# EqMap: FPGA LUT Remapping using E-Graphs

Matthew Hofmann, Berk Gokmen, Zhiru Zhang
Cornell University
{mrh259, bg372, zhiruz}@cornell.edu

***Abstract*—FPGA technology mapping is a well-studied problem and has been an area of interest in EDA tool design for decades. In most respects, the computational complexity of technology mapping is understood, and heuristic algorithms have been successfully employed to mitigate compile times. Even with an extensive body of research on technology mapping, our experiments show there is still substantial room for improvement in the quality of results. As a solution, we introduce EqMap, an e-graph driven compiler that can better span the wide gap between SAT-based exact synthesis and heuristic cut enumeration techniques. EqMap's improvements to synthesis produced circuits with 12% fewer LUTs on average over the vendor tools—without ever increasing circuit depth. We also provide an empirical analysis of the runtime of EqMap and show that it is still practical for large designs. Finally, we demonstrate that our compiler infrastructure is reusable, and future work can use our compiler for RTL equivalence checking or auditing the QoR of synthesis tools.**

## I. Introduction

Given the complexity of modern electronic systems, a high degree of automation is required to develop custom hardware within sensible timelines. At the highest level, FPGA and ASIC design flows can be split into logic synthesis and physical design (e.g., floorplanning, placement, and routing). This division of work produces suboptimal designs. Moreover, the individual compilation steps themselves may not be locally optimal. However, circuit minimization problems in general are NP-hard [1], [2], and modern EDA flows use phased compilation and heuristics to bring compile times down to the human timescale while maintaining acceptable quality of results (QoR).

With the end of Dennard scaling and Moore's Law fading, the quality of logic synthesis becomes more important. Hence, future synthesis tools will need to expand the design spaces they explore and find more optimal solutions. Still, finding provably optimal circuits is computationally intractable. In this paper, we introduce a way to augment FPGA technology mapping with e-graph data structures in order to find *more* exact solutions, without significantly increasing compile times.

Technology mapping is the hand-off between logic synthesis and physical design. It converts the abstract Boolean logic into a network of circuit elements that belong to the target cell library. For FPGAs, the primary target cell is the lookup table (LUT). Since every $k$-LUT can be reprogrammed to satisfy any $k$ input Boolean function, FPGA technology mapping has an unmistakably large solution space. Whether the circuit is optimized for latency or area, most FPGA tools approach technology mapping as a graph covering problem [3]–[6]. In the literature, a group of circuit nodes implemented by a $k$-LUT is called a $k$-feasible cut of logic, and the generation of all cuts is called cut enumeration. These structural mapping techniques rely on the topology of the input circuit, and hence they are prone to *structural bias*.

In contrast, functional mappers attempt to decompose the Boolean functionality into smaller sub-functions which can be realized by $k$-LUTs. Such mappers are a more exact approach, and often use SAT solvers to drive synthesis [7], [8]. Other works employ Boolean matching to speed up technology mapping by identifying known Boolean structures [9], [10]. However, exact synthesis tools cannot be scaled past tens of gates. As a consequence, cut enumeration and functional mapping lie on two different extremes. The former is fast but limited by the input structure, while the latter is unbiased but fundamentally unscalable.

For this reason, we propose a technology mapper which uses *e-graphs* as a way to better span the time-QoR spectrum. Equality graphs, referred to as e-graphs, are data structures which use union-find operations to reason about abstract equivalence relations [11]. By using the output of RTL synthesis as an initial solution, we can use e-graphs to incrementally remap the circuit to a more compact form. Whereas typical optimizing compilers apply a greedy sequence of transformation passes, e-graphs rewrite terms in a nondestructive fashion. Our work seeks to evaluate the suitability of e-graphs to superoptimize logic synthesis, while still utilizing the fine-tuned heuristics of existing work as a starting point.

To that end, we propose EqMap: a tool for remapping FPGA netlists into more compact forms—without increasing circuit depth. Our results show many benchmarks, big and small, which synthesize to significantly fewer LUTs over vendor EDA tools. To that end, our work makes the following contributions:

- We formulate an intermediate language and set of e-graph rewrite rules that can explore circuit topologies that heuristic approaches miss.
- We evaluate our compiler against 95 benchmarks combined from three sources: EPFL [12], ISCAS'85 [13], and LGSynth'91 [14]. The results show improvements in LUT count without significant increases to compile time.
- Finally, EqMap is packaged as a Verilog-to-Verilog tool that can be dropped into existing RTL flows. The source code is on GitHub: github.com/cornell-zhang/eqmap.

Before elaborating on our methodology and experimental setup, we first discuss related ideas in technology mapping and e-graph driven compilers. Then, the results section illustrates the typical reduction in LUT count our tool achieves without

increasing circuit depth. Lastly, we discuss the future work of our compiler.

## II. Background

### A. FPGA Technology Mapping

FPGA technology mapping is the compilation step that converts abstract RTL logic into a netlist of lookup tables [3]–[6], [15]. Due to the computational complexity of optimal LUT mapping, most implementations avoid restructuring the input logic and are essentially graph covering algorithms. Given a $k$-LUT FPGA architecture, most mappers start by enumerating the feasible cuts of logic for each node in the circuit. Then, logic cuts are selected to be used in the circuit covering, which ultimately becomes the netlist. Where competing implementations vary, however, is the set of heuristics used to extract the best cuts of logic. The heuristic nature of these algorithms means that results can be inconsistent across implementations, and the discrepancies are difficult to explain.

As an example, Fig. 1 shows two different results of mapping carry-lookahead logic to a 4-LUT FPGA architecture. In Fig. 1a, the circuit is implemented with four total LUTs. Since the sink LUT is already utilizing all four inputs, the ability to combine it with another 3-LUT is non-obvious. Fig. 1b shows how the circuit depth and cell count can both be reduced by merging the two LUTs, because they both share the inputs $A_1$ and $B_1$. This feature in the circuit topology is referred to in the literature as *non-monotone clustering* of logic [3]. In other words, a $k$-feasible cut of logic may contain within itself cuts of logic that are the same size or even larger. That is, traversing subcuts does not decrease the cut size monotonically. Hence, technology mapping must scrutinize this type of circuit topology in order to avoid suboptimal LUT count and depth, adding to tool runtime. It is important to note that this problem occurs equally on 6-LUT FPGAs.

