Reverse Engineering of RTL Controllers from Look-Up Table Netlists

Sundarakumar Muthukumaran, Aparajithan Nathamuni Venkatesan,
Kishore Pula, Ram Venkat Narayanan, Ranga Vemuri and John Emmert
Digital Design Environments Lab, ECE Department
University of Cincinnati, Cincinnati, Ohio, USA
muthuksr@mail.uc.edu, nathaman@mail.uc.edu, pulake@mail.uc.edu, narayart@mail.uc.edu ,
ranga.vemuri@uc.edu, john.emmert@uc.edu

2023 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) | 979-8-3503-2769-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISVLSI59464.2023.10238540

***Abstract*—Verification of FPGA-based designs and comprehension of legacy designs can be aided by the process of reverse engineering the flattened Look-up Table (LUT) level netlists to high-level RTL representations. We propose a tool flow to extract Finite State Controllers by identifying control registers and progressively improving the accuracy of register classification. A control unit consists of one or more Finite State Machines (FSMs) which manage the execution of datapath units. The proposed tool flow has two phases. Phase 1 extracts the potential state/control registers. Phase 2 identifies the exact list of state/control registers and groups FSMs. The main goal of the proposed work is to improve the accuracy of control register identification. Three types of controllers used for experimental evaluation are standalone FSM designs with no datapath units, datapaths with a single FSM, and datapaths with multiple FSMs. Accuracy is observed to be 73% to 100% in controllers with multiple FSMs, 100% in controllers with a single FSM and standalone FSM controller designs. The average accuracy of control register detection over all the real-world designs considered is 98%.**

***Index Terms*—FPGA, reverse engineering, controllers, state registers, register classification**

I. Introduction

FPGA reverse engineering serves many purposes such as understanding legacy designs whose high-level RTL descriptions are long lost. Other applications include the detection of counterfeit devices, hardware trojans, and investigations for patent infringement. The proposed work focuses on controller reverse engineering. The hardware reverse engineering process of FPGA-based designs has two stages which are known as Netlist Extraction and Specification Discovery [1]. FPGAs consist of Configurable Logic Blocks (CLBs) connected through a programmable interconnect matrix. These CLBs are programmed to include logic primitives like Look-Up-Tables (LUTs), storage elements, etc., based on the type of FPGA. Netlist extraction involves extracting the cells and nets from the bitstream data read from the FPGA board to create a LUT-level netlist. Project X-ray is one such tool that reads a bitstream used to program a Xilinx FPGA device and extract the LUT-level netlist [2], [3].

The extracted LUT-level netlist is then analyzed to understand the high-level description of the design and this process is called specification discovery. Since the netlist extracted from the bitstream is fully flattened, the module boundaries are hidden. In FPGAs, a considerable amount of combinational logic is absorbed into each LUT and high-level hierarchical and structural information is lost. Graph similarity and matching sub-circuits help us identify word-level structures and understand datapath units [1], [4].

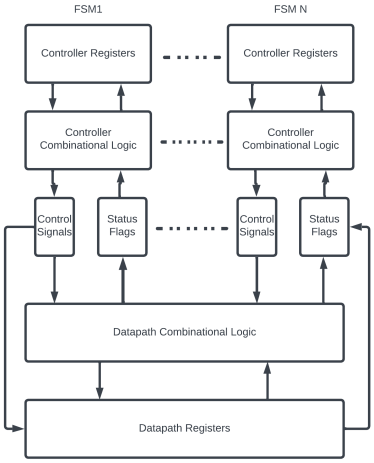

Fig. 1: Overview of interactions between datapath and controller units.

Figure 1 shows the structure present in a design with controller and datapath units. A design usually has datapath logic that constitutes adders, subtractors, comparators, or ALUs. Datapath can contain data registers and status registers. The status registers may be connected to the controller unit and influence the controller's operation. The status and state registers have similar functionality and so may have similar structural characteristics related to them. The controller contains the control/state registers and the associated combinational logic of the FSMs. Outputs of the FSMs feed into the datapath unit.

