RTL-FSMx: Fast and Accurate Finite State Machine Extraction at the RTL for Security Applications

Rasheed Kibria, M. Sazadur Rahman, Farimah Farahmandi and Mark Tehranipoor
Department of Electrical and Computer Engineering, University of Florida, Gainesville, Florida
Email: {rasheed.kibria, mohammad.rahman}@ufl.edu, {farimah, tehranipoor}@ece.ufl.edu

***Abstract*—At the early stage of the design process, many security vulnerability assessment solutions require fast and precise extraction of the finite state machines (FSMs) present in the register-transfer level (RTL) description of the design. FSMs should be accurately extracted for watermark insertion, fault injection assessment of control paths in a system-on-chip (SoC), information leakage assessment, control-flow reverse engineering in RTL abstraction, logic obfuscation, etc. However, it is quite unfortunate that, as of today, existing state-of-the-art synthesis tools cannot provide accurate and reliable extraction of all FSMs from the provided high-level RTL code. Precise identification of all FSM state registers and the pure combinational state transition logic described in the RTL code with numerous registers and other combinational logic makes it quite challenging to develop such a solution. In this paper, we propose a framework named RTL-FSMx to extract FSMs from high-level RTL codes written in Verilog HDL. RTL-FSMx utilizes node-based analysis on the abstract syntax tree (AST) representation of the RTL code to isolate FSM state registers from other registers. RTL-FSMx automatically extracts state transition graphs (STGs) for each of the detected FSM state registers and additional information of the extracted FSMs. Experimental results on a large number of benchmark circuits demonstrate that RTL-FSMx accurately recovers all control FSMs from RTL codes with various complexity and size within just a few seconds.**

***Index Terms*—Register Transfer Level, FSM Extraction, HDL Code Analysis, Security Assessment, FSM Automata Theory**

I. Introduction

Today, it is impossible to build modern System-on-Chips (SoCs) from scratch due to their complex functionality and design reuse. IC designers license high-level intellectual property (IP) blocks for their SoCs in the form of soft (RTL codes) or hard (GDSII) IPs. The growth in consumer electronics and the cost of maintaining a cutting-edge semiconductor facility has forced fabless semiconductor companies to outsource their fabrication and testing to offshore foundries. With more and more high-level design being outsourced and fabrication often taking place at offshore foundries, third-party IP cores and commercial off-the-shelf ICs have become the norm in hardware and system development. This globalized business model increased growth, lowered costs, and decreased time-to-market. With many potentially untrusted parties involved in the supply chain, multiple security vulnerabilities emerge, such as IP theft/piracy, tampering, counterfeiting, reverse engineering (RE), and IC overproduction, to name a few [1], [38].

The horizontal business model in the semiconductor industry has introduced ICs with several vulnerabilities, among which unauthorized modifications or inclusions, *a.k.a.* hardware Trojans, received considerable attention. Generally, a hardware Trojan is an intentionally modified circuit design causing undesired behavior (e.g., information leakage, denial of service, reliability concern, etc.) when deployed [2]. While researchers primarily focus on protecting the data path circuitry of security-critical components, too little attention has been given to the design's control machine. Typically, FSMs are used as controllers in digital circuits and, therefore, have a significant impact on the behavior and performance of the system, especially when working with control-intensive microprocessors. An FSM is a generalized form of sequential logic in which a machine's next state is presented as a function of its current state and the information it receives from its input signals. Researchers have shown that by using the don't-care states and transitions introduced by the synthesis process of the FSM, an attacker can implant hardware Trojans into the design [35]. Moreover, researchers demonstrated that even if the data path is adequately protected, the key to RSA encryption can be leaked [35] by injecting faults into the FSM of the cryptographic device by leveraging the Montgomery ladder algorithm. To protect against fault-injection attacks and hardware Trojans, FSMs must be detected at the earliest possible stage of the design to establish trust among different entities in the supply chain. As modern SoCs include IPs from various entities worldwide, accurate extraction of the FSMs at the higher abstraction level of the design is crucial.

IP watermarking and hardware obfuscation [3] have emerged as promising solutions to IP piracy. One of the standard IP watermarking techniques is FSM-based watermarking, where don't care states or unused transitions in the FSM are utilized to embed the ownership signature [5]. Hardware obfuscation focuses on concealing the functionality of IP/IC by inserting additional gates, which can be divided into two categories, namely *combinational* and *sequential*. While combinational obfuscation techniques concentrate on the combinational parts, sequential obfuscation techniques target the state transition characteristics of the circuits. Precise extraction of the available FSMs in the design [4] is the first step to embedding watermarking or inserting sequential obfuscation. Moreover, hardware RE can help create a better understanding of an unknown circuit to facilitate the identification of any malicious inclusion and/or tampering by untrusted entities. In order to prepare a meaningful defense, one must understand the attacker's methods. RE can be utilized by persuading potential countermeasures to prevent design functionality from being extracted. Insights into the RE process can help with obfuscation or other defenses to combat future vulnerabilities.

Aside from the security applications mentioned earlier, numerous hardware verification and design optimization algorithms require fast and precise identification and extraction of all the control FSMs. In the optimization algorithm of a typical VLSI logic synthesis process, the state space and output load of the extracted control FSMs are analyzed [6]. Different state encoding schemes can yield a highly optimized implementation of control FSMs based on each state's state transition pattern and the associated output load. Furthermore, control FSMs also need to be analyzed in the hardware verification arena. With the shrinking of device geometry, the

size of an SoC increases significantly, eventually leading to the exponential growth of the state space of control FSMs [7]. As a consequence, verification of the entire SoC becomes exceptionally complicated. A quick solution to this problem is verifying the control logic portion of the SoC separately.

The automated extraction of control FSMs has been researched for a while. Despite this, there is no good solution that can extract FSMs quickly and accurately in a complex SoC involving multiple FSMs and clock domains. In this paper, we propose a framework named **R**egister **T**ransfer **L**evel **F**inite **S**tate **M**achine E**x**tractor (RTL-FSMx) to extract control FSMs from designs specified in register-transfer level (RTL) abstraction systematically with a short computational time and 100% accuracy. RTL-FSMx utilizes node-based analysis on the abstract syntax tree (AST) representation of the RTL code to extract all control FSMs of the designs with varying complexity and size. Specifically, our contributions in this paper are as follows-

- Developing RTL-FSMx, an automated framework for fast and accurate FSM extraction from RTL codes written in Verilog HDL;
- Extracting human-readable STGs for each of the detected control FSMs present in the design;
- Demonstrating the efficacy of RTL-FSMx on extracting control FSMs of a memory controller, Z80 core, and many benchmark designs.

The rest of the paper is organized as follows. In Section II, we present definitions of the terminologies used in the paper. In Section III, we discuss the underlying motivation of this work. Section IV describes our proposed RTL-FSMx framework in detail. We present experimental results with algorithmic complexity and effectiveness analysis of RTL-FSMx in Section V. Applications of RTL-FSMx are presented in Section VI. Finally, Section VII concludes the paper.

II. Preliminaries and Background

Finite State Machine (FSM): A *Finite State Machine* can be represented as a 6-tuple entity (S, I, O, s_0, ϕ, λ) from mathematical perspective. Here, S is a finite set of states, I is a finite set of inputs, O is a finite set of outputs, s_0 is the reset (or initial) state of the FSM, $\phi : S \times I \rightarrow S$ is the state transition function that determines the next-state of the FSM and λ is the output logic function. The general architecture of a typical FSM has been illustrated in Fig. 1. The high-level structure of an FSM is comprised of three major components: (i) the *State Register* that stores the current state of the FSM (sometimes also referred as *State Memory*) and implements S, (ii) the pure combinational *State Transition Logic* that implements ϕ, and (iii) the *Output Logic* of the FSM which implements λ.

