Simulation-based Functional Verification of Dynamically Reconfigurable FPGA-based Systems

Lingkan Gong

School of Computer Science and Engineering
Faculty of Engineering

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

A Thesis Submitted for the Degree of
Doctor of Philosophy

July 2013

Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Signed:_____

Abstract

Dynamically Reconfigurable Systems (DRS), implemented using Field-Programmable Gate Arrays (FPGAs), allow hardware logic to be partially reconfigured while the rest of a design continues to operate. By mapping multiple reconfigurable hardware modules to the same physical region of an FPGA, such systems are able to time-multiplex their circuits at run time and can adapt to changing execution requirements. This architectural flexibility introduces challenges for verifying system functionality. New simulation approaches need to extend traditional simulation techniques to assist designers in testing and debugging the time-varying behavior of DRS. Another significant challenge is the effective use of tools so as to reduce the number of design iterations. This thesis focuses on simulation-based functional verification of modular reconfigurable DRS designs. We propose a methodology and provide tools to assist designers in verifying DRS designs while part of the design is undergoing reconfiguration.

This thesis analyzes the challenges in verifying DRS designs with respect to the user design and the physical implementation of such systems. We propose using a simulation-only layer to emulate the behavior of target FPGAs and accurately model the characteristic features of reconfiguration. The simulation-only layer maintains verification productivity by abstracting away the physical details of the FPGA fabric. Furthermore, since the design does not need to be modified for simulation purposes, the design as implemented instead of some variation of it is verified.

We provide two possible implementations of the simulation-only layer. Extended ReChannel is a SystemC library that can be used to model DRS at a high level. ReSim is a library to support RTL simulation of a DRS reconfiguring both its logic and state. Through a number of case studies, we demonstrate that with insignificant overheads, our approach seamlessly integrates with the existing, mainstream DRS design flow and with well-established verification methodologies such as top-down modeling and coverage-driven verification. The case studies also serve as a guide in the use of our libraries to identify bugs that are related to Dynamic Partial Reconfiguration. Our results demonstrate that using the simulation-only layer is an effective approach to the simulation-based functional verification of DRS designs.

Acknowledgments

Looking back to years ago, the sacred word "PhD" first jumped into my consciousness, and implanted itself as an honorable yet formidable title like any freshman, including me, could imagine. The time came for me to start some meaningful actions towards applying for a PhD when I attempted to submit a paper of my master's project to a prestigious conference in a foreign language that I only ever used in classrooms. It was 3 years ago, on a typically Australian bright and sunny morning, when I first landed on this land as a PhD candidate, with my first touch of the earth outside my own country. The "Sands" of time finally ticked to the point when my PhD thesis was ready to be submitted ... The three years of PhD study has not taken a long time within the context of a life time, but has been an absolutely essential experience for me to make a tiny yet solid contribution to the community, and to undertake a profound change in my life.

The first character, apart from myself, of the 3-year-long drama is Dr. Oliver Diessel, my PhD supervisor, who significantly changed my style of thinking in research and in daily life. For each paper I drafted, Oliver put great efforts in commenting on the strengths and the drawbacks, inspiring me in the direction of my writing, re-wording parts that were not clear and fixing grammatical errors. The red notes on the pile of manuscripts attest to his efforts towards a young man on his way to becoming a Doctor of Philosophy. For each meeting we sat together, he encouraged me to develop my own idea, to express thoughts in the context of an overall big picture, to summarize technical details into high-level concepts and to emphasize on WHY instead of HOW. At each conference event we attended, he introduced me to other researchers and took me to a broader research community. With his guidance, I have never felt so confident and positive in presenting my ideas in front of a big or a small audience. For every non-working minute we shared, he engaged me in the life of Australia, exchanged his experiences with me about our different cultural backgrounds, and even taught me driving with his precious Volkswagen Eos. Every time he said "Don't work too hard George", he *meant* do not work to hard George, instead of anything else. At this moment, any variation of the word THANKS can hardly express my gratefulness to him.

Another person that significantly impacted the 3-year-long story was Ms. Vivien Ye Shi, my partner, who jumped into the drama of my PhD study and provided her greatest supports and assistance to the little goal in my mind. She brought balance to my life and study, and we shared our hard times and happiness together in a country far away from our motherland. Selflessly, she took care of me at the critical moments when I had to meet a range of milestones/deadlines that came one after another throughout the 3-year-long period. Although her field of study is far from what I am doing, she showed

her patience in listening to my complaints about bugs, bugs and bugs, and has by now become a 50%-expert at the so-called "FGPA" (yes, FGPA) thing. Above all, Vivien is a PhD student herself, yet she has always been willing to sacrifice her time to back me up. Admittedly, my PhD thesis would have been finished A MONTH earlier without her distraction. But I also cannot deny the fact that her PhD thesis could have been completed A YEAR earlier if I hadn't appeared in her life. I don't think THANKS is the right word for her either. Given that we mean to share a common future, I sincerely want to express my LOVE to her.

Without a doubt, I must thank Mum and Dad, who never complained about their son not being able to take care of them. Every time I went home and opened the door, I couldn't stop noticing that more wrinkles had climbed into their faces. Even if they wouldn't understand this thesis, they are still proud of me as they have always been, and I am also proud of them.

To this big day of completing my PhD thesis, I would like to thank my former supervisors Prof. Evangeline F.Y. Young, who allowed me to pursue my own research interest, Mr. Jianxiong Zhang, Ms. Yuyan Wang and Mr. Peng Ma, who led me to an IC design career, and Prof. Hongqiang Li, who was the first person to guide me to do research. I would also like to thank Dr. Chris Nicol, who significantly broadened my view of the semiconductor industry, Mr. Jens Hagemeyer, who patiently assisted me with the state restoration technique on Virtex-5 FPGAs, Mr. Philip Ryan, Mr. Luke Darnell and Mr. Michael De Nil, who offered me the great internship opportunity at Broadcom, Dr. Gordon Bredner, who hosted my talk at Xilinx Labs, Mr. Johny Paul and Prof. Walter Stechele, who provided the AutoVision design as an important case study for assessing ReSim, and many anonymous reviewers, who provided me with valuable suggestions on my published papers. I would then like to thank all students and staff from the Embedded Systems Group at UNSW, Mr. Yulei Sui, Mr. Liang Tang and Mr. Victor Lai in particular. Last but not least I would like to thank the Australian Government for the International Postgraduate Research Scholarship, and would like to thank Xilinx for their generous donations.

These three years mark the end of a painful, dedicated, yet fulfilling student life and the start of a new journey, full of responsibility to my family, and my dream of a better future.