*Not* depicted in Fig. 1 is the bias the graph covering has to the structure of the input logic. Depending on how the input RTL is written, the optimal solution may not be reachable with a cut enumeration algorithm. First, notice that both the AND and OR operations in the circuit are commutative and associative. A poor grouping of terms can limit the efficacy of the technology mapper. In this specific case, it is important that $C_0$ is grouped with $A_0 + B_0$ and *not* $A_1 + B_1$. To summarize, technology mapping must overcome both the difficulty of cut selection and bias toward the structure of the input circuit.

### B. Equality Graphs

Equality graphs, most commonly referred to as *e-graphs*, are an automated reasoning tool built around a union-find data structure [11], [16]. E-graphs are particularly strong at equational reasoning. For example, e-graphs can be used to rewrite mathematical expressions [17] or for automated reasoning about functional programs [18]. Briefly put, e-graphs can drive logic synthesis by exploring other circuit topologies. Initially, each circuit node starts alone in its equivalence class. Then, a set of rewrite rules is used to grow the e-graph with alternative representations. When rewrite rules no longer introduce new information into the graph, we say we have reached *equality saturation*.

Equality saturation is useful for optimizing compilers, because it defers greedy program transformations. Extracting solutions from saturated e-graphs can result in more optimal—sometimes provably optimal—programs. In contrast, traditional compilers use a pass pipeline architecture which suffers from a *phase-ordering problem*. In other words, there is never an ordering of transformation passes that is optimal for all input programs. This is a deep-rooted issue in compiler design, but the problem is particularly consequential for hardware design. The exploratory nature of e-graphs are useful for combinatorial problems like LUT-based technology mapping.

## III. Related Work

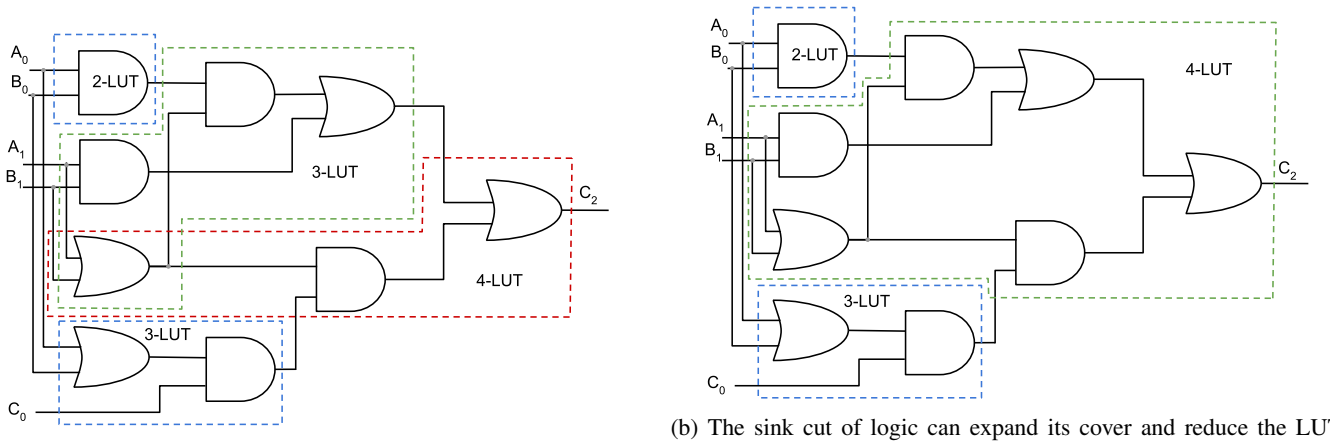
### A. FPGA Tech Mappers

Broadly speaking, FPGA mapping heuristics can be divided into architecture-specific and architecture-agnostic optimizations. Architecture-specific optimizations reduce routing congestion by more efficiently using intra-CLB routing resources. Examples include mapping dual-output functions to fractured LUTs [19] or using dedicated multiplexers to implement functions with more than 6 inputs [20]. In the literature, these types of optimizations are known as *LUT packing*, and the rough optimization goal is to reduce the number of flip-flops driven across CLB boundaries [21].

In contrast, other works attempt to mitigate the structural bias of the technology mapper in general and do not account for the specific architecture. As an example, AGDMap [22] decomposes simple logic gates with large fan-in to enable the exploration of better graph coverings. However, structural bias can take on many forms. Finding advantageous decompositions of logic in general requires more elaborate algorithms [23] that may not be practical for large designs. FlowMap [3] cites non-monotone clustering of logic as the fundamental difficulty that causes bias in LUT-based technology mapping. In other words, this is the observation that a $k$-feasible cut of logic may contain subcuts that are *not* $k$-feasible. Overcoming incidences of non-monotone clustering requires a more elaborate cut-selection algorithm, and our work lays the foundation for a formal, reasoning-based approach to the problem.

### B. E-Graph Superoptimization

In recent years, e-graphs and equality saturation have enjoyed renewed popularity within the compilers field. Several recent works use e-graph driven superoptimization to improve upon existing EDA tool flows. SEER [24] uses e-graphs to optimize and parallelize the control flow of high-level synthesis (HLS) programs. IMpress [25] also uses e-graphs at the HLS level, optimizing the datapath of large bit-width multipliers. At a lower level, ROVER [26]–[28] rewrites arithmetic data paths at the word and bit levels. ROVER straddles the RTL and physical level of abstraction, making it more general purpose than IMpress. In any case, the work




(a) An implementation that uses four LUTs contains redundancy.

(b) The sink cut of logic can expand its cover and reduce the LUT count by one.

Fig. 1: A 2-bit CLA (carry-lookahead) circuit demonstrates that non-monotone clustering can cause suboptimal mappings.

which is most similar in its goals to ours is E-Syn [29]. E-Syn uses the rules of Boolean algebra to rewrite the logic of a circuit before technology mapping. Ultimately, E-Syn is a predictive optimization that takes place during technology-independent synthesis steps, whereas EqMap applies as a post-processing step *after* technology mapping. Furthermore, EqMap's rewriting system models the netlist in terms of total functions, rather than as expressions over a Boolean algebra.

