AUTOMATED DEBUGGING METHODOLOGY FOR FPGA-BASED SYSTEMS

A dissertation submitted to
TECHNISCHE UNIVERSITÄT DRESDEN
FACULTY OF COMPUTER SCIENCE

for the degree of
Doktor-Ingenieur (Dr.-Ing.)

presented by
HABIB UL HASAN KHAN, M.SC.
born on May 08, 1978 in Attock, Pakistan

accepted on the recommendation of
Supervisor and Examiner:
Prof. Dr. Diana Göhringer (Technische Universität Dresden)
Co-examiner:
Prof. Dr. Heinrich Theodor Vierhaus (BTU Cottbus-Senftenberg)

Dresden, November 28, 2019

Declaration

I hereby certify that I have authored this Doctorate Dissertation titled *Automated Debugging Methodology for FPGA Systems* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, November 28, 2019

_____

Habib ul Hasan Khan, M.Sc.

Abstract

Electronic devices make up a vital part of our lives. These are seen from mobiles, laptops, computers, home automation, etc. to name a few. The modern designs constitute billions of transistors. However, with this evolution, ensuring that the devices fulfill the designer's expectation under variable conditions has also become a great challenge. This requires a lot of design time and effort. Whenever an error is encountered, the process is re-started. Hence, it is desired to minimize the number of spins required to achieve an error-free product, as each spin results in loss of time and effort.

Software-based simulation systems present the main technique to ensure the verification of the design before fabrication. However, few design errors (bugs) are likely to escape the simulation process. Such bugs subsequently appear during the post-silicon phase. Finding such bugs is time-consuming due to inherent invisibility of the hardware. Instead of software simulation of the design in the pre-silicon phase, post-silicon techniques permit the designers to verify the functionality through the physical implementations of the design. The main benefit of the methodology is that the implemented design in the post-silicon phase runs many order-of-magnitude faster than its counterpart in pre-silicon. This allows the designers to validate their design more exhaustively.

This thesis presents five main contributions to enable a fast and automated debugging solution for reconfigurable hardware. During the research work, we used an obstacle avoidance system for robotic vehicles as a use case to illustrate how to apply the proposed debugging solution in practical environments.

The first contribution presents a debugging system capable of providing a lossless trace of debugging data which permits a cycle-accurate replay. This methodology ensures capturing permanent as well as intermittent errors in the implemented design. The contribution also describes a solution to enhance hardware observability. It is proposed to utilize processor-configurable concentration networks, employ debug data compression to transmit the data more efficiently, and partially reconfiguring the debugging system at run-time to save the time required for design re-compilation as well as preserve the timing closure.

The second contribution presents a solution for communication-centric designs. Furthermore, solutions for designs with multi-clock domains are also discussed.

The third contribution presents a priority-based signal selection methodology to identify the signals which can be more helpful during the debugging process. A connectivity generation tool is also presented which can map the identified signals to the debugging system.

The fourth contribution presents an automated error detection solution which can help in capturing the permanent as well as intermittent errors without continuous monitoring of debugging data. The proposed solution works for designs even in the absence of golden reference.

The fifth contribution proposes to use artificial intelligence for post-silicon debugging. We presented a novel idea of using a recurrent neural network for debugging when a golden reference is present for training the network. Furthermore, the idea was also extended to designs where golden reference is not present.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor Prof. Dr. Diana Göhringer. Only her continuous support of my research, timely words of motivation, and vast amount of patience have enabled me to accomplish this thesis and transform me into an engineer that I am today. I could not think of having a better advisor and mentor than her for my Ph.D study.

Besides my supervisor, I would like to thank Prof. Dr. Heinrich Theodor Vierhaus for his valuable comments for improvement of the thesis. Moreover, I am especially grateful to Prof. Dr. Rainer G. Spallek for his constructive suggestions during the course of my research. I would also thank the rest of my Ph.D committee members: not only for their valuable comments and words of encouragement, but also for the hard questions which inspired me to broaden my research horizon from different perspectives.

I would like to thank my fellow colleagues for the thought-provoking discussions, for the tireless efforts we put in to achieve deadlines, and also for the fun we made together in the last four years. At ADS, I wish to extend my gratitude to Gökhan, Ariel and Ahmed for their collaborations. Also, I thank my friends in ESIT, Ruhr Universität Bochum who were always there to advise me whenever I faced technical issues. In particular, I am extremely grateful to Prof. Dr. Michael Hübner for his advices and encouragement.

Last but not the least, I would like to thank my family: my parents and to my brother and sister for their moral support and continuous prayers throughout inscription of this thesis and my life in general.

I dedicate this thesis to my parents and family

Preface

The research contributions presented in this thesis were published in papers [Khan1-8]. Specifically, Chapter 3 comprises the contents of papers [Khan1, Khan3, Khan4]. In this chapter, we discussed a cycle-accurate debugging solution for FPGA-based systems along with different techniques for visibility enhancement such as concentration networks and incremental insertions.

Chapter 4 contains content from papers [Khan2, Khan4, Khan8]. In these papers, we discussed debugging of multiprocessing systems and multi-clock domain systems. In Chapter 5, research work regarding priority-based signal selection has been submitted and will be published subsequently. Chapter 6 was first published in paper [Khan5]. We, later on, extended the work to include FPGA-in-the-loop and debugging for cases when the golden reference is unavailable. The extended work was published in paper [Khan6]. The idea for chapter 7 was published in [Khan7]. The idea was later extended to elaborate the artificial intelligence techniques. The enhanced paper is also submitted for publication.

In all cases, I explored the ideas, conducted the research work, performed the experiments and drew the conclusions under the guidance of Prof. Dr. Diana Göhringer, who also provided editorial support for my manuscripts as well.

[Khan1] H. Khan and D. Göhringer, "FPGA debugging by a device start and stop approach," in International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2016.

[Khan2] H. Khan, J. Rettkowski, M. Eldafrawy, and D. Göhringer, "An event-based Network-on-Chip debugging system for FPGA-based MPSoCs," in International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017.

[Khan3] H. Khan, T. Grimm, M. Hübner, and D. Göhringer, "Access Network Generation for Efficient Debugging of FPGAs," in Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, 2017.

[Khan4] H. Khan , A. Kamal and D. Göhringer, "An Intrusive Dynamic Reconfigurable Cycle-Accurate Debugging System for Embedded Processors," in International Symposium on Applied Reconfigurable Computing, 2018.

[Khan5] H. Khan and D. Göhringer, "FPGA Debugging with MATLAB Using a Rule-Based Inference System," in International Symposium on Applied Reconfigurable Computing, 2017.

[Khan6] H. Khan, A. Podlubne and D. Göhringer, "Intrusive FPGA-in-the-loop debugging using a rule-based inference system," Microprocessors and Microsystems, vol. 64, pp. 185–194, 2019.

[Khan7] H. Khan and D. Göhringer, "Cycle-Accurate and Cycle-Reproducible Debugging of Embedded Designs Using Artificial Intelligence," in 28th International Conference on Field Programmable Logic and Applications (FPL), 2018.

[Khan8] H. Khan, G. Akgün, A. Podlubne, F. Wegner, A. Moradi and D. Göhringer, "Cycle-accurate Debugging of Multi-clock Reconfigurable Systems," in International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2019.

Table of Contents

List of Figures

List of Tables

Glossary

ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CAD	Computer-Aided Design
DPR	Dynamic Partial Reconfiguration
DRAM	Distributed RAM
DSP	Digital Signal Processing
DUT	Device Under Test
FPGA	Field-Programmable Gate Array
GR	Golden Reference
HDL	Hardware Description Language e.g. Verilog, VHDL
IC	Integrated Circuit
I /O	Input/Output
ILA	Integrated Logic Analyzer
IP	Intellectual Property
LE	Logic Element
LUT	Look-Up Table
NoC	Network-on-Chip
MPSoC	Multiprocessor System–on-Chip
PAR	Place And Route
RAM	Random-Access Memory
RNN	Recurrent Neural Network
RTL	Register-Transfer Level
SoC	System-on-Chip
SRL	Shift Register Logic
GRLS	Globally Ratiochronous Locally Synchronous
GALS	Globally Asynchronous Locally Synchronous
HDD	Hard Disk Drive

SGD	Stochastic Gradient Descent
FIL	FPGA-in-the-loop
ICAP	Internal Configuration Access Port
PCAP	Processor Configuration Access Port
JTAG	Joint Test Action Group
PE	Processing Element
PS	Processing System
SD	Secure Digital

Chapter 1 Introduction

1.1 Motivation

Electronic devices make up a vital part of our lives. These are seen from mobiles, laptops, computers, home automation, etc. to name a few. The Integrated Circuits (IC) constitute the backbone of these devices [1]. As envisioned by Moore's Law [2], the modern integrated circuits have developed into extremely complex devices capable of performing sophisticated functions such as information processing, mathematical computation, signal processing, control engineering, etc. Modern ICs now comprise of billions of transistors [3]. With this evolution, ensuring that the device fulfills the designer's expectation under varying conditions has also become a great challenge.

Modern ICs are designed to perform a specific task with utmost efficiency and are then directly implemented on the silicon which are termed as Application Specific Integrated Circuits (ASIC). This practice requires a lot of design time and effort. Whenever an error is encountered, the process is restarted; termed as design re-spin. It is desired to minimize the number of spins required to achieve an error-free product, as each spin results in loss of time, cost and effort. Similarly, some use cases require frequent design up-gradation which is accomplished through Field Programmable Gate Arrays (FPGA). As the FPGAs are used in the field environment, error-free operation is necessary.

In order to ensure an error-free design, conventionally, the designers perform simulation (e.g. Mentor Graphics ModelSim [4]) for verification of the design before fabrication phase also termed as pre-silicon verification. Simulation is popular in the design community due to its ease-of-use, ability to view the behavior of internal signals which can help to debug the design, apply a fix, and re-simulate to verify the intended behavior. Though this used to work in the past when devices were smaller, modern integrated circuits designs have become too big which are proving very difficult to simulate. This concern was verified during the development of Intel Core i7 microarchitecture [5] because software simulation of the IC was found to be a billion times slower than actual silicon. Similarly, Mentor Graphics conducted a study which highlighted that though the design complexity doubled every 18 months, design productivity doubled every 39 months. This was due to the fact that half of the design effort was spent on a complex system verification flow as shown in Figure 1.1 [3].

Hence, the focus shifted towards virtual prototyping for speeding up the design verification process. However, as the designs become complex, virtual prototyping also suffers because of speed issues. As stated in the IBM report [6], the implementation of a prototype with a target frequency of 1.6 GHz was able

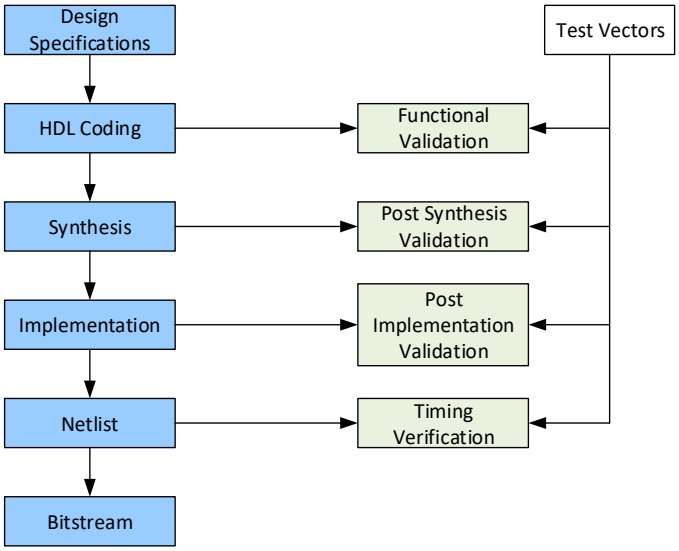


Figure 1.1: A conventional system verification flow

to achieve 10 Hz only when simulated on the HDL level. Thus, it can be concluded that using the simulation for verification becomes impractical for complex designs for complete test spectrum.

1.2 Debugging in the Post-Silicon Era

Because of such limitations of software-based simulation systems, hardware simulation also termed as post-silicon debug emerged as a better alternative. Although the designers always try to ensure an error-free hardware design through simulation before fabrication, few design errors (bugs) are likely to escape the simulation process. Such bugs subsequently appear post-silicon. Finding such bugs is time-consuming due to inherent invisibility of the hardware and hence significantly affect the time-to-market. As stated in the International Technology Roadmap for Semiconductors (ITRS) report, post-silicon validation remains a key challenge in the design process [7].

Instead of behavioral simulation of the design in the pre-silicon phase, post-silicon techniques permit the designers to verify the functionality using the physical implementations of the design. The main benefit of the methodology is that the implemented design in the post-silicon phase runs many orders-of-magnitude faster than its counterpart in pre-silicon thus allowing the designers to validate their design more exhaustively. This concept can also be used as an alternative circuit verification technique by configuring the prefabricated logic resources available on the FPGAs with any new design as many times as needed. This feature makes FPGAs best suited for prototyping [8]. Consequently, designers are increasingly turning towards FPGAs to prototype their design [9].

When unexpected behavior is observed in the post-silicon era, it becomes imperative to find its root-cause through debugging. The debugging process can be divided into two distinct phases. Firstly, it is required to collect relevant signal trace data from the circuit and secondly, to utilize the collected data from the first

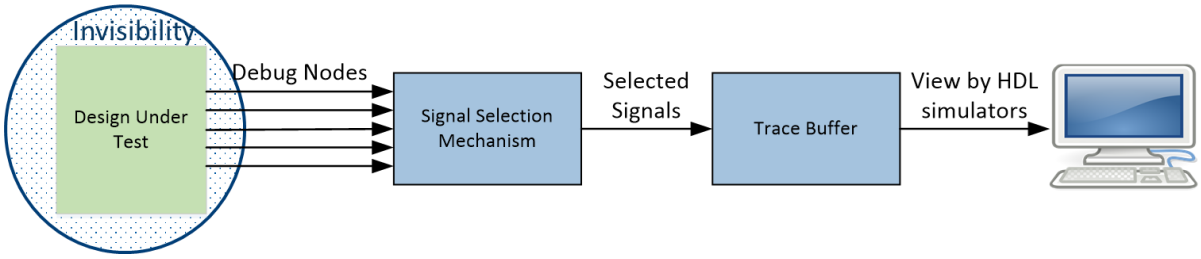


Figure 1.2: Trace buffers for visibility enhancement

step for error detection [10]. This thesis will focus both of them by first developing methods to collect the faulty trace data and subsequently perform error detection.

However, the post-silicon debugging methodology suffers from a main limitation. The debugging task becomes difficult on the hardware such as FPGAs or Application-Specific Integrated Circuits (ASICs), due to the invisibility of the hardware. During the pre-silicon simulation, all internal signals present in the circuit are observable. However, post-silicon, typically the signals present at the I/O interface device can be investigated. The problem is exacerbated due to the fact that the transistor density has surpassed the I/O pin density resulting in a scarcity of the I/O pins. Furthermore, out of those available I/O pins, only a limited number of pins are available for debugging, making the debugging task even more difficult.

Hardware visibility can be enhanced in multiple ways. Typically, trace-buffers based logic analyzers can be placed in the design before implementation. These instruments placed into the design can solve the visibility problems. The intended signals required to be debugged are connected to the probes of the Integrated Logic Analyzer (ILA) which then record the transitions of the connected signals unintrusively into on-chip memory during the at-speed operation of the design. After the signal behavior has been captured, the logic analyzer sends them to the connected PC and a suitable waveform viewer can be used for visual analysis as is done during the pre-silicon simulation. The trace-based debug environment for increasing hardware visibility is shown in Figure 1.2.

As stated, the ILA needs on-chip memory resources. But the available resources are limited. This results in a key bottleneck of the trace-based instruments i.e. only a subset of circuit signals can be observed for only a subset of the circuit operation time. Since such subset of circuit signals is quite small, the observable subset must be determined before the hardware implementation. A change in this subset of signals typically requires a reimplementation (also termed as a re-spin) of the design which may consume multiple hours, or even days [11]. Hence, a major challenge to a debug engineer is: to select a subset of trace signals when the nature of any potential bugs is still unknown. A wrong selection of the subset may limit debug productivity severely.

The other issue encountered during the debugging process is the selection of a subset of the circuit operation time. Due to the limited trace resources for the debugging process, normally a complete trace of the design

circuit cannot be captured. Hence, the trace capture is limited to a few hundred clock cycles depending on the trace window of the trace buffers which in turn depend upon the depth of the trace buffer. Debugging using limited data as described earlier is not easy. Such limited trace will be referred in this thesis as lossy trace. In the typical ILA-based debugging systems, a trigger signal is used to start the capture process which lasts depending upon the trace window of the trace buffers. Such systems are able to capture permanent errors such as functional errors, logic errors, stuck at faults because these debugging systems are able to capture them even using a lossy trace. However, certain errors may not last for long i.e. the errors may be intermittent. Examples of intermittent errors are configuration memory errors or the state changes in flip-flop or memory cells [12]. These errors are a source of major concern for safety-critical or life-critical applications which cannot be captured from lossy debugging systems like ILA. These errors can be explained better with an example. A system consists of a number of components. Suppose the system is not functioning correctly due to an open or a shorted resistor. In both cases, the system will produce permanent error. Such error can be captured even by lossy debugging since its value will not change during the course of monitoring. However, sometimes, due to certain reasons such as change in temperature, humidity or electrostatic discharge, the resistor changes its resistance for few clock cycles: resulting in an intermittent error. Capturing such errors using lossy debugging may prove to be a challenge.

Typical FPGA Debug Flow

At this stage, it seems beneficial to discuss a typical debug flow as shown in Figure 1.3. At stage 1, the circuit is designed using a Hardware Description Language (HDL) or High-Level Synthesis (HLS). Subsequently, the functionality of the design needs to be verified using behavioral simulations. As pointed out earlier, the complete verification of the design for complex systems may not be possible due to the speed limitations of behavioral simulators. Hence, only constrained-random testing [13] may be preferred since higher testing coverage may not feasible. Design verification is performed by acquiring the complete set of signals which are then displayed using waveform viewers. At this stage, complete visibility of the design is possible since all signals from the HDL designs are available for the complete duration of the test run.

Upon satisfaction with the results of the functional verification at the pre-silicon level, the design is moved to the post-silicon verification stage. This permits more comprehensive verification coverage; like booting an operating system on the hardware is possible owing to the higher speed of the design. However, at this stage, the hardware is inherently invisible. Hence, trace-buffers are inserted into the design to increase the visibility of the implemented design. The major difference from the pre-silicon verification is that only limited number of pre-selected signals can be recorded over short timeframes (owing to limited trace buffer resources) which will be referred in this thesis as lossy debugging data. Changing the signal set or the timeframe requires a complete re-spin (re-compilation) of the design which is a time-consuming process.

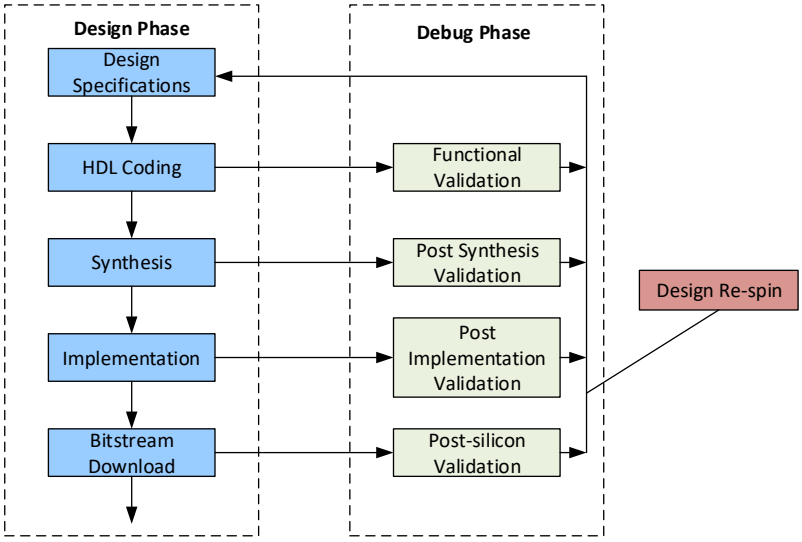


Figure 1.3: Typical FPGA debug flow

1.3 Research Challenge and Contributions

The long-term objective of this thesis is to present a comprehensive debugging solution which can be used to debug permanent such as logic errors which produce an unintended behavior and intermittent or non-deterministic errors such as the errors which result in a bit-flip for single as well as multiple clock based designs and for cycle-based as well as event-based systems. The five contributions of the current research work that enable to achieve the above-stated goal are as follows:

1. A post-silicon debugging approach is presented that allows cycle-based lossless debugging with effectively unlimited trace using limited Block RAMs (BRAM). Because of the lossless nature of the debug data, the cycle-accurate replay of the debugging data is possible. Since a very small trace-buffer is used, the proposed solution is also applicable to designs where only limited resources are available for debugging. This methodology ensures capturing permanent as well as intermittent errors. Concentration networks have been proposed to enhance hardware observability. Such networks allow changing the signal subset without re-compilation. Observability can further be enhanced by partial reconfiguring the debugging system at run-time hence saving the compilation time. Data compression can be utilized to transmit the data more efficiently. This contribution was published in papers [Khan1, Khan3, Khan4].

2. The solution for embedded processors is also presented. The debugging methodology is extended to multiple processors and multiple clock systems. This makes possible cycle-accurate replay for a broad spectrum of IPs. The solution for communication-centric designs is also presented. This contribution was published in paper [Khan2, Khan4, Khan8].

3. Although, few “important” signals can be identified manually by the designers. For complex designs, the debug engineer needs not to have complete knowledge of the whole design. In order to increase the time-efficiency of debugging, it is essential to automate the signal selection so that the most relevant signals are

selected for observation on priority. Moreover, automated connectivity generation tool is also presented. This contribution was published in papers [Khan3].

4. An automated error detection solution has been proposed which helps in capturing permanent as well as intermittent errors without continuous monitoring of debugging data which can be quite cumbersome for very large trace data files. This contribution is published in [Khan5]. Moreover, in many cases a Golden Reference (GR) of the design may not be available. Debugging such designs can be extremely difficult due to the absence of a known reference. An automated error detection methodology is introduced which can capture permanent as well as intermittent errors even in the absence of a GR. This contribution was published in [Khan6].

5. Using artificial intelligence for post-silicon debugging is a new idea. We proposed to use the Recurrent Neural Network (RNN) for debugging when a GR is present for training the RNN. Furthermore, the idea was also extended to designs where GR is not present. This contribution is published in [Khan7].

The following subsections offer a brief overview of each contribution; a detailed explanation is presented in the subsequent chapters of this thesis.

1.3.1 Cycle-accurate Lossless Debugging with Replay

A new cycle-accurate debugging methodology with effectively unlimited trace window is presented in sheer contrast of the ILA based debugging techniques which offer a limited trace window. A processor (ARM in case of Xilinx Zynq device [14] or Microblaze [15] for rest of the Xilinx FPGA families, also possible for other FPGA families with minor modifications) is utilized to collect the data from onboard trace buffers (4 KB BRAM). Once the trace buffers are full, the DUT (Device Under Test) is stopped and then the data is transferred to the terminal through Ethernet without user intervention where it can be saved to the memory devices like Hard Disk Drive (HDD), Secure Digital (SD) Card, etc. At the terminal, the data is logged to a log file and it can be used for debugging using open-source waveform viewers or HDL simulators.

Chapter 3 describes the implementation of our proposed debugging system. The verification of the response of an FPGA-based embedded design can be performed by simulation. When a problem on the physical hardware is encountered and debugging is required, traditional debugging systems are normally used. However, because of the limitations of debugging systems in capturing a large trace of data, debugging becomes difficult. If the design is not complex, traditional debugging systems like ILA may be sufficient. For more complex designs, debugging can be cumbersome with small data sets. In those situations, the proposed approach can be used. Since very small trace-buffers are used, the proposed solution is also applicable to designs where only limited resources can be spared for debugging. This methodology ensures capturing permanent as well as intermittent errors. The approach furthermore synchronizes the captured data automatically with a terminal, without any interaction by the user. The data is saved in text files and

then converted to Value Changed Dump (VCD) format which can then be displayed and analyzed by any open-source waveform viewer. The solution is therefore cost-effective as well.

The most important issue related to debugging is the limited observability of the hardware. The visibility can be enhanced through the inclusion of access networks in modern FPGAs. Access networks can be used in a variety of applications including debugging. When used in debugging systems, the number of inputs to the debugging systems can be greatly increased. This eliminates the need to re-synthesize the design due to any change in the signal set which results in saving debugging time and reducing the time to market. Using supervisory control by a processor, required networks can be configured just by minor software modification. Two different designs for access networks are introduced with respect to the available resources. The resource utilization for the two techniques has also been calculated.

Another important point to keep in mind is that the debug logic should not affect the placement of the design during instantiation or after removing the debugging circuitry from the DUT: as it may change the time response of the design and may require further analysis. This issue is addressed through Dynamic Partial Reconfiguration [16] of the debugging system. Moreover, other applications can also use the resources reserved for the debugging system when debugging is not required.

Another issue is that the trace generation is faster than trace transmission which results in a bottleneck at the transmission channel. Software-based trace compression is proposed to address the trace transmission bottleneck with no extra hardware overhead.

1.3.2 Debugging Coverage

Complex errors which cannot be traced with lossy debugging systems like ILA can be recorded with the proposed debugging solution. The extension of the proposed solution for multi-core and multi-clock systems is also be discussed. The solution for communication-centric designs is also be presented.

1.3.3 Observability Enhancement Tools

Furthermore, connecting thousands of nodes to the access network is also a complicated issue. A tool using IP-XACT files for automatic network generation is proposed in this chapter. The tool generates a Tcl file which creates the network automatically. A priority-based signal selection mechanism has also been presented which can identify the signals with most error probability. The signal selection along with the automatic connectivity generation framework can be extremely helpful during the debugging phase.

1.3.4 Automated Error Detection

Scan and trace buffer based techniques were devised to increase hardware visibility. These techniques require onboard logic resources in order to save the debugging data and consequently only provide lossy debugging trace. But the errors can be intermittent [17], also termed as soft i.e. they may not be permanent.

Errors of such type are difficult to observe using traditional trace buffers or scan chain approaches due to its random nature. Such errors can be tackled with cycle-accurate lossless debugging techniques [Khan1]. However, the problem remains to find out the technique which can help in capturing the errors without manual analysis of debugging data on each clock transition. Another problem is to find out the methodology which can help in finding the intermittent errors in the absence of a GR.

These problems were resolved by our rule-based inference system [Khan5] [Khan6] which finds the errors by providing an unlimited capture window by intruding into the DUT through clock management. It eliminates the need for human intervention required to monitor the debugging data on every clock transition. Rule-based inference system performs a correspondence analysis for identifying the relationship between the input data and the GR.

1.3.5 Automated Debugging using Artificial Intelligence

An automated error detection solution by using artificial intelligence is proposed which can help in capturing the permanent as well as intermittent errors without continuous monitoring of debugging data on every clock transition. Cycle-accurate debugging is performed through RNN. Our proposed solution depends heavily on machine learning techniques for trace diagnostics as well as for bug localization. Results have shown that localizing potential bugs in the system can be done with or without the presence of a GR. This helps in problem identification and hence increases time-efficiency.

This contribution is published in [Khan7].

1.4 Significance of the Presented Work

This thesis focus on debugging of complex designs including multi-clock and multiprocessor based systems. Fabricating a chip for design verification is quite expensive, hence, FPGA-based prototypes also become a good alternative to perform comprehensive verification. Since almost 50% of the design time is actually spent on verification [18][19][7], frequent designs spin is quite normal. Because of the reconfigurable nature, the FPGA-based prototyping can be both cost and time-efficient.

This thesis makes significant contribution in the field of automated error detection by using techniques such as inference and machine learning resulting in assisting the tedious task of debugging.

1.5 Thesis Organization

The thesis is organized as shown in Figure 1.4. Chapter 2 provides a detailed background on FPGAs, their device architecture, the current state-of-the-art in debugging and the different solutions available.

In Chapter 3 we presented our cycle-accurate lossless debugging solution capable of providing effectively an un-limited trace window. The methodology includes access network, incremental debug insertion and data compression.

Chapter 4 presents various techniques to enhance debugging coverage. Hence, improvements have been suggested so that the proposed methodology can undertake to debug multiprocessor and multi-clock systems.

Chapter 5 presents visibility enhancement tools. Such tools include automated connectivity generation and priority-based signal selection.

Chapter 6 demonstrates automated error detection. The proposed rule-based inference system is presented to automate the debugging process.

In Chapter 7, debugging based upon RNN is presented which can be used for debugging with or without the GR.

Finally, this thesis will be concluded in Chapter 8 along with future research directions.

Parts of this thesis have been published in papers [Khan1-8].

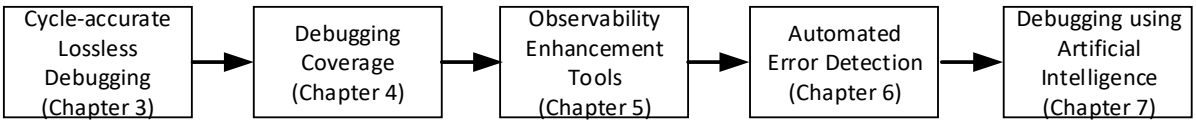


Figure 1.4: Thesis organization

Chapter 2 Background and Related Work

Since this research work will be referring to FPGAs and its device utilization, it seems appropriate to review the field-programmable gate array architecture. In section 2.2, the main challenge of post-silicon debugging along with different techniques used to enhance the hardware visibility will be discussed. In section 2.3, the state of the art with specific focus to this thesis: Cycle-accurate lossless debugging with replay, debugging coverage, observability enhancement tool, automated error detection and cycle-accurate debugging of embedded designs using artificial intelligence, will be covered.

2.1 Review of Field-Programmable Gate Arrays

FPGAs are semiconductor devices with prefabricated logic blocks that can be used to design a broad range of applications. The devices have some similarity with ASICs in the sense that both have prefabricated logic to perform specific functions. However, the main difference between the technologies is that ASICs are hard-wired to perform the specific tasks throughout its lifetime. FPGA, on the other hand, is a reconfigurable device which can be erased and reconfigured with any design. Although, due to this flexibility, the FPGA consumes more silicon (about 20–30x), operates slower (3–4x) and consumes more dynamic power (10x) as compared to an equivalent standard-cell ASIC (90nm) [20].

FPGA Architecture

FPGAs consists of flexible reconfigurable hardware due to its requirement of general-purpose applications. Commonly, FPGAs utilize a synchronous 2D island-style architecture. However, several other variants like asynchronous [21] or time-multiplexed [22] also exist. The island-style architecture comprises of the four components namely the I/O interfaces, soft-logic, hard-logic and a routing network. The routing network is flexible and hence can provide desired connectivity. As the name indicates, the I/O and logic resources are arranged in rows and columns in the form of islands across the device. As shown in Figure 2.1, the I/O pins are used to provide desired connectivity with the outside world. These bidirectional pins support various standards and voltages since the exact mode of communication is not known in advance. In an island-style architecture, I/Os are found on the FPGA peripheries although some exceptions are also present [30].

Soft-logic resources are the reconfigurable component of FPGAs. These components are configured to implement the desired logic. Combinational Logic Block (CLB) is the main constituent of the soft-logic resources. Generally, a CLB consists of slices which are connected to a switch matrix for routing to other FPGA resources. A slice consists of Look Up Table (LUT). However, the number of LUT or its inputs depend upon the specific device family [23]. The LUT is the core of the soft-logic resources comprising of memory cells connected to a multiplexer; any function can be realized by configuring the memory array of the LUT with appropriate contents (known as LUT mask). The select-lines of multiplexer allow the contents of the LUT to be forwarded to the LUT output.

The Configurable Logic Blocks (CLB) architecture depend upon the device family. For 7 series FPGA, a CLB comprises of two slices. The slices (CLBs) can be configured by the respective tools to implement any logic function. The slices are differentiated into SLICEL and SLICEM where SLICEM can use their LUTs as distributed memory [23]. A typical slice inside a CLB for 7-series FPGAs is illustrated in Figure 2.1. Being a reconfigurable component, a LUT in a 7 series FPGA can be configured either as a 6-input LUT with one output or two 5-input LUTs with separate outputs. The output of the LUTs can be registered in flip-flops. Four 6-input LUTs along with 8 flip-flops, multiplexers and arithmetic carry logic units join together to form a slice.

Hard-logic refers to the predefined blocks present in the FPGA which are tailored to accomplish certain tasks more efficiently i.e. Digital Signal Processors (DSP), memory blocks (BRAM), embedded CPUs, clock distribution networks, etc. Although the BRAM, DSP, and CPU can also be implemented through soft-logic, the dedicated hardware resources are more optimized to perform their specific tasks. The hardware logic interspersed throughout the device is also illustrated in Figure 2.1.

The routing network performs interconnection among on-chip hardware components like I/O, soft-logic and hard-logic resources for efficient communication. Multiplexers are used to make the system flexible. Based upon their utility, these are organized into either the Switch Blocks (SB) or the Connection Blocks (CB). As depicted from their name, SBs are used for selecting the output of the resource permitted to use the wire. On the other hand, CBs select the sink(s) which will receive the output of the prior block. The select lines of these multiplexers are connected to memory cells which can be programmed to implement the desired connectivity.

2.2 Challenges in the Post-Silicon Era: Error Detection

One point worth highlighting before stating the challenges addressed in this thesis is that its scope is limited to post-silicon debugging and is not related to fault injection and fault coverage. Fault injection is a technique which helps in understanding how the system will behave when subjected to unusual stresses [24]. Hence, faults are induced at the hardware level to simulate how hardware failure affect the system. Similarly, fault coverage is defined as the number of faults detected out of the potential faults [25]. In order to better predict the test escape rate, faults should be weighted by their probability of occurrence. Contrary to the above, this work focuses to find permanent or intermittent errors during prototyping and post-silicon hardware validation phase. Hence, it is necessary to check the use-case scenarios such as booting an operating system, etc. to validate its performance.

Moore in his famous paper predicted that the number of transistors per square inch would keep on doubling every 12 months [2]. Although the pace has slowed down a bit in recent years however, still the data density keeps on doubling every 18 months [26]. This exponential increase in the number of transistors made possible to realize complex digital circuits. However, this complexity of digital designs also resulted in

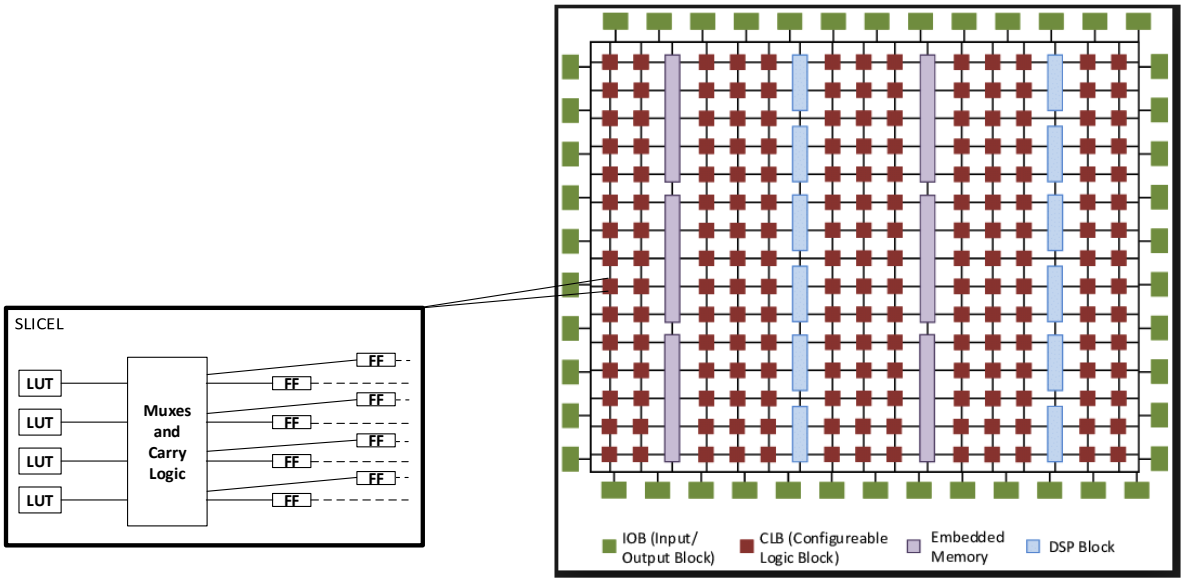


Figure 2.1: Simplified FPGA architecture

making the verification and debugging a daunting task. A recent study by Wilson Research Group and Mentor Graphics [27] found that about 50% of the total time spent is consumed in the verification phase. Furthermore, they also highlighted 42% of this verification time is actually spent in the debugging process. It is also pointed out that due to asynchronous interactions between clock domains, flaws may become visible only after system integration [28].

The preceding paragraph clearly highlights the importance of debugging in a product design phase which is the main topic of this thesis. This thesis will focus on permanent errors such as functional errors i.e. the logic errors which produce an unintended behavior as well as intermittent errors or soft errors which may produce a glitch due to fabrication defects. Instances of functional errors are a state machine caught in an endless loop or a wrong implementation of a mathematical function; Examples of intermittent errors are configuration memory errors or the state changes in flip-flop or memory cells. Such errors are a source of major concern for safety-critical or life-critical applications.

Functional errors are easy to debug through logic or behavioral simulation because of unlimited observability [29] and quick turnaround time between finding the bug, identification of the root-cause and bug-fix. Upon observation of an error, verification engineers can find out the design error manually or through the use of automated techniques [30]. However, logic simulation is slow, especially for large designs. As stated in the IBM report [4], the implementation of a prototype with a target frequency of 1.6 GHz was able to achieve 10 Hz only when simulated on the HDL level. This demonstrates that using the simulation for verification becomes impractical for complex designs for complete test spectrum.

Apart from its slow response, simulation-based verification may also be in-conclusive. Although designers try to ensure optimum performance of a hardware design through simulation before fabrication, few design errors (bugs) are expected to escape the simulation and will show up in the post-silicon process. Finding

such bugs is time-consuming and significantly affect the time-to-market. As stated in the International Technology Roadmap for Semiconductors (ITRS) report, post-silicon validation poses an important challenge [7].

Hence, the focus is being shifted towards hardware solutions. However, the main problem of the hardware remains in its invisibility [31]. In order to view the signal, it needs to be routed to an I/O pads[32] which can then be attached to external logic analyzers [33]. Although this solution works at FPGA speeds, it is limited to the number of I/O pads. Morris [34] pointed out that more than 40% of FPGA designs are I/O limited. Furthermore, the ratio of logic to I/O in FPGAs is also increasing [35] which leaves even less number of I/Os for debugging. All these limitations complicate the debugging problem. However, it is vital for designers to have full visibility inside the chip because, in the case of a malfunction, the root-cause can only be found after identification of the bug.

In order to ease the debugging process, different techniques have been discussed in the literature as illustrated in Figure 2.2. The techniques can be broadly categorized into two main types such as cycle-based synchronization/granularity and event-based. Cycle-based methods observe the states at every clock cycle. Contrarily, the event-based methods observe the states at specific events such as transactions or handshakes [36]. Normally event-based debugging is adopted in situations where cycle granularity is not really required. Cycle-based debugging can further be broadly categorized into scan-based or trace-based. In the following paragraphs, the two post-silicon debugging techniques namely scan-based and trace-based debugging will be reviewed in more detail.

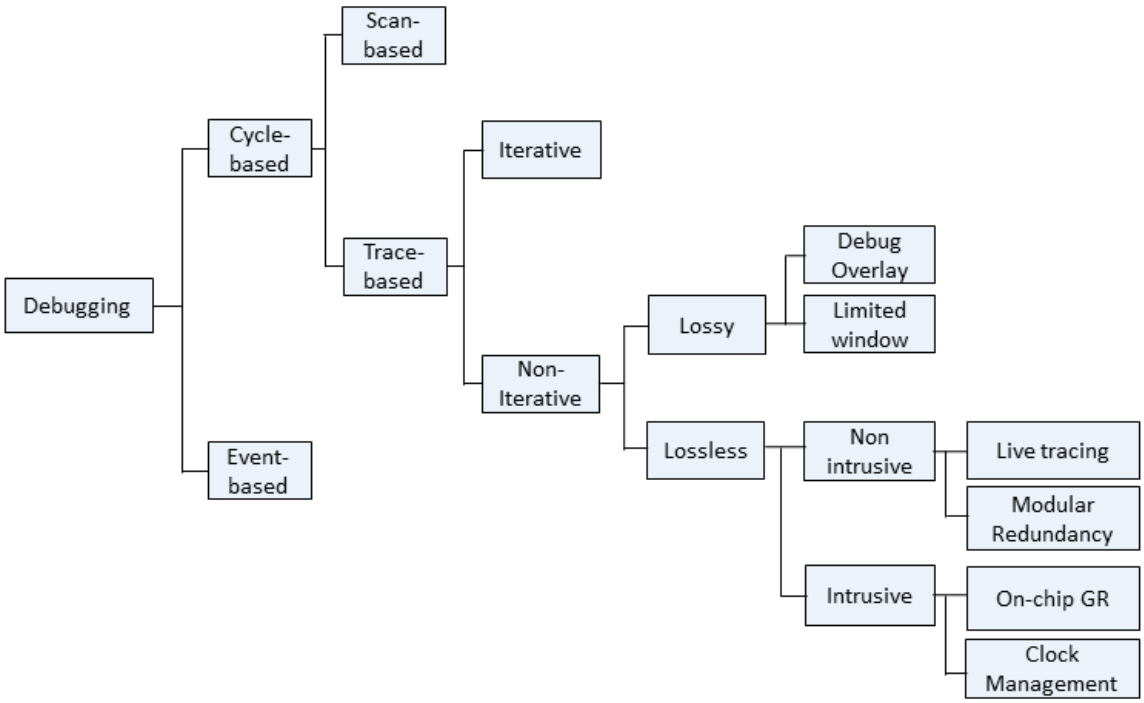


Figure 2.2: Debugging overview

2.2.1 Scan-based Error Detection

Scan chain, also known as boundary scan or Joint Test Action Group (JTAG), is the methodology to access interconnects of Integrated Circuits (ICs) for increased visibility. JTAG is not only used for testing the functionality of interconnects and ICs, but also for programming. The main component of scan-chain is the scan cell. A typical scan cell is shown in Figure 2.3. The scan cell approach involves capturing a snapshot of the DUT when required. This is achieved by either scanning all FPGA Flip-Flops (FF) and BRAMs using FPGA logic instead of transistor logic [37] or through the read-back methods explained in [38].

Typical examples of JTAG are BoardScope [39] and JBits [40] but they have been discontinued. FPGAXpose by Sandbyte can be quoted as a commercially available product in JTAG category [41]. However, the main issue with the JTAG approach is its operation at a slower clock rate than the rest of the design. Scan-based debugging provides complete visibility however for only one cycle at a time. Hence, for large designs, the debugging time increases proportionally.

The main technique for visibility enhancement in scan-based debugging is data compression. It is achieved by connecting only a subset of the flip-flops to the scan chains. This results in decreasing the time between scan-dumps thus allowing to single-step the design more quickly [42].

As claimed in [43] most of the recent ICs have built-in scan-chains already, it happens to be a zero-cost solution. Carbine et al. [44] describe using observation-only scan technique capable of observing internal nodes in Intel processors. Holdbrook et al. [45] used scan-chains for isolating timing errors between flip-flops by a "cycle-stretch" technique by comparing scan-dumps to their simulated counterparts. Rootselaar et al. [46] presented a multi-stage breakpoint technique for debugging multi-clock domain designs.

The main benefit of scan-based debugging beside zero overhead is that almost complete state of the design can be captured. However, this requires the circuit to be halted exactly at the point of interest which is quite

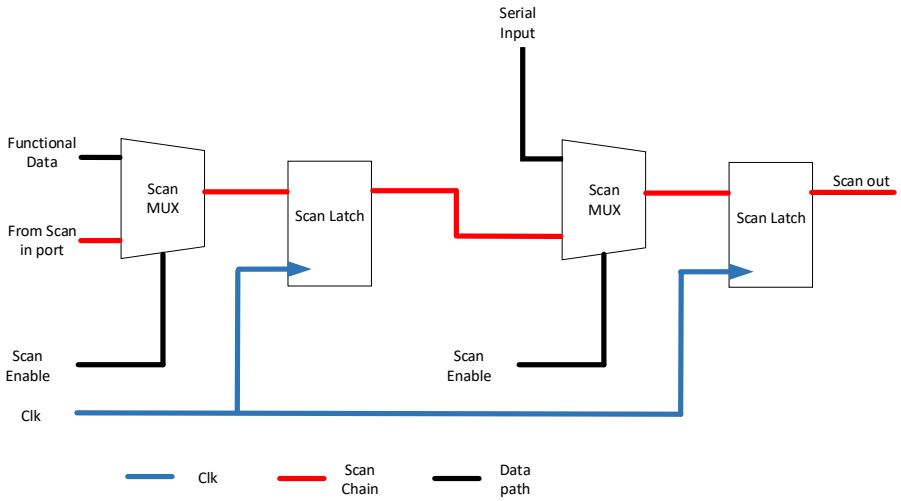


Figure 2.3: Scan-based debug instrumentation

tedious for circuits operations at MHz frequencies [47][48]. Moreover, the circuit can also be single stepped for a better understanding of the behavior of the circuit [49]. In BackSpace [50], the authors augmented formal analysis techniques with limited on-chip visibility to generate predecessor states that lead to the erroneous state. The proposed methodology captures the crash state by using a lossy signature. The retrieved information is then processed using formal technique of the respective RTL to compute a set of possible predecessors of the crash state. The computed states are then validated by using them as "breakpoints". This methodology then iteratively computes predecessors to the recent crash state hence going back in time. The methodology tolerated non-determinism to the extent that the error is not rare. Chuang et al. [42][51] proposed to copy the snapshots of the FPGA state periodically into shadow flip-flops connected together as scan-chains. This allows the values to be transferred to off-chip memory without affecting the circuit under debug. Observability can be enhanced through state reconstruction by combining successive traces.

Hsu et al. [19] presented a data expansion technique which was able to select essential signals which could then be used to find out missing signal values. The authors claimed to require only a subset of signals to attain full visibility. A similar technique claimed to monitor 4%–15% of all nodes to ensure full visibility. However, specific details of the techniques are unknown since they have been employed in a commercial product [52].

Several FPGAs have a readback [53][54] feature which permits to capture the current user state present in CLB registers, BRAMs, Distributed RAMs (DRAM) and Shift Register Logic (SRL) non-destructively through a supported interface e.g. BoardScope [39]. Readback-Capture can be carried out by anyone of these interfaces: Internal Configuration Access Port (ICAP), Processor Configuration Access Port (PCAP), SelectMap, or JTAG [55]. Tiwari et al. [56] suggested inserting a watch-point circuit which could improve the efficiency to monitor predefined signals for a runtime-configurable trigger condition. This enabled to stop the design at specific events which sped up the process [57]. However, 2~8 seconds are required to view just one flip-flop using readback-capture techniques [58]. Asaad et al. [6] proposed a cycle-accurate and cycle-reproducible system which could run at speed and stop at the point of interest. The state of the FPGA can then be read through readback. The system can then be single stepped to get a cycle-by-cycle waveform. Khan et al. [59] proposed a board-level debugging tool which was able to correlate the registers present in the netlist to their respective locations in configuration memory. The entire state of the FPGA could then be retrieved through readback. The results of the readback are compared with the simulation results and any mismatch is highlighted. Shanker et al. [60] proposed a spatial debugging methodology through readback. The author claimed to halt the DUT at any specific clock cycle without any instrumentation or re-programming. Iskander et al. [58] proposed a debugging and validation platform which could provide either a high-level abstraction or a low-level bit view for validation of hardware designs. The platform was able to place hardware breakpoints through partial reconfiguration and upon meeting the breakpoint, any register could be readback to an extra Microblaze processor which served as

primary monitor. Li et al. [61] proposed a readback based soft-core debugger which was able to monitor external memory. Tzimpragos et al. [62] proposed to use readback for multiple asynchronous clocks. Features like single stepping and waveform reconstruction could then be utilized for enhancing the debugging capability.

From the discussion it becomes clear that readback provides a complete spatial trace of the design at a specific instance. However, an important requirement for readback is to freeze the design state by stopping the clock which makes the process too slow.

2.2.2 Trace-based Error Detection

Another debugging methodology is based upon trace buffers which use block memories to log trace data by utilizing FPGA resources [4]. Data is saved to the BRAMs during runtime and extracted afterward to use it for debugging purposes. The trace buffer methodology is shown in Figure 2.4.

Due to resource limitation, often circular buffers are used to implement on-chip trace circuitry which resemble a sliding window. ILA cores offered by the FPGA manufacturers also use trigger signals to either start or stop this window resulting in a limited debug window [63]. In other words, the trigger signal remains a major bottleneck since the data can be monitored only after the core has been triggered and even after the trigger, a limited debug window is available. Consequently, ILA-based approach captures only a subset of the temporal trace depending upon the depth of the trace buffer. Debugging with a limited trace data becomes time-inefficient.

Commercial tools like Xilinx Chipscope [64] and Altera SignalTap II [65] use the trace buffer methodology. One drawback in the commercially available tools is that the signal set for the trace buffer block has to be identified during design time. Therefore, any changes require re-synthesis of the trace buffer block. The greater number of design re-spins increase debugging time [66]. In order to address this problem, a method of incremental synthesis has been proposed [67]. But recompiling would still be required which is again time-consuming. Synopsys's Identify [68] made it possible to modify the trigger condition at runtime but changing the signals under observation still required design recompilation. Certus [69] used a multiplexer

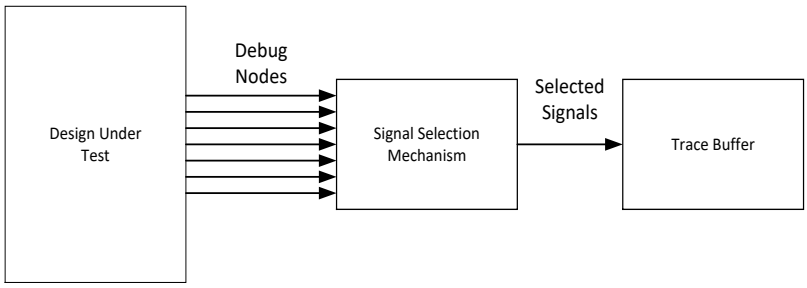


Figure 2.4: Trace buffers

network which made it possible to pre-instrument large number of interesting signals prior to compilation so that subset of signals can be selected during debugging.

Due to limited hardware resources, trace buffers only offer limited debug window. Hybrid logic analyzers have been introduced (Figure 2.5) to address the limited window size problem. This methodology combines the ILA core from FPGA manufacturers and traditional logic analyzers. One example is the Agilent and Xilinx co-developed logic analyzer including an external traditional logic analyzer and an ILA [70]. Internal signals are tapped to ILA and the data is saved to the trace buffers for debugging and output pins are monitored by the Agilent logic analyzer. However, this system is not cost-effective since extra hardware is required. Furthermore, the data is saved in trace buffers which require FPGA resources in case of ILA and also the memory of the Agilent logic analyzer for monitoring the pins. Hence in case of hybrid logic analyzers, the debugging methodology still remains memory dependent.

Emulation systems are also used for debugging. Exostiv has developed an FPGA debugging solution [71] in which a data collection and saving block technique has been implemented. This solution can be connected to a PC through its high-speed port and the data can be viewed on the PC. However, the system requires Exostiv hardware for debugging. Few other emulation platforms are available such as Cadence Palladium [72], Mentor Graphics Veloce [73] and Synopsys ZeBu [74] which provides good observability with simulation frequency up to 2MHz. However, such solutions are costly (about $1 million) and hence may not be cost-efficient in many design projects. One thing worth mentioning is that scan-based and trace-based solutions are not mutually exclusive. Cell processor designers [47][48] highlighted that starting and stopping the clock precisely can be quite challenging during scan-based debug. Hence, they suggested adding a trace and trigger functionality which allowed the full-speed data logging into a trace window of 132 bits wide by 1024 trace depth.