Contents

List of Figures vii

List of Tables ix

List of Abbreviations x

List of Publications xii

1 Introduction 1

1.1 Thesis Objectives 4

1.2 Thesis Contributions 6

1.3 Thesis Outline 7

2 Background and Related Work 9

2.1 Functional Verification of FPGA-based Designs 9

2.2 Challenges in Verifying DRS Designs 13

2.2.1 Verification Challenges Arising from the User Design Layer 15

2.2.2 Verification Challenges Arising from the Physical Layer 18

2.2.3 Simulation Accuracy vs. Verification Productivity 23

2.3 Simulation-based Verification of DRS Designs 25

2.4 Formal and Run-time Verification of DRS Designs 29

2.5 Summary 32

3 Modeling Dynamic Partial Reconfiguration **33**

3.1 The Physical Layer . . . 33

3.1.1 The Configuration Mechanisms . . . 34

3.1.2 The Characteristic Features of Reconfiguration . . . 37

3.2 The Simulation-only Layer . . . 38

3.2.1 Modeling the Configuration Mechanisms . . . 40

3.2.2 Modeling Characteristic Features of Reconfiguration . . . 44

3.3 Capabilities and Limitations . . . 46

3.4 Summary . . . 50

4 ReSim-based Simulation **52**

4.1 High-level Modeling . . . 53

4.1.1 Behavioral Level Modeling . . . 53

4.1.2 Transaction Level Modeling . . . 54

4.2 RTL Simulation . . . 56

4.3 Parameter Script . . . 59

4.4 Capabilities and Limitations . . . 61

4.5 Summary . . . 62

5 Case Studies **64**

5.1 Case Study I: In-house DRS Computing Platform . . . 64

5.1.1 Targeting a Second Application . . . 73

5.2 Case Study II: In-house Fault-Tolerant Application . . . 76

5.3 Case Study III: Third-Party Video-Processing Application . . . 80

5.4 Case Study IV & V: Vendor Reference Designs . . . 86

5.4.1 Case Study IV: Fast PCIe Reference Design . . . 86

5.4.2 Case Study V: Reconfigurable Peripheral Reference Design . . . 88

5.5 Summary . . . 91

6 Conclusions **96**

6.1 Concluding Remarks 96

6.2 Future Work 98

A Bugs Detected in Case Studies **101**

A.1 Case Study I: In-house DRS Computing Platform 101

A.2 Case Study II: In-house Fault-Tolerant Application 112

A.3 Case Study III: Third-Party Video-Processing Application 117

A.4 Case Study IV & V: Vendor Reference Designs 119

Bibliography **120**

List of Figures

1.1 Estimated chip design cost 1

2.1 Typical hardware design flow 10

2.2 Tradeoff between simulation accuracy and verification productivity (for static designs) 10

2.3 A typical simulation environment 12

2.4 User design layer of a DRS design 14

2.5 Conceptual diagram of a DRS design 15

2.6 Basic Logic and Routing Block of FPGAs 19

2.7 Tradeoff between simulation accuracy and verification productivity (for static and DRS designs) 28

3.1 Conceptual diagram of a DRS design (Same as Figure 2.5) 34

3.2 Configuration memory of the Virtex-5 FPGA family 35

3.3 Using the simulation-only layer 39

3.4 Configuration memory of the simulation-only layer 41

3.5 Fabric-independent and fabric-dependent bugs 47

3.6 Tradeoff between simulation accuracy and verification productivity (using the simulation-only layer approach) 50

4.1 Using the simulation-only layer (Same as Figure 3.3) 52

4.2 Simulating a DRS at various levels of abstraction 53

4.3 Behavioral modeling using ReChannel 54

4.4 TLM modeling using Extended ReChannel 56

4.5 ReSim-based RTL simulation 57

4.6 An example of a parameter script 60

5.1 The XDRS demonstrator 65

5.2 Development progress of the XDRS system 66

5.3 A Screen-shot of the co-simulation environment for the XDRS system . . 67

5.4 The core logic of the XDRS demonstrator 69

5.5 Waveform example for partial reconfiguration 70

5.6 Extract of test plan and selected coverage items 72

5.7 Coverage-driven verification progress with the XDRS system 72

5.8 Waveform example for state saving and restoration 75

5.9 The hardware architecture of the fault-tolerant DRS 77

5.10 Extract of the test plan section for the voter 78

5.11 Simulation-based verification of the fault-tolerant DRS 78

5.12 The hardware architecture of the Optical Flow Demonstrator 81

5.13 The processing flow of the Optical Flow Demonstrator 81

5.14 Development progress of the Optical Flow Demonstrator 82

5.15 One frame of the input and the processed video 85

5.16 Fast PCIe configuration reference design 87

5.17 Simulating the FPCIe reference design 87

5.18 Reconfigurable Processor Peripheral reference design 89

5.19 A *potential* isolation bug in the reference design 90

6.1 Tradeoff between simulation accuracy and verification productivity (using ReSim and Extended ReChannel) 97

List of Tables

3.1 An example bitstream for partial reconfiguration 36

3.2 An example bitstream for state saving 37

3.3 An example SimB for configuring a new module 43

3.4 An example SimB for state saving 44

3.5 Differences between the Virtex-5 FPGA fabric and the simulation-only layer 47

4.1 Summary of Extended ReChannel and ReSim 62

5.1 Time to simulate one video frame 85

5.2 The effect of SimB size on verification coverage 88

5.3 Summary of case studies 92

5.4 Summary of example bugs described in Chapter 5 94

List of Abbreviations

ASIC Application-Specific Integrated Circuit

BRAM Block Random Access Memory

CAD Computer Aided Design

CLB Configuration Logic Block

EDA Electronic Design Automation

DCR Device Control Register (Bus)

DPR Dynamic Partial Reconfiguration

DRS Dynamically Reconfigurable System

DUT Design Under Test

FA Frame Address

FPGA Field-Programmable Gate Array

FSM Finite State Machine

ICAP Internal Configuration Access Port

IP Intellectual Property

NoC Network-on-Chip

HDL Hardware Description Language

HLS High-level Synthesis

HVL Hardware Verification Language

LOC Lines of Code

LUT Look-Up Table

PAR Placement and Routing

PLB Processor Local Bus

RM Reconfigurable Module

RR Reconfigurable Region

RTL Register Transfer Level

SimB Simulation-only Bitstream

SoC System-on-Chip

SSR State Saving and Restoration

TB Testbench

TLM Transaction Level Modeling

List of Publications

- **Lingkan Gong**, Oliver Diessel, Johny Paul and Walter Stechele, "RTL Simulation of High Performance Dynamic Reconfiguration: A Video Processing Case Study", *Parallel and Distributed Processing, International Symposium on, Reconfigurable Architecture Workshop (RAW)*, 2013, pp. 106 - 113.
- **Lingkan Gong** and Oliver Diessel, "Functionally Verifying State Saving and Restoration in Dynamically Reconfigurable Systems," in *Field Programmable Gate Arrays (FPGA), ACM/SIGDA International Symposium on*, 2012, pp. 241 - 244.
- **Lingkan Gong** and Oliver Diessel, "ReSim: A Reusable Library for RTL Simulation of Dynamic Partial Reconfiguration", in *Field-Programmable Technology (FPT), International Conference on*, 2011, pp. 1–8. **BEST PAPER CANDIDATE**
- **Lingkan Gong** and Oliver Diessel, "Modeling Dynamically Reconfigurable Systems for Simulation-based Functional Verification", in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2011, pp. 9 - 16.

Chapter 1

Introduction

Driven by the desire to integrate more and more functionality into a single solution, Integrated Circuits (IC) are becoming increasingly complex. As a result, the development cost of cutting-edge ICs has been skyrocketing and has become a threat to the continuation of the semiconductor roadmap [3]. It has been estimated that the cost of developing a chip at 28nm technology node could reach over 170 million USD (see Figure 1.1) [105]. Furthermore, the significant investment and engineering effort do not reduce the risk of project failure. The development cycle of chips ranges from months to years with high uncertainty [3].

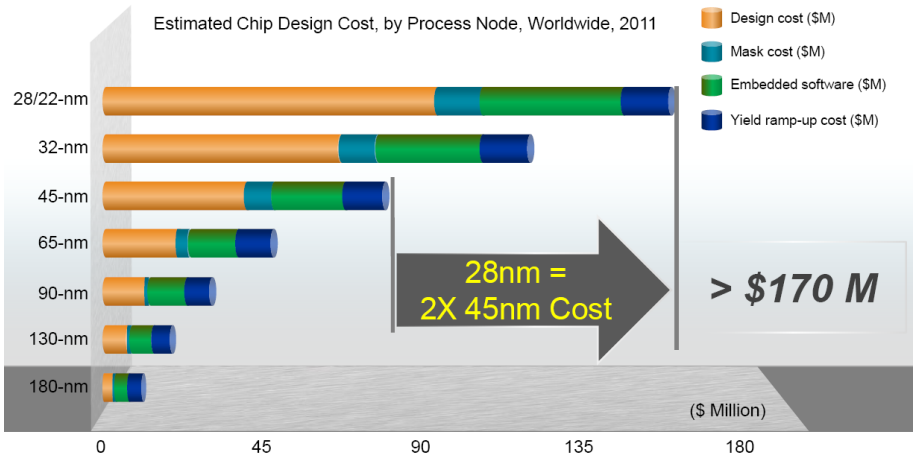


Figure 1.1: Estimated chip design cost [105]

The costs and risks associated with *customized* IC solutions can only therefore be justified for a limited number of ultra-high volume electronic products. The electronic industry has begun shifting towards using *reconfigurable* devices such as Field-Programmable Gate Arrays (FPGAs) as computing platforms. An FPGA is a special type of integrated circuit that can be programmed and reprogrammed at system design time, run time and after deployment. Compared with customized chips, the programmability of reconfigurable devices has increased the flexibility of designs while introducing acceptable overheads in power, area and performance. Hardware/software systems implemented on reconfigurable devices can achieve shorter time-to-market and are more amenable to

upgrades or bug fixes over the product life-cycle. It is predicted that by 2024, on average 70% of the chip functionality will be reconfigurable in various forms [3]. Currently, the number of design starts targeting FPGAs keeps increasing whereas the number is in decline for Application-Specific Integrated Circuits (ASIC) [2]. FPGA vendors are moving into ASIC markets by releasing programmable System-on-Chip platforms (e.g., [8, 103]).

Dynamically Reconfigurable Systems (DRS) further extend design flexibility by allowing hardware sub-modules to be partially reconfigured at run time while the remaining system continues to operate. By mapping multiple reconfigurable hardware modules (RM) to the same physical reconfigurable region (RR) of the FPGA, the system loads required modules on demand, which significantly saves resource usage, increases design density and reduces system cost [90]. An application can also flexibly time-multiplex its modules at run time to adapt to changing execution requirements. For example, a networked multiport switch [98] and a software-defined radio [75] reconfigure their protocol processing logic according to the protocol of the incoming traffic. A video-based driver assistance system swaps image processing modules as the driving conditions change (e.g., from highway to urban and from open roads to tunnel) [18]. By reconfiguring the design with new modules, the functionality of a system can be extended at run time. For example, a network flow analysis application reconfigures its flow monitor module into an intruder detection module when identifying suspected attacks [107].

Designing effective DRS poses several challenging research problems. At a high level, we lack tools for exploring the application design and implementation space so as to easily identify those run-time reconfigurable architectures that would be effective. The vendor tools provided to implement partially reconfigurable designs are relatively immature and inflexible, and unduly constrain the manner in which the technique can be applied. Furthermore, integrated support for verifying that a system undergoing reconfiguration is correct is lacking. The need to develop purpose-specific run-time reconfiguration controllers, which often require detailed knowledge of the target device and the application, adds to the complexity and overheads of a design. Finally, current device architectures impose reconfiguration delays that can often eliminate the performance advantages of employing dynamic reconfiguration.

Functional verification is the bottleneck of hardware design projects and is a significant contributing factor of the high development cost of electronic systems [106]. In cutting-edge ASIC design projects, the ratio of dedicated verification engineers to design engineers is two to one [3]. Although bugs in reconfigurable designs can quickly be recovered by downloading a correct version of the configuration bitstream instead of refabricating the entire chip, the effort required to detect and fix functional bugs does not depend on the target technology and therefore shows similar trends as for ASIC designs. Functional verification is therefore a crucial but a costly step in FPGA-based designs [99]. Accelerating verification closure can thus reduce time-to-market and thereby considerably reduce the cost and enhance the profitability of FPGA design projects.

Dynamic Partial Reconfiguration (DPR) introduces further challenges to the functional verification of DRS designs. In particular, it presents two new verification problems that do not exist in traditional static designs:

- Design time verification. At design time, simulation is the most common method for verifying hardware design functionality. FPGA vendors such as Xilinx claim that each valid configuration of a DRS can be individually tested using traditional simulation methods, but do not support simulating the reconfiguration process itself [98]. However, partial reconfiguration should not be viewed as an isolated process. Although correctly verified sub-systems are necessary, they are not sufficient for ensuring design correctness since the most costly bugs are encountered in system integration [3]. For the sake of system-level verification, the designer should verify that reconfiguration does not introduce bugs to those parts of the system that are not reconfiguring, and that the static part of the design does not introduce bugs to the reconfiguration process.
- Run-time verification. With dynamic reconfigurability, a DRS can be reprogrammed with hardware modules that are not known at system design time. These modules can be designed and verified separately and their configuration bitstreams can be transferred to the DRS system from external sources instead of using predefined bitstreams from a local storage device [98]. However, it is not known whether such modules can be correctly integrated to the DRS. A bug may exist in the module itself or in the reconfiguration process. Therefore, these modules need to be verified while the DRS is running, so that the newly configured modules *do not* introduce bugs to the rest of the system. In particular, the DRS itself requires mechanisms to check the correctness of the incoming module at run time.

