Surveying FPGA Technology Mapping Completeness

ANDREW CHEUNG, University of Washington, USA

FPGA hardware primitives are complex; high-level operations which traditionally consisted of many low-level gates can now be implemented with single units such as digital signal processors (DSPs). These units are powerful because they can be *programmed* to implement specific behaviors. However, taking full advantage of this programmabilty is difficult. FPGA synthesis tools were originally built to support architectures consisting only of simple primitives such as lookup tables (LUTs). As a result, these tools lack the capability to fully reason about the behavior of complex units like DSPs. This directly affects the quality of compiled designs, as failure to map an eligible design to DSPs can result in orders of magnitude of performance degradation. In this paper, we explore the limitations of current tools with regard to complex primitives—specifically, the Xilinx UltraScale+ `DSP48E2`. We conduct a survey of FPGA synthesis tools targeting the `DSP48E2`, attempting to map a number of common designs onto the DSP using each tool. We present a number of simple designs which existing state of the art tools fail to map, highlighting these tools' inability to completely reason about complex primitives.

Additional Key Words and Phrases: FPGA, Technology Mapping, Hardware Testing

ACM Reference Format:
Andrew Cheung. 2023. Surveying FPGA Technology Mapping Completeness. 1, 1 (May 2023), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Technology mapping — the process of outputting a low-level, FPGA-specific implementation of a high-level design fragment — is a critical step of FPGA synthesis. Traditionally, FPGA synthesis tools were built to support simple FPGA architectures whose primitives consisted primarily of simple LUTs (lookup tables) [1, 7] . Recently, more complex hardware primitives have become commonplace. These primitives, such as DSPs (digital signal processors), present a significant increase in challenge for technology mappers [6]. The UltraScale+ `DSP48E2` has over 100 ports and parameters, and the manual that explains how to configure them properly is over 75 pages long [10]. The large number of ports and parameters is due to the fact that these primitives are *programmable*, i.e., they have complex behaviors which can be controlled by configuring (or programming) their inputs. Ideally, a technology mapper should be *complete* for each primitive; that is, for a given primitive, if a high-level design can be implemented using that primitive, then the technology mapper should find a mapping of the design to that primitive. This is summarized in our "completeness theorem":

$$\forall v \in Verilog, p \in Primitive, [\![v]\!] = [\![p]\!] \rightarrow p \in map(v)$$

A complete technology mapper that satisfies this theorem would allow for the thorough use of any primitive that it supports. But how complete are mainstream technology mappers? No methodology exists to define or measure this. This paper provides the first study to answer these questions.

Author's address: Andrew Cheung, acheung8@cs.washington.edu, University of Washington, Seattle, Washington, USA.

XXXX-XXXX/2023/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
for workload in workloads:
   for bitwidth in [8, 16]:
      for signedness in [True, False]:
         for num_stages in [1, 3]:
            for apply_xor_reduction in [True, False]:
               generate_design(workload,
                  bitwidth, signedness, num_stages,
                  apply_xor_reduction)
```

```
(* use_dsp = "yes" *)
module submulor_2_stage_signed_9_bit(
      input signed [8:0] a,
      input signed [8:0] b,
      input signed [8:0] c,
      input signed [8:0] d,
      output [8:0] out,
      input clk);

   logic signed [17:0] stage0;
   logic signed [17:0] stage1;

   always @(posedge clk) begin
        stage0 <= ((d - a) * b) | c;
        stage1 <= stage0;
   end