We present a tool flow to perform register classification and FSM extraction. The tool is given a fully flattened LUT-level netlist as input and the tool performs a series of checks based on the structural characteristics of a given register's neighborhood to accurately identify state registers. These registers are further inspected to arrive at a final set of state registers in the second phase. We consider encoding techniques that include one-hot, gray, sequential, binary, and the auto mode available in Xilinx Vivado [5].

This paper is organized as follows. Section II discusses the state-of-the-art work on ASIC and FPGA gate-level netlist

reverse engineering techniques. Section III describes the proposed algorithm flow for designs with only a standalone FSM and for designs containing a datapath unit with a controller containing one or more FSMs. Section IV consists of the experimental setup and a comparison of the results obtained in Phase 1 and Phase 2. Section V concludes the work.

II. Related Work

Shi et al. [6] proposed a method to extract state registers from a flattened gate-level netlist. The method follows from the work presented by McElvain and Kenneth [7]. Their methods were unable to identify counters and they extracted an FSM logic that is a composition of smaller FSMs. We adapt the strategy proposed by Shi et al. [6] to FPGA-based netlists in the first phase of our tool flow.

The RELIC tool [8] analyzes the netlist topologically to find similar fan-in cones of cells and to identify state registers. The netlist is first pre-processed and expressed using AND-OR-INVERT logic. A recursive algorithm is then used to check for the similarity between pairs of nodes, with the graph topology serving as a similarity criterion. The criterion is checked at each stage based on the transitive fan-in of that stage. If the value of the similarity metric exceeds a predetermined threshold, two graph nodes are marked as similar. The tool is tested on a variety of small benchmark circuits and demonstrates 80% to 100% accuracy. This work is improved upon in work done by Brunner et al. [9] to speed up the RELIC algorithm by 100 times and analyze larger circuits with a few thousand registers. RELIC applies to ASIC netlists and does not classify registers.

Chowdhury et al. [10] proposed a machine-learning-based technique to identify state registers in a netlist. Firstly, the authors converted the netlist into a graph and extracted features related to state registers such as centrality measures, and trained a Graph Neural Network. The weights obtained through training were fed to a binary classifier to classify state registers and data registers. The authors validated their method with several real-world benchmarks and stated that it outperformed all the traditional machine-learning techniques in terms of accuracy and efficiency. The authors claim that the trained model can be used in classifying registers in extremely large and complex designs. The average accuracy achieved by ReIGNN is reported as 96.5%. REIGNN applies to ASIC netlists and not FPGA-based netlists.

Fyrbiak et al. [11] propose an approach to enhance state register identification by introducing an influence/dependence metric. The metric estimates how many gates are controlled by a given element or logic. FSMs usually control a large part of a netlist and a high metric would indicate an FSM-related logic or element. The user still needs to decide whether a candidate logic is part of an FSM or not. Fyrbiak et al. extend their work to FSM obfuscation.

Wallat et al. [12] developed the first open-source toolchain to aid in reverse engineering designs from flattened gate-level or LUT-level netlists to high-level RTL representations of the netlists. The tool helps in performing fundamental processing such as parsing a netlist in Verilog, and storing cell and net information in an intuitive form. It also helps in performing basic operations on a netlist and in traversing a netlist. Narayanan et al. [13] developed a toolchain for carrying out novel carry chain analysis on FPGA netlists. In order to identify operators such as adders, subtractors, comparators, and ALUs, the authors compared the subcircuits with golden library components. The authors also constructed a flip-flop dependency graph of the netlist and analyzed the data flow for sequential components. The toolchain delivers 34% to 100% gate coverage for real-world designs. [14] use proximity information on cells in the implemented design to group elements that include LUTs, CARRY, RAM blocks, and registers. The algorithm groups elements with a Normalized Mutual Information metric of 0.73 for real-world designs.