In contrast to the challenges in verifying DRS designs, there are no well-accepted verification tools for DRS designs. Traditional simulation methods assume that the design hierarchy is defined at compile time and do not support swapping modules during a simulation run. Furthermore, configuration bitstreams used to re-program the target device can only be interpreted by the FPGA and are meaningless to traditional simulation tools. This inability to simulate the reconfiguration process poses significant difficulties for verifying the functionality of a DRS design while it is undergoing reconfiguration, during which the outgoing RM needs to be properly paused, the RR needs to be isolated and reconfigured, and the incoming RM needs to be activated before it commences execution.

Apart from the need to develop simulation techniques, a more significant challenge to functionally verifying dynamic hardware designs is the *effective* use of existing tools [3]. For traditional static hardware systems, targeting either ASICs or FPGAs, effective design methodologies, such as top-down modeling and coverage-driven verification, reduce the number of design iterations, assist in ensuring verification closure, and are proven to be more important than individual design tools [3]. For DRS designs, partial reconfiguration introduces new scenarios that need to be tested. For example, in order to test whether RMs are properly paused upon the arrival of reconfiguration requests, the simulation environment needs to initiate partial reconfiguration in all possible states of the currently active RM. In order to verify that an ongoing reconfiguration doesn't introduce any error (e.g., deadlock) to the rest of the design, all legal transitions between any two RMs need to be exercised in simulation. It is therefore essential to provide designers with guidelines for verifying the correctness of a design in such DPR-related scenarios.

1.1 Thesis Objectives

This thesis aims to provide solutions to the significant challenges in verifying DPR and to provide assistance to designers in testing and debugging DRS designs. Since design engineers also spend significant time testing and debugging their designs, this thesis does not explicitly distinguish between design engineers and verification engineers but refers to both as designers. The thesis has three objectives.

Mainstream Objective. DRS can be designed in various ways and each design style introduces different types of challenges to verification. This thesis focuses on the functional verification of the mainstream DRS design flow, i.e., modular reconfiguration [98][6]. For modular reconfigurable DRS designs, reconfiguration is performed at the design module level and the system does not change configuration bits within a module (i.e., does not support fine-grained reconfiguration). Reconfigurable modules are known to the designer at design time so we do not consider the run-time verification problem mentioned above.

Reconfiguration can be performed by the design itself or by an external host [98]. The reconfiguration process for both methods is similar but the location of the reconfiguration controller differs. Since most DRS designs in the literature use internal reconfiguration, and without loss of generality, we focus our discussion on self-reconfigurable DRS designs, while also describing the verification of externally reconfigured designs.

There are two main approaches to verifying the correctness of a design. This thesis focus on the most common simulation-based verification approaches. By creating a high-level or Register Transfer Level (RTL) description of the design, simulation-based verification requires a designer to check that the responses of a Design Under Test (DUT) to given stimuli are as expected. While having the disadvantage of not being able to exercise all possible scenarios, this approach is commonly used in industry because it is efficient [3]. Although formal methods prove mathematically that a design matches a specification, their execution times do not scale with design size [3].

Furthermore, we focus on simulation-based verification using mainstream Hardware Description Languages (HDL, e.g., Verilog [37]/VHDL [38]) or high-level modeling languages (e.g., SystemC [40]), and off-the-shelf commercial HDL simulators (e.g., ModelSim [57]). We therefore use mainstream tools to verify designs described by conventional languages.

It should be noted that focusing on mainstream DRS design styles (i.e., modular reconfiguration), mainstream verification techniques (i.e., simulation), mainstream design languages (i.e., HDL and SystemC) and mainstream tools (e.g., ModelSim, FPGA vendor design flow) does not limit the contribution of this thesis. Indeed, having analyzed the state-of-the-art verification techniques for DRS designs (see Chapter 2), we believe there is no well-accepted method for verifying mainstream DRS designs. We therefore believe it is significant to focus on mainstream DRS designs before considering possible extensions to more flexible DRS design styles.

The Mainstream Objective is essential for our work to live within the ecosystem of the current DRS design community. By focusing on mainstream design styles, techniques, tools and languages, we hope to encourage the uptake of DRS designs by industry. Nevertheless, we also discuss the application of our approach to non-mainstream DRS designs and non-mainstream verification methods, so as to demonstrate the flexibility of our approach.

Functional Verification Objective. This thesis focuses on functional verification of DRS designs. Therefore, we aim to assist designers in identifying and fixing *functional bugs*, which are bugs introduced to the design specification and to the captured design (see Section 2.1). A typical functional bug in a hardware design is a cycle mismatch. Design errors introduced in the implementation step such as timing violations, incorrect constraints and failures in placing and routing modules are typically handled by the implementation tools provided by the vendor and are therefore out of the scope of functional verification.

As required by the Mainstream Objective, we focus on simulation-based approaches for functional verification. We emphasize that the purpose of simulation is functional verification instead of other usages such as design space exploration, performance analysis or power analysis. Therefore,

- The simulation environment must be accurate enough to capture the behavior of the implemented design. Simulation accuracy is crucial to reveal design defects. Mismatches between the simulated and the implemented design can lead to the identification of false positive and false negative bugs during simulation. In contrast, simulation for performance analysis purposes may hide details of a design so as to improve simulation speed.
- To improve simulation and debug productivity, the simulation should not include information that is irrelevant to detecting functional bugs. For example, we do not collect signal toggle rate in simulation, which is only useful for power analysis. Similarly, it is also not necessary to consider placement and routing information in simulation.
- The simulated design must be implementation-ready and simulation should verify the design itself instead of a variation of the design. In particular, for RTL simulation to be effective, it is desirable to implement the simulated and verified design without any changes.

Although the above-mentioned requirements look straightforward, they are not easy to fully comply with in practice. For example, some designs may contain third-party IPs, which are delivered as netlists or hard macros. Simulation using netlists or hard macros is performed at too low a level and contains information that is irrelevant to detecting functional bugs. Alternatively, the design can be simulated using a functionally equivalent simulation model, which may not be completely accurate. As will be described in Chapter 2, the conflict between simulation accuracy and productivity is much more

difficult to resolve for DRS designs. Therefore, designers for either static or DRS designs need to strike a balance between simulation accuracy and productivity in simulation and the simulated design should be implementation-ready.

Tools and Methodologies Objective. Following the Functional Verification Objective, this thesis aims to provide designers with simulation tools to verify DRS designs. The tools should be able to model and simulate the reconfiguration process of DRS designs. Furthermore, similar to the verification of static FPGA-/ASIC-based designs, there is no push-button solution that automatically guarantees the correctness of any DRS design. From a methodology perspective, it is more significant for designers to integrate the simulation tools with their design flow, and to effectively use the simulation tools to *assist* in detecting bugs. In particular:

- We emphasize the use of simulation tools to test and debug *integrated* DRS designs instead of simulating DPR as an isolated process.
- We cannot provide a complete list of all possible bugs that could exist in DRS designs. Rather, we aim to provide guidelines and examples to demonstrate the effective use of tools in detecting DPR-related bugs.
- Our simulation tool cannot and does not aim to replace other debugging methods such as on-chip debugging. Nevertheless, we aim to inspire researchers and designers to consider the value of simulation to facilitate functional verification of DRS designs, thereby reducing, if not eliminating, reliance on costly on-chip debug cycles. Generally speaking, it is essential for designers to understand the pros and cons of various tools and to choose the right tool at the right time, in order to minimize the number of design iterations.

It should be noted that the tools and methodologies aim to cope with the new challenges of verifying DRS designs so that the functional verification of DRS designs is *as easy as* that of static designs. In this regard, verifying static designs itself is not easy but is out of the scope of this thesis. Nevertheless, this thesis aims to bridge the gap between verifying DRS designs and verifying traditional static designs.

1.2 Thesis Contributions

As described earlier, traditional simulation does not support swapping modules during simulation. In addition, simulation cannot interpret real bitstreams that are used to reconfigure the target FPGA. The core idea of this work is to use a simulation-only layer, a collection of behavioral models, to emulate the FPGA fabric and the DPR-related activities (known as characteristic features of DPR) such as swapping modules and interpreting bitstreams. Using the simulation-only layer, designers can thereby exercise various DPR-related scenarios in order to expose functional bugs in the DRS design while part of it is undergoing reconfiguration. For example, we use a simulation-only

bitstream (SimB) to substitute for a real bitstream in simulation so that the process of transferring bitstreams and the subsequent module swapping can be realistically simulated and tested. The key contributions of this thesis are:

- We propose a generic method for modeling DPR, namely using the simulation-only layer. As has been described, simulating DRS designs using the simulation-only layer assists designers in detecting DPR-related functional bugs, thereby fulfilling the Functional Verification Objective.
- From a CAD tool perspective, we propose the ReSim library to support RTL simulation of DRS designs, and extend its core ideas to an Extended ReChannel library for the high-level modeling of DRS designs. The two libraries are two possible implementations of the concept of the simulation-only layer. In accordance with the Mainstream Objective, the two libraries are seamlessly integrated into the existing and mainstream DRS design flow.
- From a methodology perspective, we integrate ReSim and Extended ReChannel with well-established verification approaches, such as top-down modeling and coverage-driven verification, for the verification of DRS designs. We provide a comprehensive set of case studies to demonstrate the use of our integrated verification tools and methodologies, thereby fulfilling the Tools and Methodologies Objective.
- Release of the ReSim library as an open source tool together with examples of using the tool. ReSim is publicly available via `http://code.google.com/p/resim-simulating-partial-reconfiguration` .

1.3 Thesis Outline

The rest of this thesis is organized as follows. After reviewing state-of-the art DRS designs, Chapter 2 analyzes the challenges in verifying DRS, and assesses related verification approaches according to the challenges identified.

Chapter 3 sets forth the general concepts of the simulation-only layer. We analyze the capabilities and the limitations of our idea with regard to the verification challenges identified in Chapter 2.

Chapter 4 applies the idea of a simulation-only layer to high-level modeling and RTL simulation. Our work on high-level modeling extends a previous open source SystemC library, ReChannel [67] with concepts of the simulation-only layer. We created a ReSim library from scratch for RTL simulation of DRS designs.

We present the use of our tool on a range of case studies in Chapter 5. The case studies include in-house designs, a third-party design and reference designs. We illustrate the development progress and the bugs detected for each case study as examples to guide future designers in verifying their DRS designs. We also assess the development

overhead and simulation overhead of using our library. We demonstrate that ReSim can be seamlessly integrated into mainstream DRS design flows.