   assign out = stage1;
endmodule
```

Fig. 1. Left: Pseudocode of the core algorithm used by Cookie to generate high-level Verilog designs. Right: An example design output by Cookie. `(* use_dsp = "yes" *)` tells the synthesis tool that the design should map to a DSP48E2.

Currently, mainstream FPGA synthesis tools make no guarantee about their completeness across primitives. Yosys [9], an open-source FPGA synthesis tool, leverages automated algorithms to synthesize designs to simple primitives such as LUTs [1]. However, complex primitives such as DSPs are only partially supported with handwritten patterns. The implementation of proprietary tools such as Vivado [11] remains opaque, and therefore cannot be analyzed. However, conversations with industrial teams have confirmed that similar approaches are used in industry.

While handwritten patterns may be sufficient to target simple primitives, it is unclear how they handle the problem of completeness. With such complex primitives that can implement such a wide range of functionalities, completeness is a critical property that is needed in order for these tools to make full use of a primitive. DSPs are units that are computationally powerful, but this power is lost if tools do not map to them effectively. Forum posts from hardware designers suggest a lack of completeness, usually because tools are not efficiently mapping designs down to a single complex primitive. To make matters worse, these posts also imply that hardware designers need to implement designs in a specific coding style to get these tools to recognize the high-level design [2–4]. In summary, current discourse implies that mainstream synthesis tools are leveraging mechanisms such as handwritten patterns that cannot fully reason about the complexity of DSPs.

In this paper, we build Cookie, a testing framework aimed at assessing the completeness of mainstream FPGA synthesis tools as they relate to complex primitives. We find that even for simple operations such as a multiply add followed by a bitwise xor, state of the art tools cannot fully support putting the design onto a DSP48E2.

2 APPROACH AND INSIGHT

Cookie is a testing tool that analyzes the completeness of proprietary tools. We initially scope Cookie to the Xilinx UltraScale+ architecture, and generate high-level designs that should map to the Xilinx UltraScale+ DSP48E2.

According to the DSP48E2 user guide, the primitive should support operations such as multiplication, multiply accumulate, and multiply add. Furthermore, an ALU on the unit should support logical operations such as XOR following an arithmetic operation.

Currently, Cookie assesses a tool's correctness through *exhaustive* generating variants of a single workload. Figure 1 shows the general implementation of Cookie and a sample design generated. It is built in Python and iterates through different variations of these workloads. Variations are generated across three dimensions: bitwidth, number of stages, and signedness.

Table 1. An initial survey of Yosys and SOTA's performance across a series of workloads. The workload column contains the general design, "Signed?" indicates signedness with a ✓ and unsignedness with a X, "# Stages" shows the number of pipeline stages, and Yosys/SOTA/Lakeroad show resource allocation. ✓ symbols in "Verif?" and "Valid?" show that we verify and validate the correctness of Lakeroad's single-DSP designs.

Workload	Signed?	# Stages	Yosys	SOTA	Lakeroad	Verif?	Valid?
`((d + a) * b) | c`	X	1	1 DSP, 20 LUT	1 DSP, 10 LUT	1 DSP	✓	✓
`((d - a) * b) | c`	✓	2	1 DSP, 20 LUT	1 DSP, 10 LUT	1 DSP	✓	✓
`((d - a) * b) ^ c`	✓	3	1 DSP, 22 LUT	2 DSP, 11 LUT	1 DSP	✓	✓
`((d + a) * b) & c`	✓	3	1 DSP, 22 LUT	2 DSP, 11 LUT	1 DSP	✓	✓
`((d + a) * b) ^ c`	X	2	1 DSP, 18 LUT	1 DSP, 9 LUT	1 DSP	✓	✓

3 PRELIMINARY RESULTS

We run preliminary experiments using the canonical designs generated by Cookie on Yosys and a state of the art, industrial-grade FPGA synthesis tool (SOTA). Tools that map these designs should be placed on a single `DSP48E2`, and this is verified through synthesizing these designs using Lakeroad, an FPGA synthesis tool under active development which utilizes program synthesis to map to complex primitives.

Figure 1 shows an example design generated by Cookie. If either Yosys or SOTA fails to map a design onto a single `DSP48E2`, we use Lakeroad to verify that the design can be mapped to a single `DSP48E2`. Because Lakeroad uses program synthesis, its outputs are correct by construction; the design is automatically verified. In addition, we further validate Lakeroad's design through randomized testing with Verilator 5.009 on 10000 input points.

Figure 1 shows a sample of initial results. For each workload, Yosys and SOTA fail to synthesize the operation down to a single `DSP48E2`. The degree of failure varies across designs — in the best case, SOTA only adds an additional 9 LUTs, yet in the worst case it uses an additional DSP and 11 LUTs. It is clear from this initial survey that mainstream tools have obvious holes in completeness. Therefore, the creators of these tools could use Cookie's testing paradigm to improve the quality of their designs.

4 FUTURE DIRECTIONS

This initial survey has revealed that even for simple workloads, existing tools cannot fully utilize complex programmable primitives. In addition to the dimensions of variation shown in this paper, coding style is an additional variation worth exploring. Hardware designers should not have to adhere to a specific coding style for tools to map their designs. For example, writing a multiply as $b * a$ instead of $a * b$ should not significantly alter the output of the tool. To explore this dimension of change, we are working on extending Cookie to explore generating equivalent implementations of a workload. Mutant generation is a promising technique for this purpose [5].

This approach is novel because it differs from traditional mutation testing. Typically, mutation testing relies on the pruning of "equivalent mutants", i.e., variations of programs that are functionally equivalent [8]. However, Cookie will only generate equivalent mutants to assess for the difference in coding style. Currently, Cookie generates "easy", canonical workloads. This provides a foundation from which we can apply mutation testing to assess the variation in implementation that occurs in the real world. Therefore, in future work we will assess how equivalent variations also may cause these tools to fail.

REFERENCES

[1] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 24–40.

[2] Vivado Forums. 2019 [Online]. Vivado not optimizing my design to fully utilize DSP48e1. online. https://support.xilinx.com/s/question/0D52E00006hpmpKSAQ/vivado-not-optimizing-my-design-to-fully-utilize-dsp48e1?language=en_US Accessed 2022-05-25.

[3] Vivado Forums. 2022 [Online]. Vivado DSP48E2 Synthesis without primitive. online. https://support.xilinx.com/s/question/0D52E000074N7W0SAK/vivado-dsp48e2-synthesis-without-primitive?language=en_US Accessed 2022-05-25.

[4] Vivado Forums. 2023 [Online]. Can not correctly infer A*B+C to DSP48E2. online. https://support.xilinx.com/s/question/0D54U00006AqPXFSA3/can-not-correctly-infer-abc-to-dsp48e2?language=en_US Accessed 2022-05-25.

[5] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[6] Chris Lavin and Alireza Kaviani. 2018. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 133–140.

[7] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. 2007. Improvements to Technology Mapping for LUT-Based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2007), 240–253. https://doi.org/10.1109/TCAD.2006.887925

[8] David Schuler and Andreas Zeller. 2013. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability* 23, 5 (2013), 353–374.

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

[10] Xilinx. 2021. UltraScale Architecture DSP Slice User Guide. https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp.

[11] Xilinx. 2022 [Online]. Vivado User Guide. online. https://docs.xilinx.com/v/u/cdFc3m34xWZu~mVWdoFb_g Accessed 2022-05-01.