Reverse Engineering of High-Level Synthesis and its Applications

Thesis submitted to the
Indian Institute of Technology Guwahati
for the award of the degree

of

Doctor of Philosophy

in

Computer Science and Engineering

Submitted by
Mohammed Abderehman

Under the guidance of
Dr.Chandan Karfa

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

June, 2022

Dedicated to

My beloved Parents

Who always picked me up on time
and encouraged me to go on every adventure,
specially this one

Declaration

I, Mohammed Abderehman, certify that:

- The work contained in this thesis is original and has been done by myself and under the general supervision of my supervisor.
- The work reported herein has not been submitted to any other Institute for any degree or diploma.
- Whenever I have used materials (concepts, ideas, text, expressions, data, graphs, diagrams, theoretical analysis, results, etc.) from other sources, I have given due credit by citing them in the text of the thesis and giving their details in the references.
- I also affirm that no part of this thesis can be considered plagiarism to the best of my knowledge and understanding and take complete responsibility if any complaint arises.
- Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Place: IIT Guwahati
Date: 27 June, 2022

(Mohammed Abderehman)

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my supervisor *Dr.Chandan Karfa* for his whole hearted support, valuable guidance, continuous encouragement, and inexhaustible patience, which are of immense help to me in completing this thesis. I am thankful for his ethical beliefs, insightful feedback, and philosophy which made me sharpen my thinking as a researcher. I consider myself extremely lucky for getting the opportunity to work under him. I would also like to thank the other members of my Doctoral Committee - *Dr.Jatindra Kumar Deka*, *Dr.Hemangee K. Kapoor* and *Dr.Purandar Bhaduri* for their insightful comments and suggestions which made me improve the quality and clarity of my work.

I would like to express my sincere thanks and gratitude to the Ethiopia Ministry of defence (MoND) and Defense University, College of Engineering for letting me study my higher studies in India and providing full sponsorship during the entire period of my study. I would also like to express my heartful gratitude to the director, the deans, and other management of IIT Guwahati. I am thankful to all faculty and staff of the Department of Computer Science and Engineering for extending their co-operation in terms of technical and official support for the successful completion of my research work. I would also like to thank Jayprakash Patidar, Jay Oza, Yom Nigam, Rupak Gupta, and Theegala Rakesh Reddy for helping me the implementation part.

Finally, yet importantly, I would like to thank Almighty GOD for being able to complete this thesis with success. I am grateful to my parents for their unconditional love, and support. They are my constant source of energy, motivation, love and support and I know that my achievements give them much joy. I would like to thank my wife Saliha and the whole family for their unconditional love and for making me feel at all times that my education and my dreams are as important as their own.

Certificate

This is to certify that this thesis entitled, **"Reverse Engineering of High-Level Synthesis and its Applications"**, being submitted by **Mohammed Abderehman**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by him under my supervision and guidance. The thesis, in my opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of my knowledge, it has not been submitted elsewhere for the award of the degree.

Place: IIT Guwahati
Date: 27 June 2022

Dr.Chandan Karfa
Associate Professor
Department of Computer Science and Engineering
IIT Guwahati

Abstract

High-level synthesis (HLS) is the process of translating an abstract behavioral specification (usually written in C, C++) into a register transfer level (RTL) that realizes the given behaviour. The HLS is widely used in the semiconductor industries due to advantages like shorter design cycles, efficient design space exploration, and easy writing specifications at a higher abstraction level. In the context of the quick development of hardware accelerators, the use of HLS is also crucial. In this work, we explore if *we can reverse engineer the HLS, i.e., extracting a C code from the HLS generated RTL.* The answer is yes as identified in this thesis. Specifically, we take the advantage of the special structure of the HLS generated RTL which consists of the separate datapath and controller finite state machine and automatically generate a concise, cycle accurate, and debug friendly C code called RTL-C from the RTL. We then show several applications of the RTL-C in the context of verification and security of HLS.

At present, the RTL co-simulation is the primary platform used for HLS design verification. Although most of the state-of-art RTL simulators provide an abstracted user friendly platform for verification, they are undesirably slow and sometimes incomprehensible to non-FPGA experts to debug. In the first application, we show that the RTL-C can be used for faster simulation based verification of the HLS. In this thesis, we introduce an automatic cycle accurate simulation tool *FastSim* for the same. Our simulation tool ensures RTL correctness, provides cycle accuracy, accurate performance estimation and renders on an average around 300 times faster simulation compared to RTL simulators and comparable performance to that of software C simulators. Experiments on various HLS benchmarks demonstrate the efficiency and scalability of our simulation tool.

The formal verification of the HLS is still an open problem and the HLS tools are not bug free. The primary challenge of the formal verification is the abstraction gap between the input C and its corresponding RTL. As a second application, we show that RTL-C is helpful in reducing this abstraction gap. Specifically, we develop a formal verification tool DEEQ for checking equivalence between the C code against the RTL-C. We have taken a data-driven approach to find the correspondence of traces between two behaviours. We also merge compatible traces within a behaviour to reduce the verification complexity. Finally, the equivalence of traces is shown with help of an SMT solver. Experimental results show that our proposed method can prove the end-to-end equivalence for small to medium benchmark designs for a commercial HLS tool.

The variables of a high-level behaviour are mapped to hardware registers during the register allocation (RA) step of HLS. Due to possible many-to-many relations between the variables in C and the registers in the RTL, it is not straightforward to identify this mapping automatically. As a third application, we have shown that the RTL-C can be utilized to identify this mapping automatically. Specifically, we have taken the input C/scheduled C code and RTL-C and we come up with two methods through which we can automatically extract this mapping information. In the first approach, the scheduled C code and the RTL-C are combined state-wise and an invariant generator tool Daikon is used to identify the mapping information. In the second approach, we formulate the mapping problem as a Satisfiability (SAT) problem and use Satisfiability Modulo Theory (SMT) solver to obtain the register to variable mapping information. The frameworks are implemented and tested on a commercial HLS tool for several benchmark designs.

A hardware Trojan (HT) is a malicious modification of the design done by a rogue employee or a malicious foundry to leak secret information, create a backdoor for attackers, alter functionality, degrade performance and even halt the system. Recently, a possibility of HTs - specifically, battery exhaustion attack, degradation attack, and downgrade attack insertion, are shown in a compromised HLS tool. As a fourth application, we utilize our RTL-C to detect HTs inserted by HLS tool. Specifically, we have identified a battery exhaustion attack during generation of RTL-C. The degradation attack and the downgrade attack are detected during the C to RTL-C equivalence checking. The experimental results confirm the detection of HTs of the black-hat HLS tool.

Overall, this thesis proposes an automatic way to extract a C code from the RTL generated by the HLS tool and show various important applications of this reverse engineering process.

Keywords- *High-level Synthesis*, *Cycle accurate simulation*, *Reverse engineering*, *RTL verification*, *C to RTL equivalence checking*, *SMT Solver*, *Hardware Trojan*, *Black-hat HLS.*

Contents

List of Figures

List of Tables

1 Introduction

As system-on-a-chip (SoC) designs complexity increase, designers shift toward system level modeling by facilitating higher abstract design description (e.g., C/ C++) to generate low level descriptions (e.g., Register transfer level (RTL) models) automatically using *high-level synthesis* (HLS). The HLS is the process of translating an abstract behavioral specification (usually in C/C++) into an RTL structure that realizes the given behaviour [98]. The synthesis process passes through several mutually dependent sub-tasks such as preprocessing, scheduling, allocation and binding, and datapath and controller generation [20], [57]. Fig.1.1 shows the flow of a typical high-level synthesis tool [65]. It accepts a high-level design written in high-level languages, constructs an intermediate representation (IR) through its compiler front-end, applies a sequence of transformations like compiler optimizations, scheduling, allocation, binding, and generates targeted RTL code. Compiler optimization includes generic compiler optimizations like dead code elimination, common sub-expression elimination, copy and constant propagation, loop unrolling, etc.. The HLS tools are an attractive choice for the design houses because of their following advantages: (i) Design at a higher abstraction level means a shorter design cycle. (ii) Specification is simpler to write and less error-prone at a higher abstraction level. The user does not need to think about all the details of hardware circuits like clocks and reset/set of sequential elements. In general specification in C/C++ reduces the complexity of design 10x compared to RTL design. (iii)

It is easy to explore design space with HLS. Various RTLs can be generated within seconds from the same C/C++ specification just by tuning certain constraints/optimization parameters during HLS, and (iv) a shorter verification cycle.

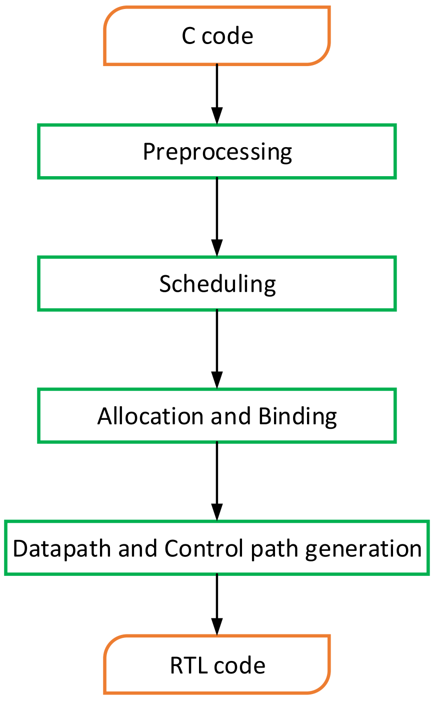


Figure 1.1: *High-level Synthesis flow*

1.1 High-Level Synthesis Flow

Starting from the high-level description of an application and specific design constraints, an HLS tool performs the following tasks:

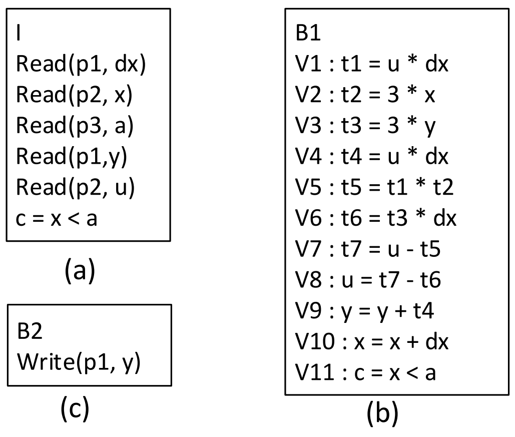
1. Preprocess the input behaviours to generate IR.
2. Schedules the operations to clock cycles.
3. Allocates hardware resources (functional units, storage components, buses, etc.).
4. Binds the operations to functional units, variables to storage elements.
5. Generates datapath and controller finite state machine (FSM) in the RTL.

```
Diffeq : (x, dx, u, a, clock, y)
input: x, dx, u, a, clock;
output: y
while(x < a)
   u1 = u-(3*x*u*dx)-(3*y*dx)
   y1 = y+(u*dx)
   x1 = x+dx
   x = x1; u = u1; y = y1
end
```

Figure 1.2: *Example of 2^{nd} order differential equation solver (DIFFEQ)*

1.1.1 Preprocessing

The HLS process begins with the prepossessing of the input specification. This first step transforms the input specification into an internal representation (IR). Since the input specification is written in higher abstraction for human readability and is not for direct translation into hardware, it is desirable to do some initial optimization of the internal representation. The HLS mostly uses the front-end of most known C/C++ compilers like GCC or LLVM to generate the IR. Since the modern compilers are loaded with various optimizations, HLS took advantage of such optimization to generate an efficient application-specific hardware accelerator. The HLS tool discards the register allocation and machine instruction generation part of LLVM/GCC. Rather it takes the highly optimized IR of LLVM/GCC and generates application-specific hardware accelerators from that.



```
I
Read(p1, dx)
Read(p2, x)
Read(p3, a)
Read(p1,y)
Read(p2, u)
c = x < a
```

(a)

```
B2
Write(p1, y)
```

(c)

```
B1
V1 : t1 = u * dx
V2 : t2 = 3 * x
V3 : t3 = 3 * y
V4 : t4 = u * dx
V5 : t5 = t1 * t2
V6 : t6 = t3 * dx
V7 : t7 = u - t5
V8 : u = t7 - t6
V9 : y = y + t4
V10 : x = x + dx
V11 : c = x < a
```

(b)

Figure 1.3: *Basic Blocks with 3-address codes for DIFFEQ*

The model produced by the compilation process from the input design shows the data and control dependencies between the operations. Data and control dependencies can be represented with the control and data flow graph (CDFG) [103]. A CDFG is a directed graph in which the edges represent the control flow. The nodes in a CDFG are commonly referred to as basic blocks and are defined as a straight-line sequence of statements that contain no branches or internal entrance or exit points. A CDFG exhibits data dependencies inside basic blocks and captures the control flow between those basic blocks.

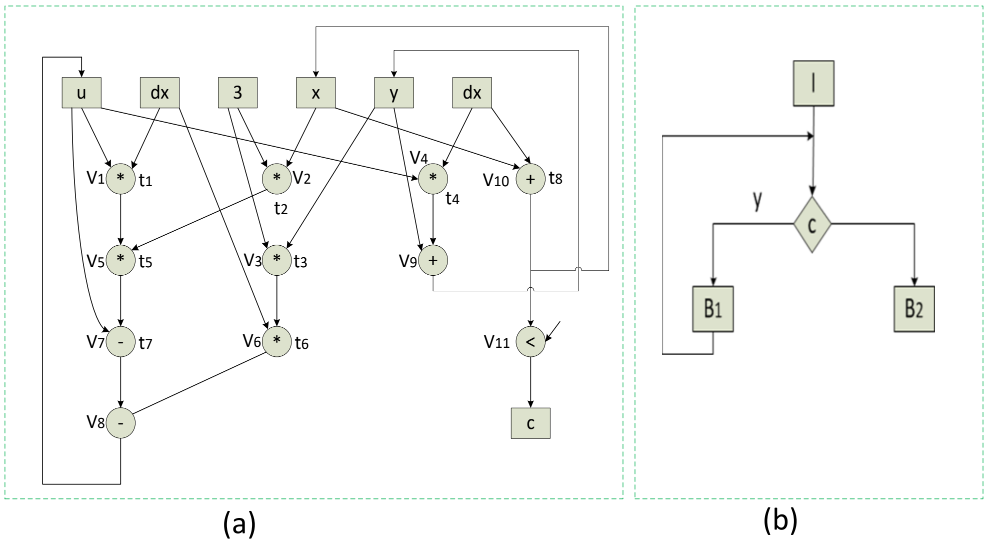


Figure 1.4: *(a) Data dependency graph, (b) Control and Dataflow graph*

Example 1. *Consider the* 2^{nd} *order differential equation solver (DIFFEQ) behaviour as shown in Fig.1.2. The preprocessing steps identify all the basic blocks in the given behaviour and break the operations into three address formats. We have 3 blocks I, B1 and B2, as show in Fig.1.3(a), Fig.1.3(b) and Fig.1.3(c), respectively. In the I block, all the inputs are read from the port and are assigned to temporary variables. In block B1, all operations are converted into three address formats, and in block B2, the final result is written into the output port. This process also analyzes data dependencies among the operations in the block. The data dependencies among the operations within a basic block are usually represented by a data dependency graph (DDG). The DDG of the basic block B1 and the CDFG of DIFFEQ are shown in Fig.1.4(a) and Fig.1.4(b), respectively.* □

1.1.2 Scheduling

Scheduling assigns the operations to so-called control steps without violating data dependencies among operations. A control step is the fundamental sequencing unit in synchronous systems; it corresponds to a clock cycle. For untimed C/C++ designs, this step adds time to the design and determines the time step or the clock cycle in which each operation of the design executes. The aim of scheduling is to minimize the amount of time or the number of control steps needed for completion of the program, given certain limits on the available hardware resources.

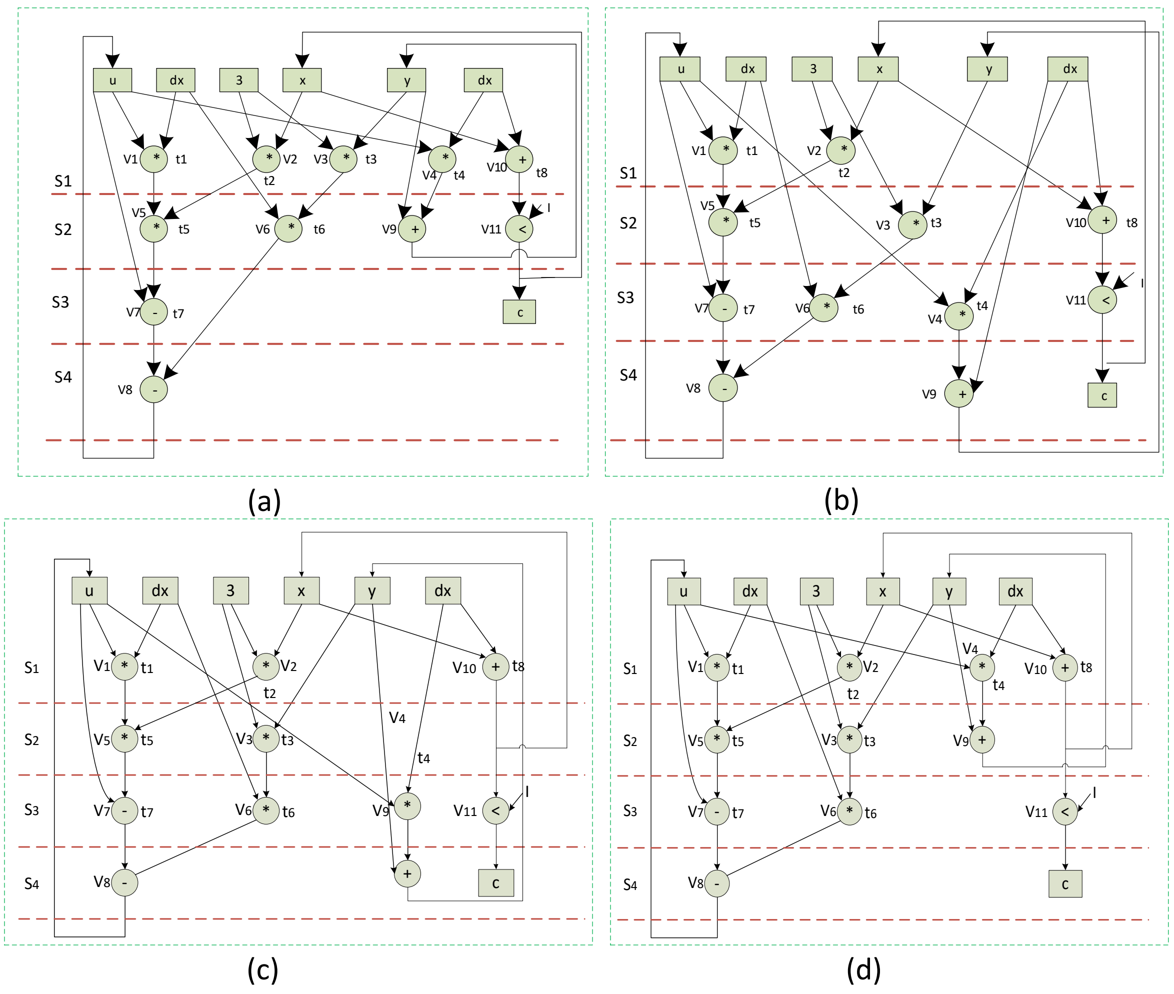


Figure 1.5: *(a) ASAP, (b) ALAP, (c) List, (D) Force directed scheduling of DIFFEQ example*

1.1.2.1 Scheduling Algorithms in HLS

Over the last few decades, many scheduling algorithms for high-level synthesis have been proposed. Scheduling problems can be of four types namely, unconstrained scheduling, time-constrained scheduling, resource-constrained scheduling, and time-resource constrained scheduling [43]. Resource and time constraint scheduling for generic DDG is in general NP-complete problem [93]. Therefore both exact and heuristic based algorithms are proposed for scheduling [123].

Exact algorithms like integer linear programming for scheduling [34], provide the optimal schedule but consume a high amount of processing time. To address the execution time issue, several heuristic based algorithms based on greedy strategies have been developed to achieve fast and near-optimal schedules [118]. Some well-known heuristic algorithms for scheduling in HLS are As Soon As Possible (ASAP), As Late As Possible (ALAP), List Scheduling (LS) and Force Directed Scheduling (FDS) [98]. In the ASAP scheduling algorithm, the operations in the DDG are scheduled step by step from the first control step to the last, i.e., an operation is scheduled if and only if all its predecessors are scheduled in earlier control steps. The ALAP schedules the operations from the last control step toward the first, i.e., an operation is scheduled if and only if all its successors are scheduled in the latter control steps.

In heuristic based algorithms, a priority function (priority list) is used to define the priority of the operations in DDG. The operation with higher priority will be selected in case of conflict. For example in list scheduling [65], the distance of a node from the sink nodes is used as a priority list. Scheduling under resource and timing constraints can be handled by list scheduling. In force directed scheduling [104], the force determines the priority of operations. Forces attract (repel) operations into (from) specific schedule steps. The main policy of this algorithm is to reduce the number of functional units, registers and buses required, by balancing the concurrency of the operations assigned to them, but without lengthening the total execution. In recent times, high-quality schedules for FPGA HLS [116, 119], power-aware [125, 59, 130], dynamically schedule in HLS [76], combining dynamic and static scheduling in HLS [40], time-aware [99, 42, 48], reliability-aware [38], area-optimization [117, 133], security-aware [115], etc. scheduling algorithms have been proposed.

Example 2. *Let us consider the ASAP and ALAP scheduling of the DIFFEQ as shown in*

Fig.1.5(a) and Fig.1.5(b), respectively. In the case of ASAP scheduling, it may be noted that operations V1, V2, V3, V4, and V10 do not have any direct predecessors, i.e., they depend on input values. So these operations are scheduled in step S1. Operations V5 has V1 and V2 as predecessors. So, control_step(V5) = maximum (control_step(V1),control_step(V2))+1=2. All other control_step of the operations can be found out in a similar manner. This schedule is complete within 4 control_steps and four multipliers, one adder, one subtractor, and one comparator resources are required to make it successful. But, in the case of ALAP scheduling, it may be noted that operations V8 and V9 do not have any direct successors. So these operations have the control_step as S4. Operation V7 is the immediate successor of V8, so, control_step(V7) = control_step(V8)-1=3. Similarly, control_step assignment for all operations can be explained. The resource requirements for ALAP scheduling are two multipliers, one adder, one subtractor, and one comparator. Compared to ASAP scheduling, ALAP scheduling requires two less multipliers to make the schedule successful. Also, the example of list scheduling with 2 multipliers (delay equal to 1) and one ALU (delay equal to 1), a latency of 4 units, and force directed scheduling under resource constraints are shown in Fig.1.5(c) and Fig.1.5(d), respectively. □

1.1.3 Allocation and Binding

Allocation determines the selection of the types of hardware components (for instance, functional units and storage) and the number for each type to be included in the final implementation to satisfy the design constraints. Binding assigns operations and variables onto the allocated functional units and storage elements (registers or memory blocks), respectively. The selection of the type and the number of resources during the allocation and binding step is usually formulated as an optimization problem. The main goal is to find the minimum number of hardware resources while fulfilling given area/time constraints. Two operations can be mapped to a FU if they schedule in different time steps. Similarly, two variables can be mapped to a single register if their lifetimes do not overlap. Since binding is a non-deterministic polynomial-time hard (NP-hard) problem, the degree of success of such solutions is restricted. Allocation and binding can be defined as graph problems. They can be formulated either as the problem of finding cliques in a compatibility graph or that of coloring vertices in a conflict graph [98].

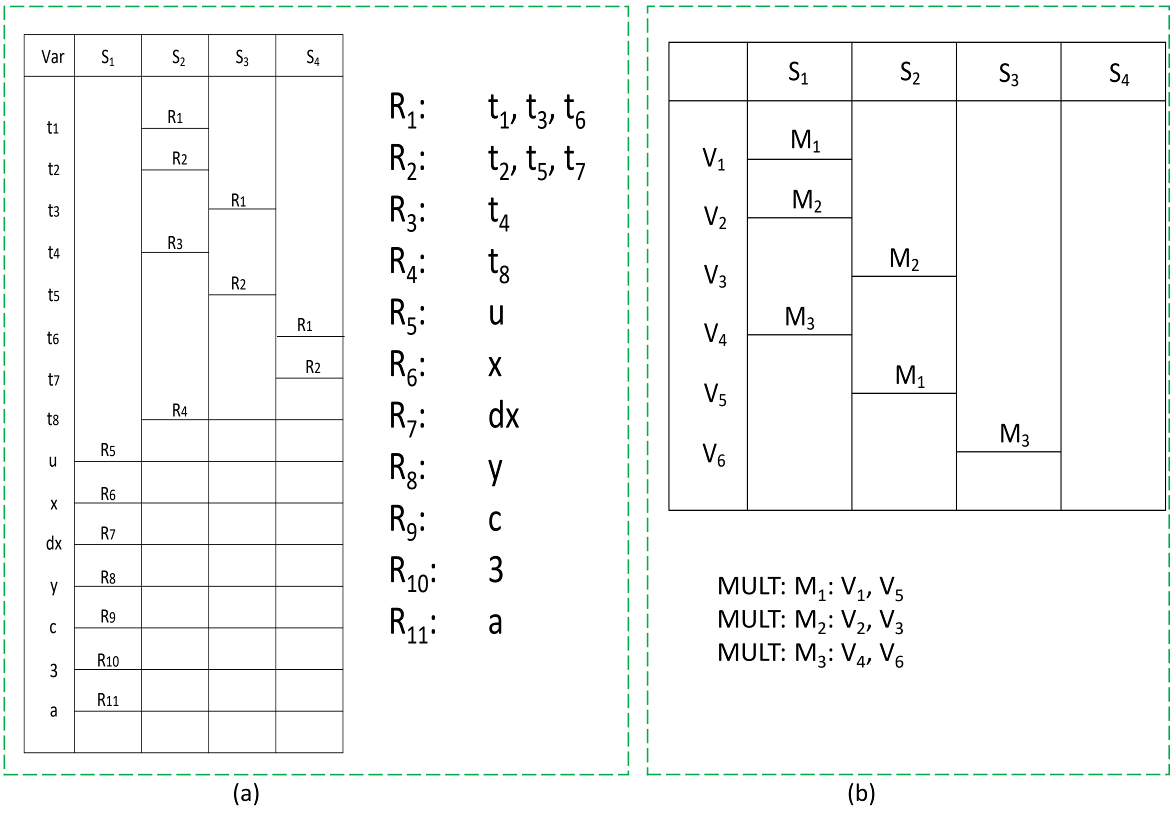


Figure 1.6: *(a) Registers allocation and binding , (b) Multiplier allocation and binding of DIFFEQ example*

Example 3. *Let us consider register and multiplier allocation and binding for the DIFFEQ example for the schedule in Fig.1.5(d) as shown in Fig.1.6(a) and Fig.1.6(b), respectively. In the case of register binding, it may be noted that t1, t3, and t6 are bound to R1. Similarly, t2, t5, and t7 are bound to R2. For all other input and intermediate variables, register binding can be found in a similar manner. In the case of multiplier binding, we need three multipliers. It may be noted that operations V1 and V5 are bound to M1, operations V2 and V3 are bound to M2, and operations V4 and V6 are bound to M3. Multiplexers at inputs of both the multipliers are required. Similarly, adders allocation and binding for all operations can be explained.* □

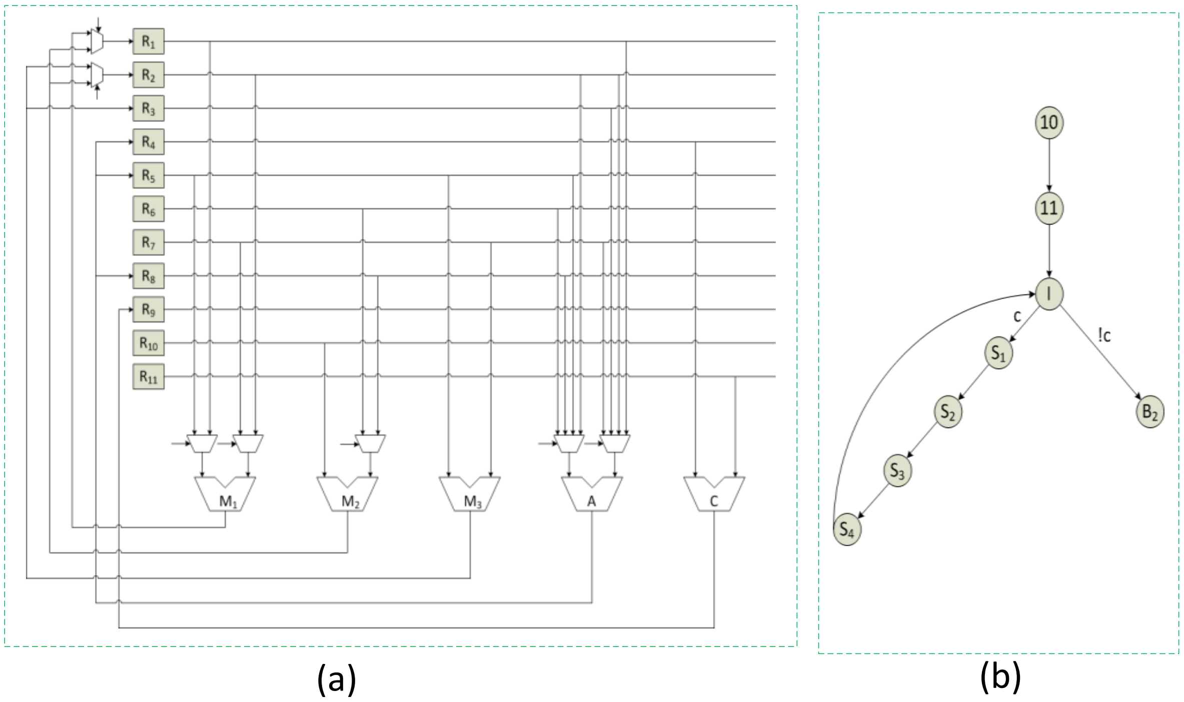


Figure 1.7: *RTL: (a) Datapath unit, (b) Controller unit*

1.1.4 Datapath and Controller Generation

In the datapath and controller generation step, the interconnections among the FUs and registers are determined. More than one data transfer between two units can share an interconnection bus if they are mutually exclusive. The datapath consists of a set of storage elements (such as registers, register files, and memories), a set of functional units (such as ALUs, multipliers, shifters, and other custom functions), and interconnect elements (such as tristate drivers, multiplexers, and buses). All these register-transfer components can be allocated in different quantities and types and connected arbitrarily through buses. Each component can take one or more clock cycles to execute, can be pipelined, and can have input or output registers. Finally, a finite state machine (FSM) is generated for the control circuit to control the execution of operations in the RTL design. This control unit generates control signals that control the flow of data through the datapath. The final RTL structure (i.e., datapath and controller) of the DIFFEQ example for the schedule in Fig.1.5(d) and the allocation and binding in Fig.1.6 is shown in Fig. 1.7.

1.2 Advances in High-Level Synthesis

High-level synthesis has been the subject of continuous research since the 1970s and getting a chance to shine with contributions from many in the field of electronic design automation [97], but these efforts were not notably successful until the early 2000s. In the early 1970s, some research work in high-level synthesis was offered by companies such as Calma and Applicon. In this era, most researchers focused on design specification, simulation, and synthesis at both the RTL and algorithmic levels. Even though the work was groundbreaking research, its acceptance had limited and had very little impact on industrial design. From the 1980s to the early 1990s, many ideas and concepts of the early era of HLS were explored that had a strong impact on HLS development. Also, domain-specific and DSP-oriented research projects were done by some researchers. Although many ideas and concepts were explored, this HLS generation era also failed the commercial due to input languages, quality of results, and domain specialization. The mid-1990s – early 2000s was the period in which major computer-aided design (CAD) companies like Synopsys, Cadence, and Mentor Graphics were present in the market. Finally, the current generation of HLS tools includes those that have been offered by several vendors since the early 2000s. This era of tools focuses on the domain of application algorithms, and system designers with the right input languages, which offers improved performance, quality of results, and user interface, leading to a significant reduction in the design times. For general information (e.g. company, target, input, output, etc.), a classification of the electronic design automation (EDA) tools for HLS presented on Table 1.1. Currently, a large majority of computer vision, data mining, edge compute applications and industrial control algorithms are developed in C/C++ by developers with little or no knowledge of underlying FPGA hardware. Recently, there is significant work in approaches to express the application in a high-level language that is more amenable for diverse applications. In [41], HLS tools with dynamic scheduling, in which an operation's clock cycle is only determined at runtime have been explained. They introduce elastic components like FIFOs to enable a dataflow-style execution, which could potentially be utilized by RapidStream in the future. The paper [53] demonstrates machine learning models to enable fast and accurate resource and timing estimations for HLS designs to effectively and efficiently bridge the accuracy gap. HLS-based FPGA implementation of convolutional deep belief network for signal modulation recognition is presented in [132]. The authors propose a system to optimize deep confidence network (CDBN) by loops pipelining and unroll, memory buffering and partitioning, and implementing an energy-efficient HLS-based FPGA convolutional CDBN accelerator for signal modulation recognition (SMR). Performance predictability is important in a design flow. A methodology to assemble and characterize virtually reconfigurable accelerators based on dataflow and functional programming principles has been proposed in [111]. The proposed method combines a dataflow compiler for generating C-based HLS descriptions from a dataflow description and a C-to-gate synthesizer for generating RTL descriptions. This work also proposes a new interface synthesis approach by using a shared memory that behaves like a circular buffer. The Rathlin Image Processing Language (RIPL) [120] is a stand-alone high-level image processing domain-specific language for developing memory-efficient image processing applications on FPGAs. The work in [74] presents domain-specific architectures (DSAs) for accelerating sparse matrix vector multiplication on FPGAs. In DSA, memory hierarchy and communication are customized to boost performance and energy efficiency and compute logic to suit the needs of the application. Other recent areas of applications on HLS include secure hardware performing logic lock at the RTL [108] [128] and the C level [22], HLS for machine learning application [60] are gaining popularity.

Table 1.1: *Overview of EDA Tools for High-Level Synthesis*

Company	Tool	Year	Target	Input	Output
Binachip, Inc.	Binachip-FPGA	2006	FPGA	C, C++, MATLAB, Java	VHDL/Verilog RTL
Cadence Design Systems, Inc	C to Silicon	2008	FPGA/ ASIC	C/C++/SystemC	Verilog RTL/ Wrap RTL
AutoESL	AutoPilot	2009	FPGA	C/C++/SystemC	Verilog/ SystemC/ VHDL RTL
Synopsys, Inc.	Synphony	2009	FPGA/ ASIC	M-language and Synphony IP blocks	RTL and C
Mentor Graphics Corporation	Catapult C Synthesis	2010	FPGA/ ASIC	C++ or SystemC	RTL netlist
LegUp	U. Toronto	2011	FPGA	C	Verilog RTL
Bambu	PoliMi	2012	FPGA/ ASIC	C	Verilog RTL
Vivado HLS	Xilinx	2013	FPGA/ ASIC	C/C++/SystemC	Verilog/ VHDL/ SystemC

1.3 Motivations and Objectives

In this section, we have identified several motivations for our work.

1.3.1 HLS tool Usage

The HLS has the potential to bridge the knowledge gap in hardware design, pushing design to a system-level requiring only software, or less hardware knowledge. Although HLS comes with huge advantages, the real use of HLS in commercial design houses is still challenging. Who are the target users of the HLS tools - the RTL developers or the algorithm developers? The RTL developers are reluctant to use HLS because the RTL generated by an HLS tool is not as optimal as compared to the RTL they develop manually in many scenarios. On the other hand, the algorithm developers do not understand the RTL well. The algorithm developers pose questions like how the behaviour is implemented in hardware, what is the problem in case of errors, what is the impact of specific optimization, how to write hardware friendly C/C++ specifications, etc. The quality of the RTL generated by the HLS tools greatly depends on the way one has written the C/C++ code. Moreover, they do not understand the real effects of all the hardware related optimization parameters of the HLS tools. As a result, it is difficult for them to write HLS friendly C/C++ code and also choose the right set of optimization parameters during HLS to generate an RTL that meets the target design constraints. *A C equivalent of the RTL hardware would be helpful for the algorithm developers to understand/analyze the output RTL of an HLS tool and hence use the HLS tool meaningfully.*

1.3.2 Slower RTL Simulation

Since the input high-level behaviour goes through several complex transformations before being translated into RTL in HLS, ensuring the correctness of the HLS translation is important for the wide adoption of the HLS tool in the VLSI Design flow. Due to the lack of formal verification support, the HLS designers still rely on low level RTL simulators (e.g. Xsim, Modelsim, or VCS) or on-board emulators (e.g., Zebu) for verification. The RTL simulators incur undesirable time overhead and the hardware detailed cycle accurate simulation of the RTL are not intelligible to non-FPGA experts. The state-of-the-art commercial HLS tools provide software-based simulators. For instance, in Xilinx Vivado HLS [10], there are two simulation frameworks for behavioural and functional design verification [110]: (i) *Pre-synthesis (C-simulation)*: where a C test-bench is used to validate the functional correctness of the algorithm against the behavioural specifications before synthesis. (ii) *Post-synthesis (RTL co-simulation)*: The same C test-bench is used to verify the functional correctness of the synthesized RTL by co-simulating the behavioural model/golden output against a cycle-accurate simulation of the RTL. Although both C simulation, as well as RTL co-simulation environments, are made user friendly, the RTL co-simulation incurs excessive time overhead over C-simulation. The RTL co-simulation is the primary way to verify the correctness of the generated RTL of an HLS tool. Even if RTL simulation relatively accelerates the process of verification and is good at quickly finding errors, it cannot guarantee the complete correctness of generated RTL. The C-simulation is faster than the RTL simulation. *Therefore, generating an equivalent C code from the RTL would greatly reduce the verification time of the HLS.*

1.3.3 Lack of Formal Verification Support

Recent years have seen VLSI systems become more and more complex resulting from progress in VLSI technology as well as growing demands on performance imposed by modern applications. Such complexities, in addition to severe time-to-market requirements, make it challenging to develop reliable, high-quality systems through Register Transfer Level (RTL) implementations. This underlines the need for modeling, synthesis, and validation of hardware at higher levels of abstraction. Since the HLS tool passes a very complex translation process, its correctness becomes a major barrier to its wide adaptation. The tools are not free from logical and implementation errors [72]. However, the verification of the synthesized model is still primarily carried out by time-consuming RTL simulations. Even after rigorous testing, bugs in HLS tools may be undetected. The impact of bugs in HLS tools includes economic damage and serious consequence in safety-critical applications. Therefore, the detection of bugs and the correctness of HLS has always been an important concern. Although phase-wise translation validation of HLS, such as scheduling verification [46], allocation and binding verification [82], and datapath and controller verification [79], are mostly explored by the researchers, a monolithic end-to-end formal verification of HLS is not yet available due to the large semantic gap between the input C/C++ and the generated RTL. It is difficult to correlate the RTL with input C. *Therefore, to developing an end-to-end formal verification method to ensure the correctness of the HLS generated RTL with respect to its input C code behavioural is a critical issue to be considered to bring behavioural synthesis into practice.*

1.3.4 Threat of Hardware Trojan in HLS

The enormous complexity of ICs necessitates the use of highly-specialized foundries that cost beyond 10 billion. As a result, most semiconductor companies operate fab-less and they outsource IC fabrication and other services to third-party vendors. Due to the global distribution of ICs manufacturing foundries, ICs come from different manufacturers. As a result, ICs security has become a new concern in the system design, regarding potential malicious modification of the ICs during the fabrication process. Such modification of ICs referred to as Hardware Trojan (HT). A hardware Trojan is a malicious modification of the design done by a rogue employee or a malicious foundry to leak secret information, create a backdoor for attackers, alter functionality, degrade performance, halt the system, etc. These hardware Trojans are usually triggered under extremely rare input sequences. As a result, they are very hard to detect by usual simulation-based verification. One approach to detect HTs is a side-channel based HT detection mechanism [17]. In this approach principle component analysis (PCA) is used as a side-channel fingerprint of the circuit to compare it with the golden model. However, the characteristics of the physical design can be modified by other factors and not only by HT. A recent study [106], [26] shows the possibilities of inserting Trojans by the HLS tool itself. Specifically, Black-hat high-level synthesis [106] automatically inserts these known HTs: battery exhaustion, degradation, and downgrade attacks in the HLS generated RTLs. *Therefore, addressing HTs detection during HLS is an important approach to designing and integrating solutions at higher levels of abstraction.*

1.3.5 Objectives

With the above motivations, the following objectives were identified:

1. Develop an efficient RTL to C reverse engineering framework to abstract a C code from an RTL generated by the HLS tool. The reverse engineering framework should

be able to recover the register to variable mapping as well as for efficient correlation between C and RTLs.

2. Develop a fast simulation framework for HLS based on the generated C from the RTL.

3. Develop an equivalence checking framework for end-to-end verification of HLS. Specifically, the proposed method will formally prove the equivalence between the input C and the generated C from the RTL.