III. State register identification process

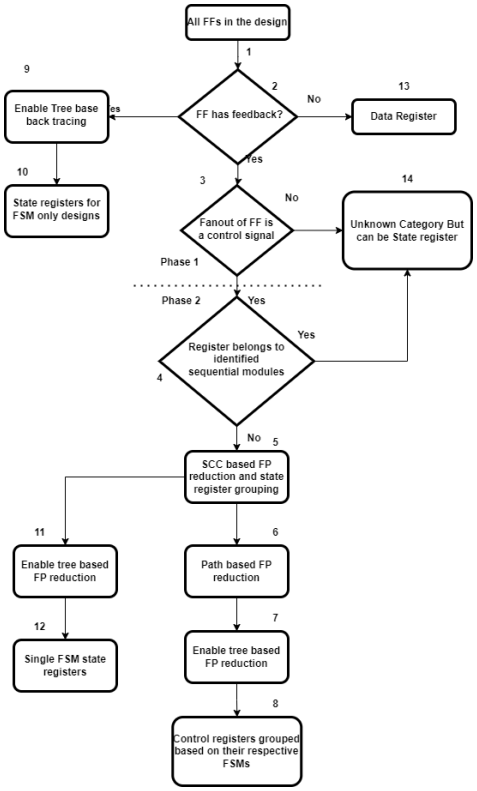

Fig. 2: Tool flow for control register identification.

The proposed controller identification tool works in two phases as shown in Fig 2. Phase 1 is an adaptation of a few existing ASIC netlist-based control register identification techniques to FPGA designs with modifications since the FPGA primitives are different from ASIC primitives. The results of Phase 1 are then processed with a sequence of algorithms in a particular order to achieve the best isolation of the control registers from data registers. The registers are classified into three sets for the purpose of a better understanding of the proposed tool flow. Set A which contains the registers that are highly likely to be control registers. Set B which contains the set of registers that cannot be concretely identified as control or data registers is left to the reverse engineer for further manual

analysis. The registers eliminated are put into a set of data registers.

The steps are separated into Phase 1 and 2. Steps 1, 2, 3, 4, 5, 6, 7, 8, 13, and 14 describe the flow to identify control registers in Single and Multiple-FSM controller designs. Steps 1, 2, 9, and 10 describe control register identification in Standalone FSM designs without a datapath.

A. Phase 1

The first step of the algorithm is to identify the potential control registers irrespective of the number of FSMs in a controller design. Based on work done by Shi et al [6] the state registers in an FSM have a unique feedback structure for both Mealy and Moore state machines. There is both a forward and backward flow of data for the control registers. However, there are also data-path registers with the same property. This technique also classifies other sequential module registers like counters with similar structures as state machines. Initially, Set A would contain all registers present in the current design extracted from the HAL tool [12]. Then by a forward Breadth First Search (BFS) algorithm, the path of the output fan-out net is explored until the same register's input net (at D pin) appears. If it encounters an output or an input buffer, that path is discarded. The resultant registers that satisfy this property will remain in Set A. The registers without this feedback property are eliminated and will fall into the set of data registers. Algorithm 1, lines 5 to 12 perform this step.

The second step is to incorporate an additional constraint to filter only the registers that control the data-path elements in their fanout and eliminate the other registers which are not control signals. The check is to see if the fan-out net is connected to a Control Enable (CE) pin of a flop or a select line of the multiplexor. FPGAs have registers in their designs with CE pins that enable/disable them. With forward BFS from the register's Q pin and by checking if the destination pin of the fanout net is tied to a CE pin, we can decide if a flip-flop's output net is a control signal. As FPGAs do not have multiplexors as a part of the primitives and the logic is usually synthesized into LUTs, we had to make some changes to adapt existing methods to FPGA designs. We consider the Boolean equation associated with a LUT connected to any potential state register. We check if the fanout net of a register is present as a variable in the Boolean equation. We check the number of occurrences of the fanout net's literal in the Boolean equation. A large number of occurrences may indicate that the fanout net is a select line and belongs to a multiplexor. We adapt this approach from [11]. If the fanout is a select line (shown in Fig 3, literals I1 and $\overline{I1}$ are counted), the register is a candidate state register else not. State registers can sometimes be directly connected to output buffers. Hence, Registers in Set A satisfying this property remain in Set A. Registers in Set A without this property are put in Set B. Algorithm 1, lines 13 to 19 perform this step.

These existing techniques allowed some false positives indicating the presence of data registers with similar structural properties as state registers. The algorithms discussed below are implemented to reduce the false positives progressively.

