

UC Santa Barbara

UC Santa Barbara Electronic Theses and Dissertations

Title

A Principled Approach to Technology Mapping

Permalink

<https://escholarship.org/uc/item/6j04r7n3>

ISBN

9798186381907

Author

Kong, Sijie

Publication Date

2026-06-07

Peer reviewed|Thesis/dissertation

University of California
Santa Barbara

A Principled Approach to Technology Mapping

A thesis submitted in partial satisfaction
of the requirements for the degree

Master of Science
in
Electrical and Computer Engineering

by

Sijie Kong

Committee in charge:

Prof. Jonathan Balkind, Co-Chair
Gus H. Smith, PhD, Co-Chair
Prof. Timothy Sherwood
Prof. Ben Hardekopf

June 2026

The Thesis of Sijie Kong is approved.

Prof. Timothy Sherwood

Prof. Ben Hardekopf

Dr. Gus H. Smith, Committee Co-Chair

Prof. Jonathan Balkind, Committee Co-Chair

March 2026

A Principled Approach to Technology Mapping

Copyright © 2026

by

Sijie Kong

To my parents.

Acknowledgements

I am deeply grateful to everyone who supported me during my master’s study and helped shape this thesis. The following names are listed in alphabetical order by last name; each person is equally important.

I would like to thank Jonathan Balkind for bringing me into the lab and into the broader PL4HW community. His guidance has supported me both academically and personally. I especially appreciate his willingness to listen carefully to my wilder ideas, think through them seriously, and help me identify the most promising directions to pursue. He has also been generous with his time and support during my PhD applications.

I am very thankful to Zachary Sisco, the first PhD student mentor I worked closely with. He introduced me to a genuinely exciting project and helped guide me into research at a time when I still felt like a latecomer. His mentorship made research feel approachable, and I am also grateful for his support during my PhD applications.

I am very grateful to Gus Smith for his mentorship. He gave me the opportunity to build on his early prototype work and invested an extraordinary amount of time and care into helping me grow as a researcher. I also appreciate his encouragement and support during my PhD applications.

I would like to thank Jingtao Xia for being a wonderful collaborator. He provided invaluable support on theory and writing, and repeatedly helped turn rough ideas into something sound and rigorous. He also played a meaningful role in sparking my interest in AI.

I am grateful as well to the members of the UCSB Arch Lab and PL Lab for creating a welcoming and intellectually exciting environment. I learned a great deal from the conversations and group meetings.

I would also like to thank the people I met at UW for their kindness and support.

Abstract

A Principled Approach to Technology Mapping

by

Sijie Kong

Technology mapping translates hardware designs into the concrete primitives provided by target platforms such as simulators, FPGAs, and ASICs. In current flows, this process is often brittle, syntax-driven, and heavily phase-ordered: designers duplicate implementations across technologies, while tools depend on narrowly structured inputs and carefully tuned pass sequences. This thesis argues that technology mapping should instead be approached in a principled way: target technologies should be exposed through explicit semantic abstractions, and compilation should reason about equivalence across representations rather than committing early to a single syntactic form.

The first part of the thesis develops this argument for memories through *Memo*, a language and compilation framework built around *fungible memories*. Fungible memories expose the semantic structure needed for automated mapping while hiding target-specific implementation details, allowing designers to specify a memory once and compile it across diverse technologies. In this setting, the thesis shows that making technology-relevant memory structure explicit in the source representation can replace hand-written, target-specific implementations with systematic compilation.

The second part turns to general hardware optimization and mapping through *Nextmap*, an equality-saturation-based framework that performs optimization and technology mapping simultaneously across word-level and bit-level representations. To support this, the thesis introduces *semantic e-graphs*, which extend traditional e-graphs with semantic identifiers that efficiently capture domain-specific equivalence and bridge multiple abstrac-

tion levels. Here, the central idea is to keep multiple equivalent representations available during compilation, avoiding the losses imposed by phase-ordered flows and exposing mappings that are invisible to syntax-driven tools.

Contents

Abstract	vi
1 Introduction	1
1.1 Permissions and Attributions	5
2 Fungible Memories	6
2.1 Motivating Example	6
2.2 Fungible Memories	8
2.3 The MEMO Compiler	17
2.4 The MEMO Decompiler	22
2.5 Implementation	26
2.6 Evaluation	29
3 Nextmap and Semantic E-Graphs	37
3.1 Semantic E-Graphs	38
3.2 Semantic E-Graphs for Hardware	50
3.3 NEXTMAP	55
3.4 Evaluation	59
4 Conclusion	67

Chapter 1

Introduction

Technology mapping is the stage of compilation that turns an abstract hardware design into the concrete building blocks available in a target flow. In EDA, this means choosing how the design will be realized using the primitives of a particular implementation setting: for simulation, an executable behavioral model; for FPGAs, resources such as lookup tables, carry chains, block memories, and DSP blocks; and for ASICs, standard cells and memory macros provided by a fabrication process. Although this problem appears throughout digital design, in practice it is often handled in narrow, technology-specific, and largely ad hoc ways. Designers routinely restructure code to satisfy vendor templates, duplicate implementations across simulation, FPGA, and ASIC targets, and rely on long sequences of brittle optimization passes whose effectiveness depends heavily on ordering. As a result, mapping is frequently treated not as a semantic compilation problem, but as a collection of special cases. This thesis argues that technology mapping should instead be approached in a principled way: target technologies should be exposed through explicit semantic abstractions, and compilation should reason about equivalence across representations rather than committing early to a single syntactic form.

A central difficulty is that modern hardware targets impose constraints that are

both diverse and deeply structural. Memories, for example, differ in port structure, width, height, timing behavior, masking support, and forwarding semantics. Likewise, arithmetic and logical operators may map efficiently only when a design is represented at the right level of abstraction: some optimizations are natural at the word level, while others are only visible at the bit level. Conventional flows cope with this diversity by forcing designers or tools to choose a representation too early, then attempting to recover lost opportunities through carefully tuned passes. This creates two recurring problems. First, the same hardware behavior must often be rewritten multiple times to target different technologies. Second, optimization and mapping become separated into phases, so improvements exposed in one stage may be hidden again before the next.

Figure 1.1 illustrates both issues on a small DSP-style arithmetic kernel. The expressions $(a + b) * c + d$ and $a * c + b * c + d$ are semantically equivalent, but an FPGA flow may map the first to a single DSP block with built-in pre-add and post-add structure while mapping the second to a larger collection of multipliers and logic. Likewise, two pipelined implementations of the same multiply-accumulate can differ only in register placement, yet only one may directly match a packed DSP pattern unless the tool can retime across arithmetic boundaries. These examples capture why current flows are often brittle: they succeed or fail not only on the meaning of a design, but on whether that meaning is presented in the particular syntactic and phase structure a tool expects.

This thesis develops a semantic view of technology mapping that addresses both problems. Its key claim is that mapping should be driven by representations that preserve the meaning of a design while exposing the structural choices needed by target technologies. Under this view, a compiler should not merely match surface syntax against templates. Instead, it should reason over families of equivalent implementations and select those that best satisfy a target’s constraints. Such an approach naturally connects programming-language ideas—abstraction, equivalence, and rewriting—with core problems in hardware

Equivalent arithmetic forms

```
// DSP-friendly form
assign y = (a + b) * c + d;
```

```
// Equivalent by distributivity
assign y = a * c + b * c + d;
```

Both descriptions simulate identically, but an FPGA flow may map the first to one DSP block with a pre-adder and post-adder, while the second may be realized as two multipliers plus extra logic unless the tool can recover the shared structure.

Equivalent pipeline placements

```
// Registers placed around the MAC
always_ff @(posedge clk) begin
    a_r <= a; b_r <= b; c_r <= c;
    y    <= a_r * b_r + c_r;
end
```

```
// Same one-cycle behavior after retiming
always_ff @(posedge clk) begin
    m_r <= a * b;
    c_r <= c;
    y    <= m_r + c_r;
end
```

These pipelines can implement the same sequential behavior, but only some placements expose a direct packed multiply-accumulate to a DSP mapper. If optimization and mapping are separated, an early extraction can prevent the necessary retiming from being considered later.

Figure 1.1: A small DSP-oriented example illustrating brittle technology mapping. **Left:** semantically equivalent arithmetic expressions can map very differently when recognition depends on surface syntax. **Right:** semantically equivalent pipeline organizations can map differently when optimization and technology mapping are forced into separate phases.

compilation.

The first part of the thesis explores this idea for memories. I present a language-level abstraction, *fungible memories*, that gives designers a single, generic way to describe memory behavior while deferring technology-specific realization to the compiler. Rather than forcing a developer to handcraft separate implementations for simulation, FPGA block memories, and ASIC SRAM macros, the abstraction captures the essential semantics of a memory system and allows compilation to derive legal target-specific organizations automatically. This perspective reframes memory mapping as the compilation of a rich algebraic object, not the recognition of a vendor-specific coding pattern. By making memory structure explicit in the source representation, the compiler can systematically split,

merge, and transform memories to satisfy implementation constraints while preserving behavior.

The second part of the thesis extends the same philosophy to a broader class of mapping problems through equality saturation. I present *Nextmap*, a technology-mapping framework that runs optimization and mapping simultaneously instead of as isolated stages. The core challenge is representation: effective mapping must connect word-level arithmetic expressions, where high-level algebraic structure is visible, with bit-level logic, where low-level implementation choices are made. Maintaining this connection using syntactic rewrites alone is inefficient and incomplete. To address this, the thesis introduces *semantic e-graphs*, which enrich equality saturation with semantic identifiers that capture domain-specific equivalence. This mechanism allows the compiler to bridge word- and bit-level views of hardware efficiently, retain many equivalent forms at once, and expose mappings that would be lost in conventional phase-ordered flows.

Taken together, these systems support a unified view: principled technology mapping requires semantic representations that make implementation choices explicit without collapsing equivalence too early. In the memory setting, this means elevating technology-relevant structure into the source language and compiling it algebraically. In the logic and arithmetic setting, this means preserving multiple equivalent implementations in a shared optimization space and using semantic equality to connect levels of abstraction. Across both settings, the goal is the same: to replace brittle, syntax-driven mapping flows with compilation techniques that are systematic, retargetable, and grounded in the meaning of the design.

This thesis makes three main contributions. First, it argues for a semantic and representation-centric formulation of technology mapping. Second, it demonstrates this formulation in the domain of memories through a language and compilation framework that separates behavioral specification from technology realization. Third, it demonstrates

the same formulation in general hardware optimization and mapping through equality saturation and semantic e-graphs, enabling multiple synthesis stages to be handled in a single reasoning framework. More broadly, the thesis shows that technology mapping is not merely a backend engineering task, but a rich compilation problem that benefits from principled abstractions and formal reasoning.

The remainder of this thesis is organized as follows. Chapter 2 presents the memory language and compilation framework, focusing on how fungible memories provide a portable semantic interface for memory implementation. Chapter 3 presents Nextmap and the semantic e-graph machinery that supports simultaneous optimization and mapping across abstraction levels, including both the implementation details and the evaluation. Chapter 4 concludes by returning to the broader case for semantic, retargetable hardware compilation.

1.1 Permissions and Attributions

1. The content of chapter 2 has previously appeared in PLDI. The content of chapter 3 has previously appeared in PLDI.

Chapter 2

Fungible Memories

This chapter is based on joint work led by Zachary D. Sisco, who initiated the project and designed the memory language and overall framework. My contributions include significant implementation work on the memory decompiler and on the dynamic-programming algorithm for memory mapping with technology constraints. The underlying work was carried out with a broader set of collaborators; a fuller account of individual contributions appears in Zachary D. Sisco’s dissertation. This work previously appeared in PLDI 2026 under the title “Fungible Memories for Automated Technology Mapping and Retargeting.” The presentation here is adapted for the thesis structure.

2.1 Motivating Example

We introduce the problem through a small but representative example. Consider a developer implementing an SoC core with a floating-point register file: a 32-entry memory of 32-bit values with three read ports and one write port, as required by instructions that read three operands Foundation [2025].

The developer needs this register file to work across three targets: software simulation,

```

1  'ifdef VERILATOR
2  reg [31:0] regfile[31:0];
3  ...
4  'endif
5
6  'ifdef ULTRASCALE
7  bram_2rw_wrapper #(...) regfile1 (...);
8  bram_2rw_wrapper #(...) regfile2 (...);
9  bram_2rw_wrapper #(...) regfile3 (...);
10 ...
11 'endif
12
13 'ifdef TSMC28
14 tsmc28_1r1w_d32_w32_2_2rf mem1 (...);
15 tsmc28_1r1w_d32_w32_2_2rf mem2 (...);
16 tsmc28_1r1w_d32_w32_2_2rf mem3 (...);
17 ...
18 'endif

```

(a)

```

reg [31:0] regfile[31:0];
// Read logic
assign rs1_data_o = regfile[rs1];
assign rs2_data_o = regfile[rs2];
assign rs3_data_o = regfile[rs3];
// Write logic
always @(posedge clk)
  if (w_valid_i)
    regfile[rd] <= w_data_i;

```

(b)

```

bram_2rw_wrapper
#(.D_WIDTH(32), .A_WIDTH(5))
regfile1 (.CLK(clk), .WEA(0),
.WEB(w_valid_i), .ADDRB(rs1), .ADDRB(rd),
.ENA(rs1_valid_i), .ENB(w_valid_i),
.DOUTA(rs1_data_o), .DINB(w_data_i));

```

(c)

Figure 2.1: **(a)** An implementation of the register file with three read ports and one write port, using `ifdef` blocks to support three technologies. **(b)** Behavioral implementation suitable for simulation which fills in lines 2–3 of **(a)**. **(c)** Module instantiation of a BRAM IP block from an FPGA vendor used in lines 7–9 of **(a)**.

Xilinx UltraScale FPGA, and an ASIC flow on TSMC 28. Figure 2.1a shows a common solution in Verilog: keep the interface fixed, but use `ifdef` blocks to swap in different implementations for each target.

For simulation, the behavioral Verilog in Figure 2.1b is sufficient, e.g. with Verilator Snyder [2024]. For deployment, however, the same logical memory must be re-expressed in technology-specific forms. On the FPGA, the target memory primitive is a dual-port BRAM, so the 3-read, 1-write register file must be decomposed into multiple `2rw` blocks. For ASIC, the available SRAM primitive may instead require splitting the design into several `1r1w` memories. The result is multiple implementations of the same memory, each with different module instances, parameters, and wiring.

This style of implementation is brittle. One logical memory becomes several hand-written implementations, and every new target or design change forces the developer to revisit those target-specific encodings. MEMO addresses this problem by letting the

developer write one generic memory description and compile it into the forms required by each technology.

2.2 Fungible Memories

In this section, we formalize the notion of a fungible memory. A fungible memory captures the shape, connections, and “portedness” of memory blocks, along with optimizations expressible abstractly or in bit-level logic. The formal rewrite rules show how to transform fungible memories into different configurations while preserving behavioral equivalence. “Fungibility” enables designers to specify memories as high-level abstractions that map to technology-specific implementations.

Figure 2.2 presents the grammar for fungible memories. A *fungible memory* denotes an $n \times m$ memory block with any combination of read, write, and read–write ports. Fungible memories compose in parallel with each other ($\oplus$), and their ports may be shared in the read/write logic of other fungible memories. Each port type has its own constructor for its specific inputs and outputs. Memories and ports may be applied with boolean functions that express various behaviors and optimizations commonly found in SoC memory blocks, such as write-to-read forwarding, asynchronous reads, write masks, and latch-last read.

2.2.1 Rewrite Rules

Figure 2.3 presents the formal algebraic rules for transforming fungible memories modulo equivalence. The rules cover transformations over memories and ports and can be read either forwards (left-to-right) or backwards (right-to-left). There are five kinds of fungible memory rules: split and merge by (1) read ports, (2) write ports, (3) width, and (4) height, and (5) function application (using write-to-read forward as an example). When reading the rules, “splitting” happens from the left-hand side of the equation to the

$$\begin{aligned}
\langle mem \rangle &::= \langle mem \rangle \oplus \langle mem \rangle \mid \text{Mem}(\langle width \rangle, \langle height \rangle, \langle port \rangle^+) \circ \text{Fwd}^? \\
\langle port \rangle &::= \langle r \rangle \mid \langle w \rangle \mid \langle rw \rangle \\
\langle r \rangle &::= \text{Read}(\langle addr \rangle, \langle data_r \rangle) \circ \text{Async}^? \\
\langle w \rangle &::= \text{Write}(\langle en \rangle, \langle addr \rangle, \langle data_w \rangle, \langle mask \rangle^?) \\
\langle rw \rangle &::= \text{ReadWrite}(\langle en \rangle, \langle addr \rangle, \langle data_r \rangle, \langle data_w \rangle, \langle mask \rangle^?)
\end{aligned}$$

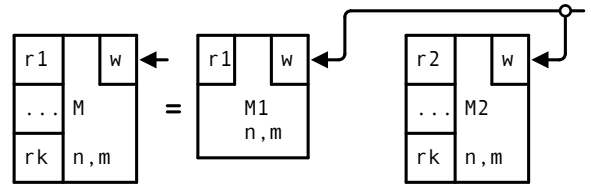
$$\langle en \rangle, \langle addr \rangle, \langle data_r \rangle, \langle data_w \rangle, \langle mask \rangle \in \mathbb{B}^k, \quad \langle width \rangle, \langle height \rangle, k \in \mathbb{N}$$

Figure 2.2: The grammar for fungible memories. The $\oplus$ operator denotes the parallel composition of two memories. **Fwd** and **Async** are supported features to denote a read–write forwarding optimization and asynchronous reads, respectively. $^?$ denotes an optional term.

right-hand side, while “merging” starts from the right-hand side to the left. Additionally, ports can be cast between read-only or write-only ports into shared read–write ports (by setting the relevant enable bit to zero).

The rules work at two levels: the level of memory blocks, and of ports connecting with memory blocks. While the symmetry of splitting and merging a memory appears obvious at the memory block level, there are intricate details at the port level, along with optimizations, that must be maintained to guarantee an equivalent memory system. To provide an intuition for the transformation each rule performs, we illustrate smaller examples of each rule type.

Read port rule. We start with the read port split/merge rule. Here, the left-hand side represents an n -by- m memory M with k -read ports ($r1, \dots, rk$) and one write port w . The right-hand side shows M split into



two memories, where $M1$ has only one read port, $M2$ has $(k - 1)$ read ports, and they both share the interfaces from the same write port.

