Can Formal Outsmart Synthesis: Improving Synthesis Quality of Results through Formal Methods

Eldon Nelson M.S. P.E. (eldon_nelson@ieee.org)
Synopsys, Inc.
690 East Middlefield Road
Mountain View, CA 94043

***Abstract*-Synthesis is the process of taking a description of a hardware design, typically in the form of an HDL (hardware description language), and transforming that description to a gate-level representation. The process of Synthesis includes Boolean reduction techniques that can reduce the number of gates required to implement the design. Reduction of gates in the final product can mean less area on the die as well as reduced power consumption - both desirable effects. Formal techniques can look at the functional behavior of an HDL design and rigorously prove how that HDL design truly operates – with a model for its valid stimulus. This paper employs novel Formal techniques to identify logic that may be removed through analysis of the functional operation of the design. The novel part of this approach is to not only do Formal Coverage Analysis of a design. But, that through this approach, it is possible to do rapid design experiments to see if certain architectural decisions can result in a reduction in synthesized gates, area and power. It may become apparent that certain features of a design account for outsized numbers of gates to implement. After finding these outsized design features, this novel process innately contains a semi-automated process to precisely remove specific design features from the RTL or from the gate-level netlist.**

I. Introduction

Synthesis is the process of taking a description of a hardware design, typically in the form of an HDL (hardware description language) and transforming that description to a gate-level representation. The process of Synthesis includes Boolean reduction techniques that can reduce the number of gates required to implement the design. Reduction of gates in the final product can mean less area on the die as well as reduced power consumption - both desirable effects.

The typical way that gates can be safely removed during Synthesis is through constant propagation whereby the external inputs to the design may be tied to either logic-high (`set_logic_high` as used in the Synopsys synthesis tool) or logic-low (`set_logic_zero`). The Synthesis tool assumes that those inputs are static and Boolean reduction techniques can be used to remove unreachable logic. However, constant propagation cannot consider interactions of input signals that have more complicated relationships.

Formal Analysis of a design is a deep and multi-faceted topic. It includes the traditional concept of using Formal to create proofs of certain custom properties created by the user to prove functional correctness. There are, however, more specialized applications of Formal Analysis such as FCA (Formal Coverage Analysis). FCA is a technique that allows for proving the reachability of the different types of Code Coverage of a design such as: Line, Toggle and more.

FCA can, most simply, identify if a line of an HDL is reachable dependent on how the input stimulus is constrained. What you may be sensing is the trajectory between FCA and Synthesis. FCA has a crucial and elegant advantage over Synthesis: FCA has the advantage of the SystemVerilog "`assume`".

The examples used in this paper are available on the GitHub project "Formal Synthesis" below. Type "`make help`" to learn more once downloaded.

```
https://github.com/tenthousandfailures/formal_synthesis
```

II. Approach

By using FCA and providing the required SystemVerilog [4] "`assume`" statements to model the DUT (Design Under Test) input stimulus, it is possible to model the relationships of the stimulus that the design will see under functional operation. These "`assume`"(s) are the same that are used for Formal Analysis. Since we can exhaustively, and confidently, use FCA to see what logic is unreachable dependent on our stimulus, modeled through the SystemVerilog "`assume`" statement, we can create reports for the simple case of what Lines of our design HDL are unreachable.

Unreachable lines of our HDL may be removed – and with that removal comes the reduction in gates we are seeking. In this paper's most basic application, simply running FCA and commenting out lines of HDL that are unreachable can result in reduction of gates, before or after Synthesis, in a way that retains functionality and matches what the stimulus constraints given.

The process is continued by analyzing toggle analysis with FCA. It was chosen to use FCA toggle analysis on the synthesized netlist resultant from the first step (Synthesis 1 in Figure 1). The reasoning behind doing this analysis at the synthesized netlist is shared during the Results section. Internal and external points in the design that cannot toggle may be tied to logic-1 or logic-0 in the synthesized netlist. Then Synthesis may be run again on the synthesized netlist to arrive at the final optimized design as in Figure 1.

