# Scaling Program Synthesis Based Technology Mapping with Equality Saturation

Gus Henry Smith, Colin Knizek, Daniel Petrisko
Zachary Tatlock, Jonathan Balkind, Gilbert Louis Bernstein, Haobin Ni, Chandrakana Nandi
gussmith@cs.washington.edu, knizek@berkeley.edu, petrisko@cs.washington.edu,
ztatlock@cs.washington.edu jbalkind@ucsb.edu, gilbo@cs.washington.edu, hn42@cs.washington.edu, cnandi@cs.washington.edu

***Abstract*—State-of-the-art hardware compilers for FPGAs often fail to find efficient mappings of high-level designs to low-level primitives, especially complex programmable primitives like digital signal processors (DSPs). New approaches apply *sketch-guided program synthesis* to more optimally map designs. However, this approach has two primary drawbacks. First, sketch-guided program synthesis requires the user to provide *sketches,* which are challenging to write and require domain expertise. Second, the open-source SMT solvers which power sketch-guided program synthesis struggle with the sorts of operations common in hardware—namely multiplication. In this paper, we address both of these challenges using an equality saturation (eqsat) framework. By combining eqsat and an existing state-of-the-art program-synthesis-based tool, we produce Churchroad, a technology mapper which handles larger and more complex designs than the program-synthesis-based tool alone, while eliminating the need for a user to provide sketches.**

***Index Terms*—technology mapping, equality saturation, sketches, program synthesis, FPGA, DSP**

## I. Introduction

State of the art FPGA hardware synthesis tools such as Xilinx's Vivado or the open-source Yosys [1] often fail to fully utilize the features of complex, programmable FPGA primitives like DSPs [2]. For example, they might fail to pack multiple operations (e.g. a multiply followed by an add) onto a single DSP, instead using a DSP and additional logic resources.

Our recent tool Lakeroad [2] uses *sketch-guided program synthesis* to more fully take advantage of all features of programmable primitives. Sketch-guided program synthesis is a technique which uses SMT solvers to completely search through a space for a solution—in Lakeroad's case, searching through possible DSP configurations. This technique allows Lakeroad to more thoroughly support all features of DSPs. However, Lakeroad suffers from two primary limitations.

**Problem 1:** *User must provide sketches.* Lakeroad requires users to provide *sketches* [3]. These sketches capture a rough outline of the compiled design and guide the SMT solvers in filling the "holes", i.e., missing parts of the design. Writing sketches is both tedious and difficult, requiring knowledge of the sketching DSL in addition to domain expertise of the target being compiled to.

**Problem 2:** *Scalability of SMT solvers.* The open-source SMT solvers used to power sketch-guided synthesis do not scale due to the inherent inefficiency of the underlying bit-blasting algorithm typically used in SMT solvers [4]. This prevents Lakeroad from compiling larger designs or supporting operations like multiplication over large bitwidths.

This paper mitigates both problems with a key insight: a program-synthesis-based tool like Lakeroad should be viewed as a specialized, high-powered subroutine which should only be called on *portions* of the design; furthermore, to orchestrate calls to specialized subroutines like Lakeroad, we can use a framework called equality saturation (or eqsat) [5]–[7]. Eqsat is a term rewriting technique built around a core data structure called an e-graph [8].

This divide-and-conquer design based on eqsat and e-graphs helps mitigate both aforementioned problems. First, by facilitating the application of semantics preserving rewrite rules [5], [9], we can break down large designs (e.g. wide multipliers) and compile them bit-by-bit, leading to more tractable problems for the SMT solvers underlying Lakeroad. Second, we can eliminate the need for users to provide sketches, by instead inferring the structure of the output from information inside the e-graph.

We showcase this idea in Churchroad, a new, open-source[1] technology mapper. Churchroad is built upon the egglog [6] eqsat framework, and calls out to Lakeroad as a specialized subroutine. In the rest of this paper, we demonstrate with a detailed example the core ideas behind Churchroad.

## II. Technical Contribution

To motivate Churchroad and to explain how it works, we use a running example. We first describe our example design, and describe how it should be optimally compiled onto a Xilinx UltraScale+ FPGA. We then explain how a single query to Lakeroad will fail to compile this design. Finally, we explain how Churchroad uses eqsat and e-graphs to break the mapping task into multiple simpler queries to Lakeroad.