The last chapter concludes the thesis and discusses directions for future work in verifying DRS designs.

The thesis has one Appendix, which describes all DPR-related bugs detected in the case studies. This Appendix can be used as a reference by future designers to help understand the potential source of bugs in DRS designs.

Chapter 2

Background and Related Work

The purpose of this chapter is to provide background related to verifying dynamically reconfigurable FPGA designs. The chapter starts with a general discussion on functional verification techniques (Section 2.1). By comparing DRS designs with traditional FPGA-based static designs, we then discuss the challenges in verifying DRS designs (Section 2.2). Sections 2.3 and 2.4 review previous work on verifying DRS designs using simulation and non-simulation methods, respectively, with the goal of emphasizing the shortcomings of previous verification approaches. The last section summarizes the chapter.

2.1 Functional Verification of FPGA-based Designs

Figure 2.1 illustrates a typical design flow for hardware systems targeting reconfigurable devices. Starting with the design intent, the designer creates, using human language, a specification, which is transcribed into a Register Transfer Level (RTL) description using Hardware Description Languages (HDL). The conversion process could also involve Intellectual Property (IP) from third parties or previous projects, and the IP can be represented as HDL code or synthesized macros. The designer then creates constraints and the captured design is synthesized and implemented using FPGA vendor tools (e.g., Xilinx ISE [97]). A design can also be captured by high-level languages such as SystemC [40] and mapped to the target FPGA using High-level Synthesis (HLS) tools (e.g., Vivado-HLS [102]). After this sequence of steps, the implemented design is ready to run on the target device.

To ensure the correctness of the design, each step needs to be verified and any mismatch in the representation between two consecutive steps is known as a design error or a bug. Assuming that the vendor tools are reliable, bugs introduced in the implementation process (e.g., incorrect design constraints, timing violation, etc) can be identified and eliminated with the assistance of vendor tools [97]. On the other hand, bugs introduced to the specification and the captured design are known as functional bugs, and functional verification is the process of identifying and fixing functional bugs so that the captured design meets the design intent [66]. It should be noted that, in general, a captured FPGA

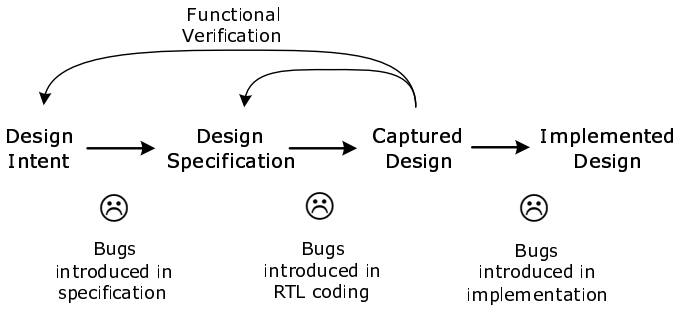


Figure 2.1: Typical hardware design flow

design can also target ASIC technology without much change. Therefore, the techniques for functionally verifying a captured hardware design can, in general, be applied to both ASIC-based and FPGA-based designs.

Simulation is the most common method for verifying hardware design functionality. By feeding the simulated design with selected stimuli, the designer checks the response of the Design Under Test (DUT) and fixes bugs exposed by the simulated scenario. Figure 2.2 illustrates the tradeoff between simulation accuracy and verification productivity when simulating a design at various levels of abstraction (i.e., high-level modeling, RTL simulation and timing simulation). Generally speaking, the more details that have been included in simulation, the more accurate the simulation is. However, since the design is modeled in more detail, the simulation throughput (i.e., the number of simulated cycles per elapsed second) decreases and it is often more time-consuming for designers to trace the root of a bug when the simulation fails. Therefore, verification productivity decreases with increasing simulation accuracy.

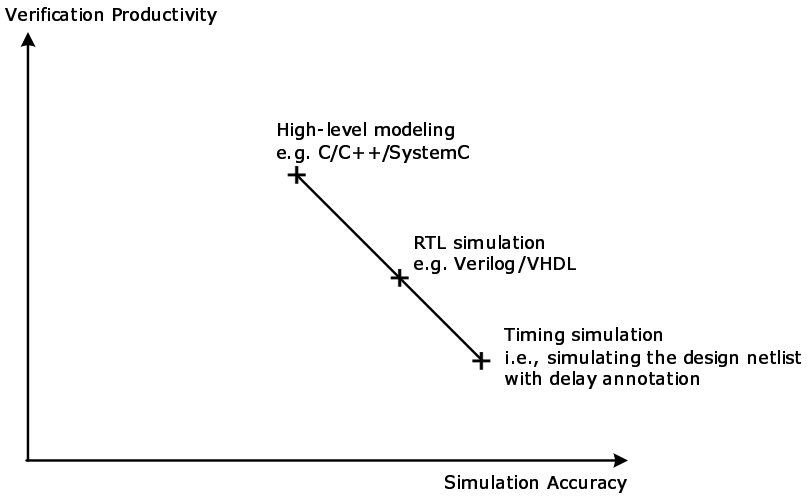


Figure 2.2: Tradeoff between simulation accuracy and verification productivity (for static designs)

Hardware designs can be modeled and simulated using high-level languages such as SystemC, which describe the design at a level of abstraction higher than RTL. Although the simulation is not cycle-accurate, high-level modeling is accurate enough to verify hardware architecture (i.e., the execution and communication flow of submodules) and run embedded software before the design is captured in more detail. High-level modeling

therefore facilitates early system integration. Furthermore, the simulation throughput of high-level modeling is orders of magnitude higher than for RTL designs and high-level models are easier to debug. Therefore, high-level modeling sacrifices simulation accuracy for the sake of verification productivity (see Figure 2.2).

Through a top-down modeling and refinement process, the verified model is typically manually converted to the synthesizable RTL design, which is optimized for performance, area and power consumption. A significant overhead of high-level modeling is the need to create a high-level model solely for verification purposes. To reduce this overhead, there is a growing trend to synthesize the high-level model directly to the target FPGA [20]. However, in order to achieve the desired level of performance, resource utilization and power consumption, HLS requires many more design iterations than RTL-based design flows and is still not widely accepted in industry. While high-level modeling is widely used as a verification method to assist the designer in detecting functional bugs, it is not commonly used for synthesis.

At the other end of the accuracy-productivity spectrum is timing simulation, which back-annotates the design netlist with timing information in simulation [99]. Timing simulation assists in identifying incorrect timing constraints, bugs on asynchronous paths, and verifies timing closure. Since the timing information is extracted from the implemented design, timing simulation is dependent on the implementation process. Therefore, timing simulation sacrifices productivity for simulation accuracy (see Figure 2.2).

Simulation is most commonly performed at RTL level. Using the high-level model as a reference, the RTL design is described using HDL languages and focuses on signal interactions, data flow and timing optimization, which are details not captured in the reference model but required for implementation. Therefore, RTL simulation focuses on verifying signal interactions and the verified design is ready to be implemented. It should be noted that a designer can also bypass high-level modeling and directly describe the design at RTL level. Referring to Section 1.1, RTL simulation meets the three requirements of the Functional Verification Objective.

- Since the simulation is *cycle-accurate*, RTL simulation can expose bugs introduced either in the specification or RTL coding, as long as the bug is exercised by the stimuli. However, the limited throughput of RTL simulation only allows exercising the design with selected scenarios. The choice of scenarios needs to be carefully planned to maximize the chances of exposing bugs.
- Since the simulation is *physically independent*, RTL simulation does not rely on any results from the implementation process or on the details of the target device to which the design is intended to be mapped. Therefore, the simulation can be performed before the time-consuming implementation step and the iterative design-simulate-debug process does not include the lengthy implementation step. Moreover, physically independent simulation improves debugging productivity, during which designers are only interested in identifying and fixing potential functional bugs in the captured design. Last but not least, physically independent simulation has acceptable simulation throughput since the design is modeled at the right level of detail to detect functional bugs.

- Since the simulated design is *implementation-ready*, the verified design can be synthesized and implemented using vendor tools without any changes to the design source. RTL simulation therefore verifies the design intent.

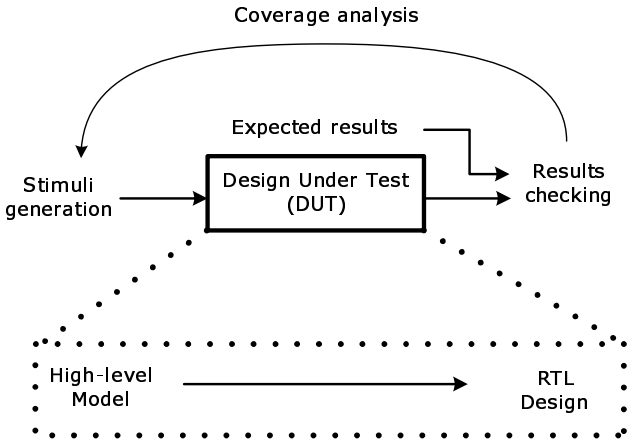


Figure 2.3: A typical simulation environment

Apart from the design to be simulated and verified, the simulation environment also includes components for stimuli generation, result checking and coverage analysis (see Figure 2.3). In particular, the designer needs to create tests to exercise various execution scenarios of the DUT, and checks the responses of the DUT against the expected results. Since RTL simulation can only exercise a limited number of scenarios, a well-managed design project adopts coverage-driven verification to guide stimuli generation so as to increase the coverage of simulation [66]. The starting point of coverage-driven verification is a test plan, which describes (using natural languages) a list of scenarios that need to be tested. Next, the designer maps the scenarios of interest to coverage items (using formal languages), which can then be compiled and kept track of by the simulator. By analyzing the coverage data collected by the simulator after a simulation run, the designer identifies coverage items that have not yet been stimulated (i.e., *coverage holes*) and creates more tests targeting the uncovered scenarios. The functional verification ends by covering all required scenarios without failure.

Although state-of-the-art simulators assist designers in collecting coverage data, coverage analysis still involves manpower to identify all scenarios to be stimulated and to map the scenarios to coverage items that can be tracked by a simulator. Coverage items can either be converted from the design source code (i.e., *code coverage*) or be derived from the design specification (i.e., *functional coverage*). Therefore, apart from the HDL source code, the designer explicitly defines functional coverage items using Hardware Verification Languages (HVL, e.g., SystemVerilog [36]), and functional verification ends when all scenarios defined by HDL and HVL items are covered.