Trace-based debugging can be categorized into iterative or non-iterative tracing. Gao et al. [75] presented the concept of a suspect window to identify a range of clock cycles in which a bug may have originated before being observed. The authors proposed to first identify the beginning of this error bearing suspect window using non-intrusive tracing. Meanwhile performing the scan dumps, circuit is single-stepped to attain fine-grained observability. Another iterative tracing technique based upon lossy compression is

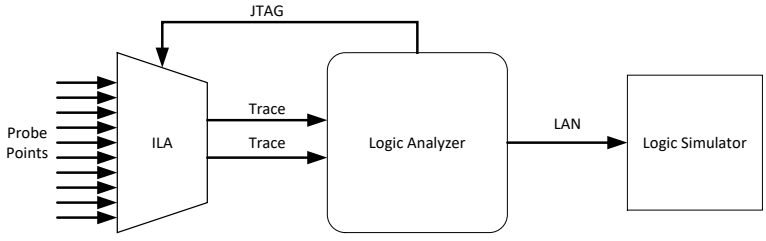


Figure 2.5: Hybrid logic analyzers

presented in [76]. It suggests to collects lossy traces of signal data surrounding an error (identified through comparison with behavioral model) iteratively hence progressively zooming in to the root cause of their bug. Yang proposed a similar iterative approach [77] where three sessions were required to capture the error. The authors used lossy compression to identify a suspect window. The second pass was used to identify the suspect clock cycles. In the third pass, the methodology was able to capture the error. However, the methodology is valid only for deterministic and reproducible errors. A similar technique is suggested by Ko et al. [78] which employs state restoration to enhance the visibility of flip-flops present in the design. The authors proposed to monitor the don't care inputs of logic gates (such as the inputs of an AND gate) and inferring the unknown values through backward and forward propagation. However, data restoration is not possible for designs having high logic depth between flip-flops.

In the above cases, data compression is used to increase the quantity of trace data which can be stored on the on-chip trace buffers, hence, enhancing the debug capability through more data collection. It is performed to compactly record the available data. Due to limited on-chip trace buffers capacity, data compression is employed to increase the amount of recorded information. Since the debugging data is not known apriori, an adaptive compression technique is needed which may require extra hardware resources making the trace compression unfeasible.

In TAB-BackSpace [79], the authors proposed to select an entry from the trace dump and set it as breakpoint hence effectively eliminating the need to perform formal analysis. In further extension of their work, the authors proposed nuTAB-BackSpace [80] which improves the earlier work by dealing non-determinism. Jung et al. [81] proposed a 2-D compaction method for expanding the depth of the trace window of a trace buffer. The authors performed three debugging sessions and were able to maintain the 2-D compaction capability irrespective of the error patterns which was not the case with [77]. However, the technique required GR for comparison which may not be available in certain cases.

The non-iterative trace-based debugging can further be classified into lossy and lossless trace. All commercial logic analyzers such as Xilinx ILA [82] or Altera SignalTap [65] fall in the lossy tracing category because of their limited trace window. Another example of this limited window based non-iterative debugging is the debug overlay as proposed by Hung [83]. Similar work was done by Eslami et al. [84] however; the main difference is the trigger circuitry insertion at debug time which was missing in earlier works. Hale et al. [85] proposed to use the leftover Shift Register Logic (SRL) for debugging in resource-constrained designs.

Error detection has proven to be difficult due to lossy debugging data. Researchers explored different lossless trace techniques to get more trace data. Lossless trace-based debugging can be classified into non-intrusive and intrusive debugging. Live tracing has been adopted by many researchers for enhancing the debugging capability. Blochwitz et al. [86] proposed a non-intrusive technique with lossless trace. The

authors proposed to utilize the free I/O pins of the FPGA for data transmission through PCIe. However, only few signals can be debugged depending upon the availability of free I/O pins. Backasch et al. [87] and Decker et al. [88] proposed modular redundancy for lossless trace. The authors proposed to have redundant copies of the design. An on-chip comparison is made at run time and only the difference is transferred for off-chip analysis. However, since multiple copies of the hardware are required on-chip, the technique is resource-intensive. Deutsch et al. [89] suggested debugging using on-chip DRAMs. They suggested to save the GR on-chip and perform comparison between the received data and the GR. Once the difference is found that difference can be transmitted for analysis. However, the technique depends on the provision of GR. In the absence of GR, these techniques will not be useful.

Panjkov et al. [90] proposed an intrusive debugging technique by saving data on the on-chip BRAMs. The authors suggested stopping the clock of the DUT when the on-chip trace buffers are full so as to avoid data loss. They adopted an emulation system which can be used to transfer the data from on-chip trace buffers to the emulation system memory. After data transfer, the emulation system can start the clock of the DUT again. However, the authors used quite big BRAMs (1MB) which leaves very little BRAMs resource for main design. Furthermore, the proposed solution required excessive user intervention for clock management.

2.3 Related Work Specific to Chapters 3–6

This section describes previous related work specifically to each of the contributions of this thesis.

2.3.1 Cycle-accurate Lossless Debugging with Replay

As stated in the previous section, only those signals can be monitored which are available at the I/O pins of the FPGA. FPGA vendors introduced ILA [82][65] cores to solve this problem which can be placed in the design and used to debug the hardware. These logic analyzers that are embedded in the design can solve the visibility problems. But since these logic analyzers consume FPGA resources, they offer only a limited window for debugging resulting in a lossy trace. Hence, a new solution is required that addresses the visibility and limited window size issues.

A post-silicon debugging methodology based upon Device Start and Stop (DSAS) approach is presented that allows cycle-based lossless debugging with effectively unlimited trace using limited BRAMs. In the rest of the thesis, we will be referring to the proposed debugging system as DSAS for short. In DSAS, we suggested to start and stop the clock based on the occupancy of trace buffers. The clock to the DUT is stopped instantly when the trace buffers are full. The clock is automatically started when the trace buffers are empty after the data transfer. Consequently, the DUT is clock gated a fixed number of clock cycles determined by the depth of the trace buffers which results in cycle-accurate lossless trace of debugging data. This lossless trace is recorded to ensure capturing permanent as well as intermittent errors. The recorded trace can then be replayed offline. Cycle-accurate replay makes it possible to replay the recorded

trace cycle-by-cycle. Hence, post-silicon, complete cycle-accurate details can be reconstructed which can be very helpful to capture permanent as well as intermittent errors during the debugging process.

Due to utilization of small trace buffer, the proposed solution is also applicable to designs where only limited resources can be spared for debugging.

2.3.1.1 Access Network Generation

ILA cores utilize scarce FPGA resources, resulting in the monitoring of only limited number of signals. This necessitates the identification of the signal set required to be monitored during verification. However, the portion of the circuit required to be debugged during verification phase may change over time and is likely un-predictable at design time. If more signals require monitoring during verification, then re-synthesis is generally required. However, for complex designs, re-synthesis consumes lot of time, adversely affecting the time to market.

A flexible network solution permits the user to select the signals of interest, then a suitable debugging system can be used to process these signals as per debugging requirements. For connecting a large number of signals without the requirement to re-synthesize, a type of flexible networks called access networks [91] were proposed. These networks allow re-configuration of signals without having the need to re-synthesize the design iteratively just for debugging. However, such networks are heavily dependent upon hardware resources. An efficient access network is needed, which is able to provide connectivity with a minimum impact on the hardware resources.

The idea of utilizing a network for SoC communication is not new [92]. These networks are generally used for communication between a CPU and connected peripherals by using packet-based networks. Packet-based networks provide a very good option for SoC communication. However, such networks are not suitable for synchronous and high bandwidth applications such as debugging. Rather, indirect, non-blocking networks similar to the ones used for conventional telephone networks are more suited for applications which require synchronous high bandwidth communication [93]. Furthermore, such networks do not require real-time configuration. It is configured once for each application and does not require to entertain incremental connection requests. Therefore, the network is only required to be re-arrangeably non-blocking [94].

Research on non-blocking multistage networks permits connections from any input to any output of the network. These networks are usually called permutation networks because they provide all possible permutations of the inputs to the output side of the network. The Clos network can be used for constructing a re-arrangeable non-blocking permutation network having a low area cost[94]. In order to further reduce the network cost, the Benes network can be utilized which recursively replaces the middle stage of a Clos network with another Clos network [91]. But, such permutation networks are probably more flexible than needed. In a few cases, connecting every network input to every network output is not required. Instead,

the ability to connect any input to any output is generally required. This flexibility can be utilized for lowering the cost of the access network.

A lot of research has also been performed on unordered networks [95][96]. A class of unordered network, called concentrators, can be used for connecting a large number of inputs to a fewer number of outputs. Hence, such networks offer a good mean as an interface. But, while some concentrator designs have lower depth as compared to permutation networks, they are not necessarily more cost-effective than permutation networks for every possible configuration. Therefore, choosing the appropriate network architecture becomes difficult since the size of the network (numbers of inputs and outputs), and its effect on cost and depth must be considered before finalizing an architecture [91].

Altera Signal Probe [97] permits designers to reserve I/O pins on the FPGA so that internal signals can be connected to the outside world. On these spare I/O pins, up to 256 predetermined signals can be multiplexed through soft-logic which can be changed through the JTAG interface. A programmable logic core-based debugging system [98] comprising an access network was introduced which can be controlled by the PLC to select the signals required to be debugged. Poulos et al. [99] proposed to employ bitstream modification for changing the selected set more quickly. However, due to I/O pins limitation [34], pin reservation for debugging may be difficult. Microsemi Action Probe circuitry makes it possible to observe any four nets of a circuit in real-time which can be chosen dynamically [100]. However, the less number of nets remain a limiting factor.

The primary purpose of the access network is to efficiently connect the nodes to the debugging system. Although, this can be performed manually but involves a lot of effort and yet error-prone. In [90], a script tool was utilized for extraction of design nodes from HDL simulators. The IP-XACT standard [101] completely describes the interfaces for the elements present in an architecture in which four schemata acquire their main descriptions namely the design, component, bus and abstraction definitions. The design description acquires all the components present inside the architecture including the interconnections between the components and the external pins. The component description acquires the configurations that are part of each component including the bus interfaces and their ports, channels, address maps, clock and reset signals. For communication bus, the bus and abstraction definitions are provided as complimentary descriptions. The bus definition acquires the high-level attributes of a bus, including the connection method and addressing. The abstraction definition, on the other hand, captures the low-level attributes including the name, width and the direction of the ports. Connectivity between an access network and a debugging system can be generated utilizing the interface definitions described in this standard. This approach helps to reduce the human intervention required to perform manual connections between the debugging system and the access network.

2.3.1.2 Incremental Debug Insertion through Dynamic Partial Reconfiguration

Trace based debugging solutions mainly operate on the design before Place And Route (PAR). Such tools instrument trace buffers and their connections to the original user circuit before mapping. This results in fewer resources to be available for the original design. Moreover, debug circuitry insertion can alter the placement of the design. It can prove to be hazardous for the design; such as the hardware design may not fit in the FPGA device due to resources used by the trace buffers. Furthermore, sometimes, the timing behavior of the design is affected due to design recompilation hence, resulting in a bug to change or disappear [84].

Another aspect is that the instrumentation is normally performed after observing a failure. Hence, an iteration of the design process is required. Certus by Mentor [69], permits pre-instrumentation of a large signal set before compilation, a subset of signals can then be chosen for observation during debugging. This methodology provides more runtime flexibility than other tools. However, preselection of a set of signals required to be observed is still needed, before the debugging data is available.

In contrast, incremental synthesis can be used to make modifications in a fully placed-and-routed circuit whilst preserving the original design as much as possible [102][103][104]. For FPGAs, the main objective is to displace the minimum number of placed design blocks and existing nets to realize the objective.

Incremental insertion provides many benefits i.e. reduction in re-compilation time during the design process, preservation of timing closure for an engineering change or for improving the debugging resilience [105]. The idea has already been used for design prototyping and debugging [106] in both Xilinx tools and Altera Quartus II. Once, any portion of the circuit is marked as preserved, it is re-implemented only when its hardware description is changed. However, when the connections required to be debugged are changed for Xilinx ChipScope, the design requires re-implementation.

Graham et al. [107] proposed to place unconnected trace-buffers in the design at compile-time. Upon requirement, incremental bitstream modifications [108] can then be performed for changing the trace connections before testing. However, this technique requires reservation of FPGA resources at synthesis time, hence, preventing their use by the original circuit. Likewise, Poulos et al. [99] suggested modifying the LUT masks which led to more area-efficient multiplexer circuitry.

Wheeler et al. [37] proposed a design-level scan to connect memory elements such as FFs and embedded RAMs in sequence by utilizing the FPGA resources. However, the technique suffers from a high area overhead as FPGA resources are utilized for implementation of the scan-chains in the design. In [109], the authors suggested to pre-insert trace buffers in the design. Subsequently, low-level bitstream modification was performed to connect the trace buffers to the desired signals by using incremental techniques. However, still pre-reservation of FPGA resources is required which makes them unavailable to the original design.

Furthermore, trace buffers are required to be removed after the completion of the debugging process, which may change the placement and routing of the design.

Hung et al. [110] proposed a virtual overlay network which multiplexes the debuggable signals to the trace buffers already instantiated into the free FPGA resources for avoiding unnecessary design re-spins. But, the technique needs spare hardware resources which may not be available. A framework termed Dynamic Modular Development (DMD) [38] utilized the Xilinx partial reconfiguration flow to accelerate the embedded design process. The design modules were partitioned into separate partially reconfigurable regions. Subsequently, the embedded modules which do not require further modifications were merged automatically into the surrounding static region. Resultantly, rapid turnaround times were achieved through partitioning of frequently modified modules into separate partial reconfigurable regions[111].

A bitstream modification methodology was presented in [107] which allows bitstream modification after the PAR process. The ILA is instantiated into the design before netlisting. The signals of interest can be connected to the ILA by modifying the partial bitstream. This results in reducing the time consumed on the PAR process. However, when the signal set is altered, re-routing is required which can affect the time to market. Moreover, logic analyzer is required to be removed from the design after design validation which may affect timing closure of the validated design. Lagadec et al. [112] presented Software-like debug features such as breakpoints or watchpoints for enhancing debugging capability in reconfigurable platforms. However, recompilation of design was required whenever breakpoints or watchpoints were changed.

A new reconfigurability based debugging methodology was proposed [99] which enables monitoring of a large number of internal signals for any arbitrary number of clock cycles through limited external pins. This eliminates the need to perform iterations of re-synthesis, placement and routing processes. The methodology is realized by instantiating a multiplexer (MUX) into the design, with the potential signals required to be traced, connected to the MUX inputs. By reconfiguring the bitstream for select signals, different signals can be selected. The main drawback of this methodology is that the register contents are required to be shifted in one clock cycle, hence, affecting the maximum frequency F_{max} of the design.

Abramovici et al. [113] proposed a design-for-debug infrastructure, called distributed reconfigurable fabric. It had components widely distributed in the FPGA and was able to debug a large number of signals. Through reconfigurable programmed logic, it was able to implement various debug paradigms, such as signal capture, assertions, and what-if analysis used to accelerate the debugging process. But, the design still requires synthesis and implementation after instantiating the debugging architecture along with the requirement of significant hardware resources.

In a few intrusive debugging pieces of research [114][115], the clock of the embedded design was controlled for retrieving the debugging data. However, such works required breakpoints for stopping the clock. Resultantly, the system state very close to the breakpoint could only be monitored. An intrusive debugging

approach [90] based upon controlling the clock, through monitoring the occupancy of trace buffers was presented. However, the approach requires scarce FPGA resources (1MB RAM), emulation hardware and along with external intervention for data handling. In our previous work [Khan1], DSAS was introduced which needed only 4KB RAM for logging the data. Consequently, even resource deficient FPGAs can be instantiated with the proposed debugging system supporting an automated data saving process. However, in these intrusive debugging methodologies, debugging system is instantiated before the PAR process which in many cases is too time-consuming. By utilizing the Dynamic Partial Reconfiguration (DPR) [16] for debugging, the time consuming frequent recompilation can be avoided. Since the reconfiguration of an embedded design is very fast as compared to recompilation (tens of milliseconds versus minutes to hours), the debugging process can be sped up by taking advantage of partial reconfiguration.

2.3.1.3 Data Compression

In order to increase the amount of trace data that can be stored on the on-chip trace buffers while keeping the trace buffer size constant, compression of debugging data was proposed [116]. In spite of trace data compression, data generation rate is faster than data transmission which results in a bottleneck at the transmission channel and consequently loss of debugging data. Hence, an efficient compression technique for lossless streaming data is required which can improve off-load time of the debugging data and help in faster data dispensation through the transmission channel.

The efficiency of the trace data compression is described in terms of Compression Ratio (CR). CR can be calculated by the following equation.

$$CR = \frac{S_i}{S_o} = n * \frac{N}{S_o} \qquad (Eq. 2.1)$$

Where S_i is the size of the uncompressed data stream (n is the total number of time frames and N is the number of bits per frame) and S_o is the size of the compressed data stream.

Similarly, compression percentage is defined by:

$$C = \left[1 - \frac{1}{CR}\right] * 100 \qquad (Eq. 2.2)$$

Data compression is classified into two main techniques: Lossy and lossless trace data compression [117]. The data compression methodology is decided by the user depending upon requirements. Lossy data compression techniques such as JPEG or MPEG use in-exact approximations and partial data discarding to represent the data and are suitable for applications where the minor loss of fidelity doesn't affect the quality of data. Lossy data compression methods result in high CR while still meeting the requirements. As suggested by [117], CR for lossy compression can be >100.

Lossless compression techniques allow the original data to be perfectly reconstructed from the compressed data. Lossless data compression is suitable for environments which are prone to errors because of data loss. However, because of this limitation, high compression ratios are not possible. Moreover, compression ratios are highly dependent on input data. Shannon [118] established the fundamental limit to lossless data compression. The limit which he termed as entropy rate H is dependent upon the statistical nature of the data. He claimed that it is possible to compress the data in a lossless manner with CR close to H. It is mathematically impossible to exceed H.

ILA-based compression methods can be classified as special-purpose methods or generic methods. Special-purpose methods usually employ trace reduction techniques by sampling data at specified intervals or by eliminating the redundant data. Generic methods can be classified into depth compression [119] and width compression [120]. Depth compression is based on capturing as large number of samples as possible during debugging by re-running the debugging process. However, the technique is valid only for deterministic input sources. In width compression, the debug window size remains the same and compression is attained by re-constructing more number of signals than are captured by the trace buffers.

Lossless trace compression techniques (not limited to ILA based data compression) can be divided into two main categories, statistical compression techniques and dictionary based compression techniques.

Statistical compression techniques such as Huffman coding or arithmetic coding techniques provide better CR [121] by using variable-length codes. Shorter codewords are used for symbols with high occurring frequency and less occurring frequency symbols with longer codewords. However, the implementation of statistical coding algorithms requires two passes of the input data. In the first pass tree-based codeword structure is generated, based upon the occurring frequency of the incoming data, and in the second pass the data is coded. Because of two passes, the coding will be slow and hence not suitable for real-time applications. An adaptive Huffman coding algorithm was introduced in [122] in which the codewords are generated based upon the change in the probability of incoming data and hence occurrence frequencies and the binary tree are updated in just one pass. However, a hardware implementation of adaptive Huffman coding based upon tree-based codewords was able to achieve a throughput of 1 bit/cycle which do not meet requirements in high throughput application. In order to solve the problem, an ordered codeword tables based Huffman coding was suggested but with extra area overhead [123].

Dictionary-based compression algorithms such as Lempel-Ziv compression algorithm (LZ77) and its various variants LZW, WDLZW [118], achieve compression by coding a symbol or a sequence of symbols with shorter codewords which are represented by the indices of the dictionary. However, the dictionary is required to be transmitted along with the data which results in an overhead added to the compressed data. An adaptive dictionary-based compression scheme was suggested to avoid this problem [124]. However, in order to search the dictionary efficiently, a search engine called content addressable memory (CAM) is

used [125]. However, such dedicated hardware requires large resources [116]. Bitmask based compression technique was introduced in [126] which considers the difference in a few bit positions (Hamming distance) and stores that information in the compressed data. However, the CR depends on the number of bit changes considered during compression. Salama discussed LZ77 implementation using systolic arrays but systolic arrays suffer due to portability between designs [127]. Basu discussed the use of a static dictionary for small amount of errors [128] however the technique is dependent upon the availability of golden trace data and limitation of the presence of only small number of errors. Zhang et al. [129] introduced a new design for compressor which focuses more on throughput but still, it requires hardware resources for implementation.

It is evident from the preceding discussion that statistical compression techniques provide high CR but throughput is quite low for real-time applications. Dictionary-based compression algorithm provides high throughput at the cost of huge area overhead. The above-mentioned solutions can be adopted for FPGA designs with spare hardware resources. However, for debugging solutions having a controlling processor for data handling, the same processor can be used for software data compression without the need to use FPGA hardware. This solution can eliminate the data transmission bottleneck without any extra hardware resource and hence can be used to achieve a high data rate. The main contributions of this chapter are:

1. It presents a software-based lossless debugging data compression technique for a processor-controlled debugging environment. As far as we know, it is the first work which presents the software data compression for any intrusive debugging technique for embedded designs.
2. The technique provides high compression efficiency with no architecture overhead.

2.3.2 Debugging Coverage

2.3.2.1 Embedded Processor Debugging

The most recent trends for FPGA based systems covered by the Wilson Group Research report shows that the number of embedded processors is constantly growing [27]. The report claims that 64% of the designs now comprise of embedded processors. Keeping in view the above, it is imperative to provide a solution for embedded processors. As the embedded processors contain software applications running on the hardware, their debugging requires different technique which will be discussed in this chapter.

2.3.2.2 Event-based Multiprocessor Debugging Methodology

The number of embedded processors in FPGA based designs is constantly growing as highlighted in Wilson Group Research report [27] (Figure 2.6). The report claims that 43% of the designs now contain 2 or more processors. Keeping in view the above, we felt it imperative to provide a solution for multiprocessor systems. Although, the solution presented in the previous section is applicable to multiprocessor systems.

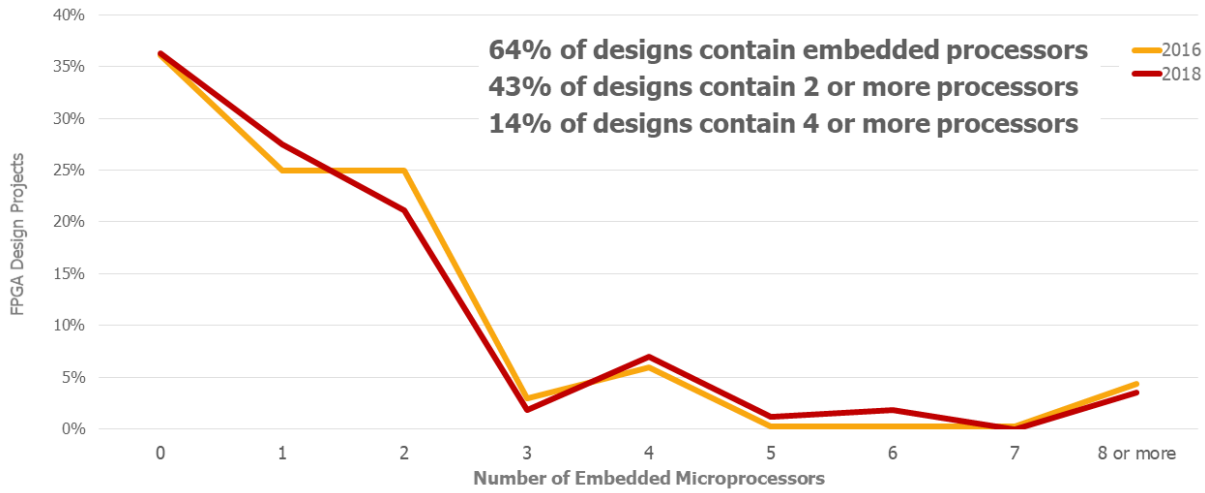


Figure 2.6: No. of embedded processors in design projects [27]

For a SoC level, it has been normally referred that the bit/cycle level is not effective for debugging; the transaction level is more desirable [130][131][132]. In this section, we will present a communication-centric (transaction-based) debugging solution centered around an event-based debugging methodology.

Communication architectures play an important role in a SoC's design and performance [133]. Before Network-on-Chip(NoC), MPSoCs and SoCs used to have a common bus for interconnection between different cores in a system. With the increase in the number of cores present in a SoC, the efficiency of the common bus system decreased and it became a bottleneck for the system, limiting its speed [134]. Hence, NoC was suggested to improve the communication speed between the SoCs or MPSoCs. The NoC consists of a number of routers depending on the network size and it uses switching methodologies such as Virtual Cut-through, Store-and-Forward and Wormhole etc. [135].

NoCs utilizes different routing algorithms for routing the data keeping in view certain conditions in the network such as congestion or the amount of traffic. The common routing algorithms are XY-, west-first-, north-last-, and negative-first-routing [135]. Such algorithms are used to increase the performance by reducing the delay and maximizing the traffic utilization of the network. This makes it possible to increase the number of cores in an MPSoCs or SoC without adversely affecting the performance. However, the major drawback is that the network starts blocking the traffic due to high data traffic. This may cause certain problems in the network such as livelock, deadlock and starvation. Livelock and deadlock problems can be resolved by choosing an appropriate routing algorithm. Starvation can be prevented through implementation of a fair system in which all the packets have the same priority or through reservation of a specific bandwidth for low priority packets [135]. However, when such problems are encountered, debugging is usually required for sorting out the cause.

For a specific system, the NoC can be connected in different ways for achieving optimum performance. Therefore, there are different basic network topologies such as tree, mesh, torus, polygon, butterfly, star and ring topology [135]. Hence, NoCs may utilize different types of switching methodologies besides a variety of routing algorithms along with diverse network topologies for inter-chip communication. Due to such complexities traditional debugging techniques used for bus-based SoCs cannot be applied. Because of such complexities of the NoCs, traditional debugging methods used for bus-based SoCs may not be sufficient.

The most commonly used debugging technique is JTAG, originally proposed to perform system management tasks. Battaline et al. [136] suggested using JTAG as an interface to perform NoC debugging. However, such solutions no longer remain cost-efficient due to area requirement and speed limitations. Ciordas [132] presented a NoC analyzer which was able to perform NoC transaction monitoring at run-time. For transaction monitoring, the analyzer offered four different levels of abstraction. The levels start from low-level monitoring of the router data to the event abstraction level, where all analyzer modes can be configured at run-time. The transaction monitor, capable of monitoring the most difficult packetization, was implemented at the cost of one-fifth of the router area, where the total increase in the NoC area was found to be around 5% for several audio/MPEG SoCs. Due to connection monitoring, traffic was introduced in the NoC. The traffic introduced in comparison to the traffic of the monitored connections varied from a penalty of 41% in the connection-oriented memory-mapped scenario to a gain of 63% in the transaction event-based data-streaming scenario. However, as the analysis results of the traffic are also transmitted through the NoC, chip resources are seriously affected.

A graphical interface for NoC debugging was introduced by Möller [137]. The authors presented a tool called Hermes debugger, which took an events list as an input exported from an MPSoC, simulated through ModelSim. Hermes debugger used a graphical interface called NoCScope [137]. The main task of Hermes debugger was to interpret the changes in NoC signals over time and update NoCScope accordingly.

Vermeulen [138] presented a NoC monitoring infrastructure which could be instantiated at design time and was able to do performance analysis at real-time. However, the results of traffic analysis were transported through dedicated links hence, negatively affecting the chip resources. Stepniewska [139] presented an embedded debugging system for NoCs. A debugging enhancement was suggested to support the communication-centric debug of the system being designed. It was possible to enable and disable the functionality for each PE inside the network. The system was modeled in Verilog HDL and synthesized for Xilinx Virtex-5 FPGA. The cost of the presented tool was found out to be not more than 15% of the whole system in the worst case.

Goeders [140] presented a new signal-tracing technique which was specially designed for circuits, optimized by an HLS tool. The solution was proposed for debugging HLS circuits in the presence of

compiler optimization. The authors utilized a dynamic tracing technique which used the HLS schedule to determine the signals relevant for tracing each cycle. Then, the behavior of these signals was stored in on-chip memory. The system was found to be capable of recording the relevant data path registers for 180x longer than conventional tracing methods.

Cheng [141] proposed an RTL debugging methodology for an event-driven FPGA-assisted co-simulation system. The authors connected the internal nodes of the hardware to a PCI-extended bus instead of scan-chains which resulted in reducing the resources utilization by the debug logic. The presented method saved 30-50% of the hardware resources and 40-70% of the compile time. Run-time debugging was also possible as the signal under debug was sampled and updated at every simulation cycle. Co-simulation performance was found to be directly affected by an increase in the number of debugging probes.

2.3.2.3 Multi-clock Domain Debugging Methodology

Debugging for multi-clock domain system remains a critical design area because of the increasing number of multi-clock domains. As stated in Wilson Group Research report [27], the number of asynchronous clock domains is increasing with more than 50% design projects currently employing at least more than 2 clock domains as shown in Figure 2.7.

Main approaches for multi-clock domain debugging can be either a real-time trace such as ILAs or a run-stop and resume technique [89]. The first techniques use trace buffers to save the data with a non-intrusive approach i.e. without interfering with the normal function of the DUT (Device Under Test). The latter approach is based upon the BreakPoint-based Debugging (BPD) which sets the breakpoint to stop the system, dump the debugging data and then resume the operation.

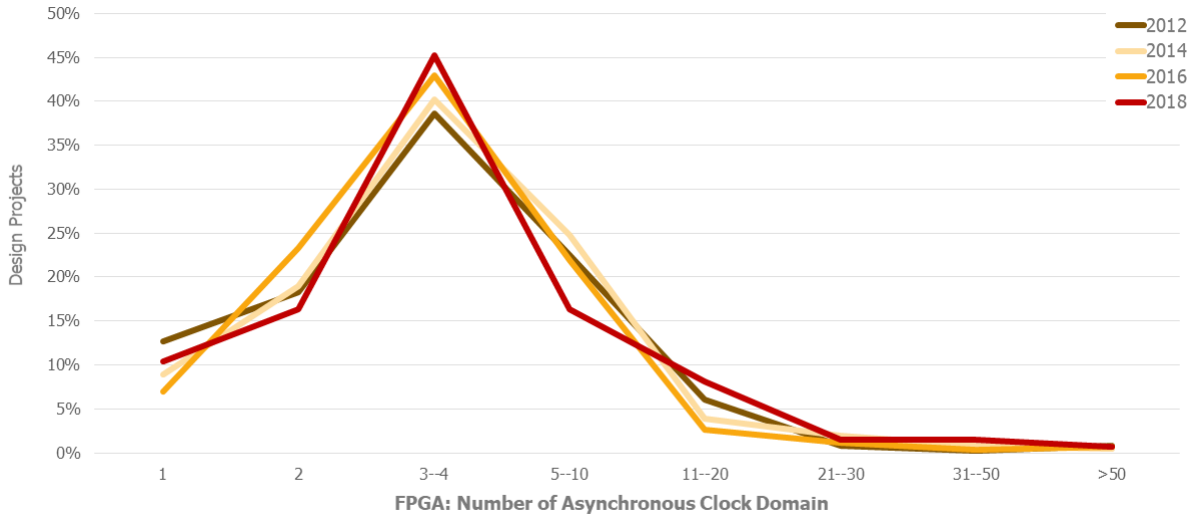


Figure 2.7: No. of clock domains in design projects [27]

For transaction-level debugging, the authors suggested a socket-based emulator [142]. Software running on the host PC was used for controlling the debugging hardware. On the contrary, this research work suggests an on-chip soft-processor such as Microblaze for controlling the debugging HW. The software platform was connected with the hardware JTAG.

In [143], the authors introduced a test access mechanism controller which used BPD at the cycle level granularity by gating the clock of all components in the SoC. The authors proposed that BPD can be superimposed with cycle-level granularity instead of handshake approaches which work on transaction-level of granularity. The authors suggested the IP to first become the master to access the bus. Hence, if the design comprises of many IPs, the IP with low priority may have to wait for a long time before it gets access to the bus. During this waiting phase, the design cannot be stopped until the handshake process is finished. Otherwise, the risk of data loss is eminent because of broken transactions. Hence, a synchronizer is necessary to take care of meta-stability issues [144]. However, because of the delays present in the synchronizer, it is difficult to gate the clocks in different clock-domains which may result in data invalidation [145]. This data invalidation was addressed by inserting a configurable synchronizer based upon the ratio of the operational frequencies.

In [146], the authors introduced a methodology to debug multiple clock designs. The authors introduced a HW/SW technique to address the data in-validation problem. The software was used to set up the breakpoint and calculate the appropriate time for issuing the pause signal. The hardware mainly was a clock controller which converted the pause signal from the software into a clock gating signal. The stop signal is necessary to stop the DUT at appropriate time to avoid any data loss. The mechanism could stop the DUT at the cycle specified.

In [147], the authors proposed an architecture for stopping the clock and dumping the scan in the presence of multiple asynchronous clock domains. The authors suggested a round-robin architecture for stopping multiple clocks so that data invalidation is prohibited. However, the authors used scan dump as the main technique. In [62], the authors proposed a solution involving readback for non-intrusive hardware debugging in the presence of multiple clocks. The clock stop signal from the main clock controller is used as an enable signal for multiple independent clocks. The main issue with the proposed solution is that it can readback the systems states at a specified clock cycle. But continuous tracing of the debugging data is not possible.

The main issue with embedded systems with multiple clocks is data invalidation which includes data repetition and data loss [148]. Two main solutions to the problem are the handshaking and the FIFO buffering [146]. The FIFO requires buffers, synchronizers, and counters; hence it suffers from latency issues. In [149], a low latency data transfer interface was presented which worked by aligning the different clock domains at a certain time. This scheme allowed data transfer without setup or hold time violations.

In [150], the authors proposed a low latency adaptive interface for Globally Ratiochronous Locally Synchronous (GRLS) designs. Such systems are defined as the designs in which several individual modules or partitions in a large design utilize a local synchronous clock within the module itself but communicate with one another at frequencies which are a submultiple of a single frequency [151]. In contrast, Globally Asynchronous Locally Synchronous (GALS) are the designs in which individual modules in a large design use a local synchronous clock within the module but communicate asynchronously with one another [152]. A periodicity cycle of the two clock domains is used in a heartbeat fashion to ensure that the data invalidation is prevented. The proposed solution is used to set breakpoints and then stop the clocks when the system hits the breakpoints. Then the data is shifted out for debugging. The clock stop signal is also communicated to other clock domains for maintaining the synchronization

As stated in the preceding paragraphs, all the methodologies for debugging multiple-clock embedded design are good. However, the methodologies present the solution to a different problem. i.e. how to avoid data-invalidation or data-aggregation when the clock is stopped through breakpoints. Such solutions are helpful when we know the location of the problem. In many cases such as intermittent errors i.e. the errors which are not permanent, the breakpoint based debugging techniques do not help. In order to capture such issues, continuous lossless debugging data is required to filter out the faulty data as was covered in our presented Device Start and Stop (DSAS) based debugging methodology [Khan1]. DSAS can be enhanced to multiple clock domains making possible the cycle-accurate debugging for multiple clock systems. In order to ensure that no data-invalidation takes place, the clock is gated based on the occupancy of trace buffers instead of stimuli from the simulation environment. Consequently, the DUT is clock gated a fixed number of clock cycles depending upon the depth of the trace buffers. Moreover, the trace buffers and the DUT are driven from the same clock hence, synchronization is not affected.

The main feature in our design is that we stop all the clocks deterministically by using the depth of the trace buffers as a clock gating parameter. The proposed design can handle GALS as well as GRLS designs. In our GALS-based debugging design, we clock the DUT and the trace buffers with its own clock. This helps in prohibiting data invalidation.

2.3.3 Observability Enhancement Tools

2.3.3.1 Signal Priority-based Connectivity Mapping

In order to meet the time to market requirements, the error-free design must be finalized at the earliest. Since 50% of the design time is actually spent on debugging, making the debugging process faster is a challenge [27]. Debugging can be made faster by circumventing the inherent invisibility of hardware. Although the DSAS approach presented in this research work permits a cycle-accurate replay of the design however, only for limited number of signals. Concentration networks were introduced to extend the observability to all available nodes. Concentration networks allow changing the signal subset without re-

compilation. However, many large designs have tens of thousands of debugging nodes. It may be cumbersome to connect these nodes to access networks. An automatic tool capable of providing connectivity generation is required. Moreover, in order to find the bugs as soon as possible, priority-based signal selection can be performed to select the signals based upon more probability for the errors. This necessitates devising an algorithm so that most relevant signals are debugged on priority.

Ko et al. [78] proposed a trace signal selection methodology that focused on an algorithm to estimate the states of the signal data which can be restated by monitoring specific gates in the circuit netlist. They also calculated the state restoration by measuring the ratio of total flip-flop states restored after applying the solution, to the original amount of trace data without restoration. The authors argue that a higher restoration ratio results in better signal selection. However, their method does not consider the "effectiveness" of the reconstructed data. For instance, a shift register's states can easily be restored, however, such restoration does not help in debugging. Furthermore, the authors applied their methods to small designs raising concerns about the scalability of the proposed methodology.

Liu et al. [153] proposed a trace signal selection methodology which focuses on the enhancement of error detection capability. Yang et al. [77] suggested a different methodology based upon error coverage. They suggested to inject the errors into the design, and through simulation, tried to find out the sensitivity of the flip-flops to such errors. This methodology enabled to choose signals which maximize error coverage. Similarly, Gao et al. [75] proposed a signal selection algorithm based upon error coverage in the fan-in cones so as to minimize the error suspect window.

Through an effective signal selection methodology, if state reconstruction is employed, even those states can be reconstructed which were not traced. Chatterjee et al. [154] proposed a state restoration based signal methodology utilization the simulation signals. In their work, they suggested to simulate the restoration process over few clock cycles and measure the restoration ratio. SAT-based methods [155][156] have also been proposed to select the signals considered to be most efficient in unobserved signals restoration. Apart from the above-referred methods, Wilton et al. [157] suggested performing signal selection at the HDL level. The main advantage of the technique is that working at a higher level of abstraction reduces complexity, resulting in a selection of signals directly consistent to the designer level of abstraction. This, in turn, helps in better understanding of the signal dependencies which makes the signal selection process easier.

Functional coverage has also been considered for design validation which takes into account how comprehensively a design has been exercised for faults [158][29]. Finite state machine (FSM) based signal selection capture the number of states, state transitions, or paths through the FSM that has been visited or taken during validation [159]. Hung et al. [160] presented a methodology which computes the expected number of state space which can be ruled-out by observing a specific set of signals. However, the proposed

methods are only valid for functional errors. Signal selection-based upon bit flip detection was proposed by Vali et al. [161]. The authors proposed an algorithm which focuses to improve the detection of bit flips during post-silicon detection. However, the authors considered very small designs for their research work and its scalability to large designs is unknown.

In all the above works, the authors used state restoration, error propagation, formal methods or state machine based signal selection methods. However, none of these research works focused on signal selection based upon the presence of patterns in the simulation results which is the focus of this research work. Furthermore, most of the above-referred signal selection methodologies focus only on functional errors. However, the proposed signal selection technique works for both the functional as well as intermittent errors.

2.3.4 Automated Error Detection

Because of the lossless nature of the proposed debugging solution (DSAS), the collected data is huge. Although, now the debugging difficulty due to limited data has been resolved since any amount of lossless trace as desired by the designer can be acquired. However, debugging is still challenging because human interpretation is required to find out the bugs. Therefore, the debugging difficulty increases with an increase in design complexity which in turn increases the design time. It is a practical requirement to test and debug any embedded design before deployment. But, testing and debugging processes have associated time and cost implications. For complex designs involving large and rapidly changing debugging data, manual analysis of the data is difficult. However, if the received debugging data can be associated to software verification environment, the debugging process can become much easier.

Currently, the main methodologies for hardware debugging through software are as follows.

2.3.4.1 Debugging through Software

Verification by simulation can be regarded as the main technique for performing functional verification of the hardware design. MATLAB / Octave [162] functions can be utilized for building a debugging and testing environment. The method is flexible and occasionally RS-232 serial cable may only be needed[163] as shown in Figure 2.8.

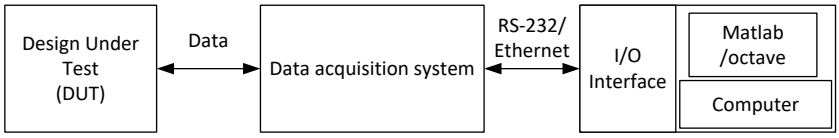


Figure 2.8: Debugging through software

Limited high-level programming may be used which can automate the verification process, reducing the amount of user intervention needed for verification and debugging of complex designs. Sometimes, the manual analysis of the test results creates a bottleneck in the verification process. Debugging of complex hardware designs can be accelerated by using high-level verification software [164]. The main benefit of using MATLAB / Octave as verification software is that complex programming tasks can be implemented in few MATLAB commands [165].

Although this debugging technique is simple, it incurs delay-time limitation. The main reason for such limitation is the analog-to-digital conversion (or vice versa depending upon the requirement) and the transmission time between the DUT and the processor. The delay for Ethernet may be less than serial communication. However, such software-based debugging solutions may not be appropriate for FPGA-based designs which operate at very high frequencies.

2.3.4.2 Hardware co-simulation based Debugging

A hardware co-simulation based debugging model is discussed in [166][167]. In this framework, first, the algorithm is implemented in MATLAB. Based upon the algorithm, system architecture is finalized and subsequently, the modules are implemented and verified in MATLAB / Simulink. After verification, GR is obtained. Then, this GR is utilized for RTL coding. Hardware co-simulation can then be performed as illustrated in Figure 2.9.

As evident, the design process needs to start from MATLAB. But, if the algorithm is difficult or completely impossible to implement in MATLAB, the process of hardware generation cannot be started.

2.3.4.3 Knowledge-based Automated Debugging System

An expert system integrates a knowledge base which contains the accumulated experience, and an inference engine which applies the knowledge base to a particular situation, based upon a set of rules [168] as shown in Figure 2.10. The expert system capabilities can be enhanced by inserting data to the knowledge base or to the rules. Such systems may contain machine learning capabilities which allow them to improve their performance based on experience.

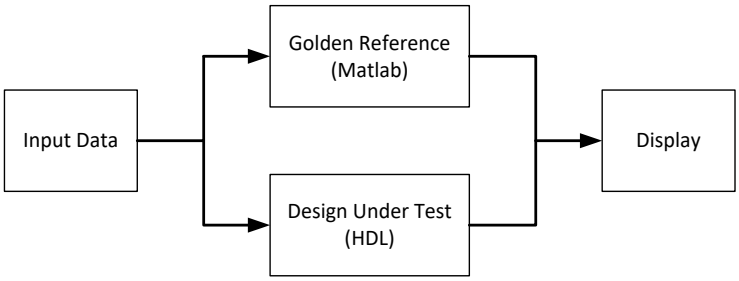


Figure 2.9: Hardware co-simulation based debugging

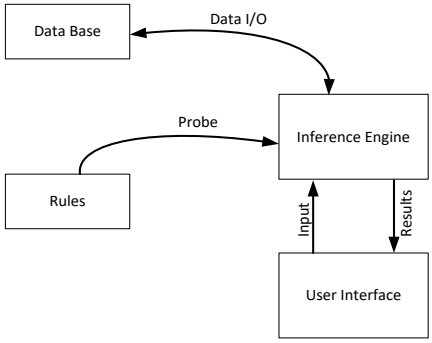


Figure 2.10: Expert system

Usage of a knowledge base for system debugging is not new. In [169][170], an expert system was presented to debug Pascal programs. This expert system proved helpful for locating and correcting errors in Pascal programs. Moreover, a knowledge-based automated debugging system was presented in [171]. The system was used for debugging of Pascal-based programs by determining possible causes for runtime, compiler and logic errors.

2.3.4.4 Rule-based Inference System

Based upon the expert system analogy, a methodology is proposed for easing the debugging process by using a visual debugging tool, implemented in MATLAB, and hence, utilizing the power of MATLAB for system debugging. We have proposed a new verification method for hardware debugging by using MATLAB as a tool and rule-based inference system as a verification method. In the proposed verification system, a GR is used which can either be defined using the inference system or user-defined. The objective is to find the bugs without requiring to run the system intermittently. This is accomplished by debugging the complete window at once and using the power of the MATLAB-based debugging system to reduce the debugging time and hence, the overall design cycle.

2.3.4.5 FPGA-in-the-loop based Debugging using Rule-based Inference System

FPGA-in-the-loop (FIL) based simulation/emulation can also be used for capturing design errors. Transactions are used as the main technique for debugging. The debugging system generates a stimulus (seed) and waits for the response hence completing a transaction. The same cycle is repeated intermittently and the data is used for debugging.

A simulation/emulation architecture was proposed in [114]. Transactions were used for communication between the DUT and the debugging environment. The authors suggested the idea of controlled and uncontrolled time. A clock based upon a controlled time was generated to control the DUT hence creating a cycle-accurate framework.

Nakamura et al. [172] introduced a similar transaction-based approach which utilized communication through shared registers for transferring the data. The framework utilized a clock enable for the verification target, hence maintaining a consistent absolute time between the software and hardware environment. As the methodology is based on transactions, it needs a stimulus in the form of a test vector. Subsequently, it records the output of the DUT through the software environment. But, limited debugging of the DUT was possible. Hutchings et al. [173] introduced unified hardware debugging environment. Simulation or hardware execution was made possible through an API by using runtime control and breakpoints. However, Xilinx configuration readback feature [174] was used for the research work. Hence, data from intermediate combinational signals were not available.

Yang et al. [175] introduced a cycle-accurate debugging methodology for FPGAs. It used an internal node probing technique which increased the FPGA internal visibility through scan chain insertion. However, the design observability was reciprocally related to the area overhead or the co-simulation speed.

An event-driven runtime debugging methodology for FPGA-based co-simulation was proposed [141]. When the stimulus was changed, an output of the DUT was updated. To keep the resource consumption low, data recording circuitry such as scan chains or trace buffers were not used. Debuggable ports and signals were directly connected to the bus through multiplexer logic. Therefore, the methodology may be valid for designs having a low frequency of event change or for small designs. Data loss is imminent for large designs.

Asaad et al. [6] proposed a cycle-accurate and cycle-reproducible emulation system for multicore systems. The approach worked by stopping the clock at the region of interest. Then, Xilinx configuration readback feature was used for debugging. However, configuration readback can only read the state of the FPGA at an instance. Hence, catching intermittent errors is extremely difficult because of their random nature. Furthermore, the process is slow as the configuration memory also contains information about the architectural details of the FPGA besides state elements. Though the authors proposed to use breakpoints, appropriate insertion of breakpoints needs a detailed knowledge of the system.

For simulation acceleration, Koczor et al. [176] considered a transaction-based method. The process was controlled by the software running on the PC. Through their suggested emulation system, FIL simulation was also possible by utilizing MATLAB modeling methodology. Huang et al. [177] presented a transaction-based HW/SW verification environment. The authors discussed different scenarios such as RTL/FPGA co-simulation or SystemC-FPGA co-simulation. However, they focused on transaction-based HW/SW validation.

In the transaction-based FIL debugging methodologies presented in the preceding paragraphs, stimuli are generated by the simulation environment. Depending upon the stimuli, response from the DUT is awaited

to capture the errors. However, the presented methodologies are not suitable to find errors (such as intermittent errors) in DUTs which do not operate on this stimulus / response cycle.

Intermittent or soft errors can only be captured through cycle-accurate lossless debugging systems. However, for such debugging systems, manual interpretation and analysis of the test results for each transition constitutes a bottleneck. Hence, an automated error detection is required which can perform correspondence resulting in automation of the analysis process.

Many researchers utilized an MATLAB/Octave model of the DUT as GR. However, for complex designs, it can be time-consuming to first create a DUT model and then perform the comparison. This problem was tacked through high-level validation methodology [38]. It works by performing either a behavioral, post-synthesis or post-implementation simulation and storing the results. The cycle-accurate response of the DUT is then compared with the simulation results. However, due to the unavailability of lossless debugging data, the authors suggested an output data capture window. Debugging data outside the capture window was discarded. Consequently, they compared only a subset of the DUT output to the simulation output. We proposed DSAS-based debugging [Khan1] which is capable of capturing functional as well as intermittent errors owing to its apparently unlimited trace window. Hence, we integrated the DSAS into the rule-based inference system [Khan5] for performing correspondence analysis between the lossless debugging data and the HDL simulation data of either a behavioral model, a post-synthesis or a post-implementation model of the DUT used as a GR. However, to the best of our knowledge, there is no debugging methodology which can capture intermittent errors in the absence of a GR.

2.3.5 Automated Debugging using Artificial Intelligence

Predictive analysis techniques can be integrated with post-silicon validation for bug detection. Machine learning can be used for detection of anomalies. i.e. any deviation from normal behavior. However, training data is required to learn the correct behavior. The training data can be labeled as passing (positive label) or failing (negative label) samples. In normal operation, training is usually done on passing samples instead of negative samples. This is because the passing samples are more frequently available than the failing samples as was done in [178].

Similarly, data prediction can also be employed for design validation. The test data retrieved from the DUT can be used as input to a machine learning algorithm which can then be used to make predictions about the bugs occurring on new designs as proposed in [179]. Machine learning methodology can be used to model a problem by using the input and output datasets. Hence, it makes possible to fully ignore the mathematical representation of the system, which in turn reduces the modeling complexity [8].

An error localization strategy was presented in [180]. The iterative machine learning-based approach was able to localize the errors by using the same regression tests multiple times on the prototype. One major difference between the proposed and the earlier approaches [50][181][182] is that failing results can also

be used for error localization. The authors used the machine-learning algorithm DBSCAN for this purpose [183]. Based on the passing and failing test results, the framework was able to identify the time and occurrence of these errors. The authors used the framework presented in [153] for their experimentation. Through a series of executions and interaction between the passing and failing trace result, the authors were able to localize errors.

In [184], the authors suggested enhancing the observability of the internal signals at the post-silicon level through a learning algorithm. The methodology is based upon the idea that the unknown signals can be decided based upon the value of its nearest neighbors. The authors proposed a debugging methodology based upon the nearest neighbor algorithm which could then be used for fault localization.

In [185], the author suggested using data mining for pattern extraction. They further suggested using the methodology for functional verification by using data mining techniques. In [186], the authors proposed to use machine learning techniques for automating the diagnosis of trace dump and bug localization. The trace dump processed through Map-reduce and K-mean clustering was used to identify the rare test segments. The authors used commercial post-silicon debugging framework [69] for trace capture.

It is evident from the preceding paragraphs that many researchers have applied clustering-based machine learning to lossy debugging systems (like ILA) for bug detection and error localization by using clustering techniques. In [187], the authors proposed that the time series data can be predicted using an RNN. Cycle-accurate lossless trace dump resembles time series data which can be achieved through clock management of the DUT. In [Khan1], a methodology is discussed which generates continuous stream of lossless data by automatically stopping the clock based upon the occupancy of the trace buffer. Similar techniques can be used to generate lossless time series debugging data. Hence, in this research work, an RNN based machine learning framework is proposed which can be used to automate the debugging process even in the absence of GR. To the best of our knowledge, nobody has used RNN to lossless cycle-accurate trace-dumps which can be extremely helpful to predict the debugging data.

2.4 Summary

When an error is encountered in a circuit, it becomes important to find the root-cause of this error. The error can be found out either through a scan-based or a trace-based instrument. In this chapter, we reviewed both of these techniques along with their merits and demerits.