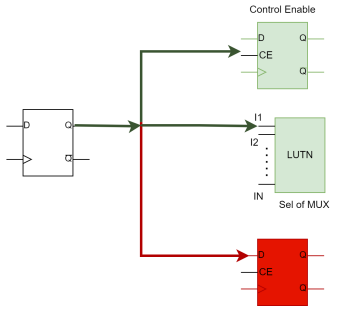

Fig. 3: Control signal identification

The methods in Phase 2 eliminate false positives (FP) by a considerable number.

B. Phase 2

1) Eliminating data registers by identifying other sequential modules: In step 4, we started by identifying datapath modules such as counters, shifters and multi-bit registers from the toolchain developed by Narayanan et al. [13]. The registers in these modules that are identified as datapath are eliminated among the set of candidate state registers obtained at the end of Phase 1 (Set A). This reduces the false positives which are counters as they mostly have a structure of FSMs. But the registers eliminated in this step are not classified as data registers but classified into the potential state register bank (Set B) that we already have. The counter and shifter registers eliminated in this step have unique control signals. Registers in Set A that are classified as sequential datapath modules in this step are put in Set B. The other registers are left untouched. There may exist counters which share the enable signal with control registers and these counters will be eliminated in the following steps. Algorithm 1, lines 20 to 25 perform this step.

2) SCC-based FP reduction technique and FSM separation: In step 5, the next type of false positive targeted is the status registers which did not get eliminated in the previous steps. As we already know the unique feature of the FSMâs control registers which is to have feedback to themselves which makes an FSM a strongly connected component. Running a strongly connected component algorithm on the resulting registers separates the data/status registers from control registers as shown in Fig 4. Using the netlist utilities of HAL tool, a strongly connected components algorithm is used to find whether a given netlist contains multiple FSMs. The algorithm gives components that contain elements that have a bidirectional path to each other. The algorithm used for identifying strongly connected components is observed to produce results similar to the groupings made by Tarjan's SCC algorithm [15]. The registers that are single register SCC are eliminated and considered data registers as a single register cannot form a state machine. If the resulting list of components contains a single SCC it is a single FSM controller design. If there are multiple SCCs found, the netlist is a controller with multiple FSMs. Registers found in the SCC components returned belong to Set A. Registers which do not have any other registers in their SCC component are put in the set of data registers. Algorithm 1, lines 26 to 35 perform this step.

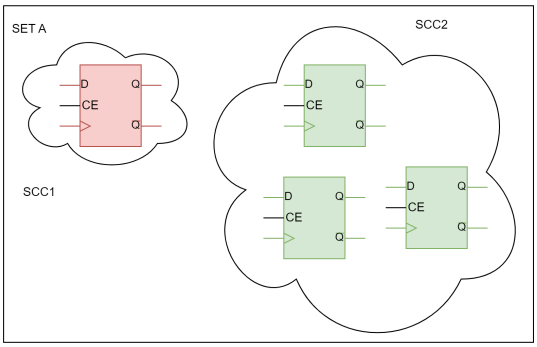

Fig. 4: SCC-based FP reduction

3) Path-based FP reduction: The next type of False positive includes registers belonging to counters which sometimes have the same structure and control/enable signal as FSMs and could not be identified in earlier steps. We use a path-based algorithm where we check for a direct bidirectional path between a register and one other register in the component, this can be described as a more constrained feedback property. The counters do not have a direct bidirectional path in their structure and it is not a constraint in the SCC algorithm before (as SCC somehow tries to find a path between a false positive counter register and a control register). In counters, Lower-order bits connect to all higher-order bits, and higher-order bits are only connected to any higher-order bits if they exist as shown in Fig. 5 and they do not have at-least one connection bi-bidirectionally to each other whereas FSM shown in the figure does. This path-based property is checked and the counter registers which escaped the detection in step 3 are classified as data registers. Registers in Set A satisfying this property remain in Set A. Algorithm 1, lines 36 to 43 perform this step.

