Verification of Counters Enumerated in Digital Circuits

Kevin Zeng
Bradley Department of
Electrical and Computer Engineering
Virginia Tech
Blacksburg, VA
Email: kaiwen@vt.edu

***Abstract*—Productivity for digital circuit design is being outpaced by the rate at which silicon is growing. Complex designs take a significant amount of engineering hours to complete in both ASICs and FPGAs. Design reuse can potentially decrease cost and increase design productivity. The main motivation behind this paper is that assisting the designer in discovering archived designs in order to complete a reference circuit could enhance design productivity. An overview analyzing a method for characterizing and comparing digital circuits is provided in order to suggest candidate circuits that engineers can reuse. Searching for key features such as counters can be used to compare the similarity of circuits. Feasible counter structures are searched for by using a topological search. These candidate counters are then verified using SAT solvers to see if the candidate fits the model that describes a counter. Results obtained show that the counters were successfully found for small circuits with the potential to be further extended to larger and more complex circuits.**

I. Introduction

FPGAs have become a targeted platform for many high-performance computing applications. In spite of this, a significant limitation in using FPGAs is the design process. As hardware circuits become more complex, design time can increase significantly, outpacing productivity [1]. The design productivity gap [2] shows that design capabilities are unable to keep up with the doubling of silicon density every two years according to Moore's Law. One solution to increase productivity is to reuse existing digital circuits, or intellectual property (IP) cores. Nelson et al. [2] suggested that, depending on the fraction of the design that is being reused and the overhead of reusing the design, the use of existing hardware can lead to a significant increase in FPGA productivity.

Design reuse in the hardware community has not gained widespread acceptance. Reasons include obsolescence of different platforms, lack of standards, overhead of designing a module to be reusable, incompatible databases [2], incomplete and inconsistent documentations, performance [3], etc. Even if reuse is seen in practice, the user would have to allocate time to search through sources, relevant IP cores, and documentations, which may or may not be organized in which designs are easily found.

There are challenges in the domain of design entry in FPGA tools. Current design environments for FPGAs lack interactive features that could potentially assist and promote design reuse. Even though standards such as IP-XACT are available to help facilitate the reuse of IP cores across various sources, many designers are unwilling to conform to these standards because of the overhead and complexity associated with them. For example, the XML data format of IP-XACT is difficult to read and modify without the support of additional tools [4]. If a majority of the disadvantages are made transparent to the user during the design phase, reuse of existing hardware can become a more appealing solution for increasing overall productivity.

There has been an abundance of effort involved in increasing the reusability of IP cores for FPGAs. The primary goal of OpenFPGA's CoreLib is to create a standard for hardware libraries in order to promote interoperability so that existing cores can be seamlessly integrated into FPGA tools. Vendors provide many pre-designed hardware modules that are optimized for their specific products. For example, Altera provides a library of parameterized modules (LPM) whose purpose is to provide efficient designs of specific functions that are independent of the technology. Altera [5] also designed Megafunctions, which are IP blocks that are optimized specifically for Altera FPGAs. Xilinx [6] also provides a unified library of numerous primitives and macros for its specific FPGAs. There are also open source communities that provide IP cores for designers to use. OpenCores, the current leading community for open source hardware IP cores, provides several hundred

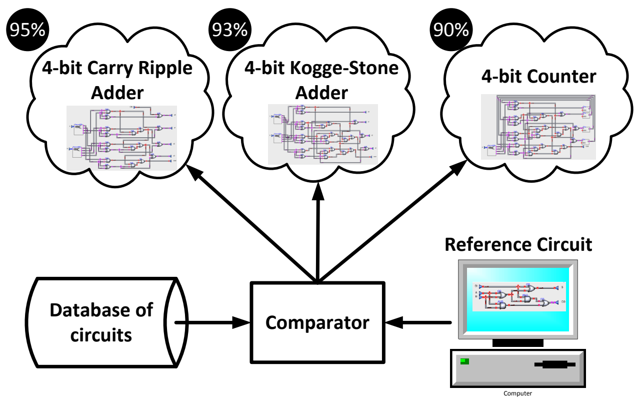

Fig. 1. Proposed usage model

existing IP cores. With more than a decade of investments, standardized design practices have not taken root. If there exists a proper community structure of easing contributions and automating access, this number can rapidly rise to tens of thousands of designs.