## IV. E-Graph Construction

As an overview, an e-graph is built by accumulating new equivalences through the iterative rewriting of terms. Rewrite rules define the equivalence relations in full generality by defining search patterns of terms to rewrite. This prompts the creation of a grammar that can represent the structure of digital circuits and lends itself well to pattern matching. As an example, one can define De Morgan's laws as a rewrite rule:

```
(NOT (AND x y)) => (OR (NOT x) (NOT y))
```

On the left of => is the *search pattern*. The right-hand side of the rule is the *application*. In the following subsections, we will define our netlist representation, `LutLang`, and the accompanying equivalence relations. While none of EqMap's rewrite rules are particularly novel at face value, together they define a transformation space that can discover compacted circuits. Hence, formalizing our language and rules is critical to both verification and finding deeper insight into the structure of our rewrite rules under composition.

### A. `LutLang` Representation

Input Verilog netlists are converted to our internal format, called `LutLang`, which is compatible with e-graph structures. When printed to text, `LutLang` takes on a Lisp-like syntax and our rewrite rules are written in such style. As an example, a 2-LUT cascaded into a 3-LUT is written as follows:

```
(LUT G x2 x3 (LUT F x0 x1))
```

`G` and `F` are the truth-tables of the LUTs. We also call them the *program* or *function* interchangeably. Truth tables are stored as 64-bit integers, but we analyze them as total functions $F : \mathbb{Z}_2^k \to \mathbb{Z}_2$. To clarify notation, $\mathbb{Z}_2 = \mathbb{Z}/2\mathbb{Z} = \mathbb{B} = \{0, 1\}$. To that end, the denotational semantics $[\![\cdot]\!] : \texttt{LutLang} \to \mathbb{Z}_2$ of a LUT is simply applying its Boolean inputs to the function:

$$[\![\texttt{(LUT F x0 x1)}]\!] = F([\![\texttt{x0}]\!], [\![\texttt{x1}]\!]) \tag{1}$$

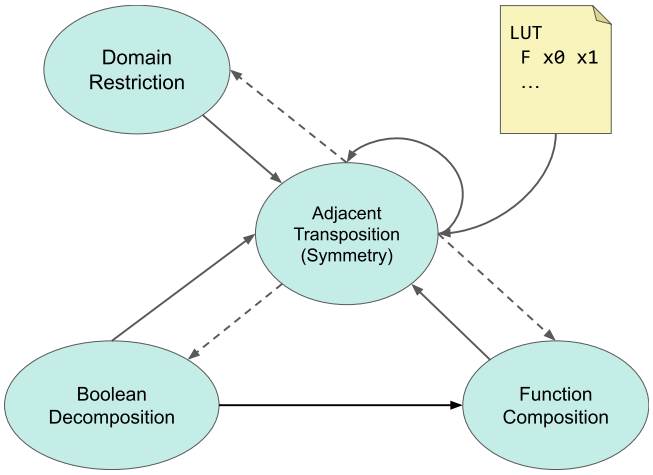



Fig. 2: Transition diagram of rewrite rules. A solid arrow means that the application of the source rule always becomes an instance of the target rule.

### B. Simplifying Degenerate LUTs

**Definition:** A LUT's configuration $F : \mathbb{Z}_2^k \to \mathbb{Z}_2$ is *degenerate* if there exists a Shannon expansion $F = x_i \cdot F_{x_i} + \overline{x_i} \cdot F_{\overline{x}_i}$ such that $F_{x_i} = F_{\overline{x}_i}$ for some input position $i \in \{0, \ldots, k-1\}$. In other words, $F = F_{x_i} = F_{\overline{x}_i}$.

The output of a degenerate LUT is not dependent on one of its inputs. Hence, it can be rewritten into a LUT which uses fewer inputs. This rule is applied by computing the Shannon expansions of LUTs and checking for equivalence. As an example, the rule takes on the following form for $k = 3$:

```
(LUT F x0 x1 x2) => (LUT F′ x0 x1)
    if F(x0, x1, false) == F(x0, x1, true)
    where F′(x0, x1) := F(x0, x1, true)
```

One rule is instantiated for each LUT size $k = 1$ through 6. One should notice that LUTs which are constant functions are also handled by this rule.

### C. Partial Application

A LUT with a constant input can be partially evaluated to a LUT with one less input. It computes the Shannon expansion along the constant variable and chooses the cofactor that matches the state of the constant input. One could show that applying this rule greedily in combination with the previous one is equivalent to constant propagation. As an example, the pseudocode for $k = 3$ is written as follows:

```
(LUT F x0 x1 false) => (LUT F′ x0 x1)
    where F′(x0, x1) := F(x0, x1, false)
```

### D. LUT Symmetries

The semantics of LUTs should not depend on the order of their inputs. If two LUTs have permuted inputs but are otherwise functionally identical, they should belong to the same e-class in the graph. That is, `(LUT F .. xi .. xj ..)` is semantically equivalent to `(LUT G .. xj .. xi ..)` if and only if $G = F \odot \sigma^{-1}$, where $\sigma \in S_k$ is the permutation applied to the inputs.

*Proof.* $\odot$ is a right-action defined for the sake of permuting the inputs to a function before they are applied:

$$\odot : \left(\mathbb{Z}_2^k \to \mathbb{Z}_2\right) \times S_k \to \left(\mathbb{Z}_2^k \to \mathbb{Z}_2\right)$$

$$F \odot \sigma : (x_0, x_1, \ldots, x_{k-1}) \mapsto F(x_{\sigma(0)}, x_{\sigma(1)}, \ldots, x_{\sigma(k-1)})$$

It is trivial to prove that this right-action is associative:

$$(F \odot \sigma_1) \odot \sigma_2 = F(x_{\sigma_2(\sigma_1(0))}, x_{\sigma_2(\sigma_1(1))}, \ldots, x_{\sigma_2(\sigma_1(k-1))})$$
$$(F \odot \sigma_1) \odot \sigma_2 = F \odot (\sigma_2 \circ \sigma_1)$$

With this property, the rest follows directly:

$$F = G \odot \sigma \iff F \odot \sigma^{-1} = (G \odot \sigma) \odot \sigma^{-1} = G \quad (2)$$

□

Therefore, we can conclude that $k$-LUTs have as much symmetry as can be generated by the group $S_k$. This insight has two main consequences. First, it precisely reveals how many e-graph rewrite rules are needed to generate all the symmetries of a LUT. For any $k$-LUT with program $F$, we need exactly as many rules as it takes to generate $F \odot S_k$. It is a well-known fact in algebra that the $k - 1$ adjacent transpositions generate $S_k$ [30].In total, there are $\sum_{k=2}^{6}(k-1) = 15$ rules to encapsulate symmetry for every LUT size. The second consequence is that every other rewrite rule can be defined for one input position, without loss of generality. This reduces the total number of rewrite rules, making it easier to rationalize about the rule system and which types of optimizations are reachable.