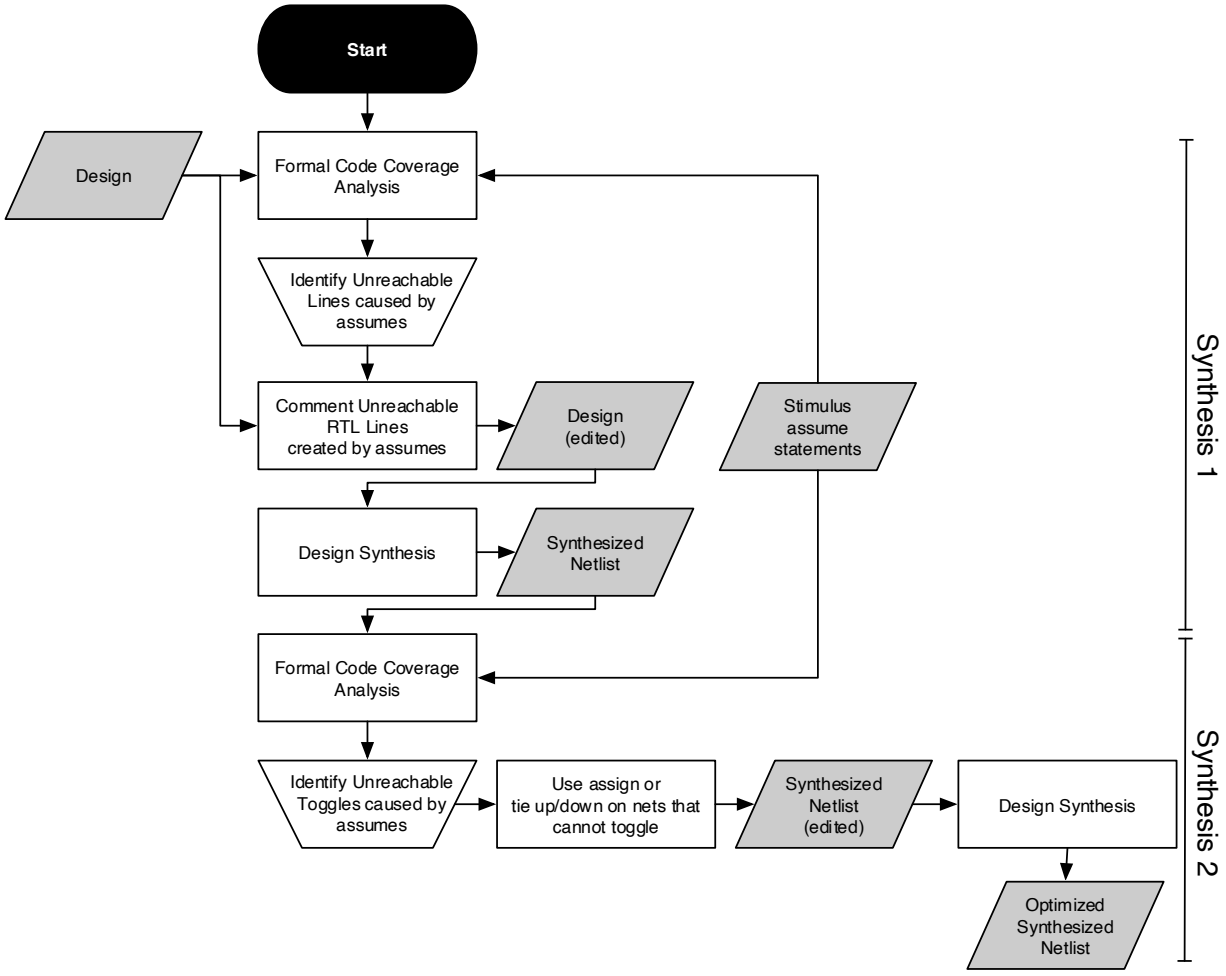

Figure 1. Optimization Flow Chart with Formal

III. RESULTS

The first application of this technique for illustration is using the TinyAlu [2] that is featured in Ray Salemi's book "The UVM Primer" [1] which has its HDL available on GitHub [5]. The TinyAlu has a SystemVerilog implementation that is synthesizable[1] and represents an ALU (Arithmetic Logic Unit) that does single-cycle operations such as: bitwise-and, addition and bitwise-xor. It can also do a three-cycle operation of multiply. The block diagram for the TinyAlu is in Figure 2.