4. Develop a formal HLS Hardware Trojan detection framework.

1.4 Contributions

In the following, we outline the contributions of this thesis on each of the objectives identified above. An overview of the contributions of the Thesis is depicted in Fig. 1.8.

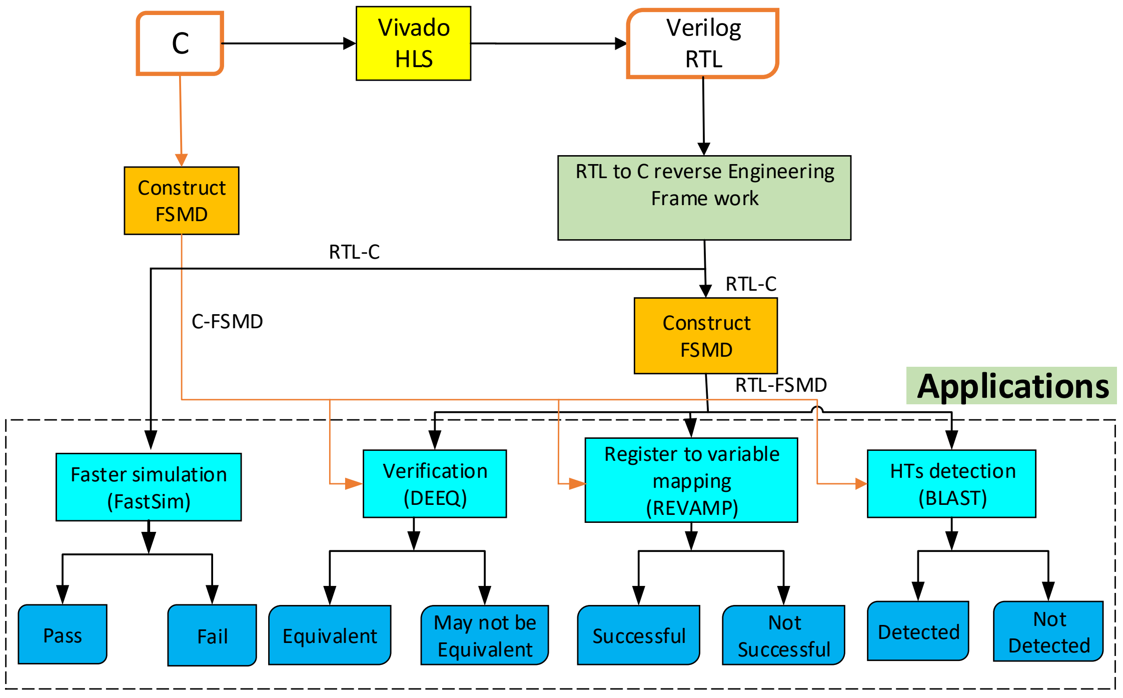


Figure 1.8: *Contributions of the Thesis*

1.4.1 FastSim: A Fast Simulation Framework for High-Level Synthesis

This work is primarily motivated by the fact that C-simulation is much faster compared to RTL-simulation. A fast simulation framework *FastSim* is developed in this work that manipulates certain unique features of HLS design to extract a concise, well-indented, and debug friendly C behaviour from the synthesized RTL by the HLS tool. Specifically, we present a completely automated, fast, and cycle accurate simulation-based verification framework for HLS generated RTLs. The framework ensures the end-to-end correctness of HLS. It is also equipped to give accurate design performance estimation. In our approach, the Verilog RTL is first converted into an abstract syntax tree (AST) format using the PyVerilog [9]. The AST is then pre-processed, i.e, the C incompatible constructs of Verilog like bit-select and part-select are replaced with equivalent compatible representations. Next, we identify the register transfer operations in the datapath concerning the control signals at each control step of the controller FSM to generate an equivalent finite-state machine with datapaths (FSMD) code. To generate the equivalent C code, the intermediate FSMD and the variables, controller state, RT operations, etc. are mapped into appropriate data types, and the equivalent C code called RTL-C is generated. Finally, the input C source code and the RTL-C code are co-simulated using C compilers like GCC or using C-simulation of HLS tools with test cases to verify the functional correctness of the generated RTL. The correctness and cycle accuracy of our tool has been proved. We then compare the simulation time of our tool with the RTL simulator, software C simulator, and the Verilator. Experimental results demonstrate that FastSim is on average around 300 times faster simulation compared to RTL simulators and comparable performance to that of software C simulators.

1.4.2 DEEQ: Data-driven End-to-End Equivalence Checking of High-Level Synthesis

This work contributes a C to the RTL equivalence checking method to prove the correctness of the HLS generated RTL with respect to its original C code. In our method, we have utilized the C-like behaviour, called RTL-C extracted from the RTL in our first work. Our method identifies all traces of both behaviours C and RTL-C, we identify the compatible traces (i.e., traces which have the same outputs) within behaviour and merged them into one. This will improve performance by reducing many-to-many equivalences of traces into one-to-one trace equivalence. The method then finds the corresponding traces between two behaviours using a data-driven approach. Finally, a satisfiability problem is formulated to prove the equivalence of corresponding traces of both behaviours. The proposed method does not take any internal information from the HLS tool. Primarily novelty of our method: (i) use of RTL-C to make C to RTL-C equivalence checking feasible (ii) use data-driven approach to find the correspondence of traces between two behaviours (iii) finding and merging compatible traces to reduce complexity. Experimental results show that our proposed method can prove the end-to-end equivalence for a commercial HLS tool for several benchmark examples.

1.4.3 REVAMP: Reverse Engineering Register to Variable Mapping in High-Level Synthesis

The primary motivation of this work is to extract the register to variable mapping in HLS for efficient correlation between input C/C++ and the corresponding HLS generated RTL. Specifically, we have developed two register to variable reverse engineering frameworks: one based on the invariant generator Daikon [4] and the other one using Satisfiability Modulo Theory (SMT) solver based tool Z3. In the Daikon based framework, the goal is achieved in three steps. In the first step, we obtain the scheduled C code (SD-C) and high-level behaviour (RTL-C) from the scheduling information generated by the HLS tool and the output RTL, respectively. In the second step, we use the invariant generator tool Daikon to find invariants at each state in a program. Since Daikon finds invariants in a program, we combine state-wise SD-C and RTL-C. From the outputs of Daikon, we extract the invariants in which there is an equality relation between a variable of SD-C and a register of RTL-C. Finally, with this mapping information, we can rewrite the RTL-C in terms of variables and finally generates an equivalent C-code from the RTL. In SMT based approach, both input C and the output RTL-C are modeled as FSMDs. Next, both FSMDs are converted into SSA form. A Satisfiability Modulo Theory (SMT) problem is then formulated to obtain the register to variable mapping. Finally, the RTL-FSMD is rewritten with this mapping information to obtain an equivalent C code from the RTL. Our frameworks can be utilized by the algorithm developers to use HLS tools efficiently. It also can be used for efficient debugging of non-equivalent scenario that arises in FastSim. The framework is implemented and tested on a commercial HLS tool for several benchmark designs.

1.4.4 BLAST: Belling the Black-Hat High-Level Synthesis Tool

We propose a formal detection framework for HLS tool inserted hardware Trojans. In this work, we show how battery exhaustion, degradation, and downgrade attacks can be detected using our C to RTL equivalence checking framework. We have utilized two of the previous works RTL to C reverse engineering method and the equivalence checking framework in this work to develop the HT detection framework. We have assumed that both the input C code and the Trojan infected RTL code are available for our analysis. Specifically, our method extracts an RTL finite-state machine with datapaths (FSMD) from the HLS generated RTL. During FSMD construction, a battery exhaustion attack can be identified. Our proposed method then compares the FSMD of the input C code with the FSMD of the RTL to identify the degradation attack and the downgrade attack. The experimental results confirm the detection of HTs of the black hat HLS tool [106].

1.5 Organization of the Thesis

The rest of the thesis is organized in the following manner.
Chapter 2 provides a detailed literature survey on existing state-of-the-art on RTL to C translation and verification of HLS methods. It also presents a survey on Hardware Trojan (HT) attacks and detection mechanisms. In the process, it identifies the limitations of the state-of-the-art verification methods and HTs and underlines the objectives of the thesis.
Chapter 3 presents a completely automated, fast and cycle accurate simulation-based verification framework *FastSim* for HLS generated RTLs. The chapter provides a detailed process (overall flow) of the FastSim framework, and models for various hardware parallelisms like loop and task-level pipelines. A detailed experimental comparison of our simulation framework for RTL generated by the Vivado HLS tool with state-of-the-art simulators is also demonstrated.
Chapter 4 presents a C to RTL equivalence checking framework DEEQ for HLS verification after identifying the limitation of existing formal verification and end-to-end verification of HLS. The chapter discusses how to merge compatible traces within a behaviour, and find corresponding traces between two behaviours using a data-driven approach and the overall equivalence checking method. The soundness, termination, and complexity of our method are also discussed. The chapter also provides experimental results for several HLS benchmarks.

Chapter 5 discusses Daikon and SMT based framework for reverse engineering of the register to variable mapping in high-level synthesis. The contribution, the overall approach of the Daikon based register to variable mapping, and the challenges resolved are discussed first. Then, the overall approach for SMT based register to variable reverse engineering and its applications are discussed in detail. The chapter finally provides experimental results for several benchmark designs.
Chapter 6 presents how the three hardware Trojan attacks can be detected using our C to RTL equivalence checking framework called BLAST. The attack models and detection mechanisms of all three HTs are discussed in detail. The chapter then provides the experimental results that confirm the detection of HTs of the black hat HLS tool.
Chapter 7 concludes the thesis and discusses potential future research directions of this work.

2 Literature Survey

In this Chapter, we overview some important research contributions in the area of RTL to C/C++ reverse engineering, simulation-based verification of HLS, and formal verification of HLS. Also, some techniques for detecting HT at different stages of design flow have been presented in this chapter. The objective of this study is to identify the prominent gaps in the existing simulation-based and formal-based verification methodologies of HLS, and also in the HTs detection mechanisms which have been addressed in this thesis.

2.1 RTL to C/C++ Reverse Engineering

The HLS process starts with a behavioural specification and builds a hardware implementation from it. Understanding the detailed implementation and functionality of a design in the form of hardware description languages like Verilog is not an easy task for algorithm developers who intended to use HLS. The C equivalent of the RTL code for circuit design would be helpful for the algorithm developers to understand the design structure, and the impact of certain HLS optimizations, analyze the output of the design and hence use the HLS tool meaningfully. Reverse engineering an equivalent C code from the RTL would greatly reduce the verification time of the design as well.

The current works in RTL to C/C++ reverse engineering of HLS can be separated into two classes.

1. Generic RTL to C/C++ to enable design space exploration through HLS.

2. Generic RTL to C/C++ for fast simulation and verification of the hardware at the C level.

2.1.1 Generic RTL to C/C++ to enable Design Space Exploration through HLS

There are works that convert generic Verilog models into C code for design space exploration (DSE) through HLS. Bombieri et al. [31] present a method to abstract RTL IP blocks into C++ code by abstracting away most of its architectural characteristics while maintaining its functionality. The goal is to recover the IP block functionality for system-level design and enable the derivation of optimized implementations through HLS. In this paper, the methodology relies on three concepts to generate the equivalent C++ code from the RTL: (i) the scheduling of RTL statements is resolved statically at compile time during the generation of equivalent C++ code, (ii) the static variables in the generated C++ code are synthesized into registers at the gate level, and (iii) to generate loops in the C++ code, the methodology performs loop-rolling transformations on both internal logic and the I/O interface.

Mahapatra [95] presented a method VeriIntel2C to convert RTL behaviours written in Verilog into C code. This technique is carried out by the use of extended Hardware Petri Nets to extract the functionality of the RTL designs and generate a CDFG that captures the different structural forms present in RTL designs. The proposed method currently focuses on generating C designs with explorable constructs only for single RTL modules. The RTL to C conversion process is established in two phases: In phase one, the RTL code is converted into a Control Data Flow Graph (CDFG) and in the second phase, the CDFG is analyzed to generate the aforementioned loops and arrays. In this paper, the authors introduce rule-based search and graph matching techniques to identify unrolled, partially unrolled loops, nested loops, and arrays in the forms of registers, wires, and memories and generate the C program with explorable constructs optimized for DSE.

The work in [92] proposed a performance prediction model based on machine learning algorithms to simplify and speed up the design space exploration (DSE) process of general purpose processors (GPPs) with reconfigurable hardware accelerators (RAs) by quickly estimating the performance of applications running on previously untested architectural configurations. The method presented in this paper also investigates different algorithms by comparing their error prediction rate to measure the accuracy of the prediction.

2.1.2 Generic RTL to C Conversion for Fast Simulation

The existing works in RTL to C conversion for fast simulation of HLS can be classified into two sub-classes: works that guarantee cycle accuracy and those that don't. The LegUP HLS simulator [37] and HLS Scope+ simulator [45] do not guarantee cycle accurate simulation. These works meet specific targets like performance and speedup prediction using synthesis information. Since the LegUp debugging platform is still under development, it does not support break-points, enabling the debugging of hybrid processor/accelerator applications and on-chip hardware debugging. Works that guarantee cycle accuracy can be further classified into automatic and manual simulators. Manual cycle accurate simulators like [101, 47, 114] would require explicit incorporation of scheduling information at the source level. This poses a tedious task for non-hardware experts and might not serve the purpose of hardware-software abstraction guaranteed by HLS tools. Finally, we have the automatic cycle accurate simulation-based verification frameworks [11, 44, 94]. Mahapatra et al.[94] presented a technique to use parsed RTL code by abstracting out the core computation of the HLS generated RTL while maintaining IO timings of iterative segments for fast performance estimation. This technique will not work for data-dependent loop bounds. The Inspect [35] is another cycle accurate simulation framework integrated with LegUP HLS [37] that correlates C source with its LLVM IR representation and synthesized Verilog RTL using an intermediate debug database automatically generated during the LegUP backend process. This framework is useful to extract the exact source of error once a bug is identified. Fezzardi et. al [63] proposed a trace-based debugging solution for verification of HLS designs by collecting hardware and software traces from intermediate source representation in HLS and performing an automatic discrepancy analysis. This technique could be used to automatically track design bugs and locate their source in the input C code.

In [66], a methodology called VTOC is proposed to convert synthesizable Verilog into C. VTOC takes a synthesizable Verilog module as an input and generates a semantically equivalent ANSI C code section. Parts of the Verilog language such as uncertain values, temporary bus fights, Verilog meta-language commands, and other simulation-level operations which are not normally compilable to hardware are not compiled to C by the VTOC compiler. The VTOC can handle either a single module or a hierarchy where a top-level module includes instances of sub-modules. Where a hierarchy of modules is used, the lowest, leaf modules are implemented only in RTL or behavioural Verilog. VTOC generates a single ANSI C output file per compilation. A report file is also generated. The methodology in [100] translates the RTL Verilog behaviour into C code. The objective is to use the C code for hardware property verification, co-verification to simulation, and equivalence checking.

The Verilator simulator [11] is designed to parse generic Verilog HDL into equivalent behavioural descriptions in C++ for fast simulation. Although over the years, Verilator has been incorporated with advanced optimizations to speedup simulations, it disregards the inherent FSMD framework of the HLS based RTLs. Consequently, the generated C++ code is complex in terms of comprehension of code behaviour and incurs performance hampering redundancies. This impacts both simulation performance as well as debugging. R2C [31] is yet another similar framework that converts generic RTL IPs into compact C++ code with the primary intention being to design space exploration using HLS. The generated C++ code could be used to verify designs but suffers from the same shortcomings as Verilator. Recent work is FLASH [44] which uses the input C source and incorporates the scheduling information to guarantee cycle accuracy. FLASH could overcome several shortcomings of software C simulation frameworks like data ordering problems, artificial deadlock scenarios, and feedback problems that are mostly related to FIFO transactions, and also provide a very fast simulation framework. However, it cannot detect any bug that might occur during allocation, bindings, and datapath and controller generation phases of HLS. Another limitation of FLASH is that it cannot simulate stalls from external memory access.

As discussed above, most of the existing works fail to handle the inherent FSMD framework of the HLS based RTLs and fail to detect any bug that might occur during allocation, bindings, and datapath and controller generation phases of HLS, and simulate stalls from external memory access. As a result, they degrade the overall simulation time and performance. In this thesis, we propose an automatic cycle accurate simulation tool, FastSim, that manipulates certain unique features of HLS design to extract a concise, well-indented, and debug friendly C behaviour from the synthesized RTL. Our simulation tool ensures RTL correctness, provides cycle accuracy, and accurate performance estimation. Since our simulation framework uses a C simulation source directly extracted from the RTL, it addresses all the hardware issues including faults during resource allocation as well as pipeline stalls. In this thesis, we also propose a reverse engineering register to variable mapping frameworks to reverse engineer an equivalent C code from the HLS generated RTL.

2.2 Verification of High-level Synthesis

The correctness of a design circuit is a major consideration in the design of digital systems. Recent years have seen VLSI systems become more and more complex, resulting from progress in VLSI technology as well as growing demands on performance imposed by modern applications. Such complexities, in addition to severe time-to-market requirements, make it challenging to develop reliable, high-quality systems through RTL implementations. This underlines the need for modeling, synthesis, and validation of hardware at higher levels of abstraction. There has been a lot of research in the field of equivalence checking of two programs. Considering the equivalence of RTL code generated from C like behavioural code, the task of equivalence checking is a little tricky. The existing literature on the verification of HLS can be primarily classified into two categories: simulation-based verification and formal verification. Simulation based verification of HLS is discussed in detail in subsection 2.1.2.

2.2.1 Formal Verification of HLS

The input C/C++ code of the HLS goes through various transformation phases like preprocessing, scheduling, allocation, binding, and datapath and controller generations before being converted into RTL descriptions. Most of the existing formal verification works target verification of a specific phase of HLS. Formal verification aimed to check that each translation performed by the HLS tool preserves the functionalities of the input behavior. The existing works on formal verification of HLS can be classified into two categories: (i) Phase-wise verification of HLS and (ii) End-to-end verification of HLS.

2.2.1.1 Phase wise Verification of HLS

The large semantic gap between the input behaviour and the RTL design is the primary challenge for end-to-end verification of HLS. Therefore, the phase-wise verification technique which can handle the difficulties of each synthesis sub-task separately is targeted by many researchers for HLS verification. Primarily, the verification of the scheduling phase is explained by researchers. Few works also target the verification of allocation and binding, and the datapath and controller generation phase of HLS. In this subsection, we briefly discuss the notable works on phase-wise verification of HLS. We can further classify the existing works on phase-wise verification into two categories: (i) Front-end verification of HLS and (ii) Back-end verification of HLS.

Front-end verification of HLS:

There has been research work to validate the correctness of compiler transformations and the scheduling phase in HLS. In [87], a translation validation approach is presented to validate the initial high-level description against the result of the scheduling behaviour of the SPARK HLS tool [67]. The method presented in this paper uses a bisimulation relation approach to automatically validate the implementation against its initial high-level specification before and after the optimization carried out by the SPARK tool. The method in [89] presents an efficient approach by redefining the bisimulation relation to validate the result of HLS against the initial behaviour using the translation validation technique and can reduce the number of automatic theorem prover (ATP) queries as compared with the method presented in [87]. Even if the methods presented in [87, 89] are suitable for structure-preserving transformations, they may not be applicable in the case of path-based scheduling since the structure may not be preserved in such case.

The work in [126] proposed a scalable verification of compiler transformations applied in the pre-processing phase based on symbolic simulation together with identification and inductive verification of loop structures. The method uses a dual-rail symbolic simulation of the input and transformed output to explore the paths. The method also uses a compositional strategy to handle the path explosion and an inductive assertions technique for handling loops. The Inspect [90] is another novel equivalence checking approach in high-level synthesis based on translation validation, cut-point, and shared-value graph (SVG) techniques. In this method, the SVG technique has been used to deal with various language structures in a unified way and avoids the “may-not-terminating” problem of existing methods.

The paper in [86] presented the formal verification of scheduling during HLS. In this method, FSMD models are used to represent the behavioral specification and the result of scheduling. The method also introduced cutpoints to construct the path cover for each FSMD. The functional equivalence between both behaviours is proved based on the equivalence of one path cover with some path of the other path cover. However, the method presented in this paper assumes that the path structure of the input behaviour is not modified during the synthesis process.

In [81], a path-based equivalence checking is proposed for the scheduling phase of HLS. In this work, a finite state machine with a datapath (FSMD) model has been used to represent both the given behavioral specification before scheduling and the one produced by the scheduler. The method consists of introducing cutpoints in one FSMD (the given behavioral before scheduling) to obtain initial path cover, visualizing its computations as a concatenation of paths from cutpoints to cutpoints, and identifying equivalent finite path segments in the other FSMD (produced by the scheduler). The method is applicable even when the scheduler changes the basic structure of the given behaviour and works only for uniform code motion techniques. The paper [33] improved the method presented in [81] to handle both uniform and nonuniform code motions applied during the scheduling phase of HLS. In this work for nonuniform code motions, some data-flow properties for equivalent path segments model checking are identified. This work [33] is further extended by in [24] to handle code motion across loops. In order to discover the mismatch in the values of some live variables, the method consists in propagating the variable values over a path to the subsequent paths. Propagating the variable values over the subsequent paths will continue until the values match or the final path segments are accounted for without finding a match. The method is also capable of handling control structure modifications along with uniform and nonuniform code motions.

In [127], a scalable equivalence checking algorithm for validating scheduling transformations has been presented. The method in [127] accounts for control/data dependency and addressed various timing modes of scheduling such as cycle-fixed mode, superstate-fixed mode, and free-Floating mode. However, the techniques presented in [126, 127] can not compare transformations that modify structures of loops through domain-specific optimizations. As a result detecting corresponding variables between the two behaviours will not succeed, causing equivalence checking to fail.

Recently, Chouksey and Karfa [46] presented an equivalence checking method for verifying of scheduling of conditional behaviours along with a new cutpoint selection scheme to overcome significant control structure modification that occurs in the scheduling of conditional behaviors. In this method, some scenarios involving path merge/split where the path-based equivalence checking approaches [81, 24] fail to show the equivalence even though behaviors that are equivalent are addressed.

Back-end verification of HLS:

Other works related to verification of allocation and binding phase, and datapath and controller phase are presented here. Ashar et al. [21] proposed a complete procedure for verifying register-transfer logic against its scheduled behaviour in a high-level synthesis environment. In this paper, the algorithm is based on the observation that the state space explosion in most designs is caused by the datapath registers rather than the number of control states. In [21], the authors are able to divide the equivalence checking task into the checking of (i) local properties which are checked on a per control state basis, and (ii) non-local properties which require a traversal of the control state space. The paper performs equivalence checking between the schedule specification and the RTL implementation of designs by model checking.

The work in [96] reported a formal methodology for verifying a synthesized register-transfer-level design based on symbolic analysis and higher-order logic theorem proving techniques. In this work, various register allocation/optimization schemes commonly found in HLS tools are accommodated. The controller extended finite state machine (EFSM) models the design implementation (the scheduled one and the RTL). In this method, the equivalence between the scheduled and the RTL behaviours has been shown based on the critical states, critical variables, and critical paths of two EFSMs. However, one may encounter an infinite number of paths from the initial state if loops are presented in the behaviour while showing the equivalence between two critical states.

Karfa et al. [82] develop techniques for verifying the correctness of register sharing using a formal method. This framework models the design (behaviour) before and after the datapath synthesis as FSMDs and the equivalence is shown for the mapping by comparing the computations for their FSMDs. Finally, the verification is done by using path covers and cut-point based equivalence checking. For that, the state-wise equivalence of all the paths forming the path covers is proved. The technique presented in this work is also applicable for both data-intensive and control-intensive input specifications.

A formal verification method of the datapath and controller generation phase of a high-level synthesis process is presented in [80]. This paper achieved the verification goal in two steps: (1) the datapath interconnection and Controller FSM are analyzed using a given control assertion pattern in each control step to identify the RT operations are executed in the datapath. It uses a state-based equivalence checking methodology to verify the correctness of the controller behavior. A rewriting method has been developed for this step

(2) an equivalence checking method is deployed to establish equivalence between the input behaviour and the output behaviour of this phase.

As discussed above, most of the existing works target the verification of the scheduling in HLS since the verification of this phase is the most challenging among all phases of HLS. These methods need intermediate information from the HLS tools. However, the availability of such information is limiting the use of phase-wise verification of HLS.

2.2.1.2 End to end Verification of HLS

There are few works that target end-to-end verification to prove the equivalence between the input (C, C++) program and the RTL output of HLS.

Leung et al. [88] proposed a translation validation technique for C to Verilog by converting them into a common intermediate representation (IR) and then used bisimulation techniques to prove the two resulting IR programs are equivalent. In this method, the authors use Daikon [4] to detect the likely invariants at cutpoints. This method, however, won't work if the number of traces is not the same in both the bahaviours and the invariants are not sufficient to prove post-conditions.

The work in [69] proposed a sequential equivalence checking (SEC) framework for behavioral synthesis. The paper presented a suite of optimizations for equivalence checking between electronic system level (ESL) specification and RTL generated through behavioral synthesis. This method cannot handle the certification of designs between the abstract clocked control/data flow graph (CCDFG) and the corresponding RTL. This work is further extended by in [129] to the presence of optimizations that violate local equivalences of internal signals. In this method, two key optimizations (operation gating and global variables) that complicate equivalence checking for high-level synthesis are identified. Subsequently, they developed a method for equivalence checking between system-level design and HLS generated RTL in the presence of these optimizations.

In [100], the authors developed a tool called v2c that translates synthesizable Verilog to C. The tool accepts synthesizable Verilog as input and generates a word-level C program as an output. Equivalence checking is then achievable on the C level with the help of either static analyzing tools or dynamic execution tools. We found that v2c generates incorrect C code from the Verilog generated by the Vivado HLS tool. Therefore, using v2c for HLS functional verification is not a natural solution.

X. Feng and A. Hu [61] presented a way to introduce cutpoints early during the analysis of the software model for checking the equivalence of high-level software against RTL of combinational components. In this paper, the algorithm they developed compares high-level behaviour given as a control-flow graph (CFG) format with the RTL code. Their method has used symbolic execution and satisfiability solving to check equivalence between the two expressions. This paper only focused on combinational equivalence checking and did not address how to extend the proposed method for sequential equivalence checking.

The semantic gap between the input C/C++ behaviour and corresponding RTL is too high to compare directly. Therefore, most of these approaches assume various simplifications regarding transformations during HLS. Therefore, an end-to-end equivalence checker that can handle the complexities of modern-day HLS tools is still not available. In this thesis, we propose a C to RTL equivalence checking method, called DEEQ, to verify the correctness of the HLS generated RTL with respect to its input C code. The proposed method is completely automated and does not take any internal information from the HLS tool.

2.3 Hardware Trojan Detection

Due to the global distribution of ICs manufacturing foundries, ICs come from different manufacturers. As a result, ICs security has become a new concern in the system design due to potential malicious modification of the ICs during the fabrication process. Such modification of ICs referred to as Hardware Trojan (HT). The HT has two main parts: (i) a trigger: a circuit (signal) that activates the Trojan and (ii) a payload: a circuit that performs the malicious function activated by the trigger signal. The HT detection mechanisms vary depending on the deployment phase (like, specification, RTL, layout, and fabrication) and the required inputs (like, golden chips, etc.). There are many works that targeted HT detection. In this section, we briefly summarize the notable works on HT detection.

In [28], optical inspection based HT detection technique is presented. In this method, the layout of the circuit under test is compared with a picture of the manufactured circuit under test, obtained by removing the layers one by one. This method requires sophisticated and highly accurate techniques to obtain and analyze the die photo of the chip under test. However, the process is expensive and time-consuming to apply.

In [75], the authors propose randomization to compare, in probability, the functionality of the original design and the final circuit. The proposed method first uses randomization arguments to construct a unique probabilistic signature of a circuit. Then, it applies a hypothesis-testing technique to statistically infer the presence of a Trojan in the circuit under test (CUT). Tehranipoor et al. [112] presented a technique to increase the probability of generating a transition in a Trojan and analyze its activation time. In both cases, however, it is not guaranteed to find the test vectors capable of triggering Trojans, and therefore detection using this technique is not guaranteed.

Side-channel based HT detection mechanism has been presented in [17]. In this method, Principle Component Analysis (PCA) is used as a side-channel fingerprint of the circuit to compare it with the golden model. However, the characteristics of the physical design can be modified by other factors and not only by HT. As a result, HT detection may not be effective and time-consuming. In [91], authors replace the requirement of the golden model by using a golden parametric signature obtained by a trusted simulation model, parameters from the die, and applying advanced statistical modeling techniques. However, the requirement of a precise model of the process makes the technique difficult.

In [30], run time detection technique of HTs has been presented. Both hardware and software have been used to detect HT. An additional circuit (logic) is added to support security monitoring at run time. But, this technique is expensive in terms of circuit area. Hicks et al. [73] presented a HT detection method at RTL level. The HT detection problem is formulated as an unused circuit identification (UCI) problem. However, how to define unused circuits is not easy and is not quite clear. Therefore, most of these approaches compared the golden model with the design circuit to detect HT.

Bao et al. [25] proposed a reverse engineering approach to automatically distinguish the features between HT-free and HT-inserted structures in the ICs. The approach adopts a well-studied machine learning method, a one-class support vector machine, to classify ICs as Trojan-free and Trojan-inserted based on features extracted from the IC images and thus simplifies the process of reverse engineering. However, the approach would not allow destructive verification of ICs to be either cost-effective or scalable. Therefore, more advanced detection mechanisms are required to eliminate the requirement of the golden model.

A method for detecting hardware Trojans in third-party digital IP (3PIPs) cores has been discussed in [131]. In this paper, first, all functions in the behaviour are defined as properties. Then a complementary flow is presented to verify the presence of Trojans in 3PIPs by identifying suspicious signals (SS) with formal verification, coverage analysis, removing redundant circuits, sequential automatic test pattern generation (ATPG), and equivalence theorems. Moreover, a verification approach of system specification and implementation was also presented to identify extra functionality in hardware designs.

HLS tools are widely used in many critical designs. The tools may contain malicious codes (HT) which collect valuable data in IPs or disrupt the performance of the design after a certain time. However, most of the HTs detection approach discussed above compared the golden model with a design circuit to detect HT. To the best of our knowledge, there are no techniques that can detect high-level synthesis inserted HTs. The method proposed in chapter 6 shows how all the three HTs inserted by [106] can be detected using our proposed approach.

2.4 Conclusion

In this Chapter, we have discussed several state-of-the-art RTL to C reverse engineering techniques. We have also presented various simulation-based verification, phase-wise verification, and end-to-end verification techniques for checking the correctness of the HLS tool. The state-of-the-art Trojan detection mechanisms are also discussed in this chapter. In this process, we have identified some limitations of existing reverse engineering, verification, and hardware Trojan detection methods of HLS. In the subsequent chapters, we present the Fast Simulation framework, End-to-end Verification of HLS, RTL to C reverse engineering for faster simulation, and Trojan detection frameworks which overcome the limitations identified in this chapter.

3 FastSim: A Fast Simulation Framework for High-Level Synthesis

3.1 Introduction

The ever-increasing complexity of digital design and the inception of information technology are driving design methodologies towards the use of high-level behaviour than the register transfer level (RTL). In this scenario, high-level synthesis (HLS) plays a significant role by enabling the automatic generation of RTL design starting from high-level descriptions. Several HLS tools like Vivado HLS [10], Catapult-C [3], Intel OpenCL HLS [6], SCC [13], etc. from industries and Bambu [107], LegUp [36], etc. from academia have been introduced both for field-programmable gate array (FPGA) and application specific integrated circuit (ASIC) hardware designs. The HLS enables the algorithm developers to use the FPGA targets by abstracting lower level details like clock, target architecture and reconfigurability.

Since its induction, HLS has made significant progress in shortening system design time by providing flexible and instant optimization opportunities at the behavioural level for pipelining, loop unrolling, enabling parallel dataflow streams, etc. However, the verification of the synthesized model is still primarily carried out by time consuming RTL simulations. Although various phase-wise formal verification methods of HLS are proposed [87, 24, 46], a monolithic end-to-end formal verification of HLS is not yet available due to the difference between high-level programming sets in various computer languages, and the generated RTL. Consequently, HLS designers still depend on Xilinx RTL simulators (e.g. Xsim, VCS) or hardware-software power emulators (e.g., Zebu [12]) for verification purposes. Due to the RTL simulator's verification time overhead and the non-FPGA experts being unable to understand the details of the RTL code, the HLS tools come up with software based verification. In Vivado HLS design Suite [10], There are two steps to verifying the design [110]: (i) *C-simulation*: where, Before synthesis, the behavioral specifications to be synthesized should be validated with a test bench using C simulation. The test bench is self-checking and validates that the results from the design to be synthesized are correct. (ii) *C/RTL co-simulation*: where the Vivado HLS can verify that the synthesized RTL is functionally identical to the C source code with the original (same) test bench. Although Vivado HLS uses both C-simulation and RTL co-simulation to determine the correctness of the design, C-simulation is faster than the RTL co-simulation.

A comparative analysis of the simulation time (in seconds) for C simulator and RTL simulators like Vivado RTL co-simulator and ModelSim are presented in Table 3.1 for a few HLS benchmarks tested for 30k input test cases. From the table, it could be concluded that C simulation is much faster compared to HLS based RTL co-simulation. The simulation times for RTL simulators are comparable.

Table 3.1: *C simulation vs RTL simulation comparison*

Bench	C-sim(s)	RTL co-Sim(s)	Modelsim(s)
des	28.01	34672	36024
mips	0.985	2620	2885
aes_enc	12.656	4389	4693
aes_dec	14.442	4467	4780

Table 3.1 gives an intuition that a C like behavioural code realized from HLS generated RTL is likely to simulate faster as compared to traditional RTL simulation. The abstraction of the high-level model from RTL has been in use in mainstream companies for decades, for cycle-based simulations in early verification phases. For example, Tenison VTOC [1] automatically generates C++ or SystemC models from an RTL hardware description. The emulation of RTL behaviour using a sequential language like C adds up several important constraints to incorporate the significant features of RTL simulators like cycle accuracy,

accurate performance estimation, the capability to simulate instruction and task level parallelisms along with proper code readability. Two closest works with these targets are FLASH [44] and Verilator [11]. FLASH [44] incorporates scheduling information in the source C code for cycle accurate simulation. Although FLASH guarantees faster simulation of scheduled C code, it does not consider the design transformations during allocation, binding and the datapath, and controller generation phases of HLS. Consequently, the correctness of RTL generated by HLS is not guaranteed by FLASH. On the other hand, Verilator [11] generates C++ code from any synthesizable Verilog RTL for faster simulation. The C++ code could be simulated faster than RTL simulation and can verify the functional correctness of RTL generated by the HLS tool. However, being a generic tool, Verilator disregards the leverages offered by the finite state machines with datapath (FSMD) oriented nature of HLS generated RTLs where the datapath and the controller are well separable and the controller is specified as a well defined finite state machine (FSM). Hence, the generated C++ code is highly complex and much slower compared to FLASH.

3.1.1 Contributions

In this work, we develop a simulator that overcomes the limitations of both FLASH and Verilator. Specifically, we propose a framework that converts an HLS generated RTL to an equivalent C-code similar to Verilator but takes advantage of the structure of the HLS generated RTL. We extract the register transfer (RT) operation(s) performed in the datapath in each state of the controller FSM from the control signal assignment of that state. This way our simulator automatically generates the behavioural FSMD in C code semantics from the HLS generated Verilog RTL while maintaining the state machine sequence of the synthesized RTL. Our framework guarantees fast simulation, functional correctness of the RTL, cycle accuracy, accurate performance estimation, and generates a highly readable and debug friendly simulation code by preserving all register and port names for easier correlation with the HLS synthesis report. In addition to typical C programming constructs, our framework supports advanced HLS constructs like array mapping to external memory modules, non-inlined function calls, parallel execution frameworks invoked by loop unrolling, pipelining, etc., and accurate simulation of pipeline stalls during external memory access. The contribution of this chapter are summarized as follows:

- We demonstrate the efficiency of FSMD aware RTL to C conversion for faster HLS design verification.

- We present a completely automated, fast, and cycle accurate simulation based verification framework *FastSim* for HLS generated RTLs. The framework ensures the end-to-end correctness of HLS. It is also equipped to give accurate design performance estimation.

- FastSim can model various hardware parallelisms like loop and task level pipelines. Our simulator also generates a well indented and arranged simulation C code for convenient design debugging.

- We also present a detailed experimental comparison of our simulation framework for RTL generated by the Vivado HLS tool with state-of-art simulators like Verilator, ModelSim, Vivado RTL cosimulator (XSIM) etc. on diverse workloads from CHStone benchmark suite and several other standard programs.

The remainder of this chapter is organized as follows. Section 3.2 presents our proposed framework and flow diagrams. The RTL to C conversion and its implementation details are discussed in Section 3.3. Section 3.4 demonstrates the major challenges faced during RTL to C conversion. In Section 3.5, we elaborate the parallel execution strategy adopted by *FastSim*. The debug strategies and design performance estimation are discussed in Section 3.6. Experimental results are presented in Section 3.7. Finally, Section 3.8 concludes the chapter.

3.2 Our Proposed Framework

The overall flow of the FastSim framework is demonstrated in Fig.3.1. Our tool takes the HLS generated Verilog RTL as the input and generates a C code from the RTL for faster simulation. The Verilog RTL is first converted into an abstract syntax tree (AST) format using the PyVerilog [9] library and further processing is done on AST. The RTL to C conversion process consists of two phases: *Pre-processing and RTL to C reverse engineering.* In phase one, the C incompatible constructs of Verilog like bit-select and part-select are replaced with equivalent compatible representations. In the second phase, we identify the register transfer operations in the datapath concerning the control signals at each control step of the controller FSM to generate an equivalent FSMD code. The FSMD has the same structure as the controller FSM as it preserves the original state sequence found in Verilog

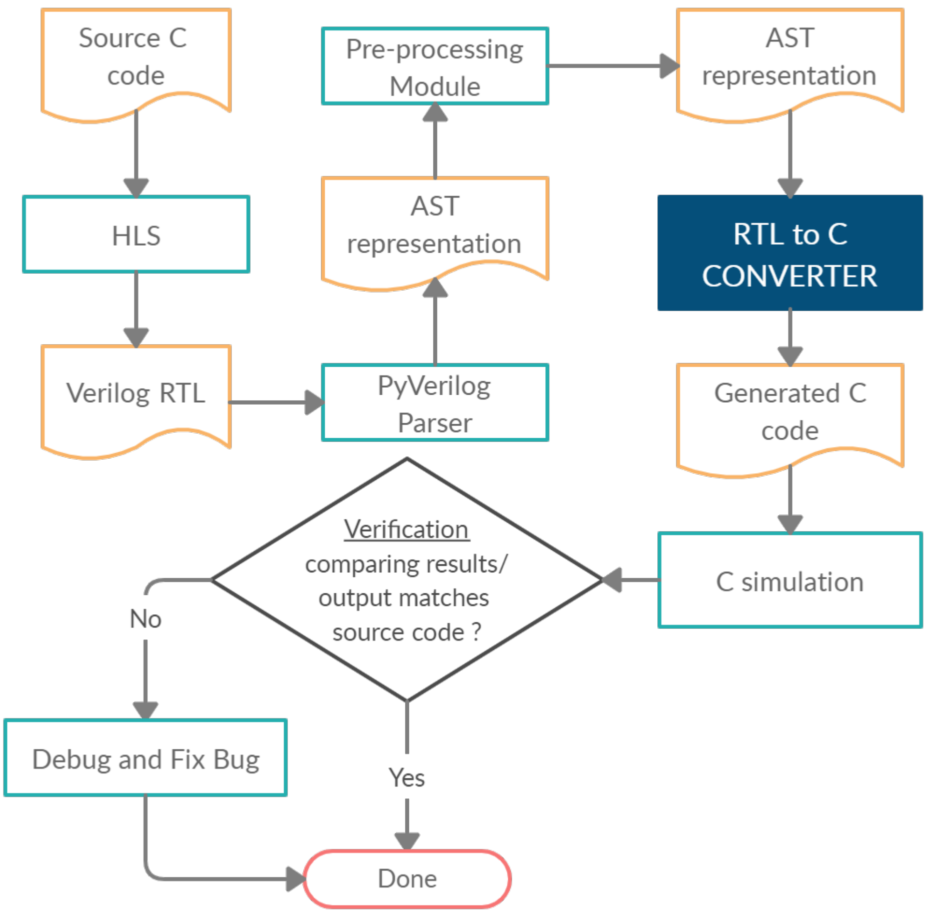


Figure 3.1: *FastSim: Overall flow*

RTL. Our simulation model can handle external hardware modules like RAM, ROM, and instantiated hardware modules. It also supports hardware parallelism like pipelining and loop unrolling. The intermediate FSMD and the variables, controller state, RT operations, etc. are mapped into appropriate data types and the equivalent C code, called RTL-C, is generated. Next, we have the *Simulation* phase where the behavioural C source and the RTL-C code are co-simulated using C compilers like GCC or using C-simulation of HLS tools with test cases to verify the functional correctness of the generated RTL. If the outputs match, we conclude with successful verification. In case of a mismatch, the RTL-C code can be used for debugging.