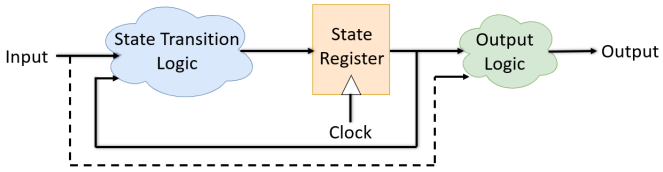

Fig. 1: Generic architecture of a typical FSM. The dashed line is present only in the general architecture of Mealy FSM.

Mealy FSM and Moore FSM: Depending on the nature of the output logic function, FSMs can be classified into two main categories: Mealy FSM [8] and Moore FSM [9]. If the output logic of an FSM depends entirely on the present state of the FSM, then the FSM is defined as a Moore FSM, and the output logic can be expressed mathematically as $\lambda : S \rightarrow O$. On the contrary, if the output logic function of the FSM depends not only on the current state of the FSM but also on the inputs, mathematically $\lambda : S \times I \rightarrow O$, then the FSM is termed as a Mealy FSM. As shown in Fig. 1, the output logic of a Moore FSM is driven by the current state of the FSM, which is provided by the FSM state register. However, the output logic function of a Mealy FSM is determined by both the present state and the inputs of the FSM (represented by the dashed line in Fig. 1).

State Transition Graph: *State Transition Graph (STG)* of an FSM is mathematically defined as a directed graph where each node (or vertex) of the graph represents a particular state $s \in S$ and each edge of the graph represents a certain transition between two states, $t = T\ (s_i,\ s_j)$ from the current state s_i to its next state s_j [35]. The current state s_i and the next state s_j can also be termed as *source state* and *destination state* respectively for that particular transition $T\ (s_i,\ s_j)$. In Fig. 2, the state transition graph of the control FSM of NIST SHA-512 design [10] is presented which was extracted using *Intel Quartus Prime* tool. As evident from the figure, the STG has 3 nodes implying a total of 3 states in the FSM: CTRL_IDLE, CTRL_ROUNDS, and CTRL_DONE. It is also obvious from the figure that there is a total of 6 edges, i.e., state transitions in the FSM. For example, the state transition *T (CTRL_DONE, CTRL_IDLE)* implies that the control FSM switches to the destination state CTRL_IDLE from the source state CTRL_DONE due to single or multiple transition conditions specified in the state transition logic.

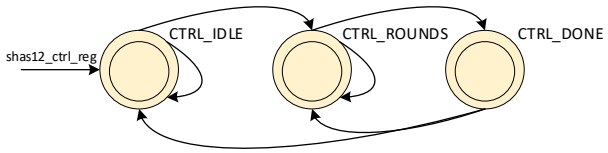

Fig. 2: State transition graph of SHA-512 control FSM. The nodes and the edges of the directed graph represent the states of the FSM and the transitions between two states.

Reset State: The *Reset State* of an FSM is defined as the entrance state to the other states existing in an FSM according to FSM automata theory [11]. As the name implies, the reset state of an FSM represents a particular state to which an FSM switches when the reset condition is applied to the FSM. The reset condition can be applied to a certain FSM by either applying a logic '0', i.e., *Active-Low Reset*, or a logic '1', i.e., *Active-High Reset*, on the reset port of the FSM. Moreover, a particular design can be driven by either global reset (reset condition is applied to the entire design) or local reset (reset condition is applied to only a specific region of the entire design, and the rest operates as expected). However, for both scenarios, the FSM transits to the *Reset State* from the current state instantaneously (if an asynchronous reset mechanism drives the FSM) or at the next positive or negative edge of the clock (if the FSM is controlled by synchronous reset mechanism). For the SHA-512 control FSM depicted in Fig. 2, the CTRL_IDLE state is the reset state.

Control FSM and Control Signal: If an FSM acts as the control logic of a particular design, it is defined as a control FSM. We provide this definition to distinguish between control FSMs and counters clearly. More precisely, control FSMs are responsible for sequencing and controlling operations

happening inside the datapath of a complex design. However, a counter is generally used to count sequentially in a pre-defined manner. If a particular signal is generated by the output logic function of a control FSM, then it is defined as a control signal since it can be used to sequence and control the operations of the datapath of an intricate design. In the general architecture of a typical control FSM as illustrated in Fig. 1, the output signals generated from the output logic of the FSM are termed as control signals in this paper.

State Encoding Schemes: States of a certain FSM can be encoded using three primary techniques: *Binary, Gray, and One-Hot*. Using a particular state encoding scheme in the design relies on the design's optimization goal, such as speed, area, or power consumption. In the binary state encoding scheme, all states of the FSM are enumerated serially initiating from 0 in order of their appearance and can be represented using a state register having a width of $|\log_2(|S|)|$ bits, where $|S|$ is the number of states of the FSM. In a one-hot state encoding approach, the FSM states are encoded so that all state encoding bits except one are equal to 0 at any point. Consequently, each state of the FSM can be presented with a state register having a width of $|S|$ bits. Finally, in the Gray state encoding technique, the states of the FSM are encoded so that the bit difference between the binary represented state encoding values of two consecutive states is 1. Like the binary state encoding method, the Gray encoding of the states of an FSM results in implementing a state register with a bit width of $|\log_2(|S|)|$ bits. The three states of the SHA-512 control FSM: CTRL_IDLE, CTRL_ROUNDS, and CTRL_DONE, shown in Fig. 2, can be encoded as 2-bit vectors: '00', '01' and '10' respectively if the binary encoding is applied. The same states can be encoded as '00', '01', and '11' if Gray encoding is employed, and finally, if the one-hot encoding is utilized, then the states need to be encoded with 3-bit vectors: '001', '010' and '100' respectively. The power, performance, and area of an FSM are affected by the choice of the state encoding scheme. The state encoding choice of the control FSMs influences the overall hardware implementation of a particular design significantly [12]–[14].

III. Related Work and Motivation

The FSM detection schemes proposed in existing literature can be broadly classified into two major categories: template matching of standard HDL coding practices and slicing of program source code. Contemporary state-of-the-art commercial synthesis tools [31], [32] are good examples of matching templates of particular HDL coding styles to detect FSMs. Traditionally, control FSMs are described in the standard template of a single or two 'always' blocks if written using Verilog HDL [27]–[29]. However, these commercial tools cannot always extract all the control FSMs present in a complex design 100% accurately due to some strict restrictions on the HDL coding style and few requirements on the design. For example, *Design Compiler* tool from *Synopsys* fails to extract control FSM if there are multiple control FSMs in the design or clock domain of the control FSM differs from the rest of the fabric. This scenario often leaves a significant portion of the control circuitry of the design sub-optimized [30]. Although some of these issues seem to be resolved in *Quartus Prime* tool from *Intel*, there still exists some inaccuracy in the FSM extraction results, which has been illustrated in Section V via experimental results on two large-scale benchmarks from [16].

The slicing of program source code is another potential candidate for extracting control FSMs. However, some initial conditions need to be satisfied for applying this methodology: the names of the control FSMs of interest are previously known, and the input program is not compiled. This technique directly works on the program source code instead of an intermediate representation. A typical program slicing algorithm yields all the codes related to a particular set of signals, given that the programmer specifies the set of signals through a line-by-line inspection of the program source code. The extraction result from such an algorithm is a truncated program source code containing only the lines correlated to the provided signals. This scheme was first introduced in the software engineering domain to extract necessary segments of the program code [19]. Eventually, this approach was adopted in hardware description languages to extract control FSMs from large-scale designs [20]. The sub-circuit implementing and utilizing certain control FSMs are the outputs of the program slicing algorithm, given that the associated signals of the control FSMs are provided. Nevertheless, there are some drawbacks to this technique. First, the slicing algorithm's line-by-line scanning process creates scalability issues for SoC-level designs since such designs may contain thousands of lines of code. Secondly, the approach is semi-automated since it requires the users to have a priori-knowledge of the names of the control FSMs. Finally, as appeared from the underlying principles, the algorithm functions similar to an extractor of program codes instead of an FSM recognizer.