This paper extends and uses concepts from /citereverse to find and verify counter circuits in order to use them as features when comparing the similarity of digital circuits. The main focus here is the search and verification of potential counters in a circuit. The next section provides background behind finding potential counter circuits as well as verifying that the function of the candidate circuit is indeed a counter

II. Background

Counters are commonly used circuits in many types of circuits. Many complex designs will have counter modules located within the design and can cover a great deal of area in the circuit. Since many circuits uses counters, using them as a feature for comparing circuits is quite plausible. In order to determine if a circuit contains a counter or not, the latches and flip flops are observed because the main components of a counter are the sequential elements. If the data and behavior of the latches resembles a counter circuit then the set of latches observed are part of a counter. However, circuits may contain a great deal of latches and therefore checking to see if every latch is part of a counter is computationally infeasible. This section describes an existing method [7] that finds potential candidate counters and then verifies them using Boolean satisfiability.

A. Previous Work

Previous work tried to match and compare circuit structurally [8], [9], [10]. Structural representations are simple; however have certain limitations. For example, specific structural patterns are needed for specific functions. A two-input XOR gate can have a significant amount of different structural patterns that needs to be searched through in order to determine if the sub-circuit is an XOR gate. Furthermore. structurally different circuits may perform the same function and should be labeled similar instead of different. However, figuring out the function of a circuit as a whole is not feasible due to the amount of computations that need to be calculated such as input variable ordering.

One way the function of the circuit can be approximated is to try and determine a suitable high-level model of the circuit. Hansen et. al. [11] looked into reverse engineering as a way to try and determine high-level structures in a specific circuit by looking at replicated structures. Subramanyan [?] extended the idea of high-level module identification to identifying combinational and sequential elements in a circuit, specifically datapath elements. They use a combination of structural and functional matchings in order to determine the overall description of the circuit.

B. Finding Candidate Counters

In order to not consider every subset of latches to see if they are part of a counter, candidate counters are determined based on the topology of the latches. In other words, by observing how the latches are connected to one another structurally, a potential counter can be determined. Counters have a basic structure that is the same for any implementation as long as the flip-flop type is the same. For example Figure 2 shows the overall topology of a counter that uses D flip-flops. The higher order bits depend on the state of the lower order bits. That is why for the lower order bits, there exists an edge to every single bit that is higher up the chain.

Essentially, the vertices of the graph represents the latches and the edges represents a path from the data output of a latch to the data input of another latch regardless of what the combinational logic between the output and input is. This is because the combinational logic that determines the next state of the latches can be implemented in many different ways. Functionally equivalent circuits can be structurally different in many ways. Therefore defining the combinational logic structurally is infeasible at this point. However, knowing that there is a combinational path from the output of a latch to each of the latches that is higher in order than the current is enough to say that this is a potential counter.

Given a reference circuit from what the user is designing, a latch connection graph (LCG) is built. The LCG of the reference circuit tells us how the different latches are connected regardless of logic. Once the LCG graph of the circuit has been built, it is searched to try and find a subgraph that matches the model of a LCG that represents a counter similar to FIgure 2. Any subgraph that matches the model is considered a candidate counter. The datapath of the latches resembles that of a counter, however, it is still unsure whether the combinational logic between the latches models that of a counter. Therefore, the next step is to verify if the candidate counter is indeed a counter.

C. Verifying Candidate Counters

After obtaining a set of candidate counters, they need to be verified in order to see if they are counters or not. Therefore functional analysis is performed by checking if the candidate counter satisfies the function typical of a counter. Functional analysis is performed because the combinational logic between the latches can be represented in any fashion. With structural analysis, every possible structure of a counter has to be enumerated in order to confidently say that there is no

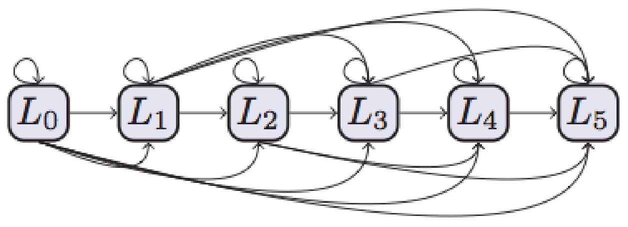

Fig. 2. Proposed usage model

other counter circuit that exists. Functional analysis captures this aspect more efficiently by specifying the function that the circuit is suppose to perform. With that, regardless of the structural composition of the circuit, the circuit can be found. Therefore, the behavior and function of a counter circuit needs to be modeled. The model can be seen below.