Read Port Split/Merge

$$\text{Mem}(n, m, (r_i)_{i=1}^k, \vec{w}) = \text{Mem}(n, m, (r_i)_{i=1}^{k-1}, \vec{w}) \oplus \text{Mem}(n, m, r_k, \vec{w})$$

Write Port Split/Merge

$$\text{Mem}(n, m, (r_i)_{i=1}^k \circ \text{Async}, w_1, w_2) = \text{Mem}(n, m, (r_i)_{i=1}^k, r_{k+1}, w'_1) \oplus \text{Mem}(n, m, (r_i)_{i=1}^k, r_{k+2}, w'_2),$$

$$\begin{aligned} \text{where } r_{k+1} &= \text{Read}(\text{addr}_{r_{k+1}}, \text{data}_{r_{k+1}}) \circ \text{Async}, \quad r_{k+2} = \text{Read}(\text{addr}_{r_{k+2}}, \text{data}_{r_{k+2}}) \circ \text{Async}, \\ w'_1 &= \text{Write}(\text{en}_{w_1}, \text{addr}_{w_1}, \text{data}_{w_1}), w'_1 = \text{Write}(\text{en}_{w_1}, \text{addr}_{w_1}, \text{Select}(\text{data}_{r_{k+2}}, \text{data}_{w_1})), \\ w'_2 &= \text{Write}(\text{en}_{w_2}, \text{addr}_{w_2}, \text{data}_{w_2}), w'_2 = \text{Write}(\text{en}_{w_2}, \text{addr}_{w_2}, \text{Select}(\text{data}_{r_{k+1}}, \text{data}_{w_2})) \end{aligned}$$

Width Split/Merge

$$\text{Mem}(n, m, (r_i)_{i=1}^k, (w_j)_{j=1}^l) = \text{Mem}(n', m, (r'_i)_{i=1}^k, (w'_j)_{j=1}^l) \oplus \text{Mem}(n - n', m, (r''_i)_{i=1}^k, (w''_j)_{j=1}^l)$$

$$\begin{aligned} \text{where } r_i &= \text{Read}(\text{addr}_{r_i}, \text{data}_{r_i}) = \text{Read}(\text{addr}_{r_i}, \text{data}'_{r_i} ++ \text{data}''_{r_i}), \\ r'_i &= \text{Read}(\text{addr}_{r_i}, \text{data}'_{r_i}) = \text{Read}(\text{addr}_{r_i}, \text{data}_{r_i}[n' - 1 : 0]), \\ r''_i &= \text{Read}(\text{addr}_{r_i}, \text{data}''_{r_i}) = \text{Read}(\text{addr}_{r_i}, \text{data}_{r_i}[n - 1 : n']), \\ w_j &= \text{Write}(\text{en}_{w_j}, \text{addr}_{w_j}, \text{data}_{w_j}) = \text{Write}(\text{en}_{w_j}, \text{addr}_{w_j}, \text{data}'_{w_j} ++ \text{data}''_{w_j}), \\ w'_j &= \text{Write}(\text{en}_{w_j}, \text{addr}_{w_j}, \text{data}'_{w_j}) = \text{Write}(\text{en}_{w_j}, \text{addr}_{w_j}, \text{data}_{w_j}[n' - 1 : 0]), \\ w''_j &= \text{Write}(\text{en}_{w_j}, \text{addr}_{w_j}, \text{data}''_{w_j}) = \text{Write}(\text{en}_{w_j}, \text{addr}_{w_j}, \text{data}_{w_j}[n - 1 : n']) \end{aligned}$$

Height Split/Merge

$$\text{Mem}(n, 2^m, (r_i)_{i=1}^k, (w_j)_{j=1}^l) = \text{Mem}(n, 2^{m-1}, (r'_i)_{i=1}^k, (w'_j)_{j=1}^l) \oplus \text{Mem}(n, 2^{m-1}, (r''_i)_{i=1}^k, (w''_j)_{j=1}^l)$$

$$\begin{aligned} \text{where } r_i &= \text{Read}(\text{addr}_{r_i}, \text{data}_{r_i}) = \text{Read}(\text{addr}_{r_i}, \text{Mux}(\text{addr}_{r_i}, \text{data}'_{r_i} ++ \text{data}''_{r_i})), \\ r'_i &= \text{Read}(\text{addr}'_{r_i}, \text{data}'_{r_i}) = \text{Read}(\text{addr}_{r_i}[m - 2 : 0], \text{data}_{r_i}), \\ r''_i &= \text{Read}(\text{addr}''_{r_i}, \text{data}''_{r_i}) = \text{Read}(\text{addr}_{r_i}[m - 2 : 0], \text{data}_{r_i}), \\ w_j &= \text{Write}(\text{en}_{w_j}, \text{addr}_{w_j}, \text{data}_{w_j}), \\ w'_j &= \text{Write}(\text{en}'_{w_j}, \text{addr}'_{w_j}, \text{data}'_{w_j}) = \text{Write}(\text{en}_{w_j} \wedge \neg \text{addr}_{w_j}[m - 1], \text{addr}_{w_j}[m - 2 : 0], \text{data}_{w_j}), \\ w''_j &= \text{Write}(\text{en}''_{w_j}, \text{addr}''_{w_j}, \text{data}''_{w_j}) = \text{Write}(\text{en}_{w_j} \wedge \text{addr}_{w_j}[m - 1], \text{addr}_{w_j}[m - 2 : 0], \text{data}_{w_j}) \end{aligned}$$

Forwarding (function application)

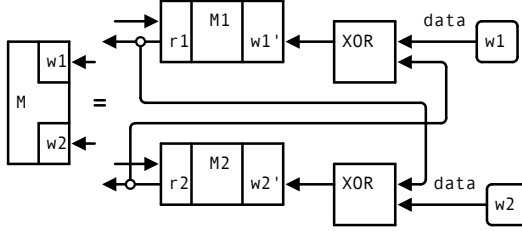
$$\text{Mem}(n, m, (r_i)_{i=1}^k, \vec{w}) \circ \text{Fwd} = \text{Mem}(n, m, (r'_i)_{i=1}^k, \vec{w})$$

$$\text{where } r_i = \text{Read}(\text{addr}_{r_i}, \text{data}_{r_i}), \quad r'_i = \text{Read}(\text{addr}_{r_i}, \text{Fwd}(\text{addr}_{r_i}, \text{data}_{r_i}, \vec{w}))$$

Cast Ports

$$\begin{aligned} \text{Read}(\text{addr}, \text{data}_r) &= \text{ReadWrite}(0, \text{addr}, \text{data}_r, _) \\ \text{Write}(\text{en}, \text{addr}, \text{data}_w) &= \text{ReadWrite}(\text{en}, \text{addr}, _, \text{data}_w) \\ \text{ReadWrite}(\text{en}, \text{addr}, \text{data}_r, \text{data}_w) &= (\text{Read}(\text{addr}, \text{data}_r), \text{Write}(\text{en}, \text{addr}, \text{data}_w)) \end{aligned}$$

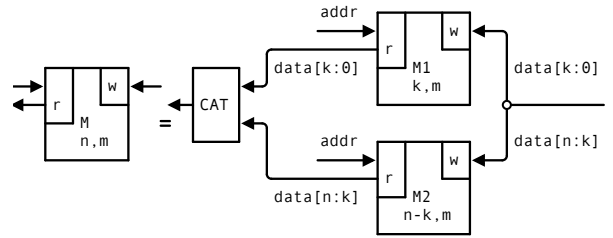
Figure 2.3: Fungible memory rules, using the convention that r is a Read port, and w is a Write port. $\vec{p}$ is shorthand for a sequence of ports where the sequence length is not relevant to the rule.

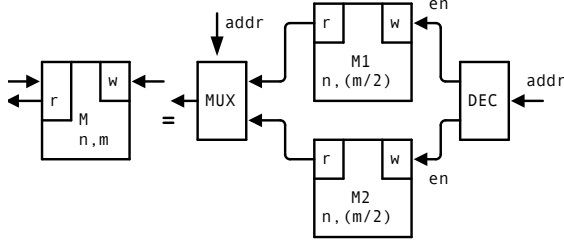


Write port rule Write port splitting must also maintain data consistency. There are many strategies LaForest and Steffan [2010], Laforest et al. [2014], Abdelhadi and Lemieux [2016]; here we show one. For space, we only illustrate the two write ports $w1$ and $w2$ and ignore any existing read ports. The new split memories $M1$ and $M2$ each have a newly

created write port $w1'$ and $w2'$, respectively. There are also two additional read ports $r1$ and $r2$ which forward a read data value to the selection function. The selection function is implemented as XOR, left generic in the formal rule in fig. 2.3. It selects the most up-to-date data value between the split memory blocks, leveraging the identity that $x \text{ XOR } y \text{ XOR } y = x$.

Width rule. Merging and splitting can also occur over the “width” and “height” dimensions (data and addresses). Rule (3) illustrates splitting a larger memory M by k , resulting in a k -width memory and $(n - k)$ -width memory. The address and enable interfaces for the write port remain unchanged; we omit them for space. CAT stands for the *concatenation* of two values.

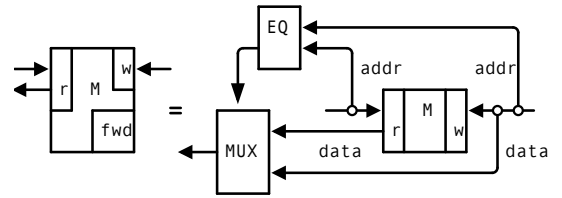




Height rule Splitting over the height of the memory uses additional logic to choose which memory block to read from (write to) based on the address. This rule splits the height of fungible memories over powers of 2 because memory blocks are only addressable in powers of 2. The MUX cell multiplexes the read-data outputs of M1 and M2 based on the address. The DEC cell is a decoder that enables the write port of either M1 or M2.

Function application Fungible memories

may be applied with functions that express behaviors and optimizations commonly found in SoC memory blocks, such as write-to-read forwarding, latch-last read, and asynchronous reads. We illustrate this via an example where a memory has write-to-read forwarding enabled (**fwd**), but the target technology does not support forwarding, so it is “pulled out” as separate logic.



2.2.2 Proof of Correctness

Formalization. We model the state of a fungible memory with word width w and address width k as a total function $\sigma : \mathbb{B}^k \rightarrow \mathbb{B}^w$, mapping each of the 2^k addresses to a w -bit data word. A write port $p = (en, addr, data)$ consists of a one-bit enable signal $en \in \mathbb{B}^1$, an address $addr \in \mathbb{B}^k$, and a data value $data \in \mathbb{B}^w$. The application of a single

write port to a state is defined pointwise:¹

$$\text{applyWrite}(\sigma, p)(a) = \begin{cases} data & \text{if } en = 1 \text{ and } a = addr, \\ \sigma(a) & \text{otherwise.} \end{cases}$$

A list of write ports $(p_1, \dots, p_n)$ is applied by left-folding **applyWrite** over the list. We define a fungible memory M inductively as either a *leaf* (σ, R, W) —carrying a state σ , read ports R , and write ports W —or a parallel composition $M_1 \oplus M_2$ of two fungible memories. A single step of evaluation applies each leaf’s write-port list to its state; parallel-composed memories step independently.

Behavioral Equivalence. We define equivalence by first *flattening* a memory into a list of leaf descriptors (σ_i, W_i) via a function **flatten**, which maps a leaf to a singleton list and parallel composition to list concatenation. Two leaf descriptors (σ_1, W_1) and (σ_2, W_2) are *leaf-equivalent* if, whenever $\sigma_1 = \sigma_2$, we have $\text{applyWrites}(\sigma_1, W_1)(a) = \text{applyWrites}(\sigma_2, W_2)(a)$ for every address a . Two memories M_1 and M_2 are then *behaviorally equivalent* when **flatten**(M_1) and **flatten**(M_2) are pointwise leaf-equivalent. Because **flatten** maps $\oplus$ to list concatenation, associativity of $\oplus$ reduces to associativity of concatenation. Note that read ports can be ignored since equivalence concerns only the write-induced state mutation.

Preservation of Equivalence. We verify each rewrite rule in Figure 2.3 by proving that the structural transformation preserves behavioral equivalence. *Read-port splitting* is immediate, since **flatten** ignores read ports. *Write-port splitting* shows that partitioning a write-port list across two parallel leaves, with a last-write-wins multiplexer selecting

¹Without loss of generality, the formalism excludes write masks and read-write ports.

which half owns each write, preserves the folded semantics of **applyWrites**.² *Width splitting* decomposes a w -bit memory into high and low slices by showing that **applyWrites** commutes with bitvector extraction $(\cdot)[i+l-1 : i]$, then recombining the slices via $data = data[n-1 : n'] ++ data[n'-1 : 0]$. *Height splitting* partitions the 2^k -row address space by the most significant bit (MSB) of the address: a write to the “upper half” has a MSB of 1 and targets only the upper sub-memory, disabling the write port of the “lower half.” The correctness of this routing rests on the lemma that two k -bit addresses sharing the same MSB are equal if and only if their lower $(k-1)$ bits coincide.

We mechanically verified all rules in the Lean 4 proof assistant de Moura and Ullrich [2021]. The proofs follow a uniform methodology: reduce each equivalence to a pointwise equality $\mathbf{applyWrites}(\sigma, W)(a) = \mathbf{applyWrites}(\sigma', W')(a)$ for an arbitrary address a , established by induction on the write-port list W with the state σ generalized. Case analysis on boolean conditions (the enable bit, MSB comparisons) and standard bitvector identities for slicing and reconstruction supply the remaining reasoning.

2.2.3 An Example

To see fungible memories in action, we return to the register file example from Section 2.1 and consider its formalization. When the developer writes the custom mapping from the original **3r1w** register file to the three-**1r1w** memories for the ASIC technology, they implicitly manipulate the configuration of the register file. Instead, if we use the rules from Figure 2.3, we *explicitly* transform the memory according to equality-preserving rewrites. For the register file, we formally express this transformation as two applications of the read-port splitting rule:

²As noted in Section 2.2.1, different selection strategies can be used in the resulting implementation.

$$\begin{aligned}
\text{Mem}(32, 32, r_1, r_2, r_3, w) &= \text{Mem}(32, 32, r_1, r_2, w) \oplus \text{Mem}(32, 32, r_3, w) \\
&= \text{Mem}(32, 32, r_1, w) \oplus \text{Mem}(32, 32, r_2, w) \oplus \text{Mem}(32, 32, r_3, w),
\end{aligned}$$

$$\begin{aligned}
\text{where } r_1 &= \text{Read}(\text{rs1}, \text{rs1_data_o}), r_2 = \text{Read}(\text{rs2}, \text{rs2_data_o}), \\
r_3 &= \text{Read}(\text{rs3}, \text{rs3_data_o}), w = \text{Write}(\text{w_valid_i}, \text{rd}, \text{w_data_i})
\end{aligned}$$

For the FPGA technology, we follow the same transformation, but then cast each **1r1w** memory to a **2rw**, following the first and second port casting rules, for the read and write port, respectively. The following demonstrates casting for (r_1, w) :

$$\begin{aligned}
\text{Mem}(32, 32, r_1, w) &= \text{Mem}(32, 32, rw_1, rw') \text{ where } r_1 = rw_1, \text{ and } w = rw', \\
rw_1 &= \text{ReadWrite}(0, \text{rs1}, \text{rs1_data_o}, _), \\
rw' &= \text{ReadWrite}(\text{w_valid_i}, \text{rd}, _, \text{w_data_i})
\end{aligned}$$

This same casting procedure will be repeated for $\text{Mem}(32, 32, r_2, w)$ and $\text{Mem}(32, 32, r_3, w)$ to form three memories composed in parallel each with two read-write ports: $\text{Mem}(32, 32, rw_1, rw') \oplus \text{Mem}(32, 32, rw_2, rw') \oplus \text{Mem}(32, 32, rw_3, rw')$.

2.2.4 Extending an HDL with Fungible Memories

To use fungible memories in real hardware designs, we extend the PyRTL HDL to support specifying fungible memories. We introduce a new class into PyRTL called `FungibleMemory`, which accepts a list of ports, specifies its width and height, and configures a number of features (e.g., write-to-read forwarding). The memory ports (inputs and

Parallel Composition $T[m_1 \oplus m_2] ::= T[m_1], T[m_2]$

Memory Creation

$T[\text{Mem}(w, h, \vec{p})] ::= \text{fresh } m \text{ where } m = \text{MemBlock}(w, h), T[p_1](m), \dots, T[p_k](m)$

$T[\text{Mem}(w, h, \vec{p}) \circ \text{Fwd}] ::= \text{fresh } m \text{ where } m = \text{MemBlock}(w, h), T_{\text{Fwd}}[p_1](m), \dots, T_{\text{Fwd}}[p_k](m)$

Write Port

$T[\text{Write}(en, addr, data_w)](m) ::= m[addr] \ll= \text{MemBlock.EnabledWrite}(data_w, en)$

$T[\text{Write}(en, addr, data_w, mask)](m) ::= \text{fresh } v \text{ where } v = m[addr],$

$m[addr] \ll= \text{MemBlock.EnabledWrite}((data_w \& mask) \mid (v \& \sim mask), en)$

Read Port

$T[\text{Read}(addr, data_r)](m) ::= \text{fresh } r \text{ where } r = \text{Register}(\text{bitwidth}=\text{len}(addr)),$
 $r.\text{next} \ll= addr, data_r \ll= m[r]$

$T[\text{Read}(addr, data_r) \circ \text{Async}](m) ::= data_r \ll= m[addr]$

$T_{\text{Fwd}}[\text{Read}(addr, data_r)](m) ::= \text{fresh } r \text{ where } r = \text{Register}(\text{bitwidth}=\text{len}(addr)),$
 $r.\text{next} \ll= addr, v_0 = m[r], data_r \ll= v_n$

for $i = 1, \dots, n$: $v_i = \text{pyrtl.select}(en_i \& (r == wa_i), wd_i, v_{i-1},)$

$T_{\text{Fwd}}[\text{Read}(addr, data_r) \circ \text{Async}](m) ::= v_0 = m[addr], data_r \ll= v_n$

for $i = 1, \dots, n$: $v_i = \text{pyrtl.select}(en_i \& (addr == wa_i), wd_i, v_{i-1})$

where $(en_1, wa_1, wd_1), \dots, (en_n, wa_n, wd_n)$ are the write ports of m .

ReadWrite Port

$T[\text{ReadWrite}(en, addr, data_r, data_w)](m) ::= T[\text{Read}(addr, data_r)](m),$
 $T[\text{Write}(en, addr, data_w)](m)$

Figure 2.4: Formal translation rules to compile a fungible memory into a list of equivalent statements in PyRTL. $T[\cdot]$ defines the translation function; $T_{\text{Fwd}}[\cdot]$ is its forwarding variant.

outputs) are PyRTL `WireVector`-typed values.