IP-reuse and platform-based design are two methodologies that aim to improve design productivity. IP-reuse significantly reduces verification effort by integrating or re-integrating proven IP from previous projects [68]. However, while thoroughly verified sub-modules and IP are essential, they do not guarantee the correctness of the integrated system [3]. It is therefore highly desirable to build a simulation environment capable of testing the integrated design.

By parameterizing and customizing a generic computing platform, a platform-based design methodology can meet the needs of a variety of applications from a specific domain, thereby reducing the associated development effort [73]. A platform-based verification approach can reuse the simulation/verification platform from previous projects so as to verify newly added modules to a derived design [68]. However, various applications may expose defects in corner cases from different parts of the platform, and it is therefore essential to thoroughly verify all execution scenarios of the platform.

It should be noted that the above-mentioned methodologies are only guides and that the designer needs to adapt them to the requirements of specific projects. For example, if the design itself is relatively small, the overhead of creating a high-level model may outweigh its benefit. On the other hand, if the system uses legacy IP from previous projects, it is not necessary to perform coverage analysis over those components, but it is still essential to test the integration of the proven IP with the new project.

Apart from simulation, a design can also be verified using formal methods and field testing. Formal methods thoroughly explore all possible scenarios and prove that the design either matches or does not match the specification. The specification can contain a list of design properties captured by formal languages such as SystemVerilog Assertions [36, 87] and Property Specification Language [39] (i.e., property checking), or represent the same design at a higher level of abstraction (as used in equivalence checking) [23]. However, the run time of formal methods do not scale well with design size, and are therefore only suitable for small designs [68]. Furthermore, there is no straightforward way to verify the correctness and the completeness of the formal specification itself. In particular, a bug can be introduced to the formal specification or it may only capture a subset of the required design functionality. As a result, formal verification is typically used to supplement simulation [68].

Field testing runs the implemented design on the target device under real-world conditions. However, tracing the cause of a bug on the implemented design requires extra effort to insert probing logic using vendor tools (e.g., ChipScope [95] and SignalTap [7]) and the design needs to be re-implemented every time a different set of user design signals is to be probed. The debug turnaround time of on-chip debugging is therefore at least as long as the time-consuming implementation stage. Furthermore, since probing logic can only visualize a limited number of signals for a limited period of time, on-chip debugging typically involves more iterations to identify the source of a bug than simulation requires. Last but not least, collecting and analyzing coverage data on chip is not guided by quantifiable metrics. As a result, it is essential to perform thorough simulation-based functional verification to detect as many logic errors as possible before testing the implemented design on the target FPGA [99].

2.2 Challenges in Verifying DRS Designs

Using DPR, hardware modules that do not need to run simultaneously can be time-multiplexed. For example, considering a traditional FPGA-based static hardware design (upper half of Figure 2.4), two modules whose temporal activities are mutually exclusive

are both statically configured on the FPGA. Since only one of the modules is needed at a time, a multiplexer is used to interleave the communication between the selected module and the rest of the design.

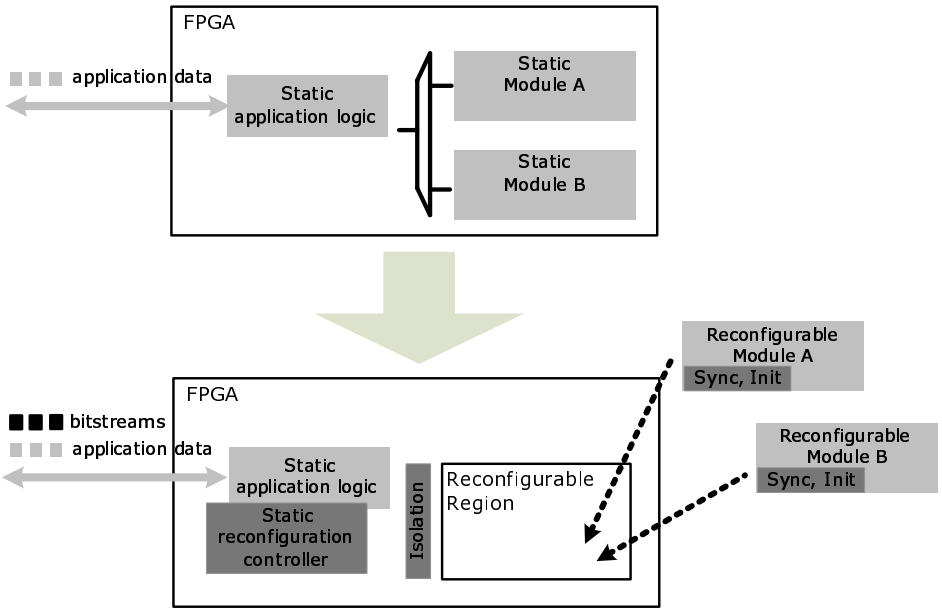


Figure 2.4: User design layer of a DRS design

In the DRS version of the same design (lower half of Figure 2.4), the two modules are mapped to the same physical region and are time-multiplexed by partial reconfiguration instead of being selected by a multiplexer. The static region of the DRS design is comprised of the common part of each configuration and is kept intact during the whole system runtime. Since only the required module is loaded to the FPGA, partial reconfiguration saves resource usage, which is more significant if a large number of RMs are used and each RM requires a large amount of FPGA resource. Since the reconfiguration delay is significantly longer than selecting the multiplexer in the static version, DPR is typically beneficial to designs that do not need frequent reconfiguration. Compared with the static version, the DRS version of a design adds synchronization and initialization logic to the RM to pause and initialize the RM before and after reconfiguration. In the static region of the DRS, a reconfiguration controller is added to transfer configuration bitstreams and the region undergoing reconfiguration is isolated. The reconfiguration controller can also reside off-chip and perform DPR via an external configuration port. We refer to the logic that performs the *synchronization*, *isolation* and *initialization* of RMs as well as the *bitstream transfer* as the reconfiguration machinery (moderately shaded blocks in the figure). Jointly, such reconfiguration machinery manages the reconfiguration process.

Figure 2.5 illustrates the reconfiguration process of DRS designs targeting commercial and off-the-shelf FPGAs (e.g., Virtex series FPGAs [94, 100, 101, 104]). During reconfiguration, configuration bitstreams (depicted by a sequence of small black squares alongside communication links in the figure) are transferred either by the static design (i.e., the static reconfiguration controller), or by an external host (i.e., the external reconfiguration controller) to overwrite selected configuration bits stored in the configuration memory. By the end of bitstream transfer, a new module (i.e., reconfigurable

module B) is swapped in to replace the old module (i.e., reconfigurable module A). It should be noted that although the latest Stratix V FPGA devices from Altera support partial reconfiguration [6], Altera has not yet integrated the DPR design flow into their CAD tools. However, the modular/ECO-based reconfiguration approaches proposed by Altera also involve a similar reconfiguration process [5].

Figure 2.5 indicates that DRS designs have two conceptual layers:

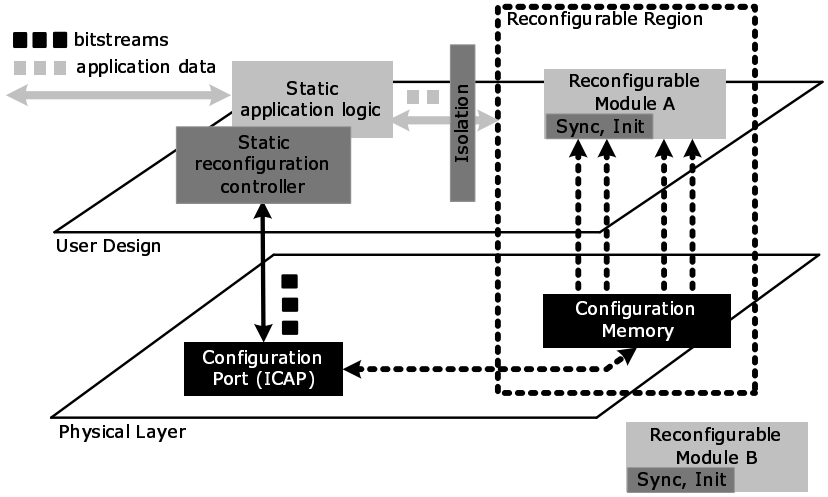


Figure 2.5: Conceptual diagram of a DRS design

- **User Design Layer:** The user design comprises all user-defined modules. These include the application logic (lightly shaded blocks) performing the required processing tasks of the application, as well as the reconfiguration machinery (moderately shaded blocks) that interact with the physical layer and manage the reconfiguration process.
- **Physical Layer:** The physical layer (darkly shaded blocks) contains elements of the FPGA fabric, such as the configuration port (e.g., Internal Configuration Access Port (ICAP) or SelectMap [94, 100, 101, 104]), the configuration memory, and configuration bitstreams, the implementation of which are proprietary to the FPGA vendors.

Compared with traditional static designs, both the user design and physical layers introduce new challenges to the functional verification of DRS designs. The rest of this section analyzes the challenges presented by both layers. Following the Mainstream Objective of this thesis (see Section 1.1), this section focuses on the challenges in simulation-based functional verification of DRS designs.

2.2.1 Verification Challenges Arising from the User Design Layer

Compared with static designs, the user design layer of DRS designs include extra reconfiguration machinery, and the designer needs to verify that the reconfiguration machinery

is correct and is correctly integrated with the rest of the system. As illustrated in Figure 2.5, the reconfiguration machinery only operates during the reconfiguration process. Therefore, it is essential to exercise various reconfiguration scenarios in simulation so as to test and verify the reconfiguration machinery. From a temporal perspective, reconfiguration scenarios can be categorized according to the phase of the partial reconfiguration process during which they occur, i.e., BEFORE, DURING, or AFTER reconfiguration. We describe the three phases in more detail below.

BEFORE reconfiguration. This phase spans the time interval from when a request for reconfiguration occurs until just before the first byte of the bitstream is written to the configuration port. During this phase, proper *synchronization* is required to pause the on-going computation between the static region and the RM. One approach is to reconfigure the RM immediately such as in fault tolerant applications, in which a faulty module is discarded and then reconfigured immediately by a fresh copy of the module [35]. In hardware multitasking, a module preempted by the Operating System can save its execution state for later restoration [80], or can roll back to its previous known state when resumed later [70]. Another approach is to wait until current processing is finished, such as in a video-based driver assistance system, in which each frame is needed to process the driving conditions and reconfiguration is blocked until the end of processing a frame [18]. In some pipelined designs, the RM needs to drain all pipelines of data before acknowledging the reconfiguration. Whatever approach is taken, the designer needs to verify the chosen synchronization mechanism is working as expected and should exercise a number of scenarios. Some examples follow.