Consider a multiplier which multiplies a 16-bit number $a$ with a 32-bit number $b$ and produces the lower 32 bits of the result (all unsigned):

```
module mul(input [15:0] a, input [31:0] b,
           output [31:0] o);
  assign o = a * b;
endmodule
```

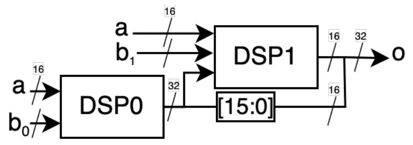
We refer to this high-level, pre-compilation implementation of the design as the *specification* or *spec*. From this point on,

[1]Churchroad is located at https://github.com/gussmith23/churchroad.

we will refer to the spec with math instead of Verilog. For notational convenience, we split $b$ into two 16-bit halves, $b_1$ and $b_0$, where $b = b_1 \mathbin{+\!\!+} b_0$, i.e., the two halves concatenated. Thus, our spec is:

$$\textsc{Spec}\,(a, b) := a \times b \ \ (\text{or } a \times (b_1 \mathbin{+\!\!+} b_0)) \tag{1}$$

To map this design onto a Xilinx UltraScale+ FPGA, it would be ideal to use the UltraScale+'s specialized DSP48E2 primitive, which efficiently implements multiplication. Implementing our design requires two DSP48E2s, arranged in the following way:



where DSP0 computes the lower 16 bits of the multiplication, $(a \times b_0)_0^{15}$, and DSP1 implements the upper 16 bits of the multiplication, $(a \times b_1 + (a \times b_0) \gg 16)_0^{15}$ (extraction of the lower 16 bits is implicit in the diagram), taking advantage of the DSP's internal shifter[2] and the partial product produced by DSP0.

Our previous work [2] demonstrated that existing open-source, pattern-matching-based technology mappers, such as those in Yosys [1], struggle to utilize all features of complex, programmable primitives like DSPs. In response, we introduced Lakeroad, an open-source technology mapper which can more completely use the features of DSPs. However, Lakeroad was only evaluated on small designs which map to single DSPs; in fact, Lakeroad struggles with designs requiring multiple DSPs. We will now explain exactly how Lakeroad will struggle to compile this design, to understand the core challenges which Churchroad solves.

At the core of Lakeroad's greater completeness, correctness, and extensibility is a technique called *sketch-guided program synthesis,* which depends on user-written *sketches.* A sketch is an outline of the compiled result, with *holes* which Lakeroad must fill in to produce a completed design. To compile our example design, we need a sketch which captures the general structure of our 2-DSP output. This sketch currently does not exist within Lakeroad, however, which is **problem 1**: users may need to provide their own sketches, which can be tricky to write and require domain knowledge of what the compiled result should look like.

Once written, the sketch looks like[3]:

```
module two_dsp_sketch(
    input [15:0] a, input [31:0] b, output [31:0] o);
  logic [31:0] dsp0_out, dsp1_out;
  DSP #(<p0>) DSP0 (.a(a), .b(b[15:0]), .o(dsp0_out));
  DSP #(<p1>) DSP1 (.a(a), .b(b[31:16]), .c(dsp0_out),
                   .o(dsp1_out));
  assign o = {dsp1_out[15:0], dsp0_out[15:0]};
endmodule
```

It captures the structure of the compiled output, but leaves the parameters[4] of each DSP as holes <p0> and <p1>, to be filled in by a later step of program synthesis. In mathematical notation, our sketch looks like:

$$\begin{aligned}
&\textsc{Sketch}\,(p_0, p_1, a, b) := \\
&\quad \textbf{let } b_1, b_0 := \ldots \\
&\quad\quad \text{DSP0} := \text{DSP}\,(p_0, a, b_0) \\
&\quad\quad \text{DSP1} := \text{DSP}\,(p_1, a, b_1, \text{DSP0}) \\
&\quad \textbf{in } \text{DSP1}_0^{15} \mathbin{+\!\!+} \text{DSP0}_0^{15}
\end{aligned} \tag{2}$$

Lakeroad then fills in the holes of the sketch, such that the output implements the spec. Specifically, Lakeroad solves the following query:

$$\exists p_0, p_1.\ \forall a, b.\ \textsc{Spec}\,(a, b) = \textsc{Sketch}\,(p_0, p_1, a, b) \tag{3}$$

That is, it searches for a configuration of the two DSPs (captured by $p_0$ and $p_1$) such that $\text{DSP1}_0^{15} \mathbin{+\!\!+} \text{DSP0}_0^{15}$ is equivalent to our spec, $a \times b$.