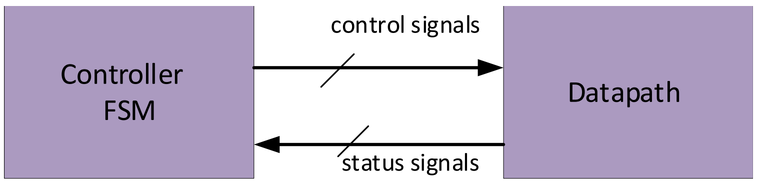


Figure 3.2: *RTL structure generated by HLS*

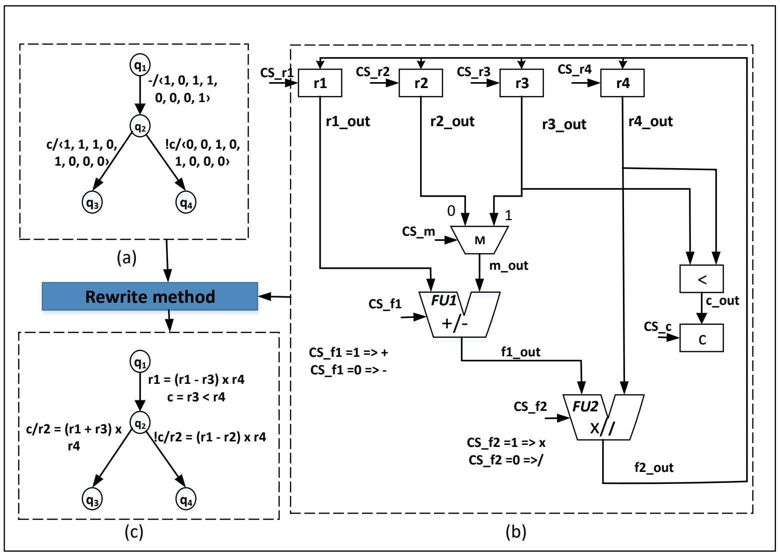


Figure 3.3: *(a) Controller FSM. (b) Datapath. (c) FSMD*

3.3 RTL to C Conversion

3.3.1 RTL Structure

The RTL generated by HLS consists of a datapath and a controller FSM as shown in Fig. 3.2. The datapath consists of registers, external memory units, functional units (FUs) and their interconnection network. The RTL operations performed in the datapath are controlled by the controller FSM. In each state, the controller assigns 0/1 value to each control signal. As a result, a set of RT operations are performed in the datapath. The datapath sends some status signals (i.e., results of some conditional checks) to the controller. The FSM state transitions depend on those status signals. We take *advantage of this RTL structure during RTL to C conversion*. In each state of the controller FSM, we analyze the datapath using control signals values in the particular state to identify the RT operation(s) executed in the datapath. We then replace the control signals in that control state with the corresponding

RT operations. This way the controller is converted into an equivalent FSMD. The FSMD is an abstract representation of the RTL. The overall idea of FSMD abstraction from the RTL is given in Fig. 3.3. We represent this FSMD as C code for faster simulation.

```
Source: (at 2)
  Description: (at 2)
    ModuleDef: findmin (at 2)
      Paramlist: (at 0)
      Portlist: (at 3)
        Port: ap_clk, None (at 4)
        Port: ap_rst, None (at 4)
        Port: ap_start, None (at 4)
        Port: ap_done, None (at 4)
        Port: ap_idle, None (at 4)
        Port: ap_ready, None (at 4)
      Decl: (at 22)
        Parameter: ap_ST_fsm_state2, False (at 22)
          Rvalue: (at 22)
            IntConst: 2'd2 (at 22)
      Decl: (at 23)
        Input: ap_clk, False (at 23)
      Always: (at 66)
        SensList: (at 66)
          Sens: posedge (at 66)
            Identifier: ap_clk (at 66)
        Block: None (at 66)
          IfStatement: (at 67)
            Eq: (at 67)
              Identifier: ap_rst (at 67)
              IntConst: 1'b1 (at 67)
```

Figure 3.4: *AST Code Snippet*

3.3.2 AST Representation

An AST is the syntactic representation of the source code written in a programming language. We use the AST representation of Verilog generated by the HLS tools as our intermediate representation (IR) in our tool. We use PyVerilog library [9] to generate this AST representation. Starting with the top module as a root node, the subsequent child nodes contain HDL constructs like parameters, port list, declarations, etc, which are arranged systematically and any required construct could be explored by traversing the AST. The snippet for AST generated for some example is shown in Fig. 3.4.

As seen clearly in Fig. 3.4, the top node in the representation is ModuleDef in our case it is “find min”. The child nodes of the module contain items like parameter list(ParamList), port list (Portlist), declarations(Decl), blocking construct(Block), etc. This provides us with an organized representation of different parts of RTL and that too in a tree-like manner. This made the task of finding the items easy and stored in a systematic manner, and hence AST forms an important part of parser implementation.

3.3.3 Automatic Pre-processing of RTL

Verilog HDL supports direct bit-level manipulation of register arrays like concatenation of bits, assigning subset of bits of one variable to another. C doesn’t provide support for such direct manipulation. Consequently, we need to convert these operations into a combination of supported constructs like bit-wise AND/OR, left/right shift, etc. to mimic an equivalent operation in C. Our pre-processing wrapper script automatically identifies such unsupported operations and converts them to an equivalent supported expressions in the AST format of Verilog RTL. The operations that needed pre-processing are explained as follows:

3.3.3.1 Concat Operation

This operation combines two or more register arrays into a single entity. The objects for *concat* operation may be scalar (single bit) or vector (multiple bits). Multiple concatenations may be performed which is known as replication.

```
// Concat operation:
reg [WIDTH_A - 1 : 0] reg_A;
reg [WIDTH_B - 1 : 0] reg_B;
reg [WIDTH_C - 1 : 0] reg_C;
assign reg_A = {{reg_B}, {reg_C}};

// After pre-processing:
wire [(WIDTH_A - 1):0] temp_0, temp_1;
assign temp_0 = reg_B << width_C;
assign temp_1 = temp_0 | reg_C;
assign reg_A = temp_1;
```

Figure 3.5: *Example of pre-processing of Concat operation*

Example 4. *Consider the example shown in Fig. 3.5. The bits of register reg_B is concatenated with the bits of register reg_C and is assigned to the register reg_A such that the* *total width is equal to that of reg_A. To convert this to an equivalent supported expression, we first left shift reg_B with the width of reg_C and store the result in temp_0. The temp_0 is then 'OR'ed with reg_C and is stored to temp_1. This operation is equivalent to appending reg_C at the LSB end of temp_0. The final value in "temp_1" is assigned to reg_A.* □

3.3.3.2 Part-select Operation

The part-select operation is used to select a part of a vector, i.e., give access to a set of contiguous bits in the vector whose range is specified by two digits separated by a colon (:). The first digit represents the least significant bit (LSB) numbered, and the second digit represents the most significant bit (MSB) numbered. Both LSB and MSB need to be constant.

```
// Part-select operation:
reg [63:0] reg_A, reg_B;
always @ (*) begin
   reg_A[40:0] <= reg_B[51:11];
end
// After pre-processing:
wire [63:0] temp_2, temp_3, temp_4;
always @ (*) begin
   reg_A <= temp4;
end
// width of register reg_A = 64 bit.
assign temp_2 = reg_B & 64'd4503599627368448;
assign temp_3 = reg_A & 64'd18446741874686296064;
assign temp_4 = temp_2 | temp_3;
```

Figure 3.6: *Example of pre-processing part-select operation*

Example 5. *For part-select operation shown in Fig. 3.6, the bits 0 to 40 of register reg_A are to be replaced with bits 11 to 51 of register reg_B. To implement this we first perform Bit-wise AND on register reg_B with a 64 bit masking vector 4503599627368448* (i.e., $(2^{41}-1) << 11$) *which extracts bits [51:11] and store the result in temp_2. For register reg_A, we perform Bit-wise AND with 64 bit vector 18446741874686296064* (i.e., $(2^{23}-1) << 41$) *to reset the bits [40:0] and store the result in temp_3. We then perform bit-wise OR between temp_2 and temp_3 to copy the contents [51:11] reg_B stored in temp_2 into reg_A stored in temp_3. Finally the result is assigned to LHS register reg_A. There are many other versions of concat*

and part-select operations. Our pre-processing script can handle all of them automatically.
□

It may be noted that the concatenation and part-select are supported in Vivado HLS C-simulation. We still converted them to make our generated RTL-C code tool independent.

3.3.4 RTL to C Conversion

The input to the RTL to C conversion is the AST representation of the Verilog. The overall flow of the process (parser) is shown in Fig. 3.7 and the steps of the conversion process are described below.

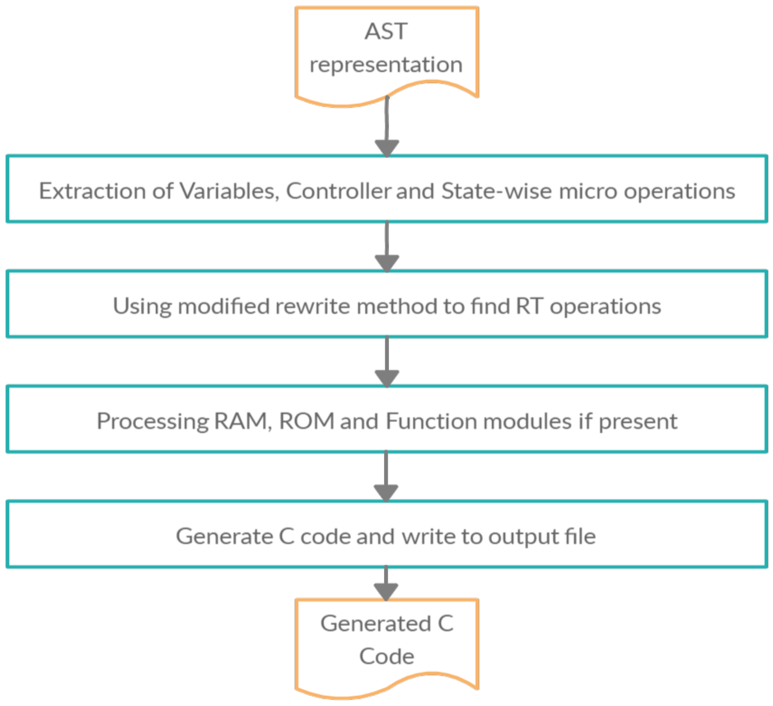

AST representation
Extraction of Variables, Controller and State-wise micro operations
Using modified rewrite method to find RT operations
Processing RAM, ROM and Function modules if present
Generate C code and write to output file
Generated C Code

Figure 3.7: *Parser Flow Diagram*

3.3.4.1 Extraction of Variables, Controller and State wise Micro-operations

Register declarations, controller FSM, and operations are located at different sections of the AST. Extraction and storage of this information constitute the first step of the conversion.

The variables used can be found under *Declaration* type of the AST. These variables are categorized as wire, register, inputs, and outputs and are stored accordingly into an appropriate data structure (see Figure 3.8) along with their data-width and sign information.

```
Data-structure (variables: input, output, registers and wires):
[
    "var_name": { "width":value, "signed":true/false },
    "var_name": { "width":value, "signed":true/false },
    "var_name": { "width":value, "signed":true/false },
    ...
]
```

Figure 3.8: *Data-structure for storing variables*

The controller forms the logical flow of the program and is the driver of the Verilog RTL. In AST, the controller is identified as type *Case Statement*. Once extracted, it is stored in a separate data structure (see Fig. 3.9) with the list of available states. For each state, their conditions and next states also persisted.

```
Data-structure (controller):
{
    "state1": [{"condition":"next_state"}, {"condition":"next_state1"}, ...]
    "state2": [{"condition":"next_state"}, {"condition":"next_state1"}, ...]
    "state3": [{"condition":"next_state"}, {"condition":"next_state1"}, ...]
    ...
}
```

Figure 3.9: *Data-structure for storing Controller*

Finally, the micro-operations taking place in the states are extracted. In the datapath, RT operations are controlled by the control signals. For each datapath component, the input to output assignments is termed as micro-operations. For example, for a multiplexer $out = MUX(in1, in2, sel)$, there are two micro-operations possible, i.e., $out \leftarrow in1$ and $out \leftarrow in2$ and the associated control signal assertions are $sel = 0$ and $sel = 1$, respectively. There are many micro-operations possible in the datapath. However, not all of them are active in a control state. Given a control signal assignment in a control state, we, therefore, identify the active micro-operations in that state. A micro-operation not associated with any control signal is always active. In RTL, each state of the controller has associated with some "always" block, and each always block has some condition that contains condition variables and micro-operations. The operations in "always" and "assign" statements are the active micro-operations in that state. These micro-operations are then stored in a state-wise manner in a separate data structure (see Fig. 3.10).

```
Data-structure (RT operations):
{
   "condition":"...", "operation": [op1, op2, op3, ...],
   "condition":"...", "operation": [op1, op2, op3, ...],
   "condition":"...", "operation": [op1, op2, op3, ...],
   ...
}
```

Figure 3.10: *Data-structure for state-wise Micro-operations with their conditions*

Algorithm 1: RTL-FSMD_Extraction (RTL)

```
   Input: RTL
   Result: RTL-FSMD
   /* RTL consists of a Datapath D and a controller FSM F */
1  foreach state S in the controller FSM F do
2      Find the active micro-operations $M_S$ for the control signal assignments in $S$;
3      $R_S = \Phi$; /* Set of RT-operations in S */
4      foreach micro-opn of the form $\mu : r \leftarrow r_{in}$ in $M_S$ do
           /* Rewrite method */
5          do
6              w = Find the left-most wire signal in the RHS exp $\mu_e$ of $\mu$;
7              Find a micro-opn of the form $w \leftarrow e_w$ in $M_S$;
8              Replace $w$ with $(e_w)$ in the $\mu_e$;
9          while (all signals in RHS exp $\mu_e$ of $\mu$ are either Input, Reg or Constant);
10         $R_S = R_S \cup \{\mu\}$;
11     Replace the control signal assignments in $S$ of $F$ with $R_S$;
12 Return $F$; /* FSM F is converted to FSMD F at this point */
```

3.3.4.2 Rewrite Method

The next task is to identify the RT operations in each state of the controller FSM from the active micro-operations in that state. We use the *rewriting method* adapted from [80] for this purpose. The rewriting method identifies the *spatial sequence* of data flow needed for an RT operation in reverse order. The method consists of rewriting terms one after another in an expression. The micro-operation of the form $r \Leftarrow r_{in}$ in which a register occurs on the left hand side (LHS) is found first. Next, the right hand side (RHS) expression r_{in} is rewritten by looking for an active micro-operation of the $r_{in} \leftarrow s$ or $r_{in} \leftarrow s1 < op > s2$. In the next step, s ($s1$ or $s2$ in the latter case) are rewritten provided they are not registers or inputs. The rewriting takes place from left to right in a breadth-first manner. The process terminates successfully when all signals in the RHS expression are registers or inputs or constant. The rewriting method is given as Algorithm 1 and is explained with an example below.

Example 6. *Let us consider the datapath and controller FSM shown in Fig. 3.3. All the control signal names start with CS. Let the order of the control signals be ⟨CS_m, CS_f1, CS_f2, CS_r1, CS_r2, CS_r3, CS_r4, CS_c⟩. Let us consider the control assertion* $A = \langle 1, 1, 1, 0, 1, 0, 0, 0 \rangle$ *of the transition* $q_2 \rightarrow q_3$. *For this control assertion, the activated micro-operations are:{r1_out ⇐ r1, r2_out ⇐ r2, r3_out ⇐ r3, r4_out ⇐ r4, m_out ⇐ r3_out, f1_out ⇐ r1_out + m_out, f2_out ⇐ f1_out × r4_out, r2 ⇐ f2_out}. Out of them, r2 ⇐ f2_out is the micro-operation with register r2 at left hand side. The sequence of rewriting process to accomplish the corresponding RT-operation are as follows:*

r2 ⇐ f2_out
r2 ⇐ f1_out × r4_out [since f2_out⇐ f1_out × r4_out]
r2 ⇐ (r1_out + m_out) × r4_out [since f1_out⇐ r1_out + m_out]
r2 ⇐ (r1_out + r3_out) × r4_out [since m_out⇐r3_out]
r2 ⇐ (r1 + r3_out) × r4_out [since r1_out⇐r1]
r2 ⇐ (r1 + r3) × r4_out [since r3_out⇐r3]
r2 ⇐ (r1 + r3) × r4 [since r4_out⇐r4]
Since both r1, r3 and r4 are registers, rewriting process stops.

So, the RT-operation r2 ⇐ (r1 + r3) × r4 is executed by the given control assertion A in the transition $q_2 \rightarrow q_3$. *The RT-operation(s) for all other state transitions of the FSM can be found in a similar manner. The obtained FSMD behaviour is given in Fig. 3.3(c).*
□

Limitation of the existing rewriting method in [80]: The existing rewrite method does not take data-width for LHS and RHS expressions into consideration. This will create an issue in C if the data-width mismatch between the RHS and LHS expression. We have enhanced the rewriting method to overcome this limitation. Specifically, we make sure at each step of rewriting method that the data-width of the LHS and RHS are matched. We perform bit-wise AND operation with each micro-operation with a constant value representing the data-width of the register on the LHS of the micro-operation. In the above example, let us assume the data-width of r1, r3, r4 are 20 bits each and data-width of r2 is 32 bits. As a result, the data-width of the RT operation r1 + r3 is 21 bits and the the data-width of RT operation r2 ⇐ (r1 + r3) × r4 is 41 bits. The final RTL operation would be $r2 \Leftarrow ((r1 + r3) \times r4)\ \ \&\ \ (2^{32} - 1)$. This process is applied in each step of rewriting method to resolve the limitations of the original rewrite method. The data-width mismatch issue is explained in detail in the next section.

3.3.4.3 RAM, ROM and Modules

FastSim supports single and dual-port RAM and ROM modules. The RTL contains operations that set either of the *read* or *write* signals to indicate reading or writing is performed on the RAM/ROM. In RTL-C, the RAM or ROM modules are defined as an array with the size information collected from their respective module. Whenever the read or write enable signal is set in a state, the corresponding read/write operation on RAM or ROM block is placed at that state. The name of the module and the number of ports for RAM/ROM are taken as input from the user for processing. The tool processes the user inputs and RAM/ROM modules to identify and store the read/write operations along with other information like instance identifier prefix, number of ports, module name, and state-wise signal flags. A sample of generated C code snippet corresponding to a read operation on a RAM is shown in Fig 3.11. *CE_x* and *WE_x* depict *chip enable* and *write enable* signals, respectively, used to perform read and write operations. The *CE_x* signal is activated at state 5 to perform a read operation to register *out_A* from an address pointed by *addr_reg*.

For a function call in input C, the HLS tool usually creates a separate module with a datapath and a controller FSM in the RTL. FastSim first creates a function modeling the FSMD of that module. Similar to the RAM/ROM modules, such modules are also triggered in RTL by setting the value of a signal corresponding to that module in a particular state(s). Our tool identifies such calls and stores information like instance identifier prefix,

```
        if(cur_state == 5){
            address_x = addr_reg; CE_x = 1; WE_x = 0;
        }
        if(CE_x)
           out_A = RAM_x[address_x];
        if(WE_x)
           RAM_x[address_x] = reg_B;
```

Figure 3.11: *RTL-C code snippet for RAM module*

state-wise signal information, and a list of parameter variables passed during the module instantiation in a data structure. This information is then used to place a function call in the corresponding state while implementing the final C code. The variables are passed as references to the called function. Multiple functions with no data flow among them can be scheduled in a state by the HLS tools. In such a case, the function calls are placed sequentially in the corresponding state. FastSim supports a hierarchy of function calls as well. We have also taken care of the common input arguments among functions in a state by sending the "_old" values of them (as discussed in Subsection 3.4.1). Since the top module waits until the completion of execution of the module it is called, cycle accurate simulation is achieved by following the states of the respective FSMs. For multiple functions called in a state, we consider the maximum of their cycles needed for cycle accurate simulation. To handle functions where dataflow optimizations are applied, we need a different strategy as discussed in Subsection 3.5.3.

3.3.4.4 Generation of Cycle Accurate RTL-C Code

After the completion of the above steps, the controller FSM is converted into a behavioural FSMD. This FSMD constitutes state-wise segregated register transfers and is a behavioural description of the RTL. We generate a C program, called *RTL-C*, from this FSMD. The RTL-C looks the same as the controller, and it preserves the original state sequence found in the RTL. This abstracted RTL-C code *ensures cycle accurate simulation of the RTL*. In RTL, everything runs in parallel and is triggered based on appropriate signals and a clock. Decoding the RTL execution flow, handling data dependencies, and pre-processing certain bit-level operations and other issues, FastSim generates the equivalent C code RTL-C, which is sequential in nature. The outline of the RTL-C is given in Fig. 3.12. At the top of the program the constants are defined, followed by the "two's_complement" function

```
#include<stdio.h>
#define CONSTANT
function_prototypes();
int main(){
    //variable declaration;
    //RAM,ROM declaration;
    state1_label:
        //copy old_var_value=new_var_value;
        //place operations which belong to all states here;
        //place operations deciding condition variables here;
        //place level-triggered blocks here;
        if(condition1){
            //operations
            ...
            RAM/ROM blocks(if any)
            function_calls(if any)
            goto state2_label;
        }
        if(condition2){
            //operations
            ...
            RAM/ROM blocks(if any)
            function_calls(if any)
            goto state3_label;
        }
    state2_label:
        ...
        ...
    end:
        return;
}
```

Figure 3.12: *Generated RTL-C code outline*

definition which is used for variable-length signed conversions. Then the function prototypes are placed in case if modules are present in the RTL and finally the top function is declared with its body. The FSM behaviour is modeled using *goto* statement in C code, where each state consists of conditions, operations and *goto* jumps to the appropriate next state. For each state, the following operations are added in RTL-C in the given order:

- New values of variables are copied to old value variables.
- Operation which belongs to all states (i.e. always block which does not have any state variable)

- All the operations related to conditional variables are identified and placed first.
- Level triggered always blocks are placed next.
- Subsequently, all the pos-edge triggered always block operations are placed along with their respective conditions.
- For RAM and ROM modules, their operations are placed with the necessary condition if the corresponding read or write signal is set in that state.
- The function calls are placed next if the corresponding signal is set in that state.

Every variable in the RTL is considered unsigned unless explicitly stated. Every variable in RTL-C is declared as “unsigned long long” or “long long”. So, FastSim supports bit width up to 64 bits data width of RTL. To support bit width larger than 64 bits, we can use ap_int library [5]. At the start of the top module function, a local copy of all the passed reference variables is declared, and at the end of the program, all the reference variables are updated with the latest value.

3.4 Challenges Resolved in RTL to C Conversion

In this section, we describe major challenges that we need to address in RTL to C conversion with the solution approaches.

3.4.1 Data Inconsistency

Problem: Different *always* code blocks in Verilog generate different hardware modules which work concurrently at a given instance of time. On the contrary, the generated RTL-C executes in a sequential manner. This highly contradicting execution approach of Verilog and C might trigger data inconsistency issues.

Example 7. *Consider a scenario where a non blocking write and read are performed on a register at a given clock cycle/state. For instance, refer the example shown in Fig. 3.13. At state 2, two RT operations are performed. Assume the current status of registers to be r_X = 10, r_Y = 20, r_A = 10, and r_M = 30. Using the previously stated rewrite method, the RT operations executing in this state are r_A = r_X + r_Y and r_C = r_A + r_M. In hardware, these two operations execute in parallel. Consequently, after execution r_A = 10 + 20 =*

```
reg [31:0] r_A, r_C, r_X, r_Y, r_A, r_M;
wire [31:0] w_B, w_D;
always @(posedge ap_clk) begin
 if(state2 == 1'b1) begin
  r_A <= w_B;
  r_C <= w_D;
 end
end
assign w_B = r_X + r_Y;
assign w_D = r_A + r_M;
```

Figure 3.13: *Data inconsistency problem*

30 and r_C = 10 + 30 = 40. Since the second assignment r_C = r_A + r_M is executed simultaneously with the first assignment, the old value of r_A is used although it updated at the same instance. Meanwhile, the generated RTL-C executes sequentially. During the execution, the instruction r_C = r_A + r_M, uses the new value of register r_A as it is already updated in the previous operation. Consequently, final output in this scenarios is r_A = 30 and r_C = 60, which is conflicting with the actual output. □

```
// Resolved RTL-C code
 unsigned long long int r_A, r_X, r_C, r_Y, r_M;
 unsigned long long int r_A_old, r_X_old, r_C_old, r_Y_old, r_M_old;
 r_A_old = r_A; r_X_old = r_X; r_C_old = r_C;
 r_Y_old = r_Y; r_M_old = r_M;
 if(state2 == 1) {
  r_A = r_X_old + r_Y_old; }
 if(state2 == 1) {
  r_C = r_A_old + r_M_old; }
```

Figure 3.14: *A Solution to Data inconsistency solution*

Solution: To distinctively identify the old and new values of registers, we keep two copies of each register in RTL-C: a normal variable and an old_variable. At any given state of FSM, we first copy the value of the normal variable into the old_variable. For any register transfer operation, we always use the value of old_variable on the RHS expression of the operation (i.e., for *read* operation) if it is a register. Consequently, the operations in the RTL-C will be as shown in Fig. 3.14. This eradicates the data inconsistency issue.

3.4.2 Sign Conversion

Problem: In Verilog, a negative value is stored in the form of two's complement number and it uses the keyword *signed* to represent signed values. Consider the example shown in Fig. 3.15. It could be observed that *$signed* is used to represent the signed value. If we use the RTL code shown in Fig. 3.15 directly in our RTL-C then it becomes: $reg\_A = 117 + reg\_B$. This is functionally incorrect as the two's complement of 117 is -11 for a data-width of 7. We need to get the two's complement representation of the signed numbers.

```
// RTL code
reg [6:0] reg_A, reg_B;
always @(posedge ap_clk) begin
 reg_A <= ($signed(7'd117)+$signed(reg_B));
end
// Resolved RTL-C code
unsigned long long int reg_B;
long long int reg_A;
reg_A = do_twos_compliment(117,7)+ do_twos_compliment(reg_B, WIDTH_B);
```

Figure 3.15: *Sign conversion issue and solution*

Solution: We have created a function that computes the two's complement value of a number and whenever *$signed* occurs while applying the rewrite method we use this function to compute the value of *$signed* constant or *$signed* variable. The function takes two parameters one is variable or constant, and the other is the bit size of the variable or the constant. The resolved C code in Fig. 3.15 shows the solution. The function body of two's complement function is obvious and omitted here for brevity. So, the two's complemented form of -117 (which is -11) will be used in RTL-C.

3.4.3 Data-width Mismatch

Problem: In RTL, the variables declared are of arbitrary bit width as per the need of the operation. But in C, the data types are of fixed size by default. If the data width in LHS and RHS are not the same, they are automatically adjusted in Hardware. If RHS bit-width is more than LHS, the extra bits of RHS are truncated during the assignment. Similarly, if the bit-width of LHS is more, the remaining LHS bits are automatically zero-padded during an assignment. A problem arises when we get an assignment constituting two mismatched register arrays. The fixed size of C data types can cause an overflow or underflow issue during the assignment in C.

```
// RTL Code
   reg [20:0] r_A;
   reg [20:0] r_B;
   reg [31:0] r_C;
   wire [31:0] w_C;
   always @ (posedge ap_clk) begin
       r_C <= w_C;
   end
   assign w_C = r_A * r_B;
// Incorrect RTL-C code
  unsigned long long int r_A, r_B, r_C;
  r_C = r_a * r_B;
// Resolved RTL-C code
  unsigned long long int r_A, r_B, r_C;
  r_C = (r_A * r_B) & 64d'4294967295;
```

Figure 3.16: *Data-width mismatch and solution*

Example 8. *As shown in the Fig. 3.16, the register r_C will store the 32 bits of the multiplication r_A * r_B although the result is 42 bits. On the other hand, the variable r_C in RTL-C will store all 42 bits of the result. Consequently, the behaviours of RTL-C code and Verilog RTL will not match.* □

Solution: To compensate for the variable width of the LHS register during assignments, we perform the "bitwise AND" operation on RHS with a mask of set bits of width equal to the width of the LHS register. This operation is performed for every micro-operation and hence the irrelevant bits in C code variables are zero-padded. Fig.3.16 shows the corrected C code, where we perform a bitwise AND on RHS with 4294967295 ($i.e., 2^{32} - 1$) so that the unwanted bits are reset and the RTL-C code will exactly imitate the RTL code.

3.4.4 Level-triggered Operations

Problem: Level-triggered operations are used to execute register transfers when the status of a sensitive signal changes. When the *always* block senses a change in the value of a variable in the sensitivity list, the operations in the block involving those sensitive variables in the RHS are performed. Generally, when a *read* as well *write* operation is performed on the same register at the same state, as already discussed in the data-inconsistency issue, we use the old value of the register for all *read* operations. This works well for edge sensitive always blocks, say an always block triggered at *posedge* of the clock. But in the level-triggered operation, this idea fails as the level sensitive always blocks could behave like combinational logic. Consequently, there would not be a distinction between old and new states. During such a scenario, if a simultaneous read and write to a given register is performed, the write updated value of the register has to be directly propagated to the LHS of the assignment where the register is read.

```
//RTL Code
reg [31:0] a, b; wire [31:0] c;
 always(*) begin
  if(cur_state == 1'd2 & reg_X == 1'd1) begin
     b <= a;
  end end
 always(*) begin
  if(cur_state == 1'd2 & reg_X == 1'd1) begin
     a <= c;
  end end
 assign c = a + 5;
```

Figure 3.17: *RTL with level triggered operation*

Example 9. *Fig. 3.17 shows an example of level-triggered operation and its corresponding RTL-C is shown in Fig. 3.18. As shown in Fig. 3.17, we have an unconditionally sensitive always block, which would effectively work like a combinational logic. If both the conditions on reg_X and cur_state are satisfied, as per the trivial generated C code as shown in Fig. 3.18, the old value of a i.e. a_old is assigned to b before updating the latest value of a. However, as per the combinational behaviour of the block, we are supposed to use the updated value of a which is (a_old + 5)) to update b. This leads to an incorrect execution of the block in C.*
□

Solution: We modify the perception of *always* block such that level triggered and edge triggered *always* blocks are handled separately. For a level triggered *always* block, if a simultaneous read and write are performed to a variable at a given state under the same condition, all the writes to the variable are *performed before reads in the generated RTL-C code*. Secondly, we will use the new value of the variable (instead of the old value as in edge triggered) in the RHS expression of the corresponding operations. For the example shown in Fig. 3.17, a is updated to $a\_old + 5$ before being copied to b as shown in the solution given in Fig. 3.18. This modification ensures the correct execution of the block in C. In general, a case may arise where a sequence of level triggered operations in a single state may modify a register. Based on our proposed solution, as we are using the new values for variables in RHS, the ordering of operations will be important. If such a case arises then a data flow analysis using a dependency graph needs to be incorporated. There is no such case found for the benchmark examples that we have considered for experimentation purposes. However, we will add to support in the future version of the tool.

```
//Incorrect RTL-C Code
 State_2:
  b_old = b;
  a_old = a;
  if(reg_X == 1) {
   b = a_old & 4294967295;
  }
  if(reg_X == 1) {
   a = (a_old + 5) & 4294967295;
  }
```

```
// Resolved RTL-C code
 State_2:
  b_old = b; a_old = a;
  if(reg_X == 1) {
   a = (a_old + 5) & 4294967295;
  }
  if(reg_X == 1) {
   b = (a & 4294967295);

  }
```

Figure 3.18: *RTL-C Code with level-triggered always block issue and its solution*

3.5 Modelling Hardware Parallelism in C

Most of the commercial and academic HLS synthesis tools support three forms of parallelism: loop unrolling, instruction level pipelining, and task level pipelining. Our simulator is equipped to support all three forms of parallelism. In this section, we explain the modeling of different forms of parallelism at the RTL in our generated RTL-C code.

3.5.1 Loop Unrolling

At the FSM level, loops are implemented in the form of an iterating set of states. Loop unrolling is implemented by executing register transfers of multiple iterations of a loop in a single iteration of the modified looping state set by considering their dependencies and using additional resources (duplicate registers and functional units). This effectively reduces the number of cycles and subsequently improves the latency of the design. In effect, multiple iterations of the loop execute in a single state. The unrolled FSM model is similar to the baseline FSM where the register transfers form the iterations of the loop. Since the overall structure of FSM remains the same, our simulation model successfully emulates the correct functionality for a loop unrolled model. FastSim supports partial unrolling of the loop as well.

```
void warp (int Img[ROWS][COLS], int Out[ROWS][COLS])
{
// a,b,c,d,e,f,i,j are defined as macros
 int x, y, destX, destY;
 loop-1.1:for(y = 0; y < 8; y += 1) {
   loop-1.2: for(x = 0; x < 8; x += 1) {
        #pragma HLS loop_flatten off
        #pragma HLS pipeline
        destX = (a*x + b*y + c*x*y + d);
        destY = (e*x + f*y + i*x*y + j);
        Out[destY][destX] = Img[y][x];
  }}
}
```

Figure 3.19: *Sample function warp with pipelined loop*

3.5.2 Instruction Level Pipelining

In HLS, a pipelined segment of a hardware module may be distributed across multiple states of the RTL FSM, each designated as different stages of the pipeline [52]. Each pipeline stage/state of FSM is further subdivided into multiple sub-stages, whose activity is controlled using specially designated sub-stage activation flags. It is to be noted that no two stages/states can execute in parallel. However, the sub-stages of the state/stage being executed may execute in parallel as per the state of their activation flag. For a *pipelined function*, the states are such that at a given clock cycle, a sub-stage of a given active stage x performs the register transfers, meanwhile, the other sub-stages of the same stage/state simultaneously compute the parameters required for the execution of stage x+1, x+2, etc. This improves the performance of the design. Similarly for *loop pipelining*, a sub-stage of a given active stage executing register transfers of iteration x so that nearly all the iterations

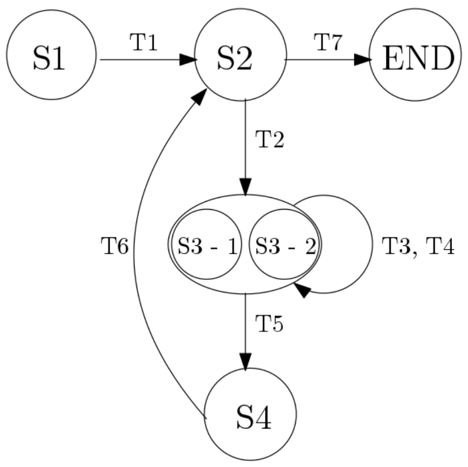


Figure 3.20: *Generated FSM for warp module*

could be completed within the specified initiation interval. For instance, Fig. 3.20 depicts the FSM generated by Vivado HLS for sample function warp demonstrated in Fig. 3.19. The FSM has a single pipeline stage/state *S3*, with two sub-stages *S3-1* and *S3-2*. The self looping transition *T3* of state *S3* indicates the inner *for* loop, *loop-1.2* of the function warp. At *S3-1*, the input matrix *Img* is read. The *x, y, destX* and *destY* for next iteration are also calculated in *S3-1*. At sub-stage *S3-2*, output array *out* is updated. For the first iteration sub-stage *S3-2* is disabled by resetting its activation flag. For all the subsequent iterations, both the sub-stages are executed in parallel. Fig. 3.21 shows an abstract of the pipelined state S3 in the RTL-C. The major difference from conventional states is that, in addition to regular FSM state variables, there are two additional variables, *S3_1_flag* and *S3_2_flag* that represents the activation flags of sub-stages S3-1 and S3-2, respectively of stage S3. We have already equipped our simulation model for parallel execution of register transfers in a single state for cases where there are simultaneous reads and writes to a given register as explained in Section 3.4.1. Consequently, the simultaneous manipulation of variables like *x, y, destX, destY* etc. in the two concurrently executing pipelined sub-stages *S3-1* and *S3-2* are automatically taken care. The *Img* and *Out* BRAM *chip enable (CE)* and *write enable (WE)* controls in the RTL-C segments are managed as per the state of sub-stage activation flags as shown in Fig. 3.21. The codes for calculation of *S3_1_flag*, *S3_2_flag* and other part of S3, and other states omitted for brevity. Representative functions *addr_calc*, *Calculate_DestX*, *Calculate_DestY* are used to calculate external BRAM address, *DestX*, *DestY*, respectively.

```
... // Code for other states.
STATE_S3: //pipelined state S3
 x_old = x; destX_old = destX; destY_old = destY; S3_1_flag_old = S3_1_flag;
     S3_2_flag_old = S3_2_flag;
 //Code to update S3_1_flag and S3_2_flag
 if(S3_1_flag_old == 1) { // S3-1 code block
   Img_CE = 1;
   Img_addr = addr_calc(x_old, x_old);
   destX = Calculate_DestX (x_old, y_old);
   destY = Calculate_DestY (x_old, y_old);
   x = x_old + 1;
 }
 if(S3_2_flag_old == 1) { // S3-2 code block
   if(x_old < 8){
      out_WE = 1;
      out_Addr = addr_calc(destX_old, destY_old);
   }
 }
 // other code in S3
```

Figure 3.21: *A representative code snippet depicting the pipelined state S3 in warp module*

3.5.3 Task Level Pipelining

Task level pipelining involves analyzing data flow/stream in a series of distinct non-inlined functions without feed-backs having producer-consumer relation and inserting first-in-first-out (FIFO)/Ping-pong (PIPO) buffers between their synthesized hardware modules so that multiple modules can execute in parallel. If the data is written into an array in producer module in the same order that is read from the array in consumer process, the array is implemented using FIFO. If it is not the case or Vivado HLS cannot determine it, then the memory is implemented using PIPO. The PIPO consists of two blocks of data, each of size of the original array. One of the block can be written by the producer process while the other block is read by the consumer process. The ping-pong ensures that the reading and writing of each block of data alternates in every execution of the tasks. In PIPO, the data can be written to or read from in any order. Whereas, the data is produced and consumed in the same order in FIFO.

In HLS, each module will be implemented as a distinct FSM and task level pipelining helps in the parallel execution of multiple FSMs. In Vivado HLS, the task level pipelining is implemented using the directive: *#pragma HLS dataflow*. By applying this pragma,

```
void model_flow(int A[LIMIT], int F[LIMIT])
    {
    int B[LIMIT], C[LIMIT], D[LIMIT],
        E[LIMIT];
    #pragma HLS dataflow
//HLS STREAM pragma is
//applied to B, C, D and E
        module_1(A, B, C);
        module_2(B, D);
        module_3(C, E);
        module_4(D, E, F);
}
void module_1(int A[LIMIT], int B[LIMIT],
    int C[LIMIT]) {
   int i;
    for(i=0; i< LIMIT; i++){
      B[i] = A[i] * 9;
      C[i] = A[i] * 2;
  } }
```

```
void module_2(int B[LIMIT], int D[LIMIT]) {
  int i;
   for(i=0; i< LIMIT; i++)
      D[i] = B[i] * B[i];
 }

void module_3(int C[LIMIT], int E[LIMIT]) {
  int i;
   for(i=0; i< LIMIT; i++)
     E[i] = C[i] * C[i];
 }

void module_4(int D[LIMIT], int E[LIMIT],
    int F[LIMIT]) {
  int i;
   for(i=0; i< LIMIT; i++)
     F[i] = D[i] + E[i];
 }
```

Figure 3.22: *Sample function "model_flow"*

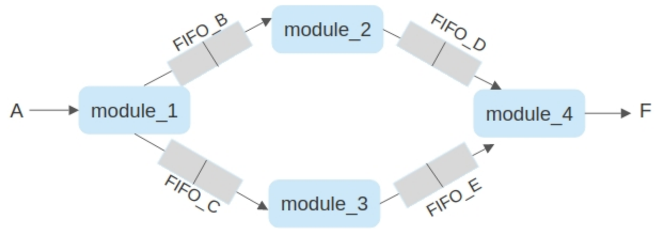


Figure 3.23: *"model_flow": Task level pipeline*

coarse-grain parallelism is achieved by overlapping computation with communication using a PIPO style buffer. As a result, all modules are running in parallel on a different set of data. Let denote this task-level parallelism as FSMD-level parallelism. For streaming applications implemented with FIFO, the producer and consumer process interactions are interleaved on the same set of data while maintaining synchronization. Let denote this task-level parallelism as data flow optimization. The dataflow directive can be combined with the pipeline directive within each loop in the producer and consumer modules to form fine grained parallelism of the operations on each data element. A typical example of data flow optimization, (function: *model_flow*) is presented in Fig. 3.22 and the corresponding data flow structure of *model_flow* generated by Vivado HLS is depicted in Fig. 3.23. After

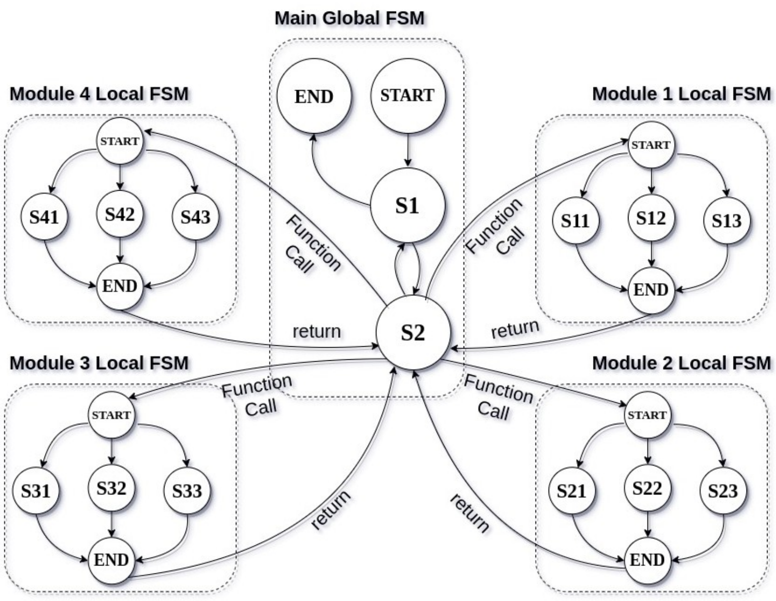


Figure 3.24: *Control flow of RTL-C: Task level pipeline*

evaluation of an iteration of *Module_1*, the output is pushed into *FIFO_B*. *module_2* will start processing its first iteration using this element while *module_1* processes the next element of array *B*. The *module_3* and *module_4* follow a similar execution pattern. A module stalls its execution either when its input buffer is empty or if its output buffer is full.