Aside from the two techniques discussed above, a method based on heuristics has been proposed recently in the control-flow analyzer of the open-source Pyverilog tool [21]. The control-flow analyzer incorporates a *state machine pattern matcher* subsystem, which is used to implement string-based pattern matching schemes for searching signals with an anticipated name, for example, *state* [22]. The main shortcoming of this heuristic-based approach is that the pattern matching scheme fails if the signals controlling the FSM are named differently, and clearly, it does not guarantee precise extraction of all control FSMs always. Moreover, the control-flow analyzer requires a tremendous amount of time to analyze a medium-sized design. The proposed RTL-FSMx framework addresses the aforementioned fundamental limitations of the existing open-source and commercial tools. Currently, RTL-FSMx focuses on analyzing the hardware designs described using Verilog HDL. However, the framework can be easily extended to support the associated constructs of other languages (VHDL and SystemVerilog).

IV. RTL-FSMx Framework

The high-level overview of the proposed RTL-FSMx framework is presented in Fig. 3. As appeared from the framework, RTL-FSMx incorporates the Abstract Syntax Tree Generator of Pyverilog toolkit [21] in the flow. Pyverilog is an open-source, comprehensive software toolkit for analyzing hardware designs written using a popular hardware description language named Verilog HDL. Verilog HDL is a widespread language to generate high-level representations of a particular hardware design in register-transfer level (RTL) abstraction. Moreover, Pyverilog is the only widespread open-source tool that supports code analysis and generation as a single software package. Pyverilog offers designers and analysts four important libraries for analyzing hardware designs: (1) Abstract

Syntax Tree Generator, (2) Data-flow Analyzer, (3) Control-flow Analyzer, and (4) Verilog HDL Code Generator [22]. However, RTL-FSMx requires only the Abstract Syntax Tree Generator of Pyverilog to generate the abstract syntax tree (AST) representation of the input register-transfer level design written in Verilog HDL.

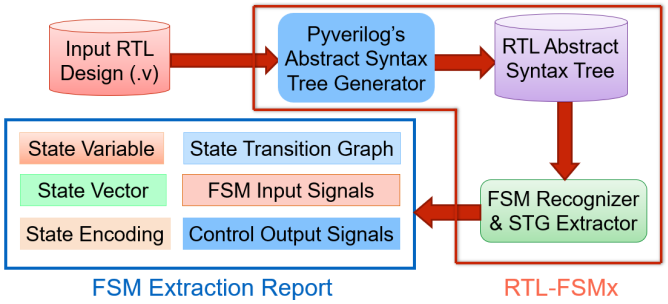

Fig. 3: Overview of the RTL-FSMx framework. The framework embeds the Abstract Syntax Tree (AST) Generator of Pyverilog for generating the AST of the provided RTL written in Verilog HDL. The FSM recognizer & STG extractor module yields an FSM extraction report by analyzing the AST.

A. Abstract Syntax Tree Generation

The Verilog HDL code compiler of Pyverilog, as shown in Fig. 4 is a vital tool for analyzing a Verilog HDL-based source code. The code compiler contains an Abstract Syntax Tree Generator to generate an AST presentation of the input HDL code for later use or external tools. Icarus Verilog [23] acts as the pre-processor for Pyverilog and this external tool is called via *-E* option. After the pre-processing phase, all macros, for instance *ifdef* and *define*, are extracted. The modified source code gets available in text format, and then the compiler of Pyverilog reads the updated source code, and the AST is built. In this manner, the AST gets ready for further analysis. Pyverilog utilizes Python Lex-Yacc (PLY) as the compiler-compiler, lightweight implementation of lexical analyzer, and LR-parser that functions as the syntax analyzer [22].

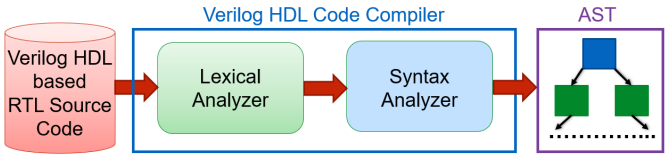

Fig. 4: Overview of the Verilog HDL code compiler of Pyverilog [22]. Pyverilog uses Python Lex-Yacc to implement the lexical analyzer and LR-parser of the Verilog HDL compiler.

The AST generated by Pyverilog acts as an intermediate representation of the input high-level RTL code. The AST representation of the Verilog HDL-based source code is somewhat construct-independent, and it does not include unnecessary punctuation and delimiters present in the source code, for instance, braces, parentheses, semicolons, etc. Moreover, the AST may contain additional information about the program being compiled for subsequent usage by the compiler. For example, the AST generated by Pyverilog stores the line number associated with each element present in the source code. This feature allows Pyverilog to print error messages while compiling a particular Verilog HDL-based RTL. The AST representation of a particular blocking substitution expression used in Verilog HDL has been illustrated in Fig. 5. The expression gets fragmented into numerous nodes and forms a hierarchical tree-like structure. Each node of the AST possesses unique identifiers (IDs), which can be utilized for further processing. Moreover, each node contains a single child node or multiple child nodes in the hierarchy of the AST.

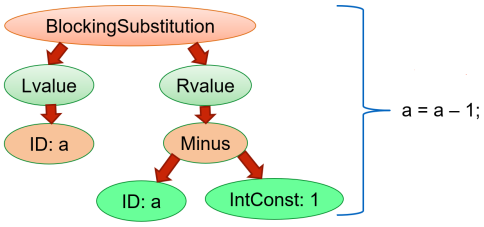

Fig. 5: The AST representation of a sample blocking substitution expression generated by the Abstract Syntax Tree Generator of Pyverilog. The expression gets decomposed into nodes with unique identifiers (IDs), which makes it suitable for further processing.

There are numerous advantages of utilizing AST for analyzing a source code written in a particular language. First of all, programming languages are inherently ambiguous by nature [24], which is also valid for Verilog HDL. As a result, a context-free and construct-independent intermediate representation of the source code written in Verilog HDL is required for analysis and further processing. Secondly, the AST presents the complex RTL source code in a tree-like structure that is easier to analyze than directly analyzing the high-level RTL code. Finally, the provision of AST of RTL source code facilitates node-based analysis. The node-based analysis approach bypasses unnecessary line-by-line searching for a particular object of interest in the input RTL source code since a particular node can be accepted or discarded based on the node's ID. Consequently, this feature helps to improve run-time. RTL-FSMx utilizes AST-based analysis of the input Verilog HDL-based RTL code for such advantages. The AST presentation of a block of RTL code written in Verilog HDL representing the state register of an FSM has been illustrated in Fig. 6. Similarly, any block of RTL codes can be treated as a group of nodes preserving hierarchy inside the AST.

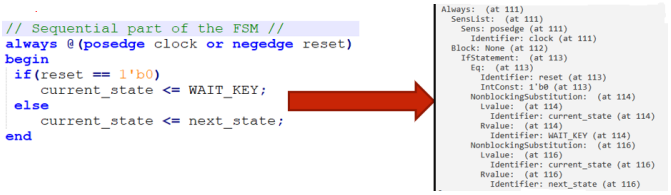

Fig. 6: The AST representation of an example block of RTL code representing the state register of an FSM yielded by Pyverilog. The block of the RTL code gets split into a collection of nodes with unique IDs. Pyverilog also reports the line numbers associated with a particular node of the AST.

B. Control FSM Recognition and STG Extraction

After the generation of AST of the input Verilog HDL-based RTL code is completed by Pyverilog, the *FSM Recognizer & STG Extractor* module of the RTL-FSMx framework comes into the picture. The high-level overview of the algorithmic flow of this module is presented in Fig. 7, which acts as the central part of the RTL-FSMx framework.