### E. Function Composition

Cascaded LUTs can be packed into a single LUT, as long as the size of the cut of logic has at most 6 leaf nodes. This is the crucial observation to LUT remapping. For instance, a circuit that implements $F(x_0, G(x_1, x_2))$ with two 2-LUTs can be rewritten as a 3-LUT that implements some $H(x_0, x_1, x_2)$. In pseudocode, this would take on the following form:

```
(LUT F x0 (LUT G x1 x2)) => (LUT H x0 x1 x2)
    where H(x0, x1, x2) := F(x0, G(x1, x2))
```

The search patterns `x0`, `x1` and `x1` can match any node. They are not necessarily principal inputs, and hence can be outputs from other LUTs. As a consequence, this rule can be chained together many times in different orders to pack a sub-circuit into a single LUT. As an example, this rule would match the 3-LUT and 4-LUT in Fig. 1a and combine them into the single 4-LUT in Fig. 1b. Since the previous rule captures LUT symmetry, we can write compositions for one specific input position, without loss of generality. Therefore, we only need to sweep over the size of the two LUTs in the search pattern. In total, there are $6 \cdot 6 = 36$ LUT packing rules. When the cut of logic is larger than 6 leaves, the rule exits gracefully and does not interfere with reaching equality saturation.

### F. LUTs with Domain Restrictions

**Definition:** A lookup table `(LUT F x0 x1 ...)` is *restricted* if $[\![\texttt{xi}]\!] = [\![\texttt{xj}]\!]$ for some $i, j \in \{0, \ldots, k-1\}$, $i \neq j$. In other words, the domain of the LUT is restricted.

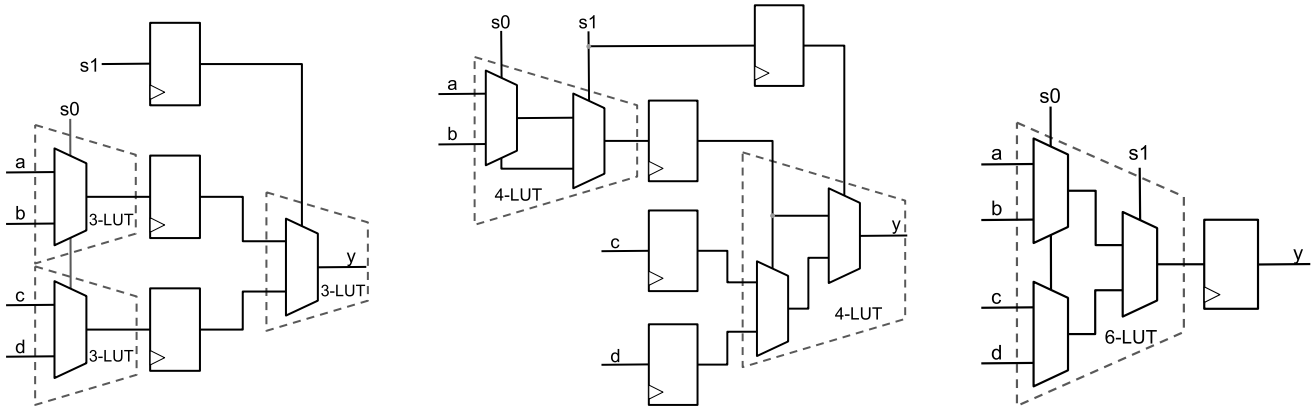
The main advantage of using e-graphs is the compact way in which it represents notions of equality. Whenever a new equivalence is found between two of the inputs to a $k$-LUT, it can be rewritten with a $(k-1)$-LUT. We simply need to define and compute `restrict(F, i, j)` which maps $F : \mathbb{Z}_2^k \to \mathbb{Z}_2$ to the domain-restricted $F|_{x_i = x_j} : \mathbb{Z}_2^{k-1} \to \mathbb{Z}_2$. In pseudocode, the rewrite rule can be written as follows:

```
(LUT F x0 x1 x1) => (LUT F′ x0 x1)
    where F′ := restrict(F, 1, 2)
```

Since rewrite search patterns already match 'modulo' e-class, this rule is automatically triggered when e-classes are merged. For instance, if the e-graph proved in Fig. 1 that $A_i = B_i$, then all the LUTs would shrink by one input—causing the next wave of rewrites to fire. Again, only one rule is needed for each LUT size $k = 2$ through 6, because LUT symmetry is represented in the graph. Considering the entire rewriting system as a whole, Fig. 2 visualizes the ways in which rules can be sequentially composed.

### G. Functional Decomposition

Decomposing boolean functions and logic minimization in general is NP-complete [1]. Correspondingly, decomposing LUTs explodes the size and build time of the e-graph. However, we can still define rewrites that look for fully disjoint decompositions in one or more variables. This rule has no structural element to search for, so it runs every time an e-class is updated. Our implementation computes the Shannon expansion of a $k$-LUT's function $F$ and checks if the first




(a) Three 3-LUT, three FF topology. (b) Two 4-LUT, four FF topology. (c) One 6-LUT, one FF topology.

Fig. 3: Three different circuit topologies for a 4:1 MUX with flip-flop.

variable canalizes or inverts the function. For instance, given $k = 3$ then it is true for $G, H \in \mathbb{Z}_2^2 \rightarrow \mathbb{Z}_2$ that:

$$F(x_0, x_1, x_2) = G(x_0, H(x_1, x_2))$$
$$\Updownarrow$$
$$F(x_0, x_1, x_2) = x_0 \cdot G_{x_0}(H(x_1, x_2)) + \overline{x_0} \cdot G_{\overline{x}_0}(H(x_1, x_2)) \tag{3}$$

In practice, the rule calculates the truth tables that represent the cofactors of $F$ and determines if either represents a constant function or if they are complements of each other.