Unlike any form of parallelism discussed earlier, task level parallelism requires actual parallel simulation of multiple states of distinct FSMs. For cycle accurate simulation of the task level pipelining, our basic idea is to simulate a single FSM state of each module in each clock. To achieve this, a current state is maintained for each module. In each clock, the current state of each module is executed, and then the current state is updated according to the FSM transition of that module. Both FSMD-level parallelism and data flow optimization are modeled in a similar manner. Specifically, we design an additional global FSM *main* as shown in Fig. 3.24 with two states, the required FIFO/PIPO buffer instances, and other variables. At the global FSM, we cycle accurately handle FIFO/PIPO transactions between different modules whose equivalent C code is generated separately using our proposed RTL to C conversion. The synchronization in PIPO and FIFO is handled differently since reading and writing happen to two different buffers in PIPO whereas reading and writing occur in the same buffer for FIFO.

The *main* module has to be handled separately since it has an additional global controller

FSM. The design of the main module is as follows:

- Extract the parallel module instances and FIFO/PIPO instances shared between the modules from the AST representation.
- Extract the global control signals and initial values.
- Create a two state controller FSM and map the operations as follows:
  - S1: Termination Statements.
  - S2: All parallel module function calls and micro-operation depends on the module execution.
- Generate the C code and write to the output file.

```
state_1:
 if(ap_done) goto end;
 else goto state_2;
state_2:
 module_1(&m1_start, &m1_done, &is_empty_B, &is_empty_C...);
 module_2(&m2_start, &m2_done, &is_empty_B, &is_empty_C...);
 module_3(&m3_start, &m3_done, &is_empty_B &is_empty_C...);
 module_4(&m4_start, &m4_done, &is_empty_B &is_empty_C...);
 /* Following code segments ensure that a module starts only if the input FIFO isn't
     empty */
 m2_start = !is_empty_B;
 m3_start = !is_empty_C;
 m4_start = (!is_empty_E & !is_empty_D);
 ap_done = (m1_done & m2_done & m3_done & m4_done);
end:
```

Figure 3.25: *Representative RTL-C code structure of main*

The C code for the global FSM *main* corresponding to sample function *model_flow* is shown in Fig. 3.25. State *S1* of *main* checks if all the modules have finished execution and either concludes the execution or proceed to state *S2*. At state *S2*, all the four modules are invoked sequentially as shown in Fig. 3.24. Note that the modules can be invoked in any order in state S2. Since a single state represents a single clock cycle and the buffers are updated post execution of all the modules, the model effectively emulates the parallel execution of all four modules. The sample C code of the `module_2` is given in Fig. 3.26. As

```
// All variables are declared static
 if(resume_state == 1) goto S21;
 else if(resume_state == 2) goto S22;
 else if(resume_state == 3) goto S23;
S21:
   if (m2_done == 1 || m2_start == 0) return;
   else   resume_state = 2; return;
S22:
   // Activate fifo_B handshakes for read
   fifo_B(ce_B, we_B, &done_B, &is_empty_B ....);
   resume_state = 3; return;
S23:
  if (is_full_D) { // check if fifo_D is full
   resume_state = 3; return;
  }
  // Code statements to Calculate D[i]
  // Activate fifo_D handshakes for write
  fifo_D(ce_D, we_D, &done_D, &is_empty_D ...);
  resume_state = 1; return;
```

Figure 3.26: *Representative RTL-C code structure of module_2*

shown in Figs. 3.26 and 3.24, only a single state of each parallel module is executed on every invocation of that module, and the control returns to state *S2* of *main* and the execution state of each module is persisted globally (static variables) for clock synchronization.

The FastSim generates either an FSMD as shown in Fig. 3.12 for non-data flow case (where task level pipelining is not applied) or an FSMD as shown in Fig. 3.25 for data-flow case (when task level pipelining is applied) based on user inputs. In both cases, the FastSim identifies distinct FSMD modules from the RTL code. In a non-data flow case, FastSim creates a function call for such FSMD in the state where the function is scheduled. The tool then generates C equivalents corresponding to each FSMD module using the flow discussed in Section 3.3.4. In the data-flow scenario, it also extracts the structural FIFO/PIPO instances and models the FIFO/PIPO transactions using FIFO/PIPO module function call. FastSim then cycles accurate models of each module from its FSMD as discussed above. It then generates a global main module for controlling the transactions between the distinct FSMDs. The current version of FastSim cannot handle a nested task level pipeline where individual modules can be further pipelined in a hierarchical manner. It would be an interesting future work to explore how the strategy proposed in this work can be enhanced to accommodate a nested task level pipeline.

3.6 Debug Framework and Performance Estimation

Commercial HLS tools are pre-packaged with GUI enabled GNU Debugger (GDB) for verification of the C source. The platform provides user friendly software push-buttons for step-over, step-in, and execution breakpoints to verify the source code by traversing through individual code statements. They also provide a live state of all variables at any point of execution in any convenient radix. We make use of such platforms for functional verification and debugging of the RTL. The *FastSim* tool generates a well arranged C code that mimics the execution of the RTL with cycle level accuracy. This generated RTL-C could be fed to a GUI or Non-GUI based GNU debugger for state-wise/cycle-wise verification of the RTL. Unlike the behavioural input C source, the RTL-C cycle accurately emulates the working of the generated RTL design. *The variables of the RTL-C are the registers of the RTL. So, all the registers are visible at any clock during debugging.* If there is any mismatch of values identified after the execution of an operation, say $r_l = f(r_i, r_j, r_k)$, in a state, say s_x, during debug, the same can be annotated back to the RTL. The s_x represents the state of the controller FSM of the RTL. As discussed in Section 3.3.4, the operation $r_l = f(r_i, r_j, r_k)$ obtained by a spatial sequence of micro-operations in the datapath from the RHS registers r_i, r_j, r_k to the LHS register r_l. Although we have not implemented this back annotation of the mismatch into RTL, it is a trivial task to show a graphical view (for example using dotty format) of the RTL datapath and highlight the transfer of register values for visualization. Hence, cycle-wise execution of our RTL-C code in GDB would reveal all the C source to RTL disparities like artificial deadlocks, output data ordering and FIFO based feedback problems described in [44] which cannot be emulated by the input behavioural C source. However, back annotation of operation of the RTL-C code to the input C code is not possible in our framework so that one can relate a given RT operation to source C. Without some internal information from the HLS tool, it is not possible to co-relate the input C operation and operation in the RTL.

The synthesis report of the commercial and academic HLS tools has failed to provide performance estimation for applications with data-dependent loop bounds or conditional statements. The C simulation performed by commercial HLS tools also does not provide any performance estimates. In contrast, the state-wise execution sequence of our RTL-C code provides an accurate estimation of performance. We use a global counter which is incremented at each state in the RTL-C to evaluate the latency of the RTL design in clock cycles. For data-dependent loop bounds and conditional statements, the execution cycles depend on the input. Therefore, we report the maximum and minimum execution cycles, like RTL co-simulation report, among all test cases used for simulation. In fact, it is also possible to report additional performance results like the number of execution cycles for each module or loop and the number of cycles when a particular FIFO was full/empty using additional tracking variables in our RTL-C code.

Table 3.2: *Characteristics of Benchmarks*

Benchmark	**#Line**	**#If-else**	**#Array**	**#Func**	**#Loop**
aes_dec	949	24	15	13	13
aes_enc	979	24	15	11	13
des	354	1	9	6	11
mips	313	0	5	1	5
dfsub	955	34	1	17	0
dfadd	554	32	1	8	0
dfmul	522	28	1	17	0
arf	53	2	0	0	0
motion	52	0	0	0	0
waka	33	2	0	0	0

3.7 Experimental Results

In this section, we have discussed the implementation detail of FastSim and detailed experimental results.

3.7.1 Experimental Setup and Benchmark Characteristics

We have implemented our proposed FastSim RTL to C conversion framework on the Verilog RTLs generated by the Vivado HLS design suite [10]. However, FastSim can be implemented for any other HLS tool as well. The *FastSim* framework is implemented using Python. The RTL to C conversion is integrated with PyVerilog toolkit [9, 121] to generate the AST representation from the Verilog RTL generated by Vivado HLS. The AST representation is persisted in memory and processed by the *FastSim* to generate the equivalent RTL-C code.

In this section, we compare our *FastSim* simulation framework with Vivado HLS C-simulation [10], Vivado HLS RTL co-simulation (XSIM), ModelSim RTL simulator [8] and

Table 3.3: *RTL to C conversion results for Benchmarks*

Benchmark	#C	#RTL	#RTL-C	Runtime(s)
aes_dec	949	3154	5776	0.334
aes_enc	979	2799	4784	0.237
des	354	2330	2856	0.189
mips	313	1779	5906	0.848
dfsub	955	2203	2856	1.097
dfadd	554	1724	2132	0.646
dfmul	522	2237	2858	0.593
arf	53	351	607	0.010
motion	52	415	780	0.014
waka	33	270	474	0.007

Verilator simulator[11] on the basis of simulation latency and performance estimates. All the designs were synthesized for Kintex 7 series FPGA target [124] clocked at 100MHz. The experiments have been performed on a system powered by Intel Core i7 - 9700KF (3.6 GHz) processor and 16GB DRAM capacity. Our experiments have been performed on several standard example programs from Bambu HLS Tool [2] and CHStone Benchmark Suite [70]. The characteristic description of the benchmark programs used for our experiments is presented in Table 3.2. The 1st, 2nd, and 3rd columns depict the name, the number of lines, and the number of conditional statements respectively in each benchmark. The 4th, 5th, and 6th depict the numbers of arrays, functions, and loops, respectively in each benchmark. Table 3.2 demonstrates the computational diversity of benchmarks used for our experimental analysis. Floating-point addition (*dfadd*), multiplication (*dfmul*) and subtraction (*dfsub*) are control intensive benchmark programs with several conditional statements but no arrays. Whereas larger benchmark programs like *des*, *aes* and *mips* are data intensive programs with several arrays and function calls. We have also considered some smaller benchmarks like arf, motion, and waka for the diversity of benchmark size.

3.7.2 RTL to C Conversion Results

The experimental details on the RTL to C conversion process are presented in Table 3.3 for the benchmark programs. For each benchmark, we record the number of lines of the source C code (#C), Verilog RTL (#RTL), generated RTL-C code (#RTL-C) and the conversion run-time (in second). The number of lines of code in RTL-C and RTL are found

Table 3.4: *Comparisons of FastSim with various RTL Simulators*

Bench mark	Simulation Time (seconds)								
	FastSim	**C-sim**	**Speedup**	**RTL Cosim**	**Speedup**	**Model Sim**	**Speedup**	**Verilator**	**Speedup**
aes_dec	20.176	14.442	0.72x	4467	221.4x	4780	236.9x	316.4	15.7x
aes_enc	19.32	12.656	0.66x	4389	227.2x	4693	242.8x	296.23	15.3x
des	34.43	28.01	0.82x	34672	1007.9x	36024	1047x	723.01	21.1x
mips	1.782	0.985	0.55x	2620	1455.5x	2885	1618.8x	20.4	11.33x
dfsub	0.807	0.717	0.89x	8	10x	14	17.5x	4.117	5.1x
dfadd	0.629	0.561	0.89x	7	11.20x	13	20.7x	3.92	6.2x
dfmul	1.062	1.374	1.29x	10	9.43x	17	16x	7.165	6.8x
arf	0.624	0.601	0.96x	6	9.7x	11	17.7x	3.93	6.33x
motion	0.491	0.565	1.15x	6	12.24x	10	20.4x	3.3025	6.73x
waka	0.411	0.454	1.10x	8	19.46x	11	26.77x	2.72	6.62x
Average			**0.91x**		**298.40x**		**326.45x**		**10.13x**

to be relatively higher in array intensive programs as compared to the non-array based programs. This is justified by the intrinsically large number of complex register transfers in data intensive workloads. As discussed earlier, the generated RTL-C cycle accurately emulates all the register transfer operations in each state. The number of lines in the RTL-C is greatly increased due to copying of each register to an old_variable in each state as discussed in Sub-section3.4.A. Consequently, the number of lines of code in the RTL-C code is much higher than that of source C code. For all the benchmarks, the conversion runtime for generating RTL-C code is found to be less than a 1.1 second. Hence the total time for simulation is still far less than RTL simulators.

3.7.3 HLS Simulation Results

Table 3.4 presents the simulation time and speedup of our *FastSim* framework relative to other state-of-art simulators when experimented on different benchmarks. For each benchmark, we run the simulation for 30k input test cases. We couldn't produce the results of FLASH simulator [44] since the tool is not made public. As reported in [44], FLASH works on scheduled C code and its performance is similar to Vivado HLS C simulator. Hence, we can safely assume that performance of our *FastSim* is comparable to FLASH. It may be noted that our *FastSim* is on average 9% slower than the C-simulation. As suggested by

Table 3.5: *Comparisons of FastSim with various RTL Simulators after applying pipeline (p) and unroll (u) pragmas*

		Simulation Speedup		
Benchmark	FastSim(s)	**RTL Cosim**	**ModelSim**	**Verilator**
aes_dec (u)	31.6	135.13x	142.47x	12.58x
aes_dec (p)	33.57	126.57	136.64x	12.77x
aes_enc (u)	26.07	155.96x	169.12x	14.48x
aes_enc (p)	32.62	127.49x	137.12x	12.58x
des (u)	41.4	697.05x	726.77x	17.4x
des (p)	46.5	797.22x	829.87x	20.35x
Average		**339.90x**	**356.99x**	**15.03x**

experimental results, Verilator is faster than RTL simulations like XSIM or ModelSim. It could be observed that on an average, C simulation (C-sim) offers the best simulation performance, whereas ModelSim and Vivado RTL Cosimulation framework (XSIM) shows the worst performance. The *FastSim* simulator on an average shows comparable performance as that of C simulator (0.91×), performs 298× as fast as XSIM, 326× as fast as ModelSim and 10× as fast as Verilator.

Table 3.6: *Comparisons of results for task level pipelining using ping pong (pp) or FIFO (ff)*

		Simulation Speedup		
Benchmark	FastSim(s)	**RTL Cosim**	**ModelSim**	**Verilator**
toy (pp)	23.7	20.46x	22.3x	6.06x
toy (ff)	26.6	18.73x	20.11x	5.4x
mergsort (pp)	31.2	25.2x	28.6x	8.7x
Insertionsort (ff)	39.5	21.7x	23.8x	7.6x
histogram (pp)	42	23.3x	26.2x	9.4x
FFT (pp)	153.2	117.6x	124.2x	10.1x
Average		**37.8x**	**40.9x**	**7.9x**

Similarly, a comparison of our FastSim simulator with various RTL simulators after applying to unroll and pipeline pragmas on some bigger benchmarks and applying dataflow (FIFO or PIPO) pragma on benchmarks from [84] are shown in Table 3.5 and Table 3.6, respectively. We observe similar performance improvements for the pipeline and unroll. As shown in Table 3.6, FastSim supports both FIFO and PIPO styled task level pipelining. As already elaborated in the introduction, *FastSim* simulates only the relevant RT operations at the behavioural level within a particular state leaving the state-exclusive register states unaltered. Meanwhile, RTL simulators emulate the complete RTL at every clock cycle. This justifies the performance of *FastSim* with respect to RTL simulators. Owing to its generic nature, the Verilator generated code is always suboptimal compared to HLS customized *FastSim* code. Consequently *FastSim* outperforms Verilator. It is encouraging to note that the speed-up achieved for larger benchmarks like des, mips,and aes are much higher than the average. Hence the experimental results substantiate our motivation of approaching the performance C-simulator.

Table 3.7: *Performance estimation in clock cycles by Vivado synthesis report, Fastsim and RTL co-simulation*

Bench	**Vivado Synthesis (Clock cycles)**		**FastSim (Clock cycles)**		**RTL Cosim (Clock cycles)**	
	Min	**Max**	**Min**	**Max**	**Min**	**Max**
aes_dec	?	?	5654	5654	5654	5654
aes_enc	?	?	3006	3006	3006	3006
des	125065	125321	125425	125427	125425	1254257
mips	?	?	3383	3683	3383	3683
dfsub	8	21	9	19	9	19
dfadd	7	20	9	18	9	18
dfmul	8	22	12	20	12	20
arf	7	7	7	7	7	7
motion	6	6	6	6	6	6
waka	2	3	3	3	3	3

3.7.4 Performance Estimation

In Table 3.7, we compare the performance estimates of *FastSim* with respect to different state-of-art simulation frameworks for 30k test cases. As shown in the table, the Vivado HLS synthesis reports fails to provide performance estimates for benchmarks like *mips*, *aes_enc* and *aes_dec* with data dependent loops. On the other hand, our simulation framework predicts the same performance as that of the RTL co-simulator (XSIM). This substantiates our claim that *FastSim* not only simulates faster than the conventional RTL simulators but also gives accurate performance estimates as that of an RTL simulator. These exact performance estimates could be explained as the consequence of cycle accurate simulation performed by *FastSim* simulation framework.

3.8 Conclusion

In this work, we exploited the separability of datapath and controller in the HLS generated RTLs to build a fast simulation framework *FastSim* which simulates around 300 times faster than traditional RTL simulators and exhibits the performance of the same order of magnitude as that of software simulators while maintaining the micro-details of the generated RTL. We have also proved experimentally that *FastSim* outperforms existing solutions for faster simulation like Verilator [11] and FLASH [44] in terms of simultaneously incorporating high simulation performance, maintaining cycle accuracy, end-to-end verification, easy debug facilitation and accurate performance estimation. The simulator is also equipped to support common HLS optimizations like loop unrolling, instruction level pipelining, and task level pipelining. In addition to regular simulation, *FastSim* outputs a well organized C code which could be conveniently used for several objectives like the selective evaluation of the performance of specific segments of the RTL or mount attack on RTL locking [77]. The current version of FastSim cannot handle data width of more than 64 bits and fails to simulate hierarchical task level pipelining and sequence of level triggered operations in a single state. However, in the future, we plan to enhance *FastSim* to overcome all such limitations. Post incorporation of all industrially significant design space exploration features, we plan to issue a complete open source package of *FastSim* for industrial and academic HLS design verification.

4 DEEQ: Data-driven End-to-End EQuivalence Checking of High-level Synthesis

4.1 Introduction

Although High-level synthesis (HLS) tools are an attractive choice for the design houses for quick development of hardware accelerators from behavioral specification, they are not bug-free. Since HLS is a complex software that involves inter-dependent sub-tasks, a bug may be introduced by the tool in handling a corner case or due to incorrect implementation of any sub-task in the tool. Therefore, techniques for establishing an equivalence between a behavioural specification and its synthesized RTL implementation are critical for wide applications of HLS. Although, there are many HLS tools, such as Xilinx Vivado HLS [10] and Siemens Catapault [3], etc. in the industry as well as Spark [67] and Bambu [107], etc. in the academia, are available, there is no end-to-end formal verification tool exist for HLS. The correctness of HLS generated RTL is primarily ensured by RTL simulation. Although RTL simulation relatively accelerates the process of verification and is good at quickly finding errors, it cannot guarantee the complete correctness of generated RTL because of the inherent limitation of simulation-based verification. The alternative approach is the translation validation of HLS [109] in which the correctness of each run of the HLS tool is verified by checking equivalence between the input C/C++ and the generated RTL. Because of the huge semantic gap between the input and the output of HLS, the end-to-end, i.e., C to RTL, translation validation of HLS is not well established yet. Instead, phase-wise translation validation of HLS, such as scheduling verification [46, 87], allocation and binding verification [82] and datapath and controller verification [79], is mostly explored by the researchers. The phase-wise verification of HLS needs the intermediate synthesis results which may not always be available or industrial tools may not want to make them public. Therefore, an end-to-end translation validation of HLS has enormous importance for the wide adaptation of HLS tools.

4.1.1 Contributions

In this Chapter, we propose a C to RTL equivalence checking method, called DEEQ, to prove the correctness of the HLS generated RTL with respect to its input C code. For checking the equivalence of two behaviours, we need to show that for every possible execution (i.e., trace) of one behaviour, there is an equivalent trace in the other behaviour and vice-versa. There are many challenges in C to RTL equivalence checking: (i) The direct trace level comparison is not possible due to the semantic gap between the C and RTL. The execution of C code is completely different from that of RTL. (ii) The control structure of C is altered most of the time during HLS. Therefore, the number of traces in C and RTL may not be the same and the equivalence problem of traces is *many-to-many.* (iii) The number of traces is also large in most cases. Therefore, finding an equivalent trace in RTL for each trace in C needs quadratic comparisons. (iv) The variables of the input C code are mapped to the registers/memories in RTL. This mapping is many-to-many and it is not available at HLS output. Consequently, direct comparison at any intermediate point is also not possible. *Due to these challenges, C to RTL equivalence checking of HLS is an open problem till today even though HLS tools are so popular.* In this work, we have taken the following *unique steps* to address the above identified challenges:

- To tackle the semantic gap, *we first abstract out a high-level C-like behaviour, called RTL-C, from the RTL.* This makes the two behaviours comparable.
- Before checking equivalence, we identify the compatible traces (i.e., traces which have the same outputs) within a behaviour and merged them into one. It *reduces the many-to-many trace equivalence into one-to-one trace equivalence in most of the cases.*

- During equivalence checking, a *data-driven approach* is taken to find the correspondence of traces between two behaviours. This idea reduces the *quadratic search of equivalence traces between two behaviours into a linear one.*

A satisfiability problem is formulated to prove or disprove the equivalence of corresponding traces of both behaviours. Our equivalence checker is tool-independent since it does not need any intermediate information from the HLS tool. Experimental evaluations show that our proposed tool is capable of showing the correctness of a commercial HLS tool [10]. Our proposed framework also successfully detected a known bug [72] in Vivado HLS.

The remainder of the chapter is organized as follows. Section 4.2 presents our equivalence checking Formulation. Equivalence checking between C and RTL-C is presented in Section 4.3. The correctness of the method is presented in Section 4.4. Experimental results and analysis are presented in Section 4.5. Section 4.6 concludes the chapter.

4.2 Equivalence Problem Formulation

In the C to RTL equivalence checking method, the input C and RTL-C are modeled as a finite state machine with datapaths (FSMDs). In the below, FSMDs and the equivalence theory are discussed.

4.2.1 The FSMD Model

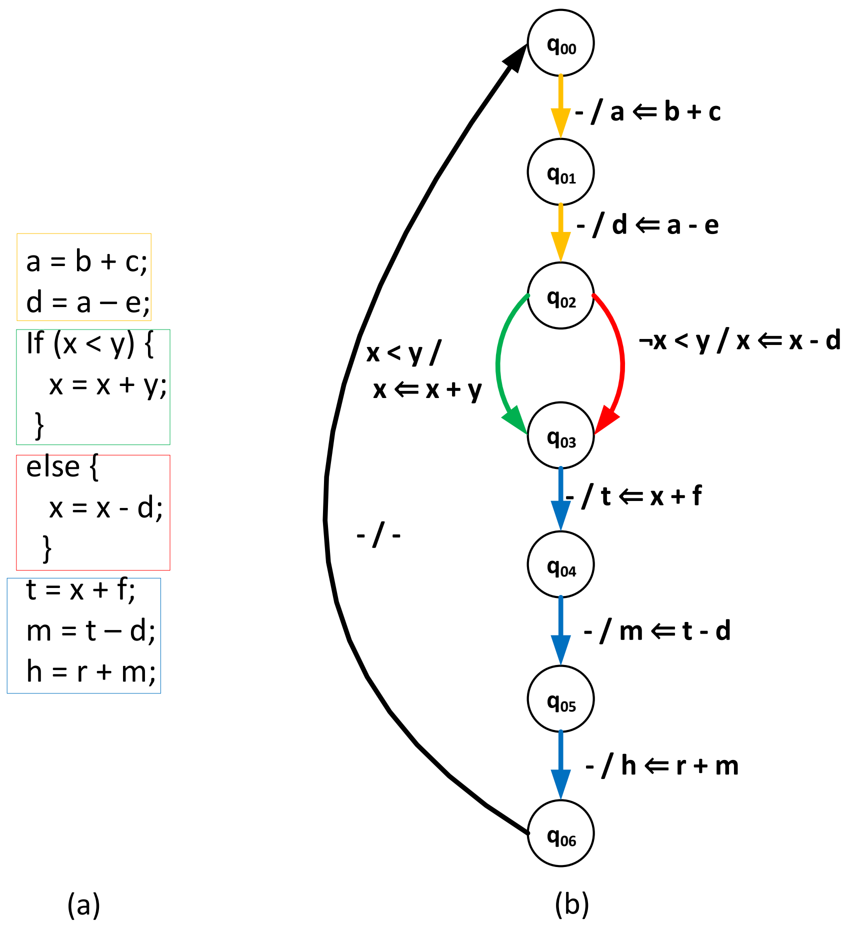
An FSMD is an inherently deterministic model that can describe the behaviour of any hardware circuits [65].

Definition 1. An FSMD M is formally defined as a 7-tuple $\langle Q, q_0, I, O, V, f, h \rangle$, where

- $Q = \{q_0, q_1, q_2, ..., q_n\}$ is the finite set of control states,
- $q_0 \in Q$ is the reset (initial) state,
- I is the finite set of input variables,
- O is the finite set of output variables,
- V is the finite set of storage variables,
- $f : Q \times 2^S \rightarrow Q$ is the state transition function,

- $h : Q \times 2^S \rightarrow U$ is the update function of the output and the storage variables, where S and U are defined as follows.

$-S = \{L \cup E\}$ is the set of status expressions where L is the set of Boolean literals of the form b or $\neg\, b$, $b \in B \subseteq V$ is a Boolean variable and E is the set of arithmetic predicates over $I \cup (V - B)$. Any arithmetic predicate is of the form $eR0$, where e is an arithmetic expression and $R \in \{==, \neq, \geqslant, >, <, \leqslant\}$.
$-U$ is a set of storage or output assignments of the form $\{x \Leftarrow e | x \in O \cup V$ and e is an arithmetic predicate or expression over $I \cup (V - B)$; it represents a set of storage or output assignments.



```
a = b + c;
d = a – e;
If (x < y) {
  x = x + y;
 }
else {
  x = x - d;
 }
t = x + f;
m = t – d;
h = r + m;
```

Figure 4.1: *Example to illustrate FSMD Model*

The FSMD models can be constructed from the high-level representations of the input

C code. Any sequential behaviour consists of a combination of three basic constructs: (i) sequences of statements (Basic Blocks) without any bifurcation of control flow, (ii) if-else constructs (Control Blocks), and (iii) loops. Therefore, capturing these three constructs in an FSMD model enables us to effectively represent any sequential behaviour as an FSMD. The input behaviour basic block consisting of a sequence of n statements $s_1, \ldots, s_n$, can be represented as a sequence of n + 1 states $q_{i,0}, \ldots, q_{i,n}$, say and n edges of the form $q_{i,j-1} \xrightarrow{-/s_j} q_{i,j}$, $1 \leqslant j \leqslant n$, in the corresponding FSMD. However, to reduce the number of states in the FSMD, we first construct a dependence graph for the basic block (BB). The graph consists of a node corresponding to each statement of the BB. There is a directed edge in the dependence graph from the statement s_1 to the statement s_2 iff there is one of read-after-write, write-after-read, and write-after-write dependencies between s_1 and s_2. A Control Blocks (CB) is of the form: if(c) then BB1 else BB2 endif, where c is a conditional statement and BB1 and BB2 are two basic blocks that execute when c is true and false, respectively. The FSMD of the CB is obtained from these two FSMDs by (i) merging the start states of two FSMDs into one start state and the end states of two FSMDs into one end state and (ii) the condition c is placed as the condition of the first transition of the FSMD corresponding to BB1 and the condition ¬c is placed in the first transition of the FSMD corresponding to BB2. The FSMD(s) for other control blocks can be constructed in a similar manner. The RTL-C is already a cycle accurate model. The FSMD will follow the control structure of the RTL-C. In fact, FastSim constructs the FSMD first from the datapath and the controller FSM and then represents the same in C. We will use the FSMD constructed by FastSim.

Example 10. *Consider the C code and its corresponding FSMD shown in Fig. 4.1 (a) and Fig. 4.1 (b), respectively. The FSMD model the behavioural specification of the model is as follows: M is formally defined as a 7-tuple $\langle Q, q_0, I, O, V, f, h\rangle$, where*

- $M = \langle Q, q_0, I, O, V, f, h\rangle$, *where*
- $Q = \{q_{00}, q_{01}, q_{02}, q_{03}, q_{04}, q_{05}, q_{06}\}$,
- $q_0 = q_{00}$,
- $I = \{b, c, e, x, y, f, r\}$,
- $O = \{h\}$,

- $V = \{a, d, t, m\}$,
- $U = \{a \Leftarrow b + c, d \Leftarrow a - e, x \Leftarrow x - d, x \Leftarrow x + d, t \Leftarrow x + f, m \Leftarrow t - d, h \Leftarrow r + m\}$
- $S = \{x < y, \neg x < y\}$,
- *Some typical values of f are as follows: $f(q_{00}, \{true\}) = q_{01}$, $f(q_{02}, \{x < y\}) = q_{03}$, $f(q_{02}, \{\neg x < y\}) = q_{03}$,*
- *Some typical values of h are as follows: $h(q_{02}, \{x < y\}) = \{x \Leftarrow x + y\}$, $h(q_{02}, \{\neg x < y\}) = \{x \Leftarrow x - d\}$, $h(q_{05}, \{true\}) = \{h \Leftarrow r + m\}$,*

□

Definition 2. A *trace* τ of an FSMD is a finite walk from the reset state q_0 back to itself and q_0 does not occur in between.

A (finite) path p of an FSMD is a finite walk from q_i to q_j, where $q_i, q_j \in Q$, is a sequence of state transitions of the form $\langle q_i \xrightarrow{c_i} q_{i+1} \xrightarrow{c_{i+1}} \ldots \xrightarrow{c_{n+1}} q_n = q_j \rangle$ such that $\forall l, i \leqslant l \leqslant n-1$, $\exists c_l \in 2^S$ such that $\mathrm{f}(q_l, c_l) = q_{l+1}$, and all the states are different, except the end state q_j that may be the same as the start state q_i. Therefore, a trace is a concatenation of paths.

Definition 3. The *condition of execution* c_τ of a trace τ is a logical expression over I and constants, which must be satisfied by the initial data state in q_0 to traverse τ.

Definition 4. The *data transformation* s_τ of a trace τ over O is an ordered tuple $\langle e_j \rangle$ of algebraic expressions using I and constants such that the expression e_j represents the value of the output o_j after execution of the trace in terms of input variables.

The condition of execution and data transformation of the path can also be defined in a similar manner. The c_τ and s_τ of a trace τ can be obtained by forward substitution [81].

Example 11. *Let us consider the FSMD in Fig. 4.1 (b). It may be noticed that the FSMD consists of two traces: $\tau_0 = q_{00} \rightarrow q_{01} \rightarrow q_{02} \xrightarrow{x<y} q_{03} \rightarrow q_{04} \rightarrow q_{05} \rightarrow q_{06} \rightarrow q_{00}$ with $c_{\tau_0} = x < y$, and $\tau_1 = q_{00} \rightarrow q_{01} \rightarrow q_{02} \xrightarrow{\neg(x<y)} q_{03} \rightarrow q_{04} \rightarrow q_{05} \rightarrow q_{06} \rightarrow q_{00}$ with $c_{\tau_0} = \neg(x < y)$. The output expression of traces τ_0, and τ_1 are r + (x + y + f -b - c + e), and r + (x + f - 2b -2c + 2e), respectively.* □

4.2.2 Equivalence of FSMDs

Let the input C behaviour be represented by the FSMD $M_0 = \langle Q_0, q_{0,0}, I, V_0, O, f_0, h_0\rangle$ and the RTL-C be represented by the FSMD $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1\rangle$. It may be noted that the inputs and outputs of both behaviours are identical. The V_0 *of* M_0 *consists of the variables in the input C program. The* V_1 *of* M_1 *consists of registers of the RTL behaviour. The state transition function may differ since the control structure may be altered due to the application of compiler transformations or by the scheduler.* Our main goal is to verify whether M_0 behaves exactly as M_1. This means that for all possible inputs, the execution traces of M_0 and M_1 produce the same outputs. So, a trace, which represents one possible execution of an FSMD, takes one assignment of inputs and produces the corresponding outputs. The equivalence of traces and FSMDs are defined as follows.

Definition 5 (Equivalence of traces)**.** A trace τ_0 of an FSMD M_0 is equivalent to a trace τ_1 of another FSMD M_1, denoted as $\tau_0 \simeq \tau_1$, if $c_{\tau_0} \equiv c_{\tau_1}$ and $s_{\tau_0} \equiv s_{\tau_1}$, where c_{τ_0} and c_{τ_1} represent the conditions of execution of τ_0 and τ_1, respectively and s_{τ_0} and s_{τ_1} represent the data transformations of τ_0 and τ_1, respectively.

Definition 6 (Containment of FSMDs)**.** An FSMD M_0 is said to be contained in an FSMD M_1, symbolically $M_0 \sqsubseteq M_1$, if $\forall \tau_0 \in M_0, \exists \tau_1 \in M_1$ s.t. $\tau_0 \simeq \tau_1$.

Definition 7 (Equivalence of FSMDs)**.** Two FSMDs M_0 and M_1 are said to be computationally equivalent, i.e., $M_0 \simeq M_1$, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$.

For deterministic model, one way containment implies equivalent, i.e., $M_0 \sqsubseteq M_1 \implies M_0 \simeq M_1$ since the union of conditions of execution of all traces in M_0/M_1 is True [46].

4.3 Equivalence Checking between C and RTL-C

There can be many traces in an FSMD. Therefore, the notion of path cover is introduced for verification of scheduling in [81]. Consequently, a set of path-based equivalence checking (PBEC) methods are proposed [24, 46]. The idea is to insert cutpoints in the FSMDs to break them into a finite set of paths such that any trace can be represented by a concatenation of paths from that set. All loops of the behaviour must be cut by a cutpoint. The finite set of paths between cutpoints without having any intermediate cutpoint is called *path cover* of the FSMD. The equivalence is then established between the path covers of

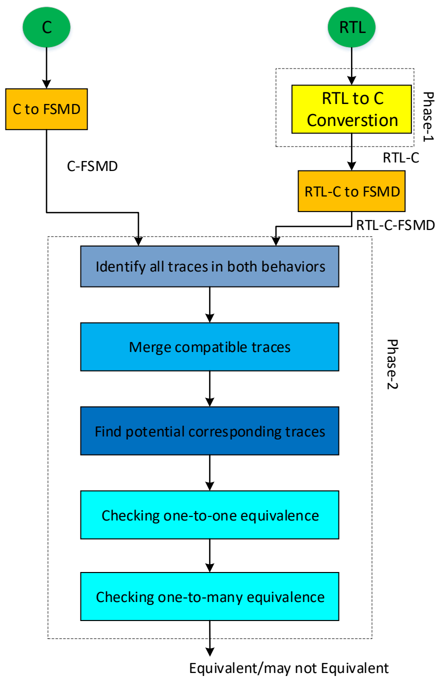

C
RTL
C to FSMD
RTL to C Converstion
Phase-1
C-FSMD
RTL-C
RTL-C to FSMD
RTL-C-FSMD
Identify all traces in both behaviors
Merge compatible traces
Find potential corresponding traces
Checking one-to-one equivalence
Checking one-to-many equivalence
Phase-2
Equivalent/may not Equivalent

Figure 4.2: *Proposed equivalence checking framework*

two FSMDs. Consequently, $M_0 \sqsubseteq M_1$ can be redefined as if there exists a finite cover $P_0 = \{p_{00}, p_{01}, \ldots, p_{0l}\}$ of M_0 for which there exists a path cover $P_1 = \{p_{10},\ p_{11}, \ldots,\ p_{1l}\}$ of M_1 such that $p_{0i} \simeq p_{1i}$, $0 \leqslant i \leqslant l$. Therefore, showing the equivalence of path cover is sufficient for verification of scheduling. However, these PBEC methods either extend a path [81] or propagate mismatch values to subsequent paths [24] when equivalence of paths cannot be shown. These path extensions or value propagation may be carried out till the reset state in the worst case. Therefore, all PBEC methods have exponential time complexity in the worst case [81].

We cannot apply the PBEC based approaches for checking equivalence between C and RTL-C because V_0 of M_0 (corresponding to input C code) and V_1 of M_1 (corresponding to RTL-C) have a *completely different set of variables*. Specifically, V_0 consists of the variables of the input program, and V_1 consists of the registers of the RTL. Consequently, the equivalence of intermediate paths cannot be shown since there are no common variables among M_0 and M_1. Therefore, PBEC methods always need exponential time for our problem. We, therefore, stick to showing equivalence between traces of two FSMDs.

To show the trace level equivalence between C and RTL-C, we need to identify all the traces in an FSMD first. Since a loop with a dynamic bound may result in an infinite number of traces, we assume that each loop in a behaviour has a static loop bound. Static loop bounds for generic program equivalence are quite restrictive. However, HLS tools do not support dynamic memory allocation. Specifically, the dynamic arrays are replaced with a static sized arrays in the input program before applying HLS [10]. The loops are primarily used for array manipulation in a program. Therefore, most of the loops in HLS applications have static bounds. Although static loop bounds for generic program equivalence are restrictive, our assumption of static loop bound is a valid one in translation validation of HLS.

The overall flow of the proposed equivalence checking approach is demonstrated in Fig. 4.2. The equivalence between the C-code and its corresponding RTL generated by HLS is established in two phases: In the first phase, we abstract out a high-level C-like behaviour, called RTL-C, from the RTL using our FastSim tool. In the next phase, equivalence checking between input C code and RTL-C is carried out.

Our proposed equivalence checking method DEEQ, given as Algorithm 2, takes two FSMDs – input C and RTL-C, representing the FSMDs of input C-code and the corresponding RTL-C obtained from the RTL at the output of HLS tools, respectively. The algorithm examines whether the trace-pairs of input-C and RTL-C are equivalent or not. The algorithmic implementation details are described below.

