: forward / backward push registers

Technology mapping rewrite rules

- User-supplied *techlib*: target cells (DSPs, BRAMs, . . .) expressed as rewrite rules.
- *Recursive patterns*: target *repetitive cell structures*, e.g. a MAC mesh.

Extraction via ILP

After saturation, one **ILP** selects a type-consistent implementation from the SEG.

- **General model:** binary variables pick *one cell per e-class*; constraints enforce driver consistency; minimize a *resource cost*.
- **Legal loops:** registers are modeled explicitly, so **feedback through registers is allowed** while **combinational cycles are forbidden**.
- **Timing-aware:** per-wire delay variables *bound the critical path*; a combinational loop implies unbounded delay, so it is rejected with no extra constraint.
- **Complex technology cells:** *nonuniform input–output timing* or *internal pipelines*; *split-and-bind* method.

PART 5

Evaluation

Research Questions

RQ1 How is NEXTMAP's quality of results vs. existing tools?

RQ2 Does running optimization *and* mapping *simultaneously* actually help?

Research Questions

RQ1 How is NEXTMAP's quality of results vs. existing tools?

RQ2 Does running optimization *and* mapping *simultaneously* actually help?

Two more (expressivity, Semantic E-Graph efficiency) in the paper.

RQ1: Quality of results vs. existing tools

Compared against **Yosys** (open source) and a **proprietary** commercial FPGA tool; lower is better.

3 of 14 *large* designs (full Table 3 in the paper):

		NEXTMAP				Yosys				Proprietary			
Benchmark	Wires	DSP	CY4	FF	LUT	DSP	CY4	FF	LUT	DSP	CY4	FF	LUT
SA(4,8)	1 842	16	**0**	**128**	**0**	16	64	448	400	**0**	256	448	1648
SA(4,32)	7 218	**48**	**594**	**1568**	3568	64	720	1792	3848	64	448	1280	1776
SA(8,32)	28 770	**192**	**2426**	**7200**	14432	256	2880	7680	15232	240	2904	6656	13456
						...							

RQ1: Quality of results vs. existing tools

Compared against **Yosys** (open source) and a **proprietary** commercial FPGA tool; lower is better.

3 of 14 *large* designs (full Table 3 in the paper):

		NEXTMAP				Yosys				Proprietary			
Benchmark	Wires	DSP	CY4	FF	LUT	DSP	CY4	FF	LUT	DSP	CY4	FF	LUT
SA(4,8)	1 842	16	**0**	**128**	**0**	16	64	448	400	**0**	256	448	1648
SA(4,32)	7 218	**48**	**594**	**1568**	3568	64	720	1792	3848	64	448	1280	1776
SA(8,32)	28 770	**192**	**2426**	**7200**	14432	256	2880	7680	15232	240	2904	6656	13456
						...							

- vs. **Yosys**: $\leq$ on every resource, **14/14 designs**; **25% fewer DSPs** at scale.
- vs. **commercial**: competitive; commercial achieves better `LUT`/`FF` packing on some designs.
- Runtime: seconds for most; largest (16×16, 439k sat. wires) **$\sim$32 min, $>$90% in ILP**.

competitive with commercial EDA tools at scale.

RQ2: Does *simultaneous* optimization + mapping matter?

Ablation: NEXTMAP vs. **phase-ordered NEXTMAP**: identical rules, but executed in *separate* phases with an extraction between each.

		NEXTMAP				**Phase-ordered**			
Benchmark	Wires	DSP	CY4	FF	LUT	DSP	CY4	FF	LUT
SA(4,8)	1 842	16	**0**	**128**	**0**	16	64	320	304
FIR(16,8)	1 082	16	**16**	**105**	**285**	16	32	225	1002
FFT(1024,32)	345 238	**37**	**982**	1482	**10410**	49	1022	1482	10428
				...					

RQ2: Does *simultaneous* optimization + mapping matter?

Ablation: NEXTMAP vs. **phase-ordered NEXTMAP**: identical rules, but executed in *separate* phases with an extraction between each.

		NEXTMAP				Phase-ordered			
Benchmark	Wires	DSP	CY4	FF	LUT	DSP	CY4	FF	LUT
SA(4,8)	1 842	16	**0**	**128**	**0**	16	64	320	304
FIR(16,8)	1 082	16	**16**	**105**	**285**	16	32	225	1002
FFT(1024,32)	345 238	**37**	**982**	1482	**10410**	49	1022	1482	10428
				...					

NEXTMAP **ties or beats** phase-ordered NEXTMAP on **all 13 designs**, often far better on logic.

PART 6

Conclusion

Takeaway

1. **Problem.** Prior eqsat for EDA stops at *one* stage.

2. **Semantic e-graphs.** More semantic equalities, detected efficiently, via *seids*: a first-class encoding of semantics.

3. **NEXTMAP.** Bridges word- and bit-level, more synthesis passes at once; QoR competitive with commercial tools.

Takeaway

1. **Problem.** Prior eqsat for EDA stops at *one* stage.

2. **Semantic e-graphs.** More semantic equalities, detected efficiently, via *seids*: a first-class encoding of semantics.

3. **NEXTMAP.** Bridges word- and bit-level, more synthesis passes at once; QoR competitive with commercial tools.

Enables optimizations beyond the reach of vanilla eqsat for EDA.

Takeaway

1. **Problem.** Prior eqsat for EDA stops at *one* stage.

2. **Semantic e-graphs.** More semantic equalities, detected efficiently, via *seids*: a first-class encoding of semantics.

3. **NEXTMAP.** Bridges word- and bit-level, more synthesis passes at once; QoR competitive with commercial tools.

Enables optimizations beyond the reach of vanilla eqsat for EDA.

Beyond hardware: *should generalize to any domain with structured semantic values (strings, sets, polynomials, . . .).*

Thank you!

Questions?

https://github.com/UCSBarchlab/nextmap.

sijie_kong@ucsb.edu | jingtaoxia@ucsb.edu