We also discussed different techniques to enhance the on-chip observability. In this regard, we reviewed access networks, priority-based signal selection, data compression and incremental insertion techniques. We also reviewed different automated error detection techniques which can eliminate human interpretation for capturing functional or intermittent errors.

Chapter 3 Cycle-accurate Lossless Debugging with Replay

Because of the limitations of software-based simulation systems, the focus shifted towards hardware simulation. However, the major problem faced by hardware remains its invisibility [31]. Due to invisibility, only the signals available at the pins of the FPGA can be monitored. FPGA vendors introduced ILA cores as a remedy, which can be instantiated in the design and used for hardware debugging. These embedded logic analyzers can solve visibility problems. However, these logic analyzers consume FPGA resources resulting in a limited trace for debugging which can also be termed as lossy trace. Debug becomes difficult due to this lossy trace. Traditional debugging solutions require a tradeoff; if more debug trace is required, more hardware resources need to be reserved for debugging. However, if maximum available resources are reserved for debugging hardware, still, only few thousand clock cycles can be monitored. Hence, a debugging solution is needed which can provide a lossless, cycle-accurate trace of debugging data while utilizing minimal resources at the same time. Furthermore, these logic analyzers also require a suitable trigger signal for debugging data logging. This requirement becomes a key bottleneck since it needs the debug engineer to have complete understanding of the hardware design. In order to solve this issue, a new methodology is introduced that addresses the limited trace size issues. The proposed methodology also gets rid of the trigger requirement.

Moreover, changing the signal set during debugging is quite common. Traditionally, when the signal set is changed, a recompilation of the design is required. Frequent recompilations increase the debugging time making the debugging process time-inefficient. Although this issue has been addressed by several researchers as already covered in Chapter 2. Still, an augmented solution providing a lossless trace of any signal along with ability to change the signal set without recompilation could be helpful during the debugging process.

Another issue faced during the debugging process is data transmission bottleneck. As the debugging data generation is faster than the data transmission through available means such as Ethernet etc, a bottleneck is created on the transmission side. Any solution which can accelerate the data transmission could improve the time-efficiency of the debugging process.

Another issue is the instantiation of the debugging circuitry besides the DUT. Then the PAR process is conducted. The placement of the debug circuitry can affect the timing closure of the hardware design. Furthermore, after verification, the debug core is required to be removed from the design. This may affect the timing closure of the design once again. Hence the main challenges are:

1. To provide a resource-efficient lossless cycle-accurate debugging solution while getting rid of the trigger signal at the same time.

2. Time-consuming recompilation should be avoided.
3. To resolve the data transmission bottleneck.
4. Resource utilization by other applications when debugging is not required.
5. The debug circuitry may affect the timing behavior of the design.

The above-mentioned challenges have been addressed in subsequent sections in this chapter.

3.1 Cycle-accurate Lossless Debugging with Replay

This chapter presents an FPGA debugging methodology utilizing device run and stop approach. As previously mentioned, the main problem in debugging is to increase the debugging trace in such a way that lossless debugging data can be acquired. Although there can be numerous solutions as mentioned in Chapter 2, we proposed the idea of managing the clock of the DUT in order to acquire lossless debugging data by using limited BRAMs. The debugging system starts and stops the DUT based upon the occupancy of trace buffers. Since the debugging data is saved to external memory on the connected PC, there is no limitation on the quantity of data being monitored. The methodology guarantees a debugging system which is capable of providing a lossless debugging trace by controlling the clock using cycle-based synchronization. The recorded trace can then be replayed offline [188]. Cycle-accurate replay makes it possible to replay the recorded trace cycle-by-cycle. Hence, post-silicon, complete cycle-accurate details can be reconstructed which can be very helpful to capture functional as well as intermittent errors during the debugging process.

3.1.1 Debugging Framework

This section presents the proposed debugging method based on the device start and stop approach. The processor (ARM in the case of Xilinx Zynq device, Microblaze for rest of Xilinx FPGA families, also possible for other FPGA families after minor modifications) is employed to gather the data from onboard trace buffers (4KB BRAM). When the trace buffer is full and the signal is sent to the clock manager for stopping the clock of DUT, Direct Memory Access (DMA) core moves the data from the BRAM to the memory connected to the processor. The processor then transports the data from memory to the terminal (PC) using Ethernet where it is saved to any memory devices like HDD, SD Card, etc. During the data transmission phase, the DUT is stopped hence no data is lost. At the terminal, the data is logged to a log file. The log file is in *.txt format. First the data is de-multiplexed and then converted to VCD [189] format so that the trace data can be examined through waveform viewers, such as GTKWave [190]. A block diagram of the debugging methodology is shown in Figure 3.1.

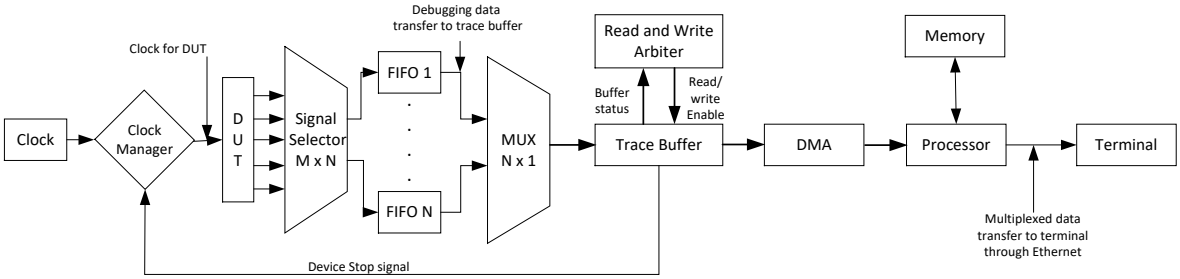


Figure 3.1: Debugging model

The main benefits offered by the proposed technique are a cycle-accurate trace, unlimited debug window, minimal use of scarce FPGA resources, no loss of debugging data and no requirement of an external emulation system. User intervention for saving the data after the BRAM is full is also not required. Furthermore, open-source waveform viewers can be used while using DSAS approach, removing the dependency to use proprietary software. Hence, the solution is cost-effective as well.

3.1.1.1 Signal Selection Module

The signals which are required to be monitored are routed through the signal selection module. The signals can be traced after selection. Although the trace signals have been limited to 16 to reduce resource consumption, the traced signals can be increased if sufficient FPGA resources are available. Hence, the signal selection module ensures full visibility of the FPGA.

3.1.1.2 Trace Buffers

Monitored data is saved to the trace buffer. BRAMs available on the FPGA fabric are utilized as trace buffers. Their size can be selected by considering the available resources. In order to limit resource utilization, only 4KB BRAM is used. Thus, during one DSAS cycle (one read/write operation), 4KB of trace data is moved from BRAM to the external memory. DMA is utilized to speed up the data move from trace buffer to memory. Besides speeding up the shifting of data, it also spares the processor for data transmission via the Ethernet interface.

3.1.1.3 Clock Manager

When the trace buffer is full, a clock gating signal is conveyed to the clock manager to stop the DUT clock as shown in Figure 3.2. Clock manager keeps polling for the signal status. Upon receiving the signal, the clock to the DUT is halted.

3.1.1.4 Read and Write Data Arbiter

This finite state machine eliminates the need for manual intervention. It starts and stops the clock based on the occupancy of trace buffers instead of stimuli from the simulation or emulation system. The clock to the DUT is stopped instantly when the trace buffers are full. The clock is automatically started when the trace

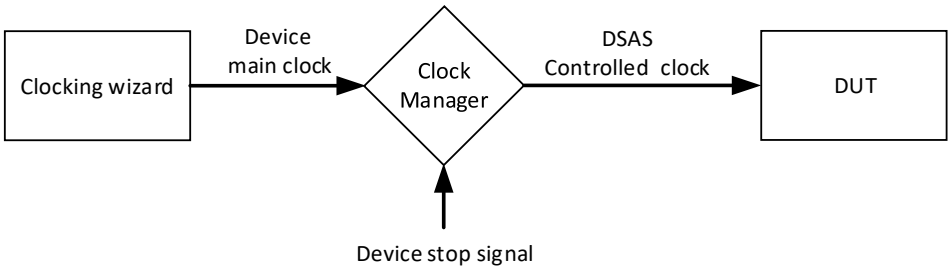


Figure 3.2: Clock manager

buffers are empty after data transfer. Consequently, the DUT is clock gated a fixed number of clock cycles depending upon the depth of the trace buffers. The block diagram of the arbiter is shown in Figure 3.3.

The read and write arbiter accepts two inputs, x (buffer full) and y (buffer empty). Based upon these signals, it generates read and write commands for the trace buffers and clock gating signal for the DUT. As the state machine depends on its present state, the proposed state machine can be categorized as a Moore Machine [191]. Initially, both inputs are 0 keeping the machine at Init State. When the input to the state machine x is 0 and y is 1, the state is changed to Write state (read=0, write=1) hence generating a write command. When the FIFO is partially latched, its input x and y changes to 0 keeping the machine in its current state. When the buffer becomes full after capturing the data; it changes the input x to the state machine to 1 and y to 0. This forces the FSM to moves to Read state (read=1, write=0) hence generating a read command. When the FIFO is partially read, input x and y changes to 0 keeping the machine in current state. After the data has been read from the buffer, the buffer is empty again forcing the input to the state machine x to 0 and y to 1. This forces the FSM to moves to Write state (read=0, write=1) and the cycle continues.

The data written and read from the buffers is moved to memory through DMA. The data is then transferred to the terminal using Ethernet.

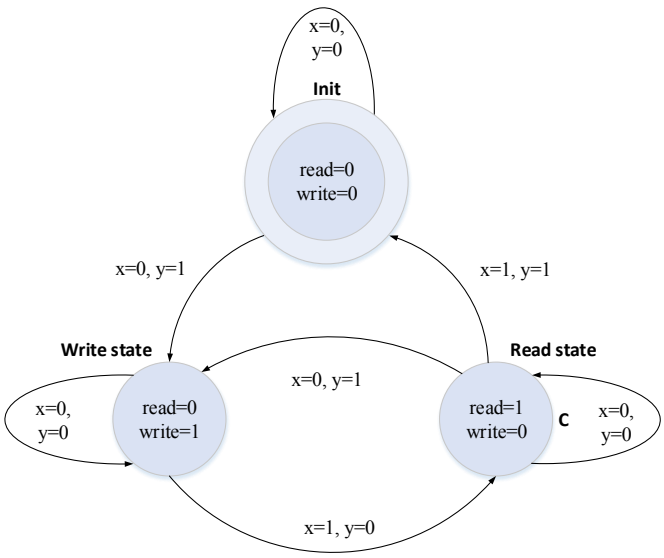


Figure 3.3: Read and write arbiter

3.2 Access Network Generation

The debugging system proposed in Figure 3.1 can be utilized to acquire lossless debugging data. However, such solutions utilize scarce FPGA resources for logging data to trace buffers. This severely limits the number of monitored signals necessitating to employ specialized techniques for enhancing the number of monitored signals. During the design verification process, the portion of the design which requires debugging may change over time and cannot be anticipated during design time. If the signal set is required to be changed during debugging, then re-synthesis is required. However, re-synthesis consumes a lot of time especially for large designs hence, adversely affecting the time to market.

The above-narrated problem can be solved by utilizing a flexible network which permits the verifier to choose the signals of interest. A suitable debugging system can process these signals as required. Such networks permit reconfigurability of the signal set at debug time hence eliminating the requirement to resynthesize the design for debugging. Such networks also need hardware resources. Hence, a resource-efficient access network is needed, which provide efficient connectivity at the cost of minimum hardware overhead.

In this section, two different access network methodologies will be presented i.e. gate-based and multiplexer-based. The focus of these two methodologies will be to reduce the hardware resources required for an access network: for large designs, abundant resources may not be available for such networks.

3.2.1 Access Network Description

During the design phase, the behavior of the design during the verification phase cannot be predicted. Hence, it is extremely difficult to identify the error-prone signals so that they can be connected to the debugging system. Instead, the designer normally selects a much larger signal set from the design for connecting them to the debugging system. In order to connect a very large set of nodes to the debugging system, large hardware resources are required. Consequently, for effective resource utilization, the number of signals that can be processed simultaneously must be limited. A processor configurable access network can be utilized to accomplish this task as shown in Figure 3.4. The technique provides connectivity to the available nodes without requiring re-synthesis of the design by changing the contents of the configuration register.

A synchronized output for all the available nodes connected to the access network can be observed through the configuration register. This can be accomplished by re-starting the test from time T0 (the starting time of the debugging session) when the signal set is changed. This technique ensures a completely synchronized state of all the monitored signals which can then be used for debugging.

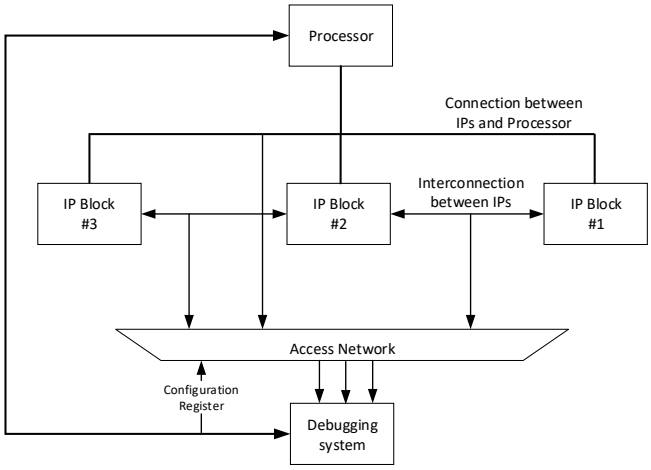


Figure 3.4: Access network interconnection applied to a DUT

There are multiple ways to realize an access network such as concentration network, multiplexer network or permutation network [91]. Concentration network has been claimed to provide the best performance by using the least resources [95]. However, because of unnoticeable difference in resource utilization between the multiplexer and concentration network, the claim has not been proven [90]. In this research work, we will be presenting two different approaches for realizing an access network namely the gate-based approach and the multiplexer-based approach.

3.2.1.1 Gate-based Access Network

For this research work, we have chosen 16 signals as the extracted set (see Figure 3.5). However, it must be highlighted that more signals can also be extracted at the cost of extra hardware resources. We have defined the upper bound on the number of signals to reduce resource utilization. Moreover, debugging may become cumbersome for the DSAS approach in case of too many extracted signals [Khan1].

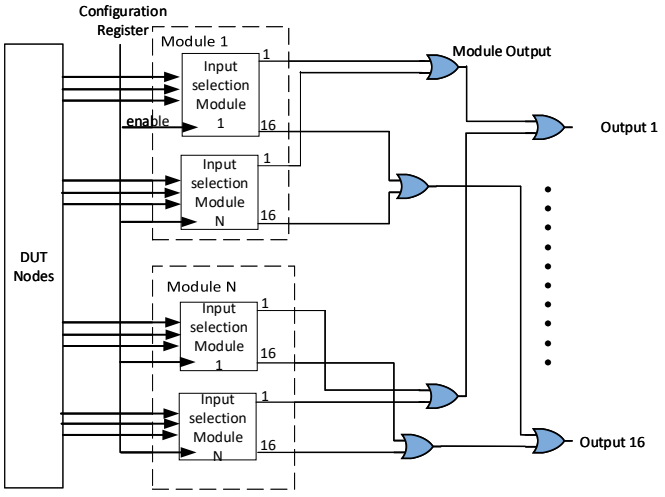


Figure 3.5: Gate-based access network

The signal set can be changed through a selection register which can be configured by the processor to realize any signal selection hence, allowing full visibility of the design. The number of debug-able signals can range from hundreds to thousands, depending on design complexity and hardware limitations. We proposed a specifically designed input selection module (m x m) having an enable port which can be utilized to disable or enable the selector. When an input selection module is enabled, the complete set of signals present at its input are connected to the OR gates. Each OR gate has as many inputs as the number of input selection modules. Only the output of an enabled selection module is valid at a specific time. The output of the rest of the selection modules is not valid. Instead of the proposed input selection module, another type of network known as a hyper-concentrator can also be utilized. A hyper-concentrator is a network with x inputs and x outputs in such a way that any input set can be mapped to any contiguous output set disregarding the ordering of the outputs. Narasimha et al. [192] presented a hyper-concentrator design, which has a low cost in terms of switches and reasonably low network depth. The outputs of hyper-concentrator are registered to ensure network operation at the speed of the integrated circuit. However, such concentrator designs are complex hence, we have used the self-designed selection module.

3.2.1.2 Multiplexer-based Access Network

We have also proposed an access network by following the multiplexer based approach. This type of network comprises of n inputs and m outputs, where n is always greater than m. The proposed network has the capability to choose any subset of inputs while the same is missing in the first methodology. The n inputs are divided into k groups, where k is the quantity of input selection modules. We suppose that the size of each of these input groups is ≤ m, where m is the total number of outputs.

The proposed network comprises of two stages as shown in Figure 3.6. In the first stage, each of the k input groups is considered independently. A special type of input selection module (m × m) is utilized whose

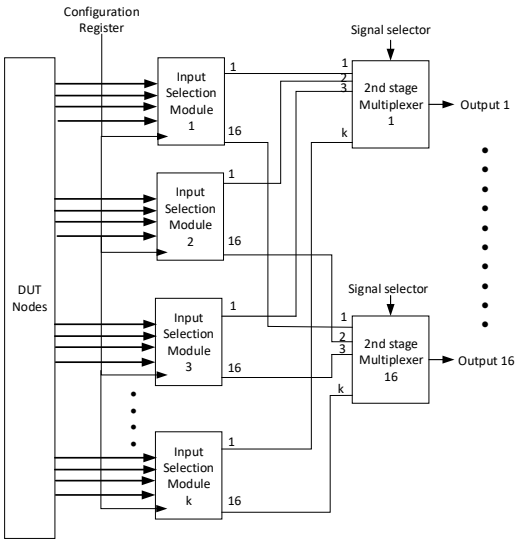


Figure 3.6: Multiplexer-based access network

control port is used to select the required signals at the output. As previously explained, a hyper-concentrator may be used instead, but in such case retaining the ordering of outputs is not possible. Hence, we have proposed an input selection module which maintains the order of the output. In the second stage, outputs of the k input selection modules are combined by utilizing a multiplexed bus structure. Multiple cascaded stages of multiplexers can be used to extend the network as required. This implementation offers advantages over the gate-based solution because the selection of the input signal to the multiplexer stage is possible. Configuration registers can control any input selection through the input selection module or the multistage multiplexers. Consequently, any set of input signals can be selected. The controlling processor can, therefore, control the access network through the configuration registers by issuing appropriate control words. The technique ensures full visibility by changing the contents of the configuration register. Hence, frequent re-synthesis of the design just for changing the signal set can be avoided. JTAG [136] can also be utilized to configure the access network instead.

3.3 Data Compression

In section 3.1, a new methodology for lossless FPGA debugging was presented. The proposed debugging system collects the data from onboard trace buffers. Once the trace buffers are full, the DUT is stopped by the clock manager and then the processor transfers the data to the terminal through Ethernet. It was highlighted that one read/write cycle resulted in moving 4KB of trace data to external memory. It was noted that data generation is faster than data transmission through Ethernet. Hence, off-loading the data to the server/terminal creates a bottleneck.

In order to increase the amount of trace data that can be stored on the FPGA while keeping the trace buffer size constant, compression of debugging data was proposed [76]. However, in spite of trace data compression, transmission channel bottleneck still exists which results in loss of debugging data. An efficient lossless compression technique for continuous streaming data is required which can improve off-load time of the debugging data and help in faster data dispensation through the transmission channel.

In this section, a new methodology is introduced that addresses the data transmission bottleneck. The research work presents an efficient lossless trace data compression technique for processor-based embedded designs. It addresses the bottleneck at the transmission channel by decreasing the off-load time of the trace data through data compression with no architecture overhead. It must be highlighted that the proposed scheme should be used if the hardware resources are limited. For designs having abundant hardware resources, any hardware-based data compression scheme may be adopted.

3.3.1 Data Compression Methodology

Many compression techniques have been introduced which provide excellent compression but at the cost of huge area overhead. However, keeping in view the FPGA resources, a compression technique is needed which can not only provide good compression but also introduces small area overhead. An important point

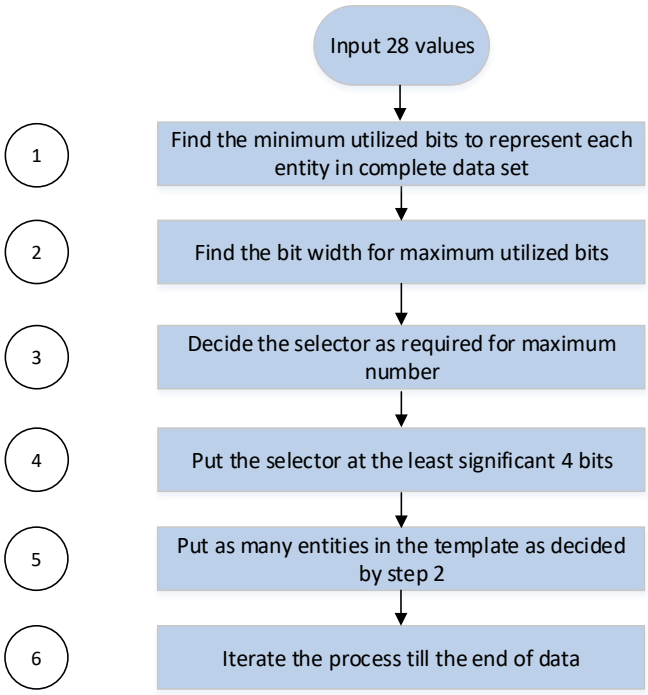


Figure 3.7: Simple-9 algorithm

to note is that during data transfer from or to memory, the data is handled as bytes or words. Based upon the above-mentioned constraint and ensuring fast processing with good compression performance, Simple-9 (Sim-9) algorithm with minor modification has been employed as shown in Figure 3.7 [193].

The Simple-9 algorithm is based upon 32 bit binary numbers. When we want to save binary numbers, normally we have lots of leading zeros [193]. The main idea of the algorithm is that one 32-bit word should be used to store as many values as possible. Based upon the algorithm, the block diagram of the Simple-9 (Sim-9) methodology is shown in Figure 3.8.

The least significant 4 bits are used to store a selector which represents the information regarding bit width used to store data in the remaining 28 bits. As the name depicts, the selector utilizes 9 distinct values to store the values as shown in Table 3-1. However, when the data is transmitted through Ethernet, it becomes

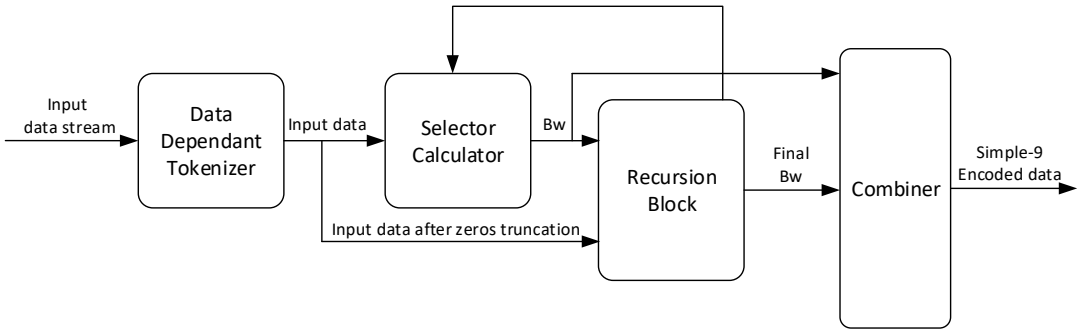


Figure 3.8: Simple-9 block diagram

important to identify the last frame along with the number of entries. We utilized the leftover selector values to identify the last frame and termed the compression methodology as Modified Simple-9 or simply MSim-9.

It is pertinent here to explain the algorithm with an example. Consider the following 14 integers as input.

{6,4,1,3,1,5,7,1,1,13,21,12,1,19}

At step 1, it is required to find the bits needed to represent the integers which are given below.

{3,3,1,2,1,3,3,1,1,4,5,4,1,5}

As evident, the maximum bit width for the first 9 integers is 3. After consulting Table 3-1, maximum of 9 entities can be encoded using a bit width of 3 utilizing 0010 as selector. After deciding the selector, the entities are put into the output word sequentially after placing the selector at the least significant bits. Hence, the output word comes out to be {001 001 111 101 001 011 001 100 110 0010}. As one bit is unused for bit width 3, the final output word after appending 0 as the most significant bit becomes {0 001 001 111 101 001 011 001 100 110 0010}. Similarly, the maximum bit width for next 5 entities is 5. Hence, these entities can be encoded using bit width of 5 with selector 0100. Hence, the resulting output word can be formulated as {10011 00001 01100 10101 01101 0100}. Unused bits can be padded at the last as {000 10011 00001 01100 10101 01101 0100}. Similarly, the algorithm can be iterated for complete data set.

After going through the previous example, the compression of data can be visualized. The 14 integers which were previously represented by 14 words can now be represented by only 1 word. In the second example, the 5 words have been compressed into 1 word.

Table 3-1: Simple-9 selector description

Selector	Number of encoded entities	Bit width (Bw)	Number of unused bits
0000	28	1	0
0001	14	2	0
0010	9	3	1
0011	7	4	0
0100	5	5	3
0101	4	7	0
0110	3	9	1
0111	2	14	0
1000	1	28	0
1001	Last Frame with 14 entries		
1010	Last Frame with 9 entries		
1011	Last Frame with 7 entries		
1100	Last Frame with 5 entries		
1101	Last Frame with 4 entries		
1110	Last Frame with 3 entries		
1111	Last Frame with 2 entries		

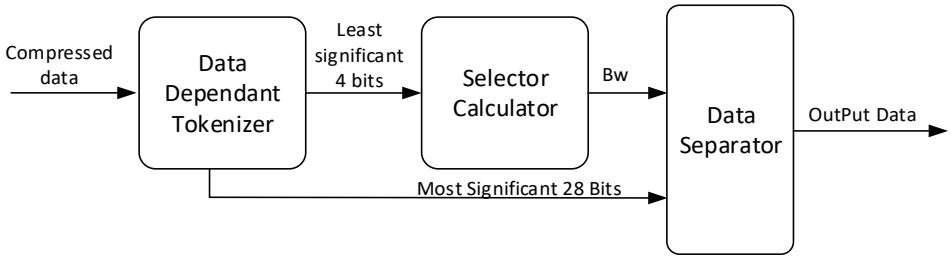


Figure 3.9: Decompression module

3.3.1.1 Data Decompression

Decompression is necessarily a reverse operation of compression. When the encoded data is received at the decompression module, the selector is examined. Based upon the selector as per Table 3-1, the module decides about the number of entities compressed in a specific 32-bit packet. The next step is to extract the data entities from the data packet. Block diagram of data decompression module is shown in Figure 3.9.

In order to explain the decompression process, the previous example can be utilized. Once the decompression module receives the compressed data such as {0 001 001 111 101 001 011 001 100 110 0010}, the unused bit is discarded. This results in the received data to be {001 001 111 101 001 011 001 100 110 0010}. Then, based upon the selector, the bit width is decided as 3. Consequently, the output integers can be decoded as {6,4,1,3,1,5,7,1,1}. Similarly, the next output word is decoded as {13,21,12,1,19}

3.3.2 Fast FPGA Debugging by Data Compression

In order to coordinate the debugging activities, we have utilized a debugging processor as mentioned in section 3.1. The same processor (ARM for Zynq [14] and an embedded processor for FPGA families without Zynq) can also be utilized for data compression in order to fix the data transmission bottleneck. The processor reads the data from the trace buffer and then compresses the data by running a software routine before data transfer. The compressed data is then transferred through Ethernet. After data reception at the terminal, the compressed data is passed on to the MATLAB environment. The received data is first

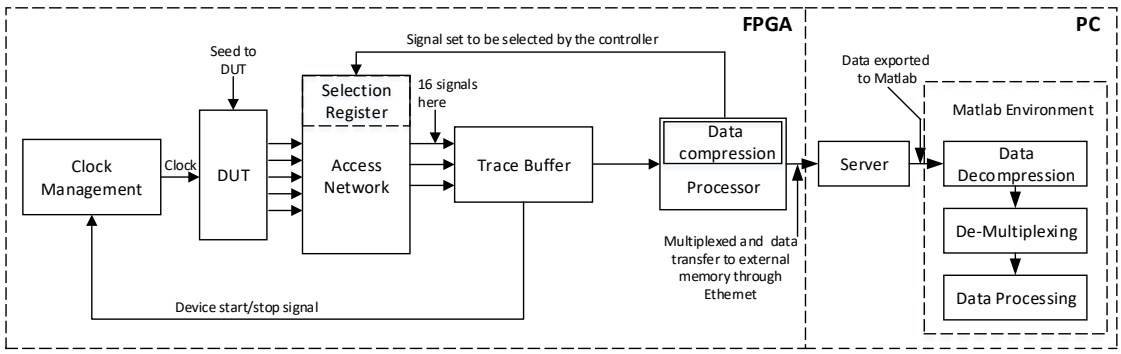


Figure 3.10: Fast debugging by data compression

decompressed and then de-multiplexed. Further data processing is then carried out inside the MATLAB environment. A block diagram of the debugging methodology is shown in Figure 3.10.

The main benefits of the technique are an effectively unlimited debug window (since the embedded design is not being clocked during data dispensation) resulting in no loss of debugging data and fast data transfer through the transmission channel by using data compression at the processor with no area overhead.

3.3.2.1 Interfacing

The debugging system hardware is connected to the terminal through Ethernet using the user datagram protocol (UDP). UDP has been implemented using Xilinx LWIP utility. But LWIP allows a maximum transmission unit (MTU) of 1500 bytes per packet with an overhead of 28 bytes [194]. In this research work, the payload is defined as 1024 bytes.

In order to receive the data at the terminal, an array of 1024 bytes has been reserved. So the server expects valid data of 1024 bytes per packet. But the output data packet has been compressed into an unknown number of bytes. Hence the transmitted packet does not have a fixed number of data bytes. When the data is received at the server, the data is saved in the 1024 bytes long reserved array. After the reception, the array is composed of valid data proceeded by garbage values. This poses a problem at data decompression. However, this problem can be solved by writing FFFFFFFFh at all memory locations of the reserved array. When the data is received at the terminal, the Winsock based application overwrites the data array. The data (FFFFFFFFh) can then be filtered before data processing for decompression. Once the debugging data has been received at the terminal platform, it is used by MATLAB for debugging. In order to control the whole process of debugging and streamlining the process, a graphical user interface has been developed using Matlab GUIDE.

The compressed data can then be transferred through Ethernet. Hence, without any change in hardware design and without using any FPGA resources, a speedup for debugging is achieved.

3.4 Incremental Compilation for Embedded Designs using Dynamic Partial Reconfiguration

On-chip visibility enhancement solutions such as ILAs mainly operate on the design before the (PAR) process. When the signal set is required to be changed, a recompilation is required which can be time-consuming for large designs. Moreover, these tools instantiate the user circuit with trace buffers before mapping, resulting in fewer resources available for the original design. It can prove to be hazardous for the design, such as the hardware design may not fit in the FPGA device due to resources used by the trace buffers. Another problem is that the insertion of debug circuitry and frequent re-compilation can alter the placement of the design. This can affect the timing behavior of the design due to design recompilation which may result in a bug to change or disappear [84]. The timing behavior may be affected both during instantiation or the removal of the circuitry after the debugging process.

Hence, we can summarize the problems encountered due to trace-based debugging circuitry as follows:

a. Time-consuming recompilation is required whenever debug instrumentation is added to the design.

b. As the trace circuitry is instantiated before the PAR process, fewer resources are available for the design.

c. The debug circuitry may affect the timing behavior of the design.

The above-mentioned problems can be addressed by utilizing DPR [16] for debugging.

3.4.1 Incremental Compilation using Dynamic Partial Configuration

By using the Dynamic Partial Reconfiguration (DPR) flow, any portion of the FPGA can be reconfigured at run-time while the rest of the design remains active. DPR provides the flexibility to change a portion of the hardware components present in the design by reconfiguring it to another mode of operation. This is achieved by reusing the hardware resources present on the FPGA without the need to halt the rest of the system. The idea can be extended for configuring the Debugging System (DS) to the DUT through DPR at runtime. Consequently, the process of repeating the FPGA design flow in order to instantiate the DS to the DUT and frequent re-implementation of the complete system on the FPGA is not required. Moreover, the hardware resources utilized by the DS can be reused for other Reconfigurable Modules (RM) when debugging is not required as shown in Figure 3.11. An added advantage is that as the locked static partition is isolated from the dynamic partition, the timing behavior of the design is not affected.

Xilinx DPR design flow [195] is utilized for the proposed debugging solution. As required by the DPR design flow, the system is partitioned into a static and a Reconfigurable Region (RR). In our case, the DUT is present in the static region which will not be changed during runtime. The DS will be partitioned on the RR. The same RR can also be used by another RM when the debugging is over. The HDL files of the DUT and the DS are required for the DPR design flow. Floorplanning is performed for ensuring efficient hardware utilization. The DPR tool flow outputs partial bitstream files for every RM of the system and a complete set of full bitstream files for every configuration mode. Time of reconfiguration t_{reconf} is the

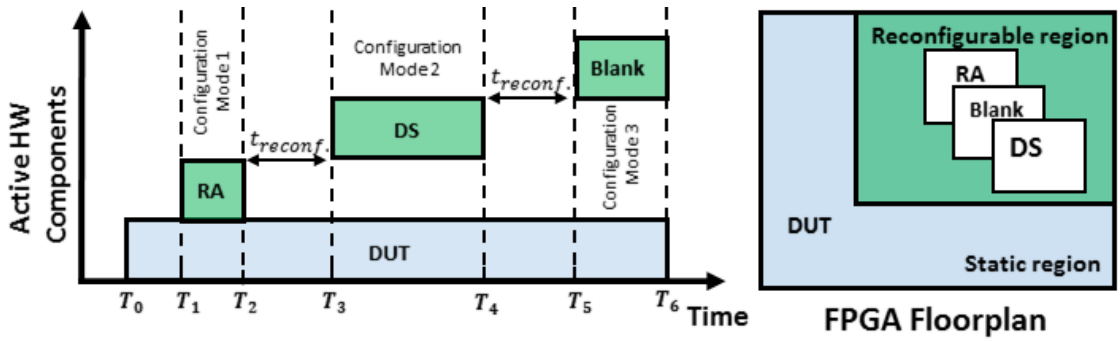


Figure 3.11: Using DPR to load the Debugging System (DS)

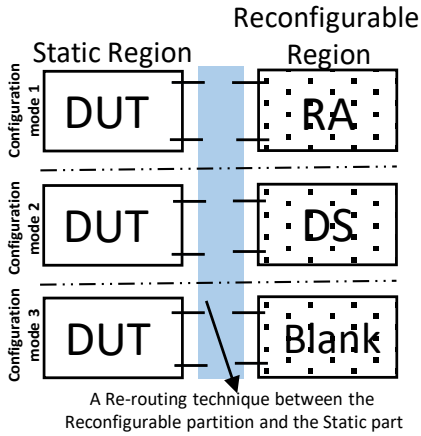


Figure 3.12: Routing between the static and RR

time consumed to switch to a new operation mode. As the time of reconfiguration t_{reconf} i.e. the time required for switching to a new operation mode is dependent on the size of the RR, it should be optimized for hosting the largest RM.

Figure 3.11 shows the proposed reconfigurable system with three RMs (DS, Blank and another Reconfigurable Application (RA)). DUT with one of the RMs constitutes a full configuration mode. Any RM is dynamically configured to the RR according to its time slot. Hence, the resources on the reconfigurable region allocated for the DS can be reused by other RMs when the DS is not activated. This requires the interconnections or the routing between the DUT present on the static partition and DS or another RM on the reconfigurable region to be changed according to the configuration mode. A reconfigurable rerouting technique should be adopted in order to maintain a validated data flow between the DUT and the RM as shown in Figure 3.12. In a previous work [196], a rerouting technique has been proposed which can also be utilized for reconfiguring the interconnections between the static region and reconfigurable region at runtime.

3.4.2 DPR-based Debugging Methodology

The proposed dynamic and partial reconfigurable methodology can be applied to any debugging system such as DSAS. This helps in solving problems such as recompilation time, resources availability and timing behavior. In this approach, the DUT is present in the static partition and the DSAS debugging system is configured as the reconfigurable part. The DUT can continue its normal operation if debugging is not required. However, when the debugging system is required, it is dynamically configured to the DUT using a partial bitstream. Then, the debugging system clocks the DUT present on the static partition and logs the debugging data to the trace buffers. When the trace buffers are full, the debugging system stops the clock to avoid any data loss. During the intermediate period, the debugging system transmits the data to external memory and once finished it starts clocking the DUT again. Hence, the methodology results in a continuous, lossless debugging stream with an effectively unlimited debug window. Moreover, the re-compilation of

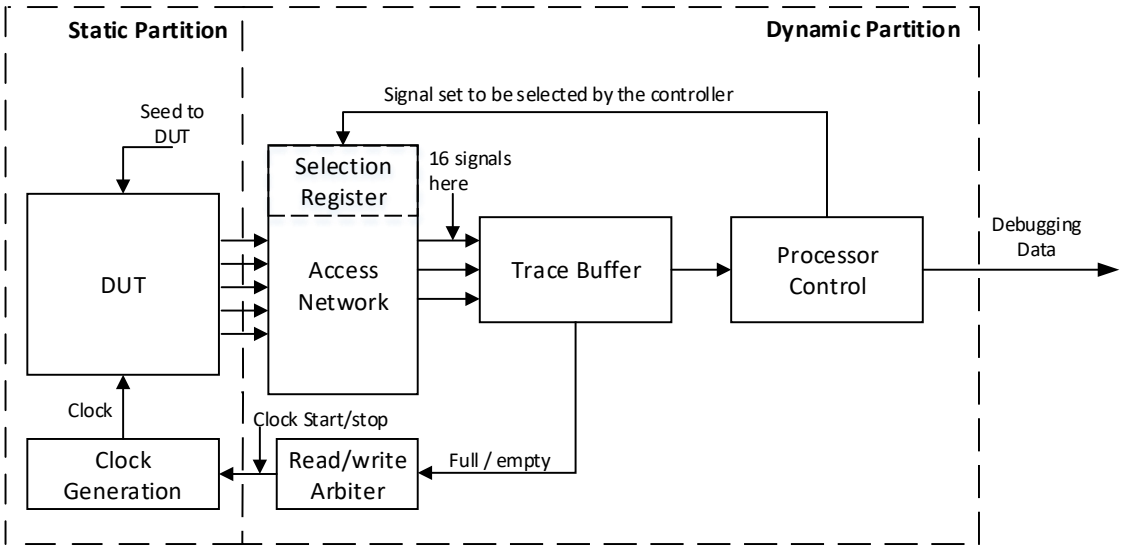


Figure 3.13: Debugging methodology

the design is no more required since the debugging system is configured to the DUT through a partial bitstream. The debugging data is sent to the terminal using an Ethernet interface or a UART which is then saved in a log file. A block diagram of the proposed debugging approach is shown in Figure 3.13.

The proposed technique offers many benefits, such as the debugging of embedded designs due to lossless debugging data. Due to the utilization of DPR for debugging; frequent recompilation of the design is not required. FPGA resources can be re-utilized for other applications when debugging is not required. Another benefit is the preservation of timing closure when the debugging system is removed from the design.

3.5 Experimentation and Results

The debugging methodology proposed in the preceding sections has been tested on the Digilent Zedboard [197], which has an XC7Z020-484 FPGA. Xilinx Vivado 2017.1 was used for the design process which was carried out on Intel Core i7-6700 CPU running at 3.4GHz and having 16GB of RAM.

3.5.1 Cycle-accurate Debugging with Replay

In order to explore different aspects of the proposed debugging solution, it has been tested with two different use cases: an obstacle avoidance system for a robotic vehicle and an image processing application. Before discussing the results, it seems beneficial to highlight that the debugging system comprises of hardware which could also malfunction. Hence, any debugging system malfunction must be ruled out before commencing any debugging session. Several researchers have addressed the problem and suggested to utilize Triple Module Redundancy (TMR) [198] to rule out any malfunction with the hardware itself. However, for the sake of this research work we have assumed that the debugging system is error-free and is qualified to debug the DUT.

3.5.1.1 Obstacle Avoidance System for a Robotic Vehicle as Use Case

The platform utilized for this research work is a 4 wheeled skid steer robot. Each wheel can be controlled independently which results in a highly controllable platform. Incremental encoders attached to each motor and eight ultrasonic sensors for closed-loop operation result in precise localization and environment sensing. Utilizing the vehicle information, the control module calculates the control signal required to maintain safe distance from the upcoming obstacle. The control signal is then transmitted to the actuation mechanism to divert the vehicle to avoid the obstacle. This results in re-shaping the path of the robot to avoid the obstacles in the traversed path.

During the course, the robot rotates at right angles to avoid the obstacle as shown in Figure 3.14. The block diagram of the technique (Figure 3.15) shows that data from the ultrasonic sensors and encoders are sent to the controller unit to perceive the environment and compute the corresponding control signals to avoid the obstacle. Then, the linear and rotational velocity from the controller are transformed into the right and left wheel velocity by using the odometry model of the robot. The Obstacle Avoidance (OA) algorithm uses this information along with the data from the sensors to localize the robot. If the ultrasonic sensors do not sense any obstacle, the robot continues to move in its designated path. However, once an obstacle is sensed and the threshold distance between the robot and obstacle is reached, the robot stops at a fixed threshold distance from the obstacle. Then, it calculates the width of the obstacle through triangulation. The robot compares the distance from its current position to the right (Option A) and to the left (Option B) corners of the object to decide the shortest path for avoiding the obstacle and follows the path until it reaches the destination.

Encoder and IMU sensors are used to determine the robot's position and orientation. However, these sensors may not give the exact position and orientation due to sensor errors. Therefore, Extended Kalman Filter (EKF) is used to apply the sensor fusion technique to reduce the effect of such errors and better estimation of the optimal position and orientation of the robot.

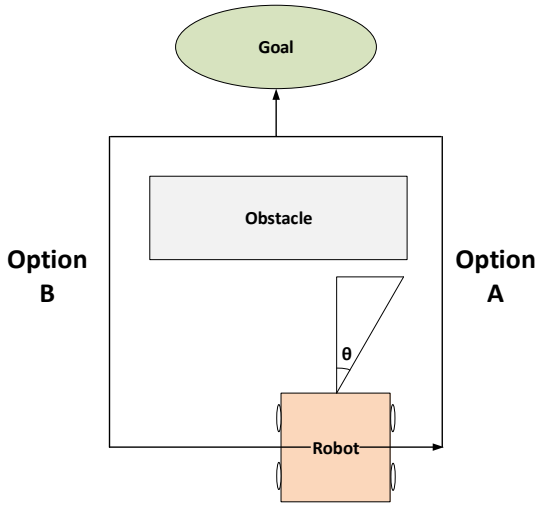


Figure 3.14: Obstacle avoidance technique

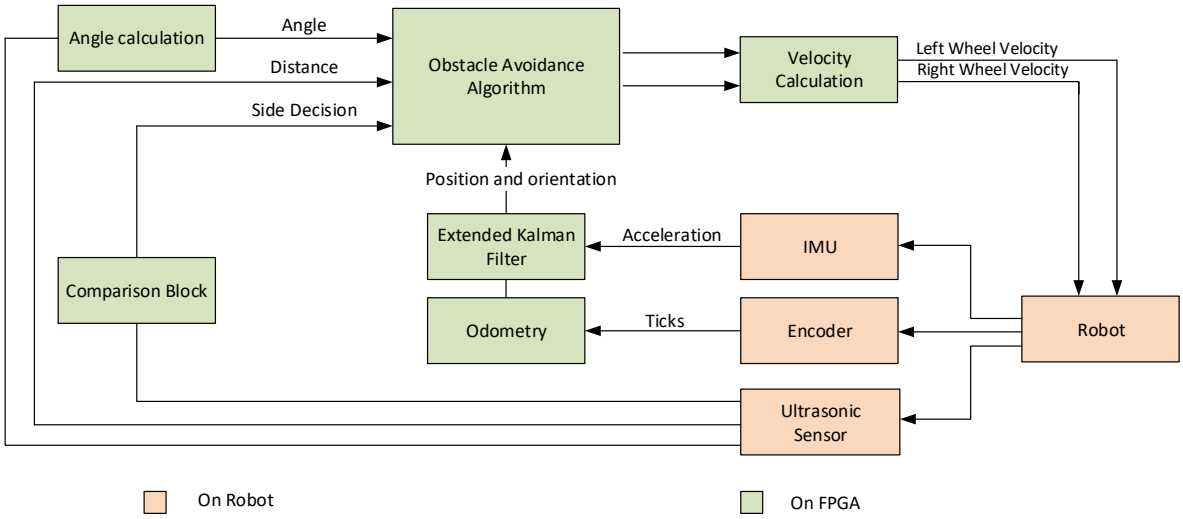


Figure 3.15: Block diagram of obstacle avoidance system

Integration between EKF and the OA algorithm is implemented as shown in Figure 3.15. Data from an accelerometer, magnetometer and gyroscope is fused using EKF to have an optimal estimation of the position and orientation of the robot. Then, these estimated values are used to avoid the obstacle.

3.5.1.2 Debugging of an image processing system

The second use case is a Gaussian filter based image processing system as shown in Figure 3.16. The filter has a window generator in which the image width, height and size of input pixel data can be specified. For the current experimentation, input image comprises of 1000 x 1700 pixels with each pixel having 8-bits. The second stage of the filter is a Gaussian 7 x 7 kernel. Output of Gaussian kernel is a 16-bit image pixel. Both of these modules are designed in VHDL. After verification and qualification, the Intellectual Property (IP) core is exported to Xilinx Vivado for its use as DUT.

3.5.1.3 Synthesis Time

When the proposed debugging system is synthesized besides the DUT, it needs more time for synthesis than the DUT alone. However, it is worth mentioning that synthesis requires more time even when ILA core is used for debugging the DUT. In order to explore the average synthesis time required for DSAS and

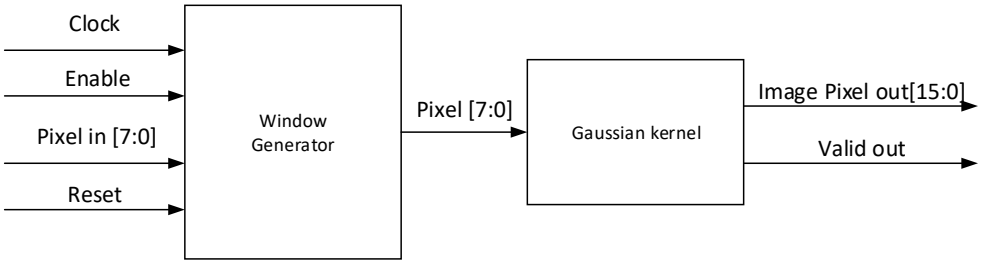


Figure 3.16: Gaussian filter

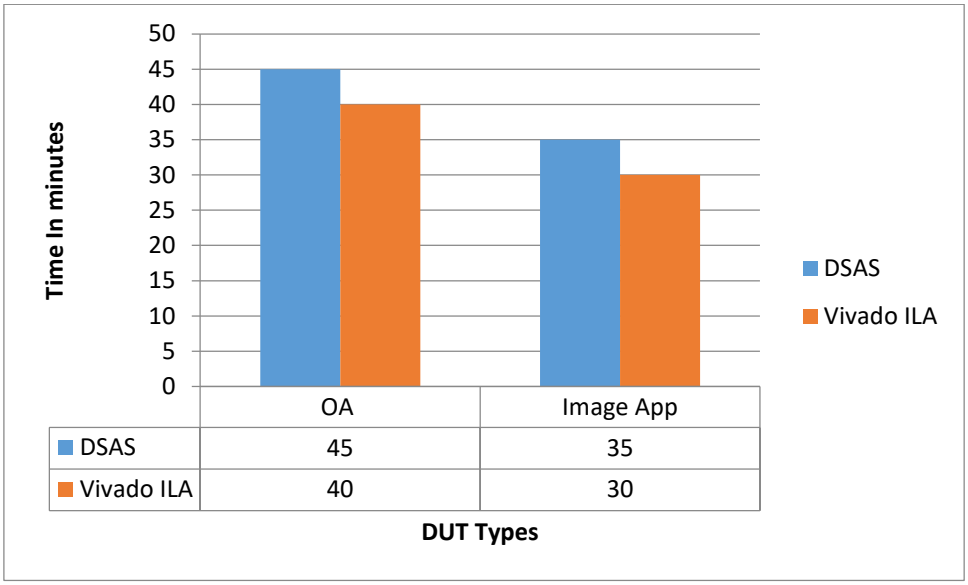


Figure 3.17: Average synthesis time (in Minutes)

ILA, we utilized the OA system and a Gaussian-based image processing application. Figure 3.17 shows the difference in synthesis time between the Xilinx ILA-based debugging design and the presented DSAS approach. The negligible difference may be due to more hardware resources required by the DSAS as compared to ILA. The advantage of the presented DSAS approach is the availability of lossless trace of debugging data.

3.5.1.4 Lossless Data Acquisition

Figure 3.18 shows the debugging data of our Gaussian image processing DUT plotted in MATLAB. The operating frequency of the design is 100 MHz which results in a clock cycle of 10 ns. Hence about 140000 samples have been acquired (1400000 ns/10 ns). Image input "*pixel in*" data entered the image processing module. Based upon this input data, corresponding filtered output *"Img out"* is presented in the second plot. As can be noticed, *"Valid out"* turns to 1 after ½*(6w+6) i.e. 3003 samples (where *w=1000* is the image width) signifying that a valid output is available. It is worth highlighting that only limited number of samples can be acquired when debugging with ILA. In order to get another trace, ILA core requires to be triggered again. But the DUT is running resulting in data loss between successive triggers. Furthermore, the maximum depth of trace buffer allowed for Xilinx ILA is 131272. Hence, in the present case, ILA would not have been able to acquire the complete data set. On the contrary, this is not the case with DSAS because the DUT is stopped during the data transfer phase. The complete lossless dataset can therefore be acquired for debugging. This completeness of data acquired through the proposed approach makes debugging of the designs faster and easier.