To find such a configuration, Lakeroad's sketch guided synthesis algorithm (implemented by Rosette [10]) first *guesses* at values of $p_0$ and $p_1$, and then uses an SMT solver to determine whether they work—that is, whether, when used to configure our sketch, they produce a design which implements our spec. To do so, it queries an SMT solver, to verify that $\forall a, b.\ \textsc{Spec}\,(a, b) = \textsc{Sketch}\,(p_0, p_1, a, b)$, or, with both expanded,

$$\forall a, b.\ a \times b = (a \times b_1 + (a \times b_0) \gg 16)_0^{15} \mathbin{+\!\!+} (a \times b_0)_0^{15} \tag{4}$$

If the SMT solver can prove this to be true, then $p_0$ and $p_1$ can be used to configure our sketches and produce an output design. If the SMT solver instead returns a counterexample—that is, values of $a$ and $b$ for which eq. (4) is not true—it can use the counterexample to pick new guesses for $p_0$ and $p_1$.

However, when we attempt to prove eq. (4) with open-source SMT solvers, they take an infeasible amount of time proving this query—**problem 2** in the introduction. To demonstrate this, we implement the query directly within the SMT solver-aided programming language, Rosette [10]:

```
#lang rosette
(define bw 16) ; Larger bitwidths begin to time out!
(define hbw (/ bw 2))
(define-symbolic a b1 b0 (bitvector hbw))
(define b (concat b1 b0))
(verify (assert ; Verify the following assertion:
 (bveq          ; The following are equal:
  (bvmul (zero-extend a (bitvector bw)) b) ; (4) LHS
  (concat                                  ; (4) RHS
   (bvadd (bvmul a b1)
          (extract (- hbw 1) 0
           (bvlshr
            (bvmul (zero-extend a (bitvector bw))
                   (zero-extend b0 (bitvector bw)))
            (bv hbw bw))))
```

[2]The DSP48E2's internal shifter actually shifts by 17 bits, but for rounder numbers in our example, we pretend that it shifts by 16. This example could be converted to a 17-bit by 34-bit multiplication and still work.

[3]We show the sketch here in Verilog for clarity, but the sketch is actually written in Lakeroad's sketching DSL.

[4]Lakeroad's sketches include holes for many of the ports of the DSP as well as the parameters, but to simplify this example, we will simply consider the parameters.

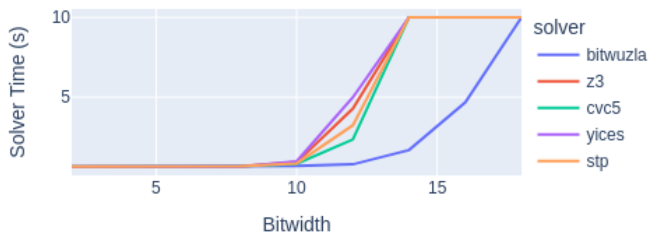


Fig. 1. Solver time to prove eq. (4) with Rosette, at various settings of `bw` and with various solvers. Timeout is set to 10 seconds.

```
    (bvmul a b0)))))
```

By adjusting `bw`, we observe how state-of-the-art SMT solvers behave at different bitwidths. Figure 1 shows our results: while the underlying solvers are able to solve the query for small bitwidths, around 12 bits, most solvers very suddenly hit a wall. At the root of the problem is multiplication. The underlying bitblasting [11] algorithm converts the query into first-order logic using low-level $\lor/\land/\neg$ operators. This conversion causes slowdowns when faced with bitwise-complex operations like multiplication [4].

**Churchroad's first fundamental insight** is to mitigate SMT timeouts by breaking up large queries into multiple smaller queries that are easier for SMT solvers to handle. For example, if we can break eq. (3) into two queries,

$$\exists p_0.\ \forall a, b.\ a \times b_0 = \text{DSP}(p_0, a, b) \text{ and}$$
$$\exists p_1.\ \forall a, b, c.\ a \times b_1 + c \gg 16 = \text{DSP}(p_1, a, b, c)$$

then, after the synthesis procedure picks concrete guesses for $p_0$ and $p_1$, we might end up with queries like

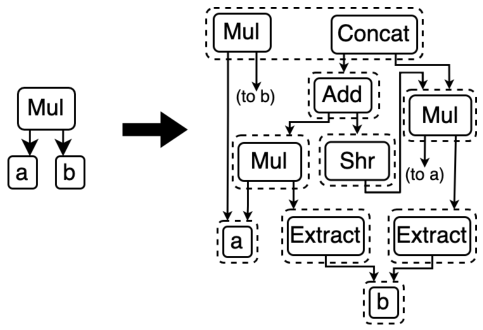
$$\forall a, b.\ a \times b_0 = a \times b_0 \text{ and}$$
$$\forall a, b, c.\ a \times b_1 + c \gg 16 = a \times b_1 + c \gg 16$$

which are trivial for an SMT solver to prove.