- For each individual reconfiguration operation, requests for reconfiguration may occur at any time (i.e., when the RM is either idle or busy) and the occurrence in each possible state of the RM should be exercised. For example, a reconfiguration request may occur precisely when the RM starts or ends computation and the system may not be able to process such concurrent events correctly.
- With respect to a sequence of reconfiguration operations that occur in a system, all possible reconfiguration *sequences* should be exercised. For example, an RM may receive multiple reconfiguration requests that conflict with each other, such as to unload an already unloaded module [69] or to cancel an acknowledged reconfiguration. If the design has multiple RRs, illegal sequences of reconfiguration requests could deadlock the system.
- For designs that save module state BEFORE reconfiguration, various scenarios need to be exercised to test the robustness of state saving. For example, the state saving process can be interrupted or delayed by other activities (e.g., higher priority bus traffic or interrupts). In the process of state saving, the RM may not have been kept frozen and the saved state could become inconsistent.
- Reconfiguration requests can arise from within or from outside of the user design. For example, depending on the desired tasks, an FPGA-based robot platform dynamically switches between internal timer-driven reconfiguration mode and external event-driven reconfiguration mode [60]. The designer therefore needs to

exercise all possible reconfiguration sources to verify various reconfiguration modes of the system.

DURING reconfiguration. This phase spans the period during which the configuration bitstream is being *transferred* to the configuration port of the FPGA. During this phase, the bitstreams may require processing such as decompression and decryption [13], whereas bitstream transfer could be performed over a dedicated datapath [52], or over a shared bus [93].

During reconfiguration, the behavior of the logic inside the RR is undefined and should be properly *isolated* to prevent the propagation of spurious RM outputs and/or to avoid breaking ongoing operations of the static part. The simplest design option for isolation is to drive a default value to the static region during reconfiguration (e.g., assert the module's reset signal throughout reconfiguration [75]). In case the static region is a master, extra logic is required to indicate the availability of the RM. Such extra logic can be a single signal from the slave, or a time-out counter in the master. In some complex designs such as the reconfigurable video processing engines of the AutoVision system (studied in Chapter 5), the slave registers have to be moved from inside the RM to the static region so that the unavailability of the RM does not break the daisy chain of software-accessible registers [4]. Data arriving during reconfiguration might be discarded [91] or buffered in the system for further processing [76]. To verify the isolation mechanism used by the system, the simulation needs to exercise various scenarios. For example,

- If the design has multiple RRs and there are communication events between the RMs associated with the RRs, all legal transitions between two RMs in the same RR need to be exercised and each transition should be simulated with all possible configurations of other RRs. Furthermore, all legal reconfiguration sequences must be tested so as to, for example, exercise potential deadlock scenarios.
- All possible scenarios for bitstream processing and transfer need to be covered. For example, if the bitstreams are to be decrypted, the length of the bitstream may or may not be a multiple of the size of the encryption key. If using a shared bus to transfer bitstreams, simulation should cover the potential traffic contention between bitstream traffic and application data. The bitstream itself could be corrupted, for example, by an invalid pointer in the system software.
- The scenarios in which requests for communication are attempted from the static part to the RM, or vice versa, should be exercised in simulation to verify that the static region and the RR are adequately isolated from each other. In particular, if the static part relies on feedback from the RR, failure to cancel a communication attempt may cause the static logic to wait for a response that will never arrive.
- Since the timing of isolation is critical, all possible scenarios of signal transitions need to be simulated. In particular, isolating one cycle too early may cause the static module to fail to transmit the last dataset to the RM, and isolating one cycle too late may cause the RM to inject spurious data into the static part.

- If the static region buffers incoming data while the RM is being reconfigured, the overflow and underflow of the buffer should be exercised during simulation.

AFTER reconfiguration. This phase spans the time interval from just after the last byte of the bitstream is written until the new module is activated. In the last phase of reconfiguration, the designer needs to verify the method used to *initialize* the newly downloaded RM, which may involve simply resetting the RM or restoring the RM to a previous execution state. Module state can be restored via the configuration port [80] or via user logic [46] and can be restored from previously saved values [80] or from other modules [44]. In complex designs such as the fault recovery application of [35], a special instruction and interrupt is added to a softcore microprocessor to restore its internal registers. The scenarios of interest to partial reconfiguration include:

- As the module can be in any possible state AFTER reconfiguration, in order to thoroughly test reset-based initialization, the design should be reset from any possible module state in simulation.
- The reset operation should be combined with the operation of the rest of the system and all possible combinations need to be tested. For example, a scenario of interest is attempting to communicate with the RM immediately before, during and after the reset operation.
- For pipelined RMs, their state can only be initialized by loading the pipelines with meaningful data, and the designer needs to verify that the undefined data flushed out from the pipeline are not inadvertently propagated to the static system [69].
- For designs restoring module state, all possible scenarios for restoring the module state need to be tested. For example, the static region should attempt to interrupt or delay the restoration process. Restoration should be initiated in various system states so as to test that the restoration does not affect the rest of the design.

* * * * * * * * * * * *

For all three phases, partial reconfiguration scenarios need to be properly simulated and thoroughly tested. It should be noted that the above-mentioned scenarios are only guides and the designer needs to adapt them to specific DRS design projects. In simulation, designers need to select stimuli to exercise scenarios of interest.

2.2.2 Verification Challenges Arising from the Physical Layer

The physical layer of a DRS design represents the FPGA device, which contains the logic/routing resources comprising the fabric of the device, the configuration memory controlling the function and interconnection of the fabric, and the configuration distribution network commencing with the configuration port. Although the organization of the FPGA fabric differs between FPGA families, the general concept of a basic building block of the fabric can be illustrated as in Figure 2.6 [19]. A basic building block has a

Look Up Table (LUT, see the `4-LUT` module) to implement combinational logic, a flip-flop (see the `DFF` module) to implement sequential elements such as pipeline registers or Finite State Machines (FSM), and Carry Logic to implement carry-related computation (e.g., add, wide AND, etc). A basic block contains multiple *configuration points* (depicted as `Pn`) that control the logic settings (e.g., P1, P2, P4), switch settings (e.g., P5) and memory elements (e.g., P3) of the fabric. Changing these configuration points leads to different user designs being instantiated. For example, user design A can be mapped to the fabric by setting the LUT to an AND gate and enabling the flip-flop while user design B is mapped by configuring the LUT to be an OR gate and bypassing the flip-flop. All configuration points are connected to an SRAM-based *configuration memory*, which can be accessed via the *configuration port* (e.g., Internal Configuration Access Port, ICAP) of the FPGA. In particular, configuration settings can be overwritten by transferring a *configuration bitstream* to the configuration port, from which the bitstream is internally parsed and distributed to the configuration memory.

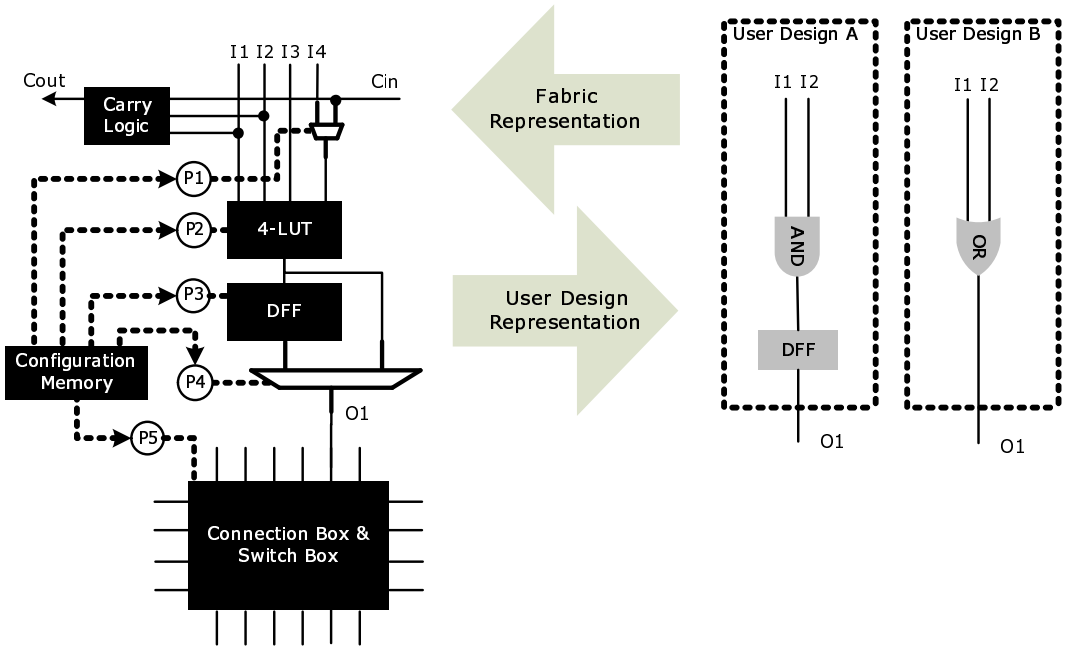


Figure 2.6: Basic Logic and Routing Block of FPGAs, after [19]

In traditional FPGA-based hardware designs, the physical layer is statically configured and is not visible to the user design. Bitstreams are only transferred at power up and are typically loaded by off-chip controllers, which are not part of the on-chip user design. In DRS designs, the target FPGA is reprogrammed at run time and the reconfiguration machinery, either on-chip or off-chip, is part of the user design. Therefore, the user design, and the reconfiguration machinery in particular, interacts with the physical layer in the reconfiguration process. We refer to such interactions as *inter-layer interactions* (see the links between the two layers in Figure 2.5) and the configuration bitstream is the *medium* for such inter-layer interactions. In particular, it is the configuration bitstreams that are transferred by the user design that subsequently change the configuration settings of the FPGA fabric and thus the functionality of the user design. Depending on how partial reconfiguration is performed, the level of interactions between the user design and the FPGA fabric varies.