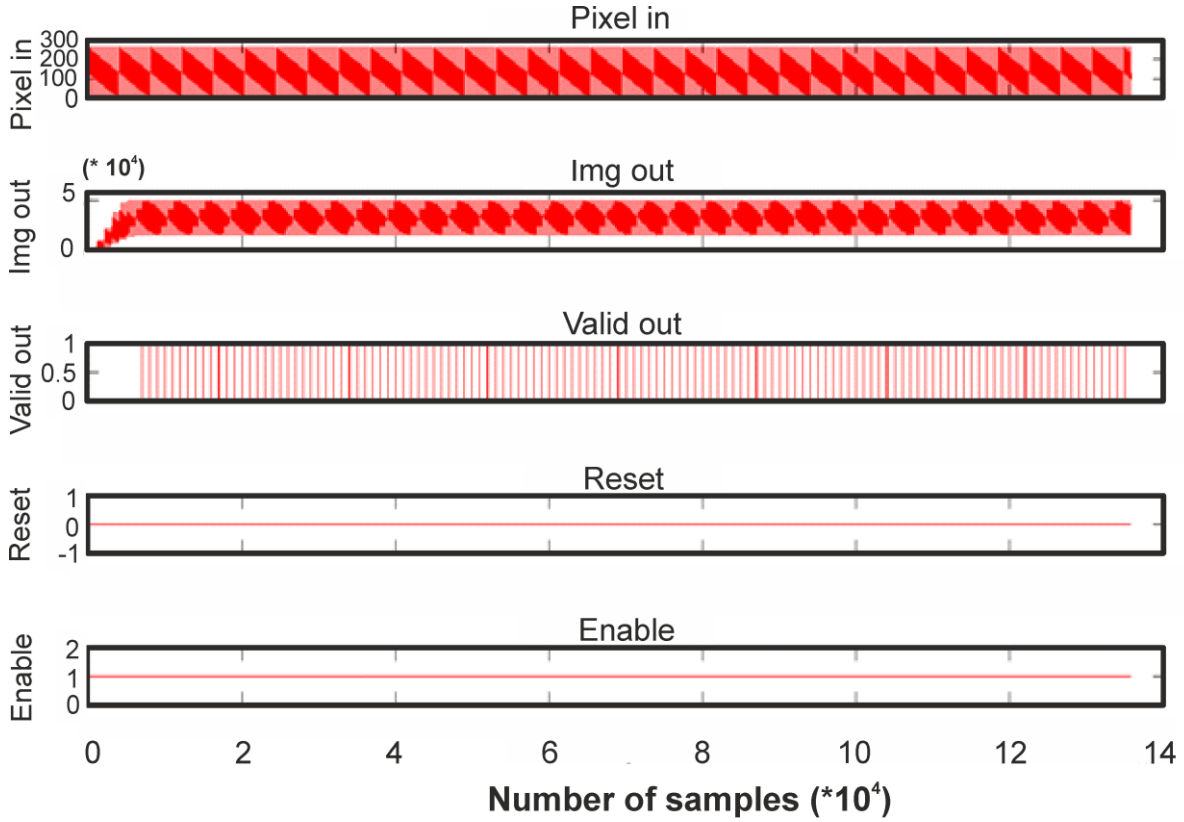


Figure 3.18: DSAS waveform

The main reason of faster debugging using DSAS is that lossless debugging trace (about 140000 samples shown in Figure 3.18 but not limited) can help in visualization of the complete set of events against lossy trace by ILA. Consequently, debugging using DSAS-based approach has resulted in 30% less time on average as compared to the ILA-based approach [Khan1]. However, the presented data is user-specific since it depends upon the debugging expertise of the user. Still, it highlights the benefits offered by the DSAS approach.

Another point worth mentioning is that debugging through DSAS is cost-effective because of the use of open-source waveform viewer like GTKwave hence, removing the dependency to use proprietary software.

3.5.1.5 Debugging Results

We used the proposed DSAS for debugging the obstacle avoidance system presented in section 3.5.1.1 As mentioned in [199], the most important parameters for robot localization are the x and y coordinates in a global frame of reference and its change in orientation angle. Hence, we used the X and Y coordinates of the robot to plot Figure 3.19. As can be seen, in the absence of any obstacle, the robot continued to move in its designated path. However, when the ultrasonic sensors detected an obstacle and the predefined threshold distance between the robot and obstacle was reached, the robot stopped and then calculated the width of the obstacle through triangulation. It then decided to avoid the obstacle by following the shorter

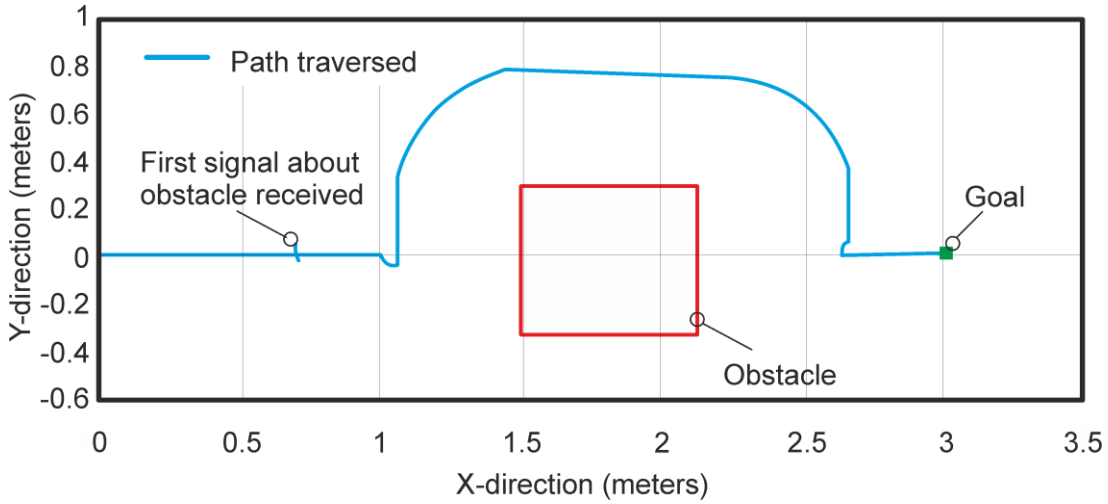


Figure 3.19: Movement in the X, Y coordinates

side. The robot then rotated 90 degrees to avoid the obstacle and continued the process until it reached the other side of the obstacle. Then it continued following its original path.

The orientation of the robot is plotted in Figure 3.20. We calculated the orientation of the robot by orientation formula ($\theta = tan^{-1}(\frac{y}{x})$) to demonstrate the usage of the proposed debugging system. The lossless tracing data in x and y coordinates accumulated by the proposed debugging system is used to plot the orientation in radians.

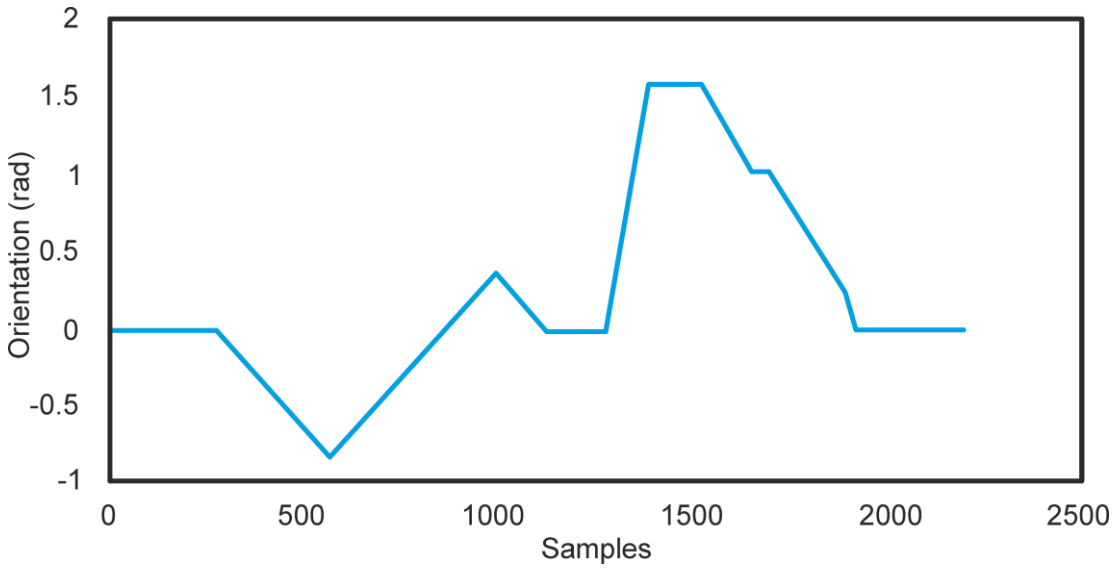
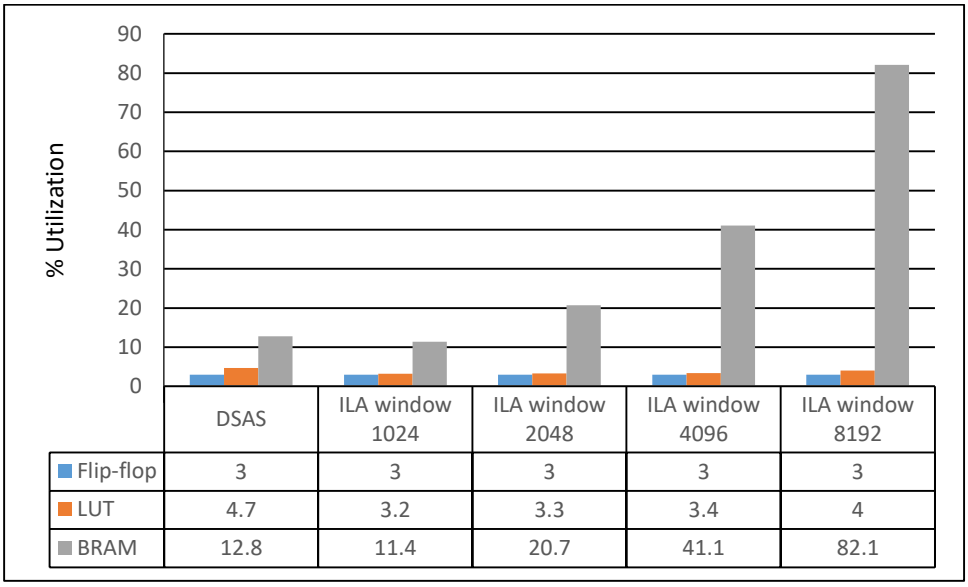


Figure 3.20: Orientation in radians

3.5.1.6 Resource Utilization

Resource utilization of the proposed DSAS system has been compared to the Xilinx ILA in Figure 3.21. The ILA core is instantiated to the obstacle avoidance system and synthesized with debug window (data depth) of 1024, 2048, 4096 and 8192. 16 signals having maximum data width of 32 bits ($\approx 16 * 32 = 512\ bits$) are monitored. It can be noticed that the resource utilization of the presented debugging approach is much less as compared to the Xilinx ILA. For the window size of 1024, the resource consumption of the presented solution is comparable. However, in the case of ILA, increasing the window size results in an increase in resource utilization. Implementation failed when the windows size reached 16384 because ILA requires more hardware resources (RAMB36) than are available on the Zedboard. This rise in resources is due to the different debugging methodologies. ILA stores debugging data on the FPGA in RAM blocks. On the contrary, DSAS approach transfers data to the external memory, consequently requiring only minimal resources.



	DSAS	ILA window 1024	ILA window 2048	ILA window 4096	ILA window 8192
Flip-flop	3	3	3	3	3
LUT	4.7	3.2	3.3	3.4	4
BRAM	12.8	11.4	20.7	41.1	82.1

Figure 3.21: Resource utilization

3.5.2 Access Network

In this portion of the results section, resource utilization of two types of access network namely the gate-based and the multiplexer-based will be discussed. It is iterated that the results are based upon implementation on the Zedboard without incorporating any DUT. The number of outputs are fixed at 16.

3.5.2.1 Gate-based Access Network Resource Utilization

The resources utilization for the gate-based approach is plotted in Figure 3.22. The percentage of resource utilization is shown in Figure 3.23.

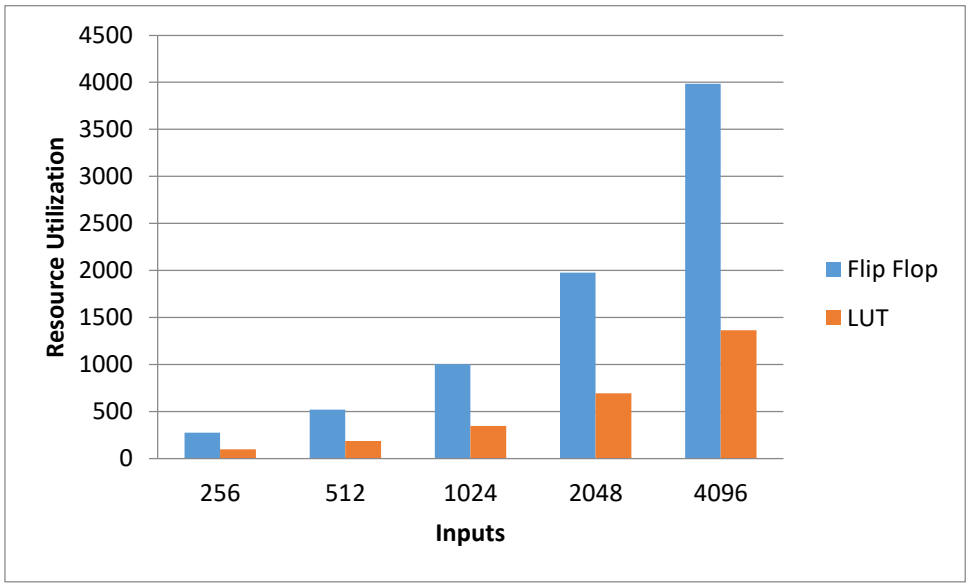


Figure 3.22: Resource utilization on a Zedboard for gate-based approach

Noticeably, the resources required for this approach are quite small. Even with 4096 inputs and 16 outputs, the utilization of LUTs and flip-flops is 2.5% and 4% respectively. With fewer inputs, resource utilization is even less. However, this technique has one drawback that the selection within the input signals group is not possible. If signal selection within the group is not required, the approach can be utilized.

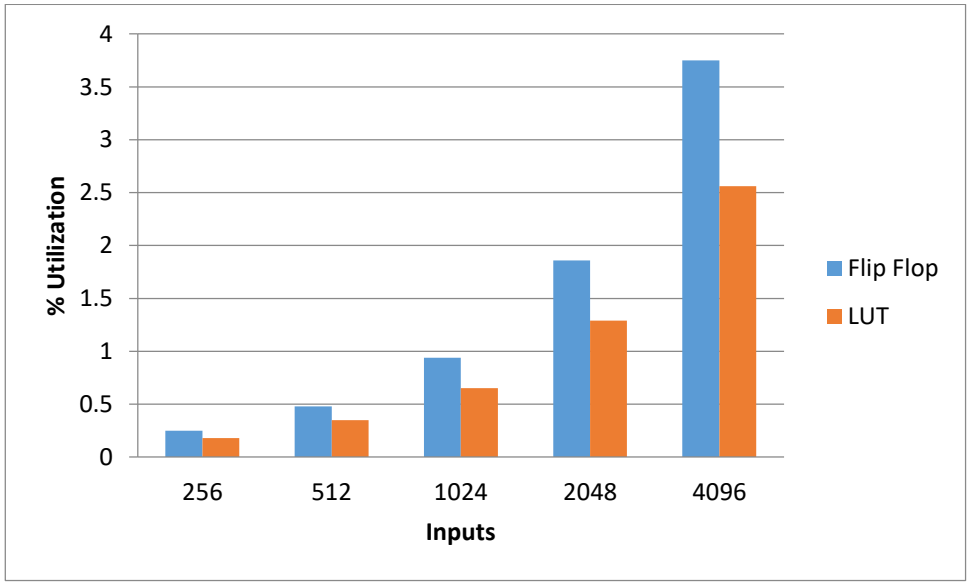


Figure 3.23: Percentage resource utilization on a Zedboard for the gate-based approach

3.5.2.2 Multiplexer-based Access Network Resource Utilization

source utilization for the multiplexer-based access network is shown in Figure 3.24 and percentage of utilized resources is presented in Figure 3.25.

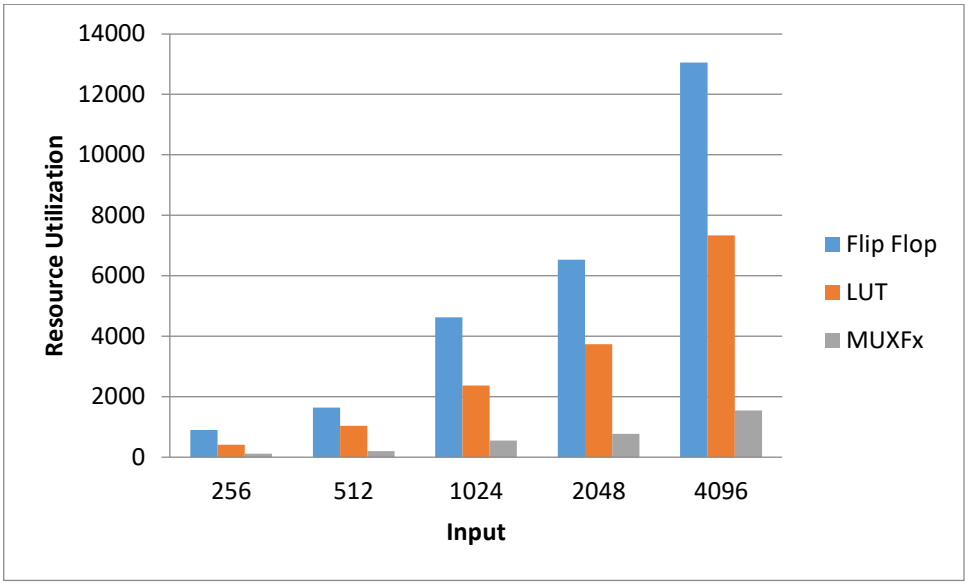


Figure 3.24: Resource utilization on a Zedboard for the multiplexer-based approach

An access network with 4096 inputs and 16 outputs consumes 12% of the resources of a Zedboard. The resource utilization is even less for networks with fewer inputs. The main benefit offered by this approach is that, instead of groups as proposed in the previous approach, any subset of signals can also be selected. Hence, for cases where redundant resources are available, this approach is suggested due to its flexibility.

As evident from the results, both approaches i.e. the gate-based and the multiplexer-based approach have their merits and demerits. Depending upon the available resources and the requirements, a suitable approach

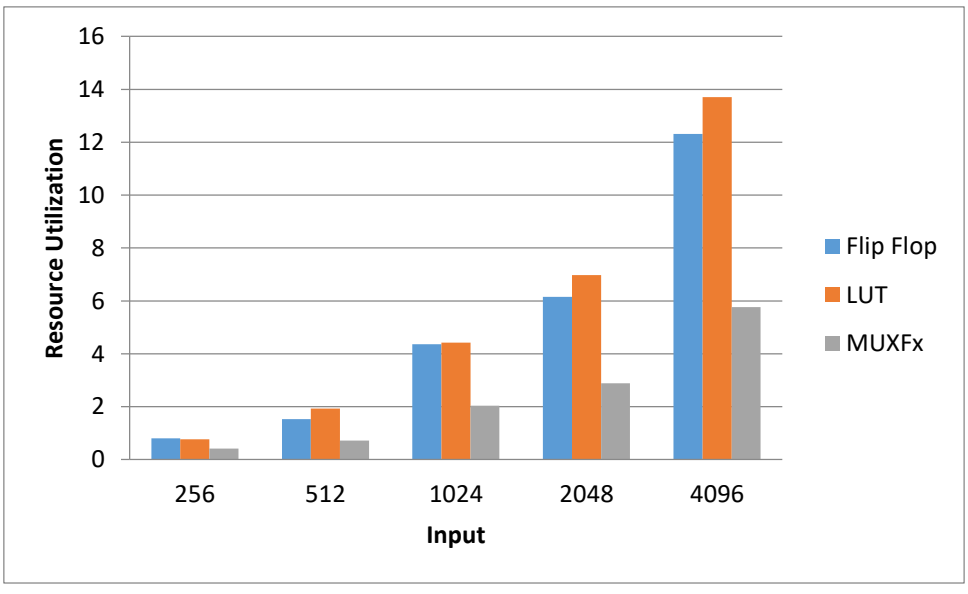


Figure 3.25: Percentage resource utilization on a Zedboard for the multiplexer-based approach

can be selected. The resulting design can then be controlled by a processor to achieve the desired connectivity.

3.5.3 Data Compression

DSAS utilizing MSim-9 has been implemented on a Zedboard comprising the ARM Cortex-A9 processor running at 100MHz. We utilized the earlier mentioned DUTs such as Gaussian filter based image processing system and the OA system to generate the trace data. The Msim-9 compression methodology was then used to compress the debugging trace. The results are displayed in Table 3-2.

Table 3-2: CR results for different trace data

S. No.	Data Stream	CR
1	Gaussian	3.27
2	OA	NA*

* Not Applicable

It is evident from Table 3-2 that the compression is highly dependent upon the incoming data stream. The CR was calculated using Eq 2.1. For the image processing DUT, the % CR was found out to be 70%. The high CR was because of the presence of 8-bit input and 16-bit output trace data. When such data were compressed using MSim-9, several debug traces could be interleaved in one word compressed data resulting in better compression. However, for the OA use case, the compression of the data was not possible due to the presence of the signed integers. It has been discovered that the compression methodology is suitable for trace data having unsigned integers, fixed-point integers etc. But when the data lacks the earlier mentioned attributes, compression is not useful.

In order to compare the MSim-9 compression methodology with other available alternatives, we have performed trace compression for the earlier mentioned data streams using LZW compression method illustrated in Figure 3.26. LZW compression methodology starts with a dictionary of 256 characters (utilizing 8 bits) and using standard character set. Then it reads successive 8-bit data and encodes it with the number that represents its index in the dictionary. Every time it encounters a new string, it adds the strings to the dictionary. Whenever it encounters a substring which is already in the dictionary, it just reads a new character and concatenates it with the current string to create a new entry in the dictionary. The dictionary is limited to 4096 entries to prevent it from consuming too much memory space.

Percentage compression results for our defined data streams are given in Figure 3.27. It can be noticed that the percentage compression is highly dependent upon the incoming data. The CR for MSim-9 is greater than LZW for Gaussian-based image processing use case but less for the OA use case.

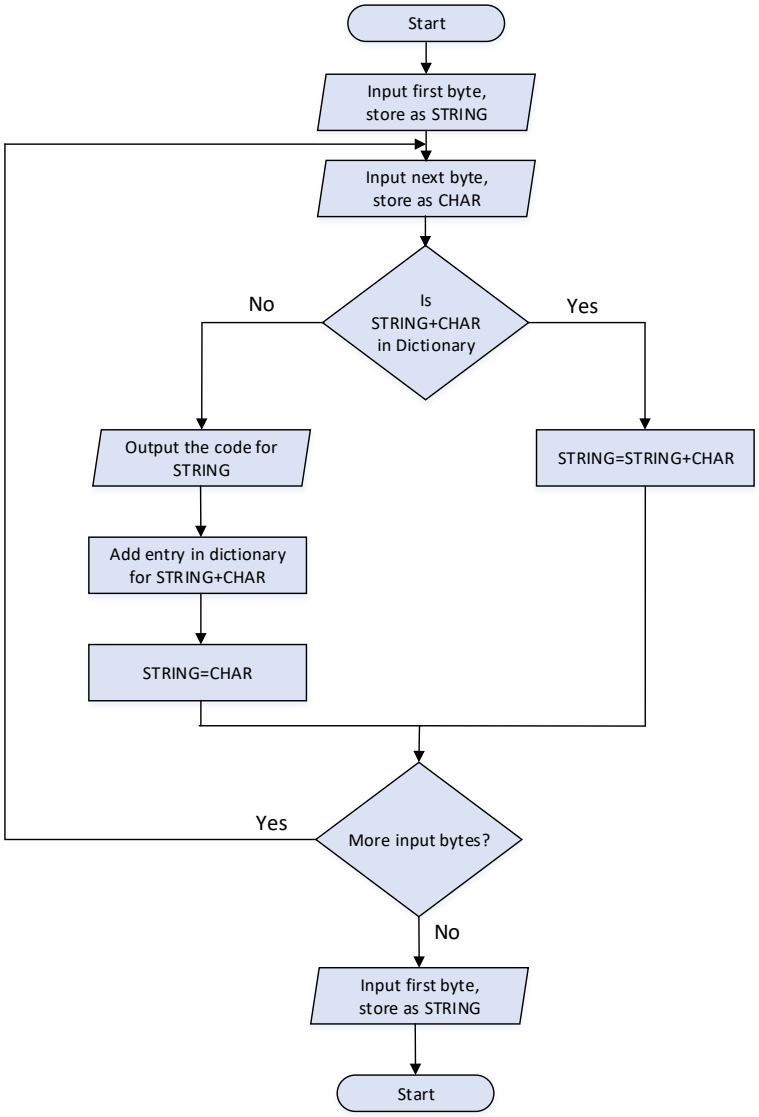


Figure 3.26: LZW algorithm [236]

In order to compare the execution speed of the two compression methodologies, the two compression methodologies were implemented on the Zedboard utilizing ARM Cortex-A9 processor. Table 3-3 shows the time taken by the two compression techniques to compress 1 KB of data. MSim-9 took about 1000µs to compress the data while LZW took about 3 times more time to compress the same amount of data. LZW takes more time for compression because LZW algorithm requires frequent dictionary consultation. However, MSim-9 does not require dictionary consultation and is not compute-intensive. Hence, it is about 3 times faster than LZW. Furthermore, as the data compression is implemented for speeding up the transmission of debugging data through Ethernet for resolving the data transmission bottleneck, MSim-9 is better suited because it consumes less time for data compression.

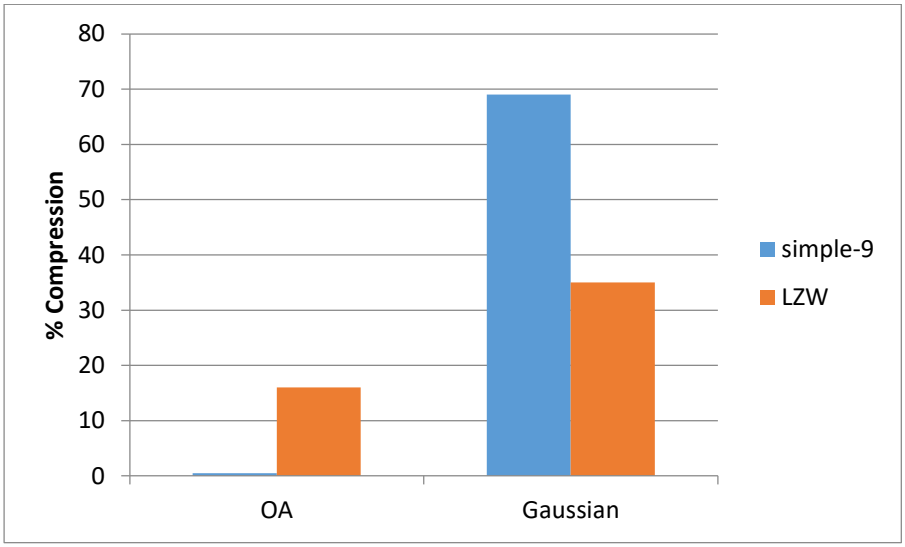


Figure 3.27: Percentage compression

The MSim-9 can be recommended for software-based debugging data compression for certain data types. The methodology provides speedup for data transmission without utilizing scarce FPGA resources and without any area overhead. For other data types, any other alternative such as LZW can be utilized.

Table 3-3: Timing comparison

S. No.	Methodology	Clock cycles	Time taken
1	MSim-9	619939	929.5µs
2	LZW	1841748	2761.24 µus

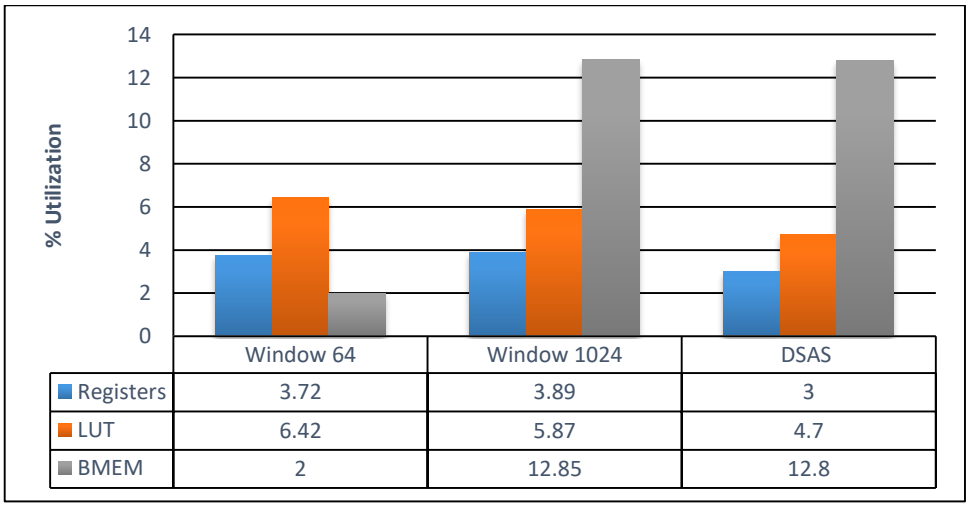
3.5.4 Incremental Compilation using Dynamic Partial Reconfiguration

As highlighted in section 3.4, the main purpose of incremental compilation methodology is to reduce the re-compilation time, better utilization of the hardware resources and addressing the timing closure due to debugging circuitry. During investigation of re-compilation time, we compared the traditional debugging flow with the proposed DPR-based debugging flow. The test specifications were the same as mentioned in section 3.4.3. The difference in synthesis time between the two methodologies was found to be negligible. When the debugging system was synthesized as a reconfigurable module, the time taken by the process was about 23 minutes as compared to the traditional flow (without DPR) which consumed 17 minutes. However, the major benefit of the proposed methodology is the capability of partial reconfiguration. The DPR-based debugging system is flexible because the debugging circuitry can be configured to the DUT at run-time, obviating the need for repeating the design flow as required in the traditional debug flow. Hence, the design time required to instantiate the debug circuitry with the DUT can be reduced. Moreover, the re-compilation

of the design is needed whenever the signal set is required to be changed in the traditional design flow. In contrast, re-compilation of the design is no longer required for the presented solution due to the concentration network. Furthermore, the hardware allocated to the debugging circuitry can be re-used by other applications when debugging is no more required.

3.5.4.1 Resource Utilization

Resource utilization for the presented debugging system is presented in Figure 3.28. The debugging system has been synthesized with a trace window of 64 and 1024. 16 signals are debugged, each having a maximum data width of 32 bits. It is evident that the presented debugging system consumes more resource with the increase in the trace window because additional BRAM blocks are needed as trace buffers. Although, the trace window size does not have any impact on the quality of the debugging data because the presented debugging methodology ensures the completeness of the data. Still, debugging with a larger trace window is beneficial because more data is acquired in one cycle and hence it is faster. We have also compared the resource utilization with our earlier work (DSAS with trace window of 64) [Khan1]. As can be seen, we have been able to significantly reduce the BRAM utilization (10%) at the cost of a minor increase in registers and LUTs utilization. The main reason for better resource utilization is that in the previous work we utilized BRAMs for FIFO synthesis. However, for the current work, we utilized FPGA logic elements which were found to be more hardware efficient. It is worth noting that the resource utilization for "Window 1024" is nearly equal to DSAS [Khan1] (having a trace window of 64). Thus, the debugging data can be acquired quickly by using the same hardware resources as the earlier version. As suggested by Xu et al. [200], it is generally recommended to keep the debugging hardware cost lower than 10% of the original design. The proposed design meets this criterion. Furthermore, in proposed solution, the debugging system is configured only if required. Otherwise, the hardware resources not in use of the debugging system can be utilized by any other reconfigurable modules. Another feature is that, unlike the traditional design flow, the proposed DPR-based debugging solution is not an integral part of the static design. Hence, after the



	Window 64	Window 1024	DSAS
Registers	3.72	3.89	3
LUT	6.42	5.87	4.7
BMEM	2	12.85	12.8

Figure 3.28: Resource utilization

DUT present on the static partition has been debugged, the debugging system can be disconnected from the DUT without affecting its routing or the timing closure; because the partitions are totally isolated at internal routing level. On the contrary, clock error may occur due to re-routing of the design when the ILA is removed from the DUT in traditional design flow.

3.5.4.2 Power Utilization

Another benefit offered by the technique is its low power consumption. Clock management during data transmission imitates clock gating which helps in reducing the dynamic power consumption. During experimentation, it was noticed that our debugging methodology also results in power saving as shown in Table 3-4.

The static power consumed by the Zedboard (including PS+PL) without configuring the FPGA with the DUT bitstream was found as 2.4W. Through current sense jumper J21, the power consumed by the DUT was measured to be about 3.3W. The DUT after configuring the DPR-based debugging methodology consumed 3.36W. Hence, the power consumed by the proposed debugging system was found to be about 0.06W (3.36W-3.3W). When we investigated the reason for such low power consumed by the proposed solution, it was revealed that the DSAS debugging methodology imitates clock gating which is known for reduction in dynamic power consumption [201]. In order to investigate the effect of clock management performed by our debugging methodology, we removed our debugging system from the DUT. We then gated the clock of the DUT at 1000 Hz to observe its effect. It was observed that DUT with modified clocking consumed 3.24W. Hence, the clock gating resulted in sparing 0.06W (3.30W-3.24W).

Table 3-4: Power utilization

	Design Description	**Total Consumed Power (Including PS+PL)**
1	DUT	3.3W
2	DUT with DPR-based debug methodology	3.36W
3	DUT with clock gating	3.24W

3.5.4.3 Deployment Time

DPR consumed 260ms for employing our debugging system to the DUT where t_{reconf} is the time of reconfiguration required for reconfiguring the FPGA (≈2MB) at run-time through the JTAG configuration mode. On the contrary, the traditional flow may need minutes to hours depending upon the design complexity.

3.6 Summary

When a hardware problem is encountered which requires debugging, generally traditional debugging approaches are utilized. For uncomplicated designs, traditional debugging solutions may be enough. However, due to unavailability of lossless data trace, debugging becomes cumbersome for complex designs. In such situations, the DSAS approach proves to be an alternative.

Before transmission, the debugging data is initially saved on the FPGA hence a trace buffer is required. The trace buffer size is adaptable and currently set to 4KB. The proposed debugging approach provides a lightweight alternative requiring fewer resources than other published work in the field. Moreover, captured data synchronization with the terminal is performed automatically eliminating user interaction. The data is logged to text files, subsequently converted to VCD format and can be presented and analyzed by any open-source waveform viewer. The solution is, therefore, cost-effective.

So far we have discussed the features and the contributions introduced by the proposed methodology. However, the work has few limitations. The main limitation of the approach is that sometimes the modules need to be debugged while running (capturing external streaming data such as an external HDMI input). The methodology in its current state cannot undertake such problem. This issue can be resolved through the inclusion of synchronizers. The debugging system disables the synchronizer between the external streaming device and the DUT during the debugging data transmission phase and will be enabled once the DUT is ready to receive the data. However, for external devices which do not follow a streaming protocol, the proposed solution may not work.

A similar problem is faced when the DUT cannot be halted e.g. a live network controller where packets are dropped if the DUT is stopped. Such situation is quite complicated. A solution could be to use the event-based debugging methodology. Another solution could be to use dedicated I/O pins for streaming without buffering the data as is done by some emulation systems. But even such emulation system is limited to spare I/O pins which may not be available for some designs.

During the debugging process, changing the signal is quite common. Traditionally, whenever the signal set is changed, a recompilation of the design is required. Access networks can be utilized to enhance the debugging system's access to the input nodes. This reduces the frequent need to re-synthesize the design just for debugging. Two different methodologies for generating an access network are proposed. The resource utilization for both approaches has also been reported. It was highlighted that the gate-based access network uses fewer resources but lacks in flexibility. On the other hand, the multiplexer-based access network consumes more resources but offers additional flexibility. The main limitation of the access network is that it requires hardware resources for implementation. The multiplexer-based access network being more flexible requires extra resources. Moreover, the resource requirement keeps on increasing with the increase in dimension of the access network. Hence, for resource deficient designs, inclusion of an access network in a design can be problematic.

Another issue discussed in this chapter is the data transmission bottleneck. A novel debugging methodology is proposed which utilizes software-based trace compression to address the data transmission bottleneck. The results indicate that the technique performs better than other alternatives. Furthermore, the proposed methodology results in no extra hardware overhead as the data compression has been carried out on the processor side. However, software processing is required to add data to packets. This processing consumes time resources. Furthermore, as already covered in the results, the Msim-9 algorithm used in this research work may not be able to compress certain data types.

Debugging requires instantiating the DUT with the debugging system and identifying a set of potential error-prone signals for debugging. Then the PAR process is conducted. The process is iterated with the changing signal set until the verification of DUT. These processes need hardware and time resources. Furthermore, after verification, the debug core is required to be removed from the design. This may affect the timing closure of the design requiring further analysis. Such issues were addressed by debugging through DPR-based incremental insertion. The proposed methodology configures the debugging system to the design at runtime which eliminates the need to go through the time-consuming PAR process. Furthermore, the resources set aside for the debugging can be used by other applications when debugging is not required. However, one limitation of the proposed solution is that, upon design finalization, the blank bitstream needs to be part of the finalized design. Although blank bitstream does not result in dynamic power dissipation, however, it still results in static power dissipation which may not be negligible.

Chapter 4 Debugging Coverage

The most recent trends for FPGA based systems covered by the Wilson Group Research report shows that the number of embedded processors is constantly growing [27]. The report claims that 64% of the designs now comprise of embedded processors. It further claims that 43% of the designs now contain 2 or more processors. Hence, it is imperative to provide a solution for embedded processors. As the embedded processors contain software applications running on the hardware, their debugging requires a different technique which will be discussed in this chapter.

DSAS methodology presented in the previous chapter utilizes cycle-based synchronization by utilizing clock gating. However, large designs which require data transmission between different IPs, stopping the clock for an individual IP may result in data loss. Such loss can be prevented by inserting synchronizers which will be presented in this chapter. The proposed solution is directly applicable to multiprocessor systems. However, it has been normally suggested that the cycle level (or cycle-based) is not effective for multi-core debugging: event-driven debugging (such as the transaction-based) is considered more desirable [130][131][132]. An event-based debugging methodology based upon transactions will also be presented. Such methodology abstracts the view from cycle granularity to transaction granularity for reducing the debugging difficulty [202].

Debugging for multi-clock domain system remains a critical design area because of the increasing number of multi-clock domains. As stated in Wilson Group Research report, the number of asynchronous clocks is increasing with more than 50% design projects currently employing at least more than 2 clock domains [27]. Hence, a solution for multi-clock domain designs will also be proposed.

4.1 Embedded Processor Debugging

DSAS can be utilized for debugging any hardware design. However, due to its ability to generate a continuous stream of lossless data, the methodology is ideally suited for complex embedded microprocessors (Figure 4.1). Two different embedded microprocessors are utilized as use cases in order to validate the proposed methodology. The embedded processor is treated as a black box and all interfaces originating from the processor are continuously monitored. This results in providing a complete picture of the activities performed by the embedded processor. The two processors are detailed below.

1. The first embedded processor utilized as the use case is Xilinx Microblaze [15]. Microblaze is debugged by connecting its external interfaces to the debugging system. However, Microblaze

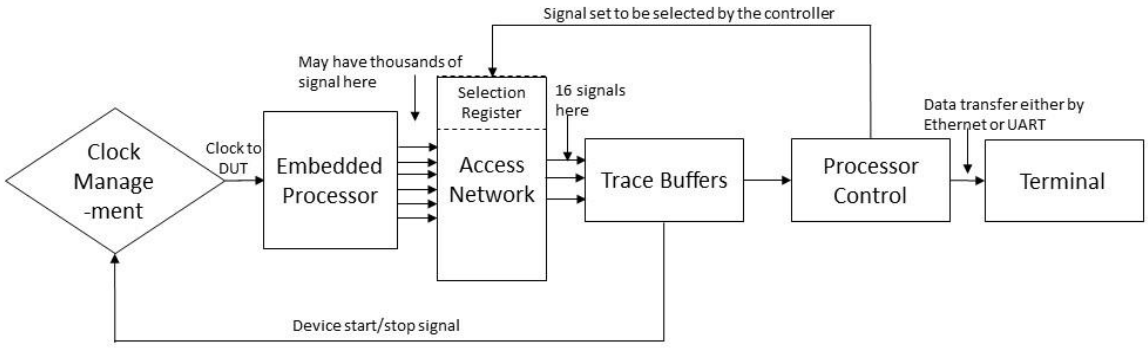


Figure 4.1: Embedded processor debugging

already has a dedicated debugging port (trace port) which can be used to retrieve the status of some internal registers which are not available on other interfaces. The proposed debugging system can be connected to this trace port for a continuous, lossless stream of data. Lauterbach has provided a debugging solution to debug Microblaze through the trace port [203]. It needs an external hardware module required to be connected to the trace port for acquiring trace data hence not cost-effective. By contrast, the proposed debugging system can be utilized to debug any interfaces (not limited to a trace port) without extra cost.

2. The second use case is an embedded processor based upon the RISC-V architecture, chosen to highlight the usability of the presented debugging system. The microprocessor (ORCA developed by Vectorblox) [204] is an open-source core utilizing RV32IM architecture. RISC-V cross compiler toolchain is also available to perform software compilation. The core has low hardware utilization and therefore, is suitable for small FPGAs [205]. However, the core does not have a dedicated debugging solution and is hard to debug. Our proposed debugging solution can be used for debugging of the core.

Black-box approach was utilized for debugging of the microprocessors. Hence, all the exposed interfaces (including AXI interfaces) are required to be monitored. The microprocessor fetches the instruction from the memory which is first decoded and then executed. The execution of the instruction results in either saving the data to the memory or using the data for processing the next instruction. In the first case, the data can be acquired by the debugging system when it is being written to memory. In the second case, after processing, the data will be saved to memory. The proposed methodology results in a continuous lossless data stream, hence, monitoring the interfaces results in the debugging of the microprocessors. The internal registers can also be debugged if they can be made visible to the debugging system.

Clock Management

Because of the data transfer bottleneck, data generation is faster than the data transmission. Hence, when the trace buffer is full, the microprocessor is required to be halted so that the data is sent to the terminal without data loss. The processor can be halted by writing to the Control and Status Register CSR (0x800).

However, we chose to halt the processor by managing the clock. This was accomplished through a custom-made clock manager which can stop the clocking of the processor once the trace buffers are full. Another solution is also available for Xilinx FPGAs. For stopping the clock, Xilinx clocking wizard IP [206] has a power-down option which can also be utilized. Although this option was provided for power gating, the same function can be used for debugging eliminating the need for any custom made IP. However, in the presence of any logic which gets reset if not clocked, the proposed design may not work. Hence, such logic must first be removed in order to get a continuous lossless stream of debugging data from the embedded processor.

4.2 Event-based Multiprocessor Debugging Methodology

This section presents our event-based debugging solution for NoC. The proposed debugger can be utilized for faults diagnosis and network analysis. The system is generic and can be easily adapted to a variety of NoCs. It also supports the analysis of NoC at different levels of abstraction. The debugging system can be used to control the DUT through an interactive computer application. This makes possible the problem detection along with data analysis such as log files, utilization reports and network visual illustration. Hence, the proposed NoC analysis solution results in the detection and correction of faults. The NoC debugger presented in this section was implemented to perform debugging on a NoC called RAR-NoC (Reconfigurable and Adaptive Routable NoC) [207]. In the following paragraph, RAR-NoC will be concisely explained to clarify the instantiation of the debugging system to the NoC and subsequent data monitoring.

RAR-NoC comprises of reconfigurable routers (Figure 4.2) which can be added or removed from the network at runtime. This allows adjusting the size and topology of the NoC for fulfilling the requirements. Furthermore, hotspots can be avoided by selecting an appropriate routing algorithm for individual message. This results in better division of traffic on the network channels resulting in increased data throughput. The routers utilize wormhole switching. The packets are transmitted as flits; header flit, payload flits and tail flit. The tail flit is required to mark the end of the packet and can also carry payload. On the other hand, the header flit includes information regarding the destination and the desired routing algorithm. Either west-first (WF) or XY routing algorithm can be chosen. The bit width of the flits can be configured at design time. However, the minimum bit width is 5-bits for holding the routing algorithm (1-bit) and the destination address (4-bits). The Local port of the router connects the Processing Element (PE) to the router, while

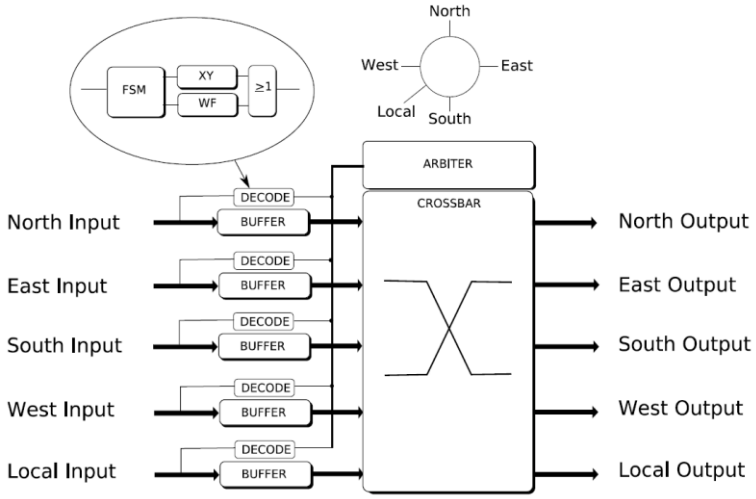


Figure 4.2: RAR-NoC router architecture [207]

North, East, South and West ports connect the router with other adjacent routers. The arbiter utilizes round robin algorithm to connect only one input port to one output port simultaneously. The latency of the routers when using WF algorithm is 2 cycles for header flits and 1 cycle for the remaining flits, while for XY algorithm, the latency is 1 cycle for all flits. The most important aspect of RAR-NoC is being deadlock-free. The utilized routing algorithms feature turn models which prevent building circles when used either individually or when the two algorithms are combined.

PE (either ARM or MicroBlaze) and the NoC are connected through Network interfaces (NI). Hence, the NI acts as a bridge between different PE interfaces; AXI for ARM and AXI-Stream for Microblaze, and the NoC. A DMA core is used to connect the ARM processor with the NoC. The DMA core receives data from the AXI Memory Mapped port of the processor, converts them into AXI Stream and sends them to the NI. Internal registers of the DMA are required to be configured previously for the DMA to work (Figure 4.3). The DMA is configured through the GP port of the ARM processor. In order to receive data at the ARM processor from the NoC, the same operation is performed in reverse.

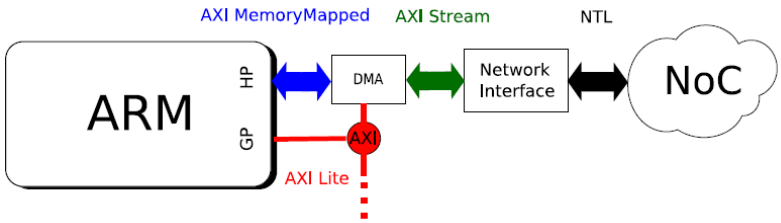


Figure 4.3: Developed network interface for connecting the ARM processor to the RAR-NoC [207]

In order to completely monitor the NoC, the debugging system (Figure 4.4) monitors all interconnections between the PEs and the routers and between the routers themselves. FIFOs are used for data collection which are continuously observed to prevent data overflow. When the FIFO occupancy reaches a certain threshold, defined at design time, the processor is interrupted to immediately perform a read operation to avoid data loss.

The data acquired is huge and increases proportionally with the number of routers and the number of endpoints. Hence, managing the complete set of data collected from all nodes simultaneously and transmitting it to the terminal results in a data transmission bottleneck. In order to cope with this bottleneck due to the generated data traffic in the network, the debugging mechanism was designed with two clock domains. The debugging system runs at a higher speed than the NoC itself; with debugging system running at 100 MHz and the NoC running at 10 MHz. Hence, the data is generated at 10 MHz but dispensed at 100MHz. This results in the faster dispensation of data to the terminal and permits a lossless data transmission.

The process undergoes the following steps:

1. The ARM processor establishes a communication link with the Terminal through Ethernet utilizing User Datagram Protocol (UDP) [208]. Once the link has been established, the processor initializes the debugging system and then waits for an interrupt from PL.

2. The debugging system starts logging the debugging data in the FIFOs. Once they are full, an interrupt is raised for the processor. Processor starts interrupt handling. First the processor initiates a burst read through the Direct Memory Access Controller (DMAC) seeking maximum bandwidth.

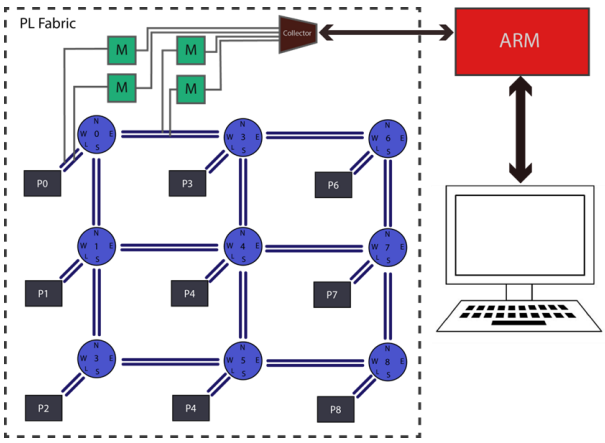


Figure 4.4: Network and debugging system diagram

3. The processor continuously transmits data to the terminal until all available data has been transmitted.

4. The data is received at the terminal and processed by a specifically designed computer application. The results are then displayed on the screen.

As illustrated in Figure 4.4, multiple monitors are required to debug each node. The number of monitors required for debugging a NoC can be calculated as given in *Eq. 4.1* and shown in Table 4-1, where r denotes the number of rows and c denotes the number of columns in the NoC:

$$f(r,c) = 6 * r * c - 2 * r - 2 * c \qquad (Eq. 4.1)$$

Where

$$r > 0, c > 0$$

Table 4-1: Number of monitors needed in debugging the system

No. of rows and columns	***f(r,c)***
2x2	16
3x3	42
4x4	80
5x5	130

In order to uniquely identify the source of debugging data during the decoding process, a unique signature is assigned to each monitor in the debugging system which is transmitted alongside the data. Figure 4.5 shows the format of the encoded debugging data before transmission through the monitor. Transactions are allocated unique IDs to avoid synchronization issues at the receiver. The transaction time besides the transaction ID is also encoded into the debugging data to identify the start or end of a transaction as shown in the figure. For a 2x2 network, 16 monitors are required. The quantity of the monitors increases with the increase in the size of the network. This in turn also increases the quantity of debugging data required to be collected and transmitted to the terminal. As illustrated in Figure 4.4, data from 4 monitors is combined in a collector through AXI stream interconnect and then logged to a FIFOs.

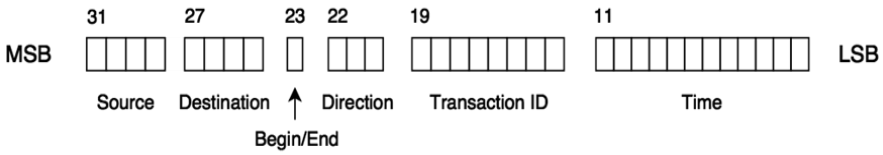


Figure 4.5: Debug data encoding description

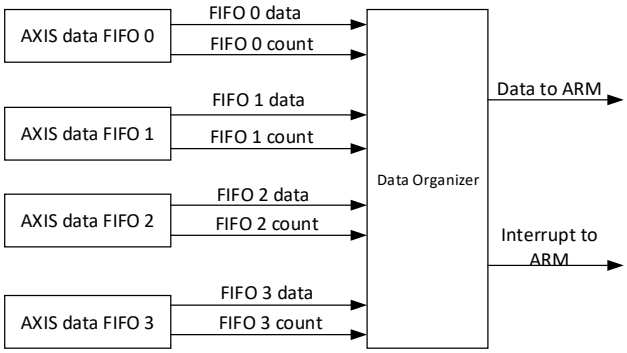


Figure 4.6: Data collection

As data from the monitors arrive simultaneously at the slave ports of the AXI stream interconnect for logging to the FIFO, a 32x4-bytes (where 4-bytes is the width of the debug data) data FIFO is also included in the design to prevent data loss during round-robin arbitration.

4.2.1 Data Collection

During the data collection process, debugging data is collected from individual FIFOs which is then organized as shown in Figure 4.6. The data is organized and the counter port of the FIFO is monitored continuously in the Data Organizer. As previously pointed out, the quantity of debugging data increases with the NoC size. Hence, for a 1x2 and 1x3 NoC, a single FIFO depth is 512; for 2x2 and 2x3 NoC, a single FIFO depth is 1024; for a 3x3 NoC, a single FIFO depth is 2048.