### H. Register Retiming

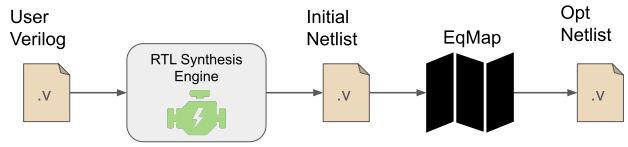
Register retiming is a purely structural rule, meaning it can be implemented with a simple search and apply pattern. An example for $k = 1$ would be written as follows:

```
(LUT F (REG x0)) <=> (REG (LUT F x0))
```

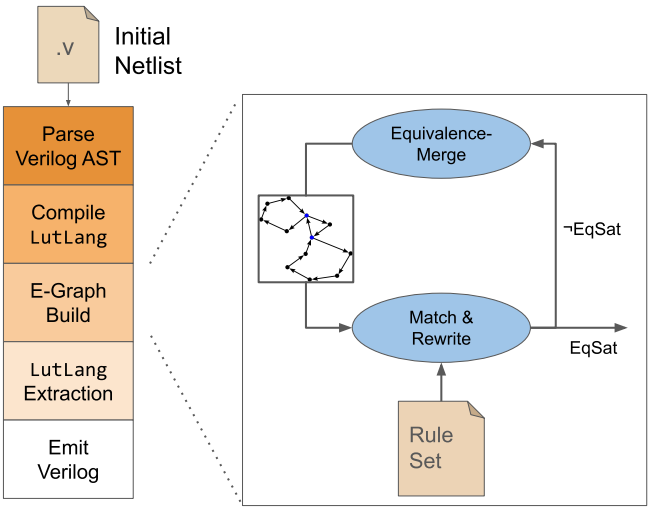
Unlike the other rules, this rule is searched for in both directions. Figure 3 illustrates an example of how register retiming can compose with LUT rewrite rules to reduce LUT count and register count simultaneously. In this case, the 3-LUTs implementing 2:1 multiplexers are pushed across register boundaries. Since this logic happens to have a 6-LUT mapping and a two 4-LUT mapping, we can explore a circuit topology that reduces cell count (Fig. 3c) or adjusts the delay paths (Fig. 3b). To best utilize register retiming, the e-graph extraction technique must have some sense of timing information. Our e-graph extractor is explained in the next section, but it should be noted that in our experiments the area of LUTs and registers are weighted equally.

## V. Tool Flow

While constructing the e-graph is the crux of EqMap, there are several other important components to consider in the full design flow. In order to test our hypothesis, EqMap must be compatible with existing synthesis flows, and the rewritten circuits must be verified. Fig. 4 provides an overview on




(a) The design flow integrating EqMap with existing RTL tools.




(b) The compilation steps internal to EqMap Verilog tool.

Fig. 4: Diagrams of the top-level integration of EqMap into existing tool flows and internal compiler architecture.

both the integration with existing RTL flows and the internal compiler architecture. The following sections will describe each step in more detail.

### A. Extraction

Regardless of whether equality saturation is achieved or not, the quality of the output circuit still largely depends on the extraction technique used. In short, *extraction* is the process of selecting the "best" circuit from the e-graph. Given that a saturated e-graph can contain hundreds of thousands of e-

nodes across tens of thousands of e-classes, a greedy extraction algorithm is the most pragmatic. The greedy extractor iterates over the e-classes, updating the cost of the cheapest e-node until the database of costs no longer change. Whenever possible, our compiler uses the built-in functionality of the egg e-graph Rust library [31]. However, e-graph extraction itself is an ongoing research area [29], [32], [33], and future work should experiment with more capable extraction algorithms. Aside from increases in compile time, better extraction has the potential to raise the QoR yet another level. In any case, the greedy cost of a LUT is always one plus the sum of the costs of its children nodes. Further interactions between extraction and the rewrite rule set are further explained in Section VI-B.

### B. Verilog Support

In order for our compiler to be compatible with as many existing design flows as possible, some level of Verilog support is necessary. Our compiler supports a subset of Verilog 2001 [34], as required to represent structural netlists. This includes support for non-ANSI C style module declarations, wires, and module instantiations with named port connections. With Verilog support, we are able to test EqMap with tool flows that use Yosys [35] or Vivado [36]. On the backend, our compiler also emits an updated Verilog netlist.

### C. Verification

While formal verification is not the primary focus of this work, using e-graphs as a formal reasoning tool helps to build trust in our synthesis results. In fact, e-graphs were originally designed for automated theorem proving [11]. Thus, constructing proofs that demonstrate equivalence between the original and remapped netlist is a built-in feature of EqMap. However, we also use two other independent sources of verification. For combinational netlists, our middle end can do exhaustive functional testing. Lastly, we use Yosys [35] for its SAT-driven equivalence checking capabilities. All in all, the mixed usage of these verification techniques build confidence in the robustness of our technology mapper built around e-graphs.

## VI. Results

Our experiments were carried out on a Red Hat 8 server hosting a Intel Xeon Gold 6242 CPU. Since EqMap is written in Rust, it mostly uses the built-in functionality of the egg library. The egg e-graph runner was ran in a time-limited configuration, meaning there was no limit in the size of the e-graph. Regardless, the median time to saturate a design was 9 seconds. As for the benchmarks, EqMap was evaluated against circuits from three test suites: EPFL [12], ISCAS'85 [13], and LGSynth'91 [14]. However, we also included an ALU and pipelined multiplication module to test how our compiler behaves with increasing levels of bit-parallelism and register pipelining. Finally, we measure how our remapping optimizations influence CLB usage.

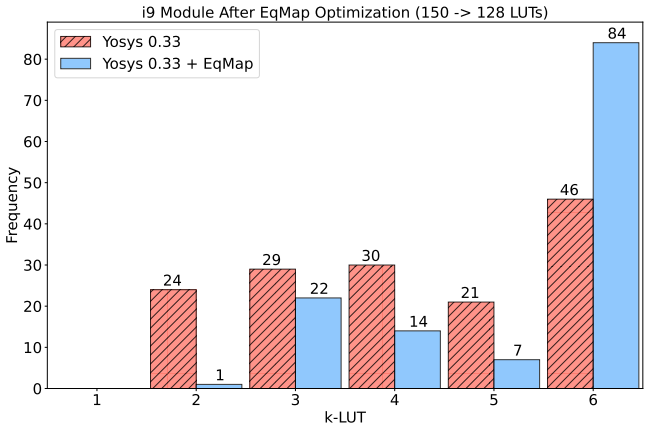
TABLE I: Results of 42 improved benchmarks from ISCAS'85, LGSynth'91, and EPFL. The percent improvements use Vivado as the baseline.