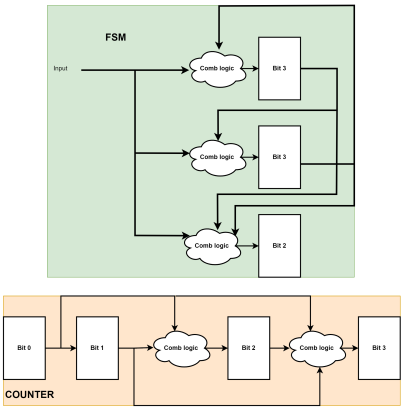

Fig. 5: Example of a counter with no bidirectional path

4) Enable tree-based classification: Finally, there are certain status registers that pass all the above checks but have different control signals. Enable tree identification is done where FSMs are identified as shown in Fig. 6 (register in green is added to the group while the registers not in this group are eliminated). Registers in Set A grouped by the enable tree remain in Set A. Registers in Set A that are not grouped by the enable tree are put in the set of data registers. This step is done in the last part of the algorithm because there could be many data registers that share the same control signal. Control signal-based grouping assumes registers grouped in the control tree to be state registers [6]. Algorithm 1, lines 44 to 54 perform this step.

Algorithm 1 Detection of datapath controllers

1: **procedure** DETECTCONTROLLERS($Netlist$, SMR)
2: ▷ SMR - Sequential Module Registers from identified counters, shifters, and multi-bit registers
3: $SetA \leftarrow all\_flops$
4: $SetDataRegs \leftarrow \emptyset$
5: **for** $Register$ **in** $SetA$ **do**
6: **if** HAS_FEEDBACK($Register$) **then**
7: $SetA \leftarrow Register$ ▷ include register in set
8: **else**
9: $SetDataRegs \leftarrow Register$
10: $SetA.REMOVE(Register)$
11: **end if**
12: **end for**
13: **for** reg **in** $SetA$ **do**
14: **if** fanout(reg) is not $control\_signal$ **then**
15: ▷ net tied to CE pin or net is select line
16: $SetA$.REMOVE(reg)
17: $SetB \leftarrow reg$
18: **end if**
19: **end for**
20: **for** $regC$ **in** SMR **do**
21: **if** $regC$ **in** $SetA$ **then**
22: $SetA$.REMOVE($regC$)
23: $SetB \leftarrow regC$
24: **end if**
25: **end for**
26: $SCC \leftarrow StronglyConnectedComponents(Netlist)$
27: ▷ list of sets
28: **for** $Component$ **in** SCC **do**
29: $Component \leftarrow Component \cap SetA$
30: **end for**
31: **for** $Component$ **in** SCC **do**
32: **if** size($Component$) $==$ 1 **then**
33: SCC.REMOVE($Component$)
34: **end if**
35: **end for**
36: **for** $Component$ **in** SCC **do**
37: **for** $Register$ **in** $Component$ **do**
38: **if** $Register$ has no $bidirectional\_path$ **then**
39: ▷ with other registers in component
40: $Component$.REMOVE($Register$)
41: **end if**
42: **end for**
43: **end for**
44: **for** $Component$ **in** SCC **do**
45: $ET\_Groupings$ = ENABLE_TREE($Component$)
46: **for** $group$ **in** $ET\_Groupings$ **do**
47: **if** size($group$) $<=$ 2 **then**
48: ▷ $group$ has $<=$ 2 registers
49: $Component \leftarrow Component - group$
50: $SCC \leftarrow Component$
51: ▷ update SCC list of FSM components
52: **end if**
53: **end for**
54: **end for**
55: $StateRegs \leftarrow \emptyset$ ▷ Final list of state registers
56: **for** $Component$ **in** SCC **do**
57: $StateRegs \leftarrow Component$
58: **end for**
59: **end procedure**

C. Stand-alone FSM design control register identification

This algorithm is the basic approach that detects the state registers in FSM-only designs and the designs in which all the registers are state registers. This is a two-step algorithm where

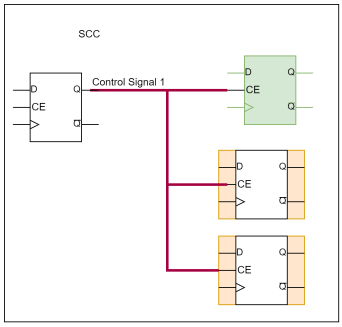

Fig. 6: Enable tree based FP reduction

in the first step all registers are checked for feedback. In the next step we perform enable tree-based grouping on Set A. The control signals are traced back one level and all the registers in the fanin cone of the target elements are identified as state registers. Algorithm 2 describes the flow of this technique.