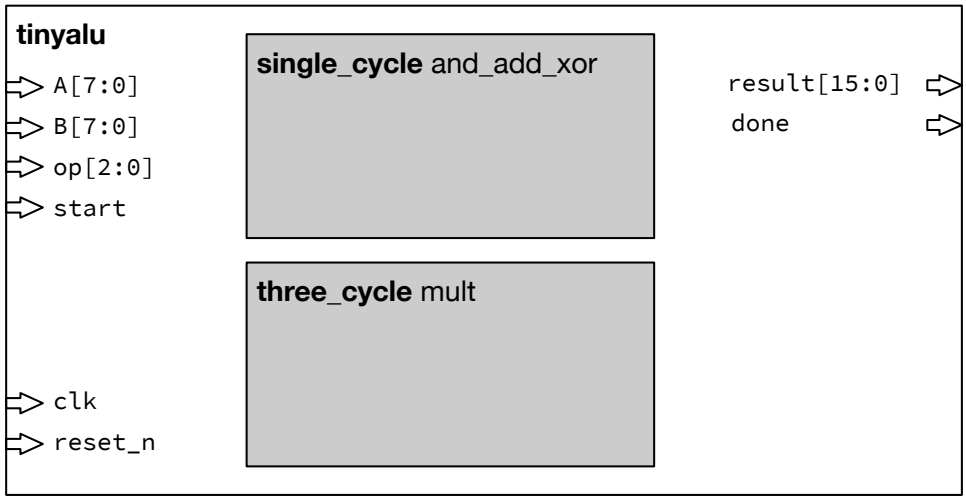

Figure 2. Block Diagram of TinyAlu

This paper is accompanied by examples and a Makefile to run though the examples. The first Makefile target we will use is:

```
make tinyalu
```

Figure 3. Provided Makefile Target tinyalu

which will launch a Formal tool with the setup required to run FCA analysis on the TinyAlu design. This design comes with many provided stimulus `assume` statements that can be toggled on or off to do different design analysis.

[1] With minor modifications from the original and provided with the paper.