| Module | LUT Count | | |
|---|---|---|---|
| | EqMap | Vivado 2024 | Yosys 0.33 |
| mem_ctrl | 10566 (-9.35%) | 11697 | 11101 |
| div | 5050 (-64.62%) | 14337 | 5319 |
| square | 3497 (-14.46%) | 4088 | 4009 |
| arbiter | 2655 (-3.21%) | 2743 | 2655 |
| sin | 1494 (-7.15%) | 1609 | 1531 |
| voter | 1440 (-10.78%) | 1614 | 1461 |
| max | 798 (-0.25%) | 798 | 842 |
| i10 | 559 (-10.99%) | 628 | 580 |
| k2 | 520 (-0.38%) | 522 | 537 |
| c6288 | 513 (-24.67%) | 681 | 519 |
| c7552 | 328 (-8.38%) | 358 | 333 |
| c5315 | 262 (-6.76%) | 281 | 263 |
| adder | 259 (-20.55%) | 326 | 276 |
| dalu | 216 (-11.11%) | 243 | 237 |
| vda | 204 (-2.86%) | 210 | 270 |
| frg2 | 199 (-9.13%) | 219 | 202 |
| rot | 176 (-6.88%) | 189 | 196 |
| apex6 | 166 (-9.29%) | 183 | 168 |
| x3 | 157 (-5.99%) | 167 | 161 |
| alu4 | 132 (-2.94%) | 136 | 204 |
| i9 | 128 (-3.76%) | 133 | 150 |
| cavlc | 92 (-13.21%) | 106 | 134 |
| c1908 | 89 (-3.26%) | 92 | 94 |
| example2 | 87 (-3.33%) | 90 | 88 |
| c880 | 80 (-12.09%) | 91 | 85 |
| i5 | 70 (-26.32%) | 95 | 70 |
| i2 | 52 (-14.75%) | 61 | 52 |
| router | 48 (-11.11%) | 54 | 54 |
| c432 | 47 (-18.97%) | 58 | 47 |
| i4 | 42 (-22.22%) | 54 | 42 |
| int2float | 35 (-7.89%) | 38 | 46 |
| b9 | 31 (-6.06%) | 33 | 33 |
| alu2 | 30 (-6.25%) | 32 | 117 |
| term1 | 24 (-29.41%) | 34 | 25 |
| pcler8 | 23 (-8.00%) | 25 | 23 |
| comp | 19 (-24.00%) | 25 | 19 |
| sct | 18 (-5.26%) | 19 | 19 |
| f51m | 14 (-12.50%) | 16 | 23 |
| pcle | 13 (-7.14%) | 14 | 13 |
| cmb | 11 (-8.33%) | 12 | 11 |
| x2 | 11 (-8.33%) | 12 | 11 |
| cordic | 9 (-10.00%) | 10 | 9 |

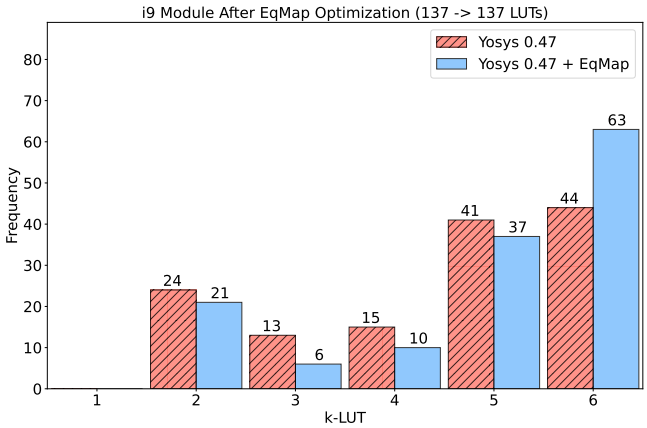
### A. Benchmarking

We used a compilation of 95 combinational benchmarks from three academic sources to test LUT mapping ability. Among the combinational benchmarks we tested, EqMap was able to reduce the LUT count 44% of the time. On average, EqMap packed those netlists to 12% fewer LUTs. The results in Table I list all the reduced LUT counts, sorted by design size. It should be noted that 53 benchmarks did not improve. However, EqMap was always able to produce a circuit of equal size at worst. AMD/Xilinx Vivado 2024 [36] was used as the baseline synthesis tool. For the EqMap flow, Yosys [35] was used to generate the initial mapped circuit. At a glance, circuits like 'int2float,' 'c6288,' 'adder,' and 'square' have the most to gain from EqMap. These circuits are all arithmetic in nature, and this pattern continues throughout the rest of the results.

One result that is *not* demonstrated by the table is the

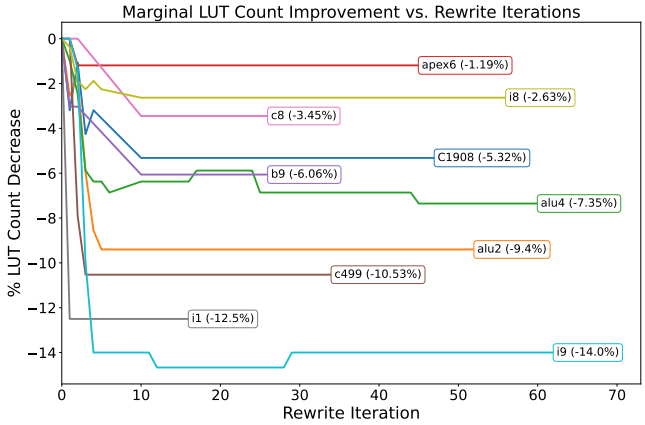



(a) Distribution of LUTs when using EqMap + Yosys 0.33.

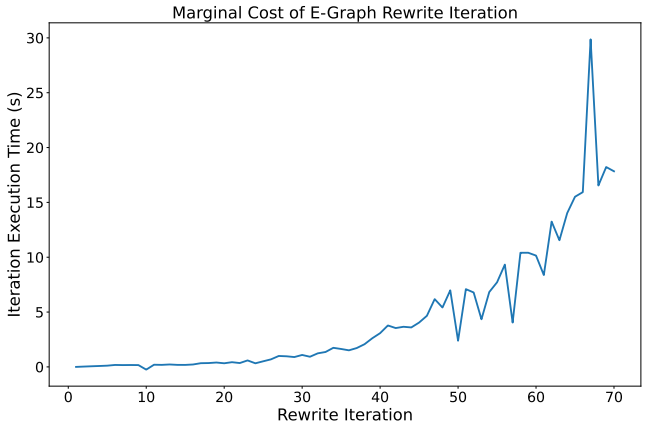



(b) Distribution of LUTs when using EqMap + Yosys 0.47.