4.3.1 Generate all Traces in Both Behaviors

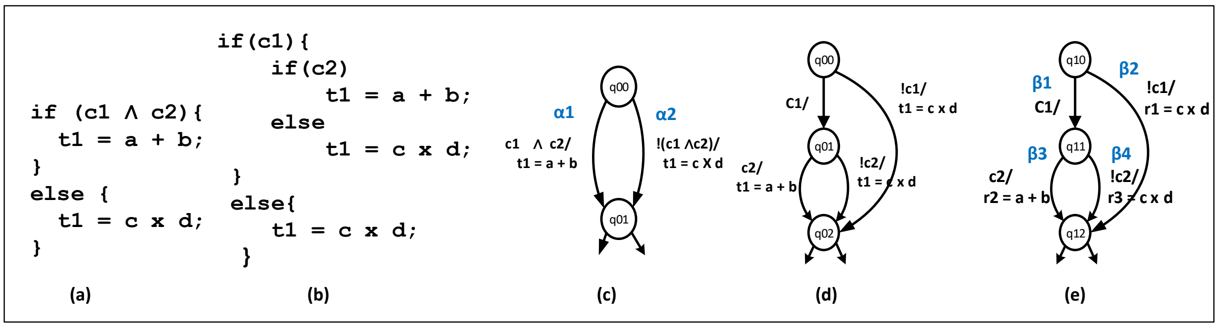
The algorithm first uses the function *constructFSMD* to automatically extract the FSMDs M_0 and M_1 from the input C and RTL-C behaviours, respectively. In order to compare these two behaviours, we need to find all traces in both behaviours. The function *findTrace* (line 1) first constructs the FSMDs from C and RTL-C and then extracts all the traces and assigns to the sets T_0 and T_1, respectively. We have used Klee [7] for this purpose. Specifically, Klee generates all the possible traces in the control flow graph of both behaviours along with the constraints on inputs which are to be satisfied to get that trace. To get the symbolic data transformation and the condition of execution of the traces, we have modified Klee's source code. We assume that the loops in the behaviours are of fixed bound which can be determined at the compile time. Therefore, the number of traces is finite in our

Algorithm 2: DEEQ (C, RTL-C)

```
Input: Input-C, RTL-C
Result: Equivalent, May not equivalent
1  $T_0$ = findTrace(C); $T_1$ = findTrace(RTL-C);
2  $T_0$ = mergeTrace ($T_0$); $T_1$ = mergeTrace ($T_1$);
3  $copyT_0 = T_0$; $copyT_1 = T_1$;
4  while $T_0 \neq \phi$ do
5      $\tau_0$ = select a trace from $T_0$;
6      TC = getTestcase($\tau_0$);
7      $\tau_1$ = getCorrespondingTrace($T_1$, TC);
8      if $((c_{\tau_0} \equiv c_{\tau_1}) \wedge (s_{\tau_0} \equiv s_{\tau_1}))$ then
9          //$\tau_0$ and $\tau_1$ are equivalent;
10         removeTrace ($\tau_0$, copyT$_0$);
11         removeTrace ($\tau_1$, copyT$_1$);
12     else if $((c_{\tau_0} \equiv c_{\tau_1}) \wedge (s_{\tau_0} \neq s_{\tau_1}))$ then
13         Report "Not Equivalent (NEq)" and Exit;
14     endif
15     removeTrace ($\tau_0$, $T_0$);
16 endwhile
17 if $(\|copyT_0\| \neq NULL)$ then
18     foreach $\tau_0 \in copyT_0$ do
19         $c_\tau = \phi$;
20         foreach $\tau_1 \in copyT_1$ do
21             if $(c_{\tau_0} \wedge c_{\tau_1} \neq \phi)$ then
22                 if $(c_{\tau_0} \wedge c_{\tau_1} \wedge s_{\tau_0} \neq s_{\tau_1})$ then
23                     Report "Not Equivalent" and Exit;
24                 else
25                     $c_\tau = c_\tau \vee c_{\tau_1}$;
26         if $(c_\tau \neq c_{\tau_0})$ then
27             Report "Not Equivalent" and Exit;
28 Report Equivalent (Eq);
```

case. *However, the number of traces may not be the same in the input C and the RTL-C. Specifically, the scheduling of conditional behaviours may change the number of traces.* An example of such a case is given below.

Example 12. *Consider the input C-code, transformed C-code and their corresponding FSMDs shown in Fig. 4.3. For efficient scheduling (hardware reuse) in HLS [105], the input*



```
if (c1 ∧ c2){
  t1 = a + b;
}
else {
  t1 = c x d;
}
```

```
if(c1){
  if(c2)
    t1 = a + b;
  else
    t1 = c x d;
}
else{
  t1 = c x d;
 }
```

Figure 4.3: *(a) The input C and (b) The transformed C for efficient scheduling (c) FSMD (M_0) of input C, (d) FSMD of transformed C, (e) FSMD (M_1) of RTL-C*

behaviour in Fig. 4.3(a) is transformed into the equivalent one in Fig. 4.3(b), where the condition c1 $\wedge$ c2 has been split. The FSMD in Fig. 4.3(e) represents the RTL-C obtained from the RTL generated by HLS tool Bambu [107]. As a result, the FSMDs of Fig. 4.3(c) and Fig. 4.3(e) do not have the same number of traces. It may be noted that the input C code consists of two traces α_1 and α_2, as shown in its FSMD M_0 in Fig. 4.3(c), whereas the RTL-FSMD in Fig.4.3(e) has three traces: $\beta_1\beta_3$, $\beta_1\beta_4$ and β_2. □

4.3.2 Merge Compatible Traces

As discussed, the number of traces may not be the same in both behaviours due to various transformations in HLS. To improve performance of equivalence checking, traces which have same output expression, i.e., compatible traces, in each behaviour are merged.

Definition 8. Two traces $\tau_0 = \langle c_{\tau_0}, s_{\tau_0} \rangle$ and $\tau_1 = \langle c_{\tau_1}, s_{\tau_1} \rangle$ of an FSMD M_0 are *compatible* iff $s_{\tau_0} \equiv s_{\tau_1}$, where c_{τ_i} and s_{τ_i} represent the condition of executions and the data transformations of τ_i, $i = 0, 1$, respectively.

The merge trace is represented as $\tau = \langle c_\tau, s_\tau \rangle$ where $c_\tau = c_{\tau_0} \vee c_{\tau_1}$ and $s_\tau = s_{\tau_1}$. Function *mergeTrace* in Algorithm 2 (line 2) merges compatible traces in T_0 and T_1.

Example 13. *Consider the FSMD of RTL-C shown in Fig. 4.3(e). It may be noted that traces $\beta_1\beta_4$ and β_2 consist of the same output expression $c \times d$. In the search for compatible trace, the trace $\beta_1\beta_4$ with $c_{\beta_1\beta_4} = c1 \wedge \neg c2$ and trace β_2 with $c_{\beta_2} = \neg c1$ are merged to a new trace β_n. The condition of execution of β_n is $c_{\beta_1\beta_4} \vee c_{\beta_2}$, i.e., $c_{\beta_n} = (c1 \wedge \neg c2) \vee \neg c1 =$*

$\neg(c1 \wedge c2)$. *As a result Fig. 4.3(e) will have the following new traces :* $\beta_1\beta_3 = q_{10} \xrightarrow{c1 \wedge c2} q_{12}$ *and* $\beta = q_{10} \xrightarrow{\neg(c1 \wedge c2)} q_{12}$. □

Merging compatible traces reduces the trace count within C/RTL-C, makes the trace count the same in both C and RTL-C in most of the cases; hence, enables one-to-one equivalence checking using a data-driven approach. In that case, the equivalence of traces can be shown efficiently, i.e., with $O(n)$ comparisons, where n is the number of traces. The `while` loop in lines 4-16 is used for this purpose. In some corner cases, function *mergeTrace* could not merge the compatible traces because of complex control transformations by the HLS tool. The output expressions of two traces are composite in such a case i.e. the output expressions are not equivalent for the entire input domain but are equivalent for a subset of the input domain. In such case, our method needs one-to-many equivalence checking (lines 17-27 of Algorithm 2). If the number of traces in T_0 and T_1 are not the same even after merging compatible traces (denoted by the $|copyT_0| \neq NULL$ in line 17 of Algorithm 2), a trace in T_0/T_1 is equivalent to a set of traces in T_1/T_0. In this case, $O(n^2)$ comparisons is needed to find the equivalence, where n is the number of traces.

4.3.3 Find Potential Corresponding Traces

After merging compatible traces, we need to find out which trace in M_0 is equivalent to which trace in M_1. i.e., finding corresponding traces between M_0 and M_1. To reduce complexity, *a data-driven* approach is taken to find the potential corresponding traces between T_0 and T_1 using Klee [7] first. Klee gives a test case for each trace in a behaviour. Hence, we know the values of input variables (test case) for each trace τ_0 in T_0. Now, we can run M_1 with this test case and find the trace τ_1 which is followed for this particular test case. Lines 5-7 of Algorithm 2 implements this idea.

Example 14. *For example, for the trace* $\tau_0 = \alpha_1$ *in Fig. 4.3(c), Klee gives* $c1 = 1$ *and* $c2 = 1$ *(and also the values of* a, b, c, d*). By using this test case, we find the trace* $\tau_1 = \beta_1\beta_3$ *is the potential corresponding trace of* τ_0. □

Since the behaviours are deterministic, it is always possible to obtain τ_1 using our data-driven approach. In general, to find the equivalence of trace between M_0 and M_1, we have to take a trace τ_0 in T_0 and compares it with each trace τ_1 in T_1. This comparison of two traces involves checking the equivalence of their respective data transformations and condition of executions which are symbolic expressions. Such formal equivalence checking relies on the modern day SMT solvers which is a time consuming process. Moreover, for finding equivalence of τ_0 in T_1, we need $O(n)$ comparisons of traces, where n is $|T_1|$. With our strategy of finding potential corresponding trace τ_1 in T_1 for τ_0 first using the data-driven approach, we need to only check the formal equivalence between τ_0 and τ_1. This reduces the complexity of finding the equivalence of τ_0 in T_1 to one comparison of traces.

4.3.4 Checking One-to-one Equivalence

In our approach, we first identify the one-to-one equivalence among the traces of T_0 and T_1 in the `while` loop in lines 4-16 of Algorithm 2. A potential corresponding trace pair are equivalent if their respective condition of executions and data transformations are equivalent (in lines 8-11). If the condition of executions match exactly but the data transformations are not equivalent, we report the non equivalence (in lines 12-13). We have used SMT solver Z3 for checking equivalence of data transformations and condition of execution of executions two traces. At the end of the while loop, the traces for which one-to-one equivalence cannot be shown remain in *copy*T_0 and *copy*T_1, respectively. For those traces, we check for a possible one-to-many equivalence in lines 17-27 of Algorithm 2.

Example 15. *The trace $\tau_0 = \alpha_1$ and the trace $\tau_1 = \beta_1\beta_3$ are potential corresponding traces in Fig. 4.3(c), with* c1 $= 1$ *and* c2 $= 1$. *(as shown in example 14). The condition of execution of traces τ_0 and τ_1 are same (c1 $\wedge$ c2). The data transformations of both traces are also equivalent $a + b$. As a result, the potential corresponding traces τ_0 and τ_1 are equivalent.* □

4.3.5 Checking One-to-many Equivalence

For each trace τ_0 in *copy*T_0, we identify the traces in iterative manner in *copy*T_1 that have overlap (common) conditions with τ_0. If the outputs of any of such trace are not equivalent with τ_0, we report the non equivalence in lines 17-23 of Algorithm 2. For each trace τ_0 in *copy*T_0, we identify all traces in *copy*T_1 that have same data transformation as that of τ_0. The union of the condition of executions of all such traces must be equivalent to the condition of execution of τ_0 (in lines 28). In this way, one trace may be equivalent to more than one trace of other behaviour. Since the behaviours are deterministic, the union of the condition of executions of all such traces must be the same as the trace of the other behaviour.

The trace-wise equivalence of potential correspondent traces is checked using SMT solver Z3 [54] in this work. The equivalence problem of two traces is modeled as the satisfiability problem. *Although the SMT expects the program to be in the static single assignment (SSA) form [32], we do not need to convert the C and RTL-C into SSA.* We do not use each line of the program in the SMT formulation. Instead, we compute the condition of execution and data transformation of a trace and use them in SMT formulation. Therefore, SSA conversion is avoided in our formulation. If the SMT solver returns SAT, it means that the corresponding formulas are not equivalent.

4.4 Correctness

In this Section, we will discuss the termination, soundness, and completeness of our method.

4.4.1 Termination

Theorem 4.4.1. The Algorithm 2 always terminates.

Proof. The set T_0 has finite number of traces. In each iteration of the while loop (lines 4-16) in Algorithm 2, the algorithm either reports a possible non-equivalence or the equivalent of the trace τ_0 of T_0 is found in T_1 and τ_0 and its corresponding equivalent trace τ_1 have been removed from T_0 and T_1 (in lines 10 and 11), respectively. The selected trace τ_0 will not be considered again in other iterations of the while loop because of the elimination of one trace from T_0 imposed in line 15. So, the while loop always terminates. Similarly, *copy*T_0 and *copy*T_1 also have finite traces and the `for` loop (lines 17-23) uses one trace in each iteration. Therefore, the `for` loop also terminates. Therefore, Algorithm 2 cannot execute the loops (while and for) infinitely long. Hence, the algorithm always terminates. □

4.4.2 Soundness

Theorem 4.4.2. If Algorithm 2 terminates at line 28, then C and RTL-C are equivalent.

Proof. If Algorithm 2 terminates through the line 28, it indicates that none of non-equivalent scenarios (i.e., line 13, line 23 and line 27) arise. Since the Algorithm 2 always terminates, it reports an equivalence of C and RTL-C when it terminates through the line 28. We have to show that C and RTL-C are indeed equivalent in this scenario. The sets T_0 and T_1 contain the set of traces in C and RTL-C, respectively. Two scenarios may arise in this case.

- *Scenario 1: (one-to-one equivalence)* This scenario arises when the *copy*$T0$ is NULL at the end of the while loop (lines 4-16). Specifically, Algorithm 2 finds that for each trace in τ_o in T_0, the potential corresponding trace τ_1 in T_1 is actually equivalent. Their respective condition of executions and data transformations are formally shown be equivalent using theorem proving. In case of equivalence, both the set contain equal number of traces. In this case, it is ensured that for each trace in T_0, there exists an one-to-one equivalent trace in T_1.

- *Scenario 2: (Both one-to-one and one-to-many equivalences)* This scenario arise when the *copy*$T0$ is not NULL at the end of the while-loop (lines 4-16). In this case, $\|T_0\| \neq \|T_1\|$. For a subset of traces in T_0, one-to-one equivalence is shown with T_1 in the while loop. For each τ_0 in the rest of the trace in T_0, a subset of traces are identified in T_1 such that the data transformation of each of them is equivalent with τ_0 and the union of the condition of executions of them are equivalent with the condition of execution of τ_0. This one-to-many equivalence is being shown in lines 17-27. When this for-loop is completed and the control reaches the line 28, it is ensured that for each trace of T_0, either there exists one trace in T_1 such that they are one-to-one equivalent or there exists a set of traces in T_1 such that their one-to-many equivalence is proven.

What remains to be proved is that there is no trace left in T_1 in both the scenarios. This is ensured since the RTL-C is a deterministic model. As discussed above, all the traces of T_0 are already covered. So, their union of the condition of executions is True. For each trace of T_0, the equivalent trace(s) is/are found in T_1. So the union of condition of executions of all such corresponding traces in T_1 is also True. Therefore, if Algorithm 2 exits through line 28, the algorithm ensures that C and RTL-C are equivalent. The control is reached here only when the equivalence of all the traces in T_0 and T_1 is formally proven using an SMT solver. Hence, the Algorithm 2 is sound. □

4.4.3 Completeness

Our equivalence checking method has two phases: RTL-C extraction from RTL and equivalence checking between C and RTL-C. The RTL-C extraction relies on the *rewriting method* which is proved to be sound and complete in [79]. Therefore, it is always possible to abstract RTL-C from the HLS generated RTL and the RTL-C is functionally equivalent to the RTL.

The equivalence problem of two programs is in general undecidable. The undecidability arises from two facts: (i) parameterized loop bound: The number of traces will be infinite in such a case. Since the loop bounds are static in our case, this situation will not arise. (ii) Arithmetic logic: Checking the equivalence of two traces involves arithmetic logic which involves the whole of integer arithmetic. Therefore, checking the equivalence of traces reduces the validity problem of a first-order logic which is, in general, undecidable. All the variables in C/RTL-C in HLS are bit-precise and finite due to the fixed width of the datapath in the hardware implementation. So, the logic of bit-vectors should be applicable here which is decidable. However, we model all the variables as *unsigned long int* in RTL-C and use a logical AND operation to truncate the not-relevant part of the variable. For example, we model 3-bit variables v as *unsigned long int* and then AND with 7 (i.e., 111) to use the 3-bits of it. Therefore, the underlying SMT theory used to encode the equivalence problem of traces is undecidable. The SMT solver may return unknown or time out in such cases. Therefore, our equivalence checking method is sound and not complete.

4.5 Experiment Results and Analysis

Implementation Detail: The end-to-end equivalent checking framework of HLS DEEQ is implemented in Python and is tested on a set of HLS benchmarks. The Vivado HLS tool [10] is used to generate Verilog RTL for the benchmarks written in C. The benchmarks are taken from [107]. We have then used the pyVerilog [9] parser to extract the abstract syntax tree (AST) from the Verilog and then implemented the rewriting method to obtain the RTL-C. Specifically, we have adapted FastSim [15] to generate the RTL-C from Verilog. The RTL-C generation time is less than 5ms for all cases. All traces of both the behaviours (C and RTL-C) are obtained by running Klee [7]. Klee also gives a test case corresponding to each trace. We forced Klee to get expressions for the output variables in smtlib format. The output equivalence of two traces is formally verified using the SMT solver Z3 [54]. We have also used Z3 to identify compatible traces for merging. The experiments have been performed on a machine with a CPU: Intel Core i7, 2.5GHz, and 8GB RAM.

Experiments: The experiment results of our benchmarks are shown in Table 4.1. The 2^{nd} (#in) and 3^{rd} (#out) show the number of inputs and outputs for each benchmark, respectively. We have recorded the number of code lines (#line) and variables (#var) of the input C, RTL, and RTL-C in the next six columns. The number of lines in the RTL-C

Table 4.1: *Experimental results for different high-level synthesis benchmarks*

Bench marks	#in	#out	C code		RTL code		RTL-C		Traces			Equivalent		Not Equivalent	
			#line	#var	#line	#regs	#line	#var	#C	# RTLC	#merged	time(s)	result	time(s)	result
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)
Waka	20	3	33	21	270	12	474	126	3	4	(3, 3)	1.709	Eq	0.669s	NEq
Arf	11	4	53	43	351	19	607	158	4	4	(3, 3)	1.890	Eq	0.949	NEq
Parker	6	1	51	14	188	2	330	100	12	23	(2, 2)	1.614	Eq	0.976	NEq
Find Min8	8	1	40	15	175	11	780	243	128	128	(8, 8)	22.246	Eq	17.141	NEq
Matrix Add	2	1	48	7	734	44	2595	241	1	1	(1, 1)	1.684	Eq	0.749	NEq
Sum Array	1	1	19	4	263	15	541	100	1	1	(1, 1)	0.754	Eq	0.706	NEq
Motion	10	3	52	43	415	29	780	235	1	1	(1, 1)	0.681	Eq	0.663	NEq
Dfadd	2	1	554	70	1975	113	2132	650	67	68	(21,42)	1016.05	Eq	960.23	NEq

code is high as compared to the RTL since each register is copied in a temporary variable at the start of each state to maintain the concurrent execution of operations of hardware in C [15]. The 10^{th}, 11^{th} and 12^{th} columns are the number of traces of the input C (#C), RTL-C code (#RTLC) and merged traces (#merged), respectively. Equivalence checking results in terms of the total time spent are given in the 13^{th}. As shown, our tool successfully established the equivalence (Eq) in all cases.

In our further experiment, we have created some non-equivalent scenarios from the RTL-C by swapping if-else conditions, changing operation type, or adding/deleting some operations. For example, one of the operation tmp6_reg_362 = (in7 + add5_reg_305_temp) taken from motion is intentionally replaced by tmp6_reg_362 = (in7 * add5_reg_305_temp); clearly these two operations are not equivalent. In this scenario, our equivalence checking method reported non-equivalent. In all cases, these errors are correctly identified by our proposed method as shown in the 15^{th} (time) and 16^{th} (result) columns. Our tool reports non-equivalence (NEq) in all cases since Z3 returns SAT with a witness of violation. Our framework can prove the equivalence of all eight benchmarks successfully within two seconds except for findMin8 and Dfadd. For findMin8, merging of compatible traces is taking almost 15 seconds. For Dfadd, the number of traces after merging is not equal. As a result, equivalence checking takes more time.

Result Analysis: There are several interesting scenarios that arise during our experiments: (i) For FindMin8, there are many compatible traces found since one of the 8 inputs will be the minimum. As a result, 128 traces are merged into only 8 traces. Therefore, *merging compatible traces reduce the complexity in equivalence checking.* (ii) For Parker and Waka examples, the number of traces becomes the same after compatible trace merging resulting in improving verification complexity. *None of the existing techniques [88] can handle such scenarios of equilavence.* (iii) In Dfadd, the number of traces is not equal even after merging. However, equivalence is proved by our method. (iv) Except dfadd, equivalence is established efficiently, i.e., in the while loop of Algorithm 1 due to merging compatible traces.

Comparisons: Although a few end-to-end HLS verification methods like VTV [88] and V2C [100] are available, *the C to RTL verification for HLS is still an open problem.* We found that V2C generates incorrect C code from the Verilog generated by the Vivado HLS tool. So, it can't be used for our purpose. The VTV is the closest to our work which is also applied to verify the results of Vivado HLS. However, VTV fails to show the equivalence when the number of traces is not the same in C and corresponding RTL since it does not support optimizations that alter the control structure. *Therefore, VTV will fail in three (i.e., waka, Parker, and dfadd) of our test cases.* This shows the novelty of our proposed method over the state-of-the-art techniques.

Usefulness: The benchmarks consist of complex if-else (findMin8), loops and arrays (matrixAdd, sumArray), complex non-linear arithmetic operations (arf and motion), and function calls (Dfadd). Also, our method works for non-equivalent cases. Therefore, DEEQ is useful in the verification of HLS results for a commercial HLS tool. The scalability of the method is demonstrated with a relatively larger benchmark having around 3K lines of RTL-C code (Dfadd). Since our method merges many compatible traces before checking equivalence and uses a data-driven approach to find the correspondence of traces, the method is expected to scale well for larger benchmarks as well. We have also tested on a recent bug reported in the Vivado HLS tool in [72] that produces the wrong result during RTL simulation in Xilinx Vivado HLS v2017.2. In this test case, a large integer value is shifted repeatedly by the values stored in an array. For this example, *our equivalence checker reported the non-equivalence.* We have tested with FastSim as well and it reports the non-equivalence with output for source code as 73741823 and for the RTL-C as 6632959. *Therefore, our tool can detect the reported bug in Vivado HLS.* Thus, experimental results show the applicability and scalability of our method for a wide variety of applications.

4.6 Conclusion

We presented an RTL to C translation validation framework for verification of HLS results. The innovative part of our framework to improve the efficiency of equivalence checking are (i) extraction of a C like behaviour from the RTL to reduce the semantic gap between C and RTL, (ii) merging compatible traces within a behaviour to handle control structure modification during HLS and (iii) use of a data-driven approach to find the potential equivalent trace pairs between two behaviours. The equivalence of potential equivalence traces is then formally proved using an SMT solver. The experimental results for a commercial HLS tool for several HLS benchmarks show that our method can efficiently check the equivalence automatically without taking any information from the HLS tool. To the best of our knowledge, *DEEQ is the first completely automated framework for C to RTL equivalence checking for HLS without taking any input from the HLS tool.* In the future, we plan to use techniques like invariant-sketching and query decomposition of the SMT formula [68] to further improve the scalability of our equivalence checker. DEEQ means 'the ruler' in Arabian. DEEQ has all the potential to rule the HLS verification.

5 REVAMP: Reverse Engineering Register to Variable Mapping in High-Level Synthesis

5.1 Introduction

In embedded system design, hardware acceleration is the use of application-specific hardware (accelerator) especially made to perform some functions more efficiently and have a dramatic impact on the speed of critical operations. The HLS is an attractive choice for the algorithm developers for hardware accelerator development. However, understanding detailed implementations and functions that operate by accelerators in the form of hardware description languages like Verilog are not an easy task for algorithm developers. The C equivalent of the RTL code for accelerator would be helpful for the algorithm developers to understand the design structure, the impact of certain HLS optimizations, analyze the output of the accelerator and hence use the HLS tool meaningfully.

To reverse engineer an equivalent C code from the HLS generated RTL meaningfully, one important step is to decode the mapping between variables in scheduled C code and registers in the RTL or the mapping between variables in input C and registers in the RTL generated by HLS. Finding such register to variable mapping is challenging because many to many mapping exist between them. It may be noted that the variables are mapped registers during the allocation and binding phase of HLS. *The number of registers in the RTL usually is much less than the number of variables in the scheduled C (SD-C) or input C. One register may store more than one variable in different time steps provided their lifetimes do not overlap. Similarly, one variable may be split into more than one register for better mapping. Therefore, the relation between registers in the RTL and the variables in scheduled C and input C is complex and it is a challenging task to recover such mapping without taking any information from the HLS tool.* In this work, we consider two problems (i) finding the relation between the variables in input C and the registers in RTL and (ii) finding the relation between the variables in scheduled behaviour (i.e., SD-C) and the registers in the RTL. In this work, we develop reverse engineering framework a REVAMP for both the problems.

5.1.1 Contributions

The first contribution in this chapter is to identify the mapping between variables in SD-C and registers in RTL-C based on an invariant generation tool Daikon. In our approach, we first obtain *SD-C* and *RTL-C* from the scheduling information generated by the HLS tool and the output RTL, respectively. Since the control structure of behaviour is not modified after scheduling, both SD-C and RTL-C programs contain the same numbers of states and state transitions. We use Daikon to find invariants at each state in a program. Since Daikon [4] finds invariants in a program, we combine state-wise *SD-C* and *RTL-C*. From the outputs of Daikon, we extract the invariants in which there is an equality relation between a variable of *SD-C* and a register of *RTL-C*. With this mapping information, we can rewrite the *RTL-C* in terms of variables in SD-C and finally generates an equivalent scheduled C-code from the RTL-C. The working of the proposed method is demonstrated on the RTLs generated by the Vivado HLS tool [10].

The second contribution of this chapter is the extraction of the mapping between the variables in input C and the registers in RTL by using an SMT solver. In our approach, we first extract a high-level behaviour (RTL-C) from the RTL. We then modeled both the input C code and RTL-C as a finite state machine with datapaths (FSMD), called *C-FSMD* and *RTL-FSMD*, respectively. Both the *C-FSMD* and *RTL-FSMD* are then converted into to static single assignment (SSA) form. A Satisfiability Modulo Theory (SMT) based satisfiability problem is then formulated to obtain the variable to register mapping. With this mapping information, we can rewrite the *RTL-C* in terms of variables of the input C and finally generates an equivalent C-code from the RTL. The working of the proposed method is demonstrated again on the RTLs generated by Vivado HLS tool [10]. To the best of our knowledge, *this is the first attempt to automatically reverse engineers the register to variable mapping in HLS.*

The rest of the chapter is organized as follows. Section 5.2 presents details of reverse engineering register to variable mapping using Daikon including challenges faced and experimental results. The details of SMT based register to variable reverse engineering method including experimental results are presented in Section 5.3. Section 5.4 present applications of our framework. Comparisons of Daikon and SMT based register to variable reverse engineering are presented in Section 5.5. Section 5.6 concludes the chapter.

5.2 Daikon based Reverse Engineering of Register to Variable Mapping

As discussed in the Introduction, the number of registers in the RTL is usually less than the number of variables in the SD-C since the values of all the variables are not stored in the registers in all states of the FSM. However, the control structure of the controller FSM of the RTL and the SD-C are exactly the same. Therefore, the task is to identify the content of each register in terms of variable in each state of the controller FSM.

Daikon [4] is software that dynamically detects likely invariants. The invariant is a property that holds at certain points in the program. Daikon observes the values of the variables at a particular point or points in the program and reports the relationship between the variables. If the values of two variables are equal at a point in all the test cases then the relationship between these variables is equal at that point. Based on values, a relation are established between the variables or between two equations.

5.2.1 Reverse Engineering Steps

In this approach, we propose a two-phase register to variable mapping framework. In the first phase, we abstract out a high-level C-like behaviour, called RTL-C, from the RTL. This RTL-C is nothing but a high-level C code in terms of the inputs, constants, registers/memories, and outputs in the RTL. In the next phase, register to variable mapping is carried out. The

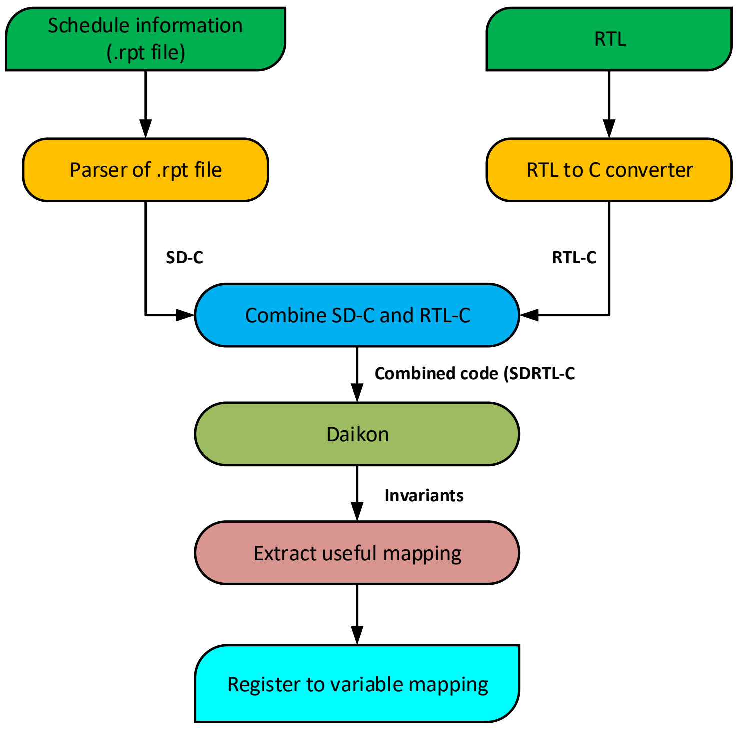


Figure 5.1: *Daikon based register to variable reverse engineering flow*

overall flow of our method framework is demonstrated in Fig 5.1. Our method takes two input behaviours – schedule information and RTL-C. The steps are discussed below with help of an example.

5.2.1.1 RTL-C Abstraction from RTL

The main technical issues addressed in this phase are the following: "Given a synthesizable RTL design generated by the HLS tool, generate a sequential C code that preserves the RTL semantics". We have used our RTL-to-C conversion technique discussed in Chapter 3 for this purpose. It may be recollected that the HLS generated RTL has a datapath and a controller FSM. In each FSM state, the controller assigns 1/0 values to the control signals to execute a specific set of RTL operations in the datapath. Our objective is to identify the RTL operations performed in each state of the controller by this control assignment. The RTL-to-C conversion involves steps taken by the parser are (i) Extraction of variables, controller and state-wise micro-operations (ii) Rewrite method to find RTL operations (iii) Processing RAM, ROM, and Function modules (iv) Generate C code.

```
 int diffeq(int x,int dx,int u,int y){
  int t1,t2,t3,t4,t5,t6,t7;
     t1 = 3*x;
     t2 = t1*u;
     t3 = t2*dx;
     t4 = 3*y;
     t5 = t4*dx;
     t6 = u-t3;
     u = t6-t5;
     t7 = u*dx;
     y = y+t7;
   }
```

Figure 5.2: *C source code of Diffeq example*

```
ST_1 :%u_read=call i32 @_ssdm_op_Read.
ap_auto.i32(i32 %u) nounwind
ST_1 :%dx_read = call i32 @_ssdm_op
_Read.ap_auto.i32(i32 %dx) nounwind
ST_1 :%x_read = call i32 @_ssdm_op_
Read.ap_auto.i32(i32 %x) nounwind
ST_1 :%shl_ln12 = shl i32 %x_read, 2
ST_1 :%t1 = sub i32 %shl_ln12, %x_read
ST_1 :%mul_ln14 = mul i32 %dx_read, %u_read
ST_1 :%shl_ln15 = shl i32 %dx_read, 2
ST_1 :%t4 = sub i32 %shl_ln15, %dx_read
```

Figure 5.3: *State 1 of diffeq.verbose.rpt file generated by Vivado HLS*

5.2.1.2 Scheduled C code

The HLS tool like Vivado HLS generates a report that contains all the information about scheduling [44]. The report contains state-wise 3-address operations of the FSM. The syntax of these operations is in the form of intermediate representation (IR) of the front-end compiler. We need to decode these operations to get the Scheduled C code (SD-C). The scheduled information generated by Vivado HLS tool for Diffeq example given in Fig. 5.2 at state 1 obtained from diffeq.verbose.rpt file is shown in Fig. 5.3. After decoding the schedule information (instructions) shown in Fig. 5.3, the scheduled C code generated corresponding to state 1 is shown in Fig. 5.4. Here, u_read, dx_read, and x_read contain the values read from input variables u, dx, and x, respectively. So, we have replaced u_read, dx_read, and x_read by u, dx, and x, respectively. Other 3-address operations are shown in the syntax

```
shl_ln12 = x << 2;
t1 = shl_ln12 - x;
mul_ln14 = dx * u;
shl_ln15 = dx << 2;
t4 = shl_ln15 - dx;
```

Figure 5.4: *State 1 of scheduled C code after decoding verbose file*

```
if((1==ap_CS_fsm_state1)&&(ap_start==1)){
   t4_reg_120=(dx << 2) - dx;
   t1_reg_110=(x__temp<<2) - x__temp;
   mul_ln14_reg_115=dx * u;
 }
goto ap_ST_fsm_state2;
```

Figure 5.5: *State 1 in RTL-C of Diffeq example*

of the C language. For Diffeq example shown in Fig. 5.2, The RTL-C contains five states. Instructions of state 1 in RTL-C are shown in Fig. 5.5.

5.2.1.3 Combine Two C Codes

Now, we have two C codes: *RTL-C* and *SD-C*. The *RTL-C* contains operations in terms of inputs, constants, registers, and RAM and ROM. The format of the RTL-C is given in Section 3.12. The *SD-C* contains operations in terms of inputs, constants, and variables and instructions are in three-address form. In both cases, the operations can be identified state-wise. As Daikon finds invariants in the same program, we need to combine *RTL-C* and *SD-C* into one program to find the mapping between registers and variables. As discussed, the number of states and the state transitions from one state to another are the same in both C codes. Diakon finds invariants at the function's entry and exit. Therefore, the instructions of both *RTL-C* and *SD-C* in each state are combined and put into a function. Daikon finds invariant only if the variables are declared globally. So all variables must be declared globally. Instructions of state 1 of both RTL-C code and SD-C code are combined and put into a function called state1(). The combined C code is shown in Fig. 5.6.

```
void state1(){
 //state 1 code of RTL-C
 t4_reg_120 = (dx << 2) - dx;
 t1_reg_110 = (x__temp << 2) - x__temp;
 mul_ln14_reg_115 = dx * u;

 //state 1 code of SD-C
 shl_ln12 = x << 2;
 t1 = shl_ln12 - x;
 mul_ln14 = dx * u;
 shl_ln15 = dx << 2;
 t4 = shl_ln15 - dx;
}
```

Figure 5.6: *State 1 of combined C code*

5.2.1.4 Invariant Generation using Daikon

After obtaining the combined C code for all states, we need to find invariants at the state entry and exit points of each function (representing a combined state) in the combined program. Invariants generated by the Daikon depend on the number of test cases and the quality of test cases. We make sure that our test cases cover all traces of the combined code. Daikon finds invariants based on values contained by the variables while running it on these test cases. The invariants found by Daikon are shown in Fig. 5.7 for the combined code of state 1 of Fig. 5.6.

5.2.1.5 Extract Useful Mapping

As shown in Fig. 5.7, the output of Daikon contains many invariants. Many of them are not relevant in our context. So, we need to identify the useful invariants in which there is an equal relationship between the registers of RTL-C and the variables in SD-C. So, in this phase, useful mapping is extracted automatically from the Daikon generated invariants. The useful mapping of the registers of RTL-C and the variables of SD-C is shown in Fig. 5.8. Registers mul_ln14_reg_115, t1_reg_110 and t4_reg_120 mapped with variables mul_ln14, t1 and t4, respectively. Registers to variables mapping for all other states of the FSM can be found in a similar manner.

```
state1():::exit
::add_ln20_fu_96_p0 == ::add_ln20_fu_96_p0_temp
::add_ln20_fu_96_p0 == orig(::add_ln20_fu_96_p0)
::t1_reg_110 == ::t1
::add_ln20_fu_96_p0 == 0
::mul_ln4_fu_56_p0 == 0
::dx != ::u
::x != ::t3
3 * ::x - ::t1_reg_110 == 0
4 * ::x - ::shl_ln12 == 0
::t5 >= orig(::t4)
::t6 >= ::sub_ln18
::dx == ::dx_read
::u == ::u_read
::x == ::x_read
::mul_ln14_reg_115 == ::mul_ln14
::t1_reg_110 == ::t1
::t4_reg_120 == ::t4
```

Figure 5.7: *Daikon invariants output for state 1*

```
state1 :
 ::mul_ln14_reg_115 == ::mul_ln14;
 ::t1_reg_110 == ::t1;
 ::t4_reg_120 == ::t4;
```

Figure 5.8: *Mapping of State 1 registers to variables*

5.2.2 Challenges Resolved

In this Subsection, we describe major challenges that we need to address in our Daikon based register to the variable reverse engineering process with the solution approaches.

5.2.2.1 Generating Quality Test Cases

Daikon output depends on the quality of test cases. So, we have to make sure that the test cases selected must cover all the traces in the behaviour. We have utilized the symbolic execution tool Klee [7] for this purpose. Specifically, we have used Klee to identify all traces in the combined code first. We then tuned Klee to provide a test case/inputs for each trace. In fact, we generate multiple test cases for each trace in combined code using Klee. This ensures that our test cases have a hundred percent functional coverage of the combined code. To add diversity to the test cases, we include some random test cases as well.

5.2.2.2 Unmapped Temporary Variables

The RTL-C code contains more than one operation in a single statement due to operation chaining in hardware. The expressions in SD-C, on the other hand, contain only one operation (since it is in three address form). Therefore, many temporary variables are created in SD-C which are not mapped to any register in the RTL. Since these operations are happening in a single state and they have no further use in any future states, we don't need to find the mapping for these temporary variables of SD-C. So before calling Daikon, the combined code needs to be pre-processed so that all the temporary variables that are not being used in other states are removed. Its value is substituted in expressions where it is being read.

Example 16. *For instance, the operation of one state of RTL-C in Waka benchmark is*
`t23_reg_365 = in3 - in4 + in22 + in7 + in12 + in8.`
The operations in the corresponding state of SD-C are
`t5 = in3 - in4,`
`add_ln21 = t5 + in22,`
`add_in12 = in7 + in12,`
`t11 = add_in12 + in8,`
`t23 = add_ln21 + t11.`
Here we see that there is operation chaining happened in the RTL-C code. The variables `add_ln21, add_in12, t5 and t11` *are temporary variables in a same state. So, we remove these temporary variables and replace their occurrence with their expressions in that state.* □

5.2.2.3 Transitive Analysis of Daikon Output

In some cases, the direct register to variable mapping may not be found in the Daikon output. Specifically, some of the mappings is missing in Daikon output due to transitive dependency. To resolve this issue, we perform transitive analysis on Daikon output to obtain relevant invariants in such cases.

Example 17. *Consider the following instructions from Matrixop benchmark:*
`mat1_load_reg_651 = mat1_q0,` *and*
`mat1_load = mat1_addr_1.`

The above two instructions are of RTL-C and SD-C, respectively. From Daikon output, we obtain the following mappings:
`::mat1_q0 == ::mat1_addr_1,`
`::mat1_q0 == ::mat1_load` *and*
`::mat1_q0 == ::mat1_load_reg_651.`
Here mapping of `mat1_load_reg_651` *and* `mat1_load` *is missing. By transitivity, we can get the required mapping.* □

5.2.2.4 One Register to Many Variables Mapping in a State

We know that in a time step one register can not hold more than one value. Both RTL-C and SD-C are cycle accurate. So, the mapping of one register to many variables in a state is not possible. But in Daikon output, we find such one-to-many mapping due to copy propagation. In such a case, we can use one of the mappings in all places since both the variables have the same value.

Example 18. *For instance, the operation of one state of RTL-C is*
`res1_load_1_reg_161 = mul_ln12_reg_661 + res1_load_1_reg_161__temp.`
The operations in the corresponding state of SD-C are
`add_ln12 = mul_ln12 + res1_load_1,`
`res1_load_1 = add_ln12.`
Here we see that there is copy propagation is happening in SD-C. Daikon gives us following mappings
`::res1_load_1_reg_161 == ::add_ln12` *and*
`::res1_load_1_reg_161 == ::res1_load_1.`
So, the register `res1_load_1_reg_161` *maps to two variables in a same state. As register* `res1_load_1_reg_161` *maps to two variables, we can use any one of two mappings in that state.* □

5.2.2.5 Mealy vs Moore Models

The controller FSM can be a Mealy or Moore machines. In our framework, we support both as discussed below.