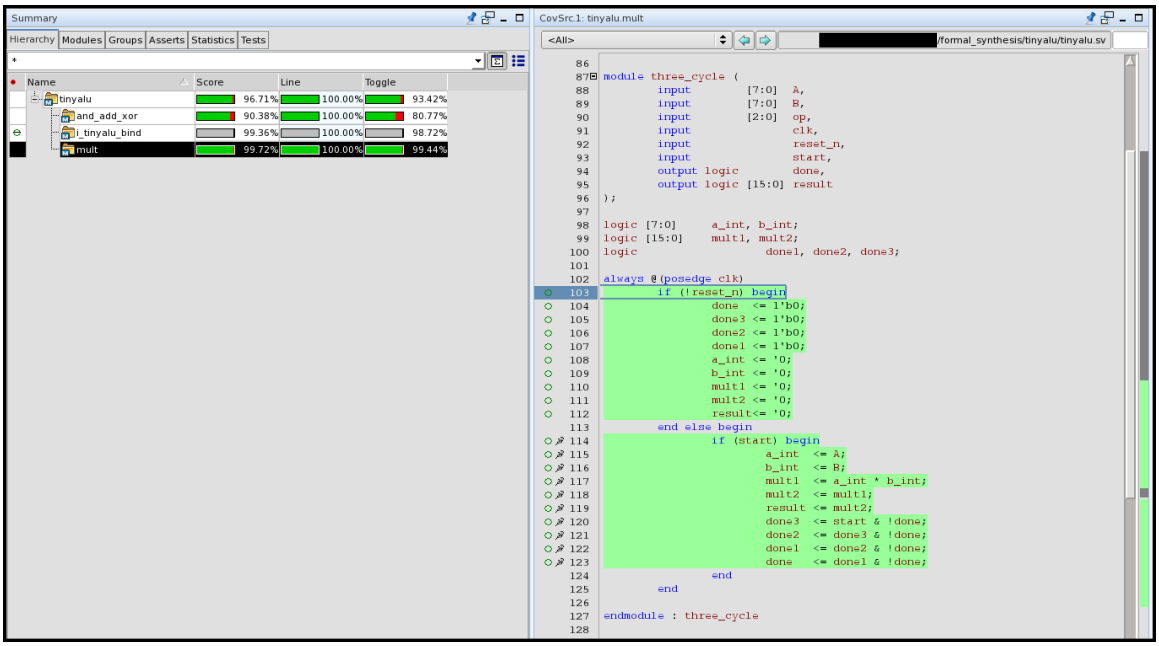

Figure 4. Formal FCA Analysis Shows that All Line(s) of Code Are Reachable on module three_cycle

When we run FCA on this design it shows all Lines of the design are reachable (100%) in Figure 4 with all of the user defined stimulus assumes disabled. This means that there is not anything in the design that can be removed – at least from the simple standpoint of FCA Line Coverage with no assumes.

The single-cycle operations and multi-cycle operations of the ALU are provided in Table 1.

Table 1. Operand Commands (edited from original)

Operand	Value	bit[2]	bit[1]	bit[0]
OP_NULL	3'h0	0	0	0
OP_MULT	3'h1	0	0	1
OP_AND	3'h2	0	1	0
OP_ADD	3'h3	0	1	1
OP_XOR	3'h4	1	0	0

We may know, from the context of how this module is used or from a request to reduce gate count, that we will not be doing any of the multiply (`OP_MULT`) operations for this design application. A design requirement change is a common occurrence in design teams. Changing requirements and schedule pressure can be manifested in design changes that need to be implemented. Luckily, we can apply a SystemVerilog "`assume`" on the design to constrain the inputs to say that we will not accept, and not operate correctly when, the operand "`OP_MULT`" (multiply) is given. We can describe this condition simply with the SystemVerilog "`assume`" statement in Figure 5.

```
always @(*) begin
     only_single_cycle_ops : assume (op != OP_MULT);
end
```

Figure 5. SystemVerilog assume Statement Applied to disable the OP_MULT operand from tinyalu/tinyalu_sva.sv

With "`only_single_cycle_ops`" assume active and then running FCA Line analysis, we can see the results reported by the Formal Tool showing what RTL Lines could be removed based on the functional behavior in Figure 6.