Modular Reconfiguration. One way of performing partial reconfiguration is modular reconfiguration [98][6], which is supported by mainstream vendor tools (e.g., PlanAhead [98]) and academic tools (e.g., GoAhead [49], OpenPR [82]). For a modular reconfigurable DRS design, RMs are swapped in and out as atomic entities. Limited by the capability of vendor tools, the affiliation between RMs and RRs is defined at design time and therefore modules cannot be relocated in DRS designs created by vendor tools. Modular reconfiguration is the most straightforward and mainstream way to design a DRS and it benefits applications in various ways.

- Using modular reconfiguration, an application can update its modules at run time to adapt to changing execution requirements. For example, a video-based driver assistance system dynamically swaps image processing engines when the driving conditions change between highway, country, urban and tunnel driving [18]. An automobile cabin controller use modular reconfiguration to swap in the required control logic on demand [86]. Depending on the desired task, an FPGA-based robot platform dynamically switches from target acquisition mode to target tracking mode by reconfiguring itself with various computer vision modules [60]. A scalable Discrete Cosine Transfer (DCT) core dynamically reconfigures the size of its DCT array (1*1 – 5*5) according to the required Signal-to-Noise Ratio (SNR) [34], which varies between day and night conditions. An FPGA-based Monte-Carlo Simulator adaptively adjusts its simulation precision by reconfiguring its number representation mechanism at run time [84]. Modular reconfiguration is used to dynamically balance workload of a network processor [43]. The network processor keeps track of the number of each packet type, and if a particular traffic type is increasing, then it reconfigures more of its eight co-processors to process that particular traffic type.

- Some applications can be partitioned into a few sequential steps and an RM can be reconfigured for each step using modular reconfiguration. For example, A DVB-T2 decoder time-multiplexes a demodulation module and an error correction module to decode wireless signals [25]. The two modules are mapped to the same physical region of the FPGA and are executed sequentially. For an FPGA-based sorting accelerator, a large sorting problem is divided into two execution stages, in which a FIFO-merge-sort engine and a Tree-merge-sort engine are reconfigured respectively [48].

- Modular reconfiguration can also be applied to change the communication infrastructure for hardware modules at run time. For example, the routers of the CoNoChi [65] Network-on-Chip (NoC) can be removed or inserted at run time to reroute the communication path. Instead of looking up the destination node as traditional routers do, the R2NOC router directly connects the initiating node with the destination node, and such direct connection is reconfigured at system run time using modular reconfiguration [22].

In modular reconfigurable DRS designs, since each RM can only be mapped to its predefined RR, the inter-layer interactions are mediated by the partial bitstream in its entity and individual configuration bits are of limited interest to the designer. Since

each RM is reconfigured as an entity, designers are not required to be experts on the relationship between the configuration memory bits and the elements of the FPGA fabric the configuration bits control.

State Saving and Restoration. Apart from reconfiguring module logic, state saving and restoration (SSR) is a common and sometimes essential requirement of DRS. For FPGA-based designs, state data include the contents of storage elements such as flip-flops and memory cells such as Block RAM (BRAM) located in the FPGA fabric. In DRS designs, state data can be saved from RMs before reconfiguration for later restoration [19]. For example, to recover from an error, a module can be restored to a saved checkpoint [46]. For periodic/phased applications, the system configures the required computational module for each period of execution and copies the results across periods [42]. In hardware multitasking, a module can be preempted and later resume execution from the interrupted state [80]. To defragment FPGA resources, modules can be relocated from their original locations to other areas of available resources [44]. The FPGA accelerator platform in [71] uses the Message Passing Interface (MPI) protocol to communicate with the host PC and state data of RMs are saved and restored by passing messages.

To save and restore module state, the designer needs to create a datapath to access the state. There are currently two approaches for creating such a datapath. The first method adds design-specific application logic to read/write storage elements (e.g., [71], [46], [58], [33]). Using this approach, state saving and restoration does not involve any inter-layer interactions. Alternatively, the other approach utilizes the configuration port of the FPGA to access state data (e.g., [44], [80]). In particular, after reading back the configuration bitstream, bits that correspond to storage elements are extracted from the bitstream and are saved in a database. These state bits are later restored to the RM during partial reconfiguration. Therefore, the media for cross-layer interactions are the state bits of configuration bitstreams.

Comparing the two methods, it is more general to save and restore module state via the configuration port since this method can access arbitrary storage elements without changing the RMs. Therefore, it does not introduce any side effect to the system such as extra resource usage or tighter timing requirements. However, the designer needs expert knowledge of the FPGA fabric in order to correctly design a system that extracts/restores state data from/to configuration bitstreams. In contrast, customized state access logic needs to be carefully inserted for each module in the system.

Module Relocation. Since the FPGA fabric has, in general, a regular structure, RMs do not have to be restricted in module placement. Module relocation can significantly increase the design flexibility of DRS. For example, an FPGA-based database query application dynamically assembles its datapath with relocatable modules from a module library to perform various queries [21]. Run-time management systems have been proposed to dynamically allocate and defragment FPGA resources [89] [26] so as to increase resource utilization.

Various methods have been proposed to enhance the relocatability of RMs. For example, the REPLICA filter is able to change the module placement information of configuration bitstreams to a different location and to restore the execution state of the relocated module [44]. By constraining RMs to use resources that are identical between the source and the target location, modules can be relocated on FPGAs with heterogeneous resource distribution [10]. Furthermore, this work provides software APIs to manipulate the placement information of modules at run time so as to relocate RMs.

Apart from relocating the RMs, the communication infrastructure of the system also needs to be carefully designed so that modules can be correctly connected to the rest of the system before and after relocation. The ReCoBus approach proposes a bus-based interconnection mechanism with a regular routing structure so as to relocate RMs to compatible reconfigurable slots [47]. In a sandbox-based approach, RMs can be placed and relocated within a large pre-defined RR and module interconnections are routed at run time to support the changing communication requirements between RMs [62].

For DRS with relocatable RMs, a bug in processing the placement information could cause an RM to be allocated to a location that overlaps another RM, and a bug in processing the routing information could cause the relocated module to be connected to incorrect logic. Correct execution of the user design thus relies on correct processing of placement and routing information from the physical layer.

Fine-grained Reconfiguration. Since the SRAM-based configuration memory cells can be individually programmed, reconfiguration does not have to be performed in a modular manner. A fine-grained reconfigurable DRS design only updates a few configuration bits of the configuration memory so as to change the design functionality. For example, by updating the content of LUTs, a network measurement application dynamically updates the rules against which the incoming network packets are checked [45]. A LUT-based, fine-grained reconfigurable switch box is used to dynamically change the communication patterns of NoCs [32]. The system determines the desired configuration bits of such LUT-based switch boxes and generates bitstreams at run-time to update the LUTs. To support fine-grained reconfiguration in processor-based FPGA designs, [12] creates a set of software APIs to manipulate configuration bits of the FPGA fabric. In these designs, the medium of cross-layer interactions are the few configuration bits that control the logic and routing of the FPGA fabric.

Since only a very small portion of the design is required to be updated, fine-grained reconfiguration significantly reduces the reconfiguration overhead. However, fine-grained reconfiguration requires the designer to have expert knowledge of the FPGA fabric so that correct configuration bits are identified and updated. Furthermore, since the organization of the FPGA fabric varies between different FPGA families and vendors, fine-grained reconfigurable DRS designs are, in general, not portable.

Coarse-grained Reconfiguration. Compared with the bit-level configurability of FPGAs, DRS designs can also target coarse-grained reconfigurable architectures. In

particular, the datapath of such systems contain optimized and commonly used processing blocks of a specific application domain and reconfiguration is performed by selecting multiplexers or multiplexer-like structures on the datapath. For example, FlexiTreP is an Application-Specific Instruction Set Processor (ASIP) for channel decoding in wireless communication [88]. Similarly, the reconfigurable functional unit presented in [31] is composed of coarse-grained reconfigurable modules optimized for Wireless Sensor Network applications. The DReAM architecture [15] is a dynamically reconfigurable DSP processor targeting baseband processing for software-defined radios.

Since coarse-grained reconfigurable DRS designs do not target off-the-shelf FPGAs, the designers need to define the organization of the physical layer. The configuration bitstreams are loaded through customized configuration mechanisms (e.g., software-accessible registers). The multiplexers that select processing blocks are explicitly instantiated in the datapath of the user design. Therefore, there is no clear boundary between the user-design layer and the physical layer for customized coarse-grained reconfigurable DRS designs.

Utilizing the Physical Layer. Apart from storing configuration bits, the physical layer of the FPGA can also be included as part of the user design to perform desired tasks. Hardware modules can utilize the configuration port as a unified access point for all resources of the FPGA so as to transfer application data [72] [79]. A hardwired NOC has been integrated to the FPGA fabric as the communication infrastructure for both application data and configuration bitstreams [30].

External Reconfiguration. Reconfiguration can be performed by the design itself or by an external host. While for externally reconfigured DRS designs, bitstreams are transferred by off-chip reconfiguration controllers, the reconfiguration process is similar to that of internally reconfigured DRS designs. The nature of inter-layer interactions is not affected by the place where the reconfiguration controller resides.

* * * * * * * * * * * *

For modular reconfigurable DRS designs, RMs are reconfigured as a whole and the medium of inter-layer interactions is the bitstream. For non-modular reconfigurable DRS designs, inter-layer interactions can involve configuration bits, state and placement information. In DRS designs that target customized reconfigurable architecture or utilize the FPGA fabric to transfer application data, the user design layer merges with the physical layer. In order to verify the behavior of DRS designs undergoing reconfiguration, it is therefore essential to model and simulate the inter-layer interactions.

2.2.3 Simulation Accuracy vs. Verification Productivity

As described in Section 2.2.1, designers need to cover all possible reconfiguration scenarios so as to thoroughly verify DRS designs. Since the reconfiguration process involves interactions between the user design layer and the physical layer (see Section

2.2.2), exercising reconfiguration scenarios requires the designer to accurately simulate the inter-layer interactions during reconfiguration. Unfortunately, it is non-trivial to model and simulate the inter-layer interactions using traditional simulation techniques. We use RTL simulation as an example to illustrate the difficulties involved.