To break up queries, Churchroad pre-applies equalities we know to be true. In this example, we'd like to apply the equality in eq. (4) to rewrite the spec $a \times b$ into its expanded form, with the eventual goal of compiling its subexpressions. This equality is not just specific to our example—it is true in general, and is in Churchroad's database of equalities. To apply these equalities, Churchroad utilizes the egglog eqsat framework [6], which provides a data structure called an *e-graph* to hold our design. We can then capture equalities like eq. (4) as *rules* over the e-graph:

```
(rule
 ((= e (Mul arg0 arg1))
  (= (bitwidth arg0) 16) (= (bitwidth arg1) 32))
 ((union e
   (Concat
    (Extract 15 0
     (Add (Mul arg0 (Extract 31 16 arg1))
          (Shr (Mul arg0 (Extract 15 0 arg1)) 16)))
    (Extract 15 0 (Mul arg0 (Extract 15 0 arg1)))))))
```

egglog rules takes two arguments: first, a list of expressions to search for in the e-graph, and second, a list of commands to run if those expressions are found. In this case, this rule looks for a multiplication expression, which it names `e`, and checks the bitwidths of the arguments `arg0` and `arg1`. For this example, we have hardcoded this rule to search for specific bitwidths, but the actual rule in Churchroad works for variable bitwidths. If an expression `e` matching the pattern is found, the rule *unions* it with the expression in the right-hand side of eq. (4). To understand what we mean by *union*, let's see what this rewrite does to our spec. When we apply this rule to our spec (left), it matches on the `Mul` node, and produces the e-graph on the right:



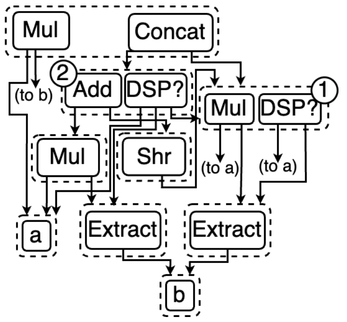
We have simplified the e-graph by dropping some extracts. Note the dotted box around the top-level `Mul` and `Concat` nodes. This indicates that, as a result of the `union` command in our rule above, these nodes are in the same *equivalence class* or *eclass* within the e-graph, and are thus equivalent. The dotted boxes around the other nodes indicate that they are the only nodes in their eclasses.

When the e-graph contained only `(Mul a b)`, the only mapping we could attempt was our ill-fated 2-DSP mapping. Furthermore, attempting this mapping required explicit user input: the user needed to provide the sketch in eq. (2). These sketches are challenging to write, and require the user to have some idea of what the output design should look like (**problem 1**). **Churchroad's second fundamental insight** is that we can entirely remove the need for user-provided sketches. Instead of requiring the user to provide a sketch of the structure of the output, we can instead infer the structure of the output from information already within the e-graph.

To infer the structure of the compiled output, Churchroad searches through the e-graph for expressions that look like they can be implemented using a DSP. For this, Churchroad again uses egglog rules:

```
(rule
 ((= e (Mul arg0 arg1))
  (<= (bw e) 48) (<= (bw arg0) 17) (<= (bw arg1) 17))
 ((union e (DSP? arg0 arg1))))
(rule
 ((= e (Add (Mul arg0 arg1) (Shl arg2 const)))
  (<= (bw e) 48) ... (<= (bw arg2) 36))
 ((union e (DSP? arg0 arg1 arg2))))
```

These rules search for expressions which may be implementable using a DSP. The first rule simply searches for multiplies; the second, for multiply–adds with a shift on the input. Any time these rules fire, they insert a DSP proposal node, "DSP?". These nodes simply mark that the eclass *may* be implementable using a DSP, and that Lakeroad should be used to determine whether such a mapping exists. These rules can fire multiple times on the e-graph above, resulting in the following e-graph:



Each eclass with a DSP? node has been identified by our rules above as potentially implementable using a DSP. To confirm whether or not each eclass can actually be implemented with a DSP, Churchroad now calls Lakeroad as a subroutine on each proposed DSP? node.