1) Identification of Sequential Blocks: The first stage is identifying all the sequential blocks in the high-level RTL design description. The sequential blocks of the RTL code represent either flip-flops or registers of the design specified

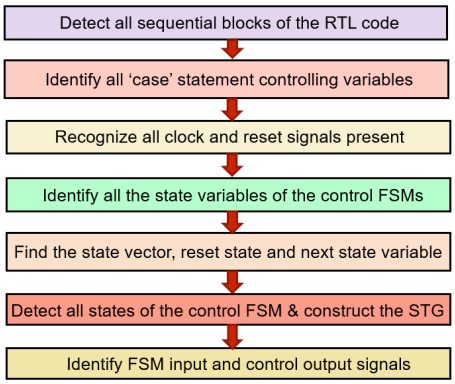

Fig. 7: Algorithmic flow of the FSM Recognizer & STG Extractor module of RTL-FSMx. It is the core part of the RTL-FSMx framework that implements a precise and fast FSM extraction technique from high-level RTL codes.

by the designer. More precisely, all 'always' blocks having sensitivity to edge-triggered signals are identified. The reason emerges directly from the Verilog HDL coding guidelines for writing good RTL codes targeted for synthesis [27], [28], which designers follow widely. In a Verilog HDL-based RTL code, 'always' blocks with edge-triggered signals represent registers, and blocks without any edge-triggered signal represent either combinational logic blocks or latch units of the design. As shown in Fig. 1, the sequential part of an FSM is known as the *State Register* of the FSM. For this reason, we are primarily interested in identifying all the registers present in the RTL design since a portion represents state registers. Perfect identification of all FSM state registers is a prerequisite for successfully identifying all control FSMs of the design.

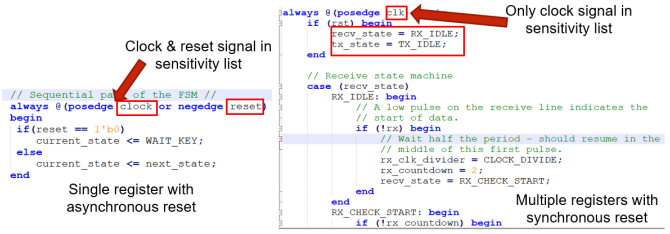

Fig. 8: Example Verilog HDL code snippets for describing a single register with asynchronous reset and multiple registers driven by synchronous reset signals [26]. Edge sensitivity type of the signals specified in the sensitivity list of the 'always' blocks must be mentioned to be compatible with existing state-of-the-art commercial synthesis tools [30], [32].

In Fig. 8, sample Verilog HDL code snippets for implementing a register with asynchronous reset and multiple registers driven by synchronous reset have been illustrated. When both clock and reset signals are present in the sensitivity list of the 'always' block of a register, the register is said to have asynchronous reset capability because driving the reset signal with either logic '1' or logic '0' causes the register to be reset. It happens suddenly without depending on the status of the clock signal. On the contrary, if the clock signal is present in the sensitivity list of the 'always' block of a particular register only, the register gets driven by synchronous reset since the register does not get reset until the next active edge (either positive or negative) of the clock appears. An example code snippet to implement a typical combinational logic block using Verilog HDL has been illustrated in Fig. 9. The 'always' block representing a combinational logic depends on all the level-triggered inputs. Such nodes of the AST ('always' blocks to describe combinational logic) are discarded by RTL-FSMx. However, a typical 'always' block implementing a register depends on both clock and reset signals (register having asynchronous reset ability) or only on the clock signal (register with synchronous reset capability). RTL-FSMx accepts nodes of the AST with such characteristics for further analysis to distinguish FSM state registers from the non-state ones.

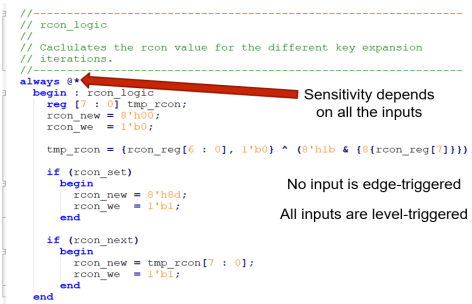

Fig. 9: Sample Verilog HDL-based code snippet to implement a typical combinational logic block in RTL abstraction [25]. The 'always' block has dependence on all the inputs, and no input is edge-triggered. All of the inputs are level-triggered.

2) Identification of 'case' Statement Blocks: The next step is quite crucial. All the controlling variables of the 'case' statement blocks present in the RTL source code are appropriately identified at this stage. The underlying reason is that among these variables, there exist the state ones. From the Verilog HDL coding guidelines for writing RTL codes for FSMs presented in [27], [28], it is evident that not all 'case' statement block controlling variables are state variables, but all state variables are variables controlling some 'case' statement blocks. These blocks represent the associated combinational state transition logic blocks of the control FSMs.

3) Recognition of Clock and Reset Signals: The clock and reset signals of corresponding registers can be recognized when the register blocks of the RTL code representing registers have been identified. As illustrated in Fig. 8, the clock and reset signals are specified in the sensitivity list of the 'always' block describing a register with asynchronous reset capability in RTL abstraction. In this scenario, RTL-FSMx can recognize the clock and the reset signal driving a register by analyzing the sensitivity list of the 'always' block. On the contrary, for a register with synchronous reset ability, the clock signal is specified in the sensitivity list of the 'always' block representing the register only, and RTL-FSMx identifies that. The reset signal can be identified by analyzing the 'if' statement block containing the reset condition of a particular register which is also illustrated in Fig. 8. The widely used state-of-the-art commercial tool for design synthesis, *Design Compiler* [31] from *Synopsys* can not extract FSMs and provide the corresponding FSM extraction report while using the 'report_fsm' command in all scenarios since it requires designs having only a single clock and an optional synchronous or asynchronous reset signal [30]. Moreover, this drawback makes the Design Compiler tool incapable of analyzing designs having multiple clock domains and more than an FSM in the module. However, RTL-FSMx addresses this shortcoming of the Design Compiler tool by analyzing all the identified register blocks and extracting the corresponding clock and reset signals. This feature of RTL-FSMx makes it unique since control FSMs can be extracted from designs with

multiple clock domains and FSMs.

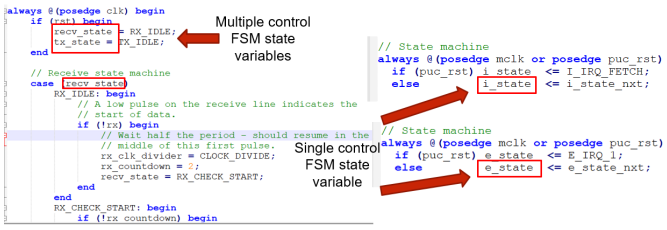

Fig. 10: Example RTL code snippets to illustrate the state variables of control FSMs. The state register block of the control FSM can hold multiple state variables, which is shown in the code snippet on the left side [26]. In the code snippet shown on the right, the two state variables are defined in two different state register blocks [64].

4) Detection of FSM State Variables: Next, the state register identification stage comes into the picture. All the register blocks and their corresponding clock and reset signals present in the RTL have been identified by RTL-FSMx. It is the perfect time to isolate all the control FSM state registers from the non-state ones. The state variable of a control FSM register is the 'case' statement block controlling variable that causes state transition of the FSM, precisely the control variable of the combinational state transition logic [27]–[29]. A 'case' statement block controlling variable should be present inside a sequential 'always' block of the RTL for being considered as the control FSM state variable. In Fig. 10, two RTL coding practices for describing state registers and state variables have been presented. These two coding practices originate from the good RTL coding guidelines for FSMs presented in [27]–[29]. Multiple state registers can be described in a single or different register block. The 'case' controlling variables are present in the sequential 'always' block implementing registers.

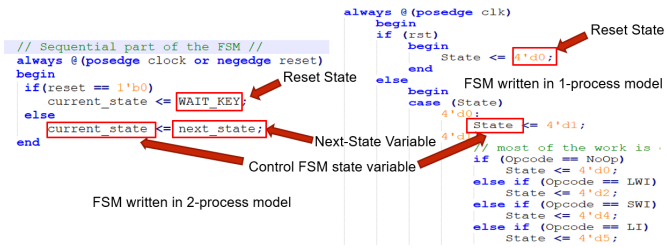

Fig. 11: Identification locations of the reset state and next-state variable of a control FSM in a sample RTL code snippet written using Verilog HDL [33]. RTL-FSMx extracts this important information of an FSM by analyzing the 'if' statement block containing the reset information of the FSM.