$$next_i = (curr_1 \wedge curr_2 \wedge ... \wedge curr_{i-1}) \wedge \neg curr_i \vee (\neg curr_1 \vee \neg curr_2 \vee ... \vee \neg curr_{i-1}) \vee curr_i \quad (1)$$

In the equation, $next_i$ represents the next state of the counter and $curr_i$ represents the current state of the counter. The first line of the equation states that the next state of bit i transitions from low to high when the previous state of the lower bits of the counter are high. This models when a bit in the counter goes high. Furthermore, the next state of bit i in the counter remains high if the previous state of bit i was high and if the bits less than i contains at least one latch that is low.Bit i will become low once all the lower order bits are all high which causes an overflow. It is assumed that the latch is always enabled, there is no set option, and the flip-flops are initially reset. With this model, the candidate circuits can be verified to see if the behavior of the circuit matches the model described above for all the bits of a candidate counter.

III. Implementation

This section explains the details of overall system used to find and validate candidate counters. The reference circuit is designed using a front-end tool. From the reference circuit, the netlist is extracted and imported into the back-end where the LCG is extracted from the circuit. The LCG is traversed to find potential counters by looking for sub-circuits a topology similar to a counter. A SAT solver is then used in order to verify if the candidate counter satisfies the counter model described in Equation 1 in the previous section. Figure 3 shows the flow diagram of the entire working system.

A. LCG Representation

In order to create a LCG representation of the entire circuit, each component in the netlist is traversed. Latches in the netlist are added as vertices to a graph data structure. The Graph Boost Library (GPL) was used to construct the LCG from the netlist. The edges of the graph are determined with a depth first search starting at each of the latch in the netlist. There exists an edge between two vertex in the LCG, V_i and V_j, if there is a combinational path from the output of V_i that goes into the input of V_j. The entire graph is traversed starting from each latch. Once all the latches have been traversed, the relationship between the latches has been determined and the datapath of each latch can be seen.

B. Finding Candidate Counters

With the LCG constructed, candidate counters can be determined. In other words we need to determine if the LCG model shown in Figure 2 is seen in the LCG of the reference circuit. Subgraph isomorphism is considered a NP-Hard problem and therefore no solution exists in polynomial time. Furthermore as the circuits become large, finding subgraph isomorphism is exponential in terms of how many vertices and edges there are in the graph. LCGs are tightly connected with many possible subgraphs. For example, given a 4-bit LCG counter model and a circuit with a potential 32-bit counter, there are 32 choose 4 different subgraphs which means that there are 35960 different subgraphs. Therefore it is inefficient to use a general purpose subgraph isomorphism algorithm to find a counter LCG in a reference LCG.

Our approach to find candidate counters is to traverse each vertex in the LCG. A list of possible latches in the counter is specified in a list called *counter*. Initially, a vertex $latch_i$ is placed in *potentialC*. Vertices that are adjacent to $latch_i$ are appended to *counter* if every vertex in *counter* has an edge to the adjacent vertex *adj*. This is because lower order bits all have edges leading into the higher level counter bits. This is essentially the same as the counter model in Figure 2. If an adjacent vertex *adj* does not have and edge from every vertex in *counter* to *adj*, it can still be part of the counter. If the first i latches has an edge that goes to *adj*, then *adj* should have and edge to the remaining *i+1* to *end* vertices. This is because traversing through adjacent vertices may not be in order and the order of latches in the counter may be traversed differently. Lower order bits have an edge going into all the higher order bits. Since not all of the elements in *counter* goes into *adj* this means that not all the elements are lower order bits and some of the bits may be higher level. After each vertex that is not in a list of *counter* has been traversed, a list of candidate counter circuits of various sizes has been obtained. The algorithm for

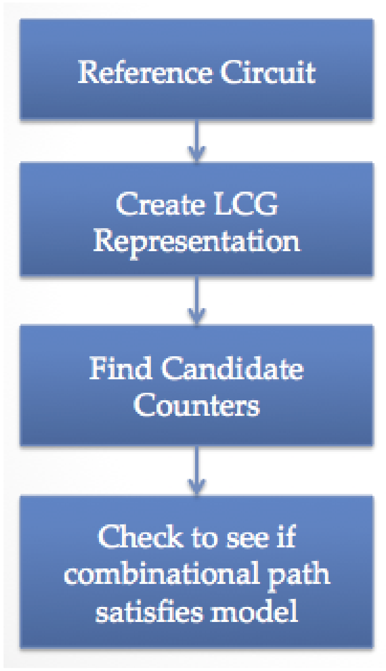
Reference Circuit
Create LCG Representation
Find Candidate Counters
Check to see if combinational path satisfies model