Mealy Model: In this model, operations are associated with the state transition. Based on transition condition, the operation performed is decided. For the Mealy machine, we

```
//Control Structure
   state1:
    o1;
    c = exp ? 1 : 0;
    if (c){
    o2;
    goto state2;
    }
    if (!c){
    o3;
    goto state3;
    }
```
(a)

```
//Mealy model:
    e1(){
     o1;
     o2;
     }
    e2(){
     o1;
     o3;
     }
    state1:
     c = exp ? 1 : 0;
     if (c){
     e1();
     goto state2;
     }
     if (!c){
     e2();
     goto state3;
     }
```
(b)

```
//Moore model:
   state1(){
        o1;
        if(c){
         o2;
        }
        if(!c){
         o3;
         }
    }
    state1:
      c = exp ? 1 : 0;
      state1();
      if(c){
        goto state2;}
      if(!c){
        goto state3;
      }
```
(c)

Figure 5.9: *(a) Control Structure (b) Code structure following Mealy Model (c) Code structure following Moore Model*

model the transitions as function in the combined code.

Example 19. *Consider the control flow example shown in Fig. 5.9(a). There are three states in the FSM, and from state1 (s1) there are two transitions based on the condition 'c'. If c is True, then control goes to state2 (s2), and if c is False, control goes to state3 (s3). Operations in state1 are o1, o2, and o3. The operation o2 executes when there is the transition from state1 to state2, the operation o3 executes when there is the transition from state1 to state3 and the operation o1 executes in all the cases. So, the operation associated with transition e1 (from state1 to state2) is o1, o2 and the same for the transition e2 (from state1 to state3) is o1, o3. The Mealy model of this scenario is shown in Fig. 5.9(b) and in Fig. 5.10(a) with a diagram. Daikon finds invariants at the entry and exit of the transition functions e1 and e2.* □

Moore Model: In this model, operations are associated with states. All the operations are performed in the state, and based on the condition transitions are made separately. For the Moore machine, we model the states as functions in the combined code.

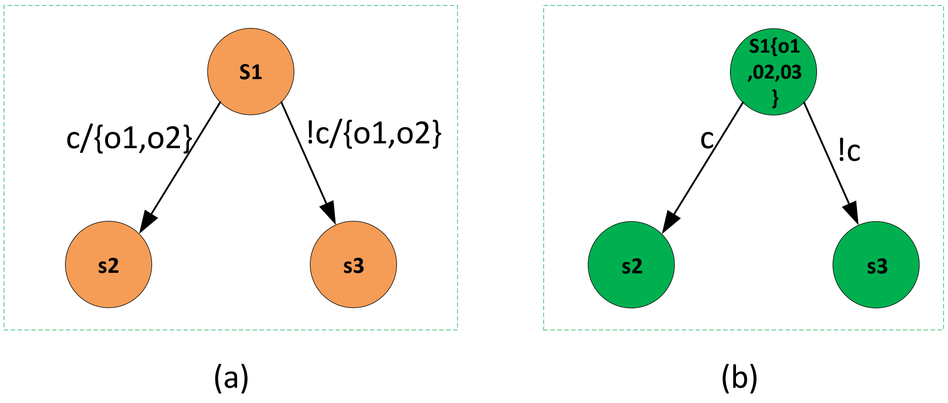


Figure 5.10: *(a) Mealy Model. (b) Moore Model.*

Example 20. *Consider the control flow Fig. 5.9(a) again. The state1 (s1) contains three operations $\{o1,\ o2,\ o3\}$. The operation o2 is performed when condition c is True. The operation o3 is performed when condition c is False. The o1 is the operation performed in all the conditions. So, all the operations are associated with state1. But, they will be executed when the corresponding condition is True. The transition is performed separately. If c is True in state 1, then transition to state2 occurs and if c is False in state 1, then the transition to state3 occurs. The Moore model of this scenario is shown in Fig. 5.9(c) and in Fig. 5.10(b) with a diagram.* □

5.2.3 Correctness of Reverse Engineering Flow

We use Klee for generating the test cases that cover all the traces in the combined code. In a state, if the values of two variables are the same for all the test cases and one is register r_i of RTL-C and the other is variable v_i of SD-C then it is sure that r_i map to v_i. As a result, the mappings obtained by Daikon are consistent with all traces of the behaviour. We have done simulation-based verification for our benchmarks to check the correctness of the register to variable reverse engineering. Specifically, we rewrite the RTL-C by replacing the register name with the corresponding variable name using the mapping obtained. We then run the test benches used for RTL co-simulation on the reverse engineered C code and compared the outputs with input C code. The outputs match for all benchmarks used which confirm the correctness of our reverse engineering flow. Although the simulation-based method does not provide formal correctness proof, it is commonly used in the HLS domain for verification.

Table 5.1: *Experimental Results for different high-level synthesis benchmarks*

Bench marks	#in	#C	#SC	#RTLC	# State	#var	#regs	M required	Mobtained	Mapp APP	NOTC	Rtime(s)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
Diffeq	4	14	30	406	5	15	7	15	15	15	100	1.32
Matrix op	2	33	274	3771	17	90	32	75	64	75	100	931.7
Waka	20	33	133	474	3	21	12	19	16	19	100	5
Motion	10	52	101	780	5	42	29	44	43	44	100	7.89
ARFNC	15	48	68	607	5	28	12	28	28	28	100	4.55
ARFNB	17	54	63	981	7	56	21	58	58	58	100	173.51
Parker	6	51	67	330	2	14	2	3	3	3	100	0.456

5.2.4 Experimental Results

The above reverse engineering framework is implemented in Python and is tested on a set of HLS benchmarks. We have used the Vivado HLS tool [10] to generate Verilog RTL for the benchmarks written in C. We then extract RTL-C from the Verilog using our FastSim tool. We have obtained the SD-C from the synthesis report of the Vivado HLS manually. The RTL-C and SD-C codes are then combined manually and the rest of the flow is automated.

The experiments have been performed on a machine with a CPU: Intel Core i7, 2.5GHz, and 8GB RAM. We evaluate our method on a variety of HLS benchmarks (Waka, Motion, Diffeq, Parker, Matrixop, auto-regressive lattice filter with no constant (ARFNC) and auto-regressive lattice filter without branch (ARFNB)), each of them written in C-code. The experiment results of our benchmarks are shown in Table 5.1. The 1st column has the name of the benchmarks used. The 2nd and 3rd columns represent the number of inputs (#in) and the number of lines (#C) of the input C code, respectively. The 4th, 5th and 6th show the number of lines of SD-C code (#SC), the number of lines of RTL-C code (#RTLC), and the number of states (#State), respectively. The number of variables (#var) in the SD-C code and the number of registers (#reg) in the RTL-C code are shown in columns 7th and 8th columns, respectively. The 9th, 10th, 11th, and 12th columns are the number of mapping required (Mrequired), the number of mapping obtained (Mobtained), the number of mapping after post-processing (MappAPP), and the number of test cases (NOTC), respectively. Finally, the last column reports the time spent by our system to map registers to variables. Mrequired is less than the product of #reg and #State because in some of the states not all the registers are used. So, the Mrequired is the sum of registers used in every state. We set our test case population to 100 because the number of paths in each benchmark is less than 100. These test cases are the combination of x test cases generated by the Klee and $(100-x)$ are the random test cases. It may be noted from columns 7 and 8 that the number of registers is significantly less than the number of variables. Also, column 10 suggests that most of the required mappings are obtained directly by Daikon. For Matrixop, waka, and Motion, the rest of the mappings are also obtained by transitive analysis of the Daikon result (as discussed in Section 5.2.2.3). Therefore, our framework identifies all the required register to variable mapping of all benchmarks successfully. The run-time is also less than a minute in most of our test cases. The run time is a bit high in Matrixop test case since it involves arrays. We have also observed that the run time is improving if we use less test cases (but cover all traces) in Daikon. Specifically, one may identify the adequate test cases for an application to optimize the run time of our tool.

5.3 SMT based Reverse Engineering of Register to Variable Mapping

Our objective is to find the mapping between the variables in the input C and the registers in the RTL. In this case, the input C code is untimed. Therefore, combining input C and RTL-C state-wise is not possible. Instead, we have to consider the complete behaviours together. In this work, we formulated the mapping problem as a satisfiability (SAT) problem and used an SMT solver Z3 to identify the mapping. The overall flow is discussed below.

5.3.1 Reverse Engineering Steps

The overall flow of our SMT based register to variable reverse engineering approach is given in Fig. 5.11. First, we extract the RTL-C from the RTL using our FastSim tool. In the next step, both the input C and RTL-C are modeled as FSMD [64], called C-FSMD and RTL-FSMD, respectively. We then convert both the FSMDs into static single assignment form in SSA steps [50]. Next, the SAT problem is formulated and the SMT solver is invoked. If the SMT solver finds the formula satisfiable, we can get the mapping from the instance provided by the SMT solver. If SMT solver timeouts or could not solve the formula, our

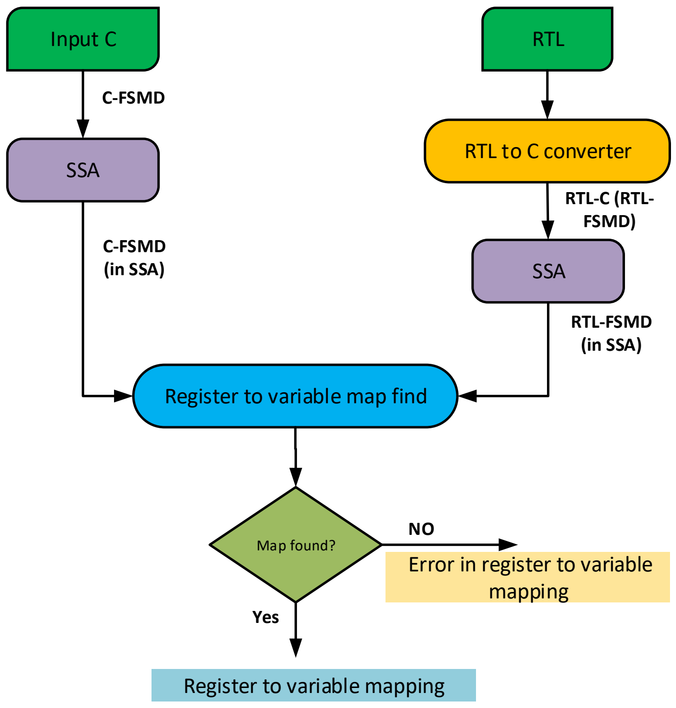


Figure 5.11: *SMT based Register to variable reverse engineering flow*

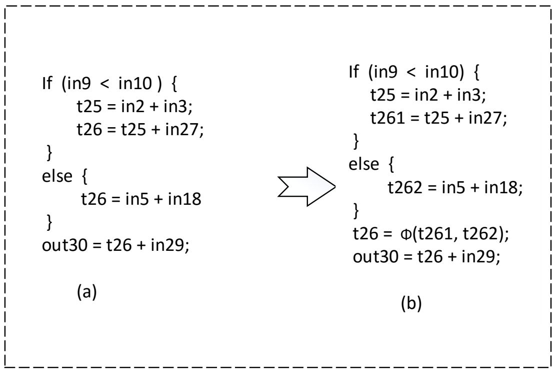
extraction process fails.

5.3.1.1 FSMD Extraction from HLS Generated RTL

The RTL structure generated by the HLS tool consists of three basic constructs: (i) a set of always statements, (ii) the controller FSM block and (iii) a set of assignment statements. The always statement blocks update the registers under certain conditions. The condition depends on the state and a conditional statement over registers/inputs. The condition of state transitions and the operations in each transition is not mentioned in the controller structure. These can be obtained by combining controller FSM with the always blocks and the assignment statements. In assignment statements, their left-hand side (LHS) is updated whenever the right-hand side value changes. Overall, the datapath and the controller FSM can be identified in the RTL with this. The extraction of the RTL-C from the HLS generated RTL is discussed in detail in Section 3.3.4. We take the FSMD representation of RTL-C in this work.

5.3.1.2 SSA Transformation

In static single assignment (SSA) representation, each variable is assigned exactly once in the behaviour, and every variable is defined before it is used [50]. Since the variables of the C specification are mapped to registers in the RTL, these two FSMDs, i.e., C-FSMD and RTL-FSMD are not comparable yet. We then convert both the FSMDs into a single assignment form in SSA steps [50]. In SSA, we first convert all the operations in 3-address form i.e., at most one operator in the right-hand side (RHS) expression of an operation. A variable in the C specification may be defined multiple times. As a result, such variables may be mapped to more than one register in the RTL. Also, multiple variables with non-overlapping lifetime may be mapped to a single register which effectively indicates that a register is defined multiple times in RTL-FSMD. The SSA step on C-FSMD ensures that each variable is defined exactly once in the C specification. The SSA on RTL-FSMD ensures that each register is defined exactly once in the RTL. Since both these sets of variables capture the lifetimes of variables of the input specification, the number of variables in the C-FSMD and the number of registers in the RTL-FSMD will be the same after this step. We perform this step because modeling redefinition of a variable is not possible in the SMT formula. However, we need to keep track of the new variables introduced in SSA with their corresponding original variable name.



```
If (in9 < in10 ) {
    t25 = in2 + in3;
    t26 = t25 + in27;
}
else {
    t26 = in5 + in18
}
out30 = t26 + in29;
```

(a)

```
If (in9 < in10) {
    t25 = in2 + in3;
    t261 = t25 + in27;
}
else {
    t262 = in5 + in18;
}
t26 = Φ(t261, t262);
out30 = t26 + in29;
```

(b)

Figure 5.12: *SSA example*

Example 21. *Consider the example shown in Fig. 5.12. In Fig. 5.12(a), the variable t26 is defined twice. This can be resolved in SSA by creating two new variables: t261 and t262, which are assigned only once. Most programs have branch and intersection (join) nodes.*

At the join nodes, we add a special form of statement called a ϕ-function as shown in Fig. 5.12(b). The operands to the ϕ-function indicate which assignments to t26 to reach the joint point. Depending on the control flow before the joint node, this function will generate a new definition t26 by choosing either t261 or t262. The SSA version of the behaviour in Fig. 5.12(a) is given in Fig. 5.12(b). □

Algorithm 3: Obtain_register_to_variable_mapping (C-FSMD, RTL-FSMD)

```
   Result: register to variable mapping
1  $V_c = \{\phi\}$, $V_r = \phi$, $I_c = \phi$, $I_r = \phi$, CTP = $\{\phi\}$.
2  repeat
3      Generate next random input.
4      Simulate the both C-FSMD and RTL-FSMD on the input to obtain the traces, $\tau_c$
       and $\tau_r$, respectively.
       /* Collect variables and the registers that involved in $\tau_c$ and $\tau_r$   */
5      $V_c = V_c \cup$ {set of all variables updated in $\tau_c$}.
6      $V_r = V_r \cup$ {set of all variables updated in $\tau_r$}.
7      $I_c = I_c \cup$ {set of all inputs involved in $\tau_c$}.
8      $I_r = I_r \cup$ {set of all inputs involved in $\tau_r$}.
9      CTP = CTP $\cup \langle \tau_c, \tau_r \rangle$.
10 until $(V_c \equiv V \wedge V_r \equiv R \wedge I_c \equiv I_r \equiv I)$;
11 $F$ = Formulate_SAT_problem(C-FSMD, RTL-FSMD, CTP).
12 Call Z3 for with all the formulas in $F$.
13 if Z3 returns SAT then
14     return the register to variable mapping.
15 else
16     Report error in register mapping.
```

5.3.1.3 SAT Formulation

An FSMD may contain many traces. We can consider all traces in the SAT formulation. However, that will make the problem complex. To simplify the formulation, our intuition is to collect enough traces so that all variables, all registers, and all the inputs are involved. For this purpose, the RTL-FSMD and the C-FSMD (obtained after SSA transformations) are simulated together. It may be noted that the inputs and the outputs are the same for both the FSMDs. The initial simulation runs with random inputs. The traces obtain in both the FSMDs for a given random input called as *corresponding trace pair (CTP)*. Let assume that the CTP obtained by initial simulation is $\langle\tau_c, \tau_r\rangle$, where τ_c is a trace in C-FSMD and τ_r is the corresponding trace in RTL-FSMD. We shall collect the variables that get defined in τ_c in a set V_c. Similarly, all the registers get defined in τ_r are collected in a set V_r. Also, all the inputs that are involved in s_{τ_c} and in s_{τ_r} are collected in I_c and I_r, respectively. We next generate the next trace using concolic testing approach [18] and continue to do the above. This process will stop when $V_c = V$ (set of variables), $V_r = R$ (set of registers) and $I_c = I_r = I$. The reverse engineering method is presented as Algorithm 3. The lines 1-10 of Algorithm 3 is performing this step.

Let the set of variables and the set of registers be denoted as V and R, respectively. As argued before, both these set contain the same number of elements. Let's assume each of them contains N number of elements. Let consider the Boolean variables m_{vr}, $\forall v \in V, \forall r \in R$. Let assume that the Boolean variable $m_{vr} = 1$ if the variable $v \in V$ is mapped to register $r \in R$; otherwise, $m_{vr} = 0$. The mapping between the variables and the registers is one-to-one. The following two constraints are captures that fact.

(i) Each variable must be mapped to exactly one register.

$$\sum_{r=1}^{N} m_{vr} = 1, \ \forall v \in V \tag{5.1}$$

(ii) For each register, exactly one variable is mapped to that register.

$$\sum_{v=1}^{N} m_{vr} = 1, \ \forall r \in R \tag{5.2}$$

In addition, expression corresponding to each output in the traces C-FSMD will be rewritten in terms of m_{rb} and r. Each variable v_i in an output expression will be replaced by $\sum_{r=1}^{N} m_{v_i r} \times r$. We loosely denote this replacement as $v_i \circ m_{vr}$. For example, the expression corresponding to output o_1 is $v_1 + v_2$ in a trace τ_c in C-FSMD. Assume that we have two registers r_1 and r_2 in S-FSMD. The v_1 will be rewritten as $m_{v_1 r_1} \times r_1 + m_{v_1 r_2} \times r_2$. This expression indicates that v_1 is replaced by the corresponding register. The corresponding register of v_1 is r_1 if $m_{v_1 r_1} = 1$; otherwise, r_2. Similarly, v_2 will be rewritten as $m_{v_2 r_1} \times r_1 + m_{v_2 r_2} \times r_2$. Let the output expression corresponding to o_1 in S-FSMD is $r_2 + r_1$. The constrains will be passed to SMT solver for o_1 is $(m_{v_1 r_1} \times r_1 + m_{v_1 r_2} \times r_2) + (m_{v_2 r_1} \times r_1 + m_{v_2 r_2} \times r_2) \equiv r_2 + r_1$. We will identify such constraints for each output for each trace pair identified in Algorithm 3.

Let us assume that the output set is denoted as O in both the FSMDs. The third constrain is as follows.
(iii) For each corresponding trace pair selected in algorithm 3, the outputs are equivalent.

$$\forall \langle \tau_c, \tau_r \rangle, \forall o \in O, R_{\tau_c} \circ m_{vr} \equiv R_{\tau_r} \tag{5.3}$$

We call Z3 with all these constraints to find the values of m_{rb} that satisfies all the constraints. If there is any solution, the SMT solvers will return the values of m_{vr}. The overall process is given as algorithm 3.

Using the value of m_{vr}, the exact mapping between variables and registers is obtained. All the registers in the RTL-FSMD can be replaced by the corresponding variable with m_{vr}. The resultant FSMD is called S-FSMD (i.e., scheduled FSMD). The overall algorithm to register to variable reverse engineering is given as Algorithm 3.

Example 22. *An example of identifying register mapping is given in listing 5.1; In this example, C code is $out = a - b$ which is mapped to $out = r2 - r1$ is RTL. The Z3 returns $m11 = 0, m12 = 1, m21 = 1, m22 = 0$. Basically, the formulation recovers that the register $r2$ maps to variable a and the register $r1$ maps to variable b.* □

Listing 5.1: *An example SMT code to obtain mapping*

```
(declare-const m11 Int)
(declare-const m12 Int)
(declare-const m21 Int)
(declare-const m22 Int)
(declare-const out1 Int)
(declare-const out2 Int)
(declare-const r1 Int)
(declare-const r2 Int)
;values are either 0 or 1
(assert (or (= m11 1) (= m11 0)))
(assert (or (= m12 1) (= m12 0)))
(assert (or (= m21 1) (= m21 0)))
(assert (or (= m22 1) (= m22 0)))
;each var is mapped to one reg
(assert (= (+ m11 m12) 1))
(assert (= (+ m21 m22) 1))
;each reg is associated to exactly one var
(assert (= (+ m11 m21) 1))
(assert (= (+ m12 m22) 1))
(assert (= out2 (- r2 r1)))
(assert (= out1 (- (+ (* m11 r1) (* m12 r2))
              (+ (* m21 r1) (* m22 r2)))))
(assert (= out1 out2))
(check-sat)
(get-model)
```

5.3.2 Experimental Results

The SMT based register to variable reverse engineering framework of HLS is implemented in Python and is tested on a set of HLS benchmarks. We have used the Vivado HLS tool [10] to generate Verilog RTL for the benchmarks written in C. Our tool invokes the SMT tool Z3 [55] as shown in Algorithm 3. A satisfiable instance will reveal the mapping between the variables and registers. One primary challenge was handling arrays. In Vivado HLS, arrays are mapped to block RAMs. For RAM, a separate module is created with an address port, data port, and write enable to access it. In each control state, the controller assigns appropriate values to these ports for RAM access. Based on these values, our rewriting method identifies the actual memory read/write operation in each state. The RTL-C extraction is automated as discussed in Chapter 3. The SSA step is not automated. The steps of Algorithm 3 i.e., identifying enough traces in the FSMD, SMT formula generation, and SMT invocation are automated.

The experiments have been performed on a machine with a CPU: Intel Core i7, 2.5GHz, and 8GB RAM. We evaluate our method on a variety of HLS benchmarks (Waka, Motion, DIFFEQ, maximum of three numbers (Max3), and auto-regressive lattice filter with branch (ARFWB) and without branch (ARFNB)), each of them written in C-code. Table 5.2 presents the experimental results. The 1st column has the name of the benchmarks used. The 2nd and 3rd columns are the lines and variables in the C code, respectively. The 4th, 5th, 6th, and 7th columns are the number of lines, number of variables, latency, and number of states of the RTL-FSMD in RTL-code, respectively. The number of variables in the C-code after SSA and the number of registers in the RTL-code after SSA are the same in all test cases. The 8th and 9th columns show the number of lines in SMT format code and SMT time, respectively. The SMT time is the time required by Z3 for register mapping. Finally, the last column reports the total (HDL parsing time + SMT code generation + Z3 run time) time spent by our reverse engineering framework for each benchmark. The run times are not high and are less than three seconds for all cases. The majority of time is taken by the SMT tool Z3. Our proposed method is applicable for moderate size benchmarks. The

Table 5.2: *Experimental Results for different high-level synthesis benchmarks*

Bench marks	C code		RTL code				Our tool		
	#lines	#var.	#lines	#reg.	latency	#States	#smtcode	SMT time(s)	Overall time(s)
Waka	33	21	270	8	8	3	658	0.380	0.432
Motion	52	42	415	21	8	5	1920	1.470	1.545
DIFFEQ	14	15	220	7	6	5	578	0.034	0.0925
Max3	15	5	150	3	2	3	377	0.025	0.0498
ARFNB	54	56	444	21	10	7	1803	1.380	1.488
ARFWB	48	28	351	12	6	5	1547	1.300	1.397
Matrixop	49	90	832	32	10	17	4803	2.486	2.798
Parker	51	23	188	12	4	2	547	0.027	0.0396

scalability of this approach depends on the SMT solver. For large benchmarks, we found SMT solver timeouts.

We have done simulation-based verification for our benchmarks to check the correctness of the register to variable reverse engineering. Specifically, we rewrite the RTL-C by replacing the register name with the corresponding variable name using the mapping obtained. We then run the test benches used for RTL co-simulation on the reverse engineered C code and compared the outputs with input C code. The outputs match for all benchmarks used which confirm the correctness of our reverse engineering flow.

5.4 Applications of REVAMP Framework

In this section, we will discuss some of the potential applications of the register to variable mapping information extracted by our proposed method REVAMP.

5.4.1 Register Allocation and Security Aspect

Register allocation is the mapping of a large number of variables to a limited number of physical registers. Let us assume that V_i be the set of variables with i number of variables in a program and R_j be the set of registers with j number of registers in the target architecture. Register allocation is a function which maps the set of variables to the set of registers and/or memory locations, $f : V_i \rightarrow R_j \cup M_k$ where usually $j \leqslant i$ and M_k is the reserved memory locations for arrays V_k where $0 \leqslant k \leqslant i$. The following scenarios may arise due to register allocation.

– A variable v_a is mapped to a register r_x, i.e. $f(v_a) = r_x$.

– More than one variable are mapped to a register. For example, $f(v_a) = f(v_b) = f(v_c) = r_x$ means the variables v_a, v_b and v_c are mapped to the register r_x.

– A variable is mapped to more than one register, i.e. $f(v_a) = \{r_x, r_y, r_z\}$ which is called the live range splitting. In this case, some re-definitions of v_a is renamed first and then different instances of v_a is mapped to different registers. Usually, it reduces the register pressure. In case of limited registers, live range splitting is used if it minimizes the register usage.

– If there are not enough registers, a array variable needs to be stored into memory, i.e. $f(v_a) = m_a$ where m_a is the reserved memory location for the variable v_a.

Securing compiler transformation is an emerging research area and very few works have been done on trusted code generation. The correctness-security gap of compiler optimizations attract some attention in recent times. D'Silva et al. [58] studied various compiler transformations to identify the leaky one. They conclude that the gap arises due to the techniques that do not model the state of the underlying machine. They only considered the transformations at the source level, nothing mentioned the information leaks in register allocation. Deng and Namjoshi presented a polynomial-time algorithm for secure dead store elimination in [56]. Recently, Besson et al. [27] proposed a formal definition of the Information Flow Preserving (IFP) program transformations in which they model the information leak of a program using the notion of attacker knowledge. The authors have shown how to validate register allocation and dead store elimination and if needed how to modify it in order to be IFP.

Register allocation is a mandatory transformation for any source program to generate the machine code. Thus, it should be properly investigated from the security point of view. Moreover, none of the existing works provide a secure register allocation algorithm. In this work, we target the reverse engineering of the register to variable mapping in register allocation. Although, this work does not target the security analysis of register allocation, the recovered mapping can be useful in analyzing the same. Specifically, this mapping information can be used to identify the information leakage after register mapping and then iteratively modify the register allocation to stop the leak. This could be interesting future work.

5.4.2 Correlating C and RTL

The quality of designing a hardware device by the HLS tools depend on the way one has written the design specification in C/C++ code. From specification document, an RTL description is created to examine the design in terms of functionality, performance, compliance with standards and other high-level issues by HLS tools. On the other hand, the high-level specification developers do not understand how the specification is represented in RTL. In order to help specification developers understand, debug and verify an RTL design that is generated from the HLS tool, it is important to bridge the knowledge gap between the two levels of abstraction. A C equivalent of the RTL model would be helpful for the specification developers to understand, debug and verify the output RTL design and hence use the HLS tool meaningfully. One of the important steps in obtaining a high-level C behaviour from the RTL is to recover the mapping between variables in input C and registers in the RTL generated by HLS. Our work is helpful in this context.

5.4.3 Fast Simulation and Debug

As discussed in Chapter 3, the RTL co-simulation is the primary way to verify the correctness of the generated RTL of an HLS tool. Specifically, the test cases developed for the input C code are used by the commercial HLS tools [10] for RTL simulation. The C-simulation is faster than the RTL simulation. Therefore, generating an equivalent C code from the RTL would greatly reduce the verification time as well. In FastSim [15], an C code obtained from the RTL is shown to be much faster for RTL verification. However, the code obtained from the RTL cannot be used for correlating the bug with the source C code of the HLS tool and hence for souce level debug. With the variable to register mapping information obtained in our work, correlating the source C with RTL is possible in debugging a bug in the RTL.

5.5 Comparisons of Daikon and SMT based Reverse Engineering Frameworks

The Daikon based framework is used to extract mapping between the variables in SD-C and RTL-C. Whereas, the SMT based framework identifies the mapping between the variables in the input-C and RTL-C. These two methods are applied to get back an equivalent C code from the RTL code generated by the HLS tools in this Chapter. Daikon [4] is software that dynamically detects likely invariants. There is no formal guarantee in the Daikon based approach. Daikon based register to variable reverse engineering fails to get an equivalent C code from the RTL when Daikon generates insufficient invariant. On the other hand, the SMT based framework fails when the SMT solver fails to prove the satisfiability of our formulation of reverse engineering. Another difference is that the Daikon based register to variable reverse engineering needs the scheduled information to obtain the corresponding C code but the SMT based framework does not need that. Both of them are using the RTL-C obtained from the Verilog RTL.

The Daikon based SD-C versus RTL-C register to variable mapping extraction process is simple and scalable. It is because both SD-C and RTL-C programs contain the same numbers of states and the state transitions, and state-wise analysis is possible. Therefore, it is also possible to handle bigger designs since the extraction process handles the combined code state-wise manner. The problem is more challenging when we want to find the mapping variable of input C code and the registers in the RTL-C in our SMT based flow. It is because of the following reasons: (i) the input behaviour is transformed into 3-address form and then is converted to static single assignment (SSA) form during the prepossessing step of HLS, (ii) various compiler optimizations may also be applied during prepossessing, (iii) Moreover, the input C code is untimed. Therefore, state-wise correlation (like SD-C and RTL-C) is not possible. Moreover, this method is also not scalable because it takes the complete behaviours of C and RTL-C in terms of some traces and generates an SAT formula for the same. If the application is very large, this formula will be complex and the SMT solver may not find the instances for the satisfiability.

5.6 Conclusion

In this Chapter, we have presented the reverse engineering framework REVAMP for extracting mapping between the variables of (i) scheduled C code (SD-C) and the RTL-C (representing the RTL) and between the variables of (ii) input C code and RTL-C for high-level synthesis. The first one uses a Daikon based invariant generation tool on the state-wise combined SD-C and RTL-C to identify register to variable mapping state-wise. For the second problem, we formulate the mapping as a satisfiability problem and use SMT solver Z3 to find the register to variable mapping between input-C and RTL-C. We have discussed a few applications of the extracted mapping information. In both methods, we have done simulation based verification on the set of benchmarks to check the correctness of our reverse engineering process. We have tested with several HLS benchmarks for the Verilog generated by the Vivado HLS tool and we found that the overall conversion time is reasonably small in both cases.

6

BLAST: Belling the Black-Hat High-Level Synthesis Tool

6.1 Introduction

The complexity of modern day Integrated circuits (ICs) is growing exponentially [71]. To keep pace with this complexity and to reduce design time, the use of electronic system level (ESL) computer-aided design (CAD) or high-level synthesis (HLS) [65, 49] tools are rapidly increasing. About 14 out of the top 20 semiconductor companies are using HLS tools for IC development [23]. The HLS tool converts the high-level C/C++ input specification into equivalent RTL design. The RTL consists of a datapath and a controller FSM. The FSM decides the operations to be executed in the datapath in each time step.

Hardware Trojans (HT) [83] are malicious design modifications by an adversary to either change functionality, degrade performance, leak information, or denial of service. Most of the HTs are activated by a rare condition. The circuit performs correctly in normal scenarios. Therefore, HTs are very hard to detect during the pre-silicon validation phase. Once the HT is activated, the circuit will start malfunctioning. The HTs may also be inserted in any phase of the design cycle by an untrusted computer aided design (CAD) synthesis tool [106, 26], a rogue employee in intellectual property (IP) development house, or by an untrusted foundry [83]. The impact of HTs includes economic damage, planned obsolesce or cyber-attack on national assets [29]. Therefore, detection of HTs is an important task for securing the design. This is an active domain of research in this decade [73, 102, 85, 19, 62].

The commercial electronic design automation (EDA) companies sold proprietary HLS CAD tools with a set of IPs as their component library. It may be the case that the licensed software is altered by a rogue employee. As result, the HLS tool will generate Trojan infected hardware which may not perform as expected after a certain time (i.e., once the Trojan gets activated). The employee may do this to create significant economic damage for the company or to give an attacker to access the secret key of cryptography hardware or to create a bad name for the company. Since the hardware Trojan primarily reuses the actual datapath components, it will be hard to detect them by the testing phase. In a recent study [106], [26], it is shown that hardware Trojan can be inserted by the HLS tool itself. The authors have shown that it is easy to insert HTs by the HLS tool compared to other EDA tools like logic synthesis and physical synthesis tools. Specifically, the Black-Hat HLS tool [106] inserts three types of HTs: battery exhaustion attack (BE) which may increase power consumption, degradation attack (DA) to degrade the performance of the IPs, and downgrade attack (DG) to reduce the security level of the design. Since the HLS process transforms an un-timed C/C++ code into a timed RTL code and applies various optimization in each of its sub-step, it is a difficult task for the formal verification tools to find the correlation between the initial specification and the generated RTL by HLS tool [39]. Therefore, simulation is the primary way to verify the correctness of the HLS result. Since it does not provide complete coverage, hardware Trojan inserted during HLS may likely be undetected.

6.1.1 Contributions

The objective of this chapter is to develop a formal HLS Trojan detection framework. Since a HLS tool user generates RTL from an initial C specification, we can assume that the attacker has access to both the initial C code and the corresponding RTL code. However, the attacker does not have access to any intermediate synthesis information like scheduling of operations, variable to register mapping information, etc. of the HLS tool. The question is "*can we detect the HLS Trojan by comparing the generated RTL with the initial C specification?*" It may be noted that our objective is not proving the equivalence between the C and the

RTL, rather, we try to find the difference between these two behaviours. This behavioural difference may lead to yhe detection of HLS Hardware Trojans.

Our HT detection framework is developed by utilizing two of the previous works FastSim[15] and DEEQ [16]. In [15], we have shown a way to extract a high-level behaviour from the HLS generated RTLs. In [16], we have used that high-level behaviour of the RTL to prove the correctness of HLS by showing the equivalence between the C and RTL. For completeness of the chapter, we discuss the ideas of [15] and [16] and briefly here as well. However, *Fastsim* or *DEEQ* cannot detect HLS inserted HTs. In this work, we developed a HT detection framework by utilizing the power of them. Specifically, we are looking for any inconsistency or difference during the extraction of a high-level behaviour from the RTL in [15] or during equivalence checking in [16]. Once such difference or inconsistency is identified, we further analyzed to detect the HTs. We have shown that all three HLS Trojans presented in the Black-hat HLS tool [106] are detectable in our framework. Specifically, the contributions of the chapter are as follows:

- A detection mechanism called BLAST for HLS tool inserted hardware Trojans [106] is presented here. This is the first attempt to detect the HLS inserted HTs.
- A high-level behaviour extraction form RTL and a C to RTL equivalence checking method are utilized for this purpose.
- We have shown that all HLS Trojans presented [106] can be identified by BLAST.
- A prototype of BLAST is implemented. Experimental results show the usefulness of the proposed method.

The remainder of the chapter is organized as follows. HLS Trojan detection framework is presented in Section 6.2. Detection of battery exhaustion, degradation and downgrade attacks are presented in Sections 6.3, 6.4 and 6.5, respectively. Experimental results are presented in Section 6.6. Section 6.7 presents performance of BLAST for HLS optimizations. Section 6.8 concludes the chapter.

6.2 HLS Trojan Detection Framework

The overall flow of our HT detection framework BLAST is given in Fig. 6.1. Our method takes C and RTL as the inputs. The high-level behaviour, called RTL-FSMD is the first abstract out from the Verilog RTL using the idea of [15]. The input C code is modeled as C-FSMD. Next, the equivalence checking between input C-FSMD code and RTL-FSMD is carried out. We can detect all three hardware Trojans inserted by the Black-hat HLS tool [106] during this process. During RTL-FSMD extraction from the RTL, any spurious form of operation will be identified. With further analysis, the battery exhaustion attack can be detected. The degradation attack and the downgrade attack will be detected using equivalence checking. The detection mechanisms are discussed in detail in the following sections.

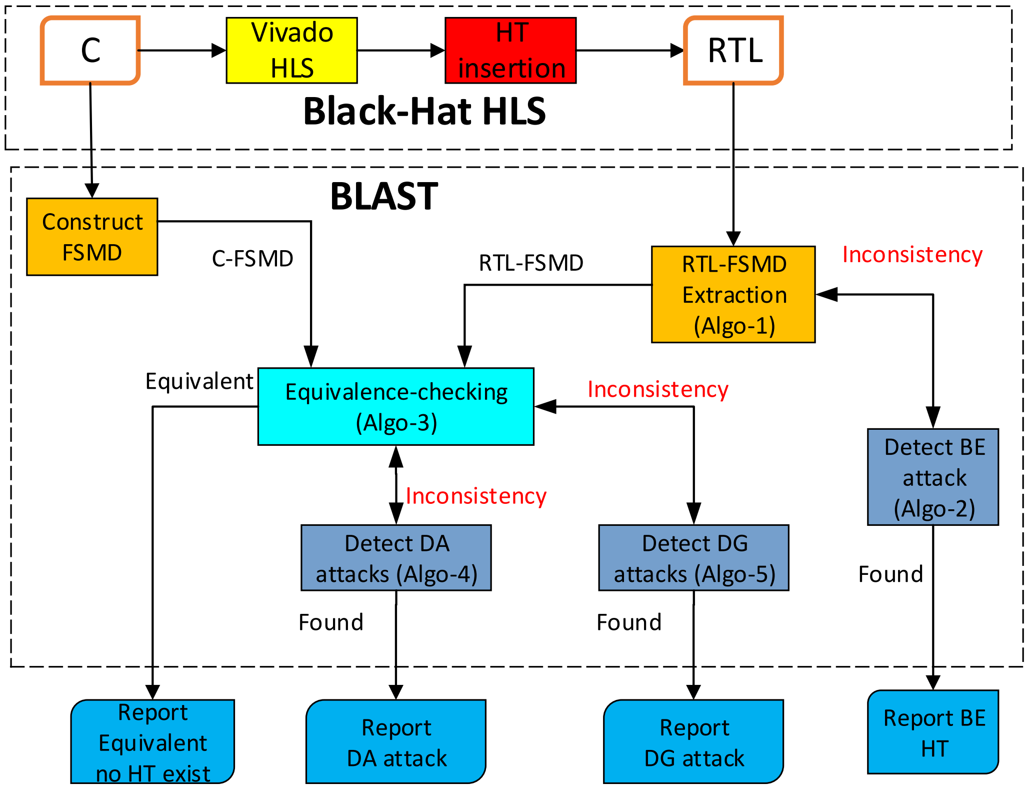


Figure 6.1: *The overall flow of our HT detection framework*

6.3 Detection of Battery Exhaustion Attack

Hardware circuits (especially small devices created with IPs) require a battery power source. Managing energy utilization is a key design principle in a circuit design. Battery exhaustion attacks can drain out power by including extra (useless) or idle functional units that use a considerable amount of power from the source. As a result, extra power will be consumed by the useless functional unit when it is switched on and the battery lifetime is shortening with no impact on the functionality.

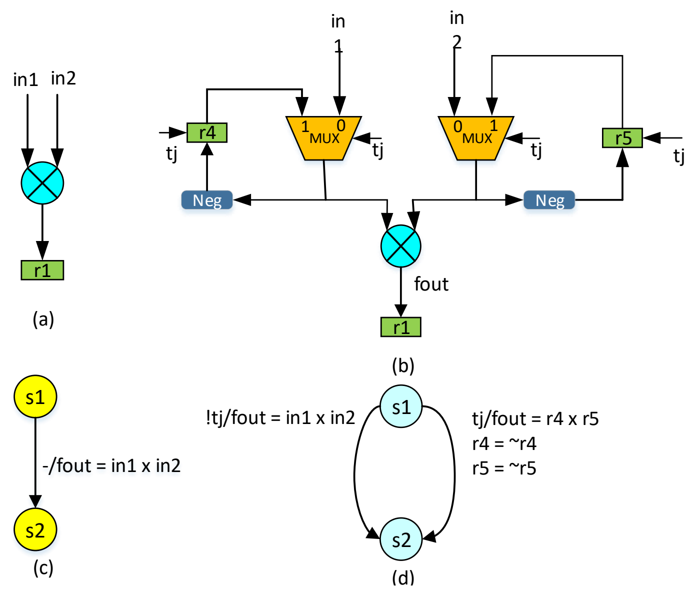


Figure 6.2: *An example to illustrate the effect of battery exhaustion attack*

6.3.1 Attack Model

In a battery exhaustion attack in [106], the idle functional Units (FUs) will be used to drain out the power when the Trojan is activated. The number of FUs required for one type of operation (e.g., multiplier) is determined by the maximum number of that operation scheduled to execute in parallel in a control state. In a control state where the number of operations scheduled is less than the number of FUs present in the datapath, some of the FUs will remain idle in that state. These idle FUs are reused to execute some fake operations in a battery exhaustion attack. The output of the FUs is bit-flipped and then multiplexed with the actual input for the FU. This multiplexer is controlled by the Trojan trigger. The results of such fake operations will not be stored in the destination register. This can be done by disabling the write enable signal of the destination register. If no idle FU is available in any control state, the tool may insert an additional state and implement the attack on that state. The idea behind this attack is to trigger the combinational functionality and enhance dynamic power consumption.