The MEMO compiler then *generates the corresponding logic* obtained from refining the fungible memory given a particular configuration of ports and parameters. This process follows the same “elaboration-through-execution” strategy that HDLs like PyRTL and Chisel Bachrach et al. [2012] use.

Here, we formalize the rules that transform a given fungible memory into valid PyRTL code. Compilation proceeds by translating each fungible memory to a PyRTL `MemBlock`. Figure 2.4 presents the translation rules to produce a PyRTL memory from a given fungible memory. The `a <=< b` syntax denotes wire assignment in PyRTL; it “drives” `a` with `b`, connecting the two signals. The translation rules also handle specific memory semantics and optimizations that need to be compiled to compatible logic in PyRTL. For example, a PyRTL `MemBlock` does not support write-read forwarding, so this optimization maps to explicit logic that exists outside of the `MemBlock` (defined in the $T_{\text{Fwd}}[\cdot]$ translation function in Figure 2.4).

Following the example from Section 2.1, we specify the same `3r1w` register file in the extension of PyRTL (switching on the write-to-read forwarding optimization):

```
regfile = FungibleMemory(32, 32, fwd=True, [ Read(rs1, rs1_data_o),
                                             Read(rs2, rs2_data_o), Read(rs3, rs3_data_o), Write(en, rd, w_data) ])
```

With fungible memories integrated into PyRTL, we can elaborate any PyRTL design with fungible memories into a simulation-capable PyRTL netlist. However, up to this point, the MEMO compiler can only map fungible memories to technologies that have a one-to-one mapping. Next, we consider technology constraints in the MEMO compiler.

2.3 The Memo Compiler

Here we show how we use the fungible memory formalism to build a memory compiler, describing how the algebra of fungible memories enables automated memory mapping following technology constraints. By default, fungible memories specified in PyRTL can elaborate into simulation-capable netlists. Then, given a par-

ticular memory technology and its constraints, we describe an algorithm to automatically map the fungible memory in a way that satisfies those constraints.

Figure 2.5 illustrates the overall flow. The black squares are the fungible memories from the design. The rewrites manipulate and reconfigure their shapes at the “abstract” level such that they can be successfully mapped to the given technology (i.e., the star shape). Once the memory satisfies the technology’s constraints, MEMO passes it to the backend to emit code—for example, as module instances in Verilog or scripts for a particular hardware synthesis tool. Thus, fungible memories enable automated technology mapping via algebraic rewrites. Lastly, for the implementation of MEMO, we show how to support different technology backends. As we describe in Section 2.5, our implementation of MEMO supports five backends. We describe the necessary pieces for users to add backends for other memory technologies.

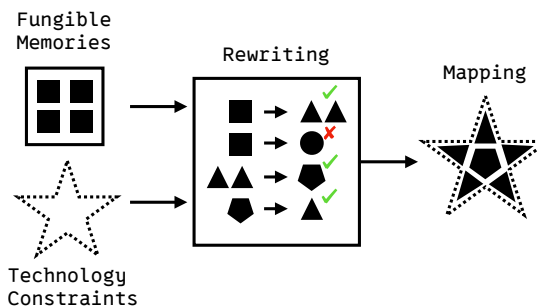


Figure 2.5: An illustration of the MEMO compiler flow.

2.3.1 Memory Mapping with Technology Constraints

Here we present a technique for transforming fungible memories according to technology constraints. The idea is, given a fungible memory and set of specific technology constraints, find an equivalent representation for the fungible memory that satisfies the constraints.

The algorithm has two inputs: a fungible memory and a set of available memory

technologies with their constraints. The technology library describes the specifications of each target memory, including data width, number of ports, different features like asynchronous read, write-read forwarding, etc., and a technology-defined cost. The output is a module with certain instantiations of the target memories and wrappers over them to satisfy the fungible memory interface.

Algorithm 1 describes how memory mapping works in two main phases. First, it applies a dynamic programming algorithm to compute the feasible port fitting plan for every memory with i read ports, j write ports, and k read-write ports given the technology library. The basis of this algorithm comes from the rules in Figure 2.3. The second phase is a greedy algorithm that deals with data width and other memory semantics. It selects memories from the technology library that have enough capacity to hold the data and satisfy the given features. It prefers memories with lower cost and those that already support optimizations such as write-read forwarding. Otherwise, it wraps the memories with additional logic; e.g., it adds forwarding logic when the target memory does not support it. Through dynamic programming, Algorithm 1 is guaranteed to find a solution and that solution will be optimal with respect to cost-per-bit (as specified in the technology's cost model).

Consider an example of mapping a fungible memory with 2 read ports and 2 write ports to a technology library with only one type of BRAM which has 2 read-write ports and costs 1, regardless of their capacity and features. First, the algorithm initializes a 3-dimensional array $f(i, j, k)$ to $+\infty$. Then, it sets $f(0, 0, 2) = 1$, since the target technology has 2 read-write ports. Then it starts from $f(2, 2, 0)$ and recursively searches for satisfying configurations. In order to implement $f(2, 2, 0)$, it needs to use two $f(3, 1, 0)$ by the write port splitting rule. For each $f(3, 1, 0)$, it again needs to use three $f(1, 1, 0)$ by the read port splitting rule where $f(1, 1, 0) = f(0, 0, 2)$. Finally, we have $f(2, 2, 0) = 6$, which means we need to use six 2-read-write-port BRAMs to implement this memory.

Algorithm 1 Port fitting dynamic programming algorithm.

Require: Number of read ports n_r , write ports n_w , read-write ports n_{rw} ,1: Technology list $tech$ with fields $(r, w, rw, cost_per_bit)$ **Ensure:** Minimum cost configuration of memories2: $n_p \leftarrow n_r + n_w + 2 \times n_{rw}$ 3: Initialize $f[i][j][k] \leftarrow (\infty, \text{None})$ for all $0 \leq i, j, k \leq n_p$ ▷ Initialization with technology options4: **for** each memory type t in $tech$ with (r, w, rw, c) **do**5: **for** $i = 0$ to r **do**6: **for** $j = 0$ to w **do**7: **for** $k = 0$ to rw **do**8: $f[i][j][k] \leftarrow \min(f[i][j][k], (c, t))$ 9: **for** $k = n_p$ down to 0 **do** ▷ Dynamic programming transitions10: **for** $i = 0$ to n_p **do**11: **for** $j = 0$ to n_p **do**12: **for** $m = 1$ to $i - 1$ **do** ▷ Split read ports13: $f[i][j][k] \leftarrow \min(f[i][j][k], (f[m][j][k].cost + f[i - m][j][k].cost, \text{"split_r"}))$ 14: **if** $k < n_p$ **then**15: **if** $i > 0$ **then** ▷ Cast read $\rightarrow$ read-write16: $f[i][j][k] \leftarrow \min(f[i][j][k], (f[i - 1][j][k + 1].cost, \text{"cast_r"}))$ 17: **if** $j > 0$ **then** ▷ Cast write $\rightarrow$ read-write18: $f[i][j][k] \leftarrow \min(f[i][j][k], (f[i][j - 1][k + 1].cost, \text{"cast_w"}))$ 19: **for** $k = 1$ to n_{rw} **do** ▷ Cast read-write $\rightarrow$ read + write20: $f[n_r][n_w][n_{rw}] \leftarrow \min(f[n_r][n_w][n_{rw}], (f[n_r + k][n_w + k][n_{rw} - k].cost, \text{"cast_rw"}))$ 21: Reconstruct memory configuration from $f[n_r][n_w][n_{rw}]$ 22: **return** Optimal cost and configuration

```

bram_1rw_wrapper
#(.DEPTH(%s), .ADDR_WIDTH(%s),
 .BITMASK_WIDTH(%s), .DATA_WIDTH(%s))
%s (.MEMCLK(%s), .RESET_N(%s),
 .CE(%s), .A(%s), .RDWEN(%s),
 .BW(%s), .DIN(%s), .DOUT(%s));

```

(a)

```

"xilinx": [ { "name": "xilinx_bram_2rw",
"cost-per-bit": 3,
"width": 36, "height": 1024,
"read_ports": 0, "write_ports": 0,
"read_write_ports": 2,
"features": [ "Forward", "Sync" ] }, ...

```

(b)

Figure 2.6: Examples of (a) a BRAM wrapper module and (b) metadata for technology mapping.

2.3.2 Extending Memo with Additional Backends

Besides PyRTL simulation, MEMO targets specific technologies via backends. Our implementation (Section 2.5) supports five backends for three technology platforms. To support additional backends, there are two required pieces. The first is a technology “template”; either instantiating the module in HDL code, or generating a script such as for Vivado Tcl or OpenRAM. Figure 2.6a shows an example of a Verilog template for a single-port BRAM wrapper module. Filling in the template requires mapping the relevant parts from the fungible memory. For example, mapping address and data widths to the `ADDR_WIDTH` and `DATA_WIDTH` parameters; mapping the address port to `A`, the write data to `DIN`, and the read data to `DOUT`.

Second, for mapping scenarios where the memory cannot be directly mapped to the specified technology, constraints metadata is required. The constraints can be derived readily from vendor manuals. Figure 2.6b is an example of metadata for a dual-port BRAM for a Xilinx FPGA.

Our methodology is applicable to other frontends (HDLs) and backends (memory technologies). The MEMO compiler can be extended to support a number of HDLs by interfacing with fungible memories. The underlying rewrites over fungible memories and automated mapping algorithm are intermediate to the HDL and target technology.

2.4 The Memo Decompiler

Here we present the MEMO decompiler with the goal of lifting memories in a gate-level netlist up to fungible memories. The input is a gate-level netlist with single-bit registers (D-flip flops or DFFs). As illustrated in Figure 2.7, we split the technique into four phases, each driven by rewrite rules in MEMO: **(1)** group registers by common enable signals; **(2)** lift read logic from gates to multiplexers to behavioral reads; **(3)** lift write logic from gates to decoders to behavioral writes; **(4)** lift behavioral reads and writes into a fungible memory. In this section, we present the set of rewrite rules that enable memory decompilation, and then how we use equality saturation to decompile memories from netlists with optimizations that obscure the logic. Figure 2.8 presents a selection of rewrite rules for memory decompilation. Other rules include standard boolean identities for rewriting combinational logic. In the following, we illustrate the key rewrite rules for each phase.

2.4.1 Register Aggregation

Because MEMO supports a multi-level hardware representation, we can use it to directly express designs in terms of gate-level logic as well as high-level abstractions. We start with the gate-level netlist. The first step is to group DFFs into wider registers. Sets of multi-bit registers of the same width will later serve as candidates for memory decompilation. To reduce the search space, we use the heuristic optimization of only considering DFFs with enable pins. This is not required; even if DFFs with enable pins are not specifically used, there is normally some logic which enables reading and writing to registers inside of a memory block for power saving.

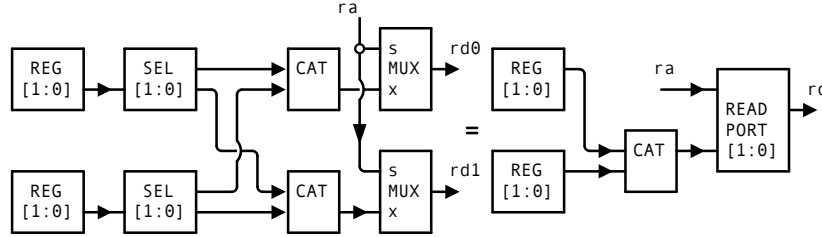
From Figure 2.8, the **Register Aggregate** rule searches for a group of DFFs that are connected to the same enable signal and then groups them into a single register

with a wider data input and output. The rule inserts concatenate (CAT) and select (SEL) components to maintain equivalence of inputs and outputs of the original DFFs. Figure 2.9a illustrates a grouping of two DFFs but this rule generalizes to any number of DFFs with a common enable.³

2.4.2 Read Port Identification

The key idea for identifying read ports is to identify a multiplexer tree that selects a bit from the same position of each register in a register group. To identify multiplexer trees, the **Mux Reduce** rule reduces two $n-1$ multiplexers and a 2-1 multiplexer into a single $2n-1$ multiplexer, as illustrated in Figure 2.9b.

A read port consists of multiplexer trees connected to the outputs of a register group:



The illustration demonstrates this rewrite rule for the simplest case of a 1-bit address with two grouped registers. Similar to multiplexer reduction, read port identification is inductive over the widths of the address and grouped registers.

2.4.3 Write Port Identification

Unlike read ports, address decoders form the basis for write ports and are conventionally implemented as and-inverter graphs. An n -bit address decoder is a family of functions $f_{target} : \mathbb{B}^n \rightarrow \mathbb{B}$, where $target \in \mathbb{B}^n$ is a constant bit vector representing the target address.

³This phase also groups by clock signal, but we omit the clock signals in the rewrite rules for ease of presentation.

The address decoder outputs 1 if and only if the input address matches the target address.

Single write port

In a single-write-port memory, the enable port of the i -th DFF is driven by:

$$f_i(\text{write_address}) \wedge \text{write_enable}.$$

Address decoders are recursively composed:

$$\begin{aligned} f_{[0]}([b]) &= \neg b, \quad f_{[1]}([b]) = b, \\ f_{\text{target}_1 ++ \text{target}_2}(bv_1 ++ bv_2) &= f_{\text{target}_1}(bv_1) \wedge f_{\text{target}_2}(bv_2) \end{aligned}$$

The base case is a single bit decoder. Similar to multiplexer reduction for read ports, we can simply apply reduction to the address decoder and recover the write address and enable signals.

Multiple write ports

The details become more subtle when it comes to memories with multiple write ports due to possible data races. In such cases, if more than one write port is enabled and writes to the same address, the final value in the memory is undefined i.e. [2001]. With HDLs such as Verilog, users either write a memory that has priorities among write ports, or omit priorities if they are not concerned about non-deterministic behavior.

In open-source EDA tools such as YosysWolf et al. [2013], each write port without user-defined priority is assigned a priority determined by the order of the write ports in the code. Suppose there are k write ports ordered as $w_1, w_2, \dots, w_k$ by their priorities. Then, the enable port of the DFF corresponding to the i th element in the memory is

driven by:

$$\bigwedge_j (f_i(\text{write_address}_j) \wedge \text{write_enable}_j),$$

where j , $1 \leq j \leq k$, is the write port index. Then, the data port at position i for write port j is:

$$d_{\{i,j\}} = \text{Mux}(f_i(\text{write_address}_j), d_{\{i,j+1\}}, \text{write_data}_i)$$

and $d_{\{i,k\}} = \text{write_data}_k$. In the write port identification process, the decompiler analyzes both enable and data ports for each DFF and checks the common parts of them, which are the address, enable and data signals for each write port.

2.4.4 Fungible Memory Extraction

With all registers grouped, and read and write ports identified, the last step is to extract the fungible memories. This phase proceeds simply by extracting sets of read and write ports which share the same registers and interfaces.

2.4.5 Memory Decompilation via Equality Saturation

The general memory decompilation technique presents an ideal scenario. In practice, however, it is insufficient to decompile memories in arbitrary netlists generated from other languages and logic synthesis tools. The reason is that logic synthesis optimizations obscure the original structure of the high-level memory block. Additional transformations may be necessary to reveal the structure, although it is not always obvious which sequence of rewrites will do so.

Solving this problem requires searching over a large space of possible transformations. Our memory decompilation technique overcomes this by using *equality saturation* Tate et al. [2009], a non-destructive term rewriting technique which uses the e-graph Nelson

[1980] data structure to group terms into equivalence classes. Instead of applying a fixed sequence of transformations over a program, equality saturation applies all possible rewrites over the whole program in a convergent process, extracting the “best” term according to a given cost function. Previous work has seen equality saturation used in other EDA tasks such as synthesis and optimization Pi et al. [2023], Coward et al. [2022], Wanna et al. [2023], Ustun et al. [2022], Smith et al. [2024c], Chen et al. [2024, 2025], Smith et al. [2024a], Coward et al. [2024], Hofmann et al. [2025]. We give more details on how we specifically apply and adapt this technique for memory decompilation in Section 2.5.

2.5 Implementation

Figure 2.10 illustrates the overall MEMO tool flow, with three use cases:

- (1) The user inputs a design expressed in PyRTL extended with fungible memories (as described in Section 2.2.4).
- (2) The user inputs a BLIF netlist Berkeley [1992] to the decompilation flow, to lift the registers in the netlist to fungible memories.
- (3) The user inputs an XCI XML file for a BRAM IP, which takes an expedited decompilation flow. After elaboration and mapping, MEMO targets one of five backends for three technology platforms: Base-

jump STL (BSG) and OpenRAM for ASIC; Vivado Tcl scripts and synthesizable BRAMs

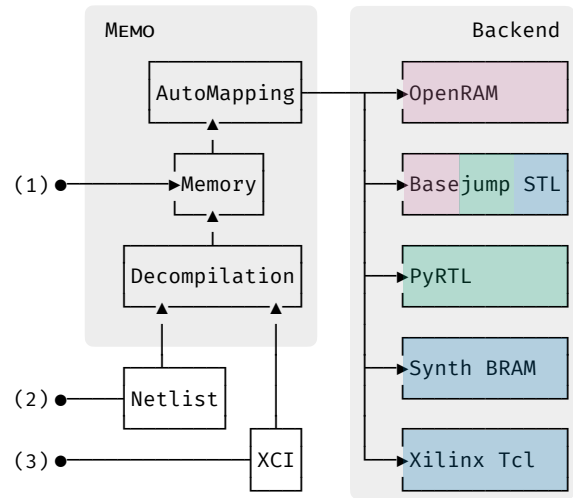


Figure 2.10: The MEMO tool flow: A user inputs either a design expressed in MEMO, a netlist, or XCI file. After rewriting and mapping, MEMO targets a particular backend. The backends are shaded by technology platform (pink = ASIC, blue = FPGA, green = simulation).

jump STL (BSG) and OpenRAM for ASIC; Vivado Tcl scripts and synthesizable BRAMs

for FPGA; and PyRTL for simulation.

Our implementation spans two parts: (1) the compiler, and (2) the memory decompiler. We implement the MEMO compiler in Python as an extension of the PyRTL compiler with fungible memories, along with the automated mapping algorithm described in Section 2.3.1. In a departure from conventional equality saturation (eqsat) frameworks, we implement the memory decompiler in SQL (`sqlite3` in Python) following the rules and technique described in Section 2.4.

2.5.1 Optimizations for Equality Saturation over Netlists

In the following three paragraphs, we describe three eqsat optimizations—structure-aware scoped rewriting, subsumption, and inverse transformations—which we used to scale to large designs. Finally, we present our reduced eqsat implementation, which allowed us to scale even further.