Fig. 3. Proposed usage model

finding candidate counters is shown below.

```
Algorithm 1: findCandidateCounters
Input  : LCG of reference circuit
Output: candidateC //Candidate Circuits
begin
    $candidateC \longleftarrow \emptyset$;
    for $latch \in V(LCG)$ do
        $AdjacentLatches \longleftarrow latch.adjacent()$;
        $counter \longleftarrow \{latch\}$;
        for $adj \in AdjacentLatches$ do
            if $counter \subseteq adj.inEdges()$ then
                $counter \longleftarrow \{counter \cup adj\}$;
            else if $c_{0-i} \in counter \subseteq adj.inEdges() \wedge$
            $c_{(i+1)-end} \in counter \subseteq adj.outEdges()$
            then
                $counter \longleftarrow \{c_{0-i} \cup adj \cup c_{(i+1)-end}\}$;
            else
                $adj$ is not part of counter
            end
        end
        if $counter.size() \geq 2$ then
            $candidateC \longleftarrow \{candidateC \cup counter\}$
        end
    end
end
```

C. Verification of Candidate Counters

With each counter, the functionality of the combinational circuit between the latches needs to be verified such that the function it performs matches the model that was specified in the previous section for a typical counter. Therefore, we need to convert the combinational path between the vertices of the LCG in the list of candidate counters into conjunctive normal form (CNF) format so that it can be passed into a SAT solver. Many modern SAT solvers require that the input to their tools be in CNF format.

1) And Inverter Graph: To perform the conversion, the circuit is converted to an and inverter graph (AIG) representation with latches. And-Inverter Graph (AIG) is a representation of a circuit or a function with only AND gates and inverters. The nodes of the graph are the AND gates and primary inputs and outputs. Primary inputs are nodes with no fanins and primary outputs are nodes with no fan-out. A marker is placed on an edge for inverters. AIGs provides a very compact, scalable, and efficient structure for analyzing the functionality of the circuit and allows the circuit to be easily expressed in CNF format. In order to transform the combinational logic of a candidate counter into the CNF format, the Tseitin transformation is used.

2) Look-Up Tables: Look up tables (LUTs) components from synthesized digital circuits need to be handled differently since a given LUT handles a different function entirely. A LUT component has a set of inputs of *n* bits, where n ranges from 1 to 6, that represents the location of the data in a truth table. The truth table of the LUT is given in the netlist. However, in order to pass the function of the LUT into the SAT solver, a CNF formula needs to be extracted from the truth table. ThIn order to do so, the truth table is converted into a sum of product circuit by looking at which bits are ones. From the sum of products, the AIG graph representation can be extracted and converted into CNF format.

3) Tseitin Transformation: Tseitin transformation [12] is able to take any arbitrary combinational circuit representation and derive a CNF formula for it. Tseitin transformation is chosen due to the fact that it provides a decent and full representation of the circuit in terms of size. If the Boolean equation was taken and has to be transformed into CNF format, the equation can end up being exponential in size depending on the circuit. Tseitin transformation provides an equation that is linear to the size of the circuit. Since the circuit is in AIG format, the Tseitin transformation is applied to each and gate and inverter gate. The Tseitin transformation for the and gate and the inverter gate is shown below.

$$\begin{gathered}(C = A \wedge B) = (\neg A \vee \neg B \vee C) \wedge (A \vee \neg C) \wedge (B \vee \neg C) \\ (C = \neg A) = (\neg A \vee \neg C) \wedge (A \vee C)\end{gathered} \tag{2}$$