5) Formation of State Vector: Once the state variable of the control FSM register has been recognized successfully, RTL-FSMx extracts the width of the control FSM register. Identification of the width of the control FSM register is a prerequisite for yielding the state vector representation of the control FSM register. For example, the state variable of a control FSM register named 'i_state' having width of 4 can be represented as the state vector as {i_state[3] i_state[2] i_state[1] i_state[0]}. However, an FSM can be described using two different models in high-level RTL codes written using Verilog HDL [27]–[29]. The models are known as 1-process and 2-process models. In the 1-process model, the combinational state transition logic and state register parts of the control FSM are described in a single 'always' block. The 'always' block has a clock and optional reset signals. The reset signal is present in the sensitivity list when the control FSM register has asynchronous reset capability; otherwise, only a clock signal is present for the FSM register with the synchronous ability. The combinational state transition logic can be implemented with a 'case' statement block containing information on all the state transitions of the control FSM, and the state variable acts as the 'case' statement controlling variable. Conversely, the state transition logic and the state register parts are written in two different 'always' blocks. In this manner, RTL-FSMx localizes the state transition logic block and state register block of a particular control FSM, crucial for the subsequent analysis stage.

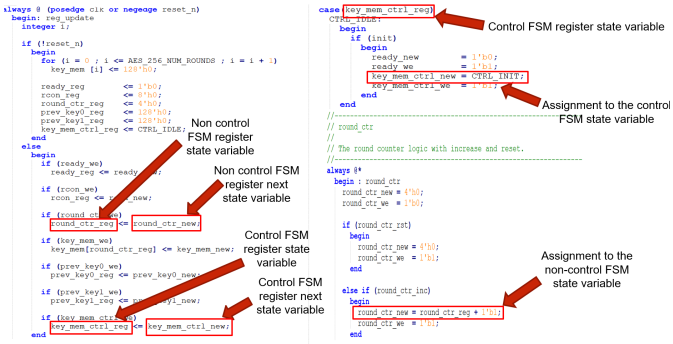

Fig. 12: Sample RTL code snippets written in Verilog HDL to illustrate assignments to a control FSM register state variable and a non-control one [25]. The difference between associated coding practices aid RTL-FSMx in distinguishing control FSMs from the non-control ones (counters and accumulators).

6) Identification of Reset State and Next State Variable: Precise identification of all the state variables of the corresponding control FSMs by RTL-FSMx also aids in detecting the reset states (for both 1-process and 2-process FSM models) and next-state variable (for the 2-process FSM model only). RTL-FSMx extracts this valuable information of the control FSM by analyzing the 'if' block specifying the reset condition of the FSM. In Fig. 11, the extraction points of the reset state and next-state variable (if applicable) have been demonstrated. It is also evident from Fig. 11 that the reset state can be declared using a symbolic variable or numerical entity. The same is true for the rest of the states of the control FSM. Conventionally, the states of a control FSM are declared using 'parameter' or 'localparam' variables or using 'define' statements [27]–[29]. In Fig. 12, example code snippets have been presented to illustrate assignments to a control FSM register state variable and a non-control one which is a counter. It is also evident from Fig. 12 that both the state transition logic blocks of the control FSM and the counter register have been described using 2-process models in the high-level RTL code. However, there is a noticeable distinctive feature of the coding practices for describing a counter and a control FSM. For the control FSM implemented using the 2-process model, the control FSM state variable is assigned to a next-state variable, and that next-state variable is assigned to a single state of the FSM directly, as evident from Fig. 12 or to multiple states of the FSM via some combinational branch determination logic. In contrast, for a counter described using the 2-process model as illustrated in Fig. 12, the non-control state variable of the counter is assigned to a next-state variable, and the next-state variable assignment is written entirely in a different

style. RTL-FSMx utilizes this difference in coding practices for implementing control FSMs and counters in the high-level RTL codes to isolate control FSMs distinctively from the counters. A similar feature also enables RTL-FSMx to separate accumulator registers since good RTL designers implement accumulators in a manner quite similar to the counters. Since RTL-FSMx is intended to extract all the control FSMs present in the provided RTL code, counter and accumulator registers are intentionally discarded.

7) Extraction of the STG: Once the reset state of the FSM has been detected successfully, the next target of RTL-FSMx is to extract the *State Transition Graph (STG)* from the combinational state transition logic of the control FSM. The state variable of the control FSM is the 'case' statement block controlling variable describing the state transition logic. However, sometimes designers specify the 'case' statement controlling variable in a concatenated format with multiple variables combined. The Design Compiler tool from Synopsys and Quartus Prime from Intel fail to handle such a scenario since the practice does not conform to conventional HDL coding practice to represent control FSMs [27]–[29]. RTL-FSMx can handle such coding practice, making RTL-FSMx more promising for detecting control FSMs written using various coding styles. Moreover, RTL-FSMx can identify the state transition logic blocks of all the control FSMs. This feature is missing in the control FSM extraction algorithm of the Design Compiler tool [30] since the FSM extraction algorithm only works perfectly when a particular design has a single FSM inside and the design is driven by a single clock. In other scenarios, using the 'report_fsm' command does not yield any FSM extraction report, which is undesirable when extracting all the control FSMs present in a design is crucial. In Fig. 13, such an example code snippet written using Verilog HDL has been presented, illustrating a coding style for implementing a control FSM described using the 1-process FSM model that existing state-of-the-art commercial tools fail to handle, but RTL-FSMx can handle such scenario.

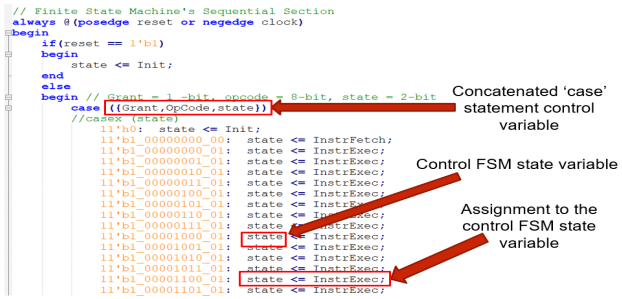

Fig. 13: RTL code snippet of the control unit of the 32-bit RISC processor written in Verilog HDL [34]. The control FSM is described in a non-conventional practice that existing state-of-the-art synthesis tools [31], [32] fail to handle.

RTL-FSMx constructs the STG of the identified control FSM, analyzing the block that implements the combinational state transition logic. All the states of the control FSM with their associated encoding are recognized by analyzing the 'parameter' or 'localparam' type variables (when the state encoding is presented using symbolic format) and getting all immediate descendent nodes of the 'case' statement node holding the state variable. Occasionally, the states of a control FSM are encoded directly using numbers with accepted number bases: binary, decimal, octal, and hexadecimal [27]–[29].

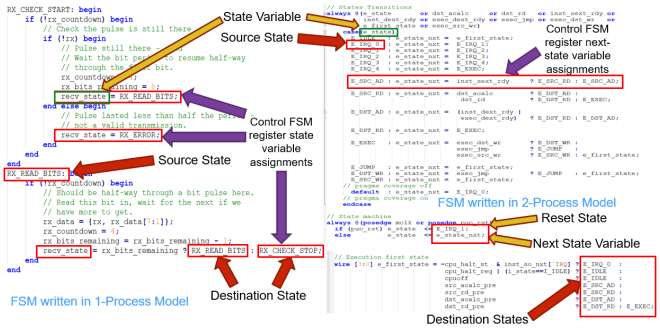

Fig. 14: RTL-FSMx constructs the STG of the control FSM, analyzing the state transition logic block. For this purpose, RTL-FSMx analyzes the assignments to the state or next-state variable, identifies the source and destination states, and generates the STG as a directed graph. The code snippets are taken from [26], [64].