Structure-aware scoped rewriting. Netlists are large; applying rewrites over netlists in an e-graph can be costly. For example, MEMO’s general purpose boolean logic rewrites can apply anywhere boolean logic appears in the netlist, which increases rewriting time and grows the netlist unnecessarily. However, because the goal is memory decompilation, we only care about signals whose source or sink is a register. Thus, we scope our general-purpose rewrites to registers’ sources or sinks. On the `nerv` benchmark (Table 2.2), scoped rewriting enabled a speed up in decompilation time from 17 seconds to less than a second.

Subsumption. We use subsumption to further control e-graph size Köstler et al. [1995], Zhang et al. [2023]. For example, during register grouping we choose a single representation of the grouped registers, rather than maintain every possible grouping as separate nodes in the e-graph. All other groupings are subsumed.

Inverse transformations. Nandi et al. [2020] proposed *inverse transformations*, in which some rewrites insert speculative equalities into the e-graph, which are later validated. We utilize inverse transformations as an alternative technique to write port finding (which we compare against in Section 2.6). Write ports are generally decomposed into a demultiplexer tree and then aggressively optimized, and are thus difficult to find. For example, RISC-V, ARMv8-A, and MIPS register files have an unwriteable “zero” register, resulting in parts of the demultiplexer tree being optimized away. Rediscovering these paths comes down to profitably “undoing” optimizations to reveal higher-level structure. To do so, we apply an inverse transformation at each read port, which speculatively inserts an “inverse” write port followed by a write port. Using other rewrites, we then attempt to match portions of the write port with expressions in the netlist.

2.5.2 A Reduced Equality Saturation Implementation

Equality maintenance in eqsat engines is expensive. However, for decompilation purposes, we do not need completeness with regards to finding equalities. Missed equalities simply represent missed decompilation opportunities. Relaxing the need for completeness radically increases our performance by allowing us to skip expensive congruence maintenance steps. Meanwhile, our empirical evaluation in Section 2.6 clearly demonstrates that MEMO can still adequately decompile.

Our reduced eqsat implementation relies on two observations: (1) our initial netlist already tracks a significant number of equalities, (2) we can preserve an adequate level of equality tracking by checking for equivalent expressions before adding a new expression.

Regarding (1), in our experimental setup, we assume netlists were synthesized from Yosys. Internally, Yosys works hard to maximally deduplicate nets, effectively finding

equalities. Thus, the netlist synthesized from Yosys already tracks a significant number of equalities.

Regarding (2), equality saturation engines contain generic congruence maintenance data structures (e-graphs) and the routines to maintain them (rebuilding). These data structures can be expensive to maintain when the goal is to find all equalities implied by the rewrite set. For decompiling netlists with millions of gates, full congruence maintenance becomes too expensive and unnecessary. Instead, we maintain partial congruence. Consider a commutative rewrite $op(a, b) = op(b, a)$. Matching on $op(a, b) \rightarrow c$, the decompiler directly inserts $op(b, a) \rightarrow c$ instead of trying to add a new gate $op(b, a) \rightarrow d$ and merging c and d . As an example, on the `sparc_ffu` RISC-V benchmark (Table 2.2), turning off rebuilding led to a 10x speed up in decompilation time, while still uncovering the same set of memories.

Given these observations, we implement a reduced eqsat using the SQLite database software. Our initial e-graph is simply the output of Yosys’s JSON backend, which tracks equalities via its concept of net IDs. Our rewrites are queries into the database (searching for matches via `SELECTs`, inserting new expressions via `INSERTs`). Importantly, rewrites preserve the existing equalities by checking whether a net ID exists before inserting it.

2.6 Evaluation

In this section, we evaluate MEMO and our memory decompiler over a suite of representative hardware design benchmarks. We run all experiments on a workstation running Debian GNU/Linux 12 with an Intel i9-10850K CPU and 64 GB RAM. Through our evaluation, we seek to answer the following questions:

- RQ 1. Can fungible memories target multiple technologies across simulation, ASIC, and FPGA platforms?

RQ 2. To what extent does our decompilation-based memory lifting technique identify memories that state-of-the-art memory inference cannot?

RQ 3. How much of a real-world open-source SoC design can be automatically retargeted from one memory technology to another using our composed techniques?

For the memory decompiler, we evaluate two techniques that we call the “baseline” and “heuristic” techniques. The “baseline” technique strictly uses the rewrite rules as described in Section 2.4 within our reduced-eqsat implementation. The “heuristic” technique further adds inverse transformations as described in Section 2.5.1 to identify write ports under scenarios where the netlist obfuscates the structure of the memory. This technique is a heuristic because it is possible that the decompiler speculatively introduces a write port where one did not exist in the original netlist. However, as we show in our experimental results, this heuristic correctly identifies write ports, and does so faster than the baseline technique for the largest benchmarks.

2.6.1 Micro-Benchmark Experiments

We answer RQ 1 and RQ 2 through microbenchmark experiments demonstrating MEMO’s automated memory technology mapping and memory decompilation.

Memory design in Memo

To answer RQ 1, we port a five-stage RISC-V processor, originally implemented in PyRTL, to use fungible memories. The processor has three memory blocks: a register file, an instruction memory, and a data memory with a write mask. The original design only targets simulation, with no other memory technologies.

Table 2.1: MEMO memory mapping validation tests across a range of port configurations (**1rw**, **1r1w**, and **2rw**) and memory sizes, for a single fungible memory targeting the Vivado Tcl and OpenRAM backends. The port notation ‘**1r1w**’ indicates a memory with 1 distinct read and write port, whereas ‘**1rw**’ indicates 1 port which can either read or write (but not at the same time). $\checkmark_{Tcl}$ and $\checkmark_{OpenRAM}$ stand for validated mapping for the Vivado Tcl and OpenRAM backends respectively.

Memory Size	1rw	1r1w	2rw
32 × 16	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$
32 × 32	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$
32 × 64	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$
32 × 128	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$	$\checkmark_{Tcl}, \checkmark_{OpenRAM}$

With MEMO, we shrink the memory implementation from 130 lines of code to *3 lines of code*, one line for each of the three memories. Because PyRTL memory blocks do not natively support write masks, the original design manually implemented the masking logic for the data memory. We can instead specify the entire data memory in one line:

```
dmem = FungibleMemory([Read(addr, rdata)], [Write(addr, wdata, mask)])
```

After porting the RISC-V core to MEMO, the implementation still passes a comprehensive set of functional validation tests. Porting the design to MEMO also comes with the added benefit of automatically targeting four additional backends. Two (Basejump STL and synthesizable BRAMs) generate Verilog module instantiations. The other two (Vivado Tcl and OpenRAM) generate configurations to be processed by their respective memory compilers. Table 2.1 shows that these backends generate valid memory configurations that successfully target the given technologies. We validate the configurations by running them through the backend’s toolchain and verify that the memories mapped correctly to the technology (the tool will report a failure if the memory cannot be mapped).

Memory Decompileation

To answer RQ 2, we compare our memory decompilation technique against state-of-the-art memory inference in a proprietary synthesis toolchain and Yosys. Note that the

Table 2.2: Memory decompilation results comparing against memory inference in Yosys and a state-of-the-art proprietary synthesis toolchain. The **Baseline** result uses the strict reversed rewrites as explained in Section 2.4. The **Heuristic** result applies heuristics presented in Section 2.5 to handle netlists with more obfuscated structure. **Gates** are measured in thousands (k). The **Memory** column indicates the expected memory in the design. Decompilation times are given in seconds (s).

Design	Gates	Memory	Baseline		Heuristic		Inference	
			Decompiled	Time	Decompiled	Time	Yosys	Proprietary
bsg_cache	92 k	1rw: $(8 \times 256) \times 8$	$\emptyset$	0.9 s	✓	3 s	$\emptyset$	$\emptyset$
bsg_fifo	34 k	1r1w: (64×256)	✓	0.3 s	✓	0.7 s	$\emptyset$	✓
bsg_fifo	527 k	1r1w: (512×512)	✓	5 s	✓	23 s	$\emptyset$	✓
bsg_mem_3r1w	9 k	3r1w: (66×32)	✓	0.1 s	✓	0.1 s	✓	✓
nerv	8 k	2r1w: (32×32)	✓	0.8 s	✓	0.1 s	✓	✓
pico	10 k	2r1w: (32×32)	✓	0.7 s	✓	0.1 s	✓	✓
sparc_ffu	27 k	1r1w: (78×128)	✓	0.9 s	✓	0.4 s	✓	$\emptyset$

comparison is not completely one-to-one because these techniques work through syntactic template-based matching in HDL code, whereas our technique works at the netlist level. Nonetheless, we report if the tools’ memory inference works over the original HDL code that generated the netlists we run through our memory decompiler. We consider a set of seven micro-benchmarks coming from netlists synthesized from open-source hardware design suites such as OPDB Tziantzioulis et al. [2022] and the Basejump STL Taylor [2018] which contain a single memory block. Since not all modules in the Basejump STL have memories, we chose the cache, FIFO, and 3r1w memory as representative micro-benchmarks. More variants of these modules (different sized FIFOs, 1r1w and 2r1w memory blocks) are exercised in the large-scale BlackParrot case study (Section 2.6.2).

We synthesize the netlists with Yosys, using standard synthesis scripts, including optimizations with ABC Brayton and Mishchenko [2010]. The selection of micro-benchmarks shows the range of memories MEMO decompiles, as they are deployed in a variety of designs from caches to cores.

Table 2.2 presents the results, which show the memory decompiler finds the expected memory with the expected configuration for each micro-benchmark. Noteworthy are `bsg_cache`, `bsg_fifo`, and `sparc_ffu`: *we successfully identify memories where either Yosys or the proprietary synthesis tool, or both, did not*. Because the state of the art

infers memories through syntactic template matching, memory inference is sensitive to the implementation. Memory decompilation, by contrast, works on the structure of the netlist.

There are instances where the heuristic approach is slower than the baseline (for example, the 512×512 `bsg_fifo`). For netlists where the cells are almost entirely memories, the baseline approach runs fast. However, as we show next in the larger case studies, netlists with more diverse and unrelated logic (e.g., an entire SPARC core) will slow down the baseline and the heuristics help to cut down the search space.

2.6.2 Large-Scale Case Studies

To answer RQ 3, we show how MEMO memory decompilation enables technology retargeting on real-world open-source SoC designs with many memory blocks. We consider four larger case studies from OPDB with multiple expected memory blocks. Table 2.3 presents the memory decompilation and retargeting results. Again, the memory decompiler identifies the expected and exact memories for each case study. These results also demonstrate how the memory decompilation technique scales to large designs, with the L2 cache totaling 1.5 million gates. Using the original source code as a reference, the first three decompiled memory blocks for the L2 cache correspond to the port configurations and dimensions of the data, tag, and state memory blocks.

Table 2.4 provides fine-grain timing data for the case studies. This reveals a bottleneck for the baseline technique in the **Write Ports** phase, where memory decompilation spends a majority of the time. The data also shows the benefits of our equality saturation optimizations (Section 2.5.1). For large netlists with diverse and unrelated logic, such as the OPDB FFT and SPARC Core, the heuristic technique successfully cuts down the search space and accelerates write port identification.

Table 2.3: Large-scale case studies from OPDB Tziantzioulis et al. [2022]. Since each benchmark contained multiple memories, the table reports them separately in the “Decompiled” column, for the baseline algorithm. Decompileation times are totals for each module including all decompiled memories. Times marked with $\dagger$ use the heuristic technique. $\checkmark_{Tcl}$ or $\checkmark_{OpenRAM}$ indicates successful retargeting to Vivado Tcl or OpenRAM, respectively.

Design	Gates	Decompiled	Time (s)	Retargeted
OPDB FFT	194 k	1r1w: (42 $\times$ 512)	129 s 6 s †	$\checkmark_{Tcl}$ $\checkmark_{OpenRAM}$
		1r1w: (34 $\times$ 128)		
		1r1w: (32 $\times$ 128)		
		1r1w: (36 $\times$ 64)		
OPDB 11.5 cache	242 k	1r1w: (158 $\times$ 512)	18 s 13 s †	$\checkmark_{Tcl}$ $\checkmark_{OpenRAM}$
		1r1w: (116 $\times$ 128)		
		1r1w: (8 $\times$ 128)		
		1r1w: (8 $\times$ 128)		
OPDB 12 cache	1.5 m	1r1w: (144 $\times$ 4096)	71 s 120 s †	$\checkmark_{Tcl}$ $\checkmark_{OpenRAM}$
		2r1w: (170 $\times$ 256)		
		1r1w: (64 $\times$ 16)		
		1r1w: (16 $\times$ 4)		
SPARC core	769 k	1r1w: (544 $\times$ 256)	254 s 56 s †	$\checkmark_{Tcl}$ $\checkmark_{OpenRAM}$
		1r1w: (576 $\times$ 128)		
		1r1w: (120 $\times$ 128)		
		1r1w: (78 $\times$ 128)		
		1r1w: (151 $\times$ 32)		

Table 2.4: Fine-grain timing results for memory decompilation over the large-scale case studies, comparing the baseline (**Base**) and heuristic (**Heur**) techniques. The phases are: **Loading** the netlist into the database, **Rewrites** DFFs and base logic gates, **Read Ports** identification via multiplexer tree reduction, **Write Ports** identification, **Extraction** of the final memory extraction based on the identified read and write ports.

Phase	OPDB FFT		OPDB 11.5		OPDB 12		SPARC Core	
	Base	Heur	Base	Heur	Base	Heur	Base	Heur
Load	0.3 s	2 s	0.5 s	2 s	4 s	13 s	2 s	7 s
Rewrites	0.1 s	2 s	0.1 s	3 s	0.1 s	4 s	.3 s	28 s
Read Ports	0.4 s	0.1 s	1 s	5 s	8 s	59 s	3 s	11 s
Write Ports	128 s	2 s	16 s	3 s	58 s	44 s	247 s	10 s
Extract	0.3 s	0.001 s	0.7 s	0.001 s	1 s	0.002 s	3 s	0.001 s

Table 2.5: BlackParrot memory decompilation results using the heuristic technique. The “Corresponding Modules” column maps the decompiled memories to a memory blocks in the source code based on the dimensions and portedness.

Decompiled Memories	Corresponding Modules
32 $\times$ 14, 64 $\times$ 184, 512 $\times$ 64 ($\times$ 8)	Instruction Cache (data, tag, stat)
64 $\times$ 15, 64 $\times$ 184, 512 $\times$ 8 ($\times$ 64)	Data Cache (data, tag, stat)
512 $\times$ 8, 64 $\times$ 50, 16 $\times$ 43	BHT, BTB, RAS
32 $\times$ 66 (2r1w, 3r1w)	Int & FP Register Files
8 $\times$ 80, 8 $\times$ 174, 4 $\times$ 114	Preissue/Issue/Cmd FIFOs
32 $\times$ 20 ($\times$ 3), 32 $\times$ 67, 8 $\times$ 15 ($\times$ 3), 4 $\times$ 10	Uncategorized
Core Size: 11.8 million gates	Time: 373 seconds

Next, we consider BlackParrot, an open-source RISC-V multicore SoC Petrisko et al. [2020]. Table 2.5 presents the results from decompiling an entire BlackParrot core, totaling

11.8 million gates. Using the faster, heuristic technique, memory decompilation took 373 seconds, and identified 92 distinct memory blocks. We do not report results for the baseline technique because the memory decompilation did not finish after two hours. It is noteworthy that memory decompilation *successfully identifies all expected memory blocks in the netlist of the BlackParrot core*.

The BlackParrot core itself is split between front end and back end (although in the netlist, there is no such distinction). Memories in the front end include a BHT (Branch History Table), a BTB (Branch Target Buffer), RAS (Return Address Stack), and 3 smaller queues as part of the PC generation module, as well as 3 memories that make up the instruction cache. Memories in the back end include an integer register file, a floating-point register file, and 3 memories as part of the data cache. The data memory block that is part of the data cache is decompiled as 64 memories of size 512×8 , due to the original memory using a byte mask—recovering a common Verilog pattern used in the original source code which implements a byte mask using a **generate** for-loop, instantiating 64 1-byte-wide memories.

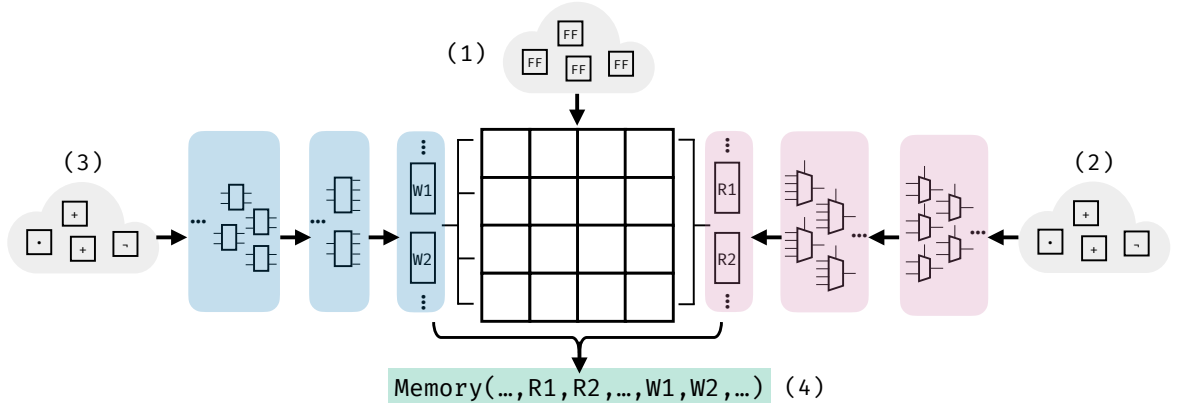


Figure 2.7: A visual overview of memory decompilation, split into four phases: (1) Group registers; (2) Identify and reduce multiplexers to form read ports; (3) Identify and reduce decoders to form write ports; and (4) Extract a fungible memory.

Register Aggregate

$$\text{Reg}(en, d_0 ++ d_1) = \text{Reg}(en, d_0) ++ \text{Reg}(en, d_1),$$

where $\text{Reg}(en, d_0 ++ d_1)[0] = \text{Reg}(en, d_0), \text{Reg}(en, d_0 ++ d_1)[1] = \text{Reg}(en, d_1)$

Mux Reduce

$$\text{Mux}(s, a ++ b) = (a \wedge \neg s) \vee (b \wedge s), \text{ where } s, a, b : \mathbb{B}$$

$$\text{Mux}(s_0 ++ s, x_0 ++ x_1) = \text{Mux}(s_0, \text{Mux}(s, x_0) ++ \text{Mux}(s, x_1)), \text{ where } x_0, x_1 : \mathbb{B}^{2^n}, s : \mathbb{B}^n, s_0 : \mathbb{B}$$

Figure 2.8: A selection of rewrite rules for memory decompilation. **Reg** is a constructor for a register (or DFF if the data signal is a single bit) which takes an enable signal and data signal. **Mux** is a constructor for a multiplexer which takes two arguments: a select signal and signal to select bits from (at least 2 bits in length). $x ++ y$ denotes concatenation of two bitvector values. $x[i]$ denotes selecting the i th bit from bitvector value x .