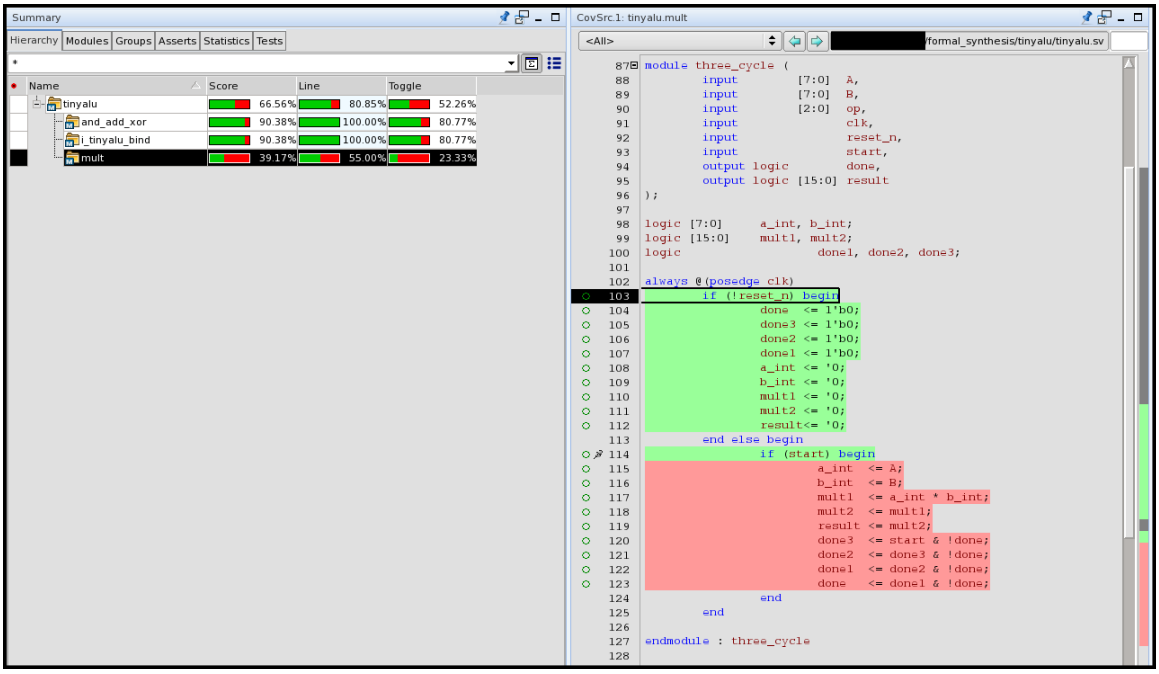

Figure 6. FCA Showing Lines in Red from the Multiplier which Are No Longer Reachable

The red lines represent unreachable Lines in Figure 6 can be removed by hand or through post-processing of the Formal analysis. Simply commenting these red lines would result in the removal of a multiplier "*" that would otherwise be Synthesized along with many other logical signals. Formal applications can give a picture of the design complexity which can give rough numbers of what Synthesis may look like through a Complexity Report in Figure 7 shows the removal of the MULTIPLY after the Formal suggested code edits.

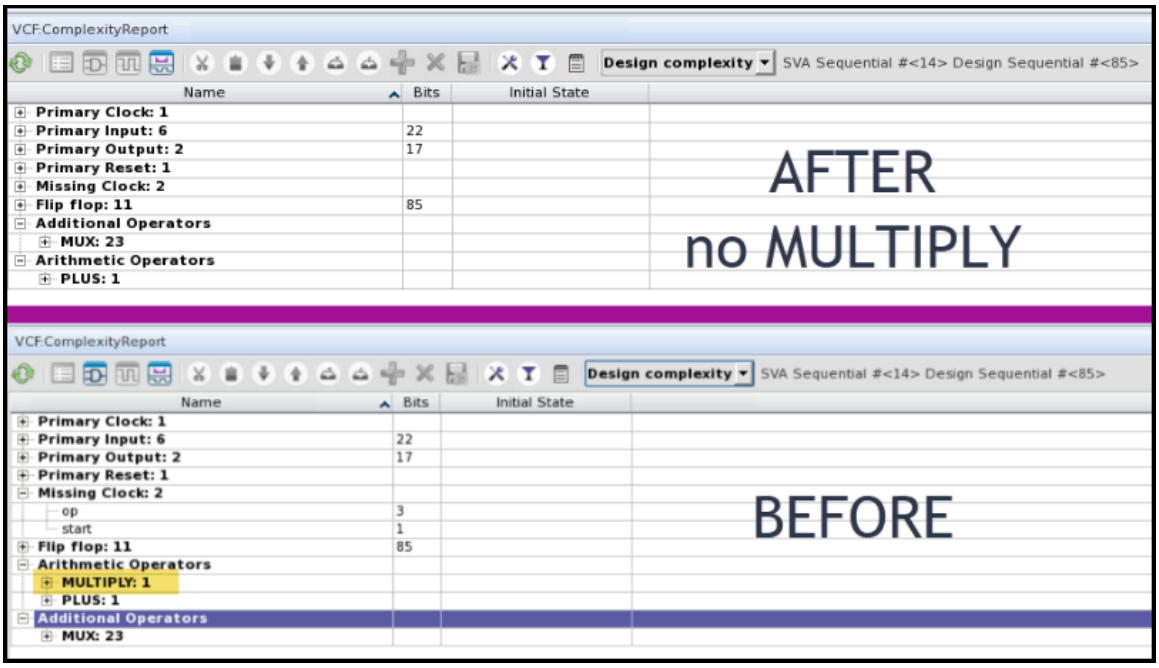

Figure 7. Formal Design Complexity Showing Removal of the (1) "MULTIPLY" after Commenting the HDL

The Formal Design Complexity report is a feature of the Formal tool Synopsys VC Formal, but does not give us an exact number of how many gates would be saved or what types. Provided with the source code are the Makefile targets

```
make tinyalu_synthesis
make tinyalu_synthesis_ex1
```

Figure 8 Makefile targets for synthesis comparison

that will synthesize our TinyAlu design into the target gate technology of lsi_10k for demonstration purposes. We can then compare the results of doing this change in Table 2.