However, RTL-FSMx can handle both practices. Moreover, designers can encode the states of the control FSM using binary, one-hot, or Gray encoding schemes depending on the optimization goals for the design. RTL-FSMx can automatically extract the employed state encoding scheme by analyzing the mathematical relationship between the encoding of the states of the identified control FSM. Moreover, RTL-FSMx yields the binary representations of all the encoded states of the control FSM as the state encoding report as mentioned in Fig. 3. As illustrated in Fig. 14, RTL-FSMx constructs the STG of the detected control FSM by analyzing the state transition logic block. First, RTL-FSMx identifies the nodes assigned to the state variable (for an FSM written in the 1-process model) or the next-state variable (for an FSM written in the 2-process model). Next, the source and destination states are identified by analyzing the assignments. Finally, the STG is represented as a directed graph. RTL-FSMx generates the STG both in textual and graphical format.

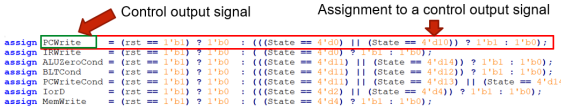

Fig. 15: Example RTL code snippet to illustrate assignments to certain control signals of the RISC controller [34]. RTL-FSMx analyzes such assignments and extracts control output signals generated from the FSM to control the datapath.

8) Extraction of FSM Control Input and Output Signals: Analysis of the state transition logic block of the control FSM serves another purpose of RTL-FSMx. RTL-FSMx can identify all the control inputs of the FSM by analyzing the active state transition conditions. Assignments to the state or next-state variable are analyzed meticulously to identify the state transition conditions. The control inputs of the FSM are not always mentioned in the sensitivity list of the 'always' block, which implements the state transition logic. As mentioned, the analysis feature of RTL-FSMx comes in quite handy for such cases. Finally, nodes of the AST assigned to certain signals influenced by the state variables are identified and analyzed. These signals act as the control signals generated from the FSM and control the datapath of a complex design. An example snippet has been shown in Fig. 15.

V. Experimental Results

The entire RTL-FSMx framework has been implemented using *Python* programming language since it embeds the AST

TABLE I: Worst-case run-time of RTL-FSMx for sample benchmarks obtained from [16]–[18].

Design Name	Module Count	Line Count	'always' Block Count	Gate Count	Flip-flop Count	Control FSM Count	State Register Size	Run-time (s.)
I2C Core [68]	5	1263	22	465	125	2	18, 5	1.09
MC6502 CPU [51]	2	1328	54	970	142	1	6	1.12
NIST SHA-512 [50]	5	1485	16	10763	3666	1	2	1.09
DDR2MC IP [56]	6	1490	24	1699	593	4	3, 2, 4, 3	1.03
GCM AES IP [54]	6	1754	39	17181	1696	1	10	1.49
IBex RISC [57]	7	2122	18	4104	1005	3	4, 1, 3	1.22
HPDMC IP [55]	11	2425	18	1051	351	1	4	1.11
NIST AES-128 [25]	7	2809	22	12976	2987	4	3, 3, 3, 3	1.19
CXD9731 IC [61]	11	2862	19	2835	501	4	6, 5, 5, 4	1.23
MODEXP Core [59]	14	3429	53	154488	57957	3	4, 4, 4	1.18
MC68000 CPU [53]	8	3488	35	9753	1051	1	5	1.23
T2600 SoC [58]	12	3853	22	2093	127	1	5	1.23
MC6809 CPU [52]	2	4233	4	4153	218	1	7	1.44
Z80 Core [62]	28	5489	28	4032	398	1	46	1.26
Memory Controller [60]	14	5719	175	3207	1051	1	66	1.36
JELLY RISC [66]	17	5869	38	9852	1158	2	2, 5	1.29
AMBER25 RISC [63]	20	8288	76	13547	3138	2	4, 4	1.52
openMSP430 Core [64]	21	9164	151	4096	734	4	3, 4, 2, 3	1.44
BIRISCV SoC [65]	33	11953	168	26494	4759	3	2, 4, 4	1.54
OR1200 Core [67]	77	31551	371	201445	69943	4	3, 3, 2, 2	1.81

generator of Pyverilog inside. All the experiments on the benchmark circuits have been performed using an Intel Core i7-1065G7 processor clocked at 1.3 GHz with 16GB RAM on a personal desktop. The high-level RTL codes of the designs were compiled using Pyverilog first to find potential syntax errors. After being assured of no syntax errors in the RTL codes written in Verilog HDL, the RTL-FSMx framework was examined to verify its effectiveness in recognizing and thus extracting all control FSMs present in the benchmark designs. Experimental results on 20 benchmarks obtained from several open-source repositories [16]–[18] have been provided in Table I to demonstrate that RTL-FSMx can extract all control FSMs very fast, varying widely in terms of functional complexity and size of the RTL designs. The associated gate count and flip-flop count were obtained from the synthesis report generated by *Design Compiler*, and *SAED90nm* was used as the target technology library. In addition, case studies on two benchmarks have been illustrated to prove that RTL-FSMx can extract all control FSMs more accurately compared to commercial synthesis tools from *Synopsys* and *Intel*.

A. Complexity of RTL-FSMx

The algorithmic complexity of RTL-FSMx can be assessed in two ways: time complexity analysis and memory (space) complexity analysis. The worst-case time complexity of the RTL-FSMx framework depends on the number of 'always' blocks present in the input RTL code written in Verilog HDL. The 'case' statement blocks implementing the combinational state transition logic portions of the control FSMs are also specified inside 'always' blocks. Hence, they are counted in the total number of 'always' blocks present. Mathematically, the time complexity of the RTL-FSMx framework is *O(M)* where *M* is the total number of 'always' blocks present in the provided high-level RTL code. Additionally, in the worst-case scenario, the entire AST containing *N* nodes in total needs to be stored in the computer memory. Therefore, the worst-case space complexity of the RTL-FSMx framework is *O(N)* where *N* is the total number of nodes present in the AST representation of the input RTL design. The linear nature of the overall algorithmic complexity of the RTL-FSMx framework considering both time and space complexity, explains why RTL-FSMx is very fast and memory-efficient for extracting all control FSMs present in the design. The worst-case run-time of the RTL-FSMx framework for large-scale benchmark RTL designs written in Verilog HDL obtained from [16]–[18] has been presented in Table I. The total number of design modules, lines, and 'always' blocks present in the RTL code, control FSM count, and corresponding state register size extracted by RTL-FSMx have also been illustrated.

B. Control FSM Extraction Accuracy of RTL-FSMx

Illustrations of two case studies have demonstrated that the RTL-FSMx framework can extract all control FSMs perfectly from RTL designs diversified in size and functional complexity. The first benchmark circuit is the RTL implementation of *Memory Controller* IP [60]. The second one is the RTL design of *Z80 Core* [62]. The extracted gate-level STGs of the control FSMs have been compared with the corresponding RTL descriptions of the FSMs to evaluate the control FSM extraction accuracy of RTL-FSMx. We define a simple metric named *Control FSM Extraction Accuracy (CFEA)* to calculate the control FSM extraction accuracy as shown in Eq. 1, where *D* is the total number of extracted control FSMs and *T* is the total number of control FSMs present in the provided RTL design. Moreover, the extracted STGs of the control FSMs of the benchmark RTL designs, listed in Table I, by RTL-FSMx were compared with the STGs extracted by the *JasperGold* formal verification tool from *Cadence*. It was found that extracted STGs always matched perfectly, manifesting that the accuracy of RTL-FSMx is 100%.