With this, the entire AIG circuit can be transformed into a conjunction of the Tseitin transformation for each gate in the combinational path of a candidate counter. Then following the model of the counter, additional clauses are added to the CNF formula for a candidate counter. For example, for a 4-bit counter, after extracting the combinational CNF formula for the combinational path in the LCG, we then need to satisfy the model of the counter. Meaning we would need to verify the satisfiability separately for each bit. Therefore, to verify the MSB of the counter we would need to add $cur_1 \wedge cur_2 \wedge cur_3 \wedge \neg cur_4$ and $\neg next_1 \wedge \neg next_2 \wedge \neg next_3 \wedge next_4$ to the clauses because the most significant changes to one when all the previous state of the latches are all high. These clauses are appended and checked one by one to the combinational CNF formula. After each bit has been checked, the fact that all the bits remain one when at least one bit is zero needs to be checked. In other words, the CNF formula for a 4-bit counter is $(cur_1 \vee cur_2 \vee cur_3) \wedge cur_4 \wedge next_4$. This basically means that if the MSB of the current state is high, the next state of the MSB will remain high as long as one of the lower order bits is zero. These clauses are added onto the combinational CNF formula and checked to see if it is satisfiable. If all the CNF formulas constructed above are satisfiable that means that the candidate counter indeed is a counter and satisfies the model described in the previous section. With all the clauses compiled, it can be sent to the SAT solver to see if it is satisfiable. If there is one set of clauses that is not satisfiable the candidate counter does not correctly model a counter and therefore is not a counter.

IV. Results and Analysis

This section discusses the results obtained from the implemented counter enumeration and verification algorithm.

TABLE I
CIRCUIT CHARACTERISTICS

Circuit	Num Counters	Num Counters Found	Num FF
counter4	1	1	4
counter8	1	1	8
counter32	1	1	32
counter64	1	1	64
goodcounter4x2	2	2	8
badcounter4x2	1	1	8
goodcounter	1	1	4
nocounter	0	0	0

The counter enumeration was implemented in C++ using the algorithms and methods described in the previous section. The reference circuits are designed in Verilog and synthesized using Xilinx ISE 14.0 in order to extract a netlist. From the netlist that is extracted from the reference circuit, the circuit is passed into the tool where the LCG is constructed using GPL. With the LCG, candidate circuits are searched for with Algorithm 1. The candidate circuits are then verified by constructing appropriate clauses for the combinational path as well as making sure the behavior models the equations that are described in Equation eq:model correctly. MiniSat 2.0 was used for solving the satifiability conditions of the CNF clauses derived.

The experiments were conducted in Ubuntu 12.04 on a Dell Vostro with a 2.8 GHz Intel Core2 Duo processor and 2.9 GB of RAM. A small set of custom circuits were used to show the feasibility of this project. Circuits contained a mix of good and bad counters and smaller designs with and without counters in them. The scope of the project was limited to basic detection of counters. Larger circuits contain many corner cases that the current tool may not detect due to circuit transformation rather than the verification process. A total of different counter circuits were designed and used to test the tool. Counters are of different sizes ranging from 4 bits to 64 bits. The limit was set to 64 bits since a 64-bit counter can count to greater than 19 magnitudes which should be take more than enough time before it over flows with a 1 GHz clock.

A. Accuracy

The first four counters in Table I are solely counter circuits of that specific size. From the results we can see that each counter has 1 counter and that specific counter was found for each of the different counter sizes. Since we know counters can be detected, we tried mixing the different types of counters into designs to see how well they matched. The circuit goodcounter4x2 is a circuit that consists of two 4-bit counters. From the results, both the 4 bit counters were found. With the badcounter4x2 circuit, the counter has one counter that is valid, and the other counter has its most significant bit inverted therefore making it not a valid counter. Therefore only 1 counter was found. The goodcounter circuit is a different implementation and functionally identical to that of counter4. This is to show that regardless of how the structure of the circuit is designed, the tool will be able to find counter circuits based on their function. The logic of counter4 uses primarily LUTs as it's combinational logic where as goodcounter uses primitive gates. From the results we can see that with both implementations, the tool was able to find the counters. The last circuit does not have a counter in it and the results were as expected.

B. Performance

Performance of the tool performed fairly well and the results were as expected. The more latches there are in the circuit, the larger the search space. The increase in time is also affected by the circuit size during the transformation of the circuit. AIGs on the other hand gives a linear time transformation based on circuit size. Furthermore more, the counter verification also depends on whether or not a satisfiable assignment is found. If there is one, then Minisat has to make sure it is satisfiable for all clauses. If there isn't one, then Minisat will automatically fail and determine it is unsatisfiable.