Let's begin with the eclass labeled 1. Recall that Lakeroad requires both a spec and a sketch, both of which were provided by the user. However, Churchroad is able to generate both specs and sketches automatically, from the information in the e-graph. This eclass contains two nodes: one node representing a subexpression of the rewritten spec, $a \times b_0$, and DSP? $(a, b_0)$ representing a potential DSP implementation of this eclass. The first node serves as our spec, while the second serves as our sketch! Thus, the synthesis query sent to Lakeroad looks like

$$\exists p.\ \forall a, b.\ a \times b_0 = \text{DSP}\,(p, a, b_0)$$

where the sketch DSP $(p, a, b_0)$ was generated by adding parameters $p$ to the original "DSP?" node. Note that this is much simpler than the query in eq. (3); consequently, the underlying SMT solver is able to solve it, and Lakeroad is able to generate a configuration of the UltraScale+ DSP48E2 which implements $a \times b_0$, which is then inserted into the e-graph.

We can repeat this process for the eclass labeled 2, which produces the following synthesis query:

$$\exists p.\ \forall a, b.\ a \times b_1 + (a \times b_0) \gg 16 = \text{DSP}\,(p, a, b_1, a \times b_0)$$

Again, Lakeroad returns a DSP mapping for this query, which we can insert into the e-graph.

The final step is to extract a compiled design from the e-graph. Currently, Churchroad does this in a straightforward manner: for each eclass, it simply chooses a node which is legal in structural Verilog. For example, module instantiations, zero-/sign-extensions, extractions, and concatenations are allowed, while behavioral operators like Mul are not. In this case, we extract our two DSP instantiations produced by Lakeroad. Finally, Churchroad converts from the internal representation extracted from the e-graph to structural Verilog, producing the design in the block diagram at the beginning of the section. From beginning to end, Churchroad currently takes about *4 seconds* to compile this example.

With that, Churchroad has successfully compiled a design which Lakeroad could not handle. Using eqsat and encoding and certain equalities like eq. (4) as axioms, Churchroad expanded the initial spec. Then, using DSP proposal rules, Churchroad found locations in the expanded spec which might be implementable using a DSP. Finally, these DSP proposals were sent to Lakeroad, which produced mappings to DSPs. Finally, with all of the Lakeroad-produced mappings stored in the e-graph, Churchroad extracted the final compiled design.

## III. Related Work

Programming language and formal methods techniques have been used for verifying hardware designs [12]–[14]. Sisco et al. [15] used sketch-guided program synthesis for "loop rerolling"—this enabled decompilation of low-level netlist to generate high-level HDL. Other papers have also explored the use of sketch-guided synthesis for generating HDL implementations from high-level designs [16], [17]. Tools like ODIN [18] and ODIN-II [19] rely on syntactic matching to map portions of a design to specialized units.

Brain et al. [4] demonstrated that bit-blasting can be optimized by changing the multiplier encoding using either; (i) constant propagation and rewrites; or (ii) polynomial interpolation. Rath et al. [20] have presented an alternative to bit-blasting with PolySAT by providing a theory-solver for bit-vector arithmetic using non-linear polynomials and a bit-vector plugin to the equality graph. Yu et al. [21]'s work enables formal verification of large arithmetic circuits. There is also a functional verification technique for multipliers presented by Yu et al. [22] that uses Galois field (GF) arithmetic to verify an n-bit GF multiplier in n threads, with experimental results up to 571 bits. Dafny supports user-defined axioms to prevent SMT solver timeouts [23]. Haploid [24] is another tool that uses axioms to simplify SMT queries in a preprocessing step using equality saturation.

## IV. Conclusions and Future Work

This work provides early evidence that equational reasoning using e-graphs can help scale program synthesis-based technology mapping techniques. We demonstrated this idea in the form of a new tool, Churchroad, which uses Lakeroad as one specialized subroutine that focuses on specific portions of the design. Using equational reasoning, Churchroad breaks down larger designs, and thus shrinks the queries sent to Lakeroad while eliminating the need for user-provided sketches.