$$CFEA = \frac{D}{T} \times 100\% \tag{1}$$

1) Memory Controller IP: The *Memory Controller* IP from [16] is intended for various embedded applications. It supports SDRAM, SSRAM, FLASH memory, ROM, and several other devices. It has eight chip selects, and each of them is programmable. Moreover, it provides default boot sequence support with other important features [60]. The complex RTL has 14 design modules and 5,719 lines of codes, as shown in Table I. 175 'always' blocks are present, and the states of the control FSM are encoded using the one-hot technique having 66 control flip-flops forming the corresponding state register. As the RTL design contains multiple modules, the 'report_fsm' command did not yield any FSM extraction report while *Design Compiler* was used. Moreover, using the state machine viewer of *Quartus Prime*, it was found that the control FSM was detected, and the FSM was shown using a yellow box. However, the STG of the FSM was not extracted at all as it displayed a blank window, which indicates the FSM extraction inaccuracy of *Quartus Prime* due to the complexity of the associated combinational state transition logic block in the RTL code and the STG. We analyzed all the design

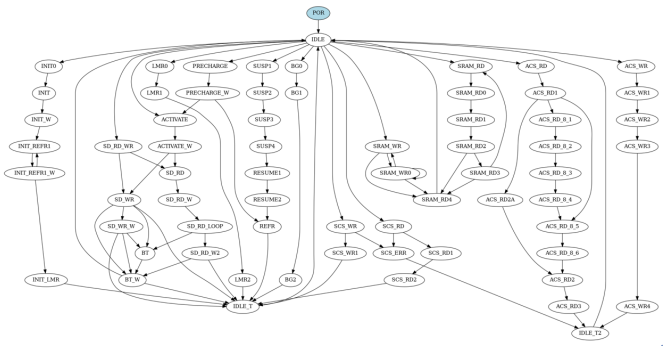

Fig. 16: The STG of the control FSM of the *Memory Controller* IP extracted by RTL-FSMx. It was not extracted by *Quartus Prime* due to the complexity of the state transition logic and the STG.

modules of this IP using RTL-FSMx, and the extracted STG of the control FSM is shown in Fig. 16. The run-time of RTL-FSMx for this IP was only 1.36 s.

2) Z80 Core: The verified Z80 compatible processor core presented in [62] is an illustration of a complex core, and the RTL design written in Verilog HDL contains 28 design modules and 5,489 lines in total. This benchmark analysis seemed quite challenging since the state transition logic of the control FSM of the Z80 core is quite complex, with 46 control flip-flops and the binary state encoding technique employed to encode the states of the complex control FSM. Due to the complexity of the state transition logic, the control FSM was not detected by the *Quartus Prime* tool even after considering the design module implementing the control FSM as the top module of the hierarchy. The *Design Compiler* tool also failed to provide an FSM extraction report since the design hierarchy contains multiple modules to be analyzed. However, in the worst-case scenario, it took only 1.26 s. for RTL-FSMx to compile all the design modules and thus extract the control FSM with the corresponding STG and associated information. The complex STG of the control FSM extracted by RTL-FSMx is shown in Fig. 17. Finally, these two case studies and results presented in Table I demonstrate that the *CFEA* of RTL-FSMx is overall 100%, but existing commercial synthesis tools [31], [32] lack such a level of accuracy, unfortunately.

VI. Applications of RTL-FSMx

The state encoding choice of an FSM significantly affects reverse engineering of the design in RTL and gate-level netlist abstractions apart from design optimization [48]. RTL-FSMx yields state encoding information along with the state variable, the size of the state register, and the state vector representation of the control FSM automatically, which may help the reverse engineers gain insight into the design strategies employed by the designer. For instance, using RTL-FSMx, if an analyst finds that Gray encoding of control FSMs is utilized in a particular design, it can be assumed that minimizing power consumption has been taken as the design strategy. Similarly, assumptions can be made for other state encoding schemes. Binary state encoding minimizes the area, and one-hot encoding is used to make the design compatible with higher clock frequency [12]–[15]. Such information obtained after control-flow reverse engineering using RTL-FSMx can help understand the employed design strategy in an SoC from a competitor in the high-level abstraction of the design flow.

The RTL-FSMx framework automatically extracts all control input and output signals of a particular control FSM. This

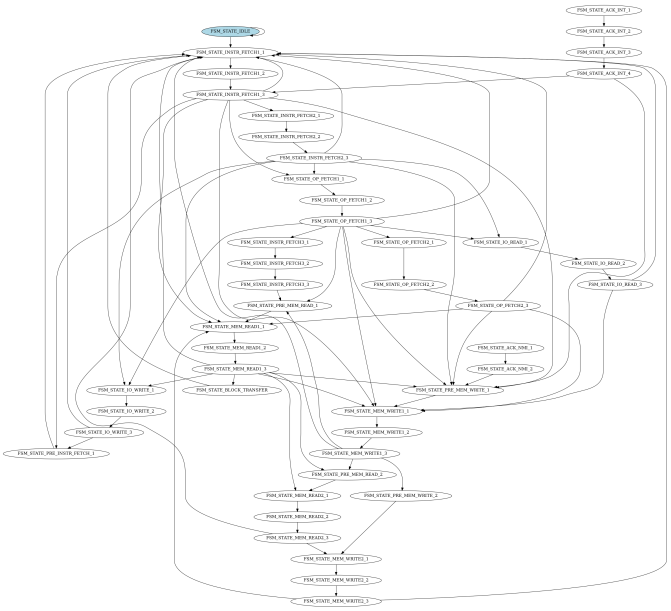

Fig. 17: The complex STG of the control FSM of Z80 core extracted by RTL-FSMx. Synthesis tools failed to recognize the control FSM though HDL coding guideline is followed.

report and the extracted STG in both textual and graphical formats can be utilized to reverse engineer the control flow of a complex SoC in high-level RTL abstraction. Additionally, these reports can be highly efficacious for some other security applications. Fault-injection assessment of the control flow of a particular design in gate-level netlist abstraction has been proposed in [35]–[37]. Similarly, assessment can also be performed at the early stage of the design, specifically in RTL abstraction, by analyzing the STG report generated by RTL-FSMx and identifying the vulnerable state transitions to thwart fault-injection and hardware Trojan attacks. Existing state-of-the-art synthesis tools [31], [32] do not provide the STG of the extracted FSM in textual format. Hence, designers cannot modify these tools for such assessment due to the proprietorship of the source codes. Sometimes, the STG of a control FSM can be quite complex, making it impossible to construct manually. The STG extraction feature of RTL-FSMx can be quite beneficial for such a scenario.

Several FSM-based IP watermarking schemes have been proposed in existing literature [39]–[44]. Besides, numerous sequential logic locking techniques have been presented in [45]–[49]. Precise identification and extraction of all control FSMs and other relevant information present in the RTL design are essential for such security applications as a pre-processing task, and RTL-FSMx is a distinctive candidate for this. Existing commercial synthesis tools [31], [32] cannot be used due to associated reliability issues in extracting all control FSMs of a complex design 100% accurately. Apart from the security applications, RTL-FSMx can be employed for several non-security applications. For instance, RTL designers can utilize RTL-FSMx to verify the entire control flow of their IPs and complex SoCs described using Verilog HDL reliably. Moreover, since ignoring FSMs results in sub-optimized control circuits [30], the RTL-FSMx framework can help the *Design Compiler* tool from Synopsys to optimize all the control FSMs present in the design, consequently generating a more optimized gate-level netlist of the entire design. In addition, if *Design Compiler* fails to recognize the

control FSM, manual efforts are required to identify the state variable of the control FSM and set the state vector with encoding for optimization [30]. It can be alleviated if RTL-FSMx is integrated with the optimization flow of the *Design Compiler*. The same reasoning goes for *Quartus Prime* since it also suffers from reliability issues in the precise extraction of all control FSMs. Therefore, RTL-FSMx is an outstanding candidate for all these applications.

VII. Conclusion

This paper presents a fast and precise technique mounted on node-based analysis of the AST representation of the input RTL source code written in Verilog HDL to automatically extract all the control FSMs present in the design with associated relevant information. Experimental results on several benchmark RTL designs varying in size and complexity have proved the efficacy of RTL-FSMx, which is absent in some of the existing commercial CAD tools. We envision incorporating RTL-FSMx to develop novel sequential logic obfuscation and control FSM-based watermarking techniques in the future. We also envision utilizing RTL-FSMx for performing fault-injection and information-leakage assessments in high-level RTL abstraction. It may open a new horizon in detecting potential security vulnerabilities at the early stage of the state-of-the-art VLSI design and implementation flow.