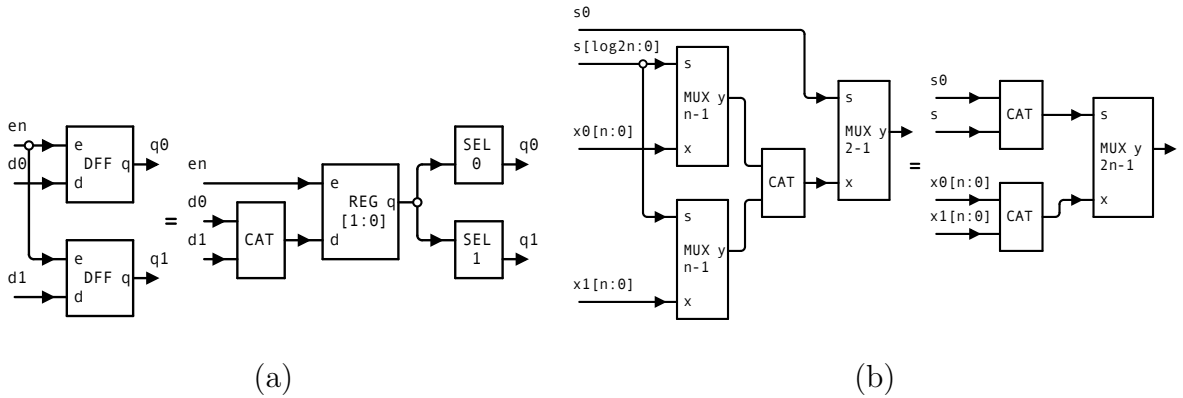


Figure 2.9: (a) Illustration of the **Register Aggregate** rule. (b) Illustration of the **Mux Reduce** rule.

Chapter 3

Nextmap and Semantic E-Graphs

This chapter is based on joint work that I led with Jingtao Xia, Daniel Ruelas-Petrisko, Zachary D. Sisco, Jonathan Balkind, and Gus Henry Smith. I led the project and developed the core direction of the work, including the semantic e-graph framework and its use in NEXTMAP. Jingtao Xia contributed extensively to the writing, especially to much of the semantic e-graph formalization and proofs. Daniel Ruelas-Petrisko helped shape the work early by providing workflow expectations from a hardware expert perspective. Zachary D. Sisco contributed to the related work and helped position the project within the broader literature. Jonathan Balkind enriched the tool and helped strengthen the implementation. Gus Henry Smith hosted the project’s weekly meetings, wrote the introduction and overview, and provided invaluable help through his Churchroad project, which helped motivate this work. This work previously appeared in PLDI 2026 under the title “Improving Equality Saturation for EDA via Semantic E-graphs.” The material is reorganized here to fit the thesis narrative.

3.1 Semantic E-Graphs

An *e-graph* Nelson [1980], Tate et al. [2009], Willsey et al. [2021], Zhang et al. [2023] is a data structure for compactly representing many equivalent expressions. It organizes expressions into *e-classes*, each of which contains a set of *e-nodes* representing equivalent terms. E-graphs maintain *congruence closure*: if two e-nodes have the same function symbol and their corresponding children belong to the same e-classes, then the two e-nodes are equivalent and are merged into the same e-class. In standard e-graphs, each e-class is identified by an opaque identifier, which serves as a representative for all expressions in the class.

Semantic e-graphs (SEGs) extend standard e-graphs by replacing opaque e-class identifiers with *semantic identifiers* drawn from a user-defined semantic domain. A semantic identifier gives a domain-specific abstraction of an e-class: for example, $1 + 3$ and $2 + 2$ can both be assigned the identifier 5, so they are recognized as equivalent and merged automatically.

In this section, we formalize semantic e-graphs and describe the maintenance algorithm needed to support semantic identifiers.

3.1.1 Semantic Domains

Semantic e-graphs represent well-typed terms from a simply typed, first-order, monomorphic term language. Each function symbol is assigned a fixed type signature, and every well-formed term has a unique type τ . We maintain the invariant that each e-class contains only e-nodes of a single type; we call this the type of the e-class. We omit type annotations when they are clear from context.

For each type τ appearing in e-nodes, we associate two semantic domains:

- a *concrete semantic domain* $\mathbb{C}_\tau$: fully interpreted values;

- a *symbolic semantic domain* $\mathbb{S}_\tau$: opaque placeholders used otherwise.

For simplicity, we write the combined semantic domain as $\mathbb{D}_\tau = \mathbb{C}_\tau \cup \mathbb{S}_\tau$. The two domains are disjoint by construction. Equality on $\mathbb{S}_\tau$ is identity: two symbolic values are equal iff they are the same placeholder. Equality on $\mathbb{C}_\tau$ is user-defined and assumed decidable; it must be sound with respect to the intended semantics, but need not be complete.

We distinguish a class of *composite types* κ (e.g., tuples and vectors). Composite domains are equipped with a fixed set of *domain constructors*, which define the internal structure of values in $\mathbb{C}_\kappa$. These are distinct from function symbols in the term language: semantic constructors $\llbracket f \rrbracket$ compute values, whereas domain constructors provide structured representations that support decomposition. For such types, the concrete domain $\mathbb{C}_\kappa$ consists of structured values of the form $K(v_1, \dots, v_n)$, where K is a domain constructor of $\mathbb{C}_\kappa$ with component types $\tau_1, \dots, \tau_n$ and each $v_i \in \mathbb{D}_{\tau_i}$. We assume that for each such constructor K , there exist projections π_i such that for all i , $\pi_i(K(v_1, \dots, v_n)) = v_i$. In particular, this implies injectivity:

$$K(v_1, \dots, v_n) = K(w_1, \dots, w_n) \Rightarrow v_i = w_i \quad \forall i.$$

This injectivity property applies only to $\mathbb{C}_\kappa$; the symbolic domain $\mathbb{S}_\kappa$ contains opaque placeholders with no such structure.

Values of all other non-composite types are treated as opaque; no decomposition property is assumed.

Definition 3.1.1 (Precision Order) *For each type τ , we define a relation $\sqsubseteq_\tau$ on $\mathbb{D}_\tau$ capturing the precision of values: symbolic values are less precise than concrete values, and two distinct symbolic values are minimal elements and incomparable with each other.*

If τ is not a composite type, then $\sqsubseteq_\tau$ is defined by

$$x \sqsubseteq_\tau y \iff (x = y) \vee (x \in \mathbb{S}_\tau \wedge y \in \mathbb{C}_\tau).$$

If κ is a composite type, then $\sqsubseteq_\kappa$ is defined by

$$x \sqsubseteq_\kappa y \iff (x = y) \vee (x \in \mathbb{S}_\kappa \wedge y \in \mathbb{C}_\kappa) \vee \begin{pmatrix} x = K(v_1, \dots, v_n), \\ y = K(w_1, \dots, w_n), \\ \forall i, v_i \sqsubseteq_{\tau_i} w_i \end{pmatrix},$$

where K is a domain constructor of $\mathbb{C}_\kappa$ as defined above. Two concrete composite values are comparable only when they share the same constructor and their components are comparable pointwise.

Lemma 3.1.1 *Each $\sqsubseteq_\tau$ is a partial order.*

3.1.2 Semantic E-Graph Structure

Figure 3.1 presents the syntax of semantic e-graphs. We extend the standard e-graph syntax Willsey et al. [2021] by associating each e-class with a *semantic identifier* drawn from the appropriate domain $\mathbb{D}_\tau$.

types	τ, κ	semantic domains	$\mathbb{C}_\tau, \mathbb{S}_\tau, \mathbb{D}_\tau$
function symbols	$f, g : \tau_1 \times \dots \times \tau_m \longrightarrow \tau$	semantic identifier	$\delta \in \mathbb{D}_\tau$
terms	$t ::= f \mid f(t_1, \dots, t_m)$		$m \geq 1$
e-nodes	$n ::= f \mid f(\delta_1, \dots, \delta_m)$		$m \geq 1$
e-classes	$c ::= \{n_1, \dots, n_k\} \triangleright \delta$		$k \geq 1$

Figure 3.1: Syntax and metavariables for semantic e-graphs. Each e-node refers to child e-classes $c_i \triangleright \delta_i$, and each e-class is annotated with a semantic identifier $\delta \in \mathbb{D}_\tau$.

Formally, an e-class is a pair (N, δ) consisting of its e-node set and semantic identifier,

but we write $\{n_1, \dots, n_k\} \triangleright \delta$ for brevity. We also write $c \triangleright \delta$ to indicate that semantic identifier δ is associated with e-class c . We write $\llbracket n \rrbracket$ for the semantic identifier of the e-class containing n .

To construct and maintain a semantic e-graph, we require a family of semantic operations associated with the term language.

Definition 3.1.2 (Semantic Constructor) *For each function symbol $f : \tau_1 \times \dots \times \tau_m \rightarrow \tau$, a semantic constructor for f is a function $\llbracket f \rrbracket : \mathbb{D}_{\tau_1} \times \dots \times \mathbb{D}_{\tau_m} \rightarrow \mathbb{D}_{\tau}$ with the following behavior:*

- *Semantic case.* If $\llbracket f \rrbracket$ can compute a concrete result from its inputs, it returns a value in $\mathbb{C}_{\tau}$.
- *Fallback case.* Otherwise, $\llbracket f \rrbracket$ returns a symbolic element of $\mathbb{S}_{\tau}$.

In the semantic case, $\llbracket f \rrbracket$ exposes equalities that would otherwise require explicit rewrite rules or e-class analysis; for a composite type κ , the result is a value in $\mathbb{C}_{\kappa}$ whose components may themselves be symbolic. In the fallback case, $\llbracket f \rrbracket$ returns an opaque identifier analogous to the e-class IDs in a classical e-graph. In the formalization, we model this symbolically as the uninterpreted term $f(\delta_1, \dots, \delta_m)$ itself, avoiding any need for effectful fresh name generation.

Example 3.1.1 (Arithmetic Constant Folding) *Consider the e-nodes $n_1 = +(c_1 \triangleright 1, c_4 \triangleright 4)$ and $n_2 = +(c_2 \triangleright 2, c_3 \triangleright 3)$, where each c_i is a constant e-class with semantic identifier in $\mathbb{C}_{Int}$. Since constants interpret to themselves, the constructor evaluates $\llbracket n_1 \rrbracket = \llbracket + \rrbracket(1, 4) = 5$ and $\llbracket n_2 \rrbracket = \llbracket + \rrbracket(2, 3) = 5$. Both identifiers lie in $\mathbb{C}_{Int}$, so the two e-nodes are immediately assigned the same semantic identifier 5. Thus, n_1 and n_2 are inserted into the same e-class without requiring any rewrite rules or e-analysis.*

Now consider the e-node $n = +(c_1 \triangleright v_1, c_2 \triangleright 4)$, where $v_1 \in \mathbb{S}_{Int}$ is not concretely interpretable. Because the semantic constructor cannot evaluate $+$ on these inputs, it falls back and produces a symbolic identifier $v_3 \in \mathbb{S}_{Int}$, yielding a new e-class $c_3 \triangleright v_3$. This matches the behavior of a traditional e-graph: the result is treated as an uninterpreted value.

Assumption 3.1.1 (Constructor Precision Monotonicity) For every function symbol $f : \tau_1 \times \dots \times \tau_m \rightarrow \tau$ and for all $x_i, y_i \in \mathbb{D}_{\tau_i}$, if $x_i \sqsubseteq_{\tau_i} y_i$ for all i , then

$$\llbracket f \rrbracket(x_1, \dots, x_m) \sqsubseteq_{\tau} \llbracket f \rrbracket(y_1, \dots, y_m).$$

This states that making inputs more precise cannot make the result less precise. In particular, once $\llbracket f \rrbracket$ produces a concrete result, further refinement of the arguments must preserve concreteness.

Lemma 3.1.2 (Ascending-Chain Condition) For each type τ , $(\mathbb{D}_{\tau}, \sqsubseteq_{\tau})$ satisfies the ascending-chain condition.

Proof: For non-composite types, every strict refinement goes from a symbolic value to a concrete value, and concrete values are maximal, so all chains are finite. For a composite type κ , the claim follows by structural induction: every value in $\mathbb{C}_{\kappa}$ is generated by a finite, acyclic constructor tree, and each component domain satisfies the ascending-chain condition by hypothesis. ■

Equality between two concrete composite values $x, y \in \mathbb{C}_{\kappa}$ is decomposed pointwise via the projections π_i :

$$\Delta_{\kappa}(x, y) = \{(\pi_1(x), \pi_1(y)), \dots, (\pi_n(x), \pi_n(y))\}.$$

Each pair $(\pi_i(x), \pi_i(y))$ should be recursively merged, rather than merging the two composite identifiers directly.

Example 3.1.2 (Vector Decomposition) *Let $\mathbb{D}_{\text{Vec}_3}$ denote the domain of length-3 vectors with projections π_1, π_2, π_3 . Consider two identifiers in $\mathbb{C}_{\text{Vec}_3}$: $\delta_1 = [\delta_{1,1}, \delta_{1,2}, \delta_{1,3}]$ and $\delta_2 = [\delta_{2,1}, \delta_{2,2}, \delta_{2,3}]$. Decomposition via projections yields:*

$$\Delta_{\text{Vec}_3}(\delta_1, \delta_2) = \{(\delta_{1,1}, \delta_{2,1}), (\delta_{1,2}, \delta_{2,2}), (\delta_{1,3}, \delta_{2,3})\}.$$

Each pair $(\pi_i(\delta_1), \pi_i(\delta_2))$ represents a required equality between the corresponding components. Thus, equality of δ_1 and δ_2 implies equality of all their aligned components.

Intuitively, semantic construction produces semantic values during insertion, while semantic decomposition exposes additional equalities through structured values that must be maintained by the e-graph. The partial order over semantic domains provides a notion of refinement. Together, these mechanisms allow the e-graph to reason about both syntactic and interpreted equality within a single incremental framework.

Definition 3.1.3 (Semantic E-Graph) *A semantic e-graph is a set of typed e-classes where each e-class is identified by a semantic identifier from $\mathbb{D}_\tau$, rather than an opaque id. It is equipped with a semantic constructor $\llbracket f \rrbracket$ for each function symbol, which computes the semantic identifier of a parent node from those of its children. For composite types, equality between identifiers is decomposed componentwise via the projections π_i .*

3.1.3 Semantic E-Graph Operations

Semantic identifiers are not merely auxiliary metadata: they are first-class participants in equivalence maintenance. One could imagine storing semantic values in an external map indexed by opaque e-class ids, but such a map would be passive — it could not drive

merging on its own. In our design, two e-classes with the same semantic identifier are merged directly, and composite identifiers propagate equality to their components through rebuild. The two primary mutating operations are **sadd** and **smerge**, both relying on a deferred *rebuild* phase that restores congruence closure and semantic consistency.

For each type τ , we maintain a union-find structure W_τ over $\mathbb{D}_\tau$ to represent equalities among semantic identifiers. We write **find** $_\tau$ and **union** $_\tau$ for its operations: **find** $_\tau(\delta)$ returns the current canonical representative of δ , and **union** $_\tau(\delta_1, \delta_2)$ merges the equivalence classes containing δ_1 and δ_2 .

Insertion. Let τ be a result type, and let $n = f(\delta_1, \dots, \delta_m)$ be an e-node of type τ . The operation **sadd** $_\tau(n)$ proceeds as follows:

1. Canonicalize each child: let $\delta'_i = \mathbf{find}_\tau(\delta_i)$ and form $n' = f(\delta'_1, \dots, \delta'_m)$. If n' already exists in some e-class c , return c .
2. Otherwise, compute $v = \llbracket f \rrbracket(\delta'_1, \dots, \delta'_m) \in \mathbb{D}_\tau$.
3. Create a new e-class containing n' and annotate it with v .

Thus, insertion performs semantic interpretation eagerly, while deferring any induced merging to rebuild.

Merge. For each type τ , the operation **smerge** $_\tau(\delta_1, \delta_2)$ with $\delta_1, \delta_2 \in \mathbb{D}_\tau$ requests that the two identifiers be made equal.

If $\tau = \kappa$ is a composite type and $\delta_1, \delta_2 \in \mathbb{C}_\kappa$, then semantic equality is propagated structurally via the projections π_i . Instead of directly invoking **union** $_\kappa$, we recursively invoke **smerge** $_{\tau_i}(\pi_i(\delta_1), \pi_i(\delta_2))$ for each component i . After the components are merged, the updates are propagated to parents during rebuild.

Otherwise, that is, when τ is not a composite type, or when at least one of δ_1, δ_2 is symbolic, smerge_τ simply invokes $\text{union}_\tau(\delta_1, \delta_2)$ in W_τ . No decomposition occurs; if one identifier is concrete and the other symbolic, union_τ records the concrete value as canonical.

Example 3.1.3 (Complex Numbers) *Assume Complex is a composite type with domain constructor $\text{cmplx} : \text{Real} \times \text{Real} \rightarrow \text{Complex}$ and projections $\pi_1(\text{cmplx}(a, b)) = a$, $\pi_2(\text{cmplx}(a, b)) = b$. Consider two e-classes $c_1 \triangleright \text{cmplx}(a_1, b_1)$ and $c_2 \triangleright \text{cmplx}(a_2, b_2)$.*

Invoking $\text{smerge}_{\text{Complex}}(\text{cmplx}(a_1, b_1), \text{cmplx}(a_2, b_2))$ applies projections to yield the component obligations $\text{smerge}_{\text{Real}}(a_1, a_2)$ and $\text{smerge}_{\text{Real}}(b_1, b_2)$. Rebuild then propagates the canonical representatives a^ and b^* upward, merging the two original e-classes with canonical identifier $\text{cmplx}(a^*, b^*)$.*

This example illustrates the two directions of propagation: projection-based decomposition propagates equality downward to components, while rebuild propagates canonical identifiers upward to composite parents.

Rebuild. As in many equality-saturation engines, semantic e-graphs use deferred rebuild for efficiency. During rebuild, every occurrence of a semantic identifier is rewritten to its current canonical representative $\text{find}_\tau(\delta)$, including occurrences inside composite identifiers. Any parent e-node affected by such changes is then re-canonicalized and rehashed, which may trigger congruence merges exactly as in a standard e-graph. In addition, its semantic identifier is recomputed using the corresponding semantic constructor. By , such recomputation can only yield a more precise identifier. If the recomputed identifier is already associated with another e-class, union_τ merges the two e-classes in W_τ .