Algorithm 2 Standalone FSM register identification

```
1: procedure DetectStandaloneFSM(Netlist)
2:     allRegs ← All_flops
3:     for Register in allRegs do
4:         if HAS_FEEDBACK(Register) then
5:             StateReg ← Register                    ▷ append to list
6:         end if
7:     end for
8:     StateReg ← EnableTreeBackTracing(StateReg)
9:                                                    ▷ append to list
10: end procedure
```

IV. Experimentation and Results

A. Experimentation Setup

1) Tools Used: The Hardware Analyzer tool (HAL) [12] is used to aid in the reverse engineering process. The tool accepts a fully flattened FPGA or ASIC synthesized design in Verilog as input. It has a Python interface that helps traverse the netlist. Python3 was used in implementing the algorithms. NetworkX [16] and igraph [17] packages are used to express the circuit as directed and undirected graphs. Xilinx Vivado 2021.1 was used to synthesize designs into flattened LUT-level netlists for analysis with the HAL tool. The designs are synthesized in desired state machine encoding techniques such as One-hot, Sequential, Binary, Gray, and an Auto mode in Xilinx Vivado. The proposed tool flow also gets a set of registers belonging to certain sequential modules detected by the tool developed by [13]. The Benchmarks chosen to evaluate our tool are drawn from ITC 99 [18] benchmarks and OpenCores org site [19]. and the GitHub repository secworks [20] where α, β and γ in the table denotes the benchmarks from the these sources respectively. Standalone FSM designs were used from [21].

B. Results

Phase 1 is executed first and the results are compared with the results after Phase 2. On average, the accuracy increment from Phase 1 to Phase 2 is observed to be 51%. Accuracy is measured considering Set A and the set of data registers obtained. It is observed that the accuracy of register identification has increased considerably and all the control registers are identified. More intricate structural analysis of state registers in Phase 2 made an increase in accuracy possible.

Tables I, II, and III show the accuracy and false positive measures. The tables include the register count of both data and state registers, the registers identified after each phase in the algorithm, True Positives i.e., state registers actually identified as state registers, False Positives - Data registers incorrectly identified as state registers, False Negatives - state registers identified as data registers and finally the accuracy of detection. There are still some false positives in some designs which indicate that there are data-path components that have similar structural characteristics when compared with FSMs in the controller.

The results after Phase 2 are also compared with the benchmarks used in state-of-the-art techniques RELIC. FastRELIC and REIGNN [8] [9] [10] in Figure 7. The results obtained with the above flow of techniques show consistently better or equal accuracy in comparison to the literature techniques with both machine learning algorithms and deterministic algorithms.