References

[1] B. Shakya et al., "Introduction to hardware obfuscation: Motivation, methods and evaluation," in *Hardware Protection through Obfuscation*. Springer, 2017, pp. 3–32.
[2] K. Xiao et al.,"Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 22, issue 1, pp 1–23*, 2016.
[3] M. Rahman et al.,"Security Assessment of Dynamically Obfuscated Scan Chain Against Oracle-guided Attacks," *ACM TODAES*, vol. 26, pp. 1-27.
[4] M. Rahman et al.,"Defense-in-depth: A recipe for logic locking to prevail," *Elsevier Integration 2020*, vol. 72, pp. 39-57.
[5] N. Anandakumar et al.,"Rethinking Watermark: Providing Proof of IP Ownership in Modern SoCs,"*Cryptology ePrint Archive*, 2022.
[6] G. D. Micheli,"Synthesis and Optimization of Digital Circuits," McGrawHill, 1994.
[7] C. J. Liu et al.,"An automatic controller extractor for HDL descriptions at the RTL," *IEEE Design & Test of Computers, Vol. 17, No. 3*, 2000.
[8] G. H. Mealy,"A method for synthesizing sequential circuits," *The Bell System Technical Journal, Vol. 34, No. 5*, pp. 1045-1079, 1955.
[9] E. F. Moore,"Gedanken-experiments on sequential machines," *Automata Studies.(AM-34), Vol. 34*, Princeton University Press, 2016.
[10] https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf, "Secure Hash Standard".
[11] Z. Kohavi and N. K. Jha,"Switching and Finite Automata Theory," Cambridge University Press, 2009.
[12] https://www.allaboutcircuits.com/technical-articles/encoding-the-states-of-a-finite-state-machine-vhdl.
[13] T. Villa et al.,"Synthesis of Finite State Machines : Logic Optimization," Springer, 1997.
[14] T. Villa et al.,"Synthesis of Finite State Machines : Functional Optimization," Springer, 1997.
[15] C. Piguet,"Low-Power CMOS Circuits: Technology, Logic Design and CAD Tools," CRC Press, 2018.
[16] https://opencores.org/, "OpenCores".
[17] https://github.com/freecores, "FreeCores".
[18] https://github.com/secworks, "SecWorks".
[19] F. Lanubile et al.,"Extracting reusable functions by flow graph based program slicing," *IEEE Transactions on Software Engineering, Vol. 23, No. 4*, pp. 246–259, April 1997.
[20] T. Li et al.,"Automatic circuit extractor for HDL description using program slicing," *Journal of Computer Science and Technology, Vol. 19*, pp. 718–728, 2004.
[21] https://github.com/PyHDI/Pyverilog, "Pyverilog Tool".
[22] S. Takamaeda-Yamazaki,"Pyverilog: A python-based hardware design processing toolkit for Verilog HDL," *International Symposium on Applied Reconfigurable Computing*, pp. 451-460, Springer, 2016.
[23] http://iverilog.icarus.com/, "iVerilog Tool".
[24] D. S. Wile,"Abstract syntax from concrete syntax," *Proceedings of the 19th international conference on Software engineering*, pp. 472-480, 1997.
[25] https://github.com/secworks/aes/tree/master/src/rtl, "Secworks AES".
[26] https://github.com/freecores/osdvu/blob/master/uart.v.
[27] "HDL Compiler™ for Verilog User Guide – Version G-2012.06," Synopsys®, June 2012.
[28] F. Vahid,"Digital Design with RTL Design, Verilog and VHDL - 2nd edition," Wiley, March 2010.
[29] https://trilobyte.com/pdf/golson_snug94.pdf.
[30] "Design Compiler® Optimization Reference Manual – Version F-2011.09-SP2," Synopsys®, December 2011.
[31] https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html, "Design Compiler".
[32] https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html, "Quartus Prime".
[33] https://github.com/gremerritt/multicycle-processor/.
[34] https://github.com/freecores/embedded_risc/tree/master/Verilog.
[35] A. Nahiyan et al.,"AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs," *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1-6, 2016.
[36] A. Nahiyan et al.,"Security-Aware FSM Design Flow for Identifying and Mitigating Vulnerabilities to Fault Attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 38, No. 6*, pp. 1003-1016, 2019.
[37] V. S. Rathor et al.,"An energy-efficient trusted FSM design technique to thwart fault injection and trojan attacks," *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, pp. 73-78, IEEE, 2018.
[38] T. Farheen et al.,"Proof of Reverse Engineering Barrier: SEM Image Analysis on Covert Gates," *ISTFA 2021*, pp. 179-189, ASM International.
[39] A. Cui et al.,"A robust FSM watermarking scheme for IP protection of sequential circuit design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 30, No. 5*, pp. 678-690, 2011.
[40] M. Lin et al.,"Watermarking technique for HDL-based IP module protection," *Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP 2007)*, pp. 393-396, 2007.
[41] A. T. Abdel-Hamid et al.,"Finite state machine IP watermarking: A tutorial," *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)*, pp. 457-464, IEEE, 2006.
[42] K. Nguyen et al.,"An FSM-based IP protection technique using added watermarked states," *2013 International Conference on Advanced Technologies for Communications (ATC 2013)*, pp. 718-723, IEEE, 2013.
[43] M. Lewandowski et al.,"A novel method for watermarking sequential circuits," *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 21-24, 2012.
[44] R. Karmakar et al.,"A cellular automata guided finite-state-machine watermarking strategy for IP protection of sequential circuits," *IEEE Transactions on Emerging Topics in Computing*, 2020.
[45] K. Juretus et al.,"Synthesis of hidden state transitions for sequential logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 40, No. 1*, pp. 11-23, 2020.
[46] J. Kuai et al.,"WaLo: Security Primitive Generator for RT-Level Logic Locking and Watermarking," *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 01-06, IEEE, 2020.
[47] A. Saha et al.,"Oracall: An oracle-based attack on cellular automata guided logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 40, No. 12*, pp. 2445-2454, 2021.
[48] M. Fyrbiak et al.,"On the difficulty of FSM-based hardware obfuscation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 293-330, 2018.
[49] C. Pilato et al.,"ASSURE: RTL locking against an untrusted foundry," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 29, No. 7*, pp. 1306-1318, IEEE, 2021.
[50] https://github.com/secworks/sha512, "SecWorks SHA-512".
[51] https://github.com/Arlet/verilog-6502, "MOS MC6502 CPU".
[52] https://github.com/cavnex/mc6809, "Motorola MC6809 CPU".
[53] https://github.com/freecores/ao68000, "Motorola MC68000 CPU".
[54] https://github.com/freecores/gcm-aes, "Galois Counter Mode AES".
[55] https://github.com/freecores/hpdmc, "HP Dynamic Memory Controller".
[56] https://github.com/freecores/genesys_ddr2.
[57] https://bitbucket.org/debjitp/goldminer/src/master/verilog/IBex/, 'IBex'.
[58] https://github.com/freecores/t6507lp, "T2600 SoC"
[59] https://github.com/secworks/modexp, "Modular Exponentiation RSA".
[60] https://github.com/freecores/mem_ctrl, "Memory Controller IP".
[61] https://opencores.org/projects/cxd9731, "PS2 Network Adaptor IC".
[62] https://github.com/Time0o/z80-verilog, "Z80 Compatible CPU".
[63] https://github.com/dwelch67/amber_samples/tree/master/amber25.
[64] https://github.com/olgirard/openmsp430/tree/master/core/rtl/verilog.
[65] https://github.com/ultraembedded/biriscv, "biRISCV SoC".
[66] https://github.com/ryuz/jelly, "Jelly CPU".
[67] https://github.com/openrisc/or1200, "OR1200 RISC Core".
[68] https://github.com/freecores/i2c, "I2C Controller Core".