Rebuild repeats this process until no union-find structure changes and no parent requires reprocessing. At that point, the semantic e-graph has reached a fixpoint: there are no pending merge obligations and no further semantic refinement is possible. In

practice, we maintain a hash map from semantic identifiers to e-classes to detect congruence merges in $O(1)$ expected time. This relies on hash functions consistent with domain equality for concrete values.

Lemma 3.1.3 (Canonical Representative) *After every **sadd**, **smerge**, and **rebuild** step, the canonical representative of each equivalence class in W_τ is the maximally precise element known for that class with respect to $\sqsubseteq_\tau$.*

Proof: For non-composite types, if the class contains a concrete value, it is the unique most precise element: merging two distinct concrete values would violate the soundness of the semantic relation R_{semantic} and therefore does not occur. If all elements are symbolic, they are incomparable, and any one may serve as canonical representative, matching the behavior of a traditional e-graph. For composite types, two concrete values are never merged directly via **union_K**. Instead, **smerge** decomposes them via projections, recursively merging each component. During rebuild, the semantic constructor recomputes the composite representative from the canonical representatives of its components. Since both original values share the same domain constructor K , the reconstructed composite identifiers from both sides are identical after component merges, naturally yielding a single canonical representative. By induction, each component’s canonical representative is maximally precise, so the reconstructed composite is also maximally precise. ■

3.1.4 Rewriting

Rewriting in a semantic e-graph follows the same overall structure as in a classical e-graph. Rewrite rules are statically type-checked and required to be type-preserving: the left- and right-hand sides must have the same type. In a simply-typed, monomorphic setting, each function symbol has a fixed type signature, so the type of any e-node is determined by its function symbol alone, and no runtime type checks are needed.

Given a rewrite rule $\ell \rightarrow r$ where both sides have type τ , e-matching is purely syntactic and unchanged from a classical e-graph: **ematch** traverses the e-graph structure, matching e-nodes by function symbol and binding each pattern variable to a matched e-class. No semantic information is consulted during matching.

For each match, **ematch** produces a substitution σ mapping each pattern variable to the semantic identifier of the matched e-class. Applying σ to the right-hand side r yields $r[\sigma]$, a term where each pattern variable is replaced by a semantic identifier. This term is then inserted via **sadd**, which recursively processes each e-node bottom-up and invokes the semantic constructor at each step; semantic interpretation occurs entirely at this stage. The resulting e-class is then equated with the matched e-class c via **smerge** $_{\tau}(\delta_c, \text{sadd}_{\tau}(r[\sigma]))$, where δ_c is the semantic identifier of c . Semantic interpretation thus occurs entirely within **sadd** and **smerge**, not during matching.

A semantic e-graph is *saturated* when no rewrite produces a new e-node and rebuild reaches a fixpoint.

3.1.5 Properties

The semantic e-graph derives equalities from two user-defined sources: the *syntactic relation* $R_{\text{syntactic}}$, induced by rewrite rules, and the *semantic relation* R_{semantic} , induced by semantic construction and semantic decomposition. More concretely, R_{semantic} includes (i) equalities discovered by **sadd**, when two terms are assigned the same semantic identifier by a constructor $\llbracket f \rrbracket$, and (ii) equalities generated by **smerge**, when merging concrete composite identifiers triggers recursive merges via projections π_i .

Theorem 3.1.1 (Algorithmic Soundness) *All equalities maintained by the semantic e-graph are contained in the least congruence containing $R_{\text{syntactic}} \cup R_{\text{semantic}}$.*

Proof: We argue by induction over the execution trace. Each maintained equality is

either introduced by a rewrite, **sadd**, or **smerge**, or inherited from previously maintained equalities through rebuild.

- *Rewrite*. Each rewrite step introduces an equality in $R_{\text{syntactic}}$ by definition.
- *sadd*. When two terms are assigned the same semantic identifier by a constructor $\llbracket f \rrbracket$, the resulting equality is in R_{semantic} by definition.
- *smerge*. When merging two concrete composite identifiers, decomposition via projections π_i generates component equalities. We define R_{semantic} to include equalities induced by semantic decomposition, so these component equalities are in R_{semantic} .
- *Rebuild*. Rebuild performs only congruence closure: it propagates canonical identifiers upward and merges parents with identical structure, introducing no new source of equality beyond closure.

The first three cases contribute edges in $R_{\text{syntactic}} \cup R_{\text{semantic}}$ by definition, while rebuild contributes only congruence closure. Therefore all maintained equalities lie in the least congruence containing $R_{\text{syntactic}} \cup R_{\text{semantic}}$. ■

Corollary 3.1.1 (Semantic Soundness) *Let $\equiv \subseteq \mathcal{T} \times \mathcal{T}$ be a ground-truth congruence on well-typed terms, where $\mathcal{T}$ is the set of well-typed terms. If $R_{\text{syntactic}} \subseteq \equiv$ and $R_{\text{semantic}} \subseteq \equiv$, then all equalities maintained by the semantic e-graph are sound with respect to $\equiv$.*

Proof: By , all maintained equalities lie in the least congruence containing $R_{\text{syntactic}} \cup R_{\text{semantic}}$. Since $\equiv$ is itself a congruence and $R_{\text{syntactic}} \subseteq \equiv$ and $R_{\text{semantic}} \subseteq \equiv$, the least such congruence is contained in $\equiv$. ■

Lemma 3.1.4 (Monotone Refinement) *During saturation, semantic identifiers evolve monotonically: for every merge of two e-classes of type τ with identifiers $\delta_1, \delta_2 \in \mathbb{D}_\tau$, the resulting canonical identifier δ^* satisfies $\delta_1 \sqsubseteq_\tau \delta^*$ and $\delta_2 \sqsubseteq_\tau \delta^*$.*

Proof: For non-composite types, the property follows directly from : W_τ maintains the most precise canonical representative, so the merged class inherits a representative at least as precise as either input.

For composite types, it follows by structural induction. Suppose $\delta_1 = K(v_1, \dots, v_n)$ and $\delta_2 = K(w_1, \dots, w_n)$, where K is the domain constructor of a composite type κ . Then **smerge** does not merge δ_1 and δ_2 directly; instead, it generates the component obligations $\{(\pi_i(\delta_1), \pi_i(\delta_2))\}_{i=1}^n$. By induction, each recursive merge yields a refined representative u_i such that $v_i \sqsubseteq_{\tau_i} u_i$ and $w_i \sqsubseteq_{\tau_i} u_i$. Therefore the reconstructed composite identifier $\delta^* = K(u_1, \dots, u_n)$ satisfies $\delta_1 \sqsubseteq_\kappa \delta^*$ and $\delta_2 \sqsubseteq_\kappa \delta^*$ by the definition of the pointwise order on composite values.

The same invariant is preserved during rebuild. If an e-node $n = f(\delta_1, \dots, \delta_m)$ is reinterpreted after its children have been refined to $\delta'_1, \dots, \delta'_m$ with $\delta_i \sqsubseteq_{\tau_i} \delta'_i$ for all i , then gives $\llbracket f \rrbracket(\delta_1, \dots, \delta_m) \sqsubseteq_\tau \llbracket f \rrbracket(\delta'_1, \dots, \delta'_m)$. Hence rebuilding after child merges can only preserve or refine the parent's identifier. ■

Theorem 3.1.2 (Termination) *Rebuild terminates.*

Proof: Every reprocessing step corresponds to strict progress in a well-founded measure. A parent e-node is re-enqueued only when one of the following occurs:

- its e-class participates in a merge; or
- one of its child identifiers is replaced by a strictly more precise canonical identifier.

The first kind of event can occur only finitely many times, because each merge strictly decreases the number of e-classes. The second kind is also finite: by , identifiers evolve monotonically, so the sequence of identifiers assigned to an e-class forms an ascending chain in $\sqsubseteq_\tau$; by , this chain is finite, so each e-class can be strictly refined only finitely many times.

Every refinement is attached to an already existing e-class, and under γ , each e-class can be refined only finitely many times.

Therefore rebuild must eventually reach a fixpoint at which no further merges occur in any union-find structure W_τ and no e-class identifier is further refined. ■

3.2 Semantic E-Graphs for Hardware

In this section, we instantiate the semantic e-graph framework for hardware by defining concrete e-class types and semantic domains for Register-Transfer Level (RTL) designs. We then demonstrate how these semantic domains enable forms of hardware reasoning that are difficult to achieve with traditional e-graphs.

3.2.1 Hardware Basics

We formalize hardware designs at the Register-Transfer Level¹ (RTL) using a small set of primitive constructs: *wires*, *wire vectors*², *tuples*, and *cells*.

Wires. A *wire* represents a single-bit signal and is drawn from a set $W = \{w_0, w_1, \dots\}$.

Wire Vectors. A *wire vector* represents a multi-bit signal and is treated as an uninterpreted syntactic constructor at the structural level. Each vector has an associated *width* $n \in \mathbb{N}$ and is denoted $v \in \text{Vec}_n$, corresponding to an ordered sequence of n wires.

Signals and Tuples. We define the set of *signals* as $S = W \cup \bigcup_n \text{Vec}_n$, capturing both wires and wire vectors. A tuple is an ordered collection of signals, $t = (x_1, x_2, \dots, x_k)$

¹In RTL, designs are represented very simply as networks of registers (state elements) and combinational logic.

²Wires and wire vectors are analogous to the bits and bitvectors referenced in Sections and ; in this section, we adopt more hardware-centric terminology.

with $x_i \in S$, used to represent multi-port cell interfaces.

Cells. A *cell* encapsulates a primitive hardware operation and is written $c = (I_c, O_c, f)$, where I_c and O_c are tuples of input and output signals, and f is the cell’s functional symbol. All hardware components—including single-bit and multi-bit logic gates, arithmetic units (e.g., adders), stateful elements (e.g., registers), structural operators such as **concat** or **extract**, and entire modules or black-box IP blocks—may be uniformly represented as cells.

3.2.2 RTL in Semantic E-Graph

We instantiate the general semantic e-graph framework for RTL hardware using three e-class types: **Wire**, **WireVec**, and **Register**. For simplicity, we omit tuples here; they are only a technical device for cells with multiple output ports and do not affect the core semantic mechanism.

Wire. A **Wire** e-class represents a single-bit signal. Its semantic domain is $\mathbb{D}_{\text{Wire}} = \mathbb{C}_{\text{Wire}} \cup \mathbb{S}_{\text{Wire}}$, where $\mathbb{C}_{\text{Wire}} = \{0, 1\}$ and $\mathbb{S}_{\text{Wire}}$ consists of symbolic wire identifiers drawn from W . Thus a wire’s semantic identifier is either a concrete Boolean value or a symbolic wire (in the implementation, an unsigned integer). There is no decomposer for **Wire**.

WireVec. A **WireVec** e-class represents a bit-vector. Its semantic domain is $\mathbb{D}_{\text{WireVec}} = \mathbb{C}_{\text{WireVec}} \cup \mathbb{S}_{\text{WireVec}}$, where $\mathbb{C}_{\text{WireVec}} = \bigcup_{n \geq 0} \text{Vec}_n(\mathbb{D}_{\text{Wire}})$ and $\mathbb{S}_{\text{WireVec}}$ consists of symbolic vector identifiers drawn from $\bigcup_n \text{Vec}_n$. Thus a vector semantic identifier is either a concrete vector of wire semantic identifiers or a symbolic vector. Unlike **Wire**, **WireVec** is composite. For concrete identifiers $\delta_1 = [x_0, \dots, x_{n-1}]$ and $\delta_2 = [y_0, \dots, y_{m-1}]$, decomposition proceeds via the projections $\pi_i([x_0, \dots, x_{n-1}]) = x_i$, generating component obligations $\{(x_i, y_i)\}_{i=0}^{n-1}$,

where $n = m$ is required since merging only occurs between same-width vectors. Equality of two concrete vectors is thus reduced to equality of aligned lanes.

Register. A **Register** e-class represents a stateful element (e.g., a D flip-flop). It is composite with domain constructor $\mathbf{reg} : \mathbb{D}_{\text{Wire}} \rightarrow \mathbb{D}_{\text{Register}}$ and projection $\pi_1(\mathbf{reg}(w)) = w$, so merging two register identifiers propagates equality to their stored values. Registers introduce sequential feedback, which is handled specially during extraction; see .

Semantic constructors. Each RTL cell f is equipped with a semantic constructor $\llbracket f \rrbracket : \mathbb{D}_{\tau_1} \times \cdots \times \mathbb{D}_{\tau_m} \rightarrow \mathbb{D}_{\tau}$ for its result type τ . The constructor may return a concrete value, a symbolic identifier, or a partially-concrete composite value depending on how much information is available from the inputs.

For **Wire**, a gate can often evaluate even when not all inputs are concrete. For **AND**, the semantic constructor can short-circuit: $\llbracket \mathbf{AND} \rrbracket(0, x) = 0$ and $\llbracket \mathbf{AND} \rrbracket(1, 1) = 1$, while $\llbracket \mathbf{AND} \rrbracket(1, x)$ remains symbolic for an unknown wire x . Thus $\mathbf{AND}(0, w)$ already evaluates to the concrete result 0, whereas $\mathbf{AND}(1, w)$ remains symbolic until the second input is refined further. This also illustrates monotonicity: refining an argument never invalidates a previously computed result. So if an **AND** node has enough semantic information to determine a concrete value, **sadd** reuses or creates the corresponding concrete wire identifier; otherwise it keeps the result symbolic.

For **WireVec**, word-level cells may interpret concrete vectors directly. For example, if $\delta_a = [a_0, \dots, a_{n-1}]$ and $\delta_b = [b_0, \dots, b_{m-1}]$ are concrete vectors, then $\llbracket \mathbf{ADD} \rrbracket(\delta_a, \delta_b)$ evaluates to a concrete output vector of the declared output width. Operationally, this means that **sadd** on a word-level adder may immediately reuse or create a concrete **WireVec** identifier when both operands are concrete.

This choice makes the semantic type of each hardware signal explicit: **Wire** is a

primitive semantic type, `WireVec` and `Register` are composite. In practice, RTLIL designs are translated into this representation by normalizing word-level arithmetic and sequential elements into a consistent form; Section 3.3 describes that translation in the context of the full NEXTMAP workflow.

3.2.3 Better Hardware Reasoning with Semantic E-Graphs

Semantic e-graphs enable forms of hardware reasoning that are difficult to achieve with traditional e-graphs. In hardware, the same operator or value often admits multiple natural representations, and exploiting these relationships is essential for effective optimization. Semantic e-graphs make these relationships explicit through semantic constructors, allowing them to unify equivalent structures across levels of abstraction and to discover additional congruences without requiring bespoke rewrite rules.

The following two examples illustrate these distinct advantages. The first shows how the semantic e-graph simultaneously maintains word-level and bit-level views of an operator, enabling cross-level optimization. The second shows how the semantic e-graph can automatically recover structural equivalences, such as extract–concat roundtrips, that traditional e-graphs would need carefully crafted rewrite rules to express.

Example 3.2.1 (Two-Bit Adder Synthesis) *We illustrate how the semantic e-graph aids hardware reasoning using a simple 2-bit adder synthesis example. Given input vectors $a \triangleright [a_0, a_1]$ and $b \triangleright [b_0, b_1]$, the adder cell $\text{adder}(a \triangleright [a_0, a_1], b \triangleright [b_0, b_1]) \triangleright [s_0, s_1]$ can be rewritten into its bit-level implementation by a synthesis rewrite rule:*

$$\text{adder}(a, b) \longrightarrow [\text{xor}(a_0, b_0), \text{xor}(\text{and}(a_0, b_0), \text{xor}(a_1, b_1))].$$

Such rules can be automatically generated by a generic adder implementation. Under this rewrite, the semantic e-graph merges $s_0 \equiv \text{xor}(a_0, b_0)$ and $s_1 \equiv \text{xor}(\text{and}(a_0, b_0), \text{xor}(a_1, b_1))$,

thus, each output bit of the high-level adder is placed in the same *e*-class as the corresponding bit of its synthesized implementation.

Because both representations are present in the semantic *e*-graph, the adder is now available simultaneously at word level and at bit level. This enables bit-level optimizations that exploit shared substructure. For example, consider a three-bit adder $\text{adder}(a' \triangleright [a_0, a_1, a_2], b' \triangleright [b_0, b_1, b_2]) \triangleright [s'_0, s'_1, s'_2]$. Its least-significant bit also rewrites to $\text{xor}(a_0, b_0)$. Since s_0 and s'_0 both merge with the same $\text{xor}(a_0, b_0)$ *e*-node, the semantic *e*-graph unifies them automatically. Thus common subexpressions, such as low-order sum bits across adders of different widths, are not duplicated but instead shared through the semantic structure of the *e*-graph.

This example shows how the semantic *e*-graph fully exploits multi-level hardware representations, enabling cross-level optimizations that a single-level *e*-graph cannot achieve.

Example 3.2.2 (Extract and Concat) Here, we demonstrate the example originally presented in the motivating overview example. This example is shown visually in the figure that follows. Consider a wire vector in *e*-class *c* with $c \triangleright \delta = [a_0, a_1, a_2, a_3]$. We create two extract *e*-nodes $\text{extr}(v, 0, 1)$, $\text{extr}(v, 2, 3)$, where the extract bounds $(0, 1)$ and $(2, 3)$ are themselves *e*-nodes in the semantic *e*-graph but are shown directly in the operator for readability.

By the semantic constructor of extr ,

$$c_0 \triangleright \delta_0 = \llbracket \text{extr} \rrbracket(v, 0, 1) = [a_0, a_1], \quad c_1 \triangleright \delta_1 = \llbracket \text{extr} \rrbracket(v, 2, 3) = [a_2, a_3].$$

Thus each extract extracts the corresponding contiguous subvector of *v*.