Table 2. Synthesis Results

Design	Cells	Combinational Cells	Sequential Cells	Total Cell Area
Original RTL (synthesis)	520	439	78	1377
Removed Multiplier Lines RTL (synthesis_ex1)	127	115	10	244
Improvement over Original RTL	**4.1x**	3.8x	7.8x	5.6x

The removal of the unreachable lines resulted in the synthesis tool choosing to completely remove the module "`three_cycle`" from the synthesized netlist. The cell gains were large in this example because the number of cells required to implement a multiplier is high in relation to the rest of the design. Can you go farther than just Line coverage? In fact, you can!

We can now use FCA to analyze the synthesized netlist and compare the toggle coverage before and after reapplying our same "`only_single_cycle_ops`" assume statement. We get an interesting result whereby there is one toggle that was made unreachable by the inclusion of the assume.

The reason why we are interested in the difference is because we want to focus on FCA toggle differences caused solely by the addition of stimulus assume statements. Unreachability analysis is a highly understood process and extant unreachable components should be handled with existing methods.

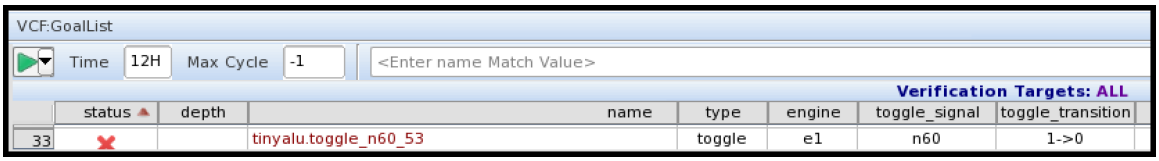

Figure 9. Toggle coverage made unreachable by applying "only_single_cycle_ops"

When we investigate what this condition is, we can see that the netlist wire "`n60`" is driven by the following logic from the `ND2` (NAND) in Figure 10 and Figure 11.

```
single_cycle and_add_xor ( .A(A), .B(B), .op(op), .clk(clk), .reset_n(
        reset_n), .start(start_single), .done(done_aax), .result({
        SYNOPSYS_UNCONNECTED_1, SYNOPSYS_UNCONNECTED_2, SYNOPSYS_UNCONNECTED_3,
        SYNOPSYS_UNCONNECTED_4, SYNOPSYS_UNCONNECTED_5, SYNOPSYS_UNCONNECTED_6,
        SYNOPSYS_UNCONNECTED_7, result_aax}) );

  NR2 U59 ( .A(op[2]), .B(op[1]), .Z(n63) );
  ND2 U60 ( .A(op[0]), .B(n63), .Z(n60) );
…
  AN2P U61 ( .A(done_aax), .B(n60), .Z(done) );
  AN2P U62 ( .A(result_aax[0]), .B(n60), .Z(result[0]) );
  AN2P U63 ( .A(result_aax[1]), .B(n60), .Z(result[1]) );
  AN2P U64 ( .A(result_aax[2]), .B(n60), .Z(result[2]) );
```

Figure 10. and_add_xor synthesized netlist excerpt

Then that signal "n60" is used to drive the `result_aax` (the output of the `and_add_xor`) onto its output ports (`result[0]`).

```
  NR2 U59 ( .A(op[2]), .B(op[1]), .Z(n63) );
  ND2 U60 ( .A(op[0]), .B(n63), .Z(n60) );
```

Figure 11. Logical Expansion of drivers of net n63

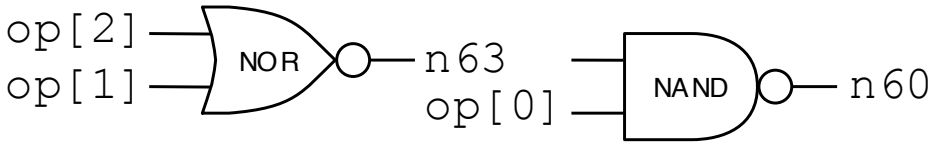

Figure 12. Netlist to Logical Schematic Representation

```
op[2] || op[1] || !op[0]
OP_MULT = 3'h1 = 3'b001
```