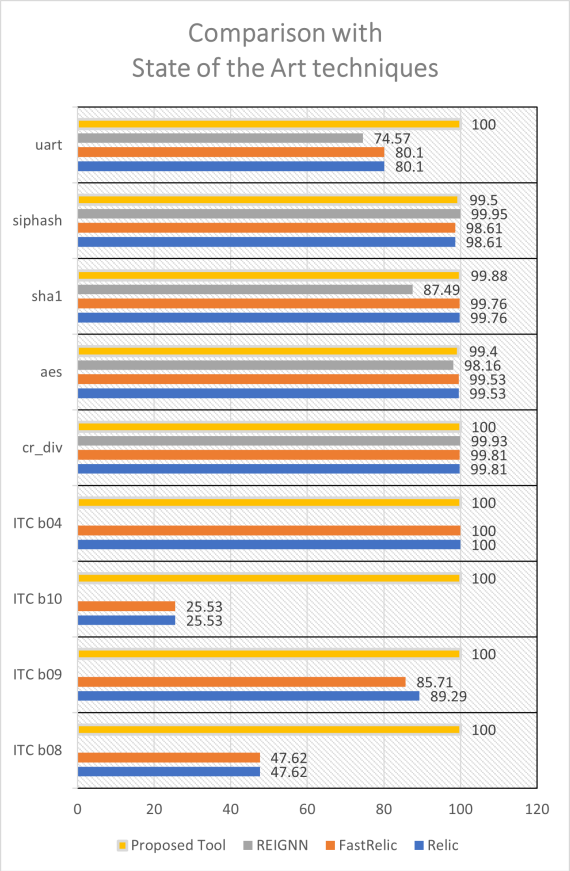

Fig. 7: *Comparison of accuracy against state of the art techniques*

V. Conclusion

We discussed a tool flow to perform state register identification in controllers with or without datapath units and controllers with single or multiple FSMs. We have adapted

Benchmark	Registers		Identified Control Registers		True Positives		False Positives		False Negatives		Accuracy	
	State Registers	Data Registers	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2
Counter	5	0	5	5	5	5	0	0	0	0	100%	100%
Garage door Controller	8	0	8	8	8	8	0	0	0	0	100%	100%
GCD Control logic	5	0	5	5	5	5	0	0	0	0	100%	100%
Triggered Monostable Circuit	12	0	12	12	12	12	0	0	0	0	100%	100%
Light Rotator	7	0	7	7	7	7	0	0	0	0	100%	100%
Signal Generator	7	0	7	7	7	7	0	0	0	0	100%	100%
FSM without bypass	15	0	15	15	15	15	0	0	0	0	100%	100%
Slow Counter	9	0	9	9	9	9	0	0	0	0	100%	100%
FSM of RTC I2C	5	0	5	5	5	5	0	0	0	0	100%	100%

TABLE I: Standalone FSM Designs

Benchmark	Registers		Identified Control Registers		True Positives		False Positives		False Negatives		Accuracy	
	State Registers	Data Registers	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2
ITC b01 $^{\alpha}$	8	6	10	8	7	8	2	0	1	0	80%	100%
ITC b02 $^{\alpha}$	7	4	8	7	5	7	1	0	2	0	87.5%	100%
ITC b03 $^{\alpha}$	3	12	15	3	3	3	12	0	0	0	20%	100%
ITC b04 $^{\alpha}$	3	34	11	3	1	3	8	0	2	0	27.27%	100%
ITC b05 $^{\alpha}$	5	42	18	5	5	5	13	0	0	0	27.77%	100%
ITC b06 $^{\alpha}$	7	5	12	7	3	7	5	0	4	0	58.33%	100%
ITC b07 $^{\alpha}$	7	37	9	7	7	7	2	0	2	0	77.77%	100%
ITC b08 $^{\alpha}$	4	18	11	4	2	4	7	0	2	0	36.36%	100%
ITC b09 $^{\alpha}$	4	45	13	4	1	4	9	0	3	0	30.76%	100%
ITC b10 $^{\alpha}$	11	32	16	11	6	11	5	0	5	0	68.75%	100%
sha1 $^{\gamma}$	3	850	3	3	3	3	12	1	0	0	94.76%	99.88%
siphash $^{\gamma}$	8	794	5	8	5	8	33	4	3	0	88.75%	99.5%
cr_div $^{\beta}$	3	4172	16	3	3	3	13	0	0	0	94.3%	100%

TABLE II: Single FSM Controller Designs

Benchmark	Registers		Identified Control Registers		True Positives		False Positives		False Negatives		Accuracy	
	State Registers	Data Registers	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2	After Phase 1	After Phase 2
UART $^{\beta}$	20	214	41	20	17	20	21	0	3	0	48.7%	100%
SPI $^{\beta}$	12	907	30	9	5	12	18	0	7	0	40%	100%
I2C $^{\beta}$	9	135	39	15	9	9	30	3	0	0	23%	80%
OpenFPU $^{\beta}$	16	744	74	21	9	16	63	5	2	0	17.44%	73.33%
aes $^{\gamma}$	15	2994	45	15	11	15	30	18	4	0	57.5%	99.4%

TABLE III: Multiple FSM Controller Designs

some techniques which are used in ASIC netlist reverse engineering and proposed techniques to reverse engineer FPGA-based designs. With some knowledge of registers belonging to identified sequential modules such as counters, shifters, and multi-bit registers, and analyzing several structural properties concerning the neighborhood of a given register, our toolchain was able to identify state registers with high accuracy. On average 51% increase in accuracy was gained over Phase 1 at end of Phase 2 in the case of real-world designs. This is accompanied by a significant reduction in false positives. The work can be further extended by expressing the FSMs containing the state registers and the associated combinational logic in high-level RTL described in Verilog or VHDL.