4.2.2 Data Organization

The Data Organizer is required to maintain the correct order of the debugging data. Ordering the data at the terminal is not time efficient keeping in view the huge quantity of the data. Data Organization at the hardware provides a speed-up due to hardware acceleration. This relieves the off-chip host (terminal) from the data ordering process which is left to decode the data and display the results.

In order to prevent data loss, the data organizer is implemented as an array called order array as illustrated in Figure 4.7. The array uses two pointers, one is called p_pull and the other is the p_push. The two pointers initialize from zero. Whenever new data is logged to the array, p_push is incremented until the array is filled. Similarly, when the data is read from the order array, p_pull is incremented until the array is empty. Both pointers work in a circular buffer fashion, which implies that they do not start from zero and end at the maximum value of the array. If p_push reached its final value and the array is not full, it will be initialized to zero forming a circular buffer array. Similarly, if the p_pull reached the end of the array when the array is not yet empty, it will be set to zero. The main operation is illustrated by the order array in Figure 4.7. Initially, both pointers are at position zero as shown in Figure 4.7(a). When data is pushed to the array,

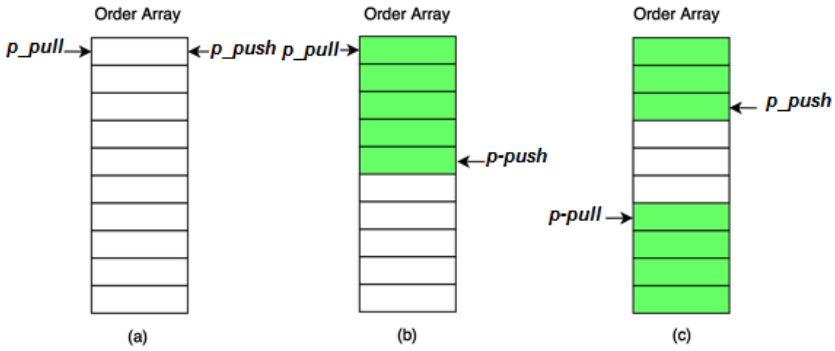


Figure 4.7: Order array circular accessing fashion. (a) Initial condition of the pointers, (b) data added to the array but none is read yet, (c) data added and read which makes the circular fashion become clear. (Green means occupied cell of the array)

p_push is incremented as shown in Figure 4.7(b) where green color indicates the occupied cell of the array. Since no data has been retrieved from the array, the pointer pull is still at its initial position. The operation continues in the same fashion. Figure 4.7(c) indicates the data added and read from the array which makes the circular fashion become clear. As can be seen, after certain read and write operations the valid data lies between p_pull and p_push in a circular fashion.

Since the data organizer is used for ordering data from 4 FIFOs, it is 4-bits wide. Each bit is assigned to a specific FIFO; bit 0 is assigned to FIFO 0 and bit 1 corresponds to FIFO 1 and so on. The array has (4xFIFO depth) cells. Each cell of the order array represents a time slot indicating the data has been stored in one or more FIFO(s). As a worst-case scenario, data will be stored in one FIFO at an instance hence requiring the number of cells in an order array 4 times the depth of one FIFO.

At each clock cycle, the FIFO count is monitored. An increase in FIFO count shows new debugging data is stored in the FIFO. Hence, the bit specific to the FIFO in the order array is set to 1. At each clock instance, the count values of all FIFOs are monitored and new cell value (between 0000 and 1111) is formulated and then added to the order array. The p_push is then incremented.

When the data is required to be extracted from the FIFOs in response to a read request, the four-bit cell pointed by the pointer (p_pull) is analyzed. The data from the FIFO specified by the cell is then extracted. If the cell indicates to extract data from more than one FIFO, FIFO 0 and FIFO 3 has the highest and lowest priority respectively. This approach ensures the correct ordering of the debugging data upon extraction. Besides correct ordering, data organizer also safeguards against data loss. On each clock cycle, the FIFO count is compared to preset threshold value and an interrupt is raised upon violation. In response to the interrupt, the FIFOs specified by the order array are read by the processor. The ARM processor reasserts

the interrupt after the data transmission process is complete. Hence, a two-way communication channel is established between processor and PL. Data organizer also compares the FIFO count to FIFO maximum depth in order to identify any data overflow. In case of an overflow, output of the Data Organizer is set to a specific value which signals an overflow and hence the data collected should not be trusted. However, such a scenario should not occur since the debugging system has been designed to ensure that the overflow is not possible.

Furthermore, the data organizer also handles another case in which the communication is terminated or interrupted during a transaction. Terminating a transaction during data transmission results in data loss. In order to solve such a problem, an additional register was implemented at the output which saves the last transaction. When the transaction is resumed after interruption, the data in this register is transmitted first and then normal process is continued.

4.2.3 Software Running on ARM

ARM is used as a controlling processor to perform the data transfer between the on-chip debugging system and the off-chip host used to perform the required data processing. The ARM processor is used as part of the NoC along with its role in debugging. In order to speed up the data transmission, the ARM processor uses the DMA for burst read transactions of varying lengths.

A dependency mechanism is implemented on the PEs in the network. The PEs seek permission before starting the main C function. The ARM sends permission after enabling the interrupt in the PL. Subsequently, the NoC starts receiving data from different PEs while the debugging system starts the data collection and interrupts generation.

Upon detection of an interrupt, the processor reads the total occupation of the four FIFOs through the Data Organizer. The processor then sets the DMA parameters based upon the FIFO occupancy and data is read. After the read operation, the interrupt is exited. The processor then sends the data to the terminal. The process is repeated iteratively.

Once the terminal receives the end signal from ARM, it starts generating the output in either a log file, a utilization report of the NoC connections or a visual representation of the transactions through the Qt based framework [209] which controls the whole debugging process.

4.3 Multiple Clock Domain Debugging Methodology

So far we have been discussing debugging single clock domain designs. Multiple clock domain debugging becomes complicated because of data invalidation due to synchronization issues. In the following sections, two different techniques will be presented to tackle the data invalidation problem.

Two frameworks using GALS and GRLS methodology will be presented in the preceding paragraphs. GALS designs utilize synchronous clocks in individual modules but such modules communicate asynchronously with one another. On the other hand, GRLS designs are synchronous at the module level but such modules communicate at rationally-related frequencies with one another.

4.3.1 Cycle-accurate Multiple Clock Domain Debugging using GALS Methodology

In order to solve the data in-validation issue, we have proposed a HW/SW debugging scheme as presented in Figure 4.8. The main modules of the debugging system are the trace buffer, synchronizer and clock controller.

4.3.1.1 Data logging

DSAS clocks the DUT only when the trace buffers are empty. At the same time, the trace buffers are provided with a controlled write *(Con_write)* which latch the debugging data from the connected nodes of the DUT. The debugging data is latched for a fixed number of clock cycles depending upon the depth of

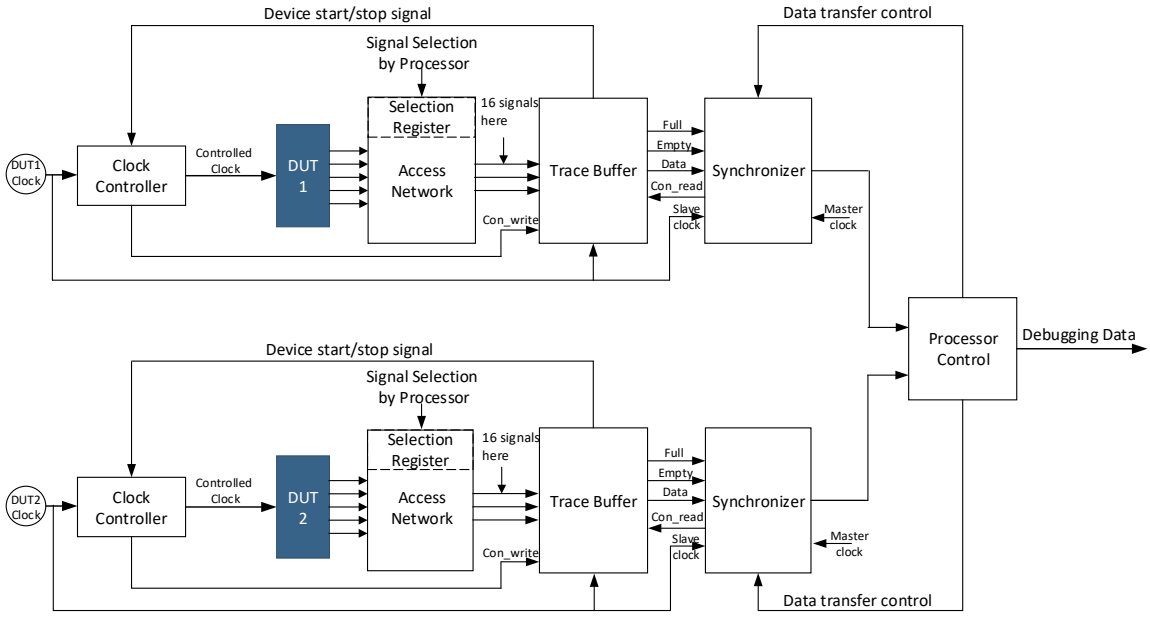


Figure 4.8: Multi-clock trace-based debugging using GALS methodology

the trace buffers. When the trace buffer is almost full, the clock of the DUT is gated. At the same time *Con_write* signal is asserted low to avoid any data loss.

4.3.1.2 Synchronization

Once the data is latched in the trace buffers, the trace buffer full signal is communicated to the synchronizer. The synchronizer then waits for the data transfer command from the controlling processor. The controlling processor issues a read command to the synchronizer which subsequently asserts controlled read *(Con_read)* to the trace buffer to high. The transfer of the data then takes place. As can be noticed during the controlled read process, the DUT is clock gated. When the trace buffer is empty, the synchronizer asserts the *Con_read* to low. Upon noticing that the trace buffers are empty, the clock controller automatically clocks the DUT and simultaneously asserts the *Con_write* to high and the data is again latched in the trace buffers and the cycle continues.

The synchronizer has two clock domains. The Slave clock is the DUT clock and the Master clock is the debugging system clock. In order to resolve the clock domain crossing, we used a synchronizer which removed cycle mismatch between the two different clock frequencies. The synchronizers utilize an independent clock asynchronous FIFO buffer with one clock input connected to the local clock and the other clock input connected to the global clock. Although other solutions have also been reported by several researchers as already covered in Chapter 2, we have adopted the asynchronous FIFO since it resolves the meta-stability and clock domain crossing issues once and for all. The main advantage of the FIFO-based synchronization is that they do not affect the locally synchronous module operation [210].

4.3.1.3 Clocking

One important aspect of the proposed debugging system is the clocking scheme. We used the DUT clock as a clock source for all sub-modules of the debugging system [Khan1]. This removed the data invalidation issue as the debugging system was completely synchronized with the DUT due to being clocked with the same clock hence either data invalidation or data aggregation did not take place.

The methodology proposed here is somehow similar to the one presented in [211]. The problem highlighted in [211] was that setting debug granularity at the cycle level for a communication-centric approach may break the undergoing handshake which leads to broken transactions resulting in data invalidation. In this research work, we resolved this issue by linking the clock control and *Con_write* with the capacity of trace buffers. This resulted in complete synchronization. The synchronizer not only helps in clock domain crossing but also synchronizes the *Con_read* signal with the transactions. The data transfer control signal from the processor is matched with the *Full* signal from the trace buffer. Once both conditions are fulfilled,

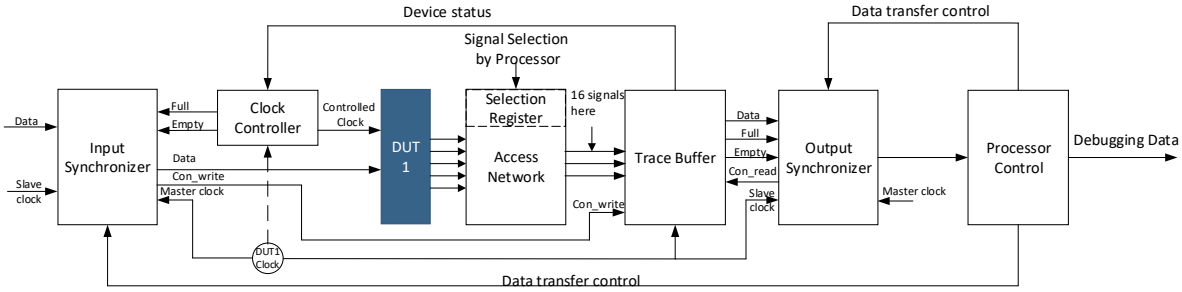


Figure 4.9: Multi-clock trace-based debugging with input and output synchronizers

the *Con_read* signal is issued. The data transfer control signal becomes an integral part of the handshake procedure between the DUT and the debugging system.

As obvious from Figure 4.8, the DUT does not receive an input data to generate the output. Hence no data in-validation takes place at the input side. The Output synchronizer takes care of the output synchronization. However, for cases where the DUT receives input from any other module IP through a handshake, a similar synchronizer is required at the input as well. The block diagram of the model with input and output synchronizers is shown in Figure 4.9.

The input synchronizer accepts the data from any input module once the empty signal from the trace buffers is high and data transfer control from the processor control is also asserted. Then it transfers the data to the DUT and also asserts *Con_write* to enable the trace buffers for data write. The methodology ensures lossless cycle-accurate debugging data for multiple-clock systems. The only requirement is that the debug-able DUTs need to connect to other modules through the proposed input and output synchronizers.

4.3.2 Cycle-accurate Multiple Clock Domain Debugging using GRLS Methodology

The methodology mentioned in section 4.3.1 ensures that no data-invalidation takes place because of multiple clock domain crossings. This is ensured by using the DUT clock for synchronization inside the debugging modules. This methodology works well for debugging the DUT with reference to its own clock. However, sometimes it becomes eminent to debug the DUT with reference to the debugging system clock because it becomes easy to relate signals from different clock domain with reference to a standard clock domain.

A solution for the described problem is shown in Figure 4.10. A notable difference of the proposed solution from the one proposed in section 4.3.1 lies in the connection of the debugging modules i.e. the trace buffers, input synchronizer, output synchronizer and the access network with the debugging system clock instead

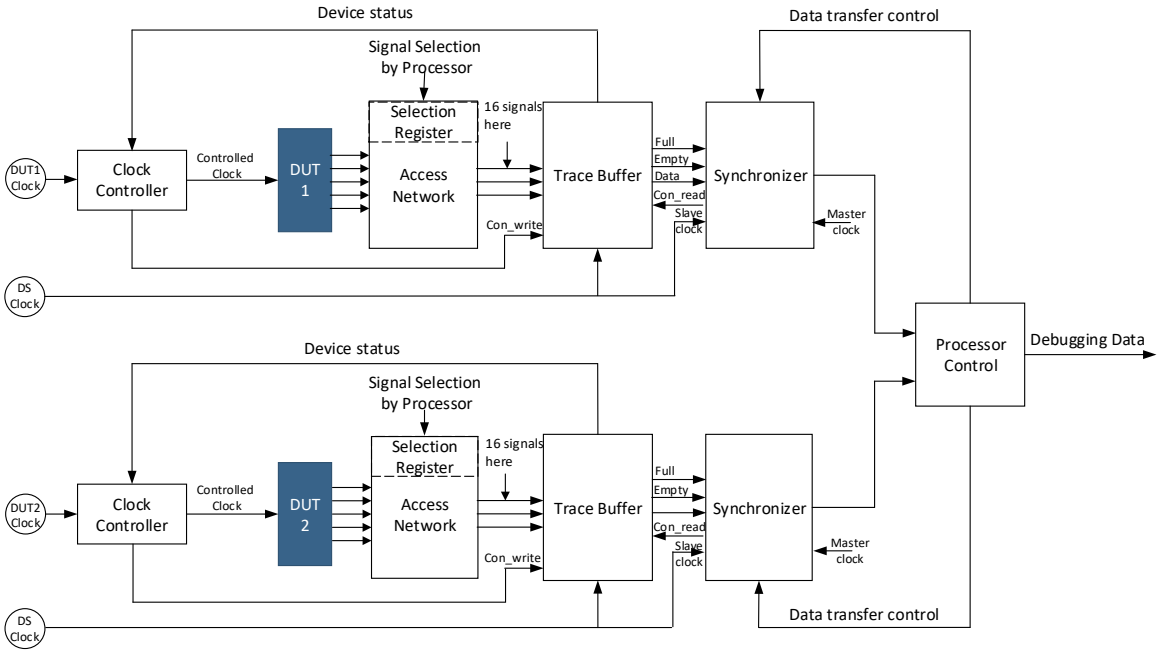


Figure 4.10: Multi-clock trace-based debugging using GRLS methodology

of the DUT clock. This methodology helps in sampling the DUT data at the debugging clock frequency and produces debugging results at the granularity of debugging clock.

4.4 Experimentation and Results

The experiments presented in this section has been tested on the Digilent Zedboard comprising an XC7Z020-484 FPGA. Xilinx Vivado 2017.1 was used for the design process which was carried on Intel Core i7-6700 CPU running at 3.4GHz and having 16GB of RAM.

4.4.1 Embedded Processor Debugging

After instantiating the DUT with the debugging circuitry, a continuous lossless stream of debugging data can be received. We chose a small data management application to illustrate the functionality of the proposed debugging system. The application performs simple data handling but when facing an interrupt, it leaves its normal mode of operation and starts a data printing task. In order to debug the microprocessor, we connected the debug test probes to the data bus for monitoring the data written to and read from the memory. The results of the test are shown in Figure 4.11.

Figure 4.11 shows the *read data* and *write data* plotted as RDATA and WDATA respectively. Similarly, *read address* and *write address* are plotted as ARADDR and AWADDR. Although, the figure shows only 18000 samples of each metric to make the figure more readable. It must be highlighted that the methodology can be used to acquire millions of continuous lossless data samples for every metric. Hence, the lossless

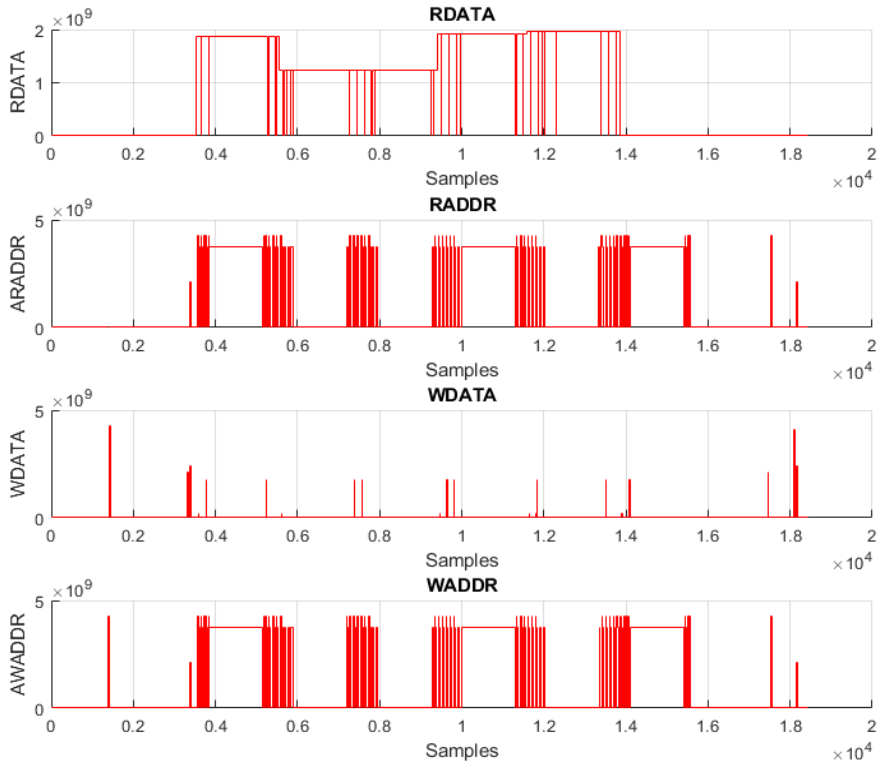


Figure 4.11: Debugging data plotted by MATLAB

data so acquired can be used to perform software re-construction which can be beneficial for software debugging.

4.4.2 Multiprocessor Debugging Methodology

The testing approach is centered on the idea of generating traffic on all lines in the NoC. The PEs seek permission from the ARM to start the application. The permission is granted only if the interrupt controller, DMA and the debugging system have been set up and the communication channel with the terminal has been established to avoid any debugging data loss.

The debugging system generates data at the beginning and the end of a packet. Therefore, the packet length or the number of flits of the packet are important for observing the debugging performance. Consequently, the debugging system should be tested using the worst-case scenario comprising the minimum packet length (i.e. two flits). It is considered as the worst case because the debugging system needs to monitor every time a PE sends the data which proves that if the system performs for minimum packet length, it can be assumed to work for different traffic scenarios.

The debugging results are then compared to the total transactions logged by the monitors during the debugging process. The data flow illustration is also tracked to ensure normal operation. The debugging

system is considered reliable if no discrepancy is found. It should be kept in mind that debugging system latency from the monitors to the FIFO is 2 clock cycle. Logging data to the FIFO can take multiple clock cycles depending upon the frequency of the transactions.

4.4.2.1 Debugging Overhead

The most important issue regarding NoC debugging is the overhead. In order to calculate the overhead, we first measured the resource utilization for different NoC sizes with or without the debugging system. The NoC sizes considered for this research work were 1x2, 1x3, 2x2, 2x3 and 3x3. The resource utilization of the NoC and the debugging system overhead is presented in Table 4-2.

It is evident that the debugging system constitutes about 33% percent of the NoC resources on average. However, the debugging system consumes only a portion of the resources. After investigation, it was found out that the interconnects used in the debugging system are the main source of debugging system overhead.

Table 4-2: Debugging overhead for different NoCs (in percents of FPGA hardware resources)

NoC size		LUT	Registers	Memory LUTs	Slices	BRAMS
1 x 2	NoC	8.04	4.67	1.60	14.14	11.79
	Debugging System	5.45	3.22	2.52	9.62	1.43
1 x 3	NoC	11.44	6.19	2.52	19.11	20.36
	Debugging System	6.50	4.15	3.49	11.96	1.43
2 x 2	NoC	15.08	7.77	3.44	23.05	28.93
	Debugging System	8.07	5.24	5.56	14.22	2.86
2 x 3	NoC	21.45	10.60	5.28	32.50	44.64
	Debugging System	10.09	7.01	7.17	18.56	2.86

In our design, we have utilized 4 AXI Stream interconnect and 1 AXI memory mapped interconnect in order to connect to the FIFOs and other slave interfaces during data collection. Those interconnects consume hardware resources. In order to investigate the effect of these interconnects on the debugging system overhead, we calculated the hardware overhead for a 3x3 NoC with and without the interconnects as shown in Figure 4.12 and Figure 4.13. Interconnects were found to consume about 60% of the debugging system resources.

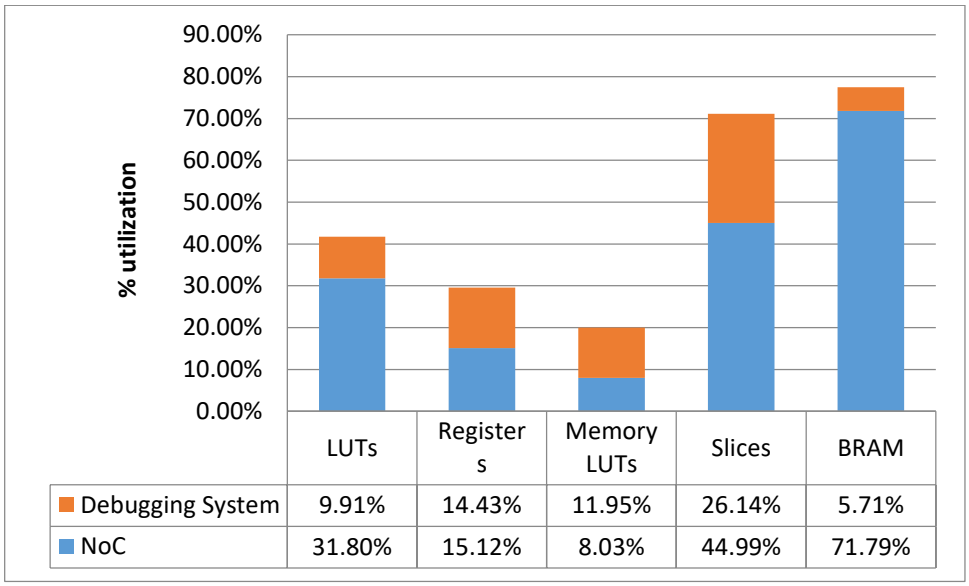


Figure 4.12: Debugging Overhead for 3x3 NoC

Another reason for the comparatively higher overhead of the debugging system, particularly the Memory LUTs, is the order array. The order array discussed in section 4.2.2 was implemented using Memory LUTs which could also be implemented using BRAMs if reduction in Memory LUTs is desired. The increase in debugging system resource utilization as a ratio of the NoC size is illustrated in Figure 4.14. It can be observed that the resource utilization ratio of NoC to debugging system decreases with an increase in NoC size.

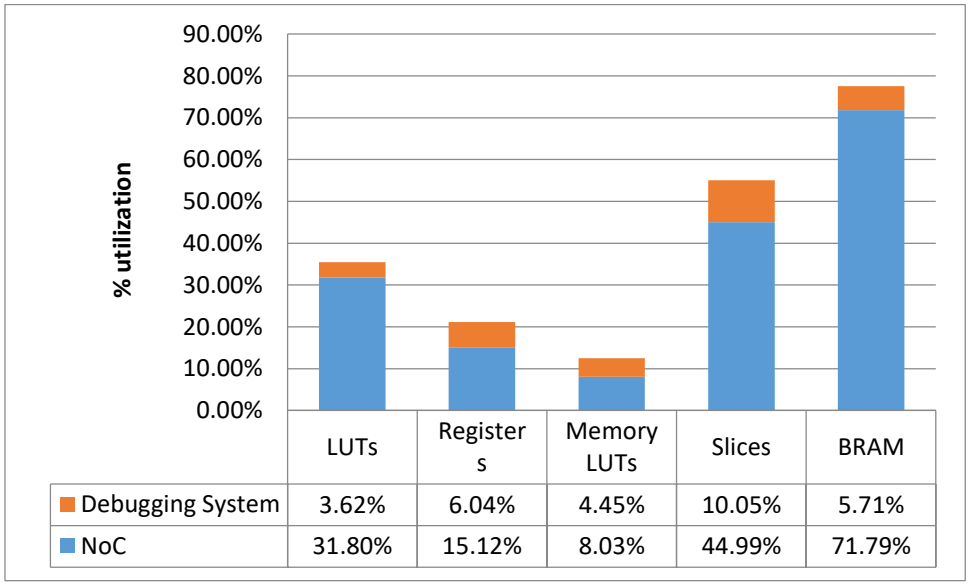


Figure 4.13: Debugging overhead for 3x3 NoC without interconnects

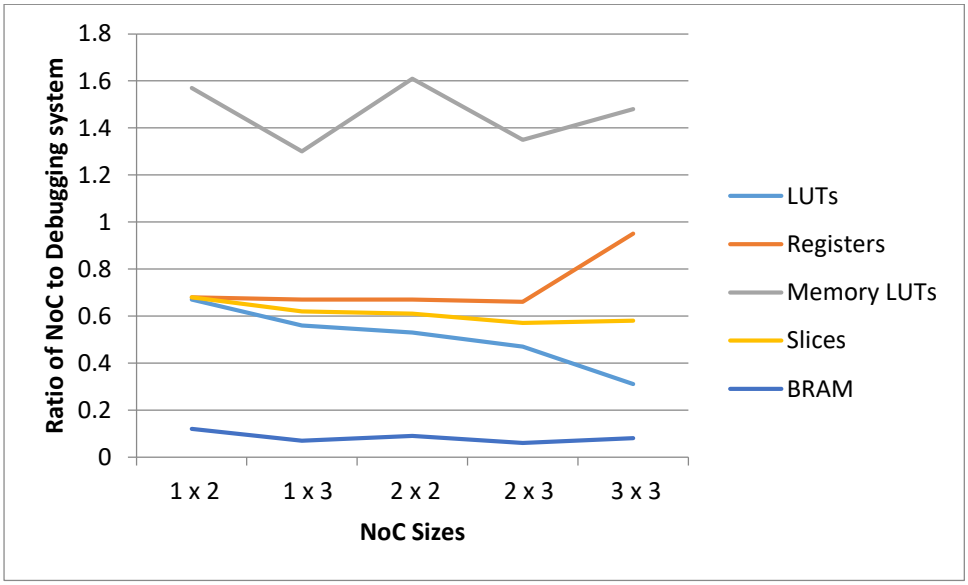


Figure 4.14: Resource utilization ratio of NoC to debugging system

4.4.3 Cycle-accurate Multiple Clock Domain Debugging Methodology

In this section, we will use two case studies to explain our proposed multiple clock debugging methodology. In the first case study, we will use cycle-accurate multiple clock domain debugging using GALS methodology for debugging of a secure IoT system. In the second case study, we will use a skid steer robotic platform used for obstacle detection to illustrate the debugging using GRLS methodology.

As mentioned in [62], the main areas of comparison for the debugging system should be the following: (a) resources utilization, (b) configuration time and (c) usability. We have added data completeness as the fourth metric to take care of lossless nature of debugging data.

4.4.3.1 Multiple Clock Domain debugging using GALS

Our first case study represents a secure IoT system with two IP cores. The first IP core realizes a constant-time implementation of Curve 25519 [21] which enables the authenticated exchange of a secret session key. The second IP core realizes a side-channel protected implementation (similar to [22]) of the low-energy block cipher PRINCE [212] to enable bulk data encryption, e.g., sensor data or network traffic. The hardware components are connected through the 32-bit standardized AXI-Lite interface to the PS-side as shown in Figure 4.15.

The Processing System (PS) is used to encrypt and decrypt the data with the PRINCE IP core. To reach a high encryption throughput, it is a realistic assumption that the cipher block runs at a faster clock domain than the key exchange (which happens only once at the beginning of a session). Consequently, Curve runs

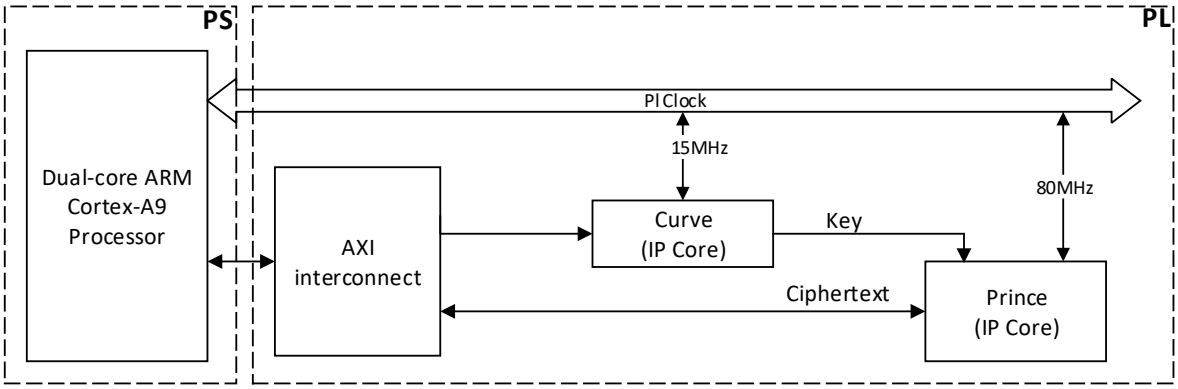


Figure 4.15: Secure IoT system

with a frequency of 15 MHz and PRINCE runs with 80 MHz having an asynchronous relationship with one another and with the debugging clock which runs as 100 MHz. The design is implemented as a side-channel protected design to defend against the well-known threat of power side-channels [20] which is relevant whenever an adversary can obtain physical access to a device.

The data encryption system was connected to the debugging system as explained in section 4.3.1. Figure 4.16 shows the debugging data acquired by the proposed debugging system. We used the encryption system to perform encryption on a sine wave as shown in first subplot. The data was encrypted as shown in the second subplot. We used the same encryption system to decrypt the data. The data after decryption is shown in the third subplot. Clearly, the data after decryption resembles the input data stream shown in first subplot.

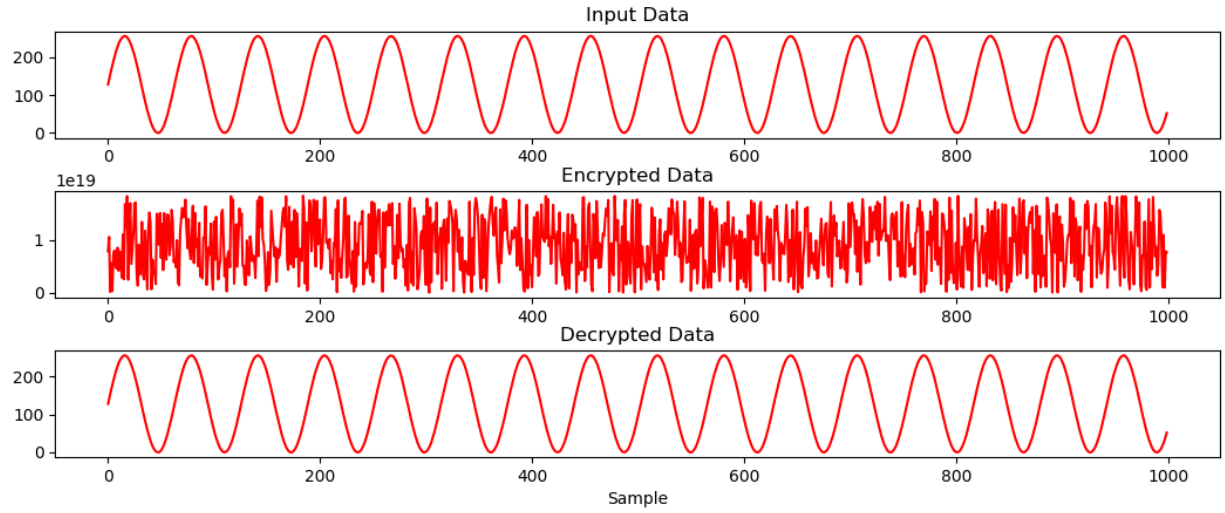


Figure 4.16: Debugging results

Hence, the debugging system was able to acquire the data-trace without any data loss which is the distinguishing characteristic of the DSAS-based debugging system.

4.4.3.2 Multiple Clock Domain Debugging using GRLS

In order to explain our proposed debugging system for GRLS-based DUTs, we used a skid steer robotic platform hardware as shown in Figure 4.17. Each of the wheels can be controlled individually, allowing the robot to translate and rotate freely in any direction. It is also possible to know its current position due to the sensors that are attached to the axis of each DC motors to move the wheels. Additionally, ultrasonic sensors are mounted on its edges, allowing the implementation of the obstacle avoidance algorithm. Consequently, there is a need to implement different IP cores to control the actuators and sensors mentioned before.

The robotic vehicle receives a command to move to a certain destination at a certain velocity. The PS computes the velocity commands and sends them to the velocity control module which translates such commands to a PWM signal. The signals are then applied to the DC motors so that the robotic vehicle can move in the desired direction. As a closed-loop, the quadrature encoders attached to the wheels are used to identify the direction in which they are rotating. Besides, this IP counts the number of pulses in a fixed period of time (ticks) to compute the speed of the robot. Consequently, its frequency is tightly linked to the DC motors and their drivers. In this case, four PWM signals are produced, one for each motor and the duty cycle is varied to change the speed accordingly. The obstacle identification module is used to trigger the Ultrasonic Sensors and read their output so that the processor can calculate the distance of any obstacle in the vicinity of the robot accordingly.

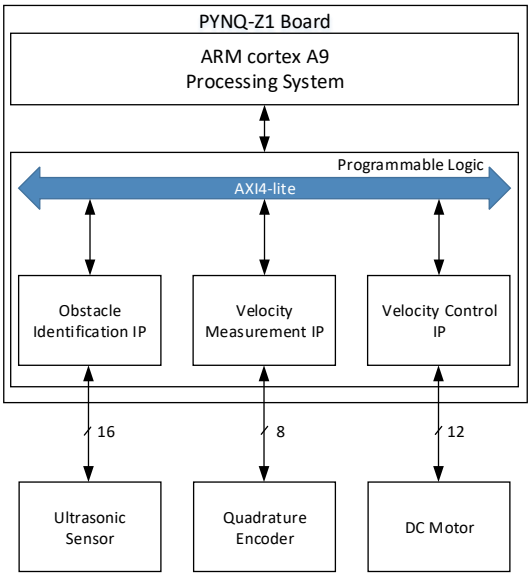


Figure 4.17: Robotic hardware

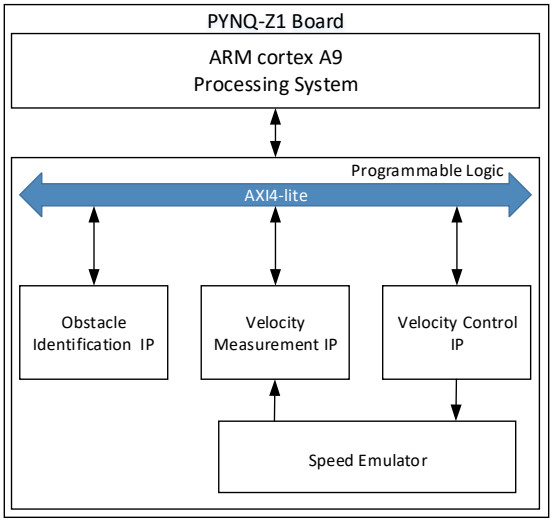


Figure 4.18: Robotic hardware verification

Based upon the description of the platform, it can be concluded that the verification of the hardware is quite complex because of its tight coupling to actual sensors. Consequently, the verification cannot be performed in the absence of actual sensors. But, the hardware verification is desired before its integration with actual sensors. We circumvented the problem by devising an in-loop test as shown in Figure 4.18. We used the processor to generate velocity commands based upon the destination as usual. These velocity commands are used by specially designed speed emulator IP introduced to emulate the DC motor and quadrature encoder. Based upon the velocity commands from the PS, the speed emulator generates different signal frequencies synchronous with the global clock (100MHz) making it a GRLS system. These changing signal frequencies emulate different speed by altering the number of pulses which are then used by velocity measurement IP to figure out the speed of the robot.

During the verification phase, the speed emulator generated 5MHz, 10MHz and 20MHz frequencies synchronous to the debugging clock (100 MHz). When the speed emulator output signal was altered based upon the signal from the velocity control module, it was found that the change in frequency resulted in a change in the number of ticks as expected. We further found out that doubling the frequency resulted in an almost doubling the tick count. This points out that the hardware is performing its desired function.

4.4.3.3 Resource Utilization

Resource utilization of the presented debugging system for multiple clock domain is shown in Figure 4.19. 16 nodes with data width of 32 bits each (≈16*32 signals) were monitored for the said research work. In comparison to earlier published work [Khan4] [Khan1], the resources consumption has increased. The main reason for the increase in the resources is the use of separate trace buffers and synchronization modules for

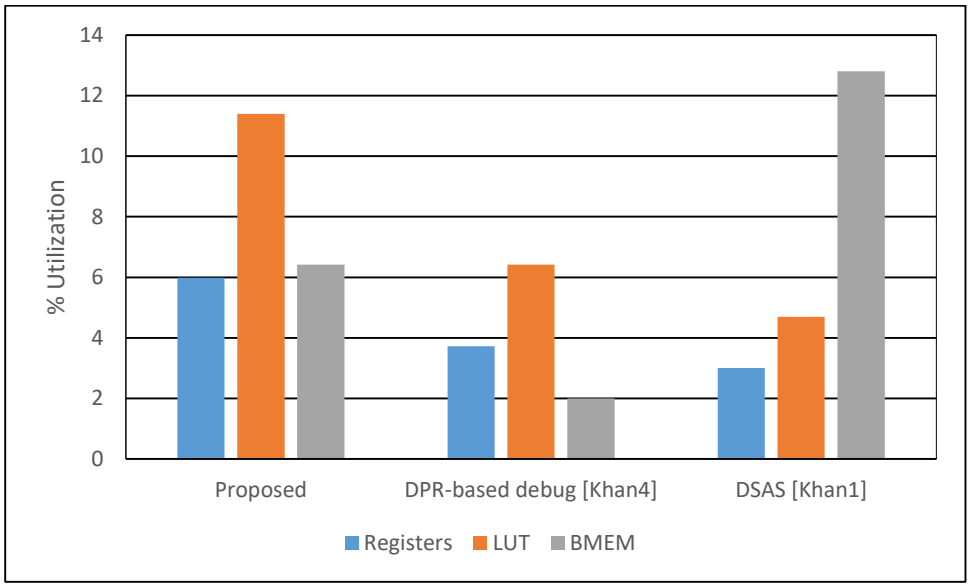


Figure 4.19: Resource utilization

multiple clock domains. The resource utilization increases proportionally with the number of clock domains. The almost linear trend is shown in Figure 4.20.

4.4.3.4 Configuration Time

The second most important metric for evaluation of the debugging system is the configuration time. Bitstream regeneration is not required for selecting new signals to analyze since the signal set can be changed at debug-time through the use of selection register instead of compile time. This methodology, by

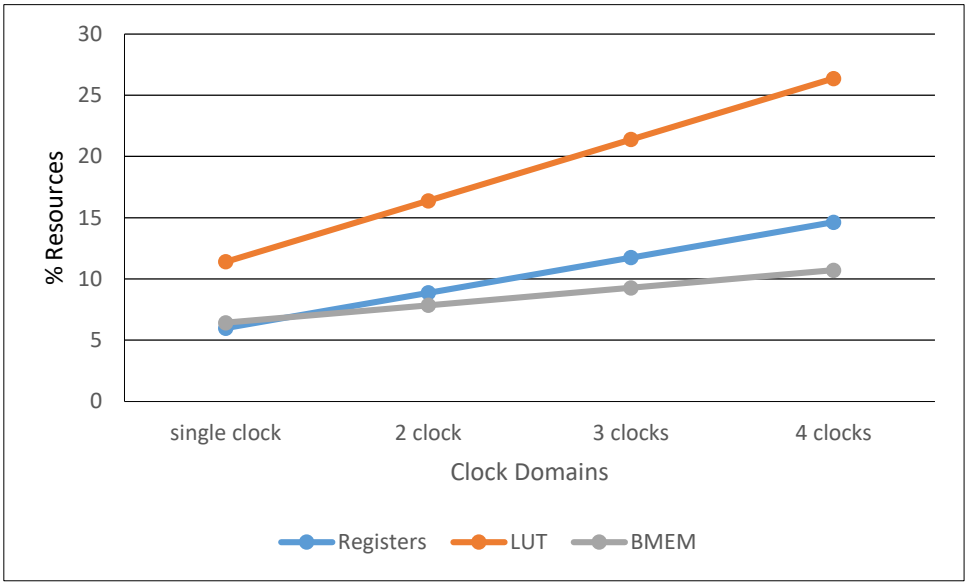


Figure 4.20: Effect of multiple clocks on hardware utilization

itself, results in shorter development times. In contrast, ILA based approaches monitor a small number of signals at a time.

Any change in the signal set requires new re-compilation of the design which is a time-consuming process. One technique to address the problem is to anticipate the problem causing signals and select all of them in advance for complete special trace. However, this solution requires the necessary resources allocated for debugging purpose. Furthermore, even if re-compilation is not required in such case, still ILA-based approaches capture only a subset of the temporal trace depending upon the depth of the trace buffer.

4.4.3.5 Data Completeness

The proposed methodology fundamentally differs from the existing debugging techniques such as readback through scan [62] or breakpoint-based [143] as mentioned in the related work. The existing solutions can capture data after hitting a breakpoint and hence data after the breakpoint can be captured and that too for limited clock cycles. This essentially results in the lossy debugging data. On the other hand, the proposed solution captures a lossless stream of debugging data because of its cycle-accurate nature and hence allows a complete replay of the debugging data which can capture intermittent errors.

4.5 Summary

In this chapter debugging of embedded processors, multiprocessors and multi-clock systems have been covered in detail. During debugging of embedded processors, the processor is treated as a black box and all interfaces originating from the processor are continuously monitored. This results in providing a complete log of the activities performed by the embedded processor. It has been found that the proposed solution can be used to debug embedded processors taking advantage of the continuous lossless debugging data stream. However, the design has one limitation; it can monitor interfaces and nodes external to the processor. However, internal nodes and registers such as program counter etc. which are not visible to the debugging system cannot be debugged.

An event-based NoC debugging framework was also proposed for debugging of NoCs. The system has two main components, on-chip debugging hardware and an off-chip host. On-chip debugging is based on the idea of debugging all connections between the PEs and the routers and the routers themselves. Monitors connected directly to each interconnection in the NoC can monitor the start and end of the packets being transferred. Data is collected and saved until the memory is almost full. The processor is then interrupted to read the data and send it to the off-chip host for data processing and analysis which can then be used for debugging the NoC.

Importance of NoCs is increasing with each passing day due to the requirement of faster devices which suggests more PEs inside the system. Consequently, the NoC consumes a lot of resources resulting in the

availability of limited resources for the debugging system. As evident from the results, with an increase in the size of the NoC, the resource required for debugging increased proportionally. This emphasizes the need to reduce the debugging system overhead. As a first approach for decreasing this overhead, AXI stream interconnects utilized in the debugging framework should be avoided, replacing them with a custom interconnects specifically designed for this purpose. Internal FIFOs of the AXI stream interconnect could also be avoided by replacing them with a pipelined architecture which performs the same role with fewer resource. Furthermore, utilization of the Memory LUTs can be reduced by implementing the order array using BRAMs. This introduces a latency of one clock cycle when the data is first time written to the array. However, it does not affect system performance.

The third problem addressed is the debugging of multi-clock systems. The multi-clock debugging becomes difficult due to the presence of multiple clock domains because of data in-validation or data aggregation issues. In order to carry out the task, we presented a debugging solution for multiple clock domains. We proposed debugging using GALS and GRLS methodologies. Synchronizers were used to take care of data in-validation problems. Results have proven that the proposed techniques can be efficiently used for debugging with small resource overhead.

Chapter 5 Observability Enhancement Tools

The DSAS debugging methodology greatly enhanced the lossless trace data collection due to clock management. Later, the access network was augmented to the debugging system which increased its signal monitoring capability. The access network allowed to change the signal set at runtime through its configuration register. This eliminated the re-compilation required for changing the signal set and enhanced the capability of the debugging system to monitor large number of signals. However, connecting thousands of nodes manually can lead to errors. An automatic tool which can connect the DUT to the access network can be extremely helpful. Such connectivity generation tool flow will be presented in this chapter.

During the connection of the DUT with the access network, a criterion is required to map the signals of DUT to the Network. Such mapping can be generated based upon heuristics. However, connecting the signal set heuristically cannot be considered a smart design. As the access network can have thousands of nodes, probing these nodes to find the error-prone signals is time-inefficient. If the error-prone signals can somehow be identified, monitoring such signals at the first hand can point towards the health of the hardware design. Hence, a new criterion for signal selection will be presented which can increase the debugging efficiency by improving the time efficiency.

In this chapter, we will first introduce an automated connectivity generator which can enhance on-chip visibility of signals buried deep inside the hardware. Subsequently, we will be introducing a selection methodology based upon the signal priority.

5.1 Automated Connectivity Generation

Concentration networks allow changing the signal subset without re-compilation by providing connectivity from the nodes to the debugging system. Although, the connections can be performed manually but needs a lot of effort and the procedure is still error-prone. In [90], a script tool was utilized for extraction of design nodes from HDL simulators. The IP-XACT standard [101] completely describes the interfaces for the elements present in an architecture in which four schemata acquire their main descriptions namely the design, component, bus and abstraction definitions. The design description acquires all the components present inside the architecture including the interconnections between the components and the external pins. The component description acquires the configurations that are part of each component including the bus interfaces and their ports, channels, address maps, clock and reset signals. For a communication bus, the bus and abstraction definitions are provided as complimentary descriptions. The bus definition includes the high-level attributes of a bus, such as the connection method and addressing. The abstraction definition, on the other hand, captures the low-level attributes, such as the name, width and the direction of the ports.

Connectivity between an access network and a debugging system can be generated utilizing the interface definitions described in this standard. This approach helps to reduce the human intervention required to perform manual connections between the debugging system and the access network.

The connectivity-generation tool flow presented in this section extends from the work [213], adding a step for automatic connectivity generation between the DUTs and an access network. The tool uses IP-XACT descriptions to capture the interfaces from all the components present in a design, including the design itself and the interconnections between the components. The main purpose of IP-XACT [214] standard is to ease design integration so that it can be reused in different development tools. It was created by Spirit Consortium for facilitating the exchange of IPs between different vendors so that they can be integrated easily in different electronic design automation (EDA) tools. Currently, it is regulated by Accellera as an IEEE 1685-2014 standard. An IP-XACT description acquires the parameter choices made for configurable aspects of an IP. It then becomes easier to integrate such IPs in different designs using development tools from different vendors.

The connectivity generation process is started by declaring the interconnections and the ad-hoc connections in the design schema. Interconnections are defined as the point to point connections between two bus interfaces. Ad-hoc connections directly connect two ports without using the interconnects or bus interface. This includes external pin (input and output) connections to component ports, e.g. clock and reset signals, or any component port feeding to other components without a protocol such as a clock divider.

The connectivity-generation tool flow comprises of three steps, as shown in Figure 5.1. The generated files help in automating the debugging process by ensuring the correct connectivity. Each step will be described in the proceeding sections.

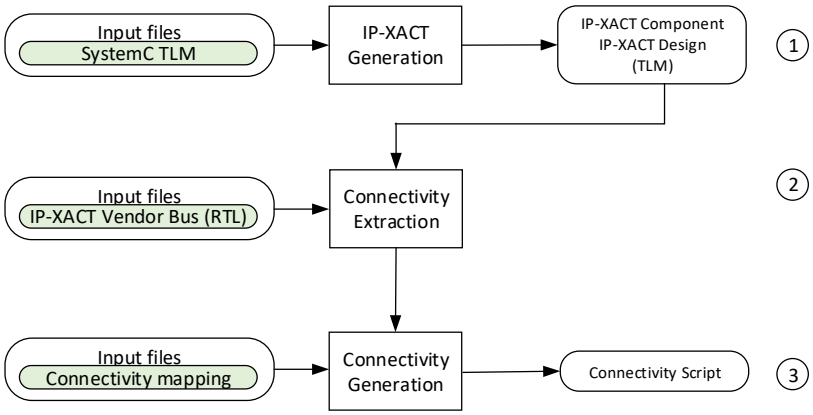


Figure 5.1: Flow of the developed tool

5.1.1 IP-XACT Descriptions Generation

As shown in part 1 from Figure 5.1, the first step of the proposed connectivity tool uses a SystemC model to generate the IP-XACT descriptions. This is the main entry point of the tool. The tool accepts model files as input and outputs the IP-XACT descriptions for every component in the design and also for the design itself.

Tool SCiPX was used for translation of the SystemC model files to IP-XACT whose detailed description can be found in [214].

5.1.2 Connectivity Extraction

As illustrated in Figure 5.1, the second step of the proposed tool accepts files from two sources. First, the files generated in step 1 are inputs to the second step. Secondly it gets the IP-XACT descriptions for the communication protocols, provided by the vendors, which are segregated into bus definitions and abstraction definitions. This results in lowering the complexity of the previous step.

This step can also serve as an entry point for designs in which the internal components of the architecture are already available and only the connectivity from the DUT to the access network needs to be generated.

During this step, connectivity information present in the IP-XACT descriptions is extracted and then passed on to the next step. The tool analyzes the design descriptions which have a complete list of all connections. For the interconnections, it is important to find the correct bus and abstraction definitions used for defining the signals. However, each ad-hoc connection has two possibilities. It is required to find out if an input pin is feeding internal pins, or a component's output pin is feeding other pins.