In the future, we are eager to demonstrate how e-graphs can be used to orchestrate more tools like Yosys, Lakeroad, ABC, etc. to further improve technology mapping. We also plan to generate correct rewrite rules for Churchroad, rather than writing them manually.

## Acknowledgements

We thank Zachary D. Sisco, Vishal Canumalla, and Jin Yang for feedback on our work and Thanawat Techaumnuaiwit for helping with the implementation.

## References

[1] C. Wolf, J. Glaser, and J. Kepler, “Yosys-a free verilog synthesis suite,” in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013.

[2] G. H. Smith, B. Kushigian, V. Canumalla, A. Cheung, S. Lyubomirsky, S. Porncharoenwase, R. Just, G. L. Bernstein, and Z. 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*, ser. ASPLOS ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 416–432. [Online]. Available: https://doi.org/10.1145/3620665.3640387

[3] A. Solar-Lezama, “Program synthesis by sketching,” Ph.D. dissertation, USA, 2008, aAI3353225.

[4] M. Brain, “Further steps down the wrong path : Improving the bit-blasting of multiplication,” in *SMT’21: 19th International Workshop on Satisfiability Modulo Theories*, 2021.

[5] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, “egg: Fast and extensible equality saturation,” *Proc. ACM Program. Lang.*, vol. 5, no. POPL, jan 2021. [Online]. Available: https://doi.org/10.1145/3434304

[6] Y. Zhang, Y. R. Wang, O. Flatt, D. Cao, P. Zucker, E. Rosenthal, Z. Tatlock, and M. Willsey, “Better together: Unifying datalog and equality saturation,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591239

[7] G. H. Smith, Z. D. Sisco, T. Techaumnuaiwit, J. Xia, V. Canumalla, A. Cheung, Z. Tatlock, C. Nandi, and J. Balkind, “There and back again: A netlist’s tale with much egraphin’,” *arXiv preprint arXiv:2404.00786*, 2024.

[8] C. G. Nelson, “Techniques for program verification,” Ph.D. dissertation, Stanford, CA, USA, 1980, aAI8011683.

[9] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: a new approach to optimization,” *SIGPLAN Not.*, vol. 44, no. 1, p. 264–276, jan 2009. [Online]. Available: https://doi.org/10.1145/1594834.1480915

[10] E. Torlak and R. Bodik, “Growing solver-aided languages with rosette,” in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013, pp. 135–152.

[11] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View*. Springer Berlin, Heidelberg, 2016.

[12] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, “Kami: a platform for high-level parametric hardware specification and its modular verification,” *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3110268

[13] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, “The essence of bluespec: a core language for rule-based hardware design,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 243–257. [Online]. Available: https://doi.org/10.1145/3385412.3385965

[14] R. S. Nikhil, “Bluespec system verilog: efficient, correct rtl from high level specifications,” *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE ’04.*, pp. 69–70, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:5196158

[15] Z. D. Sisco, J. Balkind, T. Sherwood, and B. Hardekopf, “Loop rerolling for hardware decompilation,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591237

[16] A. Ardeshiricham, Y. Takashima, S. Gao, and R. Kastner, “Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1623–1638. [Online]. Available: https://doi.org/10.1145/3319535.3354246

[17] A. Becker, D. Novo, and P. Ienne, “Sketchilog: Sketching combinational circuits,” 01 2014, pp. 1–4.

[18] P. Jamieson and J. Rose, “A verilog rtl synthesis tool for heterogeneous fpgas,” in *International Conference on Field Programmable Logic and Applications, 2005.*, 2005, pp. 305–310.

[19] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, “Odin ii - an open-source verilog hdl synthesis tool for cad research,” in *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM ’10. USA: IEEE Computer Society, 2010, p. 149–156. [Online]. Available: https://doi.org/10.1109/FCCM.2010.31

[20] J. Rath, C. Eisenhofer, D. Kaufmann, N. Bjørner, and L. Kovács, “Polysat: Word-level bit-vector reasoning in z3,” 2024. [Online]. Available: https://arxiv.org/abs/2406.04696

[21] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, “Formal verification of arithmetic circuits by function extraction,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.

[22] C. Yu and M. Ciesielski, “Efficient parallel verification of galois field multipliers,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 238–243.

[23] T. dafny-lang community, *Dafny Reference Manual*, 2024.

[24] “Haploid: Speed up smt via preprocessing using egg,” 2024. [Online]. Available: https://github.com/IanBriggs/haploid