Next, consider the concat *e*-node $\text{concat}(e_0, e_1)$. Its semantic constructor yields

$$c_2 \triangleright \delta_2 = \llbracket \text{concat} \rrbracket(\delta_0, \delta_1) = [a_0, a_1] \parallel [a_2, a_3] = [a_0, a_1, a_2, a_3].$$

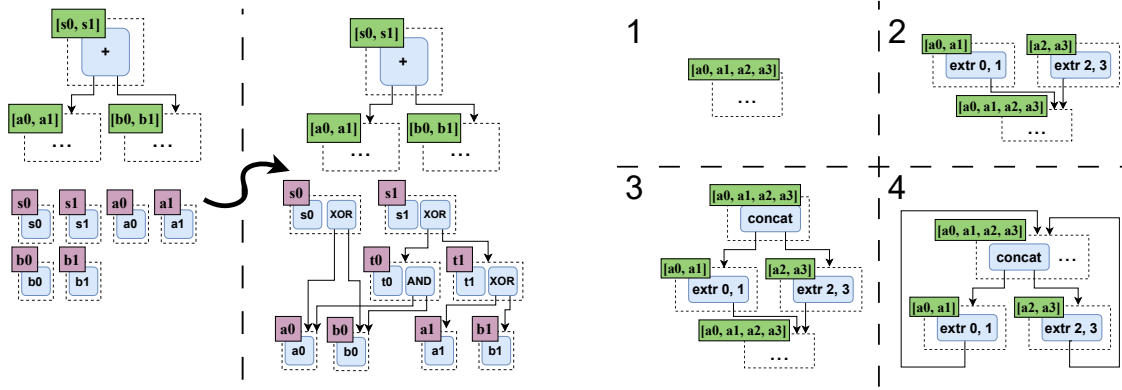


Figure 3.2: Left: Synthesizing an adder. Right: Semantics-based automated unification of nested extracts and concat.

Hence $\delta_2 = \delta$, and the semantic e-graph merges c_2 with the original e-class c . Without any explicit rewrite rule, the semantic e-graph discovers that slicing a vector and concatenating the extracts reconstructs the original value.

This example illustrates how semantic identifiers enable the semantic e-graph to recover such congruences automatically, capabilities that would otherwise require intricate hand-written rewrite rules in traditional e-graphs.

These examples demonstrate how semantic e-graphs enable reasoning across multiple abstraction levels and automatically recover equivalences that would otherwise require extensive rewrite rules.

3.3 Nextmap

We now present NEXTMAP, a technology-mapping and optimization framework that leverages semantic e-graphs to unify synthesis, optimization, and technology mapping.

Conventional mappers operate at a single abstraction level, typically at the bit level or the word level, thereby missing optimization opportunities that span across these

layers. NEXTMAP encodes both the syntactic structure and semantic meaning of hardware components, from individual wires to composite arithmetic cells, within a single SEG, enabling cross-level optimization. This section describes NEXTMAP’s workflow, rewrite strategy, technology mapping, extraction, and database backend.

3.3.1 High-Level Workflow

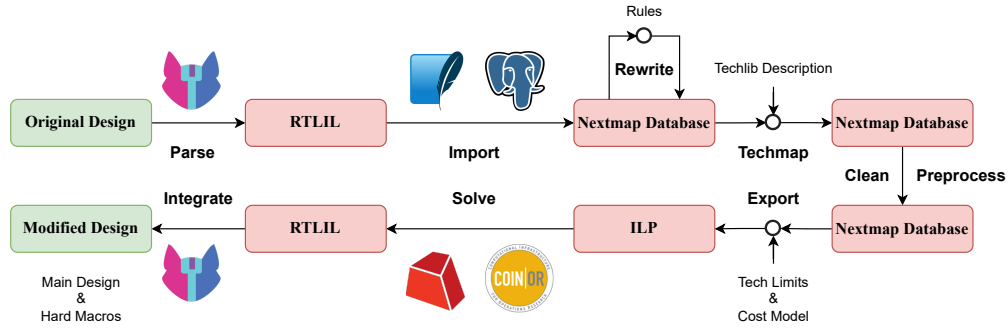


Figure 3.3: Overview of NEXTMAP’s Workflow.

As illustrated in Figure 3.3, NEXTMAP accepts a design in Yosys’s RTLIL format and translates it into our database-backed SEG representation.

After import, users may apply rewrite rules—either the built-in set or user-defined extensions—to saturate the SEG with technology-independent transformations.

Following these rewrites, users provide a technology library (techlib). The library may be specified in JSON for simple cells or in Python for complex or parameterized cells. A techlib defines both the target cells that may appear in the output and the matching procedures that recognize when a region of the SEG can implement one of those cells. Users may also supply technology limits and cost models that capture extraction constraints and optimization objectives.

NEXTMAP then formulates the extraction task as an ILP instance and delegates the optimization to an external solver, such as Gurobi or the open-source CBC solver. After receiving the solver’s result, NEXTMAP interprets the solution and generates the correspond-

ing RTLIL. The final RTLIL includes both the technology-mapped portion—expressed as black-box instances of the target hard macros—and any remaining unmapped abstract cells, which can be handled by downstream synthesis flows.

3.3.2 Technology-Independent Rewrites

NEXTMAP applies a collection of technology-independent rewrites to normalize and optimize hardware designs before technology mapping. These rewrites fall into several built-in rulesets, each targeting a specific class of structural or arithmetic transformations.

Table 3.1: A selection of representative technology-independent rewrite rules in NEXTMAP.

Ruleset	Representative Rule	Description
Arithmetic	Commutative	$(\text{Add } ?a \ ?b) \rightarrow (\text{Add } ?b \ ?a)$
	Associative	$(\text{Add } (\text{Add } ?a \ ?b) \ ?c) \rightarrow (\text{Add } ?a \ (\text{Add } ?b \ ?c))$ when widths and signedness agree
	Width-Reduction	Removes redundant leading zeros in unsigned arithmetic.
	Karatsuba	Splits a wide multiply into three narrower multiplies plus adders/subtractors.
	Partial-Product-Split	Splits a 16×32 multiply into two smaller multiplies plus adders.
Retiming	Forward	Pushes a register forward across combinational logic.
	Backward	Pulls a register backward across combinational logic.
Bitblast	Adder	Expands an adder into a bitwise ripple-carry chain.
	Comparator	Expands $<$, $<=$, $==$ into bitwise compare chains.
	Word-Mux	Expands word-level mux into per-bit selects.
Logic	Double-Negation	$(\text{Not } (\text{Not } ?x)) \rightarrow ?x$
	Idempotence	$(\text{And } ?x \ ?x) \rightarrow ?x$
	Absorption	$(\text{And } ?x \ (\text{Or } ?x \ ?y)) \rightarrow ?x$
	De Morgan	$(\text{Not } (\text{And } ?x \ ?y)) \rightarrow (\text{Or } (\text{Not } ?x) \ (\text{Not } ?y))$

Table 3.1 lists several built-in rulesets along with representative rules from each. The ruleset is organized around the representations that NEXTMAP must connect during optimization. Arithmetic rewrites preserve high-level algebraic structure long enough to expose DSP and datapath opportunities; bitblast and logic rewrites make low-level boolean structure available when that representation is more profitable; and retiming rewrites allow sequential boundaries to move so that these opportunities can be matched in context. For example, width-reduction shrinks operand bitwidths when high bits are constant, while Karatsuba and partial-product splitting expose alternative multiplication

decompositions that may align better with a target library.

To improve scalability, NEXTMAP applies rewrites in stages so that each phase works over a narrower search space. It also scopes expensive rewrites to the structures where they are most useful. For example, retiming is applied to a register and its surrounding arithmetic logic rather than indiscriminately across all gates, which limits e-graph growth while still exposing the sequential transformations needed for mapping.

3.3.3 Technology Mapping

After saturation, NEXTMAP performs techmap by pattern matching against the input techlib.

DSP Mapping For DSP blocks, NEXTMAP uses multiplication as the anchor for matching, since it is the computational core around which DSP datapaths are organized. Starting from each candidate multiplier, the mapper incrementally explores the connected pre- and post-processing logic to determine whether the surrounding structure can be absorbed into a DSP instance, rather than relying on a large enumerated rule set. This basis-driven recursive exploration generalizes across DSP blocks whose internal datapaths are multiplier-centered.

ML Accelerator Blocks For regular structures, such as MAC-based processing elements in machine learning accelerators, NEXTMAP performs hierarchical matching. It first identifies the local compute units—for example, multiply-accumulate cells together with their immediate interconnect—and then combines these matches into larger arrays such as 2×2 and 4×4 meshes. This recursive strategy is more scalable than flat matching because it reuses previously recognized substructures instead of searching for an entire array pattern from scratch.

3.3.4 Extraction

Extraction selects a single, type-consistent hardware implementation from the saturated SEG using an integer linear program (ILP). Unlike traditional e-graphs, SEG distinguishes between *combinational* and *sequential* loops, ensuring that only the latter can be preserved. In the ILP, sequential elements are modeled explicitly to permit valid feedback through registers, while combinational cycles are ruled out through bounded propagation-delay variables. This unified formulation yields precise, technology-aware extractions directly from the SEG.

3.3.5 Database Backend

SEG is implemented atop SQLite because the system relies heavily on relational operations such as joins, aggregation, grouping, and indexed lookups during rewrite application and rebuild. These operations map naturally onto SQL, and SQLite gives explicit control over indexes and query structure while remaining lightweight enough to embed directly in the toolchain. As the evaluation in Section 3.4.4 shows, this pragmatic choice is fast enough to remain competitive with specialized implementations while keeping the backend easy to inspect and extend.

3.4 Evaluation

Here, we evaluate NEXTMAP and the semantic e-graph, addressing the following research questions:

RQ1 How does NEXTMAP’s quality of results (QoR) compare to existing tools?

RQ2 Can we attribute NEXTMAP’s QoR improvements to the simultaneous optimization and mapping enabled by the semantic e-graph?

RQ3 Does the semantic e-graph enable us to build a representation general enough to implement a broad set of EDA tasks?

RQ4 Does the semantic e-graph enable a more efficient representation for EDA tasks?

All experiments were executed on a server running Ubuntu 22.04.5 LTS (Jammy) with Linux kernel 5.15.0-153-generic. The system features two Intel[®] Xeon[®] E5-2420 processors (24 logical cores) and 31 GiB of RAM. The software environment is based on Python 3.10.12. Unless otherwise stated, our default hardware target is the Xilinx UltraScale+ family of FPGAs.

3.4.1 Nextmap’s Quality of Results vs. Existing Tools

To evaluate NEXTMAP’s quality of results against existing tools, we compare NEXTMAP against existing EDA tools: the open-source hardware synthesis tool Yosys Wolf et al. [2013] and a proprietary hardware synthesis tool. We ran all three tools on two sets of benchmarks: one set of small but challenging designs, and one set of large, real-world designs. The small benchmarks were primarily adapted from Xilinx’s user guide examples, and capture edge cases for which proprietary tools are optimized, but that remain difficult for open-source flows. The large benchmarks contain fewer edge cases, but are more realistic in scale. For the large benchmarks, $SA(N,W)$ denotes a systolic array of size $N \times N$ with W -bit weights used for matrix multiplication; $FIR(T,W)$ denotes a finite impulse response filter with T taps and W -bit coefficients; $FFT(P,W)$ denotes a pipelined radix-2² single-path delay feedback FFT of P points with W -bit data. All three families use fixed-point arithmetic. `nerv` is a simple RISC-V CPU design from YosysHQ. The benchmarks range from 1,082 wires ($FIR(16,8)$) to 345,238 wires ($FFT(1024,32)$) before optimization.

Table 3.2: Resource comparison across three tools (NEXTMAP, Yosys, Proprietary) on small but challenging hardware design benchmarks. A tool’s columns are shaded for a row when no other tool uses fewer resources on any reported metric. Multiple groups may be shaded when tools tie. CARRY4 is a specialized logic element which implements a carry-chain, useful for adders and comparison operations. FF is short for flip-flop (register). NEXTMAP is quick ($< 5\text{sec}$) in each case, and so tool timing is not included.

Name	Description	NEXTMAP				Yosys				Proprietary			
		DSP	CARRY4	FF	LUT	DSP	CARRY4	FF	LUT	DSP	CARRY4	FF	LUT
bad_multiplier	16-bit truncated multiplier	1	0	0	0	1 (0%)	0 (0%)	64 (+∞)	0 (0%)	1 (0%)	0 (0%)	0 (0%)	0 (0%)
complex_multiplier	$(a + bi) \times (c + di)$	3	0	64	0	4 (+33%)	16 (+∞)	128 (+100%)	64 (+∞)	4 (+33%)	0 (0%)	64 (0%)	0 (0%)
dot_product	$a \times b + c \times d$	2	0	32	0	2 (0%)	8 (+∞)	64 (+100%)	32 (+∞)	2 (0%)	0 (0%)	32 (0%)	0 (0%)
multiplier_with_rst	16-bit multiplier w/ sync reset	1	0	0	0	1 (0%)	0 (0%)	64 (+∞)	0 (0%)	1 (0%)	0 (0%)	0 (0%)	0 (0%)
redundant_adders	Three adders w/ shared inputs	0	8	0	32	0 (0%)	14 (+75%)	0 (0%)	32 (0%)	0 (0%)	14 (+75%)	0 (0%)	57 (+78%)
signed_mac	$a \times b + c$	1	0	32	0	1 (0%)	8 (+∞)	64 (+100%)	32 (+∞)	1 (0%)	0 (0%)	32 (0%)	0 (0%)
signed_reg	Sign-extension and delay	0	0	32	0	0 (0%)	0 (0%)	32 (0%)	0 (0%)	0 (0%)	0 (0%)	32 (0%)	0 (0%)
square_diff	$(a - b)^2$	1	0	64	0	1 (0%)	5 (+∞)	83 (+30%)	16 (+∞)	1 (0%)	5 (+∞)	32 (-50%)	16 (+∞)
unsigned_mac	$a \times b + c$	1	0	32	0	1 (0%)	8 (+∞)	64 (+100%)	32 (+∞)	1 (0%)	0 (0%)	32 (0%)	0 (0%)
wide_multiplier	16-32-bit multiplier	2	0	17	0	2 (0%)	4 (+∞)	64 (+276%)	15 (+∞)	2 (0%)	0 (0%)	17 (0%)	0 (0%)

For these experiments, all three tools used the same limit for the number of DSPs available. The cost model employed by NEXTMAP was intentionally simple: the cost of logic cells, adders, and registers scales linearly with bit-width, while the cost of multipliers scales quadratically. We equipped NEXTMAP with a rich rewrite rule set consisting of arithmetic, logic, retiming, and DSP techmapping. As described in Section 3.3.3, NEXTMAP only explicitly technology maps to DSPs. The rest of the design is optimized by our arithmetic, logic, and retiming rewrites, but LUT/MUX/flip-flop mapping is left to Yosys.

Results are summarized in for the small designs and

for the large designs. NEXTMAP largely dominates Yosys, i.e., uses equal or fewer resources in all categories, while using fewer resources in at least one category. Although Yosys provides basic passes for mapping multipliers to DSP slices and packing registers into DSPs, it lacks two critical features: (1) retiming transformations that can move registers across combinational logic *in conjunction with DSP techmap*, and (2) comprehensive techmap rules for DSP slices. As a result, Yosys often consumes more logic cells and registers than either NEXTMAP or the commercial tool.

NEXTMAP is also competitive relative to the commercial tool, often tying its results

and occasionally dominating its results. Commercial tools, while equipped with rich vendor-specific techmap rules, are comparatively rigid and often limited to a fixed set of recognizable patterns Smith et al. [2024b]. They are less effective at identifying behaviorally equivalent but syntactically different forms of computation. These results demonstrate that, despite millions of dollars and many engineer-hours of investment, there are still advancements to be made over commercial tools.

NEXTMAP’s runtimes on the larger benchmarks range from 0.3 s to 1938 s. NEXTMAP runtime includes the time spent techmapping with Yosys. Most complete in under a minute: SA(4,8): 1.4 s; SA(4,16): 1.2 s; SA(4,32): 5.9 s; FIR(16,8): 0.3 s; FIR(16,16): 0.4 s; FIR(16,32): 5.7 s; FIR(32,32): 14.3 s; FIR(64,32): 41.2 s; FFT(64,16): 1.5 s; FFT(128,16): 2.4 s; FFT(1024,32): 33.0 s; **nerv**: 8.2 s. Only the two largest systolic arrays take substantially longer: SA(8,32) at 145 s and SA(16,32) at 1938 s. In all cases, the vast majority of time is spent in the ILP solver. Efficient extraction from e-graphs is a well-known issue He et al. [2023], Yin et al. [2025], Goharshady et al. [2024], Cai et al. [2025], and though NEXTMAP contains many optimizations in its ILP formulation, there is still room for improvement in many cases. However, in the vast majority of cases, NEXTMAP’s runtime is more than adequate.

The user-provided technology limits subtly influence the final mapping. In a case study, we observed that NEXTMAP explores a wider range of valid implementations under these limits than the proprietary tool, yielding better results on most resource-constrained targets.

Table 3.3: Resource comparison on larger benchmarks.

Name	NEXTMAP					Yosys					Proprietary				
	DSP	CARRY4	FF	LUTx	MUXFx	DSP	CARRY4	FF	LUTx	MUXFx	DSP	CARRY4	FF	LUTx	MUXFx
SA(4,8)	16	0	128	0	0	16 (0%)	64 (+∞)	448 (+250%)	400 (+∞)	0 (0%)	0 (-100%)	256 (+∞)	448 (+250%)	1648 (+∞)	0 (0%)
SA(4,16)	16	0	256	0	0	16 (0%)	128 (+∞)	896 (+250%)	864 (+∞)	0 (0%)	16 (0%)	0 (0%)	16 (-94%)	0 (0%)	0 (0%)
SA(4,32)	48	594	1568	3568	288	64 (+33%)	720 (+21%)	1792 (+14%)	3848 (+8%)	0 (-100%)	64 (+33%)	448 (-25%)	1280 (-18%)	1776 (-50%)	0 (-100%)
SA(8,32)	192	2426	7200	14432	1152	256 (+33%)	2880 (+19%)	7680 (+7%)	15232 (+6%)	0 (-100%)	240 (+25%)	2904 (+20%)	6656 (-8%)	13456 (-7%)	0 (-100%)
SA(16,32)	768	9629	30752	57488	4608	1024 (+33%)	11520 (+20%)	31744 (+3%)	60928 (+6%)	0 (-100%)	840 (+9%)	19956 (+107%)	29793 (-3%)	101464 (+76%)	0 (-100%)
FIR(16,8)	16	16	105	285	88	16 (0%)	4 (-75%)	240 (+129%)	1011 (+255%)	799 (+808%)	0 (-100%)	255 (+1494%)	240 (+129%)	1537 (+439%)	0 (-100%)
FIR(16,16)	16	32	225	620	184	16 (0%)	8 (-75%)	496 (+120%)	2154 (+247%)	1713 (+831%)	16 (0%)	0 (-100%)	210 (-7%)	0 (-100%)	0 (-100%)
FIR(16,32)	48	384	465	8009	4796	64 (+33%)	464 (+21%)	1008 (+117%)	6176 (-23%)	3576 (-25%)	64 (+33%)	304 (-21%)	434 (-7%)	1697 (-79%)	0 (-100%)
FIR(32,32)	96	752	961	16045	9563	128 (+33%)	912 (+21%)	2016 (+110%)	14024 (-13%)	8880 (-7%)	128 (+33%)	624 (-17%)	930 (-3%)	3457 (-78%)	0 (-100%)
FIR(64,32)	192	1488	1953	35485	23085	256 (+33%)	1808 (+22%)	4032 (+106%)	29801 (-16%)	19641 (-15%)	256 (+33%)	1264 (-15%)	1922 (-2%)	6977 (-80%)	0 (-100%)
FFT(64,16)	6	153	425	958	0	8 (+33%)	185 (+21%)	478 (+12%)	1084 (+13%)	0 (0%)	8 (+33%)	180 (+18%)	553 (+30%)	1165 (+22%)	0 (0%)
FFT(128,16)	9	174	503	1204	35	12 (+33%)	222 (+28%)	571 (+14%)	1395 (+16%)	37 (+6%)	12 (+33%)	200 (+15%)	663 (+32%)	1467 (+22%)	29 (-17%)
FFT(1024,32)	37	982	1482	10410	1227	65 (+76%)	1042 (+6%)	1482 (0%)	10343 (-1%)	2838 (+131%)	64 (+73%)	728 (-26%)	1921 (+30%)	8765 (-16%)	806 (-34%)
nerv	0	895	4183	15132	6919	0 (0%)	904 (+1%)	4163 (>-1%)	14028 (-7%)	7229 (+4%)	0 (0%)	620 (-31%)	2290 (-45%)	8255 (-45%)	63 (-99%)

Table 3.4: Ablation study: NEXTMAP vs. phase-ordered NEXTMAP.