5.1.3 Connectivity Generation

In the third step of the proposed method as illustrated in part 3 from Figure 5.1, the tool receives a mapping file of the DUT nodes desired to be connected to specific pins of the access network.

Using the extracted information after parsing the IP-XACT descriptions in the second step along with the desired port mapping, the tool generates connections for the signals present in the mapping.

The output of this step is a Tcl script which contains the commands required to automatically connect the DUT nodes to the access network. Once connected, the controlling processor can be used to provide desired connectivity for debugging or other requirements.

5.2 Priority-based Signal Selection

As stated in section 5.1.3 and also evident from Figure 5.1, a connectivity mapping file is required to map the signals from the DUT to the access network for observation. In order to increase the time-efficiency of

debugging, it is essential to select the most relevant signals for observation. The challenge is compounded by the fact that the signal with greater probability of errors must be selected before design implementation, even when the response of the design is unknown. Although, some "important" signals can be selected manually by the designers. However, for complex designs the debug engineer needs not to have complete knowledge of the whole design. Another benefit of the automated signal selection is that it will decrease the turnaround time between implementation re-spins which may take hours for design recompilation hence reducing the debugging time.

Several researchers have tackled the signal selection problem from various dimensions. Priority decision based upon the signal restoration of untraced signals was performed by Ko et al. [78] and Basu et al. [128]. Similarly, signal relevance based upon error propagation was done by Yang et al. [77]. Chatterjee et al. [154] proposed simulation-based priority for signal restorability. Hung et al. [160] suggested performing priority decision based upon connectivity. We suggest to conduct the signal selection based upon the presence of correlation in the trace data.

This was accomplished by using an inference system [Khan5] [Khan6] which makes inference regarding the prioritization of signals (Figure 5.2). The inference system accepts the debugging data from the simulation model and can draw certain inferences by using the rules in its ruleset. Rules set consists of cross-correlation, autocorrelation and linear regression. As the signal selection is made when the hardware is still not available, the behavioral simulation model is utilized as the input to our inference system. However, if post-synthesis or post-implementation simulation results are available, they must be given preference.

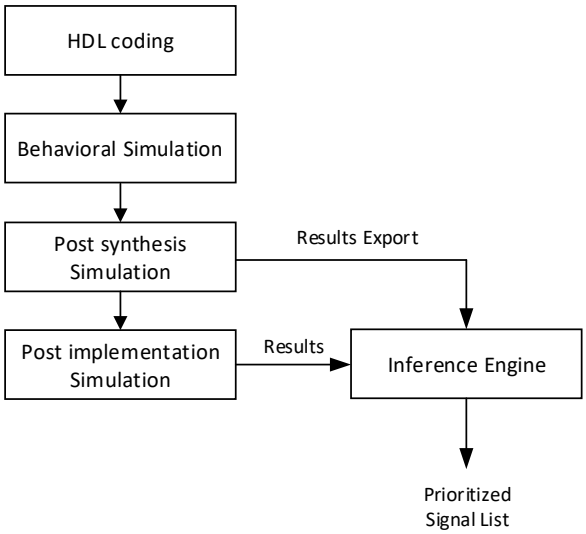


Figure 5.2: Inference system

The inference system uses different rules to search for the presence of a pattern in the received simulation data so that the signal selection can be made based upon the found pattern. Autocorrelation is conducted to reach two important conclusions:

i. To detect the non-randomness in data

ii. To recognize the repeating data series if the data is non-random.

The main algorithm performed by the inference engine (Figure 5.3) is:

1. It performs autocorrelation and finds the peaks present in the autocorrelation coefficient. For non-random data, one or more values of the autocorrelation coefficient should be significantly non-zero.
2. For normalized autocorrelation function, if the second and third highest peaks are greater than 0.2, the data is assumed to be non-random.
3. The corresponding dataset between the maximum and second-highest autocorrelation coefficient is retrieved.

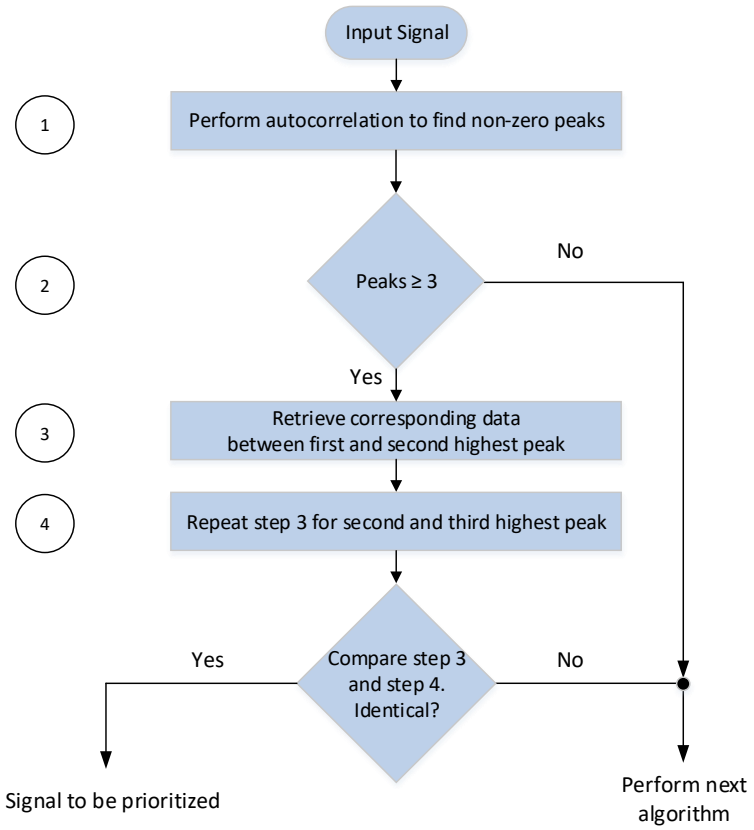


Figure 5.3: Signal priority flow diagram (algorithm 1)

4. Step 3 is repeated for the second-highest and third-highest autocorrelation coefficient.
5. The two datasets are compared. If identical, the signal is selected for higher priority.
6. If the autocorrelation coefficient approaches zero except the maximum at lag 0, the inference engine exits the autocorrelation function.

As a side note, the uncorrelated data does not essentially mean randomness. Data, not exhibiting significant autocorrelation, can still demonstrate non-randomness. Therefore, the inference engine applies another rule i.e. linear regression. During linear regression, the inference engine performs linearization of the debugging data. Then, it uses Eq. 5.1. to calculate the slope at each clock interval.

$$\mathrm{m} = \frac{y_2 - y_1}{x_2 - x_1} \qquad (Eq. 5.1)$$

In order to construct the template, we also need the x-intercept which is calculated by Eq. 5.2.

$$\mathrm{b} = y_2 - \left(\frac{y_2 - y_1}{x_2 - x_1}\right) * x_2 \qquad (Eq. 5.2)$$

After linearization, the inference engine performs autocorrelation of the slope function of the received signal. If the data exhibits a relationship within itself, it should be manifested when the autocorrelation of the slope function is performed. The dataset between the maximum and the second-highest autocorrelation coefficient (of slopes of the data set) is extracted from the dataset. Similarly, the process is repeated for the second and third highest peaks of the autocorrelation function as explained in the algorithm in Figure 5.4. If the retrieved slope dataset from the two steps matches with one another, the concerned signal is prioritized for signal selection. In few cases, the data sets at step 6 do not compare. In such cases, the signal is tested for the number of states in the data. The number of states is calculated for the entire signal set at each cycle transition. The signal with the lowest number of states is given priority over other competing signals. The reason for priority to minimum number of states in data is that debugging may become easier by using clustering-based debugging techniques.

5.3 Experimentation and Results

We used a specifically designed data sorter IP as DUT to illustrate the proposed signal selection methodology. The basic building block for the implemented data sorter is shown as highlighted in Figure 5.5. It has two inputs and two outputs. The “sort” block makes a comparison between the two inputs. It transfers the bigger input to Out_A and the smaller one to Out_B. This basic block can be concatenated to

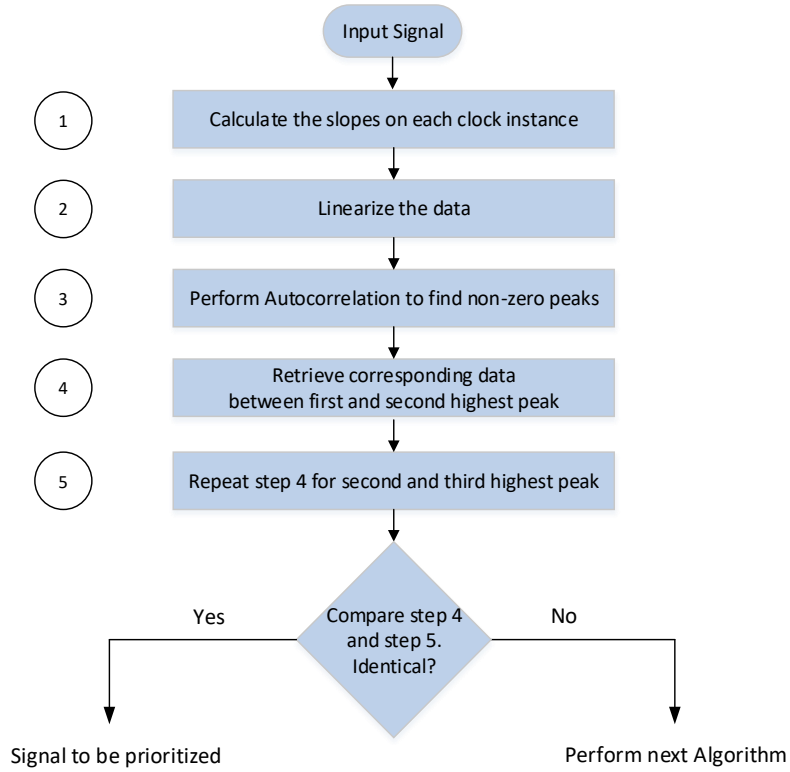


Figure 5.4: Signal priority flow diagram (algorithm 2)

extend the design for more than two inputs. In our design, we extended the basic block to sort out four inputs as shown in Figure 5.5.

In order to illustrate our signal selection methodology, we connected the input of the data sorter IP with different pseudorandom signal generators to provide different inputs to be sorted overtime on each clock

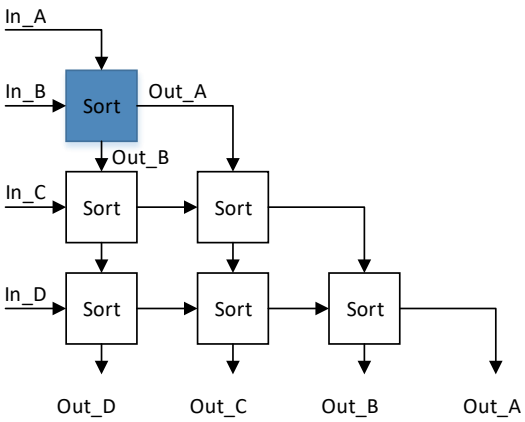


Figure 5.5: Data sorter

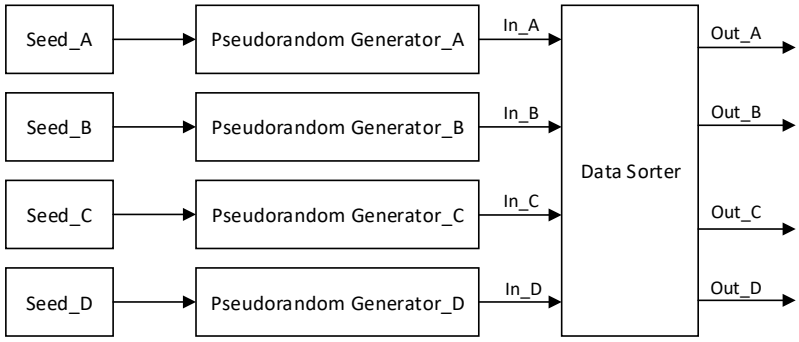


Figure 5.6: Data sorter with number generators

cycle. Each pseudorandom number generator block takes a constant value as its initial seed as shown in Figure 5.6.

Based on the seed, the pseudorandom number generators create input for the sorter DUT. The data sorter produces an output on every clock cycle as shown in Figure 5.7. The HDL simulation data from the DUT is then exported to the terminal. It is evident from the figure that signal selection can be quite challenging if the decision is based upon visual features. Once the simulation data is received, the inference system performs the signal selection algorithms in the predefined order to decide the signal priority.

In order to decide the priority of signals, the inference system started from Algorithm 1. As specified in the algorithm, the inference system performed the autocorrelation of the input signals to decide the priority. In the above-referred use case, the inference system performed the algorithm on the four signals namely

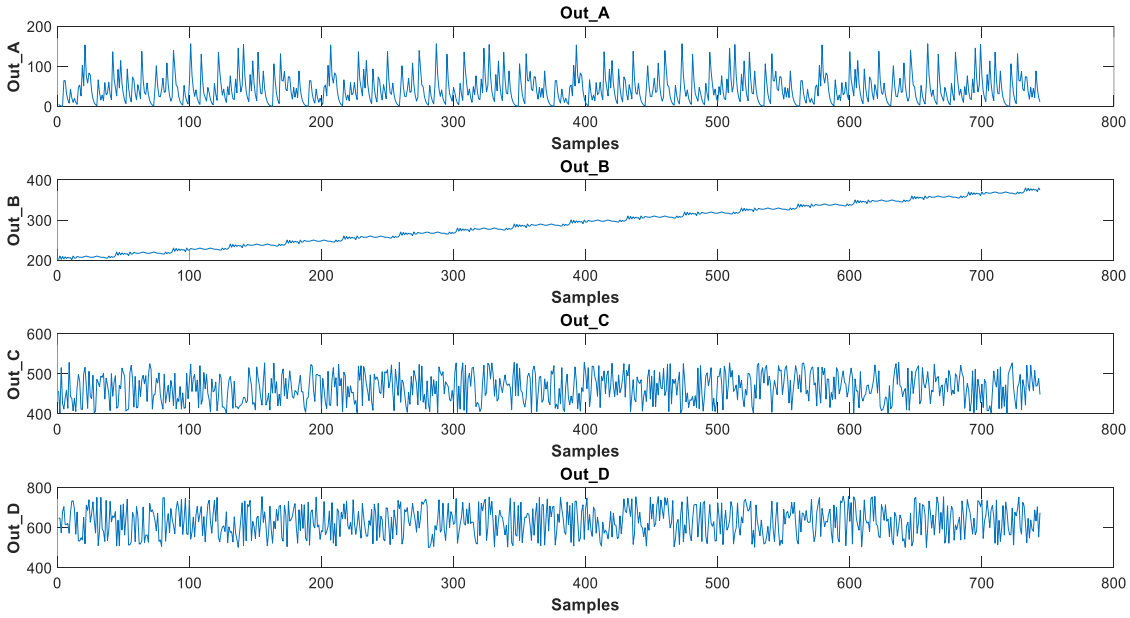


Figure 5.7: Simulation results for signals Out_A, Out_B, Out_C, Out_D

Out_A, Out_B, Out_C and Out_D. The results of the autocorrelation performed as per Algorithm 1 (Figure 5.3) are shown in Figure 5.8. In the figure, the autocorrelation coefficient has been plotted against the lag for four output signals. The autocorrelation coefficient for signal Out_A has multiple peaks demonstrating some sort of autocorrelation among different sequences of the signal. Hence, signal Out_A is selected for the highest priority among other candidate signals. However, for the remaining three signals, no peaks were found indicating that the signals do not have an autocorrelation. Resultantly, priority for remaining signals could not be decided at this stage.

Based on the results from Algorithm 1, the inference system moved to Algorithm 2. Following the algorithm, the inference system switched to linear regression. First, the linearization was performed at each transition and then the slopes of the complete data were calculated. The autocorrelation of the slopes, as shown in Figure 5.9 clearly shows that certain peaks are present in the slope dataset of signal Out_B demonstrating a strong correlation in signal Out_B. Then the algorithm as per Figure 5.4 was followed to ascertain the presence of patterns. However, no such peaks were reported in signals Out_C or Out_D demonstrating that the signals Out_C or Out_D do not exhibit enough correlation amongst the slope dataset. Hence Out_B can be prioritized as an outcome of algorithm 2.

When the signals have been prioritized based upon the proposed algorithms, the priority among the left-over signals is decided by the number of states present in the signals. Since the signals Out_C or Out_D do not demonstrate any correlation, the priority amongst them is decided based upon the number of distinct states as shown in Table 5-1.

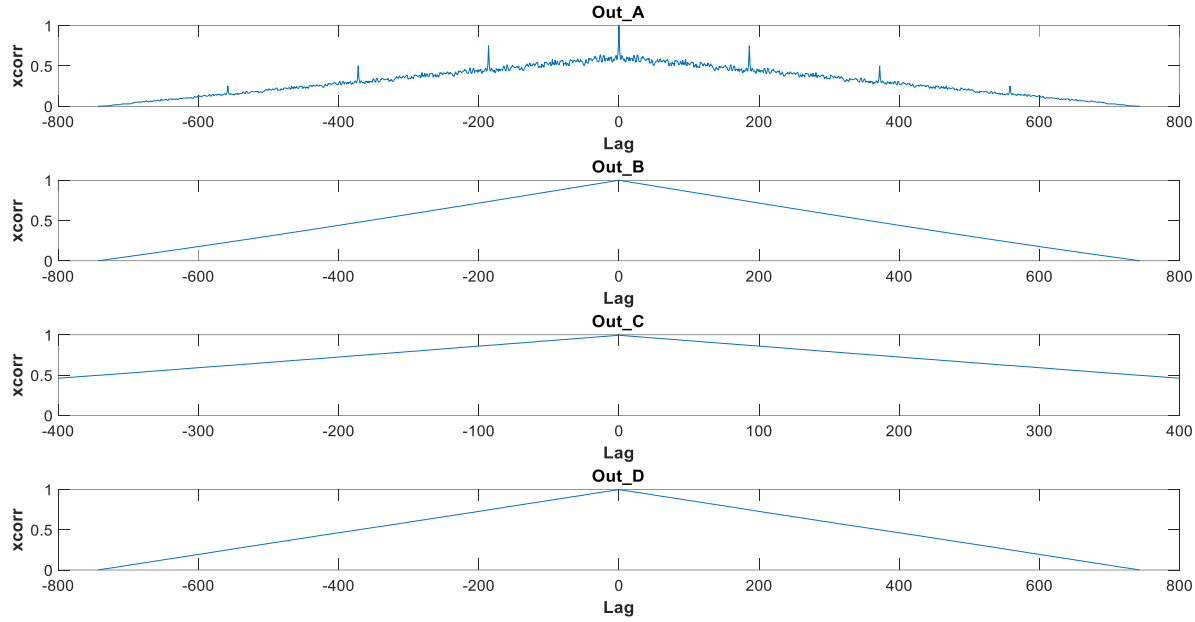


Figure 5.8: Signal priority decision by algorithm 1

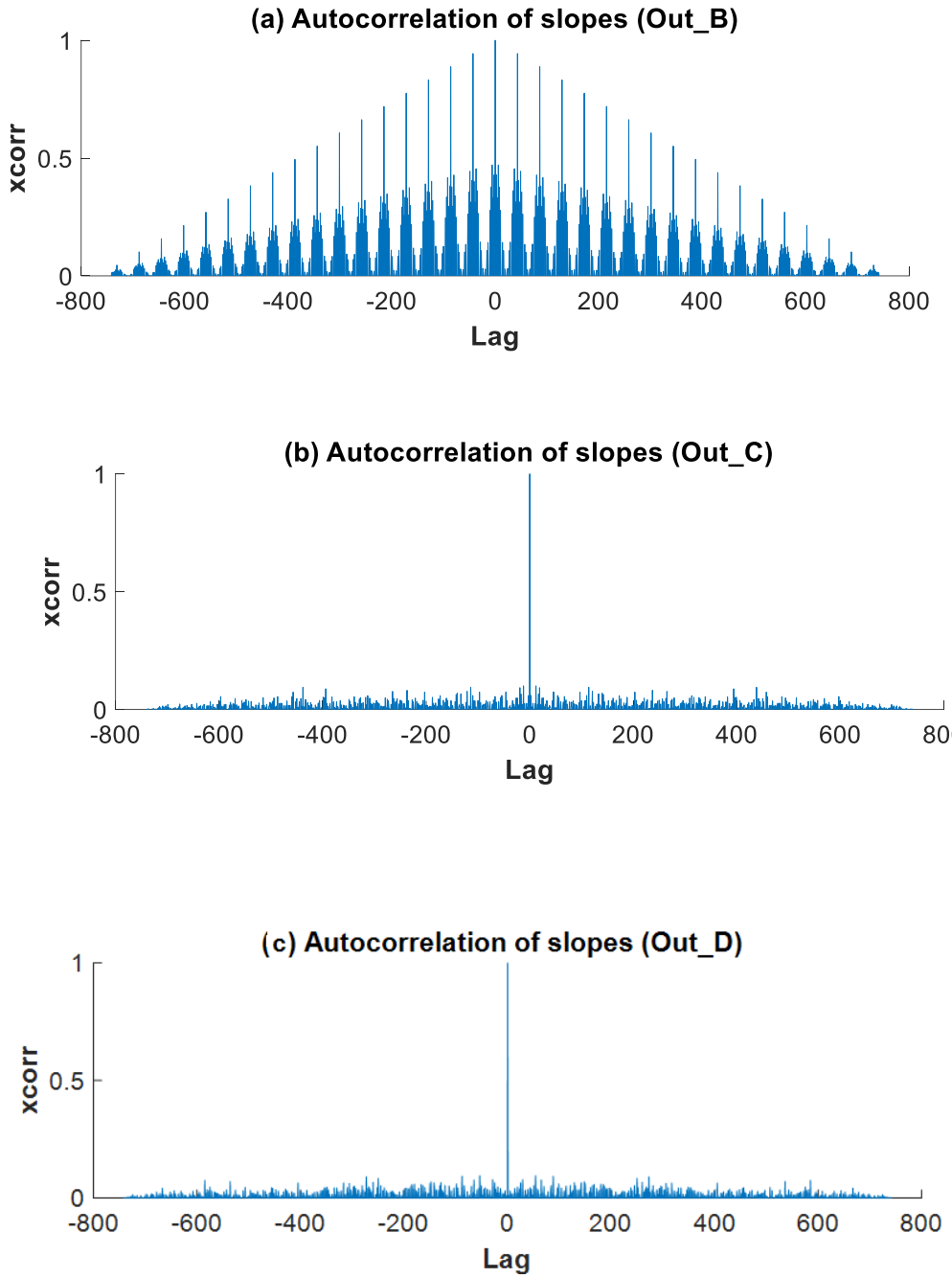


Figure 5.9: Signal priority decision by algorithm 2

Hence, based upon the number of distinct states present in signals Out_C and Out_D, signal Out_C is selected for priority mapping ahead of signal Out_D. In case of multiple signals falling in the same category, the priority is decided based upon the first occurrence. In the current use-case, we have considered four signals and used our proposed signal selection methodology to prioritize the signals. However, the proposed methodology can be utilized for any number of signals.

Table 5-1: Number of states

Signal name	Number of states
Out_C	128
Out_D	246

5.4 Summary

In this chapter, signal priority-based connectivity mapping was proposed. The connectivity generation tool accepts input files in several formats and generates a Tcl script which can be used to automate the connection process. The tool works well for simple hierarchy. However, one key limitation is that when the hierarchy becomes complex due to several levels of the parent-child relationship; the tool fails to respond.

A signal selection methodology for trace buffer-based post-silicon debugging of hardware designs was also presented. The methodology uses the simulation results to identify the signals which can be prioritized. Based on two proposed algorithms, the signals exhibiting pattern sequence are identified which can then be prioritized for debugging at first place. Such signals can be automatically connected to concentration networks thus eliminating human errors. During the debugging phase, the signals selected through the proposed methodology can identify the functional as well as intermittent errors. The proposed methodology can be used with debugging systems capable of generating lossless trace of debugging data.

Chapter 6 Automated Error Detection

As stated by Wilson Group Research report [27], almost 50% of the design time is spent on verification. The report further claims that 42% of the verification time is spent on the actual debugging process. The main reason behind the enormous time being spent on debugging is the increased size, complexity and higher operating frequencies of the FPGA designs. All these factors drastically affect the vulnerability of such devices to bugs or errors.

Such bugs or errors can be broadly divided into permanent, intermittent or transient errors [12]. Permanent errors may result due to design or logic errors. Instances of permanent errors are a state machine caught in an endless loop, wrong implementation of a mathematical function, open or short transistors, or timing violations, etc. Such errors may be easy to debug due to their permanent or stuck-at nature. Upon observation of an error, verification engineers can find out the design error manually through simulation or ILA-based debugging methods. Intermittent errors or soft errors arise from Single Event Upsets(SEU) which occur due to electromagnetic interference, manufacturing imperfections or operating conditions. Such SEUs result in configuration memory upsets or the state changes in flip-flops or memory cells which result in intermittent errors. Similarly, transient errors may occur due to neutron or alpha particles. As there is very minute difference between transient or intermittent errors; for the sake of this work we will consider both intermittent or transient errors in the same category. These errors are a source of major concern for safety-critical or life-critical applications. Such errors are difficult to observe by using traditional trace buffers or scan chain approaches owing to their random nature. However, they can be captured through cycle-accurate lossless debugging techniques. However, the issue remains:

1. Is there any technique which can help in capturing both the permanent and intermittent errors without continuous monitoring of debugging data on every clock transition manually?
2. Is there any methodology which can help in finding the intermittent errors when the GR is not available?

In order to tackle the issue (i), we introduced a rule-based inference system [Khan5] which can find the permanent and intermittent errors by providing an unlimited capture window by intruding the DUT through clock management. It also eliminates the human intervention required to monitor the debugging data on every clock transition. This is achieved by performing a correspondence analysis to identify the relationship between the input data and the GR. The effectivity of such solution can further be enhanced by

supplementing a stimulus-driven environment. Hence, rule-based inference system is augmented with an FIL debugging technique to achieve debug automation.

The issue (ii) can be addressed by enhancing the rule base of the inference system which can select an appropriate rule to identify the intermittent errors when the GR is not present.

In order to highlight how the above-mentioned issues have been tackled in this research, the following two use case will be used:

1) Image processing techniques are very useful in robot localization and mapping. A configurable convolutional filter can be utilized for illustrating the proposed debugging methodology. In this case study, the rule-based inference system uses a GR to find debugging errors without requiring to monitor each transition manually. By employing the power of the numerical computation-based Software Debugging Environment (SDE), debugging the complete trace at once, reduces the debugging time and resultantly the overall design cycle.

2) A data sorting algorithm which receives N inputs and sorts them in ascending order. Here, the rule-based inference system can capture the errors in the absence of a GR.

Therefore, the preceding case studies will be used for highlighting the following main contributions:

1. Rule-based inference system is introduced which can find permanent and intermittent errors by using the lossless debugging trace.
2. Rule-based inference system will be extended to include the FIL simulation. SDE generates a stimulus for the DUT. The response of the DUT is utilized to perform the correspondence analysis between the received debugging data and the simulation data from either a behavioral model, a post-synthesis or a post-implementation model of the DUT used as a GR.
3. Rule-based inference system can be enhanced to find intermittent errors in the absence of a GR by augmenting the rules set

6.1 Debugging using a Rule-based Inference System

DSAS provides cycle-accurate lossless debugging trace due to clock gating. The data is then transferred to the terminal for analysis. However, as the trace data generated by DSAS is huge, it is painstaking to find permanent or intermittent errors without monitoring of debugging data on every clock transition manually. In this section, a new verification method for hardware debugging namely the rule-based inference system is proposed as shown in Figure 6.1. In the proposed verification system, the rule-based inference running on the PC uses the GR, which can either be defined using the inference system or user-defined, to find the

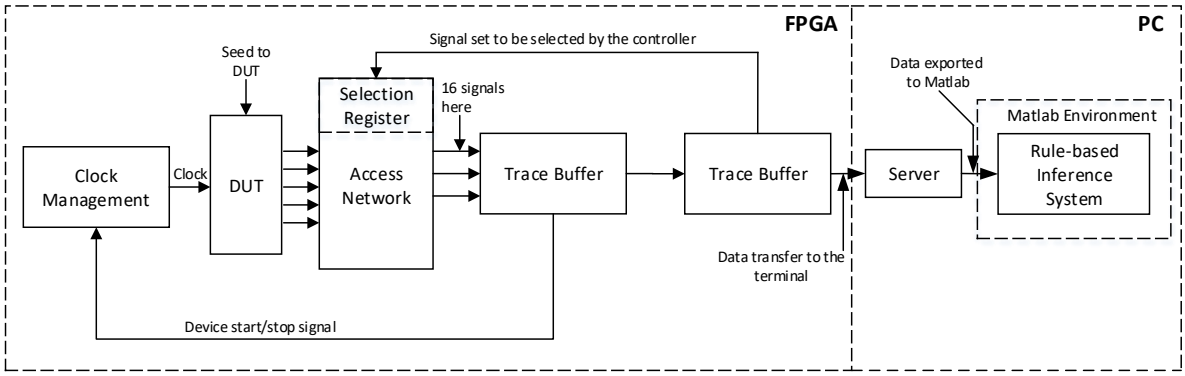


Figure 6.1: Debugging by rule-based inference system

bugs without requiring to run the system incessantly / intermittently. This is accomplished by debugging the complete trace at once by utilizing numerical computing environment such as MATLAB or Octave; this reduces the debugging time and hence, the overall design cycle.

6.1.1 Rule-based Inference System

Rule-based inference system is the main computing engine of the proposed methodology. Hence, it is worthwhile to explain it in more detail. Figure 6.2 illustrates three main parts of the inference system namely the knowledge base, the inference engine, and the rules set. The inference system integrates a knowledge base containing the accumulated experience and an inference engine which applies the knowledge base to a particular situation, based upon a set of rules. The inference system capabilities can be enhanced by inserting data to the knowledge base or to the rules. The knowledge base can be populated with three types of data sets namely model results, database or the user-defined results. The inference engine is the core of the system. Its main features include retrieving data from the knowledge base, selecting an appropriate rule from the ruleset, applying the rule to the received trace data and the retrieved data from the knowledge base

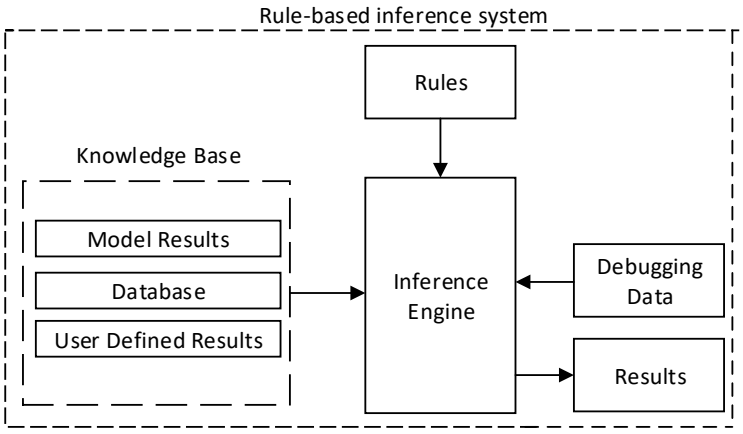


Figure 6.2: Rule-based inference system

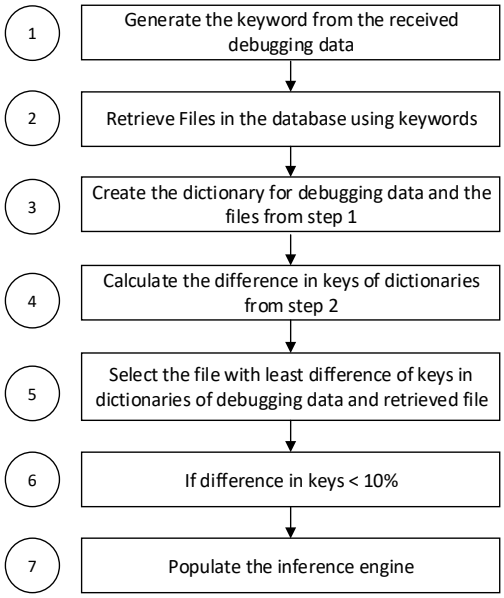


Figure 6.3: Data mining process

and the interpretation of the results. If the model results are unavailable, the inference engine can mine for one in the database (if a similar design was debugged in the past and results were saved to the knowledge base) and uses the dataset as a template.

The data mining process to mine for appropriate results in the database is illustrated in Figure 6.3. When the inference engine is loaded with the logged debugging data-trace, it generates a list of keywords based upon the received data. The keyword is generated by using the maximum occurrences of a sample and maximum consecutive occurrences of a specific sample in the debugging data. Using the keywords, the database is searched and all relevant files are retrieved. At step 3, the log file containing the debugging data and the retrieved files from the database are employed to create dictionaries. Then, the generated keys from the dictionaries are compared against each other to find out the difference. At step 5, the file having the minimum key difference between the debugging data and the retrieved files is picked as a potential candidate for loading to the inference engine. If the difference in keys between the debugging data log file and the selected candidate is less than 10%, the candidate file is populated to the inference engine. Otherwise, the file is discarded and the logged debugging data is saved in the knowledge base by using the keyword at step 1 for future use.

Rules set consists of cross-correlation, autocorrelation and linear regression. If an appropriate file from the knowledge base is available, the inference engine performs cross-correlation to find the disparity between the debugging data and the retrieved data from the knowledge base. Cross-correlation function indicates the correspondence between the database and the debugging data. The maximum cross-correlation value is

obtained when the two datasets perfectly match. The lag between them can also be found.

Result interpretation is the last task of the inference engine. A cross-correlation coefficient of 0.0 portrays no match. A cross-correlation coefficient of 1.0 depicts a perfect match. This implies that one dataset can be inferred from the other, either directly or by using a positive scale factor. A cross-correlation coefficient of -1.0 represents maximum negative correlation, implying that one dataset can be derived from the other using a negative scale factor. Coefficient values between 0 and 1 depict a partial match. A cross-correlation coefficient greater than 0.90 may suggest a very good similarity between the datasets [215]. However, for debugging, a perfect match is needed. The inference engine also points towards the lag against maximum cross-correlation. Then, the lag is utilized to synchronize the two data sets. After synchronization, all mismatch instances can be found. Hence, the rule-based inference system can make debugging easier by performing the correspondence analysis. It may save a significant amount of time by eliminating the human interpretation.

6.1.2 FIL Debugging with a Rule-based Inference System

The rule-based inference system can be made more effective by augmenting a stimulus driving mechanism (used to generate a stimulus) which can be collectively termed as SDE. This results in a hardware-software co-debug environment. In the proposed FIL debugging methodology, the DUT is connected to the DSAS. A UART or Ethernet interface is utilized for connecting the debugging system to the SDE. The debugging controller receives the stimulus from the SDE and applies it to DUT, which responds to this stimulus. The response from the DUT is logged to the trace buffers. The debugging system retrieves the data from onboard trace buffers and transmits it to the terminal. Saved data is utilized by Octave-based SDE employing a rule-based inference system approach. A block diagram of the DSAS-based FIL debugging is illustrated in Figure 6.4.

Additionally, the model of the DUT may not be available in certain cases or it may be too time-consuming to generate one. To sort out this problem, we have adopted a High-Level Validation (HLV) strategy in our FIL-based debugging solution as presented in Figure 6.5. HLV performs a behavioral, post-synthesis or a post-implementation simulation using a stored, prepared or randomly generated input data. Then, it saves the results in the database. Such simulations produce a continuous stream of cycle-accurate simulation data. Consequently, during the normal workflow, we already have verified simulation results. Such verified results can be utilized as a GR for DUT validation. After software verification, we move towards hardware validation. At the second stage, the log file containing the simulation results along with the input data is exported to the SDE. SDE extracts the input data from the log file and applies it to the DUT. The cycle-accurate response of the DUT is then compared with the simulation results obtained at stage one.

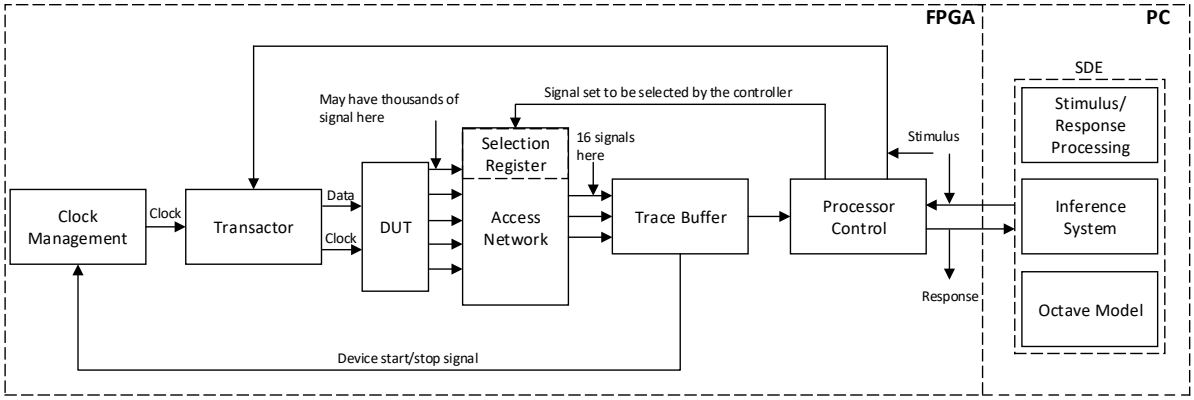


Figure 6.4: DSAS-based FIL

Thus, the approach compares the hardware trace data against the set of expected results obtained from simulation. However, this self-checking approach does not function with pseudorandom inputs [216]. One remedy to such a problem is to generate a pseudorandom input dataset and apply the same inputs first to the simulation model and then to the DUT. Both outputs can then be compared.

Different high-level simulation models from HDL simulators can be utilized. Hence, no additional time is required in devising a simulation model as the case with MATLAB-based FIL [166]. Subsequently, the rule-based inference system can be used to detect discrepancies. The main advantage of this technique is an FIL testing of an embedded design which can provide cycle-accurate debugging data. Moreover, the debugging data can be compared against behavioral simulation, post-synthesis or post-implementation simulation results.

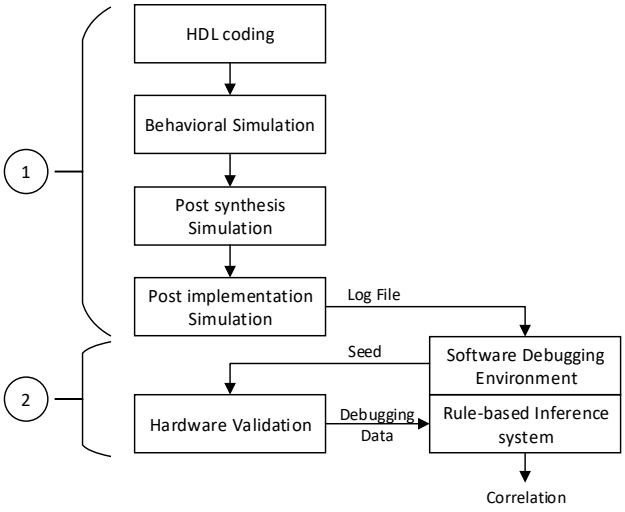


Figure 6.5: Flowchart for HLV

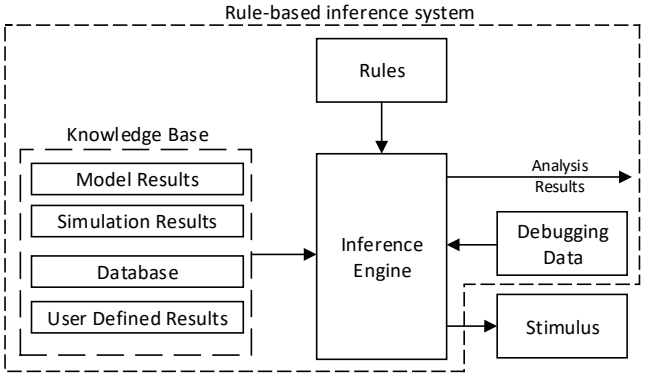


Figure 6.6: Software debugging environment

6.1.3 Software Debugging Environment (SDE)

The software debugging environment controls the debugging process by generating the stimulus required for the DUT, receiving the response and analyzing the results. For debugging systems capable of providing a continuous lossless trace of data such as DSAS, manual comparison required for result analysis creates a bottleneck. Usage of the high-level numerical computing environment can accelerate the verification process. Consequently, it reduces the user intervention needed to verify and debug complex designs.

Rule-based inference system has already been described in the previous section. Figure 6.6 illustrates three main parts of the inference system namely the knowledge base, the inference engine, and the rules set. One notable difference, in this case, is that the simulation results generated by HLV are also utilized by the inference system for decision making. These results are mined from the knowledge base and used as a GR as explained in the previous section.

However, sometimes the GR may not be available. The proposed debugging methodology provides a solution to such problems by enhancing the capabilities of the rule-based inference system as shown in Figure 6.7 . When the inference engine is unable to find the GR, it switches its rule from cross-correlation and conducts autocorrelation of the received trace data. During such autocorrelation operation, the inference engine tries to locate any pattern strings in the received trace data so that its values may be predicted based on the located pattern. Autocorrelation is performed to reach two important conclusions.

1. To detect the non-randomness in data
2. To recognize the repeating data series if the data is non-random.

The main algorithm implemented by the inference engine in this scenario is:

1. It performs autocorrelation and finds the peaks present in the autocorrelation coefficient. For non-random data, one or more values of the autocorrelation coefficient should be significantly non-zero.

2. For normalized autocorrelation function, if the second and third highest peaks are greater than 0.2, the data is assumed to be non-random.

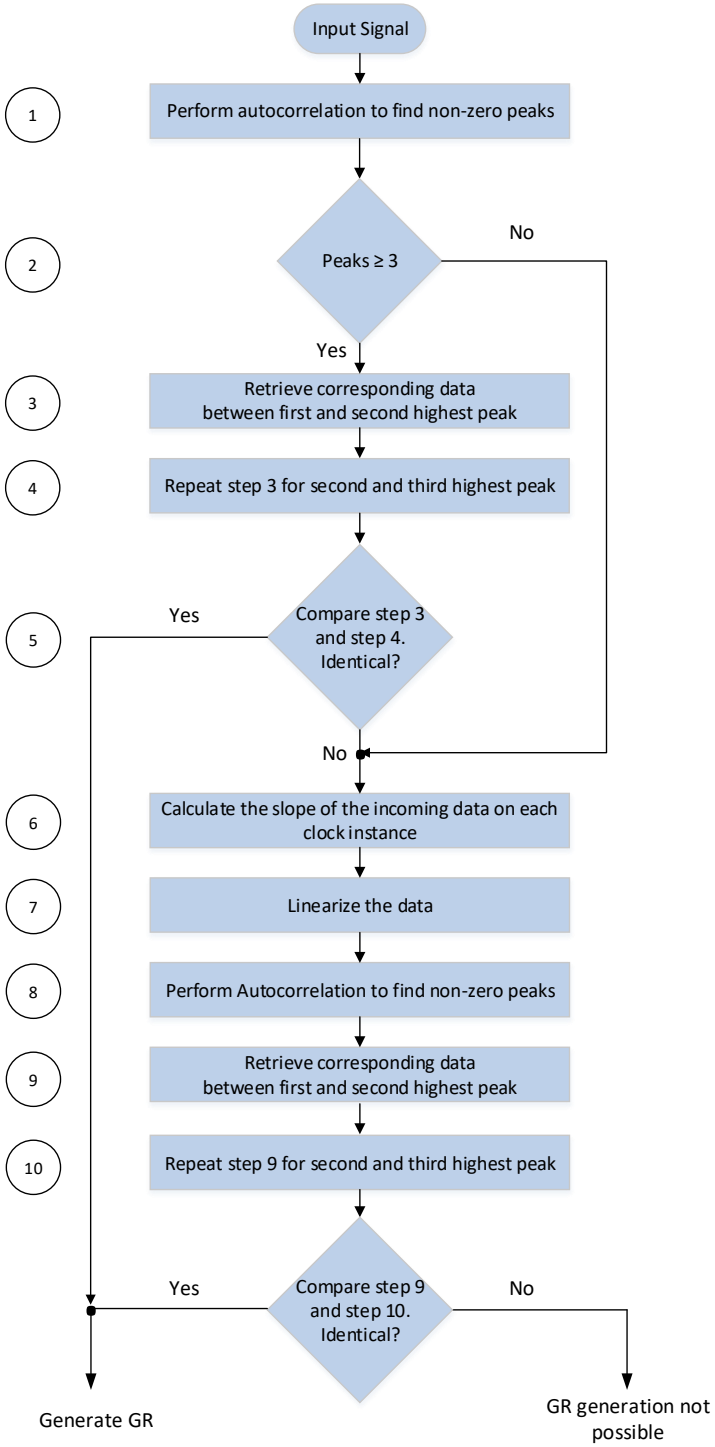


Figure 6.7: GR generation algorithm

3. The corresponding dataset between the maximum and second highest autocorrelation coefficient is retrieved.
4. Step 3 is repeated for the second highest and third highest autocorrelation coefficient.
5. Compare the two datasets. If identical, use the retrieved data to generate the GR.
6. Debugging using the generated GR.

As a side note, the uncorrelated data does not essentially mean randomness. Data, not exhibiting significant autocorrelation, can still demonstrate non-randomness. Therefore, the inference engine applies another rule i.e. linear regression. During linear regression, the inference engine performs linearization of the debugging data. Then, it calculates the slope and the x-intercept at each clock interval using equation $Eq.5.1$ and $Eq.5.2$ respectively. The GR template can be constructed using the calculated slope and x-intercept.

After linearization, the inference engine performs autocorrelation of the slope function of the received signal. If the autocorrelation coefficient approaches zero except the maximum at lag 0, the inference engine exits the autocorrelation function. When a relationship exists within the data, it should be manifested when the autocorrelation of the slope function is performed. The dataset between the maximum and the second highest autocorrelation coefficient (of slopes of the data set) is extracted from the dataset. Similarly, the process is repeated for the second and third highest peaks of the autocorrelation function as previously described in the algorithm. Subsequently, the two retrieved datasets between the first-second and second-third peaks are compared. If identical, the slope dataset can be used for generating the debugging template. Finally, the cross correlation between the generated template and the received data is performed. The results are then interpreted to find the disparity between the generated GR and the received trace. Hence, the rule-based inference system can perform debugging even in the absence of GR by conducting the correspondence analysis. It may save a significant amount of time by eliminating the human interpretation.

The main benefit of the proposed debugging methodology is that contrary to the limited window based debugging systems, the DSAS approach can acquire an extremely large lossless trace. It can monitor 16 signals (with 32 bits each resulting in 512 bits) simultaneously where each signal may have millions of samples. Manual comparison of such huge dataset may become time-inefficient since every single transition needs to be compared with the corresponding transition from the GR (sample number in this case). By adopting the rule-based inference system, debugging becomes easier as the debugging data is plotted with relevant GR overlay. Moreover, the system performs the cross-correlation (or an appropriate rule) and plots the results. The problem aggravates due to the unavailability of the GR. In such a case, rule-based inference system attempts to generate a GR by identifying any pattern strings after analyzing the debugging data. However, if the simulation results are neither present nor can be generated, the user can load his own

template for populating the knowledge base; because, the expected output is generally known. However, if the knowledge base is devoid of any template, manual debugging can still be performed. After debugging, when the user is satisfied with the results, the database can be populated for future use.

6.2 Experimentation and Results

In this section, we will present two use cases to highlight our debugging methodology. In the first use case, we will utilize the GR generated by the Octave-based model of the convolutional filter. In the second use case, we will implement a pseudorandom data sorter IP to explain the debugging using GR developed through HLV methodology. The same IP will be used to present the case in which a GR is not available. Subsequently, we will use the inference engine to perform debugging. The proposed methodology has been verified on the Zedboard [197] featuring a Xilinx's XC7Z020-484 FPGA. An Intel Core i7-6700 CPU running at 3.4GHz with 16GB RAM was used for the SDE.

6.2.1 Error Detection by Model-generated GR

The methodology has been validated by an image processing application as DUT, using Xilinx Vivado HLS 2017.4 [217] for the design process. The IP consists of two AXI4-Stream ports that serve as input and output ports for data and an AXI4-Lite port that is utilized to configure the 3x3 convolutional kernel. Besides, the control port of the IP is utilized for controlling the IP from the processor. The IP generated through HLS is imported into Vivado.

SDE generates a stimulus and sends it to the DUT through the debugging system, and also to the Octave-based simulation model of the DUT. The DUT responds to the stimulus which is then latched to the trace buffers and subsequently, transmitted to SDE for FIL validation.

The SDE running on the PC has five main tasks as shown in Figure 6.8. The first one is to establish a communication channel between the PC and the FPGA. Later, the SDE reads each image from the hard drive of the PC, previously resized to 256x256 and converted to grayscale, and sends them one after another to the FPGA as stimulus. Then, it receives the processed image from the FPGA, as a transaction. The SDE waits till the complete response from the FPGA has been received. Subsequently, the same stimulus is applied to the Octave-based simulation model based on *Eq. 6.1*. Finally, the original image and the results from both sources, i.e. the FPGA and Octave are displayed. Correlation between these last two is performed to verify the accuracy of the implemented DUT.

Code implemented in Vivado HLS is employed in Octave with minor modifications. The main difference between the filter implemented in HLS and the one in Octave is that in Vivado HLS, the image is streamed via the AXI-Stream interface. Hence, the generated IP only waits until the number of pixels shown in *Eq. 6.2* have been received for starting the convolutional process, regardless of the fact that the entire image is

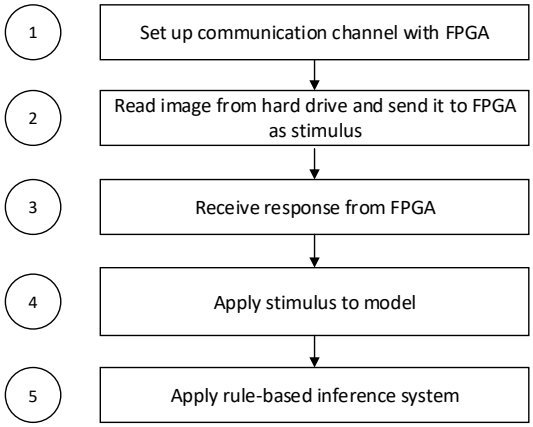


Figure 6.8: Process flow for SDE

not available yet. This is achieved by using a line buffer to cache the incoming data. Then, this sliding window is multiplied by the filter kernel to obtain the convolution as shown in *Eq. 6.1*. The proposed debugging system can then be utilized to perform the functional verification of the hardware IP. Debugging system probes are connected to the input and output ports of the DUT to achieve cycle-accurate debugging data.

$$\sum_{y=2}^{ROWS}\sum_{x=2}^{COLS}\begin{pmatrix} img(y-1,x-1)*kernel(1,1)+ \\ img(y-1,x)*kernel(1,2)+ \\ img(y-1,x+1)*kernel(1,3)+ \\ img(y,x-1)*kernel(2,1)+ \\ img(y,x)*kernel(2,2)+ \\ img(y,x+1)*kernel(2,3)+ \\ img(y+1,x-1)*kernel(3,1)+ \\ img(y+1,x)*kernel(3,2)+ \\ img(y+1,x+1)*kernel(3,3)+ \end{pmatrix} \quad \textit{(Eq.6.1)}$$

$$(COLS*(kernel_{Dim}-1))+(kernel_{Dim}) \quad \textit{(Eq.6.2)}$$

Where $COLS$ is the number of Columns in the image and $kernel_{Dim}$ is the dimension of the kernel.