ACKNOWLEDGEMENT

This work was supported in part by the National Science Foundation under IUCRC-1916762, and CHEST (Center for Hardware Embedded System Security and Trust) industry funding.

REFERENCES

[1] L. Azriel, J. Speith, N. Albartus, R. Ginosar, A. Mendelson, and C. Paar, "A survey of algorithmic methods in IC reverse engineering," *Journal of Cryptographic Engineering*, vol. 11, no. 3, pp. 299–315, 2021.

[2] "Project X-Ray." https://f4pga.readthedocs.io/projects/prjxray/en/latest/index.html.

[3] F. Benz, A. Seffrin, and S. A. Huss, "BIL - a tool-chain for bitstream reverse-engineering," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 735–738, IEEE, 2012.

[4] B. Cakir and S. Malik, "Revealing cluster hierarchy in gate-level IC's using block diagrams and cluster estimates of circuit embeddings," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, jun 2019.

[5] T. Feist, "Vivado design suite," *White Paper*, vol. 5, p. 30, 2012.

[6] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren, "A highly efficient method for extracting FSMs from flattened gate-level netlist," in *2010 IEEE International Symposium on Circuits and Systems*, pp. 2610–2613, 2010.

[7] K. S. McElvain, "Methods and apparatuses for automatic extraction of finite state machines," Jan. 30 2001. US Patent 6,182,268.

[8] J. Geist, T. Meade, S. Zhang, and Y. Jin, "RELIC-FUN: Logic identification through functional signal comparisons," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.

[9] M. Brunner, J. Baehr, and G. Sigl, "Improving on state register identification in sequential hardware reverse engineering," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 151–160, 2019.

[10] S. D. Chowdhury, K. Yang, and P. Nuzzo, "ReIGNN: State register identification using GNN for circuit reverse engineering," in *2021 IEEE/ACM International Conference On CAD (ICCAD)*, pp. 1–9, 2021.

[11] M. Fyrbiak, S. Wallat, J. DÃ©chelotte, N. Albartus, S. BÃ¶cker, R. Tessier, and C. Paar, "On the difficulty of fsm-based hardware obfuscation." Cryptology ePrint Archive, Paper 2019/1163, 2019. https://eprint.iacr.org/2019/1163.

[12] S. Wallat, N. Albartus, S. Becker, M. Hoffmann, M. Ender, M. Fyrbiak, A. Drees, S. Maaßen, and C. Paar, "Highway to HAL: open-sourcing the first extendable gate-level netlist reverse engineering framework," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pp. 392–397, 2019.

[13] R. V. Narayanan, A. N. Venkatesan, K. Pula, S. Muthukumaran, and R. Vemuri, "Reverse engineering word-level models from look-up table netlists," *https://arxiv.org/abs/2303.02762*, 2023.

[14] A. Nathamuni-Venkatesan, R.-V. Narayanan, K. Pula, S. Muthukumaran, and R. Vemuri, "Word-level structure identification in FPGA designs using cell proximity information," *arXiv:2303.07405v1 [cs.AR] 7 Mar 2023*, 2023.

[15] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[16] A. Hagberg and D. Conway, "Networkx: Network analysis with P ython," *URL: https://networkx. github. io*, 2020.

[17] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006.

[18] S. Davidson, "ITC-99 benchmark circuits - preliminary results," in *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, pp. 1125–1125, 1999.

[19] "The reference community for Free and Open Source gateware IP cores." https://opencores.org/.

[20] "secworks." Online - available https://github.com/secworks.

[21] V. Pedroni, *Finite State Machines in Hardware: Theory and Design (with VHDL and SystemVerilog)*. MIT Press, 2013.