Name	NEXTMAP					phase-ordered NEXTMAP				
	DSP	CARRY4	FF	LUTx	MUXFx	DSP	CARRY4	FF	LUTx	MUXFx
SA(4,8)	16	0	128	0	0	16 (0%)	64 (+∞)	320 (+150%)	304 (+∞)	0 (0%)
SA(4,16)	16	0	256	0	0	16 (0%)	128 (+∞)	640 (+150%)	640 (+∞)	0 (0%)
SA(4,32)	48	594	1568	3568	288	48 (0%)	634 (+7%)	1346 (-14%)	4240 (+19%)	288 (0%)
SA(8,32)	192	2426	7200	14432	1152	192 (0%)	2531 (+4%)	5999 (-17%)	16944 (+17%)	1152 (0%)
SA(16,32)	768	9629	30752	57488	4608	768 (0%)	10134 (+5%)	18126 (-41%)	67808 (+18%)	4608 (0%)
FIR(16,8)	16	16	105	285	88	16 (0%)	32 (+100%)	225 (+114%)	1002 (+252%)	775 (+781%)
FIR(16,16)	16	32	225	620	184	16 (0%)	8 (-75%)	465 (+107%)	2165 (+249%)	1709 (+829%)
FIR(16,32)	48	384	465	8009	4796	64 (+33%)	465 (+21%)	1009 (+117%)	6218 (-22%)	3587 (-25%)
FIR(32,32)	96	752	961	16045	9563	128 (+33%)	913 (+21%)	2017 (+110%)	14077 (-12%)	8903 (-7%)
FIR(64,32)	192	1488	1953	35485	23085	256 (+33%)	1809 (+22%)	4033 (+107%)	29805 (-16%)	19598 (-15%)
FFT(64,16)	6	153	425	958	0	6 (0%)	209 (+37%)	478 (+12%)	1143 (+19%)	0 (0%)
FFT(128,16)	9	174	503	1204	35	9 (0%)	258 (+48%)	571 (+14%)	1479 (+23%)	34 (-3%)
FFT(1024,32)	37	982	1482	10410	1227	49 (+32%)	1022 (+4%)	1482 (0%)	10428 (+<1%)	1638 (+33%)

3.4.2 Nextmap’s Quality of Results vs. “Phase-Ordered” Nextmap

To better demonstrate the benefit of simultaneous optimization, we conducted an ablation study in which we compared NEXTMAP with a “phase-ordered” version of NEXTMAP. Both tools are equipped with the same rewrite rules: synthesis, optimization and techmapping. However, “phase-ordered” NEXTMAP separates out the rewrites by phase and runs each phase separately, extracting a single representative from the e-graph after each stage, and repopulating a fresh e-graph using just that representative in the next stage.

The results are shown in . None of the results from the “phase-ordered” NEXTMAP beat plain NEXTMAP, while plain NEXTMAP dominates “phase-ordered” NEXTMAP in a number of cases. This clearly demonstrates that *simultaneous* optimization and mapping is key to NEXTMAP’s quality of results. Though we provide reasonable cost models for

Table 3.5: NEXTMAP vs. E-Syn on logic synthesis over the EPFL Combinational Benchmarks.

Name	Wires	NEXTMAP			E-Syn		
		Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)
dec	671	351.55	69.63	336	351.55	69.63	399
cavlc	2303	484.52	107.09	41	473.79	114.69	1389
adder	6024	1144.94	1964.39	15	1102.25	2088.60	683
max	14542	2483.50	1854.88	408	2534.35	1775.79	35459

every intermediate extraction in phase-ordered NEXTMAP, they cannot perfectly reflect the optimal extraction choice at each point. Thus, some useful information will always be lost in extraction when moving between phases.

3.4.3 Expressivity of Nextmap

This experiment examines whether NEXTMAP’s representation, enabled by the semantic e-graph, is general enough to implement a range of EDA tasks. To do so, we demonstrate that it is possible to reimplement the equality saturation functionality of a variety of recent equality saturation-for-hardware works (E-Syn Chen et al. [2024] for bit-level optimization, ROVER Coward et al. [2024] for arithmetic optimization, and Churchroad Smith et al. [2024a] for DSP techmapping), plus a new task: mapping to hardware blocks specialized for machine learning. In all cases, we successfully reimplement the equality saturation contributions of prior work using *only* the core NEXTMAP infrastructure described in this paper.

Bit-level optimization We benchmark NEXTMAP against E-Syn Chen et al. [2024] for AIG-based synthesis using equivalent rewrites and ILP-based extraction. Table 3.5 shows four representative examples of varying size. NEXTMAP achieves comparable area and performance results to E-Syn, with consistently shorter runtime, demonstrating native support for bit-level optimization. Note that the hardware target in these experiments is the ASAP7 ASIC PDK Clark et al. [2016].

Arithmetic optimization ROVER Coward et al. [2024] focuses on RTL-level arithmetic and datapath optimizations. As ROVER itself is closed-source, we cannot directly benchmark against its implementation. However, based on the published descriptions, we implemented its core bitvector arithmetic and logic rewrites in our system. For ROVER’s open-source benchmarks, our evaluation includes equivalent or larger designs. We did not implement specialized rule sets such as constant expansion, and thus do not include MCM (multiple constant multiplication) benchmarks in our evaluation.

DSP mapping Churchroad operates at the RTL level, focusing on arithmetic-to-DSP mappings with hardcoded rules in `egglog`, producing feasible but not optimized designs. Using equivalent DSP techmap rules, NEXTMAP matches the mapped outputs of every test case in the Churchroad repository, and additionally enables algebraic and structural optimizations on other designs.

Large accelerator mapping We evaluated NEXTMAP on structured, repetitive computations representative of the systolic-array-based machine learning processors in Achronix and other FPGAs. Using the recursive pattern-matching approach described in Section 3.3.3, NEXTMAP efficiently recognized and mapped large systolic arrays onto various target architectures. For example, a 16×16 matrix-multiplication array was successfully decomposed and mapped onto 16 4×4 MAC meshes in about 40 seconds, maintaining structural consistency across all processing elements. This demonstrates the true range of NEXTMAP. Not only can it implement the lowest, bit-level optimization; it can also capture mapping tasks larger than what most EDA tools generally handle.

Table 3.6: `egglog-vanilla` vs. `egglog-spec`.

Name	Description	Wire Count	Cell Count	<code>egglog-vanilla</code>		<code>egglog-spec</code>		NEXTMAP	
				Build	Rewrite	Build	Rewrite	Build	Rewrite
<code>alu_w32</code>	32-bit ALU	1079	69	3.77 s	0.03 s	0.04 s	0.03 s	0.04 s	0.03 s
<code>multi_alu_n4_w32</code>	Four 32-bit ALUs	4732	276	34.78 s	0.12 s	0.15 s	0.04 s	0.10 s	0.03 s
<code>multi_alu_n8_w32</code>	Eight 32-bit ALUs	9464	552	199.95 s	0.56 s	0.24 s	0.04 s	0.20 s	0.03 s
<code>multi_alu_n16_w32</code>	Sixteen 32-bit ALUs	18928	1104	>600 s (timeout)	N.A.	0.63 s	0.07 s	0.51 s	0.07 s
<code>aes_sbox</code>	256-bit AES S-box	11768	2548	>600 s (timeout)	N.A.	0.82 s	0.01 s	0.73 s	0.01 s

3.4.4 Efficiency of the Semantic E-Graph

To evaluate the efficiency of semantic e-graphs, we implement two simplified versions of NEXTMAP directly within `egglog`, one with semantic e-graphs and one without (`egglog-vanilla` and `egglog-spec`, respectively). Both versions implement the same set of optimization and mapping rewrites. `egglog-vanilla` is the naive implementation described in the motivating overview example which falls victim to the $O(n^2)$ `egglog` inefficiencies. `egglog-spec` (specialized) implements the semantic e-graph’s efficient upward and downward propagation of equalities using custom `egglog` primitives. Given the same ruleset and design, we measure how long it takes each tool to saturate the e-graph. The results are in Table 3.6, with NEXTMAP included for reference. `egglog-spec` clearly dominates `egglog-vanilla`. As the primary difference between these two implementations is the use of the semantic e-graph, this data clearly demonstrates how the semantic e-graph contributes to the efficiency of our representation.

Chapter 4

Conclusion

This thesis has argued that technology mapping should be treated as a semantic compilation problem rather than as a collection of target-specific templates and phase-ordered heuristics. Across memories, logic, and arithmetic, the central challenge is the same: a compiler must preserve the meaning of a design while still exposing the structural choices needed by an implementation substrate. When compilation commits too early to a single representation, many legal and useful implementations disappear from view. When mapping is instead grounded in semantic abstractions, the compiler can reason about families of equivalent realizations and choose among them systematically.

The two systems presented here develop that idea in complementary settings. Fungible memories show that memory mapping can be lifted into the source language itself. By giving designers a uniform way to describe memory behavior, the compiler can derive target-specific organizations without requiring the source program to be rewritten around a particular FPGA primitive or ASIC macro. This makes memory implementation more portable, more retargetable, and less dependent on brittle coding disciplines. Nextmap extends the same perspective to general optimization and technology mapping. By combining equality saturation with semantic e-graphs, it keeps word-level and bit-level

views of a design connected in a single reasoning framework, allowing optimization and mapping to proceed together instead of in isolation.

Taken together, these results suggest a broader direction for hardware compilation. Technology-specific structure should not be hidden behind ad hoc passes, nor should it be pushed entirely onto designers. Instead, compilers should expose semantic interfaces that describe what a target can implement and use principled reasoning to connect those interfaces to source designs. This viewpoint creates a path toward retargetable compilation flows that can accommodate heterogeneous memories, specialized arithmetic units, and emerging accelerator fabrics without demanding a separate hand-tuned methodology for each one.

Several opportunities follow from this work. One is to broaden the range of targets captured by these techniques, especially in settings where memories, datapaths, and control structures must be mapped jointly. Another is to improve cost models and extraction procedures so that semantic search spaces can better reflect physical objectives such as latency, area, energy, and placement constraints. A third is to further unify the ideas in this thesis: memory compilation and semantic e-graph reasoning have so far been developed as separate systems, but both address the same underlying problem of preserving equivalence while navigating implementation choices. Bringing them into a common framework could support richer forms of retargeting across whole-system designs.

More broadly, this thesis supports a simple claim: technology mapping deserves the same kind of semantic and representational care that compiler research has long applied to programming languages. When mappings are expressed through explicit abstractions and equivalence-preserving transformations, the resulting flows are not only more robust, but also more capable. By treating mapping as semantic compilation, we can build hardware toolchains that are easier to retarget, better aligned with designer intent, and more able to exploit the diversity of modern implementation technologies.

Bibliography

Ieee standard verilog hardware description language. *IEEE Std 1364-2001*, pages 1–792, 2001. doi: 10.1109/IEEESTD.2001.93352.

Ameer M. S. Abdelhadi and Guy G. F. Lemieux. A multi-ported memory compiler utilizing true dual-port brams. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 140–147, 2016. doi: 10.1109/FCCM.2016.45.

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference, DAC '12*, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311991. doi: 10.1145/2228360.2228584. URL <https://doi.org/10.1145/2228360.2228584>.

UC Berkeley. Berkeley logic interchange format (BLIF). *Oct Tools Distribution*, 2:197–247, 1992.

Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided*

Verification, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14295-6. doi: 10.1007/978-3-642-14295-6_5.

Yaohui Cai, Kaixin Yang, Chenhui Deng, Cunxi Yu, and Zhiru Zhang. Smoothe: Differentiable e-graph extraction. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1020–1034, 2025.

Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC '24*, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706011. doi: 10.1145/3649329.3656246. URL <https://doi.org/10.1145/3649329.3656246>.

Chen Chen, Guangyu Hu, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. E-morphic: Scalable equality saturation for structural exploration in logic synthesis. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7, 2025. doi: 10.1109/DAC63849.2025.11133110.

Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. Asap7: A 7-nm finfet predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

Samuel Coward, George A. Constantinides, and Theo Drane. Automatic datapath optimization using e-graphs. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 43–50, 2022. doi: 10.1109/ARITH54963.2022.00016.

Samuel Coward, Theo Drane, and George A. Constantinides. Rover: Rtl optimization via verified e-graph rewriting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(12):4687–4700, 2024. doi: 10.1109/TCAD.2024.3410154.

- Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.
- RISC-V Foundation. *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*. RISC-V Foundation, 2025.
- Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. Fast and optimal extraction for sparse equality graphs. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):2551–2577, 2024.
- Mike He, Haichen Dong, Sharad Malik, and Aarti Gupta. Improving term extraction with acyclic constraints. In *EGRAPHS 2023 workshop*, 2023.
- Matthew Hofmann, Berk Gokmen, and Zhiru Zhang. EqMap: FPGA LUT Remapping using E-Graphs. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2025. doi: 10.1109/ICCAD66269.2025.11240672.
- Gerhard Köstler, Werner Kießling, Helmut Thöne, and Ulrich Güntzer. Fixpoint iteration with subsumption in deductive databases. *Journal of Intelligent Information Systems*, 4(2):123–148, 1995. doi: 10.1007/BF00961871.
- Charles Eric LaForest and J. Gregory Steffan. Efficient multi-ported memories for fpgas. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA ’10, page 41–50, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589114. doi: 10.1145/1723112.1723122. URL <https://doi.org/10.1145/1723112.1723122>.
- Charles Eric Laforest, Zimo Li, Tristan O’rourke, Ming G. Liu, and J. Gregory Steffan. Composing multi-ported memories on fpgas. *ACM Trans. Reconfigurable Technol.*

Syst., 7(3), September 2014. ISSN 1936-7406. doi: 10.1145/2629629. URL <https://doi.org/10.1145/2629629>.

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386012. URL <https://doi.org/10.1145/3385412.3386012>.

Charles Gregory Nelson. *Techniques for program verification*. Stanford University, 1980.

Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. Blackparrot: An agile open-source risc-v multicore for accelerator socs. *IEEE Micro*, 40(4):93–102, 2020. doi: 10.1109/MM.2020.2996145.

Yan Pi, Hongji Zou, Tun Li, Wanxia Qu, and Hai Wan. Esfo: Equality saturation for firrtl optimization. In *Proceedings of the Great Lakes Symposium on VLSI 2023*, GLSVLSI '23, page 581–586, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701252. doi: 10.1145/3583781.3590239. URL <https://doi.org/10.1145/3583781.3590239>.

Gus Henry Smith, Colin Knizek, Daniel Petrisko, Zachary Tatlock, Jonathan Balkind, Gilbert Louis Bernstein, Haobin Ni, and Chandrakana Nandi. Scaling program synthesis based technology mapping with equality saturation. *arXiv preprint arXiv:2411.11036*, 2024a.

Gus Henry Smith, Benjamin Kushigian, Vishal Canumalla, Andrew Cheung, Steven Lyubomirsky, Sorawee Porncharoenwase, René Just, Gilbert Louis Bernstein, and Zachary Tatlock. Fpga technology mapping using sketch-guided program synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 416–432, 2024b.

Gus Henry Smith, Zachary D Sisco, Thanawat Techaumnaiwit, Jingtao Xia, Vishal Canumalla, Andrew Cheung, Zachary Tatlock, Chandrakana Nandi, and Jonathan Balkind. There and back again: A netlist’s tale with much egraphin’. In *Workshop on Languages, Tools, and Techniques for Accelerator Design*, 2024c. doi: 10.48550/arXiv.2404.00786. URL <https://doi.org/10.48550/arXiv.2404.00786>.

Wilson Snyder. Verilator. <https://www.veripool.org/verilator/>, 2024.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44(1):264–276, jan 2009. ISSN 0362-1340. doi: 10.1145/1594834.1480915. URL <https://doi.org/10.1145/1594834.1480915>.

Michael Bedford Taylor. BaseJump STL: SystemVerilog needs a standard template library for hardware design. In *Proceedings of the 55th Annual Design Automation Conference, DAC ’18*, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357005. doi: 10.1145/3195970.3199848. URL <https://doi.org/10.1145/3195970.3199848>.

Georgios Tziantzioulis, Ting-Jung Chang, Jonathan Balkind, Jinzheng Tu, Fei Gao, and David Wentzlaff. OPDB: A scalable and modular design benchmark. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1878–1887, 2022. doi: 10.1109/TCAD.2021.3096794.

Ecenur Ustun, Ismail San, Jiaqi Yin, Cunxi Yu, and Zhiru Zhang. Impress: Large integer multiplication expression rewriting for fpga hls. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–10, 2022. doi: 10.1109/FCCM53951.2022.9786123.

Andy Wanna, Samuel Coward, Theo Drane, George A. Constantinides, and Miloš D. Ercegovic. Multiplier optimization via e-graph rewriting. In *2023 57th Asilomar Conference on Signals, Systems, and Computers*, pages 1528–1533, 2023. doi: 10.1109/IEEECONF59524.2023.10476812.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.

Jiaqi Yin, Zhan Song, Chen Chen, Yaohui Cai, Zhiru Zhang, and Cunxi Yu. e-boost: Boosted e-graph extraction with adaptive heuristics and exact solving. *arXiv preprint arXiv:2508.13020*, 2025.

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591239. URL <https://doi.org/10.1145/3591239>.