In Figure 6.9, the input and output of the image processing DUT are presented. Figure 6.9(a) shows the input to be applied to the DUT and the Octave model. The Sobel filter kernel is applied to the DUT and the Octave model. Then, the SDE sends the image to the DUT for performing the image filtering. The debugging system gathers data from the probes and logs it to on-chip trace buffers. The controlling processor acquires the data from the trace buffers and transmits it to the SDE. Figure 6.9(b) shows the debugging data after Sobel edge detection by the DUT.

(a) Image Sent from PC to FPGA

(b) Image Sent from FPGA to PC

(c) Image filtered in Octave (Correlation 1.0)

Figure 6.9: Debugging by FIL

In many cases, the visual response of the DUT is considered adequate. However, the accuracy of the results acquired from the hardware IP and the Octave model can be confirmed by utilizing the rule-based inference system. Once, the relevant data has been loaded into the inference engine, it performs cross-correlation between the two datasets and shows the result. The outcome of the cross-correlation function is an array of data presenting the similarity between the debugging data and the database. A 256 x 256 8-bit pixel image is equal to 64 KB of data. Manual comparison between such huge datasets is time-consuming. However, the inference engine provides a time-efficient comparison. Figure 6.9(c) shows the debugging data plotted besides the output from the Octave-based model including the computed correlation coefficient. The correlation coefficient is 1.0 which indicates that the response from the FPGA matches perfectly with the octave model of the DUT. This highlights that the hardware is free from permanent or intermittent errors.

6.2.2 Error Detection by HLV-generated GR

In the absence of a model of the DUT, debugging can be performed by utilizing HLV methodology. In order to explain our HLV-based debugging methodology, we have utilized the data sorter IP as DUT. The data sorter already explained in section 5.3 is used to sort inputs in ascending order. We used the IP shown in Figure 5.6 as it allows to alter the seed from the SDE if desired.

Vivado tools, explained in Figure 6.5, are employed to perform the simulations. After post-implementation simulation, our test bench saves the result in a log file. It also appends the seed values to the simulation results. The simulation results are then loaded into the knowledge base of the inference system. SDE reads the log file containing the simulation results. It retrieves the seed and applies them to the hardware-based pseudorandom number generator. Utilizing this seed, the pseudorandom number generator generates input for the data sorter. Then, the data sorter produces an output at each clock cycle. Subsequently, the DUT is debugged through DSAS. The debugging system probes are connected to the DUT pins. Once, the trace buffers connected to the DUT are full, the DUT is stopped by the clock manager. Then, the debugging data is exported to the SDE. Accordingly, the debugging environment is split between the hardware data capture

mechanism and the Octave-based SDE. This technique reduces the simulation time because, instead of a full software solution, a hardware/software environment has been created for providing hardware acceleration. After receiving the debugging data, the inference system loads the post-implementation simulation results from the knowledge base. Then, the inference engine performs cross-correlation between the data received from the DUT and the simulation results, and plots the cross-correlation coefficient against the lag.

In order to debug the data sorter, the design was operated at 100 MHz resulting in a clock cycle of 10 ns. At each clock cycle, it is essential to analyze the outputs of the data sorter IP with the post-implementation simulation results to validate the IP design. Manual analysis of the output at each clock cycle is highly inefficient. It is worth highlighting that limited number of samples for each signal have been shown in the Figure 6.10 only to improve the visibility of the transitions. However, the DSAS can acquire un-limited number of such samples. Consequently, it is imperative to adopt an automated solution, such as rule-based inference system, to improve the time efficiency. In Figure 6.10, acquired debugging data has been plotted besides the simulation results (generated through HLV) from the knowledge base. Although, the received and the GR have been superimposed, still, it cannot be conclusively claimed that the two are identical. In order to find out the dissimilarities between the debugging trace and the simulation results, inference process is performed. The inference engine performs cross-correlation between the acquired debugging data and the dataset from the knowledge base and displays the results. Hence, if a mathematical model is not available or modeling is too time-consuming, HLV can be utilized for populating the knowledge base with the simulation results. This saves the time spent on generating the simulation model.

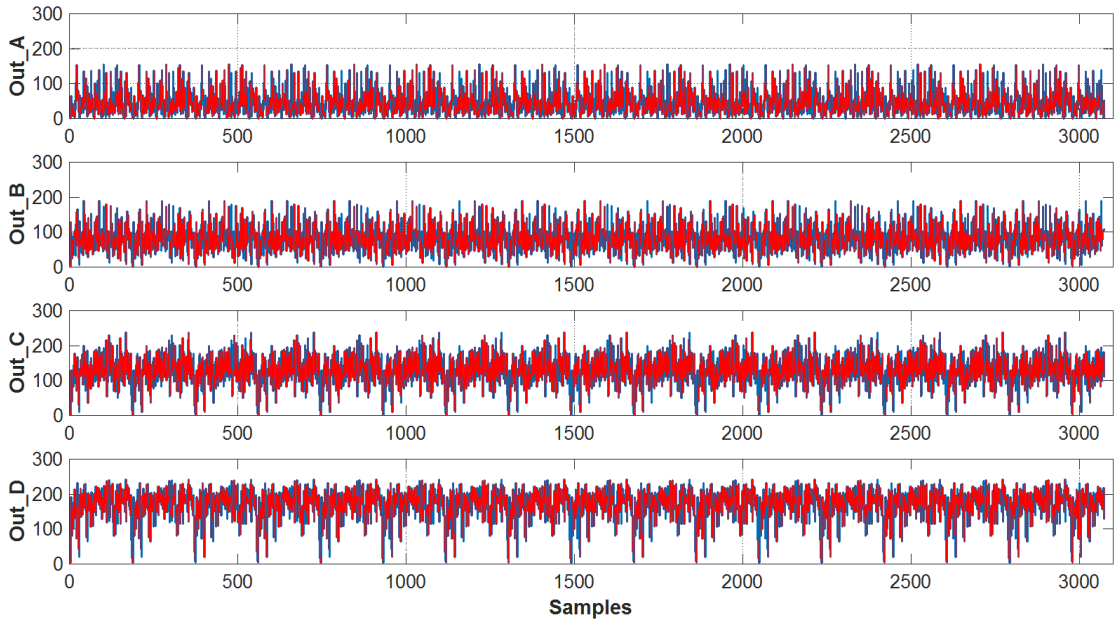


Figure 6.10: Debugging data after applying rule-based inference system

The output of the rule-based inference system output is plotted in Figure 6.11. The cross-correlation between the post-implementation simulation results and the debugging data from the hardware IP has been performed. At lag 0, the correlation coefficient between the two results is 1.0 which indicates a perfect match between the debugging data and the simulation results. This also highlights that the data sorter hardware IP matches perfectly with its simulation results. Hence, it can be concluded that the hardware is free from permanent or intermittent errors.

However, the plots also indicate a potential design weakness. As is evident, the plots have certain peaks, repeating after regular interval. Consequently, it indicates that the pseudorandom number generators replicate the number generation process after fixed number of cycles. In practical scenarios, it can be a potential weakness in the design required to be addressed.

From the above two experiments, it is evident that the inference system can be used for decision making without going through all the data, resulting in a time-efficient solution. In the presence of an error, either permanent or intermittent, the cross-correlation coefficient will have a value less than 1. In such case, the inference system indicates the lag against the maximum cross-correlation coefficient. This provides a starting point for further investigation. Once, the maximum correlation against the lag is found, the inference system synchronizes the two datasets with each other. It then points out all mismatch instances between the two datasets which are the errors in this case.

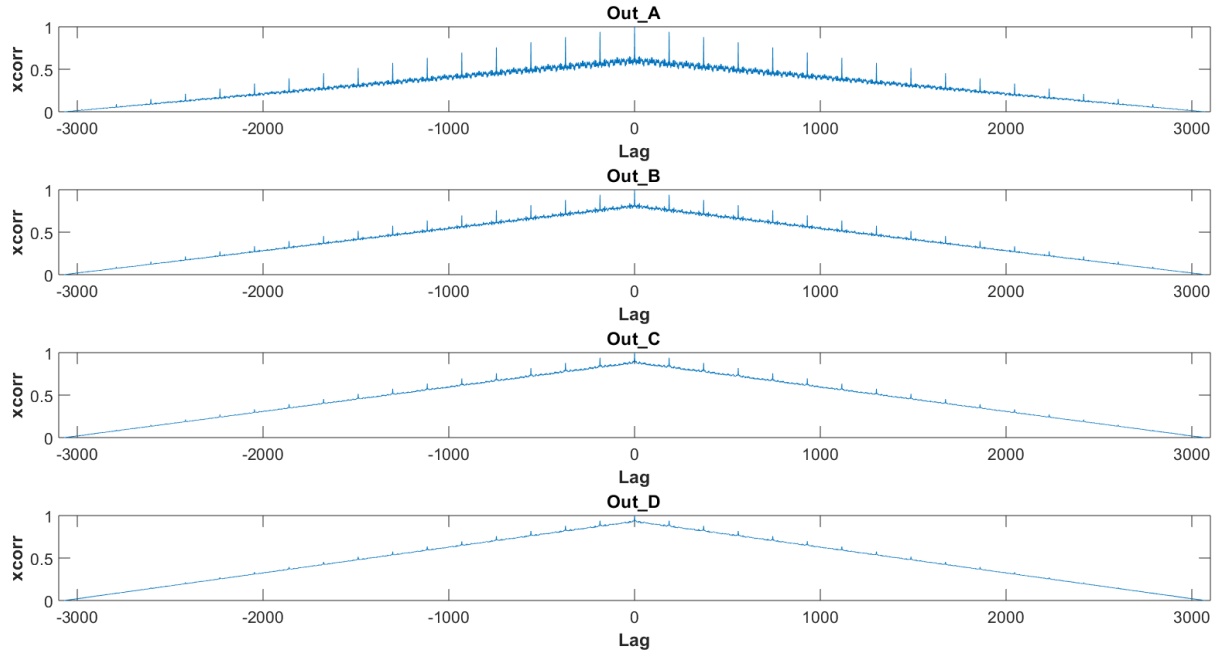


Figure 6.11: Cross-correlation between the simulated and actual results

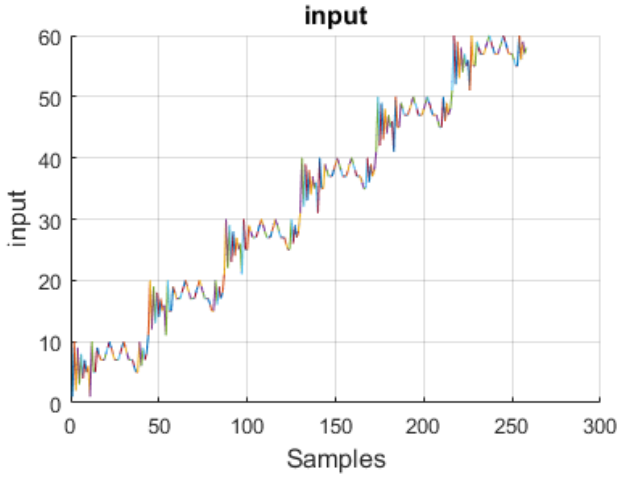


Figure 6.12: Arbitrary generated data

6.2.3 Error Detection in the absence of GR

The debugging is possible even if the GR is not available. In the absence of a template for comparison, the rule-based inference system attempts to generate a template based upon the received debugging data. In order to test this particular scenario, the GR for data sorter was removed from the knowledge base. Upon realizing the absence of a GR, the inference system switched to *autocorrelation rule*. The acquired data from the DUT was utilized to perform autocorrelation. Then, the algorithm mentioned in section 6.1.3 was followed to generate the GR. The generated GR was utilized to perform cross-correlation. The results from *autocorrelation rule* were identical to the ones as displayed in Figure 6.11. This experiment demonstrated that the inference engine was able to successfully generated the GR. Hence, it can be concluded that the proposed methodology can be used for debugging designs which do not have a GR.

In order to investigate further, an arbitrary stream of data was generated as presented in Figure 6.12. Since, the data was arbitrarily generated, its GR was unavailable. When the inference system remained unable to find the GR in the database, it switched to *autocorrelation rule*. The autocorrelation of the signal is displayed in Figure 6.13. It is evident that the autocorrelation is in-conclusive due to the absence of the peaks. Consequently, the rule was again switched to *linear regression*. Initially, linearization was performed for finding the slopes of the complete dataset at each transition. Subsequently, autocorrelation was conducted for finding the randomness in the slope set of the generated data as shown in Figure 6.14. The data has multiple peaks clearly indicating that the data is not random. Then, the algorithm as mentioned in Figure 6.7 was applied to find the slope set between the peaks in the autocorrelation function. Finally, the GR was generated based upon the retrieved slope set and cross-correlation was performed to identify the errors.

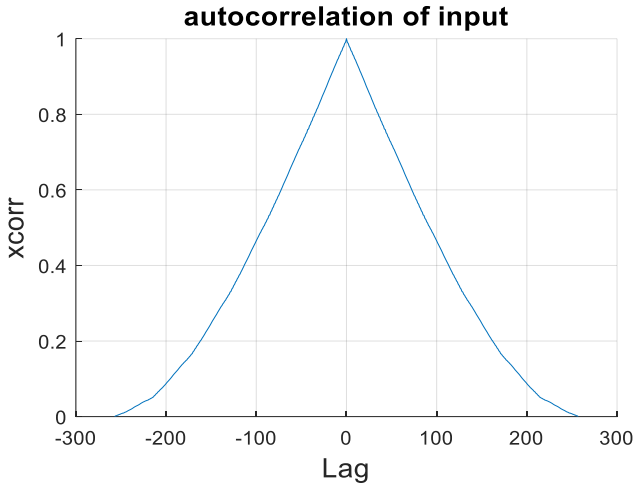


Figure 6.13: Autocorrelation of generated data

However, it is important to highlight one key point. The methodology works only if the received data trace contains recurring patterns and is highly correlated. In certain cases, permanent errors which occur due to design or logic errors, such as wrong implementation of a mathematical function, may also result in recurring patterns. Such permanent errors cannot be captured through the proposed solution. However, intermittent errors which may occur due to electromagnetic interference, manufacturing imperfections or operating conditions can be captured.

6.2.3.1 Resource Utilization

The resource utilization for the proposed debugging system is presented in Figure 6.15. The results have been compared with the FIL [218]. As can be seen, the resource consumption of registers in [218] is less than the proposed solution. However, for most designs, registers are not the limiting factor e.g. in Zedboard,

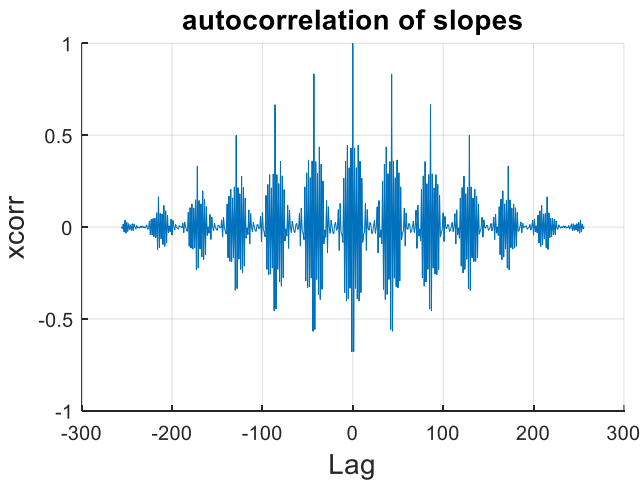


Figure 6.14: Autocorrelation of slopes of the generated data

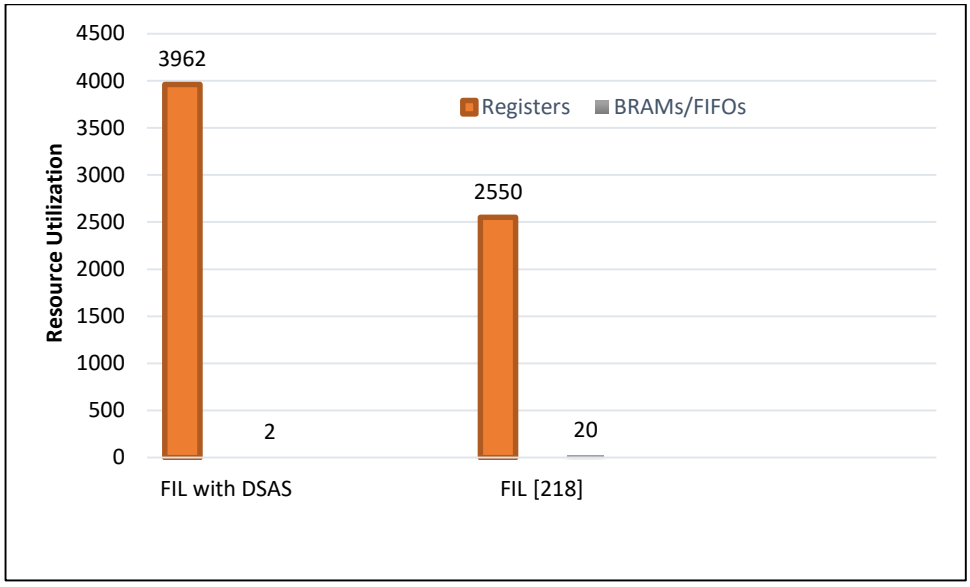


Figure 6.15: Resource utilization

there are 106400 registers available. Hence, the difference in the register consumption between FIL with DSAS and [218] is about 1%. In terms of resource utilization, the main limiting factor are BRAMs. In case of BRAM/FIFO, we have achieved better resource utilization.

6.3 Summary

FPGA-based embedded designs need to be verified. A suitable debugging system can save a lot of time and effort. Besides, if a hardware-software debugging approach is utilized, debugging can become easier. Such hardware-software solution is presented in this chapter which proposes to utilize inference for decision making. A rule-based inference system was proposed which can find the permanent or intermittent errors by performing cross correlation between the lossless debugging data and the GR. It eliminates the need for human interpretation required for monitoring the debugging data on every clock transition by performing a correspondence analysis for identifying the relationship between the input data and the GR.

However, sometimes it is more useful to perform FPGA-in-the-loop for finding errors. The presented approach was found useful for DUTs which utilize a stimulus/response cycle. The proposed work showed that debugging can still be performed by using behavioral, post-synthesis or post-implementation simulation results if a MATLAB/Octave model is not present. Moreover, in the absence of GR, the proposed rule-based inference system can be used to identify the errors by finding the recurring patterns in the received debugging data and subsequently using the pattern for generating the GR.

The main limitation of the methodology comes into question when the received trace data does not contain recurring patterns and is not correlated e.g. the data is highly random. In such cases, the methodology will not be able to generate the GR and hence will not function as intended.

Chapter 7 Automated Debugging using Artificial Intelligence

The visibility can be increased by the inclusion of scan or trace-based techniques. Still, the manual analysis of the massive amount of trace data is not time-efficient. During the post-silicon debugging process, the following two scenarios are of utmost concern:

- Diagnostic of the design to verify how closely it resembles with any available (GR).
- Design debugging, in the absence of any reference behavior for the Design Under Test (DUT), to identify the root cause of the error by identifying the cycle and the debugged design signals responsible for defect occurrence.

In this chapter an intrusive debugging technique is presented which permits a cycle-accurate lossless debugging by managing the clock of the DUT. The proposed solution depends heavily on Recurrent Neural Network (RNN) based machine learning technique for trace diagnostics as well as for bug localization even in the absence of a GR which in many scenarios may not be available. This helps in automating the problem identification and hence will increase time-efficiency.

The main contributions of this chapter are:

1. Propose an RNN based debugging methodology for automating the debugging process in the post-silicon validation cycle when the GR is available.
2. Show that the presented debugging methodology for automating the debugging process still works in the absence of the GR.

7.1 Cycle-accurate Debugging by RNN

7.1.1 Design Methodology

The proposed debugging methodology utilizes DSAS which results in a lossless cycle-accurate debugging trace of the DUT as shown in Figure 7.1. The acquired lossless debugging data resembles a time data series. This time data series can be used to predict future samples by using RNN. Hence, this research work is based upon utilizing an RNN, running on the terminal (PC), to predict the future samples of the time data series which in this case is the cycle-accurate debug trace.

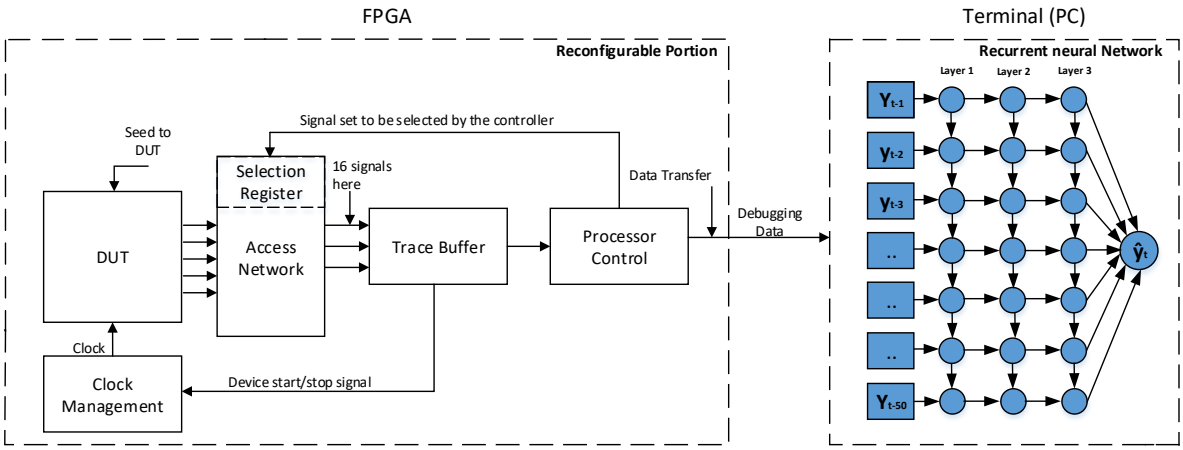


Figure 7.1: Debugging through RNN

Different factors like seasonality, trend, randomness etc. can affect the accuracy of the time series prediction. The authors in [183] studied the effect of such factors and pointed out that randomness has the biggest negative impact on prediction. They also highlighted that the accuracy decreased with an increase in the forecasting horizon. Keeping in view the above two factors and emphasizing upon the accuracy of predicted data being used in the debugging process, we laid the following conditions upon the trace data to be debug-able.

1. The data should not be random. This can be ensured by checking the randomness using the Wald-Wolfowitz run tests [219]. It is a non-parametric test which checks the hypothesis that a series of data is random. It finds the total number of runs along with the positive and negative values from a reference value. It then tests whether the number of positive and negative runs are distributed equally in time.

2. Trace data to be covariance stationary. This implies that the time series does not hold any hidden relationships between different time points and the behavior is stable. Stationarity can be ascertained by measuring the mean and the variance. A statistical test which gives a good insight into the behavior of the data is Augmented Dickey-Fuller test (ADF) [220]. ADF tests whether a unit root is present in a time series sample; the presence of unit roots indicates non-stationarity. The proposed methodology for debugging should be used if the training data is stationary. RNN can still be used for forecasting time series data even if the time series is not stationary. However, while using the RNN for debugging, it is a better approach to apply it for stationary trace data.

3. The forecasting horizon should be minimum. The prediction of future samples can be performed with the following two options

i. Train on $\{y_t, y_{t-1}, y_{t-2} \ldots\}$ to predict $\{y_{t+i,}\ for\ 1 \geq i \geq s\ holds\ for\ small\ s\}$

ii. Train to predict $\{y_{t+1}\}$, iterate to get $\{y_{t+i,}\ for\ any\ i\}$

In order to keep the forecasting minimum, the second option is followed as recommended in [183]. Furthermore, training and testing is performed on (80%, 20%) basis respectively.

The time-series prediction algorithm was implemented on the terminal (PC) by using RNN. RNNs are best suited for time series data prediction because of their ability to remember their inputs by making use of their memory. This enables the RNN to make precise predictions about the upcoming data. The methodology utilizes the sliding window method by using the data from previous 50 time steps to predict the value at the next time step. Using the previous samples, the trained RNN makes prediction for t+1 sample. Hence, the RNN can be represented by the $Eq.7.1.$

$$\hat{y}_t = \alpha_0 + \sum_{j=1}^{3} \alpha_j\, g\left(\beta_{0j} + \sum_{i=1}^{50} \beta_{ij} y_{t-i}\right) \qquad (Eq.7.1)$$

Here $y_{t-i,}\ where\ (i = 1,2, \ldots \ldots 50)$ are the time lagged inputs and $\hat{y}_t$ is the predicted output. α_j and β_{ij} are the connection weights. α_0 and β_{0j} are the bias terms. $g(x)$ is the activation function.

The error e_t between received y_t and predicted $\hat{y}_t$ at time t is:

$$e_t = y_t - \hat{y}_t \qquad (Eq.7.2)$$

As the RNN is highly trained and cross-validated, assuming $e_t = 0$, the received input sample should be:

$$y_t = \hat{y}_t \qquad (Eq.7.3)$$

which states that the predicted value is similar to the received sample and hence can be used to verify the received one. Similarly, the continuous stream of previous input samples can be used to predict the forthcoming sample iteratively.

However, when $e_t \neq 0$, it indicates that the received and predicted values do not coincide with one another highlighting the presence of a bug in either the trace data or the trained model itself. As the RNN is highly trained and the trace data observes the three conditions mentioned earlier, it can be assumed that the predicted value is correct thus highlighting the need to debug the suggested error at the indicated location. However, after analyzing the proposed error, if it is found that the trace data do not contain any error, it points towards the error in the trained model. The newly received error-free trace data can be used for further training of the RNN and subsequently the improved RNN can then be employed for debugging.

7.1.2 RNN Implementation

The proposed RNN comprises of 1-dimensional input layer, three hidden layers of sizes 50, 100, 50 and eventually a 1-dimensional output layer. Single input and output stages are used for the proposed methodology. However, the same idea can be extended to many to one or many to many neural networks as well. The last layer is the dense layer because it is a feedforward case. The proposed neural network is based upon the Keras library [221] using tensor flow as the back-end. A sequential model including stacked layers is used.

Mean squared error model is used to find the error during the forward propagation. The partial derivative of the error, adjusted through the back propagation process, is used by the Adam optimizer [222]. Adam is chosen because of its ability to converge faster as it "adapts" to the gradient and loss updates. An important hyper parameter is the learning rate. It controls how quickly or slowly the RNN will learn by determining the size of the step. During experimentation, different initial learning rates were tried. However, due to highly varying training dataset, 0.0001 was selected as initial learning rate as it gave a good approximation of the function. Exponential decay rates for the estimates of first and second moments i.e. (mean and variance) are chosen to be 0.9 and 0.999 respectively.

In order to cater for exploding gradient problem, ReLU is used as the squashing function. The squashing function does not allow the gradient to saturate which helps in resolving an abnormally high gradient. LSTM is utilized to include a memory to the neural network so that it can memorize its past states as well. The LSTM also solves the vanishing gradient problem. In order to improve convergence, normalization is performed on the data before feeding it to the RNN. The data is de-normalized before plotting because it is easy to correlate it with the input data. Number of epochs have a significant effect on the optimization of the RNN. For a big dataset, one epoch may be enough. However, for a debugging problem the training dataset is usually small. Hence more epochs may be needed for an optimized solution. This issue is discussed in detail in the results section.

The RNN processes the trace data for training. The RNN can be trained based upon a GR for the specific DUT. However, in the absence of a GR, the neural network can be trained from the received trace from the debugging system. It can use a portion of the data for training and validation. Hence, 5% of the training data was used for validation. Once the data has been segregated into the training and validation datasets, the datasets are divided into batches i.e. the number of samples required by the network to perform a parameters update. After experimentation, a batch size of 512 was found to produce good results.

However, as stated by [223], in some datasets the Adam optimizer do not generalize as good as some other optimizers available (such as Stochastic Gradient Descent (SGD) with momentum). In such cases, the loss

function after the use of optimizers may be compared with different alternatives to find out the best option to be used as an optimizer for the encountered dataset.

Moreover, sometimes, the loss function does not converge to a minimum defined value. In such cases, training with RNN using different optimizers and hyper parameters should be performed to find an optimized solution which results in a converging loss function. The trained neural network can then be used in the prediction phase. Once the RNN is trained, any new input data is validated by the trained network. The incoming test data along with the predicted data is presented to the user for view. Furthermore, manual analysis of the test results constitutes a bottleneck for cycle-accurate debugging systems due to the enormous lossless stream of data. A trained neural network is used in this research work which helps to localize potential bugs and draw conclusions to speed up the debugging process.

7.2 Experimentation and Results

The result section has been divided into three subsections. In the first subsection resource utilization for the proposed debugging system with unlimited trace window will be explained. In the second subsection, the results of debugging using the proposed methodology will be explained. In the third one, the training requirements will be described.

The proposed methodology has been tested on the Digilent Zedboard, which has an XC7Z020-484 FPGA. Xilinx Vivado 2017.1 is used for designing the hardware. The RNN portion of the debugging system was implemented on the terminal (PC) using Python programming language. The Python-based RNN was implemented on Intel Core i7-6700 CPU running at 3.4GHz and having 16GB of RAM.

7.2.1 Resource Utilization

Resource utilization of the presented debugging methodology is shown in Figure 7.2. Since the debugging system does not constitute the main part of the design, it is desired to restrict its hardware utilization. It was noticed that the resource utilization of the presented solution is growing with an increase in debug window, because more BRAM blocks are required as trace buffers. Hence, the debugging system is synthesized with a trace window of 64 samples. 16 signals are monitored with a data width of 32 bits each.

The resource utilization is compared with a similar research [Khan1]. It is evident that the resources have been reduced to almost 2% of the available ones. Consequently, the proposed solution can be applied to designs where availability for debugging resources is limited.

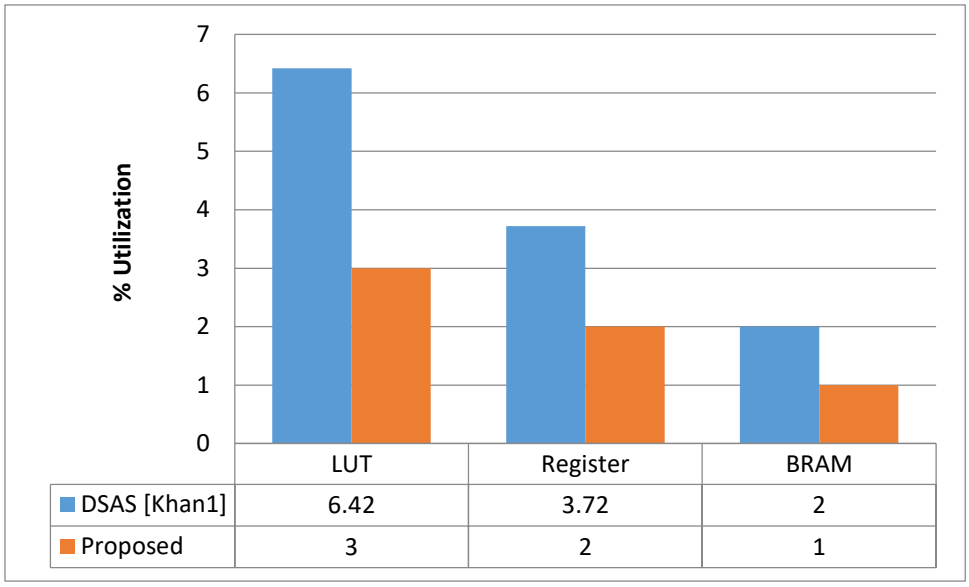


Figure 7.2: Resource utilization

7.2.2 Debugging through RNN

The Obstacle Avoidance (OA) system first mentioned in section 3.5.1.1 is used as use case for this research work. The OA algorithm was implemented in hardware. The controller, EKF, odometry calculation, comparison module etc. are some of the hardware implemented modules. In order to increase the visibility of the hardware, the debugging system as mentioned in section 7.1.1 was incorporated into the hardware design for debugging.

The robot in question was used for obstacle avoidance. During the course of its operation, it traversed a path in the x, y coordinates. The resulting orientation of the robot is shown in Figure 7.3. We will use the time series data of the orientation to illustrate the usage of proposed RNN for debugging. The lossless tracing data accumulated by the proposed debugging system resembles a time data series. Hence, an RNN can be used to ease the debugging process. If a GR is available, the debug trace from the GR can be used for training phase. Once trained, the debug trace data from the DUT can be predicted using the trained RNN which can then be used for bug localization. This methodology is explained in section 7.2.2.1.

However, in many cases, the GR is not available. In the absence of GR, the debug trace data generated by the DUT can be used for training. The portion of trace data known to be error free can be used to train the RNN. Then this trained RNN is used for debug data prediction. This methodology will be explained in section 7.2.2.2.

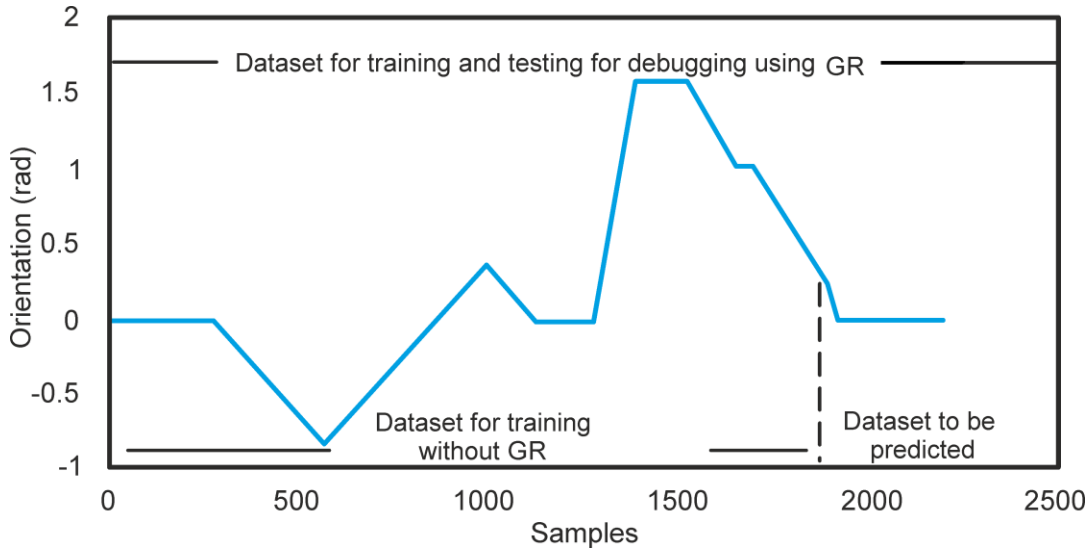


Figure 7.3: Orientation in radians

7.2.2.1 Debugging using GR

In the first experiment, we supposed to have a verified OA system which can be used to generate GR. First the robot was moved from the same start point and end goal and avoid the obstacle as shown in Figure 7.3. After data collection, normalization was performed during the data preparation phase. The data was then used to train the RNN. Subsequently, the trained RNN was used to predict the time data series. The predicted data is de-normalized so that it can be compared with the debug trace data.

Predicted time series data has been plotted against the actual time series data as shown in Figure 7.4. The red line shows the actual time series data and the olive green line in the figure shows the predicted data. As can be seen, the predicted data closely resembles the actual data.

7.2.2.2 Debugging without GR

During the last experimentation, the OA system was used to generate the GR and trained our RNN based upon the GR. However, in most cases a GR is not available for training. In such cases, the received continuous lossless debugging time series trace data can be used for RNN training. The trace data used for training should be error free. This can be ascertained by using hardware checkers as suggested by Bertacco et al. [224]. Furthermore, the received data for training should fulfill the conditions mentioned in Section 7.1.1. For the current experiment, we used about 1800 samples for training as indicated in Figure 7.3. The

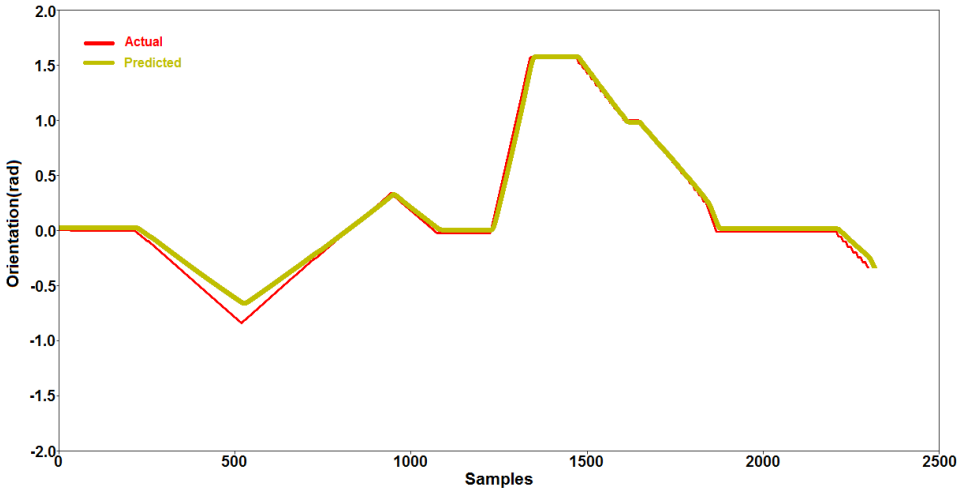


Figure 7.4: Actual vs predicted time series data

aim of the experiment was to let the RNN predict the received data after training. Then the trained RNN was utilized for prediction of about 300 actual incoming samples. This predicted data sample is compared with the received trace data for bug identification as shown in Figure 7.5. The actual received trace is shown in red color while the predicted data is shown in olive green.

It can be seen that the RNN was able to predict the incoming data quite efficiently. After about 30 samples, the predicted data mismatched with the actual data for few samples. However, the RNN was still able to identify the trend. Then, it kept on predicting the incoming data exactly. As evident from the figure, the RNN has been able to predict the incoming data after training from the received data. This eliminates the

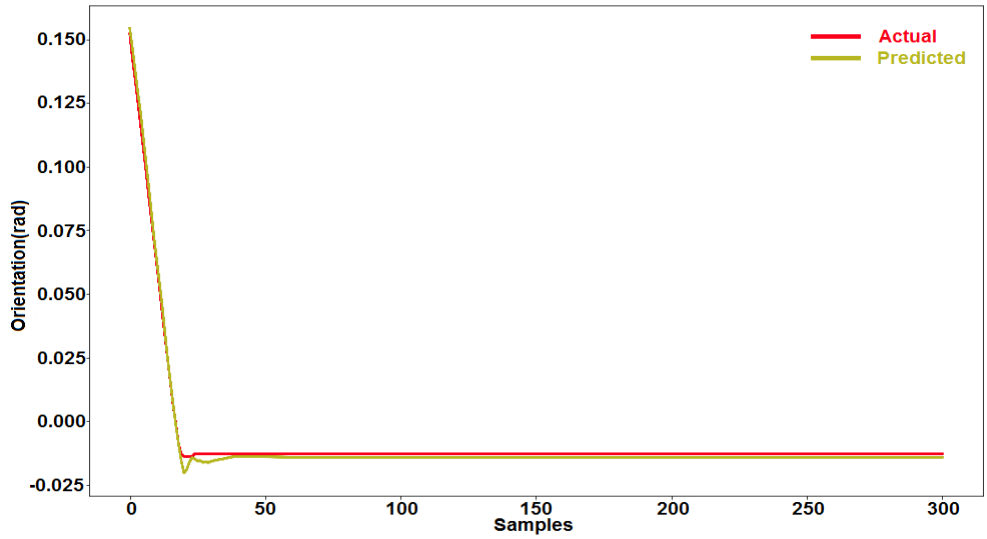


Figure 7.5: Debugging without GR

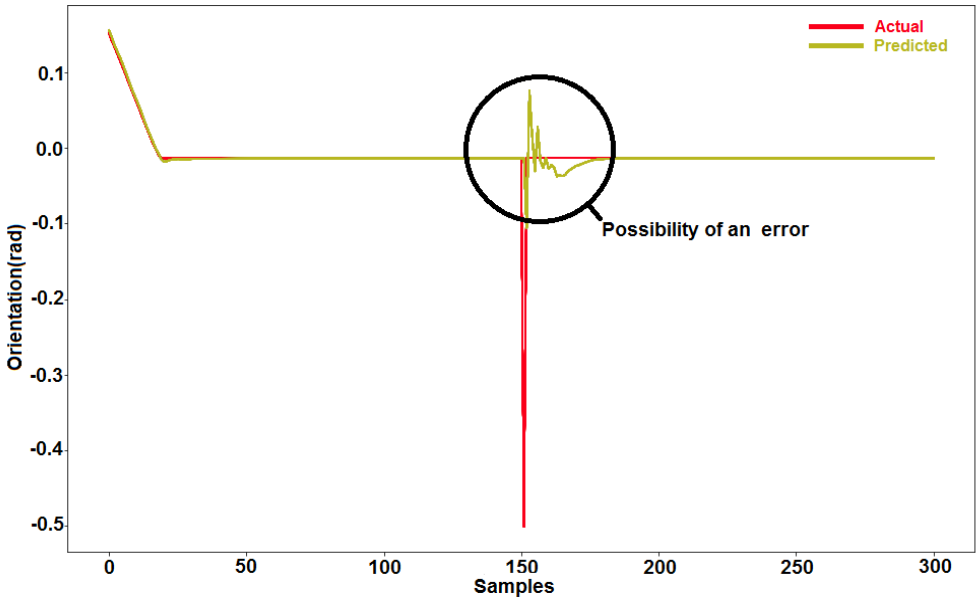


Figure 7.6: Debugging with introduced bug

requirement of GR and lets the user to debug the DUTs once they have been trained from incoming error-free data.

The error normally resembles rarely occurring behavior i.e. intermittent or outlier for which the neural network is not trained. Consequently, the RNN fails to follow the trend of such outlier. Such outliers (bugs) can then be easily isolated during the debugging phase. In order to illustrate the effectiveness of the proposed approach, we induced a self-created error into the test data as depicted in Figure 7.5. As, the RNN is trained based upon the error free received data, it was able to predict well during the test phase. However, since the RNN was not trained for the rarely occurring induced error, it failed to predict the test data during the occurrence of error. This phenomenon is illustrated in Figure 7.6 by a possible error. If we compare the response of the RNN when encountered with this self-induced bug to the previous illustration of sudden data variation, it becomes evident that RNN was able to follow the trend shortly after the occurrence of sudden data variation because it was trained for such variation. However, the RNN utterly failed to predict the data during the presence of self-induced bug because the RNN was not trained for the bug and hence was unable to predict it. This can be witnessed from the abrupt response of the RNN during the occurrence of self-induced error. Whenever an error is encountered during the test phase, a similar behavior is expected. Consequently, this phenomenon can be utilized in bug identification. Intermittent errors or bugs, resembling an outlier behavior, can be easily identified using the presented RNN approach.

7.2.3 Training Requirement

7.2.3.1 Training Dataset

The most important condition for training an RNN and later use it for testing is the provision of the training data. The error in a DUT is considered to be random i.e. it can occur at any time. Hence, a large dataset for training may not be available. We considered this scenario quite reasonable because if training with small dataset can be performed, training with large datasets will be definitely more accurate (assuming similar variance in both datasets). In the presented use case, training was performed with 1800 samples only. The data between training and testing was split as (80%,20%) respectively which resulted in predicting about 300 samples for the dataset in question.

7.2.3.2 Time Requirement

An important consideration during the experimentation was the training time required by the RNN. During the course of experimentation with the OA use case, the loss function was found to be quite high because of small dataset for training. This issue was resolved by increasing the number of epochs. We performed the experimentation with different epochs, each time using the trained RNN for data prediction as shown in Figure 7.7. It was found out that the increase in the epoch have a positive effect on the data prediction.

The training time taken by the RNN for different epochs is shown in Table 7-1. It is evident that the increase in the number of epoch results in a decrease in loss function while increasing the training time. Another point worth mentioning is that initially the loss function decreases quite rapidly with an increase in the number of epochs. Subsequently, the increase in the number of epochs has very nominal effect on the loss.

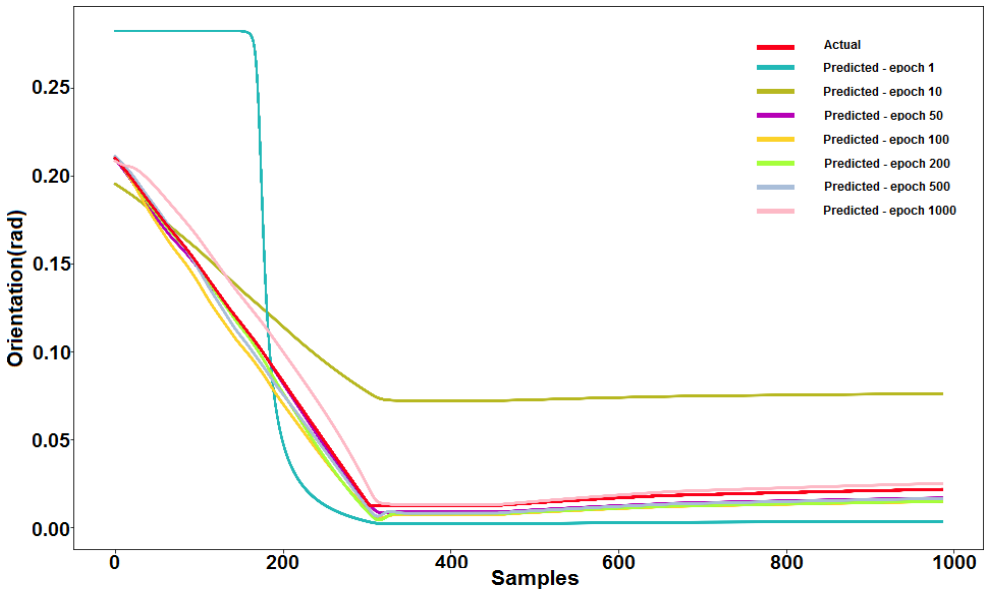


Figure 7.7: Effect of increase in epochs on the data prediction

It was found out that epoch 50 or 100 gave a good prediction of the OA system with small training dataset. However, if big dataset is available for training, less number of epochs are needed. In our results in section 7.2.2, we used 100 epochs for the generation of results.

Table 7-1: Training time

Epoch	Training time (s)	Loss function
1	9.9	0.9179
10	45.5	0.1779
50	210.8	0.1648
100	516.0	0.1642
200	1159.8	0.1640
500	3115.6	0.1637
1000	6947.9	0.1635

7.2.3.3 Bug Localization

The proposed methodology can be iterated for bug localization. Once a bug has been identified in a specific node in a DUT, the remaining nodes of the DUT in its vicinity can be probed further to localize the bug.

7.3 Summary

This research work presents a cycle accurate debugging technique using RNN for trace diagnostics as well as for bug localization. Our proposed solution depends heavily on machine learning techniques for trace diagnostics as well as for the bug identification. Results have shown that localizing potential bugs in the system can be done with or without the presence of a GR. The methodology will help in problem identification and hence will increase time-efficiency.

One limitation of the proposed methodology is that it is valid for stationary data. This limitation restricts the proposed work to DUTs whose output is not derived from continuously changing inputs. Although this restricts the scope of the work, using the predicted data for debugging needs to be error free which can be ensured through the stationarity of the data. Another limitation of the proposed solution is that it is valid for non-random data. Hence, the technique should not be applied to random or near random data. These conditions restrict the debugging solution to certain data types. If trace data does not fulfill the above mentioned conditions, the proposed solution should not be applied. However, the problem occurs when some of the conditions are fulfilled and the others do not. In such cases, it remains an intelligent guess to adopt the presented methodology for debugging.

Sometimes, the presented methodology suffers from a limitation that the loss function does not converge to a minimum defined level (say 0.2). In such cases, different optimizers and hyper parameters can be tried

to train the RNN for an optimized solution. The proposed solution should be used if the loss function has been converged to the minimum defined level during training.

One point is worth highlighting. The methodology works irrespective of the presence of GR. When the GR is present, the RNN can be trained and later used to predict all type of errors such as permanent errors or intermittent errors. However, when the GR is not available, the methodology can be used to capture intermittent errors or outliers.

Chapter 8 Conclusion and Future Work

This thesis entails five contributions that together provide an automated solution for error detection in FPGA-based designs. In this concluding chapter, the main challenges of this research work are re-iterated along with the summary of the individual contributions, and their significance. In the last section of this chapter, limitations of this research work along with future research directions are discussed.

8.1 Conclusion

Due to continuous rise in the number of hardware resources, the design verification is becoming increasingly difficult for FPGAs. Pre-silicon techniques such as behavioral simulation have been found to ensure the functional verification. However, comprehensive verification coverage is not possible due to speed limitation of simulation-based verification methodology.

For the reasons mentioned above, focus shifted towards post-silicon prototyping for design validation. Post-silicon validation results in speeding up the design verification since designs can run many orders-of-magnitude faster when implemented on FPGAs. However, the main challenge faced in post-silicon phase is the limited observability. One of the main solutions to enhance the visibility of on-chip hardware is to insert trace-buffers into the design. This results in logging a limited number of signals for limited number of clock cycles on the trace-buffers. In order to receive the data again, these trace buffers need to be triggered once again after being emptied through data transfer for off-line analysis. But the real-time embedded design is on the run resulting in loss of trace data. All errors which occur prior to trace buffer triggering are lost. These errors sometimes referred to as intermittent errors can be extremely problematic.

Due to resource limitation, limited number of signals can be monitored for limited number of clock cycles. All these factors result in limited observability which restricts the debugging capability. Another important issue is the manual analysis of trace data. For debugging systems which provide lossless trace data, manual analysis of the trace data can be too time consuming which may hamper the debugging process. Furthermore, intermittent errors are a one-off event. Finding such an error in lossless trace logs manually is not time-efficient.

The main contribution of Chapter 3 is a debugging system capable of providing a lossless cycle-accurate trace of debugging data. Lossless data can be captured with minimal resources in contrast to other solutions which are either cost-inefficient or require on-chip hardware resources. Such solutions are not practical for smaller FPGAs due to the absence of the required on-chip hardware. Hence, we proposed to stop the clock of the DUT during the data transmission phase and re-start it once the data has been transferred and the trace-buffers are ready to receive the data from 16 signals with very small trace buffers (only 4KB).

Although, limited number of signals are sufficient for small designs. However, often the designs are complex having thousands of signals. Observability enhancement of such designs is also important.

In order to increase the observability, we introduced a processor configured access network. We presented two solutions with a multiplexer-based approach and a gate-based approach. The multiplexer-based approach is more resource demanding however it offers more flexibility. However, if the design is short in resources, gate-based access network can also provide the desired observability. All the desired signals can be connected through the access network to the debugging system. From the available ones, 16 signals are selected heuristically through the selection register. Since, the error-prone signals are not known, we may keep on iterating on the signals without being able to capture the error.

Another contribution is the software-based data compression. For DSAS, the bottleneck lies in the data transmission. Hence, we proposed to compress the data before Ethernet transmission. This results in speeding up the data transmission. It may be noted that the same results could have been achieved through hardware-based data compression. However, the compression-core requires hardware resources making the design only feasible for resource-available designs. However, the proposed data compression solution is suitable for resource deficient designs.

Preservation of timing closure is another challenge. The main technique for embedded system design is that it is synthesized and implemented to find any errors. If an error is encountered, the design is synthesized and implemented after inserting the debug circuitry. However, the debugging circuitry uses the same resources as the embedded design which may affect the timing closure. After debugging, the debugging circuitry need to be removed from the design which may again affect the timing closure. We resolved this issue through DPR. We synthesized and implemented the debugging system as a dynamic partition at design time. If needed, the debugging system can be reconfigured at run-time. After design finalization, the blank bitstream can be left as an integral part of the design. This helps in preservation of timing closure. Furthermore, when the dynamic debugging partition is not in use, the resources are free to be used by other applications.