Fig. 5: Yosys 0.47 maps 'i9' with fewer LUTs than Yosys 0.33. However, remapping with EqMap inverts the trend.




(a) Marginal improvement in LUT count versus iteration count. The labels mark equality saturation.




(b) Marginal cost in compile time versus iteration count.

Fig. 6: The marginal gain in QoR and marginal cost in time as the e-graph grows in size.

apparent importance of the initial structure inputted to EqMap. We have not eliminated all sources of structural bias, and hence our superoptimization tool still occasionally gets stuck at a local minimum. Fig. 5 illustrates the issue by depicting the different distributions of $k$-LUT usage by different tool flows. In short, an overly packed LUT network will fare worse in attempts to superoptimize it. Future work will investigate which qualities make an RTL synthesis engine work well with our tool versus ones that do not. For example, EqMap optimized Yosys 0.33 netlists (Fig. 5a) better than ones provided by Yosys version 0.47 (Fig. 5b). A future version of EqMap should implement new rewrite procedures than can break out of these local minimums on their own.

### B. Marginal Gain and Cost

Given that EqMap is fundamentally a superoptimization tool, we want to provide evidence that significant gains can be found within reasonable time bounds. To that end, we empirically studied the marginal gains in QoR as increasingly longer rewrite sequences are added to the e-graph. Within the e-graph infrastructure, this phasing of applying rewrites and rebuilding the union-find data structure is referred to as an *iteration*. As shown in Fig. 6a, nearly all the performance gains are discovered within the first 10 iterations—well before equality saturation. Some notable exceptions occur, such as 'alu4' being reduced in size after the 40th iteration. Lastly, the marginal cost of executing an iteration of rewrites becomes prohibitive as the e-graph grows much larger. Fig. 6b illustrates that after approximately the 30th iteration, the added cost in compile time rapidly increases. However, the far majority of results are reached within 20 iterations of rewriting, which on average takes only 3 seconds.

### C. Case Study: Pipelined Designs

Hardware designs with feed forward pipelines provide interesting opportunities to find a higher-level of area optimization. When closing timing on FPGA designs, the critical path is often dominated by routing, more so than ASIC design. Hence, reducing the cell count and circuit depth along the max delay path is a valid optimization strategy for FPGA design. As a caveat, it is important to note that other work has also observed the opposite trend [37]: decreasing depth too much can strain

TABLE II: Post-implementation results of pipelined multiplication circuit optimized with EqMap. Yosys 0.33 + EqMap is used for synthesis, and Vivado 2024 is used for placement and routing.

| × Num. Stages | LUTs | | | Depth | | | CLBs | | |
|---|---|---|---|---|---|---|---|---|---|
| | Initial | EqMap | % Difference | Initial | EqMap | % Difference | Initial | EqMap | % Difference |
| 1 | 982 | 917 | -6.62% | 14 | 14 | 0% | 148 | 146 | -1.35% |
| 2 | 971 | 883 | -9.06% | 21 | 21 | 0% | 137 | 131 | -4.38% |
| 4 | 1041 | 882 | -15.27% | 33 | 31 | -6.06% | 147 | 136 | -7.48% |
| 8 | 1243 | 963 | -22.53% | 53 | 52 | -1.89% | 175 | 155 | -11.43% |
| 16 | 1317 | 917 | -30.37% | 64 | 58 | -9.38% | 191 | 168 | -12.04% |

the router. In any case, EqMap can be utilized to reduce total CLB usage in pipelined designs. To test this hypothesis, our compiler was used to optimize a 32-bit integer multiplier with a varying number of pipeline stages. The data in Table II clearly shows that as the number of pipeline stages increases, an inefficiency in the mapping is accumulated. EqMap is able to repack the LUTs into roughly the same amount of logic as the single stage design—without increasing the number of flip-flops. While our current technique only supports acyclic designs, we wanted to add an additional case study on a pipelined operator to prove the potential of extending our approach beyond combinational logic.

While the drop in CLB usage is desirable, we anticipate that more exact extraction techniques would find even greater area reductions. Unlike other logic rewrites, register retiming changes the topology of the stateful elements—i.e. the flip-flops. Consequently, greedy extraction cannot take into account the fan-out these flip-flops. While these results *do* demonstrate that optimizing for CLB count over raw cell count is a feasible strategy, a different extraction method will be needed.

### D. Case Study: Bit-Parallel Designs

While the academic benchmarks enable direct comparisons to the rest of the literature, the circuits are relatively small. On average, the designs map to 670 LUTs. Among the 42 improved benchmarks in Table I, only six exceed 1000 LUTs. Although this e-graph driven technology provides promising results for smaller benchmarks, FPGA technology mapping is especially difficult—and important—for larger designs By studying a parameterized design, we can demonstrate that EqMap's benefits hold the same with the scaling of logic.

To test how EqMap scales with gate count, we created a synthetic ALU benchmark and varied the input and output bit widths from 8 bits to 4096 bits. Table III lists the LUT counts from the initial synthesis by Vivado 2024, followed by the packed LUT counts. Even though the 1024-bit, 2048-bit, and 4096-bit ALU designs had up to 15,000 LUTs, EqMap was able to achieve up to almost 15% improvement over Vivado within 5 minutes of extra build time. While longer runs lead to slightly better improvements, these results demonstrate how EqMap can achieve area reductions within a short period of time, even when input designs are scaled to beyond 10,000 LUTs. In this specific case, EqMap can be used to audit synthesis tools and perhaps even reverse-engineer adverse behavior. Hence, EqMap proves to be a practical tool to quickly improve a baseline synthesis run.