C. Limitations

Larger circuits from the IWLS benchmark, specifically from ITC'99, were tested but eventually timed out or an error occurred. This is primarily because the transformation of the circuit to a suitable CNF representation is still somewhat faulty. The main problem is at the candidate circuit generation where the candidate circuits are determined. As stated before, using the vf2_subgraph_isomorphism function from GPL to find a suitable subgraph that fits the counter topology model is difficult and cannot be found in polynomial time. The state space completely explodes especially for large counters or in general, a large LCG. Therefore, a manual traversal and search was performed by traversing the entire LCG. The problem is probably that the traversal done does not take care of all the cases needed to determine if a candidate circuit is valid or not. Otherwise, the CNF clause generation should be correct as well as the LCG generation. Therefore, more work is needed in order to have the tool work for larger and more complex circuits. Future works for the tool is described in the next section.

V. CONCLUSION

Circuit designers can reduce design time significantly by not having to redesign circuits that already exist. By reusing

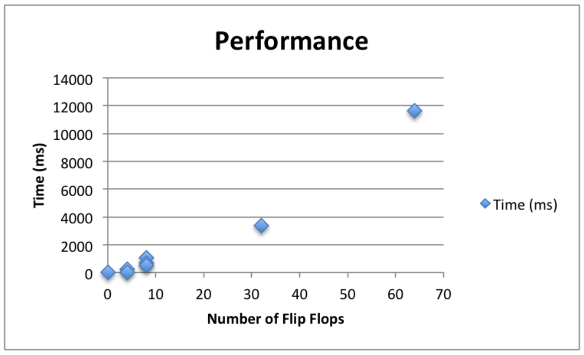

Fig. 4. Performance of counter enumeration

existing hardware, the designers can focus more on the application rather than the verification of existing hardware. The main focus of this paper is to introduce a feature extraction method of finding counters in circuits. The reason counters are searched for is because they are a common component in many digital circuits and can cover a large area of a circuit. By looking for these counters, and using them in the search process, it can potentially increase the accuracy of the matches.

When finding counters, the number of sequential elements can be very large. To reduce the search space, candidate circuits are determined by looking for a set of latches that match the topology of a counter. Candidate circuits are then validated using verification techniques such as SAT solvers. A number of transformations are needed in order to have the circuit described in a format where SAT solvers can take it in easily.

A. *Future Works*

Future works include extending this to larger circuits. Right now, the tool can handle simple circuits due to the number of corner cases that may occur. The candidate counter generation has some limitations that can be improved upon. Furthermore, this technique can be applied to the circuit similarity testing aspect to see if searching for counters does indeed increase the accuracy when comparing two circuits. Additional sequential components can be searched for as well such as shift registers, certain RAM modules, etc in order to encompass more of a circuit. The methods for finding such components will be similar in the fact that latches are looked for whose topology matches the certain component and the verification techniques are used to validate that the combinational paths between the latches fit the model that describes the component.

Even if this method does not prove useful in terms of circuit similarity matching, high level functionality of the circuit can be extracted based on counters and can be used to at least classify and/or compile a database of unorganized circuits into libraries that can be more easily searched through.

References

[1] R. Chandra, "Ip-reuse and platform base designs," 2002.

[2] B. Nelson, M. Wirthlin, B. Hutchings, P. Athanas, and S. Bohner, "Design productivity for configurable computing," in *ERSA08: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2008, pp. 57–66.

[3] E. Girczyc and S. Carlson, "Increasing design quality and engineering productivity through design reuse," in *Proceedings of the 30th international Design Automation Conference*, ser. DAC '93. New York, NY, USA: ACM, 1993, pp. 48–53. [Online]. Available: http://doi.acm.org/10.1145/157485.164565

[4] T. Schattkowsky, T. Xie, and W. Mueller, "A uml frontend for ip-xact-based ip management," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 238–243.

[5] Altera, "Introduction to megafunctions," 2011.

[6] Xilinx, "Virtex 4 libraries guide for hdl designs," 2009.

[7] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 1277–1280.

[8] K. Zeng and P. Athanas, "Enhancing productivity with back-end similarity matching of digital circuits for ip reuse," in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, Dec 2013, pp. 1–6.

[9] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, "Subgemini: identifying subcircuits using a fast subgraph isomorphism algorithm," in *Proceedings of the 30th International Design Automation Conference*. ACM, 1993, pp. 31–37.

[10] J. Whitham, "A graph matching search algorithm for an electronic circuit repository," *Univ. of York*, 2004.

[11] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the iscas-85 benchmarks: a case study in reverse engineering," *Design Test of Computers, IEEE*, vol. 16, no. 3, pp. 72–80, 1999.

[12] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning*. Springer, 1983, pp. 466–483.