The main contribution of Chapter 4 is the debugging system capable of debugging multiprocessor and multi-clock designs. Stopping the clock can be problematic for multi-clock systems since this may result in data-invalidation. We presented a synchronizer-based design which can manage clocking of the multi-clock designs using GRLS and GALS methodology. This solution takes a cycle-based approach for debugging. However, another debugging paradigm is the event-based debugging. Such systems record data only when an event takes place. We also provided a solution for such designs for the sake of completeness though it is not the main problem tackled in this thesis. This solved the problem of limited clock cycles. However, the limited number of signals were addressed in Chapter 4.

In Chapter 5, we tackled the observability issue from another angle. Although the observability can be enhanced using an access network, the signals required to be connected are selected heuristically. Since, the error-prone signals are not known, we may keep on iterating on the signals without being able to capture the error. A better approach would be to identify the signals with more restoration probability of errors and map them to the access network on priority, so that we may not be wandering around to capture the error. The problem was resolved through the inference system which checks the presence of correlation in the trace data and prioritize signals accordingly.

In Chapter 6, two problems were encountered. The main issue of lossless debugging systems is to propose a technique for capturing the errors without going through manually analyzing the debugging data on every clock transition. Another issue is to find a methodology for finding the intermittent errors when a GR is not available. In order to resolve the mentioned problems, automated error detection through a rule-based inference system was proposed. The inference system performs correspondence analysis to draw conclusions. For DUTs which work on stimulus/response phenomenon, FIL-based debugging was also proposed. The debugging system generates a stimulus for the DUT. The response of the DUT is used to perform the correspondence analysis between received debugging data and the HDL simulation data from either a behavioral model or a post-synthesis/post-implementation model of the DUT used as a GR.

Another issue encountered in this chapter was debugging in the absence of a GR. We devised a methodology which can search for correlation in the received debugging data and can generate its own GR. Once the GR has been generated, the rule-based inference system can perform correspondence analysis between the received data and the generated GR. The proposed methodology can not only identify the errors but it can also identify the location of error generation and the clock cycle.

In Chapter 7, debugging of embedded designs by using recurrent neural networks was proposed. Lossless trace data presents a unique opportunity since it can be used as time series data. Recurrent neural networks have been found to be very useful in time-series data prediction. An RNN-based debugging methodology was suggested for automating the debugging process. The proposed solution depends heavily on machine learning techniques for trace diagnostics as well as for the bug localization. Results have shown that localizing potential bugs in the system can be done with or without the presence of a GR, which in many scenarios may not be available. This will help in problem identification and hence will increase time-efficiency.

8.2 Future Directions

In this part of the thesis, the future directions of the current research work are discussed.

In Chapter 3, DSAS debugging system for lossless debugging was proposed. One point worth highlighting is that the proposed and the ILA-based debugging approaches are not mutually exclusive. Hence, DSAS can be used in conjunction with ILA. If the designer has complete understanding of the design and can set the ILA to trigger appropriately and is able to debug by using the limited window, ILA gives a very good solution. But in cases where FPGA resources are limited or debugging becomes difficult due to size limitations, DSAS can be used for debugging of complex designs. As a future work, a trigger circuitry can also be augmented with DSAS which can be used in a specific debugging problem.

Another issue is that trace buffers depth can be variable which is decided by the user. The depth directly affects the debugging time. Debugging time decreases with the increase in trace buffer depth. Hence, an automated solution using intelligent methods which can help in deciding an optimum trace-buffer depth can be quite interesting.

Sometimes, the modules need to be debugged while receiving external streaming data as input. A solution to this problem has been addressed through the inclusion of synchronizers. However, if the external devices do not follow a streaming protocol, the proposed solution may not work. A similar problem is faced when the DUT cannot be halted. In such cases, event-based methodology as already presented can be utilized. Another solution could be to use I/O pins for transmission. In such systems, few I/O pins are dedicated solely for debugging and the debugging trace data is sent to the terminal through such dedicated pins as is done for some emulation systems.

An access network was proposed to enhance the observability. The main limitation of the access network is that it also requires hardware resources for implementation. As the number of spare hardware resources are not known before design implementation, the optimized access network is not possible. Debug overlays can be utilized as possible solution in such cases.

In order to resolve data-transmission bottleneck, software-based data compression was proposed. The main benefit in using software-based compression is that hardware resources are not required. However, it requires processing time which may not be negligible. Furthermore, the Msim-9 algorithm proposed in this research work may not be able to compress data for certain data types. In such cases, the time required to compress the data goes in vain.

Similarly, debugging through DPR-based incremental insertion was also proposed. It is meant to utilize hardware resources when debugging is not required and preserving the timing closure. However, upon design finalization, the blank bitstream needs to be part of the finalized design in order to preserve the

timing closure. The blank bitstream consumes less power than the reconfigurable module partial bitstream, however, it still dissipates power which must be kept in mind before leaving it as part of the design [201].

In Chapter 4, DSAS-based debugging solution was utilized for embedded processors. It has been found that the solution can be used to troubleshoot bugs in complex SoCs where it is difficult to identify issues in the absence of a continuous stream of lossless data. As the methodology is based upon debugging external interfaces and nodes, all memory accesses can be recorded which can be used to perform software reconstruction.

In the same chapter, we proposed event-based debugging for NoC. The system is divided into two main parts, an on-chip debugging hardware and an off-chip debugging software. The on-chip hardware is based on the idea of debugging all connections through hardware monitors. With an increase in NoC sizes, more resources are utilized by the NoC, leaving aside limited resources for the debugging system. Hence reducing the resources required for the debugging system is extremely important. The first way to decrease this overhead is to avoid using the AXI stream interconnect and replace it with a custom interconnect that should be designed for this specific purpose. Another solution could be to avoid the use of the internal FIFOs of the AXI stream interconnect and replace them with a pipelined architecture which could perform the same role with less resource. The hardware could also be improved to add different abstraction levels. For example, the system could monitor the payload being sent beside the start and end of a packet. Some information could be collected such as the monitoring number of flits inside a packet and other information that could be needed for a specific application. All those functionalities could be added without major changes in the hardware.

Similarly, software debugging application can also include a fault detection algorithm such as deadlock or livelock detection algorithms. Those algorithms can make use of the data already being collected and then the routers can be configured to solve the problem in real-time. This would make the proposed debugging system as a self-healing system. Real time debugging and fault detection could be also added to the system. This could be achieved by adding a new functionality to the debugging software: stop or pause some PEs in the system. This could be done by sending a request to the ARM processor which could then send an interrupt signal to the targeted PE forcing it to stop executing its code and to wait for resume permission.

In Chapter 5, signal priority-based connectivity mapping was proposed. The connectivity generation tool accepts input files in several formats and generates a Tcl script which can be used to automate the connection process. The methodology uses the simulation results to identify the signals which can be prioritized through the proposed pattern extraction technique. The connectivity generation tool works well for simple hierarchy. However, one key limitation is that when the hierarchy becomes complex due to

several levels of parent child relationship; the tool fails to respond. As a future work, the tool need to be made generic for complex hierarchies.

In Chapter 6, rule-based inference system was proposed which can find the errors by performing cross correlation between the lossless debugging data and the GR. It eliminates the need for human interpretation required for monitoring the debugging data on every clock transition by performing a correspondence analysis for identifying the relationship between the input data and the GR.

Similarly, FIL-based debugging was proposed to capture permanent or intermittent errors which work on stimulus/response cycle. The debugging system software environment generates a stimulus and waits for a response from the DUT which is then analyzed by rule-based inference system against a GR for its correctness. The methodology also works in the absence of GR. It was demonstrated that errors could still be identified by finding recurring pattern in the received debugging data and using the pattern for generating GR which could then be used for debugging. However, the methodology works for signals with strongly correlated data. The main limitation of the methodology comes into question when the received trace data is not correlated e.g. random data. The methodology will not be able to generate the GR in such cases and hence will not function as intended. It is a challenge to propose algorithms for signals which do not exhibit strong correlation in itself.

In Chapter 7, debugging framework using recurrent neural network was presented which permits a cycle-accurate lossless debugging by using RNN based machine learning technique for trace diagnostics as well as for bug localization. However, one limitation of the proposed methodology is that it is valid for stationary data. Another limitation of the proposed solution is that it is valid for non-random data. Hence, the technique should not be applied to random or near random data. Such conditions restrict the proposed solution to certain data types. If trace data does not fulfill the above mentioned conditions, the proposed solution should not be applied. However, the problem occurs when some of the conditions are fulfilled and the others do not. As a future work, it sounds interesting to analyze the effect of such conditions on data prediction.

Sometimes the loss function does not converge to a minimum defined level during the training phase. In such cases, different optimizers and hyper parameters need to be tried to train the RNN for an optimized solution. Work can also be done to identify the best set of optimizers and hyper parameters for an optimized solution.

Bibliography

[1] M. H. Rashid, “Electronics-what should we teach and why?,” in International Conference on Electrical & Computer Engineering (ICECE 2010), 2010, pp. 66–69.

[2] G. E. Moore and others, “Cramming more components onto integrated circuits.” McGraw-Hill New York, NY, USA:, 1965.

[3] Y. Taur and T. H. Ning, Fundamentals of modern VLSI devices. Cambridge university press, 2013.

[4] “Mentor Graphics. ModelSim: ASIC and FPGA Design.” [Online]. Available: http://www.mentor.com/products/fv/modelsim/. .

[5] R. Kaivola et al., “Replacing Testing with Formal Verification in Intel CoreTMi7 Processor Execution Engine Validation. 2010.” .

[6] S. Asaad et al., “A cycle-accurate, cycle-reproducible multi-FPGA system for accelerating multi-core processor simulation,” ACM/SIGDA Int. Symp. F. Program. Gate Arrays - FPGA, pp. 153–161, 2012.

[7] “International Technology Roadmap for Semiconductors (ITRS). International Technology Roadmap for Semiconductors, 2007 Edition.” [Online]. Available: http://www.itrs.net/Links/2007ITRS/2007 Chapters/2007 Design.pdf, February 2008.

[8] H. Krupnova, “Mapping multi-million gate SoCs on FPGAs: industrial methodology and experience,” in Proceedings of the conference on Design, automation and test in Europe-Volume 2, 2004, p. 21236.

[9] H. Foster, “Challenges of Design and Verification in the SoC Era.” [Online]. Available: http://testandverification.com/files/DVConference2011/2 Harry Foster.pdf.

[10] “Vennsa Technologies. OnPoint.” .

[11] S. Y. L. Chin and S. J. E. Wilton, “An analytical model relating FPGA architecture and place and route runtime,” in International Conference on Field Programmable Logic and Applications, 2009, pp. 146–153.

[12] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch, “Methods for fault tolerance in networks-on-chip,” ACM Comput. Surv., vol. 46, no. 1, p. 8, 2013.

[13] N. Kitchen and A. Kuehlmann, “Stimulus generation for constrained random simulation,” in Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design, 2007, pp. 258–265.

[14] “Xilinx Inc. ‘Zynq-7000 SoC Data Sheet: Overview’. User Guide DS190 (v1.11.1).” [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf .

[15] “Xilinx Inc. ‘ MicroBlaze Processor Reference Guide’ . UG984 (v2018.2).” [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf.

[16] W. Lie and W. Feng-Yan, “Dynamic partial reconfiguration in FPGAs,” in Third International Symposium on Intelligent Information Technology Application, 2009, vol. 2, pp. 445–448.

[17] B. Vermeulen and K. Goossens, “Interactive debug of socs with multiple clocks,” IEEE Des. Test Comput., vol. 28, no. 3, pp. 44–51, 2011.

[18] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, “A reconfigurable design-for-debug infrastructure for SoCs,” Proc. - Des. Autom. Conf., pp. 7–12, 2006.

[19] Y. C. Hsu, F. Tsai, W. Jong, and Y. T. Chang, “Visibility enhancement for silicon debug,” Proc. - Des. Autom. Conf., pp. 13–18, 2006.

[20] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in Annual International Cryptology Conference, 1999, pp. 388–397.

[21] D. J. Bernstein, “Curve25519: new Diffie-Hellman speed records,” in International Workshop on Public Key Cryptography, 2006, pp. 207–228.

[22] A. Moradi and T. Schneider, “Side-channel analysis protection and low-latency in action – Case study of PRINCE and midori,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2016.

[23] “Xilinx Inc. Tabula. 7 Series FPGAs Configurable Logic Block: User Guide467 (v1.8).” [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf, September 2016.

[24] J. M. Chabloz and A. Hemani, “A GALS network-on-chip based on rationally-related frequencies,” in Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2011.

[25] M. Moradi, B. Van Acker, K. Vanherpen, and J. Denil, “Model-Implemented Hybrid Fault Injection for Simulink (Tool Demonstrations),” in Cyber Physical Systems. Model-Based Design, Springer, 2018, pp. 71–90.

[26] C. A. Mack, “Fifty years of Moore’s law,” IEEE Trans. Semicond. Manuf., vol. 24, no. 2, pp. 202–207, 2011.

[27] “Mentor Inc. The 2018 Wilson Research Group ASIC and FPGA Functional Verification Study.” [Online]. Available: https://www.mentor.com/products/fv/events/the-2018-wilson-research-group-asic-and-fpga-functional-verification-study.

[28] J. Keshava, N. Hakim, and C. Prudvi, “Post-silicon validation challenges: How EDA and academia can help,” in Design Automation Conference, 2010, pp. 3–7.

[29] S. Tasiran and K. Keutzer, “Coverage metrics for functional validation of hardware designs,” IEEE Des. Test Comput., vol. 18, no. 4, pp. 36–45, 2001.

[30] B. Keng, S. Safarpour, and A. Veneris, “Bounded model debugging,” IEEE Trans. Comput. Des. Integr. Circuits Syst., vol. 29, no. 11, pp. 1790–1803, 2010.

[31] S. Teig, “Programmable logic devices in 2032?,” 2012.

[32] “Intel Inc. Quartus II Handbook Version 12.1 Volume 3: Verification.” [Online]. Available: http://www.altera.com/literature/hb/qts/qts qii5v3.pdf, November 2012. [Accessed: 20-Sep-2005].

[33] “Tektronix Inc. Simplifying Xilinx and Altera FPGA Debug.” [Online]. Available: https://www.tek.com/application-note/simplifying-xilinx-and-altera-debug.

[34] K. Morris, “On-chip debugging—Built-in logic analyzers on your FPGA,” 2004.

[35] B. Caslis, “Effectively Using Internal Logic Analyzers for Debugging FPGAs. FPGA and Structured ASIC Journal.” [Online]. Available: http://www.eejournal.com/archives/articles/20080212 lattice/, February 2008.

[36] P. Fogarty, C. MacNamee, and D. Heffernan, “On-chip support for software verification and debug in multi-core embedded systems,” IET Softw., vol. 7, no. 1, pp. 56–64, 2013.

[37] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, “Using design-level scan to improve FPGA design observability and controllability for functional verification,” in International Conference on Field Programmable Logic and Applications, 2001, pp. 483–492.

[38] Y. S. Iskander, C. D. Patterson, and S. D. Craven, “Improved abstractions and turnaround time for FPGA design validation and debug,” in 2011 21st International Conference on Field Programmable Logic and Applications, 2011, pp. 518–523.

[39] D. Levi and S. A. Guccione, “BoardScope: A debug tool for reconfigurable systems,” Config. Comput. Technol. its uses High Perform. Comput. DSP Syst. Eng., pp. 239–246, 1998.

[40] S. Guccione, D. Levi, and P. Sundararajan, “JBits: A Java-based interface for reconfigurable computing,” in 2nd annual military and aerospace applications of programmable devices and technologies conference (MAPLD), 1999, vol. 261, pp. 1–9.

[41] “Sandbyte Inc. ‘FPGAXpose 1.5-Fix Runtime Errors Faster.’” [Online]. Available: http://www.sandbyte.com/FPGAXposeDataSheet.pdf.

[42] C. L. Chuang, W. H. Cheng, D. J. Lu, and C. N. J. Liu, “Hybrid approach to faster functional verification with full visibility,” IEEE Des. Test Comput., vol. 24, no. 2, pp. 154–162, 2007.

[43] R. Goering, “Scan design called portal for hackers.” [Online]. Available: http://eetimes.com/electronics-news/4050578/Scan-design-called-portal-for-hackers, October 2004. .

[44] A. Carbine and D. Feltham, “Pentium (R) Pro processor design for test and debug,” in Proceedings International Test Conference 1997, 1997, pp. 294–303.

[45] K. Holdbrook, S. Joshi, S. Mitra, J. Petolino, R. Raman, and M. Wong, “Microsparc: a case-study of scan based debug,” in Proceedings., International Test Conference, 1994, pp. 70–75.

[46] G.-J. Van Rootselaar and B. Vermeulen, “Silicon debug: scan chains alone are not enough,” in International Test Conference 1999. Proceedings (IEEE Cat. No. 99CH37034), 1999, pp. 892–902.

[47] M. W. Riley and M. Genden, “Cell broadband engine debugging for unknown events,” IEEE Des. Test Comput., vol. 24, no. 5, pp. 486–493, 2007.

[48] M. Riley, N. Chelstrom, M. Genden, and S. Sawamura, “Debug of the CELL processor: Moving the lab into silicon,” in 2006 IEEE International Test Conference, 2006, pp. 1–9.

[49] B. Vermeulen, T. Waayers, and S. K. Goel, “Core-based scan architecture for silicon debug,” in Proceedings. International Test Conference, 2002, pp. 638–647.

[50] F. M. De Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang, “BackSpace: Formal analysis for post-silicon debug,” Proc. 2008 Int. Conf. Form. Methods Comput. Des. FMCAD, pp. 1–10, 2008.

[51] C.-L. Chuang, D.-J. Lu, and C.-N. Liu, “A snapshot method to provide full visibility for

functional debugging using FPGA," in 13th Asian Test Symposium, 2004, pp. 164–169.

[52] "SpringSoft. Siloti: Visibility Automation System, Datasheet." [Online]. Available: http://www.springsoft.com/assets/files/Datasheets/SS Siloti Datasheet E5 US.pdf, October 2008.

[53] "Xilinx Inc. 7 Series FPGAs Configuration (UG470 v1.13.1)." [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf.

[54] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," IEEE Trans. Very Large Scale Integr. Syst., vol. 12, no. 3, pp. 288–298, 2004.

[55] N. T. H. Nguyen, "Repairing FPGA configuration memory errors using dynamic partial reconfiguration," Ph. D. Diss. Univ. New South Wales, 2017.

[56] A. Tiwari and K. A. Tomko, "Scan-chain based watch-points for efficient run-time debugging and verification of FPGA designs," in Proceedings of the 2003 Asia and South Pacific Design Automation Conference, 2003, pp. 705–711.

[57] J. Tombs et al., "The implementation of a FPGA hardware debugger system with minimal system overhead," in International Conference on Field Programmable Logic and Applications, 2004, pp. 1062–1066.

[58] Y. Iskander, C. Patterson, and S. Craven, "High-level abstractions and modular debugging for fpga design validation," ACM Trans. Reconfigurable Technol. Syst., vol. 7, no. 1, p. 2, 2014.

[59] A. Khan, R. N. Pittman, and A. Forin, "gnosis: A board-level debugging and verification tool," in International Conference on Reconfigurable Computing and FPGAs, 2010, pp. 43–48.

[60] P. Shanker, "Spatial Debug & Debug Without Re-programming in FPGAs," pp. 3–3, 2016.

[61] C. Li, A. Schwarz, and C. Hochberger, "A readback based general debugging framework for soft-core processors," Proc. 34th IEEE Int. Conf. Comput. Des. ICCD 2016, pp. 568–575, 2016.

[62] G. Tzimpragos, D. Cheng, S. Tapp, B. Jayadev, and A. Majumdar, "Application debug in FPGAs in the presence of multiple asynchronous clocks," Proc. Int. Conf. Field-Programmable Technol. FPT 2016, pp. 189–192, 2017.

[63] C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A partial reconfiguration framework," in Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012, 2012.

[64] K. Arshak, E. Jafer, and C. Ibala, "Testing FPGA based digital system using XILINX ChipScope logic analyzer," in 29th International Spring Seminar on Electronics Technology, 2006, pp. 355–360.

[65] "Intel Inc., 'Design Debugging using the signalTap II Embedded Logic Analyzer.'" [Online]. Available: http://www.pcs.usp.br. [Accessed: 20-Sep-2005].

[66] K. Balston, M. Karimibiuki, A. J. Hu, A. Ivanov, and S. J. E. Wilton, "Post-silicon code coverage for multiprocessor system-on-chip designs," IEEE Trans. Comput., vol. 62, no. 2, pp. 242–246, 2012.

[67] C. Beckhoff, D. Koch, and J. Torresen, "Short-circuits on FPGAs caused by partial runtime reconfiguration," in International Conference on Field Programmable Logic and Applications, 2010, pp. 596–601.

[68] "Synopsys. Identify RTL Debugger: Simulator-like Visibility into FPGA Hardware Operation."

[Online]. Available: https://www.synopsys.com/cgi-bin/verification/dsdla/pdfr1.cgi?file=identify-rtl-debugger-ds.pdf.

[69] “Mentor Inc. Certus Silicon Debug.” .

[70] “Agilent Technologies, Inc., ‘Deep Storage with Xilinx ChipScope Pro and Agilent Technologies FPGA Trace Port Analyzer.’”

[71] “Exostiv Inc., ‘Exostiv Dashboard User’s Guide’, Rev. 1.0.12- February 26, 2019.” [Online]. Available: https://www.exostivlabs.com/files/documents/UG601 - EXOSTIV Dashboard.pdf.

[72] “Cadence Inc. Cadence Palladium Series with Incisive XE Software —Hardware/software co-verification and system-level verification.” [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-xp.html.

[73] “Mentor Graphics. A Closer Look at Veloce Technology: Taking Hardware-Assisted Verification to the Next Level.” [Online]. Available: https://www.mentor.com/products/fv/emulation-systems/veloce-strato.html. [Accessed: 20-Sep-2005].

[74] “Synopsys. ZeBu-Server— Enterprise Emulator.” [Online]. Available: https://www.synopsys.com/verification/emulation/zebu-server.html. [Accessed: 20-Sep-2005].

[75] J. Gao, Y. Han, and X. Li, “A new post-silicon debug approach based on suspect window,” in Proceedings of the IEEE VLSI Test Symposium, 2009.

[76] E. Anis and N. Nicolici, “Low cost debug architecture using lossy compression for silicon debug,” Proc. -Design, Autom. Test Eur. DATE, pp. 225–230, 2007.

[77] J. S. Yang and N. A. Touba, “Automated selection of signals to observe for efficient silicon debug,” in Proceedings of the IEEE VLSI Test Symposium, 2009.

[78] H. F. Ko and N. Nicolici, “Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug,” IEEE Trans. Comput. Des. Integr. Circuits Syst., 2009.

[79] F. M. De Paula, A. Nahir, Z. Nevo, A. Orni, and A. J. Hu, “TAB-BackSpace: Unlimited-length trace buffers with zero additional on-chip overhead,” Proc. - Des. Autom. Conf., pp. 411–416, 2011.

[80] F. M. De Paula, A. J. Hu, and A. Nahir, “nuTAB-BackSpace: Rewriting to normalize non-determinism in post-silicon debug traces,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2012.

[81] W. Jung, H. Oh, D. Kang, and S. Kang, “A 2-D compaction method using macro block for post-silicon validation,” in 2015 International SoC Design Conference (ISOCC), 2015, pp. 41–42.

[82] “Xilinx Inc. ‘Integrated Logic Analyzer v6.1’. LogiCORE IP Product Guide.” .

[83] E. Hung and S. J. E. Wilton, “Speculative debug insertion for FPGAs,” in Proceedings - 21st International Conference on Field Programmable Logic and Applications, FPL, 2011.

[84] F. Eslami, E. Hung, and S. J. E. Wilton, “Enabling Effective FPGA Debug using Overlays : Opportunities and Challenges,” Olaf, 2016.

[85] R. Hale and B. Hutchings, “Enabling low impact, rapid debug for highly utilized FPGA designs,” Proc. - 2018 Int. Conf. Field-Programmable Log. Appl. FPL 2018, pp. 81–84, 2018.

[86] C. Blochwitz, R. Klink, J. M. Joseph, and T. Pionteck, “Continuous live-tracing as debugging

approach on FPGAs," 2017 Int. Conf. Reconfigurable Comput. FPGAs, ReConFig 2017, vol. 2018-Janua, pp. 1–8, 2018.

[87] R. Backasch, C. Hochberger, A. Weiss, M. Leucker, and R. Lasslop, "Runtime verification for multicore SoC with high-quality trace data," ACM Trans. Des. Autom. Electron. Syst., vol. 18, no. 2, 2013.

[88] N. Decker et al., "Rapidly adjustable non-intrusive online monitoring for multi-core systems," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2017.

[89] S. Deutsch and K. Chakrabarty, "Massive signal tracing using on-chip DRAM for in-system silicon debug," Proc. - Int. Test Conf., vol. 2015-Febru, pp. 1–10, 2015.

[90] Z. Panjkov, A. Wasserbauer, T. Ostermann, and R. Hagelauer, "Hybrid FPGA debug approach," 25th Int. Conf. F. Program. Log. Appl. FPL 2015, no. Imc, 2015.

[91] B. R. Quinton and S. J. E. Wilton, "Concentrator access networks for programmable logic cores on SoCs," Proc. - IEEE Int. Symp. Circuits Syst., pp. 45–48, 2005.

[92] V. Raghunathan, M. B. Srivastava, and R. K. Gupta, "A survey of techniques for energy efficient on-chip communication," in Proceedings - Design Automation Conference, 2003.

[93] J. F. C. Kingman and V. E. Benes, "Mathematical Theory of Connecting Networks and Telephone Traffic.," J. R. Stat. Soc. Ser. A, 1966.

[94] F. K. Hwang, The mathematical theory of nonblocking switching networks, vol. 15. World Scientific Publishing Company, 2004.

[95] M. Pinsker, "On the Complexity of a Concentrator," 7th Annu. Teletraffic Conf., 1973.

[96] F. R. K. Chung, "On Concentrators, Superconcentrators, Generalizers, and Nonblocking Networks," Bell Syst. Tech. J., 1979.

[97] "Intel Inc. Quick Intel® Arria® 10 Design Debugging Using Signal Probe and Rapid Recompile." [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an799.pdf.

[98] B. Quinton and S. Wilton, "Programmable logic core based post-silicon debug for SoCs," 4th IEEE Silicon Debug Diagnosis Work., no. March 2012, 2007.

[99] Z. Poulos, Y. S. Yang, J. Anderson, A. Veneris, and B. Le, "Leveraging reconfigurability to raise productivity in FPGA functional debug," in Proceedings -Design, Automation and Test in Europe, DATE, 2012.

[100] "Microsemi Inc. Silicon Explorer II: User's Guide." [Online]. Available: https://www.microsemi.com/document-portal/doc_view/130911-silicon-explorer-ii-user-s-guide.

[101] D. Q. G. Wkh, IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows. 2010.

[102] D. Chen, J. Cong, and P. Pan, "FPGA design automation: A survey," Foundations and Trends in Electronic Design Automation. 2006.

[103] O. Coudert, J. Cong, S. Malik, and M. Sarrafzadeh, "Incremental CAD," in IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD, 2000.

[104] J. M. Emmert and D. Bhatia, "Incremental routing in FPGAs," in Proceedings of the Annual

IEEE International ASIC Conference and Exhibit, 1998.

[105] S. Dutt, V. Shanmugavel, and S. Trimberger, “Efficient incremental rerouting for fault reconfiguration in field programmable gate arrays,” in Proceedings of the IEEE/ACM international conference on Computer-aided design, 1999, pp. 173–177.

[106] S. M. Trimberger, Field-programmable gate array technology. Springer Science & Business Media, 2012.

[107] P. Graham, B. Nelson, and B. Hutchings, “Instrumenting Bitstreams for Debugging FPGA Circuits,” Proc. - 9th Annu. IEEE Symp. Field-Programmable Cust. Comput. Mach. FCCM 2001, pp. 41–50, 2001.

[108] E. Keller, “JRoute: A run-time routing API for FPGA hardware,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2000.

[109] E. Hung and S. J. E. Wilton, “Accelerating FPGA debug: Increasing visibility using a runtime reconfigurable observation and triggering network,” ACM Trans. Des. Autom. Electron. Syst., vol. 19, no. 2, 2014.

[110] E. Hung and S. J. E. Wilton, “Towards simulator-like observability for FPGAs: A virtual overlay network for trace-buffers,” in ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA, 2013.

[111] A. Chandrasekharan et al., “Accelerating FPGA development through the automatic parallel application of standard implementation tools,” in Proceedings of International Conference on Field-Programmable Technology, FPT’10, 2010.

[112] L. Lagadec and D. Picard, “Software-like debugging methodology for reconfigurable platforms,” in 2009 IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1–4.

[113] M. Abramovici, “In-system silicon validation and debug,” IEEE Des. Test Comput., vol. 25, no. 3, pp. 216–223, 2008.

[114] M. Kudlugi, S. Hassoun, C. Selvidge, and D. Pryor, “A Transaction-Based Unified Simulation / Emulation,” pp. 623–628, 2001.

[115] B. Vermeulen, “Functional debug techniques for embedded systems,” IEEE Des. Test Comput., vol. 25, no. 3, pp. 208–215, 2008.

[116] E. A. Daoud and N. Nicolici, “Real-time lossless compression for silicon debug,” IEEE Trans. Comput. Des. Integr. Circuits Syst., vol. 28, no. 1, pp. 1387–1400, 2009.

[117] M. Nelson and J. Gailly, The Data Compression Book 2nd edition. 1995.

[118] C. E. Shannon, “A Mathematical Theory of Communication,” Bell Syst. Tech. J., 1948.

[119] J. S. Yang and N. A. Touba, “Expanding trace buffer observation window for in-system silicon debug through selective capture,” in Proceedings of the IEEE VLSI Test Symposium, 2008.

[120] E. Anis and N. Nicolici, “On using lossless compression of debug data in embedded logic analysis,” in Proceedings - International Test Conference, 2008.

[121] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” Proc. IRE, 1952.

[122] D. E. Knuth, “Dynamic huffman coding,” J. Algorithms, 1985.

[123] M. B. Lin, J. F. Lee, and G. E. Jan, “A lossless data compression and decompression algorithm and its hardware architecture,” IEEE Trans. Very Large Scale Integr. Syst., 2006.

[124] R. N. Williams, Adaptive data compression, vol. 110. Springer Science & Business Media, 2012.

[125] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable memory (CAM) circuits and architectures: A tutorial and survey,” IEEE Journal of Solid-State Circuits. 2006.

[126] S. W. Seong and P. Mishra, “Bitmask-based code compression for embedded systems,” in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008.

[127] M. A. Abd El Ghany, A. E. Salama, and A. H. Khalil, “Design and implementation of FPGA-Based systolic array for LZ data compression,” in Proceedings - IEEE International Symposium on Circuits and Systems, 2007.

[128] K. Basu and P. Mishra, “Efficient trace data compression using statically selected dictionary,” in Proceedings of the IEEE VLSI Test Symposium, 2011.

[129] J. Zhang, L. J. Fritz, L. Liu, and E. Larsson, “Compressor design for silicon debug,” in Proceedings of the European Test Workshop, 2016.

[130] K. Goossens, B. Vermeulen, R. Van Steeden, M. Bennebroek, and T. Design, “Transaction-Based Communication-Centric Debug,” 2007.

[131] A. M. Gharehbaghi and M. Fujita, “Transaction-based debugging of system-on-chips with patterns,” Proc. - IEEE Int. Conf. Comput. Des. VLSI Comput. Process., pp. 186–192, 2009.

[132] C. Ciordas, K. Goossens, T. Basten, A. Radulescu, and A. Boon, “Transaction monitoring in networks on chip: The on-chip run-time perspective,” in Industrial Embedded Systems - IES’2006, 2006.

[133] D. Shanthi and R. Amutha, “Design of efficient on-chip communication architecture in MpSoC,” in International Conference on Recent Trends in Information Technology, ICRTIT 2011, 2011.

[134] S. Pasricha, N. Dutt, and M. Ben-Romdhane, On-chip communication architectures: system on chip interconnect. 2010.

[135] V. Rantala, T. Lehtonen, J. Plosila, and others, Network on chip routing algorithms. Citeseer, 2006.

[136] R. P. Battaline, J. R. Robinson, E. H. Welbon, and R. J. Williams, “Performance monitoring through JTAG 1149.1 interface.” Google Patents, 1998.

[137] L. Möller, H. Jesus, F. Moraes, L. S. Indrusiak, T. Hollstein, and M. Glesner, “Graphical interface for debugging RTL networks-on-chip,” BEC 2010 - 2010 12th Bienn. Balt. Electron. Conf. Proc. 12th Bienn. Balt. Electron. Conf., pp. 181–184, 2010.

[138] B. Vermeulen, J. Dielissen, K. Goossens, and C. Ciordas, “Bringing communication networks on a chip: Test and verification implications,” IEEE Commun. Mag., vol. 41, no. 9, pp. 74–81, 2003.

[139] M. Stępniewska, O. Stankiewicz, A. Łuczak, and J. Siast, “Embedded debugging for NoCs,” Proc. 17th Int. Conf. - Mix. Des. Integr. Circuits Syst. Mix. 2010, pp. 601–606, 2010.

[140] J. Goeders and S. J. E. Wilton, “Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAS,” Proc. - 2015 IEEE 23rd Annu. Int. Symp. Field-Programmable Cust. Comput. Mach. FCCM 2015, pp. 127–134, 2015.

[141] X. Cheng, A. W. Ruan, Y. B. Liao, P. Li, and H. C. Huang, “A run-time RTL debugging

methodology for FPGA-based co-simulation," in International Conference on Commun. Circuits Syst. ICCCAS 2010 - Proc., pp. 891–895, 2010.

[142] T. H. Chang, S. C. Hou, and I. J. Huang, "A unified GDB-based source-transaction level SW/HW co-debugging," In IEEE Asia Pacific Conf. Circuits Syst. APCCAS 2016, pp. 506–509, 2017.

[143] H. C. Chen, C. R. Wu, K. S. M. Li, and K. J. Lee, "A breakpoint-based silicon debug technique with cycle-granularity for handshake-based SoC," Proc. -Design, Autom. Test Eur. DATE, vol. 2015-April, pp. 1281–1284, 2015.

[144] R. Ginosar, "Metastability and synchronizers: A tutorial," IEEE Des. Test Comput., vol. 28, no. 5, pp. 23–35, 2011.

[145] S. K. Goel and B. Vermeulen, "Hierarchical data invalidation analysis for scan-based debug on multiple-clock system chips," in Proceedings. International Test Conference, 2002, pp. 1103–1110.

[146] S. Hong and K. Lee, "A run-pause-resume silicon debug technique for multiple clock domain systems," in International Test Conference in Asia (ITC-Asia), 2017, pp. 46–51.

[147] A. Majumdar, B. Jayadev, D. Cheng, and A. Lin, "Architecture for reliable scan-dump in the presence of multiple asynchronous clock domains in FPGA SoCs," Proc. Asian Test Symp., pp. 134–139, 2018.

[148] J. Gao, Y. Han, and X. Li, "Eliminating data invalidation in debugging multiple-clock chips," Proc. -Design, Autom. Test Eur. DATE, pp. 691–696, 2011.

[149] S. Balasubramanian, N. Natarajan, O. Franza, and C. Gianos, "Deterministic low-latency data transfer across non-integral ratio clock domains," in 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06), 2006, pp. 5--pp.

[150] J. M. Chabloz and A. Hemani, "Low-latency maximal-throughput communication interfaces for rationally related clock domains," IEEE Trans. Very Large Scale Integr. Syst., vol. 22, no. 3, pp. 641–654, 2014.

[151] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," IEEE Micro, 2003.

[152] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2007.

[153] X. Liu and Q. Xu, "On multiplexed signal tracing for post-silicon validation," IEEE Trans. Comput. Des. Integr. Circuits Syst., 2013.

[154] D. Chatterjee, C. McCarter, and V. Bertacco, "Simulation-based signal selection for state restoration in silicon debug," in IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD, 2011.

[155] S. Prabhakar and M. Hsiao, "Using non-trivial logic implications for trace buffer-based silicon debug," in 2009 Asian Test Symposium, 2009, pp. 131–136.

[156] Y. Yang, B. Keng, N. Nicolici, A. Veneris, and S. Safarpour, "Automated silicon debug data analysis techniques for a hardware data acquisition environment," in 11th International Symposium on Quality Electronic Design (ISQED), 2010, pp. 675–682.

[157] S. J. E. Wilton, B. R. Quinton, and E. Hung, "Rapid RTL-based signal ranking for FPGA prototyping," in 2012 International Conference on Field-Programmable Technology, 2012, pp. 1–

7.

[158] J.-Y. Jou and C.-N. J. Liu, “Coverage analysis techniques for hdl design validation,” Proc. Asia Pacific CHip Des. Lang., pp. 48–55, 1999.

[159] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, “Abstraction techniques for validation coverage analysis and test generation,” IEEE Trans. Comput., 1998.

[160] E. Hung and S. J. E. Wilton, “On evaluating signal selection algorithms for post-silicon debug,” Proc. 12th Int. Symp. Qual. Electron. Des. ISQED 2011, pp. 290–296, 2011.

[161] A. Vali and N. Nicolici, “Bit-Flip Detection-Driven Selection of Trace Signals,” IEEE Trans. Comput. Des. Integr. Circuits Syst., vol. 37, no. 5, pp. 1076–1089, 2018.

[162] J. W. Eaton, D. Bateman, and S. Hauberg, GNU Octave manual. Network Theory Ltd. Bristol, UK, 2002.

[163] M. J. Goris and P. Box, “Using Matlab to debug software written for a digital signal processor,” in Proceedings of Benelux Matlab User Conference, 1997.

[164] Z. Wang, “Real-time debugging and testing a control system using Matlab,” Open J. Appl. Sci., vol. 3, no. 02, p. 61, 2013.

[165] U. Hatnik and S. Altmann, “Using ModelSim, Matlab/Simulink and NS for simulation of distributed systems,” in International Conference on Parallel Computing in Electrical Engineering: Workshop on System Design Automation, SDA, PARELEC 2004, 2004.

[166] G. Liang, D. He, J. Portilla, and T. Riesgo, “A hardware in the loop design methodology for FPGA system and its application to complex functions,” In International Symposium on VLSI Des. Autom. Test, VLSI-DAT 2012 - Proc. Tech. Pap., pp. 1–4, 2012.

[167] J. C. T. Hai, O. C. Pun, and T. W. Haw, “Accelerating video and image processing design for FPGA using HDL coder and simulink,” in IEEE Conference on Sustainable Utilization And Development In Engineering and Technology (CSUDET), 2015, pp. 1–5.

[168] D. Waterman, “A guide to expert systems,” 1986.

[169] D. Geis, R. Morscher, and J. Kiper, “Expert system for debugging novice programmers’ Pascal programs.”

[170] C.-K. Looi, “Analysing novices programs in a prolog intelligent teaching system,” in Proceedings of the 8th European Conference on Artificial Intelligence, 1988, pp. 314–319.

[171] A. M. Zin, S. Aljunid, Z. Shukur, and M. Nordin, “A Knowledge-based Automated Debugger in Learning System,” Proc. Fourth Int. Work. Autom. Debugging (AADEBUG 2000), 2000.

[172] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura, “A fast hardware/software co-verification method for System-On-a-Chip by using a C/C++ simulator and FPGA emulator with shared register communication,” in Proceedings - Design Automation Conference, 2004.

[173] B. L. Hutchings and B. E. Nelson, “Unifying simulation and execution in a design environment for FPGA systems,” IEEE Trans. Very Large Scale Integr. Syst., vol. 9, no. 1, pp. 201–205, 2001.

[174] “Xilinx Inc. ‘Configuration Readback Capture in UltraScale FPGAs’. Application Note XAPP1230.” [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1230-configuration-readback-capture.pdf.

[175] S. Yang, H. Shim, W. Yang, and C. M. Kyung, “A new RTL debugging methodology in FPGA-based verification platform,” In Proceedings of IEEE Asia-Pacific Conf. Adv. Syst. Integr. Circuits, pp. 180–183, 2004.

[176] A. Koczor, L. Matoga, P. Penkala, and A. Pawlak, “Verification approach based on emulation technology,” in Proceedings of the IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2016, 2016.

[177] C.-Y. R. Huang, Y.-F. Yin, C.-J. Hsu, T. B. Huang, and T.-M. Chang, “SoC HW/SW verification and validation,” in Proceedings of the 16th Asia and South Pacific Design Automation Conference, 2011, pp. 297–300.

[178] A. De Orio, Q. Li, M. Burgess, and V. Bertacco, “Machine learning-based anomaly detection for post-silicon bug diagnosis,” in Proceedings -Design, Automation and Test in Europe, DATE, 2013.

[179] A. Chandramouly, B. D. D. Owner, I. T. R. Narkhede, I. T. V. Mungara, I. T. G. Rueda, and I. T. A. Diggs, “Reducing Client Incidents through Big Data Predictive Analytics,” Intel IT Big Data Predict. Anal., 2013.

[180] P. Simon, Too big to ignore: the business case for big data, vol. 72. John Wiley & Sons, 2013.

[181] C. S. Zhu, G. Weissenbacher, and S. Malik, “Post-silicon fault localisation using maximum satisfiability and backbones,” 2011 Form. Methods Comput. Des. FMCAD 2011, pp. 63–66, 2011.

[182] K. Basu, P. Mishra, and P. Patra, “Efficient combination of trace and scan signals for post silicon validation and debug,” in Proceedings - International Test Conference, 2011.

[183] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise,” in Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, 1996.

[184] A. Jindal, B. Kumar, N. Jindal, M. Fujita, and V. Singh, “Silicon debug with maximally expanded internal observability using nearest neighbor algorithm,” in Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI, 2018.

[185] L. C. Wang, “Experience of data analytics in EDA and test - Principles, promises, and challenges,” IEEE Trans. Comput. Des. Integr. Circuits Syst., 2017.

[186] E. El Mandouh and A. G. Wassal, “Application of Machine Learning Techniques in Post-Silicon Debugging and Bug Localization,” J. Electron. Test. Theory Appl., 2018.

[187] W. Groß, S. Lange, J. Bodecker, and M. Blum, “Predicting Time Series with Space-Time Convolutional and Recurrent Neural Networks,” Comput. Intell., 2017.

[188] B. Vermeulen and K. Goossens, “Debugging multi-core systems-on-chip,” in Multi-Core Embedded Systems, CRC Press, 2018, pp. 185–230.

[189] Value change Dump Format, IEEE Standard 1364-2005. .

[190] T. Bybell, “GtkWave electronic waveform viewer.” 2010.

[191] E. F. Moore, “Machine models of self-reproduction,” in Proceedings of symposia in applied mathematics, 1962, vol. 14, no. 1962, pp. 17–33.

[192] M. J. Narasimha, “A Recursive Concentrator Structure with Applications to Self-Routing Switching Networks,” IEEE Trans. Commun., 1994.

[193] V. N. Anh and A. Moffat, “Inverted index compression using word-aligned binary codes,” Inf. Retr. Boston., vol. 8, no. 1, pp. 151–166, 2005.

[194] “Xilinx Inc. ‘LightWeight IP Application Examples’. Application Note XAPP1026.” .

[195] “Xilinx Inc. “Partial Reconfiguration User Guide UG909" v2017.1.” [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug909-vivado-partial-reconfiguration.pdf.

[196] A. Kamaleldin, I. Ahmed, A. M. Obeid, A. Shalash, Y. Ismail, and H. Mostafa, “A cost-effective dynamic partial reconfiguration implementation flow for xilinx FPGA,” in Proceedings of 1st New Generation of CAS, NGCAS 2017, 2017.

[197] “Digilent Inc. ZedBoard Hardware User’s Guide.” [Online]. Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf.

[198] C. Carmichael, “Triple module redundancy design techniques for Virtex FPGAs,” Xilinx Appl. Note XAPP197, vol. 1, 2001.

[199] A. Lakhani, I. Gorlach, and F. Smith, “Design of a position and orientation measurement robot,” in Presented at the 4th Robotics and Mechatronics Conference of South Africa (ROBMECH 2011), 2011, vol. 23, p. 25.

[200] Q. Xu and X. Liu, “On signal tracing in post-silicon validation,” Proc. Asia South Pacific Des. Autom. Conf. ASP-DAC, pp. 262–267, 2010.

[201] S. Liu, R. N. Pittman, and A. Forin, “Energy reduction with run-time partial reconfiguration (abstract only),” no. September 2009, p. 292, 2010.

[202] B. Vermeulen, K. Goossens, and S. Umrani, “Debugging distributed-shared-memory communication at multiple granularities in networks on chip,” in Proceedings - Second IEEE International Symposium on Networks-on-Chip, NOCS 2008, 2008.

[203] “Lauterbach GmbH. Connecting to MicroBlaze Targets for Debug and Trace.” [Online]. Available: https://www.lauterbach.com/frames.html?home.html.

[204] “VectorBlox Computing Inc. VectorBlox/risc-v.” [Online]. Available: https://github.com/VectorBlox/orca.

[205] H.-C. Ng, C. Liu, and H. K.-H. So, “A soft processor overlay with tightly-coupled FPGA accelerator,” arXiv Prepr. arXiv1606.06483, 2016.

[206] “Xilinx Inc. ‘ Clocking Wizard v5.3’ . LogiCORE IP Product Guide PG065.” [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v5_3/pg065-clk-wiz.pdf.

[207] J. Rettkowski and D. Göhringer, “RAR-NoC: A reconfigurable and adaptive routable Network-on-Chip for FPGA-based multiprocessor systems,” in International Conference on Reconfigurable Computing and FPGAs, ReConFig 2014, 2014.

[208] J. Postel, “RFC 768: User Datagram Protocol,” 1980.

[209] A. Ezust and P. Ezust, An introduction to design patterns in C++ with Qt 4. Prentice-Hall, 2007.

[210] M. Krstić, E. Grass, F. K. Gürkaynal, and P. Vivet, “Globally asynchronous, locally synchronous circuits: Overview and outlook,” IEEE Des. Test Comput., vol. 24, no. 5, pp. 430–441, 2007.

[211] K. J. Lee, S. Y. Liang, and A. Su, “A low-cost SOC debug platform based on on-chip test architectures,” Proc. - IEEE Int. SOC Conf. SOCC 2009, pp. 161–164, 2009.

[212] J. Borghoff et al., “PRINCE--a low-latency block cipher for pervasive computing applications,” in International Conference on the Theory and Application of Cryptology and Information Security, 2012, pp. 208–225.

[213] T. Grimm, D. Lettnin, and M. Hübner, “Automatic generation of RTL connectivity checkers from SystemC TLM and IP-XACT descriptions,” in 2nd IEEE NORCAS Conference, 2016.

[214] J.-F. Le Tallec, J. DeAntoni, R. De Simone, B. Ferrero, F. Mallet, and L. Maillet-Contoz, “Combining SystemC, IP-XACT and UML/MARTE in model-based SoC design,” in Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011), 2011.

[215] R. Taylor, “Interpretation of the Correlation Coefficient: A Basic Review,” J. Diagnostic Med. Sonogr., 1990.

[216] M. Kantrowitz and L. M. Noack, “I’m Done Simulating; Now What?,” in 33rd Design Automation Conference, 1997, pp. 325–330.

[217] “Xilinx Inc. ‘Vivado Design Suite High Level Synthesis’. User Guide UG902 (v2017.4).” [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-%09synthesis.pdf.

[218] P. Zadek, A. Koczor, M. Golek, L. Matoga, and P. Penkala, “Improving efficiency of FPGA-in-the-loop verification environment,” IFAC-PapersOnLine, vol. 28, no. 4, pp. 180–185, 2015.

[219] J. H. Friedman and L. C. Rafsky, “Multivariate Generalizations of the Wald-Wolfowitz and Smirnov Two-Sample Tests,” Ann. Stat., 1979.

[220] Y. W. Cheung and K. S. La, “Lag order and critical values of the augmented dickey-fuller test,” J. Bus. Econ. Stat., 1995.

[221] F. Chollet, Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek. MITP-Verlags GmbH & Co. KG, 2018.

[222] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv Prepr. arXiv1412.6980, 2014.

[223] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, “The marginal value of adaptive gradient methods in machine learning,” in Advances in Neural Information Processing Systems, 2017.

[224] V. Bertacco and W. Bonkowski, “ItHELPS: Iterative high-Accuracy error localization in post-silicon,” in Proceedings of the 33rd IEEE International Conference on Computer Design, ICCD 2015, 2015.

Publications of the Author

[1] H. Khan and D. Göhringer, “FPGA debugging by a device start and stop approach,” in International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2016, pp. 1–6.

[2] H. Khan, J. Rettkowski, M. Eldafrawy, and D. Göhringer, “An event-based Network-on-Chip debugging system for FPGA-based MPSoCs,” in International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017, pp. 30–37.

[3] H. Khan, T. Grimm, M. Hübner, and D. Göhringer, “Access Network Generation for Efficient Debugging of FPGAs,” in Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, 2017, p. 25.

[4] H. Khan , A. Kamal and D. Göhringer, “An Intrusive Dynamic Reconfigurable Cycle-Accurate Debugging System for Embedded Processors,” in International Symposium on Applied Reconfigurable Computing, 2018, pp. 433–445.

[5] H. Khan and D. Göhringer, “FPGA Debugging with MATLAB Using a Rule-Based Inference System,” in International Symposium on Applied Reconfigurable Computing, 2017, pp. 106–117.

[6] H. Khan, A. Podlubne and D. Göhringer, “Intrusive FPGA-in-the-loop debugging using a rule-based inference system,” Microprocess. Microsyst., vol. 64, pp. 185–194, 2019.

[7] H. Khan and D. Göhringer, “Cycle-Accurate and Cycle-Reproducible Debugging of Embedded Designs Using Artificial Intelligence,” in 28th International Conference on Field Programmable Logic and Applications (FPL), 2018, pp. 449–4491.

[8] H. Khan, G. Akgün, A. Podlubne, F. Wegner, A. Moradi and D. Göhringer, “Cycle-accurate Debugging of Multi-clock Reconfigurable Systems,” in International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2019.

Other Publications of the Author

[9] G. Akgün, H. Khan, M. Hebaish and D. Göhringer. "System Identification using LMS, RLS, EKF and Neural Network," in International Conference on Vehicular Electronics and Safety (ICVES'2019).

[10] G. Akgün, H. Khan, M. A. Elshimy and D. Göhringer, “Dynamic tunable and reconfigurable hardware controller with EKF-based state reconstruction through FPGA-in the loop,” in International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2018, pp. 1–8.

[11] A. Podlubne et al., “Low power image processing applications on FPGAs using dynamic voltage scaling and partial reconfiguration,” in Conference on Design and Architectures for Signal and Image Processing (DASIP), 2018, pp. 64–69.

Submitted Publications

[12] H. Khan, G. Akgün, A. Podlubne and D. Göhringer, “Cycle-accurate Debugging of Embedded Designs using Recurrent Neural Networks,” in International Symposium on Applied Reconfigurable Computing, 2020.

Bachelor Thesis Supervised

[1] **Mohamed ElDafrawy**. "Development of a Network-on-Chip Debugger". Electrical Engineering and Information Technology. Ruhr University Bochum, Germany. 2016

[2] **Mohammed Khaled El-molla**. "Design of Obstacle Avoidance System for Robotic application based on SoC". Fakultät Informatik. Technische Universität Dresden, Germany. 2018.

[3] **Mahmoud El-shimy**. "FPGA-in-the-loop for an Inverted Pendulum". Fakultät Informatik. Technische Universität Dresden, Germany. 2018.

[4] **Mohamed Ahmed Mohamed Taher Ahmed Ellaithy**. "Design and Implementation of Path Planning and Control of a Mobile Robot". Fakultät Informatik. Technische Universität Dresden, Germany. 2019.

[5] **Marawan Azmy Hebaish**. "Design and Implementation of Online Identification Algorithms and Discrete Controller on Reconfigurable Systems". Fakultät Informatik. Technische Universität Dresden, Germany. 2019.