Figure 13. Sum of Products Form of Logical Schematic Representation Compared to Value of OP_MULT

Which is only zero for only one input condition, as simplified in Figure 12 and Figure 13, the OP_MULT condition `3'b001`! The Synthesis tool doesn't know that OP_MULT is a forbidden stimulus and cannot optimize this operand away completely even if the `MULT` module has been stripped from the design, it keeps this conditional logic in the design. Yet Formal, which has the ability to model the more complicated input assumes, can correctly identify this extra logic from the netlist and tells us how to fix it.

In Figure 14 is the edit done to the synthesized netlist postulated by FCA toggle. We change the driver of n60 from the output of a `ND2 (NAND)` to be driven by a constant. This step is done by hand in this example for educational purposes. The true implementation does not have the user hand editing a netlist. All of the information to make this edit existed in the Formal FCA Toggle analysis and can be automated. So the end user will never make this hand edit, but instead run some type of automation to implement this edit through an ECO (electronic change order) file generated by the Formal tool.

```
// BEFORE
// ND2 U60 ( .A(op[0]), .B(n63), .Z(n60) );

// AFTER assign n60 to constant
assign n60 = 1'b1;6
```

Figure 14. Edit to synthesized netlist based of FCA toggle

Making the above edit to the synthesized netlist and then rerunning synthesis on the edited synthesis netlist results in the following improved results in Table 3.

Table 3. Synthesis Results with Toggle Removal

Design	Cells	Combinational Cells	Sequential Cells	Total Cell Area
Original RTL	520	439	78	1377
Removed Multiplier Lines RTL	127	115	10	244
Improvement over Original RTL	4.09x	3.82	7.80	5.64
Above with Toggle Removal	103	91	10	196
Improvement with Toggle Removal	**5.05x**	4.82	7.80	7.03

How does one assign statement remove 24 cells from the design? It is because Synthesis engines are remarkable at reducing Boolean logic. Giving the Synthesis engine one hint can trim tremendous cones of logic. But, as we are describing in this paper, Synthesis engines do not have a robust input stimulus description language such as SystemVerilog assertions and the `assume` statement to guide them.

You may ask what was the logic in the original RTL that was involved with the now unreachable toggle? It was from the line below in Figure 15.

```
assign result = (op == OP_MULT) ? result_mult :  result_aax;
```

Figure 15. Stranded assign statement created by assume on stimulus from tinyalu/tinyalu.sv

Notice that Synthesis cannot optimize out the operand OP_MULT from possible input stimulus, so this line of RTL in Figure 15 must remain active in the synthesized design according to the Synthesis tool. But if we assume that this op can never be `OP_MULT`, then this result can be simplified using this method!

There is an obvious question why the author didn't run FCA toggle analysis on the RTL instead of the first synthesized netlist (Synthesis 1 from Figure 2)? The answer is that RTL is a high-level design language and the way it describes RTL is quite abstract. RTL optimization described by FCA toggle analysis is generally harder to recode into readable RTL. But FCA toggle analysis targets are trivial when described at the netlist level because we are looking at individual busses and FCA can truly be set loose to analyze the design at the bit-level optimizations instead of at high levels of abstraction. The end result from using the netlist is easier ECOs to implement, instead of creating a system to automatically write correct and human-readable SystemVerilog, and more cells that can be removed at the netlist bit-level instead of high level RTL.

Another fair question is why not do the FCA Line and all coverage analysis at the synthesized netlist? One immediate problem with doing FCA Line analysis on a synthesized netlist is there are no longer any lines to analyze in synthesis netlist. A synthesis netlist is a sea of interconnected gates and there is no longer the concept of lines for FCA to work on. There are other features of the proposed flow you would miss if you did all analysis at the synthesized netlist level. For example, while our stimulus assumes were only applied to the ports of our TinyAlu, it is also possible to do stimulus assumes inside of the DUT. Those assumes require consistent hierarchy and signal names to operate. In the synthesized netlist those signal names may be bit-blasted away and no longer suitable for stimulus assumes. Running at least the first FCA Line analysis with the RTL allows for those assumes to be properly applied.