TABLE III: EqMap synthesis results of $n$-bit ALU

| ALU Bit Width | LUT Count | | |
|---|---|---|---|
| | Vivado 2024 | EqMap | % Improvement |
| 8 | 14 | 14 | 0.00% |
| 16 | 30 | 30 | 0.00% |
| ... | ... | ... | 0.00% |
| 512 | 1540 | 1540 | 0.00% |
| 1024 | 4292 | 3679 | 14.28% |
| 2048 | 8592 | 7363 | 14.30% |
| 4096 | 15516 | 14401 | 7.19% |

## VII. Conclusion and Future Work

While technology mapping has been studied for decades, the end of transistor scaling will require new logic synthesis tools that are less heuristic in nature. This work seeks to demonstrate that there are practical solutions that can bridge the gap between SAT-based exact synthesis and cut enumeration techniques. With EqMap, we can use e-graphs to improve FPGA technology mapping with a post-processing compilation step. Specifically, our results show that EqMap improves synthesis by 12% fewer LUTs on average without increasing circuit depth. While other techniques may approach or beat these gains, equality saturation as a formal method makes e-graphs a particularly trustworthy method by which to transform circuits. Lastly, e-graph construction is decoupled from extraction, meaning this type of flow is particularly adaptive to new optimization objectives.

As for future work, the main research problem is the integration of more sophisticated extraction techniques. The final QoR is still largely contingent on e-graph extraction, and more advanced extraction techniques would likely improve the results even further. While simple greedy extraction clearly serves a purpose, it fails to capture the full potential of EqMap's rewriting system. Nonetheless, EqMap is a promising step towards EDA tools that can better span the gap between exact and fully heuristic logic synthesis.

## VIII. Acknowledgements

This work was supported in part by ACE, one of the seven centers in JUMP 2.0: a Semiconductor Research Corporation (SRC) program sponsored by DARPA and NSF Awards #2019306 and #2403135. In addition, this material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant #2139899.

## References


[1] A. H. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 11, pp. 1319–1332, Nov. 2006.

[2] C. Umans, T. Villa, and A. Sangiovanni-Vincentelli, "Complexity of two-level logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1230–1246, 2006.

[3] J. Cong and Y. Ding, "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[4] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pp. 752–759, 2004.

[5] N.-S. Woo, "A heuristic method for FPGA technology mapping based on the edge visibility," *ACM/IEEE Design Automation Conference (DAC)*, pp. 248–251, 1991.

[6] V. Manohararajah, S. Brown, and Z. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 11, pp. 2331–2340, 2006.

[7] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan, "Efficient SAT-based boolean matching for FPGA technology mapping," *ACM/IEEE Design Automation Conference (DAC)*, pp. 466–471, 2006.

[8] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 871–884, 2020.

[9] H. Savoj, M. J. Silva, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Boolean matching in logic synthesis," *Conference on European Design Automation (EURO-DAC)*, pp. 168–174, 1992.

[10] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast boolean matching based on NPN classification," *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 310–313, 2013.

[11] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 5, no. POPL, Jan. 2021.

[12] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," *International Workshop on Logic & Synthesis (IWLS)*, 2015.

[13] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a targeted translator in FORTRAN," *IEEE International Symposium on Circuits and Systems*, 06 1985.

[14] S. Y. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," *MCNC International Workshop on Logic Synthesis*, 1991.

[15] S. Jang, B. Chan, K. Chung, and A. Mishchenko, "WireMap: FPGA technology mapping for improved routability and enhanced LUT merging," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 2, Jun. 2009.

[16] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pp. 264–276, 2009.

[17] I. Briggs and P. Panchekha, "Synthesizing mathematical identities with e-graphs," *ACM SIGPLAN Int'l Symp. on E-Graph Research, Applications, Practices, and Human-Factors*, pp. 1–6, 2022.

[18] C. Kurashige, R. Ji, A. Giridharan, M. Barbone, D. Noor, S. Itzhaky, R. Jhala, and N. Polikarpova, "CCLemma: E-graph guided lemma discovery for inductive equational proofs," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 8, no. ICFP, Aug. 2024.

[19] D. Dickin and L. Shannon, "Exploring FPGA technology mapping for fracturable lut minimization," *2011 International Conference on Field-Programmable Technology*, pp. 1–8, 2011.

[20] "Ultrascale architecture configurable logic block user guide (ug574)," 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug574-ultrascale-clb/CLB-Overview

[21] T. Ahmed, P. D. Kundarewich, J. H. Anderson, B. L. Taylor, and R. Aggarwal, "Architecture-specific packing for virtex-5 FPGAs," *ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, pp. 5–13, 2008.

[22] L. Fan and C. Wu, "FPGA technology mapping with adaptive gate decomposition," *ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, pp. 135–140, 2023.

[23] A. Mishchenko, R. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks," *2008 IEEE/ACM International Conference on Computer-Aided Design*, pp. 38–44, 2008.

[24] J. Cheng, S. Coward, L. Chelini, R. Barbalho, and T. Drane, "SEER: Super-optimization explorer for high-level synthesis using e-graph rewriting," *ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 1029–1044, 2024.

[25] E. Ustun, I. San, J. Yin, C. Yu, and Z. Zhang, "IMpress: Large integer multiplication expression rewriting for FPGA HLS," *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1–10, 2022.

[26] S. Coward, T. Drane, and G. A. Constantinides, "ROVER: RTL optimization via verified e-graph rewriting," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 12, pp. 4687–4700, 2024.

[27] S. Coward, G. A. Constantinides, and T. Drane, "Automatic datapath optimization using e-graphs," *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pp. 43–50, 2022.

[28] ——, "Automating constraint-aware datapath optimization using e-graphs," *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2023.

[29] C. Chen, G. Hu, D. Zuo, C. Yu, Y. Ma, and H. Zhang, "E-Syn: E-graph rewriting with technology-aware cost functions for logic synthesis," *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[30] I. M. Isaacs and T. Zieschang, "Generating symmetric groups," *The American Mathematical Monthly*, vol. 102, no. 8, pp. 734–739, 1995.

[31] "egg - Rust — docs.rs," 2025. [Online]. Available: https://docs.rs/egg/latest/egg/

[32] Y. Cai, K. Yang, C. Deng, C. Yu, and Z. Zhang, "SmoothE: Differentiable e-graph extraction," *ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 1020–1034, 2025.

[33] A. K. Goharshady, C. K. Lam, and L. Parreaux, "Fast and optimal extraction for sparse equality graphs," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 8, no. OOPSLA2, Oct. 2024.

[34] "IEEE standard verilog hardware description language," *IEEE Std 1364-2001*, pp. 1–792, 2001.

[35] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: An open source framework from verilog to bitstream for commercial FPGAs," *Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[36] "Vivado," 2025. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html

[37] E. Vansteenkiste, A. Kaviani, and H. Fraisse, "Analyzing the divide between FPGA academic and commercial results," *Int'l Conf. on Field Programmable Technology (FPT)*, pp. 96–103, 2015.