Example 23. *Let us consider the example given in Fig. 6.2. The expected datapath is shown in Fig. 6.2(a). The inputs $in1$ and $in2$ to the multiplier are from many time multiplexed registers. The details are not important in our context. We consider only the relevant part of the datapath to explain the effect of the battery exhaustion attack. The datapath is modified as shown in Fig. 6.2(b) to introduce the attack. Specifically, the output of the multiplier is negated (i.e., bit-flipped) and is multiplexed with the actual input of the multiplier. The bit-flipped data is stored in the register to avoid the combinational loop. These two additional multiplexers and the registers are controlled by the Trojan trigger tj.* □

The battery exhaustion attack will be detected during the RTL-FSMD extraction from the Verilog RTL. Specifically, HLS generated RTL has a separate datapath and controller FSM. So, in the FSMD extraction phase as explained in [15], the datapath is analyzed for the control signal assignment of each state, and the RTL operations executed in that particular state are identified. This way the controller FSM and datapath can be converted into an equivalent FSMD which is nothing but a high-level behaviour. The overall idea of the RTL-FSMD extraction process will be explained in the next subsection and how BE attack will be detected during that phase in the subsequent subsection.

6.3.2 RTL-FSMD Extraction

In the datapath, signal flow is controlled by the control signals. For each datapath module, input to output assignments is termed micro-operations. For example, for a multiplexer $out = MUX(in1, in2, sel)$, there are two micro-operations possible, i.e., $out \leftarrow in1$ and $out \leftarrow in2$ and the associated control signal assignment are $sel = 0$ and $sel = 1$, respectively. Given a control signal assignment in a control state, we have a set of active micro-operations in each transition of the controller FSM. All the assignment operations are active in all control steps. The RT operations in each state are then obtained by application of the rewriting method of the work [15]. Starting from a micro-operations of the form $r \Leftarrow r_{in}$, the rewriting method identifies the spatial sequence of data flow needed for an RT-operation in reverse order. The method consists in rewriting terms one after another in the right-hand-side expression using the active micro-operations. The method stops when all the terms in the RHS are either registers, inputs, or constants. The rewriting takes place from left to right in a breadth-first manner.

The above process will identify the RTL operation(s) executed in a state of the controller

FSM. The same process can be applied for each state of the controller FSM to extract the RTL-FSMD from the RTL. We use Algorithm 1 discussed in chapter 3 and subsection 3.3.4 for the RTL-FSMD extraction process. The lines 5-9 in the Algorithm represent the rewriting process. In this method, we use a Mealy Model representation. In this model, operations are associated with the state transition. Based on the transition condition, the operation performed is decided.

6.3.3 Detection

The BE attack will be identified during FSMD extraction from the RTL. The idea is to identify the Trojan pattern shown in Fig. 6.2(b) in the datapath during FSMD extraction. The trigger to identify such a pattern is when an RTL operation of the form $R \leftarrow \neg R$ (bit-flipped) is found in a state during the rewriting method. For each state s_i, there are some active micro-operations. The rewriting method takes a micro-operation in which a register R_i presents in LHS and keeps rewriting the RHS terms one by one until the RHS expression consists only of inputs, registers, or constants. If the rewriting method starts with a micro-operation $R_i \leftarrow w$ (where w is a wire signal) and stops with an RTL operation $R_i \leftarrow \neg R_i$, then we store the instance of the register in the set R_{bf} (set of bit-flipped register). Here, the RHS consists only of LHS register in negation form. We perform the following analysis to identify the attack.

We shall collect all the bit-flipped registers in a state in a set R_{bf} in each control state. We shall also collect all the idle FUs (F_{idle}) in each control state. Let assume that the FU f_i is idle in control state s_n. A FU is idle in a control state if it is not used by any of the RTL operations in that state. The same can be identified by some additional book-keeping in Algorithm 1. Since all the control signals have some value in each control state, data from some registers are coming to the inputs of an idle FU as well. We now apply the rewriting method from the output of the idle FU. This will identify the operation starting from the idle FU output (and not from a destination register). This operation for f_i is called as *input pattern* to f_i. If the input pattern of f_i contains any register from the set R_{bf}, we shall report a possible battery exhaustion attack in the state s_n with all relevant details to the user. The overall idea is that if some register's value is bit-flipped whenever the trigger signal is on and that register's value is used in some functional unit and the output of the functional unit is not utilized then that is the instance of battery exhaustion attack. The overall detection mechanism is presented as Algorithm 4.

Algorithm 4: *Detect_Battery_Exhaustion_Attack* (RTL, s_n)

```
   Result: An instance of the battery exhaustion attack in the state $s_n$.
1  Collect all bit-flipped registers in the state $s_n$ in $R_{bf}$ from $R_{Sn}$ (Ref Algorithm 1);
2  Collect all idle FUs in the state $s_n$ in $F_{idle}$.
3  for each Register $r_i$ in $R_{bf}$ do
4      for $f_i \in F_{idle}$ do
5          Apply the rewrite method from the output of the idle FU $f_i$.
6          if the input pattern of $f_i$ contains any register from $R_{bf}$ then
7              Report "A possible instance of battery exhaustion attack" with relevant
               detail.
8  Report "No battery exhaustion attack is found in the state $s_n$."
```

Example 24. *Let us now consider the example given in Fig. 6.2. Let assume that the multiplier in Fig. 6.2(a) is idle in state* $s1$. *In FSMD, the operation in a state is placed in the corresponding transitions from that state. Since the multiplier is idle in* $s1$, *the register* $r1$ *will not be updated in this state. The multiplier output can be visualized as* $fOut \leftarrow in1 \times in2$ *in normal mode as shown in Fig. 6.2(c). This is the correct version of the controller for this state transition. However, the controller FSM behaviour will be affected by the battery exhaustion attack as shown in Fig. 6.2(d). In this case, there will be two transitions controlled by the Trojan trigger* tj *from the state* $s1$ *in the HT affected design. In a normal mode when the HT is disabled, i.e.,* tj *is False, no spurious operation will be executed. However, when the Trojan is triggered, the spurious operations as shown in the transition with condition* tj *will be executed.*

During FSMD construction, the Algorithm 4 will identify the operations $r4 = \neg r4$ *and* $r5 = \neg r5$ *in line 1 and store* $r4$ *and* $r5$ *in* R_{bf}. *Since the multiplier is idle in this state, the rewriting method will identify the input pattern* $r4 \times r5$ *for the multiplier. Since* $r4$ *and* $r5$ *occur in the input pattern of the multiplier, Algorithm 4 will report a possible battery exhaustion attack. In addition, it will also report the involved multiplexers, registers, the FU for further debugging.* □

6.4 Detection of Degradation Attack

The IPs are reused in different applications of system design to reduce design costs. They are excellent candidates for hardware accelerator design. Many circuit design companies use HLS tools to create reusable IPs at high-level specifications. Therefore, an attacker is more interested in potential modification of the IPs during the design process. In the degradation attack (DA), the attacker inserts empty states in the controller FSM. As a result, the performance of the IPs will be degraded when the Trojan trigger is activated.

6.4.1 Attack Model

The degradation attack inserts a few empty states (i.e., bubble) in the controller FSM. These bubbles create a divergence in control flow (i.e. an alternative path) from a specific state s_i before coming back to the next state s_i. The transitions are controlled by the trigger signal of the Trojan. This alternative trace just consumes some extra clock cycles before coming back to the original behaviour. The circuit will perform properly in a normal scenario. The Trojan can be activated only after a predefined amount of time or in the case of a specific input sequence. When a Trojan is activated, the alternative trace is executed and it will slow down (i.e., degrade) the actual computation.

```
  int FIR(int ntaps, int sum){
    for(int i=0; i < ntaps; i++){
        sum + = h[i] * z[i];
    }
    return sum;
}
```

Figure 6.3: *C code for FIR filter*

Example 25. *Let us consider the FIR behaviour in Fig. 6.3. The bahaviour is simplified by removing a few loops related to the memory read and wait for the start signal for simplification. The corresponding C-FSMD obtained from the input C is shown in Fig. 6.4(a). The RTL-FSMD obtained from the RTL-C is shown in Fig. 6.4(b). The HLS tool inserts a bubble state t7 as shown in Fig. 6.4(b). The degradation attack creates a divergence of control flow from the state t4. In the normal scenario, the transition* $t4 \xrightarrow{!tj} t5$ *will be executed. So, there will be no degradation. The execution will follow the bubble state* ($t4 \xrightarrow{tj} t7 \rightarrow t5$)

when the Trojan trigger tj is True. As a result, for every iteration, there will be one cycle of degradation. The bubble is inserted inside the loop which iterates ntaps time. Therefore, total degradation will be ntaps cycles. □

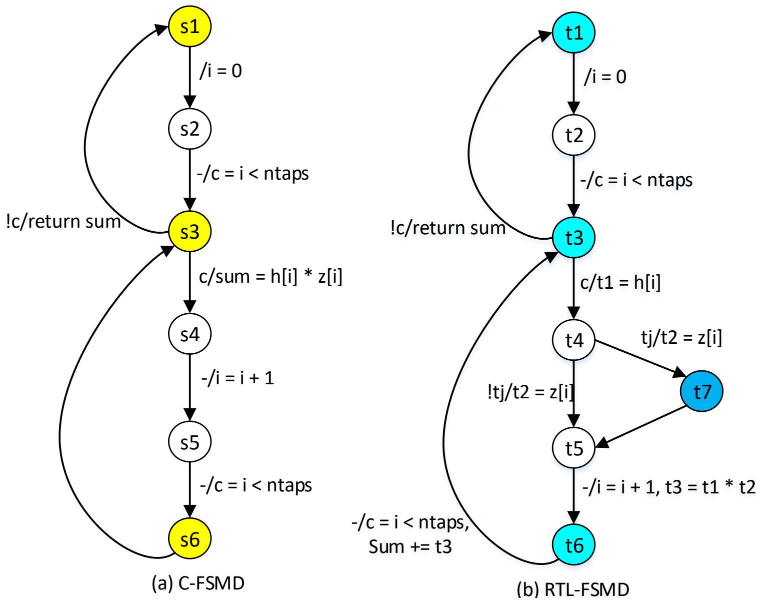


Figure 6.4: *An example to illustrate the effect of degradation attack: (a) C-FSMD obtained from the input C (b) RTL-FSMD obtained from the RTL after HT insertion*

The DA will be detected during equivalence checking of two FSMDs (between C-FSMD and RTL-FSMD). The DA changes the behaviour of the controller FSM. As a result, the number of traces has increased in the RTL-FSMD and the condition of execution of some traces has also changed. As a result, the equivalence of a few traces of C-FSMD could not be found in RTL-FSMD. In such a situation, we will analyze to find a set of traces in RTL-FSMD whose union is equivalent to the trace in C-FSMD. With further analysis of the condition of executions of those traces, the DA can be detected. In the following, we briefly discuss the equivalence checking method followed by the detection of DA.

6.4.2 C to RTL Equivalence Checking

The primary challenge in C to RTL equivalence checking is the abstraction gap between the C and the RTL codes. Therefore, RTL-FSMD is abstracted first from the RTL using Algorithm 1 in chapter 3. We also represent the input C using C-FSMD. How the FSMD model can be constructed from the input C is discussed in detail in [33].

Algorithm 5: Detect_DA_DG_Attack (C, RTL)

```
Input: C is the input C to HLS, RTL is generated by a HLS tool from C
Result: Equivalent, detect degradation or downgrade attack
1  $M_0$ = constructFSMD (C);
2  $M_1$ = RTL-FSMD_Extraction(RTL);
3  $T_0$ = findTrace($M_0$); $T_1$ = findTrace($M_1$);
4  while ($T_0 \neq \phi$) do
5      $\tau_0$ = select a trace from $T_0$;
6      TC = getTestcase($\tau_0$);
7      $\tau_1$ = getCorrespondingTrace($T_1$, TC);
8      if (($c_{\tau_0} \equiv c_{\tau_1}$) $\wedge$ ($s_{\tau_0} \equiv s_{\tau_1}$)) then
9          //$\tau_0$ and $\tau_1$ are equivalent
10         removeTrace ($\tau_1$,$T_1$);
11     else if {($c_{\tau_0} \wedge c_{\tau_1} \neq \phi$) $\wedge$ ($s_{\tau_0} \equiv s_{\tau_1}$)} then
12         Detect_DA_Attack ($\tau_0$, $T_1$);//Algorithm 6
13         if (not DA) then
14             Detect_DG_Attack ($\tau_0$, $T_1$);//Algorithm 7
15     else if {($c_{\tau_0} \wedge c_{\tau_1} \neq \phi$) $\wedge$ ($s_{\tau_0} \neq s_{\tau_1}$)} then
16         Detect_DG_Attack ($\tau_0$, $T_1$);//Algorithm 7
17     removeTrace ($\tau_0$, $T_0$);
18 if (no DA or DG) then
19     Report "No attack is found";
```

The Algorithm 5 is used to detect degradation and downgrade attacks. The equivalence checking method used here is adapted from Algorithm 2 in chapter 4. The algorithm examines whether the trace pairs of C-FSMD and RTL-FSMD are equivalent or not. In our method (Algorithm 5), we do not use the merging compatible (traces that have the same output) trace concept. The merged trace condition of execution is represented as the union of traces that are merged. As a result, the effect of HTs may not be shown. However, in the s adapted algorithm, we include techniques whose goal is to detect degradation attacks and downgrade attacks. The algorithmic steps are described briefly below.

1. Generate all traces in both the behaviours: All the traces of both behaviours input C (M_0) and RTL-c (M_1) have been extracted and assigned to the sets T_0 and T_1, respectively. The tool Klee [7] has been used for this purpose. We have modified Klee's source code to get the symbolic the data transformation (s_τ) and the condition of execution (c_τ) of a trace τ in M_0 and M_1.

2. Find potential corresponding traces between two behaviours: For checking the equivalence between M_0 and M_1, we need to check equivalence between the traces. A naive algorithm will take $O(n^2)$ comparison (n is the number of traces in a FSMD) to find the equivalence because it will compare each trace in M_0 with all traces in M_1 (to the worst case) to find the equivalence. To reduce complexity, a data-driven approach is taken to find the potential corresponding traces between T_0 and T_1. Klee gives a test case for each trace in a behaviour. Hence, we know the values of input variables (test case) for each trace τ_0 in T_0. Now, we can run T_1 with this test case and find the trace τ_1 which is followed for this particular test case. Lines 5-7 of Algorithm 5 implements this idea. This data-driven approach will reduce the complexity of equivalence checking to $O(n)$ comparisons.

3. Equivalence checking of traces between two behaviours: Finally, the trace-wise equivalence of potential correspondent traces is checked using SMT solver Z3[54] in this work. A potential corresponding trace pair are equivalent if their respective condition of executions and data transformations are equivalent (in lines 8-10). If they are not equivalent(in lines 12-14), we check for possible instance of degradation or downgrade attacks.

6.4.3 Detection

We can detect degradation attacks with the help of the equivalence checker tool. The tool will give us the trace level equivalence between the C-FSMD and RTL-FSMD. As shown in [106], this bubble effectively creates an alternative trace in the behaviour with no effective operation inside it. Therefore, our objective is to identify such spurious traces in the RTL-FSMD. It may be noted that each of these traces is associated with a condition (i.e., the trigger of the Trojan) and those conditions are not present in the initial behaviour. During equivalence checking, therefore, the equivalent trace cannot be found for these traces. Let $p : \langle q_i \Rightarrow q_j \rangle$ be one path from the state q_i to q_j in the RTL-FSMD in which the condition to enable the HT is incorporated by the attacker. In fact, there will be another parallel path (or a set of paths) from q_i in RTL-FSMD which is associated with the negation of the trigger condition. It is, therefore, possible to find two (or a set of) traces in RTL-FSMD through q_i and q_j whose union will be equivalent to the corresponding trace in C-FSMD. By analyzing the conditions of these traces, the HT trigger condition will be identified. The overall degradation attack detection mechanism is presented in Algorithm 6. The Algorithm 6 invokes in line 12 of Algorithm 5 when the equivalence of τ_0 of M_0 and τ_1 of

M_1 cannot be shown because the respective condition of executions are not equivalent but their intersection is not NULL (i.e., there is some common condition of executions between them). In Algorithm 6, we first identify a set of traces in the RTL-FSMD M_1 each of them has a stronger condition of execution than that of τ_0 of C-FSMD M_0. The stronger condition of execution is indicated by implication in line 4 of the Algorithm 6. Then, we check if the union of condition of executions of all these traces turns out to be equivalent to the c_{τ_0}. This indicates that some spurious traces may be added in the RTL-FSMD by the attacker for implementing DA or DG attacks. A trace of C-FSMD can also be split into multiple traces in RTL-FSMD during scheduling as shown in [105]. In such case, a possible DA attack will be detected which is false positive. Therefore, a careful manual inspection of the condition of executions of the traces will identify if any spurious HT trigger condition is added in the RTL-FSMD.

```
Algorithm 6: Detect_DA_Attack ($\tau_0$, $T_1$)
   Input: $\tau_0$, $T_1$
   Result: Instance of the degradation (Trojan trigger condition)
1  $c_{comb} = \phi$; //combined condition of traces
2  foreach $\tau_i \in T_1$ do
3      check if $c_{\tau_i}$ is stronger condition than $c_{\tau_0}$ and data transformations match
4      if $((c_{\tau_i} \rightarrow c_{\tau_0} \wedge \boldsymbol{s_{\tau_i}} \equiv \boldsymbol{s_{\tau_0}}))$ then
5          $c_{comb} = c_{comb} \vee c_{\tau_i}$;
6  //check the union of the strong condition $c_{\tau_i}$ is equal to condition $c_{\tau_0}$
7  if $(c_{comb} \equiv c_{\tau_0})$ then
8      Report "Possible degradation attack is found in T$_1$".
9  else
10     Report "No degradation attack is found in T$_1$."
```

Example 26. *Consider the FSMDs in Fig. 6.4. During checking equivalence from the corresponding states pair $\langle s1, t1\rangle$, the equivalence checking method (given as Algorithm 5) could not found any equivalence for a trace $\tau_0 = s1 \rightarrow s2 \rightarrow (s3 \xrightarrow{c} s4 \rightarrow s5 \xrightarrow{c} s6 \rightarrow s3)^{ntaps} \rightarrow s1$ of C-FSMD in Fig. 6.4(a) in the RTL-FSMD in Fig. 6.4(b). Then Algorithm 5 calls Algorithm 6 to check a possible instance of degradation attack. It found two traces $\tau_1 = t1 \rightarrow t2 \rightarrow (t3 \xrightarrow{c} t4 \xrightarrow{tj} t7 \rightarrow t5 \rightarrow t6 \xrightarrow{c} t3)^{ntaps} \rightarrow t1$ and $\tau_2 = t1 \rightarrow t2 \rightarrow (t3 \xrightarrow{c} t4 \xrightarrow{!tj} t5 \rightarrow t6 \xrightarrow{c} t3)^{ntaps} \rightarrow t1$ in the RTL-FSMD such that union of these two traces are* *equivalent to τ_0 and c_{τ_1} and c_{τ_2} are stronger than c_{τ_0}. After a careful inspection of c_{τ_1} and c_{τ_2}, we found the trigger condition is tj.* □

6.5 Detection of Downgrade Attack

A compromised HLS tool can inject a malicious functionality to access the behaviours of the design. This malicious functionality can be used by the actual attacker to extract useful information from the circuit. As a result, the security properties of the design like cryptography algorithms (AES, SHA) will be compromised. The level of trust of these algorithms depends on the number of rounds that are executed. If the algorithms execute some rounds below a given count, the design becomes vulnerable. It is sufficient to reduce the number of the executed round in the cryptography algorithms to compromise the security of the algorithms.

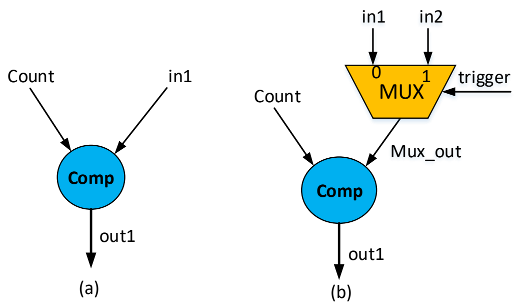


Figure 6.5: *An example to illustrate the effect of downgrade attack*

6.5.1 Attack Model

Downgrade attack changes the functionality of the input specification. The tool might reduce the number of the executed round in Secure Hash Algorithm (SHA) [14]. This can be done by modifying the loop constants in the RTL or by pre-loading a value higher than 0 as the initial value of the loop iterator. As a result, the security of the algorithm will be compromised. The value of the loop constant depends on the trigger signal of the Trojan. For example, message pairs with a collision can be generated for the SHA algorithm when the number of a round is reduced from 64 to 18 [113]. The following example explain the downgrade attack (DG).

Example 27. *Let us consider the example given in Fig. 6.5. Consider the datapath shown in Fig. 6.5(a). The input $in1$ (an input that determines the loop bound) and count (loop variable) are inputs to the comparator and the output determines whether the loop body will be executed or not. We consider only the relevant part of the datapath to explain the effect of a downgrade attack. The datapath is modified as shown in Fig. 6.5(b) to introduce the attack. Specifically, an extra constant (i.e., $in2$) is introduced and is multiplexed with $in1$. The output of the multiplexer depends on the trigger condition of the HT. If the trigger condition is True, the loop will iterate $in2$ times instead of $in1$ times. The value of the loop bound is changed based on the trigger condition.* □

Algorithm 7: Detect_DG_Attack (τ_0, T_1)

```
Input: $\tau_0$, $T_1$
Result: Instance of the downgrade (Trojan trigger condition)
1  $c_{comb} = \phi$; //combined condition of traces
2  foreach $\tau_i \in T_1$ do
3      //check if $c_{\tau_i}$ is stronger condition than $c_{\tau_0}$
4      if $(c_{\tau_i} \rightarrow c_{\tau_0})$ then
5          $c_{comb} = c_{comb} \vee c_{\tau_i}$;
6  //check the union of the strong condition $c_{\tau_i}$ is equal to condition $c_{\tau_0}$
7  if $(c_{comb} \equiv c_{\tau_0})$ then
8      Report "Possible downgrade attack is found in T$_1$".
9  else
10     Report "No downgrade attack is found in T$_1$."
```

```
#define nb 64
  int AddRoundKey(int stmt[nb], int
      ret[nb]){
    for(int i=0; i < nb; i++){
        stmt[i*4] = ret[i*4];
    }
  }
```

(a)

```
#define nb 64
  int AddRoundKey(int stmt[nb], int
      ret[nb]){
    nb = tj? 32 : nb;
    for(int i=0; i < nb; i++){
        stmt[i*4] = ret[i*4];
    }
  }
```

(b)

Figure 6.6: *(a) C Code from AES before adding tj, (b) Representative RTL-C after downgrade attack*

Since the functionality has changed in this attack, our equivalence checking method will report a possible non-equivalence between the C-FSMD and the RTL-FSMD. In addition to that, the traces for which equivalence could not be found by the method will also be reported. These traces are corresponding to the block where functional changes are made by the tool. An attacker in the foundry or a rogue user can activate the Trojan after a predefined amount of time or by a specific input sequence. With the careful inspection of this information, the user can pinpoint the Trojan inserted by the HLS tool.

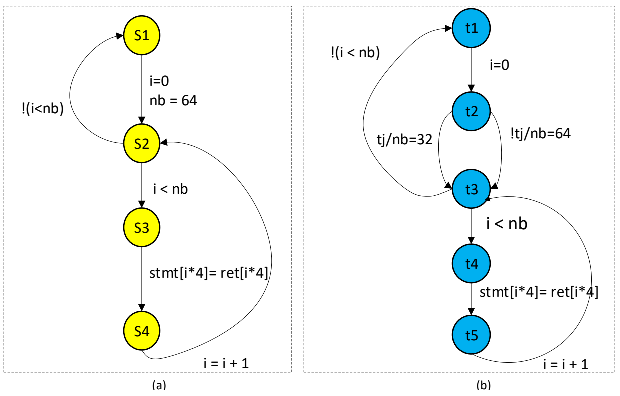


Figure 6.7: *An example to illustrate the effect of downgrade attack: (a)C-FSMD (M_0) obtained from the input C (Figure 6.6(a)) (b) RTL-FSMD (M_1) obtained from the RTL-C (Figure 6.6(b))*

6.5.2 Detection

Downgrade attack (DG) compromises the circuit behaviour only when the Trojan is activated. As long as the Trojan trigger is not activated, the design produces correct results and the Trojan stays undetected. We use our equivalent checker tool to detect the HT trigger condition for DG. In a DG attack, the number of traces in M_0 and M_1 are different due to the HT trigger condition. Specifically, M_1 contains more traces than that of M_0. The overall downgrade attack detection mechanism is presented in Algorithm 7. Algorithm 7 is invoked in Algorithm 5 when our equivalent checker could not find equivalent of a trace of τ_0 of M_0 in M_1 in two scenarios (in line 13 and in line 15). In the former case (line 13), the respective condition of executions have some overlapping but are not equivalent but the data transformations are equivalent. In the latter case (line 15)), both conditions of executions and the data transformations are not equivalent. In Algorithm 7, we first identify a set of traces τ_i in the RTL-FSMD for which the condition of execution is stronger (stronger condition is indicated by implication) than that τ_0. Then, we check if the union of the condition of executions of all these traces c_{comb} turns out to be equivalent to the condition of executions of c_{τ_0}. This indicates that some spurious traces may be added to the RTL-FSMD by the attacker for implementing the DG attack. By a careful analysis of the traces involved in c_{comb}, we can figure out the spurious HT trigger condition of the downgrade attack, and the value of the loop counter is changed.

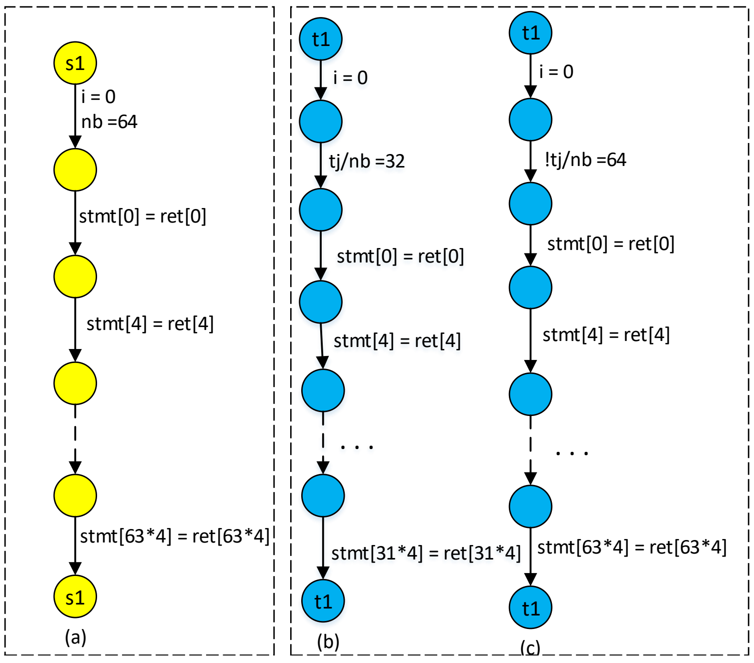


Figure 6.8: *An example to illustrate the effect of downgrade attack: (a) Trace of τ_{00} from C-FSMD obtained after unrolling (nb = 64) (b) Trace of τ_{10} from RTL-FSMD obtained after unrolling (nb = 32) (c) Trace of τ_{11} from RTL-FSMD obtained after unrolling (nb = 64)*

Example 28. *Consider the input C-code and the corresponding RTL-C (after HLS tool inserts Trojan) as shown in Fig. 6.6(a) and Fig. 6.6(b), respectively, and their corresponding FSMDs are shown Fig. 6.7(a) and Fig. 6.7(b), respectively. The traces of the respective behaviours are shown in Fig. 6.8. Specifically, the C-FSMD has one trace and the RTL-FSMD has two traces. During equivalence checking between traces in M_0 and M_1, the equivalence of τ_{00} could not be found. But, the Algorithm will select τ_{10} (or τ_{11}) and Algorithm 7 will be called in line 13 (or line 15). Since $c_{\tau_{10}}$ and $c_{\tau_{11}}$ are stronger than $c_{\tau_{00}}$ and $(c_{\tau_{10}} \vee c_{\tau_{11}}) \equiv c_{\tau_{00}}$, the Algorithm 7 will report a possible DG attack. After a careful inspection of the data transformation and condition of execution of of τ_{10} and τ_{11}, we identify* *that the hardware Trigger condition tj and the value of loop count (variable) is decided by tj. The loop executes a reduced number of rounds (from 64 to 32) in case of HT activated. In addition to loop count, the actual operations are also affected by the modification of the loop count.* □

6.6 Experimental Results

Implementation detail and Experimental setups: Our HT detection framework BLAST is implemented in Python. The BLAST first extracts an abstract syntax tree (AST) from the Verilog using pyVerilog [9] parser and then implemented the rewriting method on the AST to obtain the RTL-FSMD. Specifically, we have adapted the FastSim [15] to extract the RTL-FSMD in our work. The RTL-FSMD of our flow and the RTL-C of FastSim represents the same reverse engineering high-level behaviour of RTL. The C-FSMD is extracted from the input C behaviour. The experiments have been performed on a machine with a CPU: Intel Core i7, 2.5GHz, and 8GB RAM on a set of HLS benchmarks. We have used the Vivado HLS tool [10] to generate Verilog RTL for the benchmarks written in C. We then manually inserted all three hardware Trojans (battery exhaustion, degradation, and downgrade attacks) on the RTLs and generate various versions of the RTL[1]. *In battery exhaustion attack*, the attacker is interested in compromising the execution of the design by draining out the power. In HLS, the number of FUs presented in datapath is greater or equal to the number of operations scheduled in a control state. As a result, some of the FUs will remain idle in some control states. We reused these idle FUs to execute fake operations to implement a battery exhaustion attack in the HLS generated RTLs. If no idle FUs are available, we manually inserted an additional state in the control FSM and implemented the attack on that state. In the case of *degradation attack*, we inserted a few empty states in the control FSM and add the corresponding transitions. These empty states create alternative paths in the behaviour when a Trojan is activated. The alternative path consumes extra clock cycles. As a result, it will slow down the actual computation. *In downgrade attack*, we identified the loop bound constant. We provided an additional constant to control the loop iterations. We then added the HT trigger condition to activate the HT scenario. We have not added any trigger circuit in the RTLs to activate the HTs. We perform the RTL

[1]The BlackHat HLS tool [106] is not available online. Therefore, we have implemented the same idea on the Vivado HLS generated RTLs.

simulation to ensure the functional correctness of the modified RTLs when the HTs are not activated and the desired behaviours when the HTs are activated.

Table 6.1: *Experimental Results of Battery exhaustion degrade, and downgrade attacks for different high-level synthesis benchmarks*

Bench (1)	Type (2)	#C (3)	#RTL (4)	#RTLC (5)	#TC-SC (6)	#TC-RTLC (7)	#Instance (8)	#DT (s) (9)
Parker	BE	56	235	366	12	271	2	2.304
	DA	63	200	358	12	311	1	133.300
Waka	BE	34	309	741	3	2	1	1.802
	DA	34	294	646	3	5	1	2.46
ARFNC	BE	53	381	844	4	6	1	2.002
	DA	65	385	971	4	8	2	3.887
ARFNB	BE	62	477	1559	1	2	2	1.76
	DA	62	455	1680	1	2	2	2.63
Motion	BE	50	395	339	1	1	2	2.248
	DA	50	376	1056	1	2	1	3.152
Array_add	BE	19	285	790	1	2	1	0.064
	DA	19	263	614	1	2	1	0.299
	DG	19	275	541	1	2	2	0.101
FMM	BE	30	446	1716	81	93	2	0.147
	DA	40	419	1955	81	16	2	191.401
Find_min	BE	40	208	376	128	63	1	753.347
	DA	40	187	354	128	66	1	789.835
DFadd	DA	554	1724	2132	67	69	1	797.3
AES	BE	979	2799	4784	-	-	1	0.487
DES	BE	354	2330	2856	-	-	1	0.439

Experiments: We evaluated our method on a variety of HLS benchmarks (Waka, Motion, Parker, Array add, Find min, FMM, and auto-regressive lattice filter with branch (ARFNC) and without branch (ARFNB)), each of them is written in C-code. The benchmarks are taken from the distribution of Bambu HLS [2]. The experimental results of our benchmarks are shown in Table 6.1. The 2^{nd} (type) and 3^{rd} (#C) columns show the type of attacks and the number of input C code lines for each benchmark, respectively. We have recorded the number of code lines (#RTL) of RTL and the number of code lines (#RTLC) of RTL-C in 4^{th} and 5^{th} columns, respectively. The 6^{th} and 7^{th} columns are the trace count (#TC) in the source C and RTL-C, respectively. It may be noted that the trace count is not

Table 6.2: *Comparisons of area overhead for RTL (original) with respect to the RTL(BE), RTL(DA) and RTL(DG)*

Bench mark	Device utilization	RTL codes						
		RTL (Original)	**RTL (BE)**	**Overhead**	**RTL (DA)**	**Overhead**	**RTL (DG)**	**Over head**
Parker	#Slices	332	366	+0.55%	340	+0.13%	-	-
	# Flip Flops	108	166	+0.47%	120	+0.1%	-	-
	#LUTs	624	697	+0.59%	635	+0.12%	-	-
Waka	#Slices	345	398	+0.86%	354	+0.14%	-	-
	# Flip Flops	196	228	+0.26%	207	+0.09%	-	-
	#LUTs	657	754	+0.79%	662	+0.04%	-	-
ARFNB	#Slices	2292	2315	+0.37%	2304	+0.2%	-	-
	# Flip Flops	728	840	+0.91%	810	+0.67%	-	-
	#LUTs	4163	4220	+0.47%	4198	+0.29%	-	-
ARFNC	#Slices	309	328	+0.31%	315	+0.1%	-	-
	# Flip Flops	325	387	+0.51%	340	+0.13%		
	#LUTs	464	512	+0.39%	489	+0.2%		
Motion	#Slices	1008	1882	+14%	1774	+12.5%	-	-
	# Flip Flops	458	599	+1.14%	529	+0.59%	-	-
	#LUTs	1659	3362	+13.86%	3032	+11.17%		
Array_add	#Slices	21	87	+1.1%	31	+0.16%	21	+0.0%
	# Flip Flops	20	90	+0.57%	32	+0.1%	25	+0.04%
	#LUTs	61	183	+0.99%	91	+0.24%	57	-0.036%
FMM	#Slices	145	168	+0.37%	159	+0.23%	-	-
	# Flip Flops	113	157	+0.31%	139	+0.21%	-	-
	#LUTs	271	307	+0.3%	292	+0.2%	-	-
Find_min	#Slices	316	347	+0.51%	331	+0.25%	-	-
	# Flip Flops	143	167	0.2%	157	+0.12%	-	-
	#LUTs	271	293	0.17%	307	+0.3%	-	-
DFadd	#Slices	1897	1955	0.93%	1923	0.42%	-	-
	# Flip Flops	1055	1135	0.65%	1107	0.43%	-	-
	#LUTs	3655	3710	0.47%	3689	0.32%	-	-
AES	#Slices	1600	1654	0.88%	1625	0.41%	-	-
	# Flip Flops	1491	1552	0.56%	1519	0.24%	-	-
	#LUTs	2135	2210	0.64%	2175	0.40%	-	-
DES	#Slices	2294	2355	0.97%	2318	0.46%	-	-
	# Flip Flops	2378	2415	0.65%	2397	0.16%	-	-
	#LUTs	3490	3542	0.85%	3508	0.15%	-	-

measured for BE attack since it is identified in RTL-FSMD extraction. The number of instances (#instance) of HT insertion is given in column 8^{th}. The column 9^{th} represents the HT detection time (in seconds) by our tool BLAST. Each row (BE/DA/DG) represents a battery exhaustion/degradation/downgrade HT scenario created from the original RTL. In the case of an exhaustion attack, the detection time is less compared to other attacks because BE attack is identified during the FSMD extraction phase. For the FMM example, although, the traces are high in FMM, the BE attack in FMM is detected in the FSMD extraction phase itself. So, equivalence checking is not needed to detect BE attacks in FMM. The DA attack detection takes more time in Parker, Find_min, and FMM, as compared to the DA attack detection in other test cases because the equivalence checking, is taking more time since the number of traces are more in Parker, Find_min, and FMM compared to other test cases. In all cases, HT attacks are correctly detected by our proposed method. Generally, the detection time for our framework is not high. We have tested that the runtime of our tool is not much impacted by the applications of HLS optimizations on our benchmarks since the overall steps to be checked are mostly the same in all cases.

HT Overheads: We synthesized the original RTL code (RTL(original)), RTL code after the inclusion of BE (RTL(BE)), DA (RTL(DA)), and DG (RTL(DG)) to check the resource utilization overheads of HT implementation. We evaluated the attacks implemented in Vivado HLS generated RTL code. All the designs were synthesized for Virtex4 XC4VCX15 series FPGA. From the device utilization summary report obtained after synthesis, we calculate the overhead (Slices, Flipflop and LUT) needed by the additional logic added to implement BE, DA, and DG in the original RTL. Table 6.2 presents device utilization summary and area overhead of RTL(BE), RTL(DA) and RTL(DG) as compared to the RTL(original). As shown, the hardware needed to implement RTL(BE), RTL(DA), and RTL(DG) is slightly more than the hardware needed to implement RTL(original). The area overhead is less than 1% in most cases. In general, these results show that the HTs minimally impact the area. Similarly, a comparison of minimum input arrival time before clock (MIATBC), maximum output required time after clock (MORTAC), and maximum combinational path delay (MCPD) of RTL(BE) and RTL(DA) compared to RTL(original) obtained from timing report after synthesis and extra delay added by the inclusion of HTs is shown in Table 6.3. This report only provides estimated results. It can be observed that on an average 1ns increase in the delay. Hence, we conclude that extra logic added to implement HTs into the original RTL minimally impacts the area and speed of the design.

Table 6.3: *Comparisons of increase in delay for RTL (original) with RTL(BE) and RTL(DA)*

Benchmark	Time Summary(ns)	**RTL codes**				
		RTL (original)	**RTL (BE)**	**Increase in delay**	**RTL (DA)**	**Increase in delay**
Parker	MIATBC	6.425	7.724	1.299	7.969	1.544
	MORTAC	10.785	11.857	1.072	11.913	1.128
	MCPD	9.625	10.574	0.949	10.689	1.064
WAKA	MIATBC	5.585	6.714	1.129	6.925	1.340
	MORTAC	7.921	8.985	1.064	9.566	1.645
	MCPD	7.243	8.123	0.880	8.964	1.721
Motion	MIATBC	9.265	10.148	0.883	10.282	1.017
	MORTAC	12.709	13.814	1.105	13.965	1.256
	MCPD	9.869	10.871	1.002	11.120	1.251
Array_add	MIATBC	2.151	4.179	2.028	4.653	2.502
	MORTAC	5.212	6.871	1.659	6.789	1.577
	MCPD	4.947	5.986	1.039	6.125	1.178
DFadd	MIATBC	2.880	4.200	1.320	4.542	1.662
	MORTAC	5.330	6.812	1.482	7.152	1.822
	MCPD	4.947	6.100	1.153	6.512	1.565
AES	MIATBC	1.900	2.810	0.91	3.010	1.110
	MORTAC	5.781	6.662	0.881	6.753	0.972
	MCPD	4.977	6.170	1.193	6.341	1.364
DES	MIATBC	2.891	3.743	0.852	3.934	1.043
	MORTAC	5.542	6.336	0.794	6.482	0.94
	MCPD	4.943	5.891	0.948	5.982	1.039

6.7 Performance of BLAST for HLS Optimizations

Modern day HLS tools are equipped with various software and hardware oriented optimizations to provide efficient hardware from a given C/C++ behaviour. In the front-end of the HLS tool, a compiler like GCC or LLVM is used to parse the input bahaviour. Since these C/C++ compilers consist of hundreds of software code optimizations, they are now available in the HLS tool. On the other hand, the HLS tool applies various hardware oriented optimizations like array partitioning, loop pipelining, loop unrolling, data flow optimizations, etc. in the back-end of HLS to make the input C/C++ code hardware efficient. In this section, we have analyzed how HLS optimizations affect the performance of the BLAST framework.

The BLAST has two phases - an RTL-FSMD extraction phase and a C to RTL equivalence checking phase. The RTL-FSMD extraction phase is relied on the FastSim [15] tool. This tool is equipped to handle all kinds of optimizations applied in HLS. To detect BE attack, BLAST essentially adds a module (i.e., Algorithm 4) in FastSim flow to analyze the BE attack in presence of bit-flipped operations in a state as discussed in Subsection 6.3.3. Therefore, the BE attack can be detected by BLAST irrespective of what HLS optimizations are applied by the HLS tool. Since BLAST analyzes the RTLs generated by the HLS tool in a state-wise manner of the controller FSM, the run time of BE attack detection is not impacted much by the applications of HLS optimizations. Usually, the BE attack is identified in seconds.