Where could you go from here? The demonstrated optimization was from one `assume` statement. But many assumes can be layered and contrasted for deeper insight into the cost of design elements and architectures. Below are some examples in Figure 16. These examples demonstrate the major application of the paper, lowering the barrier to reduce logic by constraining the inputs in ways that might not be possible with constant propagation. In your Formal software it is often possible to enable and disable assume statements from the GUI or TCL interface. Using this interface, you could start with combinations of assume statements below or be able to mix in combinations programmatically.

```
always @(*) begin
     only_single_cycle_ops : assume (op != OP_MULT);
     only_even_input_A : assume (!(A[0]));
     only_even_input_B : assume (!(B[0]));
end

always @(*) begin
     only_mult_ops : assume (op == OP_MULT);
     only_odd_input_A : assume (A[0]);
     only_odd_input_B : assume (B[0]);

always @(*) begin
     only_add_ops : assume (op == OP_ADD);
     only_xor_ops : assume (op == OP_XOR);
     only_and_ops : assume (op == OP_AND);
end
```

Figure 16. Mix and Match assumes for more reduction from tinyalu/tinyalu_sva.sv

IV. Summary

FCA analysis is a known technique for searching for dead code in a design. The novel part of this approach is to not only do FCA analysis of a design. But, to consider that it is possible to do design experiments to see if certain architectural and usage constraints can result in a reduction of synthesized gates, area and/or power. It may become apparent that certain features of a design or even acceptance of certain stimulus patterns account for outsized numbers of gates to implement. After finding these outsized design features, this novel process innately contains a semi-automated process to precisely remove specific design features.

It was shown that this process could correctly remove an operand completely from an ALU design when used in conjunction with a synthesis tool resulting in a cell reduction of 5x. The process described can be automated. The approach made choices to get to the result in the form of a synthesized netlist instead of trying to rewrite the source RTL for reasons described in Results.

The conventional approach to removal of logic is to rely heavily on parameters or SystemVerilog defines embedded by the designers into the RTL. The process of parameterizing the design to add or remove features is error prone. This paper proposes that removing features could be handled by this process in a way that is just as safe and perhaps exceeds what a designer could implement themselves.

Imagine if a design came with a menu, not unlike a restaurant menu, that listed the feature and then the gates cost next to it. That could lead to incredible optimization of inflight and reuse derivative designs as each feature can easily - at any time in the design cycle - be removed and weighed, in gates, per their value to their application.

V. Future Work

While this paper looked deeply at the impact of Line and Toggle coverage, there are other code coverage optimizations that can be analyzed with FCA that may yield interesting optimizations beyond what is published here: perhaps conditional analysis could be easily used?

Since we are removing functionality from the design, we absolutely need to protect the design from receiving illegal stimulus. This would best be handled by retaining the assume statements used for restricting stimulus and pushing those out as assertions that upstream elements can never drive illegal stimulus. What we have created through this process is hollowed out design that does not work if it receives illegal stimulus. There is interesting work in hardening a design to reject or accept illegal stimulus after the optimizations in this paper have been applied.

VI. Acknowledgement

Thanks goes to Alex Nettering at Synopsys, for brainstorming with me on use cases and feedback. I thank my employer Synopsys for sponsoring my attendance at DVCon to share this paper and time to create it.

References

[1] R. Salemi, The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology, Boston Light Press, 2013.
[2] R. Salemi, “The UVM Primer Code Examples,” 2013. [Online]. Available: https://github.com/rdsalemi/uvmprimer. [Accessed 8 8 2019].
[3] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE Standard for SystemVerilog 1800-2012, New York, NY: IEEE, 2013.
[4] Synopsys, "VC Formal 2019.06," 2019. [Online]. Available: http://www.synopsys.com
[5] GitHub, "GitHub," 2019. [Online]. Available: http://www.github.com