The DA and DG attacks detection rely on the C to RTL equivalence checking in which the RTL-FSMD extracted in the phase one is formally compared with the input C behaviour (i.e., C-FSMD). Since BLAST checks the trace level equivalence between these two behaviours, a major change in the control flow due to HLS optimizations will impact DA and DG attacks detection probability. The front-end optimizations like constant propagation, copy propagation, common sub-expression elimination, dead code elimination, static single assignment, code motion, operator strength reduction (e.g., multiplication by constant is replaced by left shift by constant), etc. mostly impact the data dependence in the behaviour. Such optimizations do not impact much on the control flow of the input behaviour. Therefore, the performance of BLAST won't be impacted by applications of such software optimizations in the front-end of the HLS.

Let us now discuss the hardware oriented optimizations. The array partitioning essentially breaks an array into multiple arrays to map them into multiple RAMs in order to improve memory access time. The array merging is the reverse process of array partitioning. In our case, the RAMs are represented as arrays in RTL-FSMD. So, we have two behaviours where the number of intermediate arrays are different. The control structure of the input behaviour is not impacted by this optimization. Therefore, array partitioning/merging won't impact our DA and DG detection. Loop unrolling unrolls the loop of input C. In algorithm 5, we use Klee to identify traces in the behaviours. Klee unrolls loops to identify the traces. Although loop unrolling changes the control structure, it won't impact the detection of DA and DG attacks in BLAST since loops are unrolled during detection.

The loop pipelining creates multiple stages within a state where each stage works on the data of different iterations of the loop. This helps in running the multiple iterations of the loop in parallel to improve the latency. For a pipelined function, the pipelined stages

```
stage1:
  a = I1 + 10;
  b = I2 + 5;
stage2:
  c = a + b;
state3:
  d = c * c;
        (a)
```

```
State 1:
  //Code to update stage1, stage2 and stage3 flags
  a_t = a, b_t = b, c_t = c;
  if(stage1){
       a = I1 + 10; b = I2 + 5; }
  if(stage2)
    c = a_t + b_t;
  if(stage3)
    d = c_t * c_t;
        (b)
```

Figure 6.9: *Representation of pipelined loop in C*

work in a similar manner. The FastSim creates a sequential representation of the pipelined stages with suitable logic to handle the inherent dataflow among the subsequent stages. Consider the example in Fig. 6.9 to understand the fact. Assume the operations within a loop body are scheduled in three pipeline stages as shown in Fig. 6.9(a). The corresponding RTL-FSMD behaviour is shown in Fig. 6.9(b). Each pipeline stage is activated by a flag. In the first clock, only stage 1 is active and in the second clock, both stage 1 and stage 2 are active. From the third clock, all stages are active. FastSim copies the value of each intermediate variable into a temporary variable and uses them in the right-hand expression of the operations. Consequently, at i^{th} clock, stage 1 works on i^{th} inputs, stage 2 works on the $(i-1)^{th}$ data and state 3 works on $(i-2)^{th}$ data. During equivalence checking between C-FSMD and RTL-FSMD, such pipelined loop will result in a single trace. The corresponding loop of C-FSMD also results in a single trace. Thus, there won't be any change in the control flow between C-FSMD and RTL-FSMD in presence of loop pipelining. Therefore, BLAST can detect DA and DG attacks in presence of loop and function pipelining.

In data-flow optimization, the producer-consumer relation between various modules in the input C code is identified and such modules are executed in parallel in RTL. The first-in-first-out (FIFO) or Ping-Pong buffer is used between a producer-consumer pair for asynchronous data communication between them. To model such parallel behaviour in RTL-FSMD (which is a sequential behaviour), we extract the RTL-FSMD for each module first. We then generate a global RTL-FSMD in which one of the states of each module will be executed in each clock[1]. The next state to be executed in a module is determined by the state transition of RTL-FSMD of the corresponding module. The detail of such modeling

[1]Since RTL-FSMD is a cycle accurate model, operations executed in each clock can be tracked.

may be found in [15]. Since the control flow of the RTL-FSMD in presence of data flow optimization is completely different from that of the C-FSMD, BLAST cannot detect DA or DG attach in such a scenario. Specifically, the Algorithm 5 returns false-negative (in line no 19) if dataflow optimizations are applied. In general, if the control flow of the input behaviour is modified significantly by HLS, BLAST may return false-negative results. We have used Klee to obtain the traces in a program (line 3 of Algorithm 5) and Z3 SMT solver for checking the equivalence of traces in our Algorithms. So, the run time of BLAST largely depends on these two tools to detect DA and DG attacks.

6.8 Conclusion

In this Chapter, we presented the BLAST framework to detect high-level synthesis tool inserted Hardware Trojan in the RTL. The strength of our detection framework is the ability to construct a C-like behavioural specification from the RTL generated by the HLS tool. This helps us to correlate the RTL with the input C-specification. We have shown that all three Trojan inserted by the Black-hat HLS tool are detected in our framework. Our framework will identify the possible instances of the HT. The developer can identify the actual HT with further analysis. We have also analyzed the area and delay overheads of our benchmarks. We concluded that the inclusion of HTs minimally impacts the area and the delay. The experimental results for a commercial HLS tool for several HLS benchmarks show that our method can efficiently detect the attacks automatically without taking any information from the HLS tool.

Based on the impact of HTs, they can be categorized into (i) change functionality (ii) degrade performance (iii) leak information, and (iv) denial of service. While denial of service type HT will be hard to implement at HLS, the others HTs are possible. In fact, the BE attack degrades performance by consuming more power, the DA attack degrades performance by increasing the latency of the design and the DG attack changes the functionality. Similar to BE and DA attacks, the other attacks can also be formulated to leak information about secret data in terms of power or latency. The novelty of our work lies in the fact that we have identified the generic impacts of the HTs at a higher abstraction level. Specifically, the HTs result in either (i) executing spurious bit-flipping operations around an idle FUs in a state, or (ii) any HT trigger condition effectively creates branches in the controller FSM. This results in a situation where a set of traces are created from a single trace of the input behaviour. The approach proposed in this work is trying to find these two scenarios. Therefore, BLAST be should able to detect any kind of HTs that changes functionality or degrades performance, or leak information.

7 Conclusions and Future Work

In this chapter we summarize the work done, highlight the contributions, and suggest the directions for possible future work.

7.1 Summary of Contributions

The contributions of this thesis are summarized below:

FastSim - A Fast Simulation Framework for High-Level Synthesis: The state-of-the-art commercial HLS tools provide C simulation and RTL co-simulation framework for behavioural and functional design verification. The RTL co-simulation incurs undesirable time overhead and the hardware's detailed cycle accurate simulation of the RTL is incomprehensible to non-FPGA experts. Consequently, some research works have been devoted to converting synthesizable Verilog RTL to C/C++. However, the generated C/C++ code is complex in terms of comprehension of code behaviour and incurs performance hampering redundancies and also is not cycle accurate and is not capable to estimate design performance. In Chapter 3, we presented an automatic cycle accurate and performance estimation simulation framework *FastSim* for HLS generated RTL to C conversion. The Verilog RTL is represented as an abstract syntax tree (AST) format using the PyVerilog [9] library and the replacement of the C incompatible constructs of Verilog like bit-select and part select with equivalent compatible representations is done on AST. Next, the register transfer operations at each control step of the controller FSM is identified to generate an equivalent FSMD code. Finally, the FSMD code and the variables, controller state, RTL operations, etc. are mapped into appropriate data types and the equivalent C code. FastSim is equipped to handle the impact of advanced optimizations like loop pipelining, data-flow, and other optimizations available in modern day HLS tools. It is shown through experimentation that our tool is cycle accurate, ensures RTL correctness, and on an average around 300 times faster simulation compared to RTL simulators and comparable performance to that of software C simulators.

DEEQ: Data-driven End-to-End EQuivalence Checking of High-level Synthesis: Although HLS comes with huge advantages, it is still a complex translation process to generate RTL. As a result, its correctness becomes a major barrier to its wide adaptation. Therefore, developing techniques to check equivalence between a behavioural specification and its synthesized RTL implementation is critical for wide applications of HLS. Although many path-based approaches have been proposed for verification of HLS, the end-to-end (input C/C++ code and its corresponding RTL) equivalence checking of HLS is not well established yet. In Chapter 4, we presented a HLS verification framework called DEEQ for C to RTL equivalence checking. In the proposed method, a high-level specification is extracted from the RTL. We introduced merging compatible traces within a behaviour, finding corresponding traces between two behaviours using a data-driven approach, and formulating a satisfiability problem that is strong enough to prove the equivalence of corresponding traces of input C/C++ and corresponding RTL. Correctness issues of the method have been dealt with. The experimental results also show that our framework DEEQ can validate the end-to-end equivalence for a commercial HLS tool for several benchmarks.

REVAMP: Reverse Engineering Register to Variable Mapping in High-level Synthesis: The variables of a high-level behaviour are mapped to hardware registers during register allocation (RA). Reverse engineering such mapping is a challenging task due to possible one register may store more than one variable provided their lifetimes do not overlap or one variable may be split into more than one register for better mapping. Daikon and SMT based frameworks for reverse engineering of the register to variable mapping in high-level synthesis are presented in Chapter 5. In the Daikon based framework, first, we introduced a method to extract schedule C code from the scheduling information. We also presented how to use Daikon to find invariants at each state on the combined program (Schedule C code

(SD-C) and high-level behaviour (RTL-C) from the RTL). Then the state-wise mapping between registers in RTL-C and variables in SD-C is extracted. Since Daikon based reverse engineering of the register to variable mapping relies on Daikon's invariant generation, there is no formal guarantee of the mapping obtained. In SMT based frameworks for reverse engineering of the register to variable mapping, we use SMT solver Z3 to find the register to variable mapping. Specifically, the mapping between the variables of the input C code and the registers of the RTL is extracted in this method. The mapping problem is formulated as SAT on the SSA versions of C and RTL-C. Although there is a formal guarantee of the correctness of the extracted mapping, scalability is not guaranteed. The prototypes of both frameworks have been implemented and tested on several HLS benchmarks. With this mapping information, we can rewrite the RTL-C code in terms of variables and finally generates an equivalent C-code. Experimental results showed that the proposed frameworks could generate equivalent C code from RTL in reasonably small time.

BLAST: Belling the Black-Hat High-level Synthesis Tool: Due to the increasing cost of ICs manufacturing foundries, most ICs manufacturing companies do not have their own foundry. As a result, most of the ICs companies outsource the fabrication of the ICs to a third-party foundry. Manufacturing ICs from a third-party raised concerns over potential malicious modification of the ICs (Hardware Trojans) during the fabrication process. The HTs may also be inserted in any phase of the design cycle by an untrusted computer aided design (CAD) synthesis tool. Most of the HTs are hard to detect since they are activated by a rare condition. Since the HTs may be inserted at any phase of the design cycle, it is important to detect them early in the design process. In Chapter 6, we proposed a formal framework for HLS Trojan detection. Specifically, the framework can detect battery exhaustion attacks, degradation attacks, and downgrade attacks inserted by a malicious HLS tool. Specifically, BLAST detects HTs at RTL. A battery exhaustion attack is identified during FSMD construction from the RTL. The equivalence checking method proposed in this chapter can detect degradation attacks and downgrade attacks. We also have analyzed the area and delay overheads of our benchmarks after inclusion of HTs, and inclusion of HTs minimally impacts the area and the delay. The experimental results show that our methods detect HTs of the black-hat HLS tool.

.

7.2 Future Perspectives

In this section, possible future works for further improvement of the proposed methods and possibilities of application of our method in other domains are discussed.

- *Enhancement of FastSim:* The work in Chapter 3 of this thesis presented a framework called *FastSim* for the conversion of HLS generated Verilog RTL to equivalent C code. However, our current version of *FastSim* cannot handle datawidth more than 64 bits. Our framework *FastSim* can further be enhanced to handle data width more than 64 bits. In addition, the current version of our simulator fails to simulate hierarchical task level pipelining and sequence of level triggered operations in a single state. In future, we would like to improve *FastSim* to accommodate such pipeline variants by extending our approach given in Chapter 3. Finally, we also plan to make *FastSim* a complete open source package for industrial and academic HLS design verification.

- *Scalability of DEEQ:* The work presented in Chapter 4 of this thesis is the first one completely automated framework for C to RTL equivalence checking for HLS without taking any input from the HLS tool. However, our current version of *DEEQ* needs scalability improvement. We observed that SMT solver fails to show the equivalence of traces. In future, we plan to use techniques like invariant-sketching and query decomposition of the SMT formula [51] to further improve the scalability of our equivalence checker.

- *Detection of other hardware Trojans:* The hardware Trojans detection framework *BLAST* proposed in Chapter 6 can detect all three HLS Trojans presented in the Black-hat HLS tool [106]. Currently, we manually inserte hardware Trojans (battery exhaustion, degradation and downgrade attacks) on the RTL and generate various versions of the RTLs. In the future, we plan automate this process on the HLS generated RTLs. We plan to use the scheduling information to insert a predefined number of bubbles defined by the HLS developer automatically. In the case of battery exhaustion attack, the idle functional units in each clock cycle can be determined from the allocation and binding information of HLS. In the future, we also plan to develop other possible hardware Trojans that can be inserted by HLS tool in our automated HT insertion framework and then check if BLAST can detect such new hardware Trojans.

- *Fuzzing the HLS tool:* It has been identified that HLS tools are inherently prone to introducing bugs in the final RTL functionality [37]. Recently a fuzzing [72] campaign on four popular HLS tools reveals that "all four HLS tools can be made to generate wrong designs from valid C programs and one tool could be made to crash". In [72], HLS tools are run on automatically generated C programs and is checked whether that the RTL generates the same output as the input C program. Although fuzzing usually does not check the correctness of the output, this work reveals an interesting approach to find bug in HLS tool. This work fuzzes the HLS tool rigorously. However, it is also important to rigorously fuzz the RTL produced by an HLS tool from an real application to find if there is any corner bugs in the RTL. Since RTL simulation is slow, the recent trend [122] is to generate a C model from RTL and fuzz the RTL using software fuzzing tools. So, the RTL-C obtained from HLS generated RTL in this work can be used to fuzz the RTL at the software level to identify bugs. It will be interesting to explore how an RTL bug is behaved in software level Fuzzing.

- *Reporting Critical Path at Behavioural Level:* In circuit design, the critical path is the longest combinational path in the circuit must be completed within the target clock period. If a critical path delay is greater than the target clock period, measures has to be taken to reduce critical path delay. In context of HLS, if generated RTL does not meet the target clock period, the user has to apply appropriate optimizations or sometime has to rewrite the input specification to improve the timing of the resultant RTL. Therefore, identifying and reporting critical paths in a C equivalent behaviour of RTL (RTL-C) would help the algorithm developers to take appropriate measure at higher abstraction level. This could be another interesting application of our reverse engineering framework.

- *Critical Path Aware HLS Locking:* To mitigate an attacker from IP piracy and counterfeiting, the RTL obfuscation/locking technique can be applied. Logic locking is a technique by which the underlying intellectual property of hardware is modified to intentionally conceal or lock its functionality in order to prevent an untrusted party from reverse-engineering the hardware design. In recent time, there is an interesting use of HLS tool for hardware obfuscation to create secure description at structural and functional level. the logic locking can be performed on input C [22], during HLS [108] or at the output RTL [78]. However, these methods lock design without knowing the critical path of the design. Precisely, these technique inserts logic for keys by randomly picking constants, operations or constants from the specification. Although, the area overhead is not significant of such locking, the additional logic may be inserted into the critical path of the design. As a result, it might make the critical path worse. In order to minimize timing degradation due to logic locking, such insertion should be done only on noncritical paths. To achieve that, the critical path information must be available at C or during HLS. One approach could be run HLS once on the input C, identify the critical paths, lock C accordingly or instruct HLS tool not to add locking in certain positions and run HLS again. Identifying the critical paths and reporting them to C level, our RTL to C reverse engineering process will be useful.

7.3 Conclusion

This dissertation presents an automatic RTL to C reverse engineering frameworks by taking advantages of the RTL structure generated by the HLS tool. Our generated RTL-C code guarantees fast simulation, functional correctness of the RTL, cycle accuracy, accurate performance estimation and generate a highly readable and debug friendly simulation code. We have shown four applications of our reverse engineering framework. We truly believe that our FastSim framework has the potential to influence future HLS simulation routines. We are pretty sure it will stimulate further works in the field of high-speed simulators for HLS. To make it happen, we make our existing results available at `https://github.com/ckarfa/FastSim`.

Publications Related to Thesis

1. Mohammed Abderehman, Jayprakash Patidar, Jay Oza, Yom Nigam, TM Ab- dul Khader, and Chandan Karfa. “FastSim: A Fast Simulation Framework for High-Level Synthesis”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1371-1385, May 2022.

2. Mohammed Abderehman, Rakesh Theegala Reddy and Chandan Karfa. “DEEQ: Data-driven End-to-End Equivalence Checking of High-level Synthesis”, *23rd IEEE International Symposium on Quality Electronic Design (ISQED’22)*, Pages 64-71, 2022.

3. Mohammed Abderehman, Rupak Gupta, and Chandan Karfa. “Reverse engineering register to variable mapping in high-level synthesis.” In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 37–42, 2021.

4. Mohammed Abderehman and Chandan Karfa. “An SMT-based Reverse Engineering of Register Allocation in High-level Synthesis”, *in 5th International Symposium on Devices, Circuits and Systems (ISDCS 2022)*, Springer, 12 Pages, March 2022.

5. Mohammed Abderehman, Rupak Gupta, Theegala Rakesh Reddy and Chandan Karfa, “BLAST: Belling the Black-Hat High-level Synthesis Tool,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (ESWEEK-TCAD special issue), 12 pages, July 2022.

6. Mohammed Abderehman, Chandan Karfa “Reverse Engineering of High-level Synthesis and Its Applications” (Communicated to an ACM TODAES)

Bibliography

[1] Arc international, vtoc for soc models. http://www.arc.com/. [Pg.34]

[2] Bambu tool reference. https://panda.dei.polimi.it/. [Pg.64], [Pg.133]

[3] Catapult high-level synthesis. http://calypto.com/en/products/catapult/catapult_overview/productinfo. [Pg.33], [Pg.69]

[4] Daikon. https://plse.cs.washington.edu/daikon/. [Pg.17], [Pg.29], [Pg.90], [Pg.91], [Pg.112]

[5] Hls arbitrary precision types library. https://github.com/Xilinx/HLS_arbitrary_Precision_Types. [Pg.49]

[6] Intel.(2019) intel fpga sdk for opencl pro edition. https://www.intel.com/. [Pg.33]

[7] Klee. https://github.com/klee/klee. [Pg.77], [Pg.80], [Pg.84], [Pg.96], [Pg.125]

[8] Modelsim hdl simulator. https://www.mentor.com/products/fv/modelsim/. [Pg.63]

[9] pyverilog. https://pypi.org/project/pyver/. [Pg.16], [Pg.36], [Pg.39], [Pg.63], [Pg.84], [Pg.132], [Pg.141]

[10] Vivado high-level synthesis. http://xilinx.com/support/download.html. [Pg.13], [Pg.33], [Pg.34], [Pg.63], [Pg.69], [Pg.71], [Pg.77], [Pg.84], [Pg.90], [Pg.91], [Pg.101], [Pg.108], [Pg.111], [Pg.132]

[11] W. snyder. (2017) verilator:. https://www.veripool.org/. [Pg.23], [Pg.24], [Pg.35], [Pg.64], [Pg.68]

[12] Zebu server 4 emulation system verification datasheet. `https://cdn.weka-fachmedien.de/whitepaper/files/ZeBu_White_Paper_V5.0.pdf`. [Pg.34]

[13] Synphony C compiler. `http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx`, 2010. [Pg.33]

[14] Secure Hash Standard—SHS: Federal Information Processing, Standard FIPS 180-4, U.S. Department of Commerce and National Institute of Standards and Technology (NIST). https://csrc.nist.gov/publications/detail/fips/180/4/final, 2012. [Pg.128]

[15] M. Abderehman, J. Patidar, J. Oza, Y. Nigam, T. A. Khader, and C. Karfa. Fastsim: A fast simulation framework for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(5):1371–1385, 2022. [Pg.84], [Pg.85], [Pg.111], [Pg.117], [Pg.118], [Pg.120], [Pg.132], [Pg.137], [Pg.139]

[16] M. Abderehman, R. T. Reddy, and C. Karfa. Deeq: Data-driven end-to-end equivalence checking of high-level synthesis. In *ISQED'22*, 2022. [Pg.117]

[17] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using ic fingerprinting. *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 296–310, 2007. [Pg.14], [Pg.31]

[18] A. Ahmed and P. Mishra. Quebs: Qualifying event based search in concolic testing for validation of rtl models. In *2017 IEEE International Conference on Computer Design (ICCD)*, 2017. [Pg.106]

[19] U. Alsaiari and F. Gebali. Hardware trojan detection using reconfigurable assertion checkers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–12, 2019. [Pg.116]

[20] S. Amellal and B. Kaminska. Scheduling of a control data flow graph. In *1993 IEEE International Symposium on Circuits and Systems*, pages 1666–1669 vol.3, 1993. [Pg.1]

[21] P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama. Verification of rtl generated from scheduled behavior in a high-level synthesis flow. In *1998 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (IEEE Cat. No.98CB36287)*, pages 517–524, 1998. [Pg.28]

[22] H. Badier, J.-C. L. Lann, P. Coussy, and G. Gogniat. Transient key-based obfuscation for hls in an untrusted cloud environment. *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1118–1123, 2019. [Pg.11], [Pg.145]

[23] B. Bailey. Is High-Level Synthesis Ready for Prime Time? `http://www.edn.com/design/integrated-circuit-design/4375454/Is-high-level-synthesis-ready-for-prime-time`, June 2012. [Pg.115]

[24] K. Banerjee, C. Karfa, D. Sarkar, and C. A. Mandal. Verification of code motion techniques using value propagation. *IEEE Trans. on CAD of ICS*, 33(8):1180–1193, 2014. [Pg.27], [Pg.33], [Pg.75], [Pg.76]

[25] C. Bao, D. Forte, and A. Srivastava. On application of one-class svm to reverse engineering-based hardware trojan detection. In *Fifteenth International Symposium on Quality Electronic Design*, pages 47–54, 2014. [Pg.31]

[26] K. Basu, S. M. Saeed, C. Pilato, M. Ashraf, M. T. Nabeel, K. Chakrabarty, and R. Karri. Cad-base: An attack vector into the electronics supply chain. *ACM Trans. Des. Autom. Electron. Syst.*, 24(4):38:1–38:30, Apr. 2019. [Pg.14], [Pg.115], [Pg.116]

[27] F. Besson, A. Dang, and T. Jensen. Securing compilation against memory probing. In *PLAS '18*, pages 29–40, 2018. [Pg.110]

[28] S. Bhasin, J.-L. Danger, S. Guilley, X. T. Ngo, and L. Sauvage. Hardware trojan horses in cryptographic ip cores. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 15–29, 2013. [Pg.30]

[29] S. Bhunia and M. Tehranipoor. *The Hardware Trojan War: Attacks, Myths, and Defenses*. Springer International Publishing, 2017. [Pg.116]

[30] G. Bloom, B. Narahari, and R. Simha. Os support for detecting trojan circuit attacks. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 100–103, 2009. [Pg.31]

[31] N. Bombieri, H. Liu, F. Fummi, and L. Carloni. A method to abstract rtl ip blocks into c++ code and enable high-level synthesis. In *50th ACM/EDAC/IEEE DAC'13*, pages 1–9, 2013. [Pg.22], [Pg.24]

[32] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and efficient construction of static single assignment form. In *Compiler Construction*, pages 102–122, 2013. [Pg.82]

[33] C. Karfa, C. Mandal and D. Sarkar. Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM Trans. Design Autom. Electr. Syst.*, 17(3):30, 2012. [Pg.27], [Pg.124]

[34] C.-T.Hwang, J.-H.Lee, and Y.-C.Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design*, 10(4):464–475, 1991. [Pg.6]

[35] N. Calagar, S. D. Brown, and J. H. Anderson. Source-level debugging for fpga high-level synthesis. In *24th FPL'14*, pages 1–8, 2014. [Pg.23]

[36] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM TECS*, 13(2):1–27, 2013. [Pg.33]

[37] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson. From software to accelerators with legup high-level synthesis. In *CASES'13*, pages 1–9, 2013. [Pg.23], [Pg.145]

[38] L. Chen, M. Ebrahimi, and M. Tahoori. Reliability-aware resource allocation and binding in high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 21:1–27, 01 2016. [Pg.6]

[39] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. Wang. Challenges and trends in modern soc design verification. *IEEE Design Test*, 34(5):7–22, Oct 2017. [Pg.116]

[40] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson. Dass: Combining dynamic amp; static scheduling in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(3):628–641, 2022. [Pg.6]

[41] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson. Dass: Combining dynamic amp; static scheduling in high-level synthesis. *IEEE Transactions*

on Computer-Aided Design of Integrated Circuits and Systems, 41(3):628–641, 2022. [Pg.10]

[42] J. Cheng, J. Wickerson, and G. A. Constantinides. Probabilistic scheduling in high-level synthesis. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 195–203, 2021. [Pg.6]

[43] M. Chinnadurai. Survey on scheduling and allocation in high level synthesis. *International Journal of Computer Science and Engineering Survey*, 3:43–51, 10 2012. [Pg.6]

[44] Y.-K. Choi, Y. Chi, J. Wang, and J. Cong. Flash: Fast, parallel, and accurate simulator for hls. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4828–4841, 2020. [Pg.23], [Pg.24], [Pg.35], [Pg.62], [Pg.65], [Pg.68], [Pg.93]

[45] Y.-k. Choi, P. Zhang, P. Li, and J. Cong. Hlscope+: Fast and accurate performance estimation for fpga hls. In *ICCAD'17*, page 691–698, 2017. [Pg.23]

[46] R. Chouksey and C. Karfa. Verification of scheduling of conditional behaviors in high-level synthesis. *IEEE TVLSI*, pages 1–14, 2020. [Pg.13], [Pg.27], [Pg.33], [Pg.70], [Pg.75]

[47] M. Chung, J. Kim, and S. Ryu. Simparallel: A high performance parallel systemc simulator using hierarchical multi-threading. In *2014 ISCAS'14*, pages 1472–1475. [Pg.23]

[48] J. Cong, P. Wei, C. H. Yu, and P. Zhou. Latte: Locality aware transformation for high-level synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 125–128, 2018. [Pg.6]

[49] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8–17, 2009. [Pg.115]

[50] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct 1991. [Pg.102], [Pg.104]

[51] M. Dahiya and S. Bansal. Black-box equivalence checking across compiler optimizations. In B.-Y. E. Chang, editor, *Programming Languages and Systems*, pages 127–147, Cham, 2017. Springer International Publishing. [Pg.144]

[52] S. Dai, G. Liu, R. Zhao, and Z. Zhang. Enabling adaptive loop pipelining in high-level synthesis. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 131–135, 2017. [Pg.55]

[53] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 129–132, 2018. [Pg.10]

[54] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. [Pg.82], [Pg.84], [Pg.126]

[55] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008. [Pg.108]

[56] C. Deng and K. S. Namjoshi. Securing a compiler transformation. In X. Rival, editor, *Static Analysis*, pages 170–188, 2016. [Pg.110]

[57] D.Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE transactions on Design and Test of Computers*, pages 44–54, 1994. [Pg.1]

[58] V. D'Silva et al. The correctness-security gap in compiler optimization. In *IEEE Security and Privacy Workshops*, pages 73–87, 2015. [Pg.110]

[59] S. Dutt and O. Shi. Power-delay product based resource library construction for effective power optimization in HLS. In *18th International Symposium on Quality Electronic Design, ISQED 2017, Santa Clara, CA, USA, March 14-15, 2017*, pages 229–236. IEEE, 2017. [Pg.6]

[60] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. Di Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo,

J. Mamish, S. Orgrenci-Memik, T. Aarrestad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, and Z. Wu. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices, 2021. [Pg.11]

[61] X. Feng and A. Hu. Early outpoint insertion for high-level software vs. rtl formal combinational equivalence verification. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 1063–1068, 2006. [Pg.29]

[62] N. Fern and K. T. Cheng. Evaluating assertion set completeness to expose hardware trojans and verification blindspots. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 402–407, March 2019. [Pg.116]

[63] P. Fezzardi, M. Castellana, and F. Ferrandi. Trace-based automated logical debugging for high-level synthesis generated circuits. In *33rd ICCD'15*, pages 251–258, 2015. [Pg.23]

[64] D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE transactions on Design and Test of Computers*, pages 44–54, 1994. [Pg.102]

[65] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, USA, 1992. [Pg.1], [Pg.6], [Pg.71], [Pg.115]

[66] D. Greaves. A verilog to c compiler. In *Proceedings 11th International Workshop on Rapid System Prototyping. RSP 2000.*, pages 122–127, 2000. [Pg.23]

[67] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *16th VLSID'03.*, pages 461–466, 2003. [Pg.26], [Pg.69]

[68] S. Gupta, A. Saxena, A. Mahajan, and S. Bansal. Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In *SAT 2018*, pages 365–382, 2018. [Pg.87]

[69] K. Hao, F. Xie, S. Ray, and J. Yang. Optimizing equivalence checking for behavioral synthesis. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 1500–1505, 2010. [Pg.29]

[70] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009. [Pg.64]

[71] S. Heck, S. Kaza, and D. Pinner. Creating value in the semiconductor industry. In *McKinsey, New York, NY, USA*, pages 1–15, August 2011. [Pg.115]

[72] Y. Herklotz, Z. Du, N. Ramanathan, and J. Wickerson. An empirical study of the reliability of high-level synthesis tools. In *FCCM '21*, 2021. [Pg.13], [Pg.71], [Pg.86], [Pg.145]

[73] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *2010 IEEE Symposium on Security and Privacy*, pages 159–172, 2010. [Pg.31], [Pg.116]

[74] A. K. Jain, H. Omidian, H. Fraisse, M. Benipal, L. Liu, and D. Gaitonde. A domain-specific architecture for accelerating sparse matrix vector multiplication on fpgas. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 127–132, 2020. [Pg.11]

[75] S. Jha and S. K. Jha. Randomization based probabilistic approach to detect trojan circuits. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 117–124, 2008. [Pg.30]

[76] L. Josipović, R. Ghosal, and P. Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 127–136, New York, NY, USA, 2018. Association for Computing Machinery. [Pg.6]

[77] C. Karfa, R. Chouksey, C. Pilato, S. Garg, and R. Karri. Is register transfer level locking secure? In *Proceedings of the 23rd Conference on*, DATE '20, page 550–555, 2020. [Pg.68]

[78] C. Karfa, T. A. Khader, Y. Nigam, R. Chouksey, and R. Karri. Host: Hls obfuscations against smt attack. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 32–37, 2021. [Pg.145]

[79] C. Karfa, D. Sarkar, and C. Mandal. Verification of data-path and controller generation phase of high-level synthesis. In *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, pages 315–320, 2007. [Pg.13], [Pg.70], [Pg.83]

[80] C. Karfa, D. Sarkar, and C. Mandal. Verification of datapath and controller generation phase in high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):479–492, 2010. [Pg.28], [Pg.44], [Pg.46]

[81] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar. An equivalence-checking method for scheduling verification in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):556–569, 2008. [Pg.27], [Pg.74], [Pg.75], [Pg.76]

[82] C. Karfa, D. Sarkar, C. Mandal, and C. Reade. Hand-in-hand verification of high-level synthesis. In *21st GLSVLSI'07*, pages 429–434, 2007. [Pg.13], [Pg.28], [Pg.70]

[83] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 43(10):39–46, Oct 2010. [Pg.115], [Pg.116]

[84] R. Kastner, J. Matai, and S. Neuendorffer. Parallel Programming for FPGAs. *ArXiv e-prints*, May 2018. [Pg.66]

[85] F. Khalid, S. R. Hasan, O. Hasan, and F. Awwad. Runtime hardware trojan monitors through modeling burst mode communication using formal verification. *Integr. VLSI J.*, 61(C):62–76, Mar. 2018. [Pg.116]

[86] Y. Kim, S. Kopuri, and N. Mansouri. Automated formal verification of scheduling process using finite state machines with datapath (fsmd). In *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720)*, pages 110–115, 2004. [Pg.26]

[87] S. Kundu, S. Lerner, and R. Gupta. Translation validation of high-level synthesis. *IEEE Transactions on CAD of ICS*, 29:566 – 579, 05 2010. [Pg.26], [Pg.33], [Pg.70]

[88] A. Leung, D. Bounov, and S. Lerner. C-to-verilog translation validation. In *MEMOCODE*, pages 42–47, 2015. [Pg.29], [Pg.86]

[89] T. Li, Y. Guo, W. Liu, and C. Ma. Efficient translation validation of high-level synthesis. In *International Symposium on Quality Electronic Design (ISQED)*, pages 516–523, 2013. [Pg.26]

[90] T. Li, J. Hu, Y. Guo, S. Li, and Q. Tan. Equivalence checking of scheduling in high-level synthesis. In *Sixteenth International Symposium on Quality Electronic Design*, pages 257–262, 2015. [Pg.26]

[91] Y. Liu, K. Huang, and Y. Makris. Hardware trojan detection through golden chip-free statistical side-channel fingerprinting. *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014. [Pg.31]

[92] A. Lopes and M. Pereira. A machine learning approach to accelerating dse of reconfigurable accelerator systems. In *2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6, 2020. [Pg.22]

[93] T. Loveless, J. Ott, and P. Brisk. *Time- and Resource-Constrained Scheduling for Digital Microfluidic Biochips*, page 198–208. Association for Computing Machinery, New York, NY, USA, 2021. [Pg.6]

[94] A. Mahapatra, Y. Liu, and B. Schäfer. Accelerating cycle-accurate system-level simulations through behavioral templates. *Integration*, 62, 03 2018. [Pg.23]

[95] A. Mahapatra and B. Schäfer. Veriintel2c: Abstracting rtl to c to maximize high-level synthesis design space exploration. *Integration*, 64, 08 2018. [Pg.22]

[96] N. Mansouri and R. Vemuri. Accounting for various register allocation schemes during post-synthesis verification of rtl designs. In *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, pages 223–230, 1999. [Pg.28]

[97] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, 2009. [Pg.10]

[98] G. D. Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Higher Education, indian edition edition, 2011. [Pg.1], [Pg.6], [Pg.7]

[99] K. Mittal, A. Joshi, and M. Mutyam. Timing variation-aware scheduling and resource binding in high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 16(4), oct 2011. [Pg.6]

[100] R. Mukherjee, M. Tautschnig, and D. Kroening. V2c — a verilog to c translator. In *22nd TACAS'16*, page 580–586, 2016. [Pg.24], [Pg.29], [Pg.86]

[101] M. Nanjundappa, H. Patel, B. Jose, and S. Shukla. Scgpsim: A fast systemc simulator on gpus. In *ASP-DAC'10*, page 149–154, 2010. [Pg.23]

[102] X. T. Ngo, J. Danger, S. Guilley, Z. Najm, and O. Emery. Hardware property checker for run-time hardware trojan detection. In *2015 European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4, Aug 2015. [Pg.116]

[103] A. Orailoglu and D. D. Gajski. Flow graph representation. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, page 503–509. IEEE Press, 1986. [Pg.4]

[104] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989. [Pg.6]

[105] O. Peñalba, J. M. Mendías, and R. Hermida. A global approach to improve conditional hardware reuse in high-level synthesis. *J. Syst. Archit.*, 47(12):959–975, June 2002. [Pg.78], [Pg.127]

[106] C. Pilato, K. Basu, F. Regazzoni, and R. Karri. Black-hat high-level synthesis: Myth or reality? *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(4):913–926, April 2019. [Pg.14], [Pg.18], [Pg.32], [Pg.115], [Pg.116], [Pg.117], [Pg.118], [Pg.119], [Pg.126], [Pg.132], [Pg.144]

[107] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *23rd FPL'13*, pages 1–4. IEEE, 2013. [Pg.33], [Pg.69], [Pg.79], [Pg.84]

[108] C. Pilato, F. Regazzoni, R. Karri, and S. Garg. Tao: Techniques for algorithm-level obfuscation during high-level synthesis. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, 2018. [Pg.11], [Pg.145]

[109] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, pages 151–166, 1998. [Pg.69]

[110] S. Poledna. Vivado design suite user guide, ug902(2012.2). `https://docs.xilinx.com/v/u/2012.2-English/ug902-vivado-high-level-synthesis`, july 2012. [Pg.13], [Pg.34]

[111] C. Rubattu, F. Palumbo, C. Sau, R. Salvador, J. Sérot, K. Desnos, L. Raffo, and M. Pelcat. Dataflow-functional high-level synthesis for coarse-grained reconfigurable accelerators. *IEEE Embedded Systems Letters*, 11(3):69–72, 2019. [Pg.11]

[112] H. Salmani, M. Tehranipoor, and J. Plusquellic. A novel technique for improving hardware trojan detection and reducing trojan activation time. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(1):112–125, 2012. [Pg.31]

[113] S. K. Sanadhya and P. Sarkar. Attacking reduced round sha-256. In S. M. Bellovin, R. Gennaro, A. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security*, pages 130–143, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. [Pg.128]

[114] T. Schmidt, G. Liu, and R. Dömer. Exploiting thread and data level parallelism for ultimate parallel systemc simulation. In *2017 54th ACM/EDAC/IEEE DAC'17*, pages 1–6. [Pg.23]

[115] A. Sengupta, S. Bhadauria, and S. P. Mohanty. Tl-hls: Methodology for low cost hardware trojan security aware scheduling with optimal loop unrolling factor during high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(4):655–668, 2017. [Pg.6]

[116] M. Shen, H. Chen, and N. Xiao. Entropy-directed scheduling for fpga high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2588–2601, 2020. [Pg.6]

[117] S. Sinha and T. Srikanthan. Dataflow graph partitioning for area-efficient high-level synthesis with systems perspective. *ACM Trans. Des. Autom. Electron. Syst.*, 20(1), nov 2014. [Pg.6]

[118] A. M. Sllame and V. Drábek. An efficient list-based scheduling algorithm for high-level synthesis. *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*, pages 316–323, 2002. [Pg.6]

[119] S. Soldavini, S. L. Alarcón, and M. Łukowiak. Using reduced graphs for efficient hls scheduling. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020. [Pg.6]

[120] R. Stewart, K. Duncan, G. Michaelson, P. Garcia, D. Bhowmik, and A. Wallace. Ripl: A parallel image processing language for fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 11(1), mar 2018. [Pg.11]

[121] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040, pages 451–460, Apr 2015. [Pg.63]

[122] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks. Fuzzing hardware like software. *CoRR*, abs/2102.02308, 2021. [Pg.145]

[123] R. Walker and S. Chaudhuri. Introduction to the scheduling problem. *IEEE Design Test of Computers*, 12(2):60–69, 1995. [Pg.6]

[124] Xilinx. *Xilinx Kintex 7 series FPGA configuration manual*, 2018. [Pg.64]

[125] S. ya Abe, M. Yanagisawa, and N. Togawa. Energy-efficient high-level synthesis for hdr architectures. *Information and Media Technologies*, 7(4):1319–1330, 2012. [Pg.6]

[126] Z. Yang, K. Hao, K. Cong, L. Lei, S. Ray, and F. Xie. Scalable certification framework for behavioral synthesis front-end. In *2014 51st ACM/EDAC/IEEE DAC'14*, pages 1–6, 2014. [Pg.26], [Pg.27]

[127] Z. Yang, K. Hao, K. Cong, L. Lei, S. Ray, and F. Xie. Validating scheduling transformation for behavioral synthesis. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1652–1657, 2016. [Pg.27]

[128] M. Yasin, C. Zhao, and J. J. Rajendran. Sfll-hls: Stripped-functionality logic locking meets high-level synthesis. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–4, 2019. [Pg.11]

[129] Z. Yang, K. Hao, S. Ray, and F. Xie. Handling design and implementation optimizations in equivalence checking for behavioral synthesis. In *2013 50th ACM/EDAC/IEEE DAC'13*, pages 1–6, 2013. [Pg.29]

[130] X. Zhang, O. Shi, J. Xu, and S. Dutt. A power-driven stochastic-deterministic hierarchical high-level synthesis framework for module selection, scheduling and binding. *J. Low Power Electron.*, 15(4):388–409, 2019. [Pg.6]

[131] X. Zhang and M. Tehranipoor. Case study: Detecting hardware trojans in third-party digital ip cores. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 67–70, 2011. [Pg.31]

[132] J. Zhao, Y. Zhao, H. Li, Y. Zhang, and L. Wu. Hls-based fpga implementation of convolutional deep belief network for signal modulation recognition. In *IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*, pages 6985–6988, 2020. [Pg.10]

[133] R. Zhao, M. Tan, S. Dai, and Z. Zhang. Area-efficient pipelining for fpga-targeted high-level synthesis. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015. [Pg.6]

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

Guwahati 781039, India