- As described in Section 2.1, RTL simulation is cycle accurate. However, as FPGA vendor tools do not openly provide simulation models for the physical layer, it is non-trivial for designers to accurately model the impact of the physical layer on the user design layer during the reconfiguration process. In particular, the configuration bitstreams, which are the medium of inter-layer interactions, can only be interpreted by the target FPGA. Without an accurate simulation model of the FPGA fabric, traditional RTL simulations of DRS designs cannot interpret the bitstreams and thus cannot perform *cycle-accurate* simulation of the inter-layer interactions. Without accurate simulation, potential bugs, either arising from the reconfiguration process or from integrating partial reconfiguration with the rest of the system, cannot be detected until the integrated design is tested on the target device.
- On the other hand, even if the simulation model of the FPGA fabric were available, the simulation would be performed at too low a level of detail. In particular, using the simulation model of the FPGA fabric, the user design has to be represented by configuration bits (as in the left half of Figure 2.6) instead of as the desired user logic (as in the right half of Figure 2.6). As a result, simulating the FPGA fabric does not meet the *physical independence* requirement described in Section 2.1. For the sake of verification productivity, it is desirable that functional verification is independent of the FPGA fabric.

An ideal simulation method therefore needs to strike the right *balance* between the level of accuracy of the simulation and the level of detail of the fabric being modeled. Furthermore, this balance is constrained by the desire for the simulated design to be *implementation ready* (see Section 2.1). Thus, the captured design should not be changed for simulation purposes.

It should be noted that for high-level modeling, designers also need to model and verify the behavior of the user design during reconfiguration while abstracting away the internal activities of the FPGA fabric. For formal verification, since a designer needs to describe the intended behavior of the system in the formal specification (see Section 2.1), it is desirable to model the FPGA fabric at the right level of detail so as to formally capture the intended inter-layer interactions.

Referring to the Functional Verification Objective of this thesis and the general description of functional verification in Section 2.1, the requirements for simulation/modeling accuracy, physical independence and implementation-readiness can, in general, be met for static designs. However, since DRS designs involve inter-layer interactions during reconfiguration, these requirements conflict with each other. We therefore conclude that the conflicting requirements of simulation/modeling accuracy and verification productivity constitute the fundamental difficulty in the functional verification of DRS designs,

whatever functional verification methods (i.e., high-level modeling, RTL simulation, or formal verification) are used.

2.3 Simulation-based Verification of DRS Designs

Simulation can be performed at various levels of abstraction (i.e., high-level modeling, RTL simulation, or timing simulation) and it is inevitable that simulation accuracy and verification productivity are traded off (see Figure 2.2 of Section 2.1). However, for DRS designs, it is particularly challenging to find the right balance between accuracy and productivity in simulation (see Section 2.2.3). This section reviews previous simulation-based verification methods.

Virtual Multiplexing. In traditional HDL languages, hardware modules are statically compiled before running simulation and cannot be swapped in the middle of a simulation run. The most common approach for simulating DPR has been to use a Virtual Multiplexer to interleave the communication between reconfigurable modules connected in parallel [53] [54]. During implementation, modules connected by Virtual Multiplexers are implemented separately, and are loaded onto the FPGA as required at run time [54]. The MUX can also be mapped to an actual multiplexer, which switches modules that are statically configured on the FPGA [53].

MUX-based simulation is the basis for more recent efforts in simulating DPR. However, since MUX-based simulation was initially proposed to model partial evaluation applications on XC6200 FPGAs, it fails to simulate the inter-layer interactions of DRS designs targeting more recent FPGAs. For example, Virtual Multiplexing associates module swapping with a constant reconfiguration delay and does not simulate the process of bitstream transfer. As a result, potential bugs on the bitstream transfer datapath cannot be detected in simulation.

Dynamic Circuit Switch. The Dynamic Circuit Switch (DCS) [69] was the first systematic study of simulating DPR. The idea is to automatically add simulation-only artifacts to the RTL code so as to model reconfiguration activities. For example, to model module swapping, the *dynamic task selector* artifact ensures mutually exclusive access to one of the RMs connected in a mutex set. The *dynamic task modeler* artifact forces RM outputs to undefined "x" values so as to model the spurious outputs of RMs undergoing reconfiguration. The *reconfiguration condition detector* artifact monitors designer selected RTL signals to trigger module swapping. In DCS, state saving and restoration is modeled by modifying the RTL description of all storage elements (i.e., flip-flops and memory cells) in simulation.

Since DCS also instantiates RMs in parallel and selects one active module at a time, it can be viewed as being similar to Virtual Multiplexing. On the other hand, a Virtual Multiplexer can be viewed as being an artifact that mimics module swapping. Therefore, DCS can be considered as an extension to Virtual Multiplexing. Compared with [54],

the artifacts of DCS capture more inter-layer interactions of DPR and improve the modeling accuracy. However, the added artifacts do not exist on the target device and the simulated design therefore doesn't replicate what is implemented. For example, on FPGAs, reconfiguration is triggered by writing a bitstream to the configuration port instead of the *reconfiguration condition detector* artifact. Module state is saved and restored via reading and writing bitstreams instead of modifying the storage elements. As a result, DCS changes the design for simulation purposes and the simulation therefore verifies a *variation* of the design instead of the implementation-ready design.

Modifying the Simulator Kernel. Apart from simulating the design at RTL level, DRS designs can also be modeled using high-level modeling languages such as SystemC. Like HDL languages, SystemC does not support modeling run-time reconfiguration. By modifying the SystemC kernel, [14] keeps track of the status of all RMs and stops executing a thread if the corresponding RM is deactivated. The simulated user design performs partial reconfiguration by calling newly added SystemC functions (i.e., `dr_sc_turn_on()` and `dr_sc_turn_off()`), which changes the status of RMs in the modified SystemC kernel. The modified SystemC kernel is used in the design and verification of a dynamic reconfiguration manager [50]. A significant drawback of this work is that the modified SystemC kernel is not standard and is incompatible with the SystemC simulators of other vendors. Furthermore, it does not support the simulation of configuration readback.

ReChannel. ReChannel [67] is a SystemC-based, open source library to model DRS designs. Using a library-based approach, ReChannel is more flexible compared with [14] and can be applied to SystemC simulators from various vendors. In ReChannel, RMs are derived from the `rc_reconfigurable` base class and are connected to the static design through an `rc_portal`, a MUX-like SystemC class. Furthermore, RMs are modeled with novel resettable threads instead of standard SystemC threads so that they can be reset to their default state after reconfiguration. To model state saving and restoration, the library provides call-back functions for user designs to save and restore user-specified variables of the modeled design. To further simplify the modeling of DRS designs, the library has a built-in `rc_control` class that models a user-defined reconfiguration controller, which unloads and activates RMs mapped to an RR. The library also has built-in synchronization mechanisms to properly pause an RM before reconfiguration.

Although the classes and utility functions of the library simplify the modeling of DRS designs, the modeled DRS design does not capture enough design details to assist with debugging the reconfiguration scenarios. For example, since the reconfiguration controller is modeled as a built-in `rc_control` class instead of a user-defined SystemC module, the design-specific details of the reconfiguration controller (e.g., software-accessible registers, interrupts, FIFO depth) can not be verified. As another example, the library uses non-synthesizable call-back functions to model state saving and restoration, while there is no equivalent user logic to implement such call-back functions on target FPGAs. ReChannel is thus insufficiently accurate for verification purposes.

OSSS+R. OSSS+R [74] is a design methodology for automatic modeling, synthesis, and simulation of DRS. This approach uses the polymorphism feature of C++ to model RMs mapped to the same RR. In particular, RMs share a common C++ interface while deriving their module specific implementations. During synthesis, the tool automatically generates reconfiguration control logic to manage the swapping of RMs. Therefore, this approach is more akin to a HLS approach than a functional verification approach. Although HLS significantly improves the design productivity of DRS, the synthesized DRS can only use the pre-defined reconfiguration control mechanism, which potentially limits the design space for DRS.

Dynamically Reconfigurable Fabric (DRCF). The DRCF [64] approach uses a wrapper, a template SystemC class that represents a RR, to model the reconfiguration operations of RMs. In particular, the DRCF wrapper instantiates RMs and has member functions/classes to schedule and swap RMs mapped to it. However, the DRCF wrapper can only be used as a slave attachment to a system bus, which significantly limits the wider use of this approach.

Fabric-accurate Simulation. If a simulation model of the FPGA fabric were available, the user design could be represented by configuration bits and partial reconfiguration could be modeled as the process of rewriting the content of the configuration memory. We refer to this approach as *fabric-accurate* simulation. Fabric-accurate simulation can be viewed as the cycle-accurate simulation of an FPGA device, whereby the configuration data of a user design can be viewed as data items that are processed by the device. As described in Section 2.2.3, fabric-accurate simulation would be completely cycle accurate but would also include a multitude of unnecessary details for functional verification. Since the entire FPGA fabric would be simulated, the simulation speed could even be slower than the timing simulation of static designs. Furthermore, since the user logic would be represented by configuration bits instead of RTL signals, the designer would not be able to focus on the desired logic in the user design layer and verification productivity could thus be significantly reduced. Since vendors do not generally provide fully accurate simulation models of their FPGAs, this approach is only considered as a concept for comparative purposes.

Device Simulator. Instead of using a simulation model of the FPGA, VirtexDS [56] is a customized simulator of the FPGA device. By modeling all resources of the target FPGA, the VirtexDS simulator can simulate implemented DRS designs with full accuracy. It should be noted that VirtexDS is a special-purpose simulator, rather than a model of the FPGA fabric, which can be compiled and simulated by a general purpose HDL simulator such as ModelSim [57]. Since VirtexDS is optimized for simulating designs targeting Virtex FPGAs, the simulation is faster than general-purposed HDL simulators. Unfortunately, since the simulator is dependent on the target device, it needs to be updated for each FPGA family. Furthermore, the customized simulator cannot be integrated with mainstream verification methodologies such as coverage analysis.

* * * * * * * * * * * *

This section reviews previous work on simulation and modeling methods of DRS designs. Since traditional HDL languages cannot be used to simulate DPR, researchers have tried to model DPR using artifacts [54] [69], by modifying the simulator kernel [14], by providing libraries [67], using high-level synthesis [74], using templates [64], and building customized simulators [56]. Figure 2.7 assesses previous DRS simulation approaches in terms of accuracy and productivity. Since there is no well-established quantitative measurement of either simulation accuracy or verification productivity, Figure 2.7 only assesses previous approaches from a qualitative perspective.

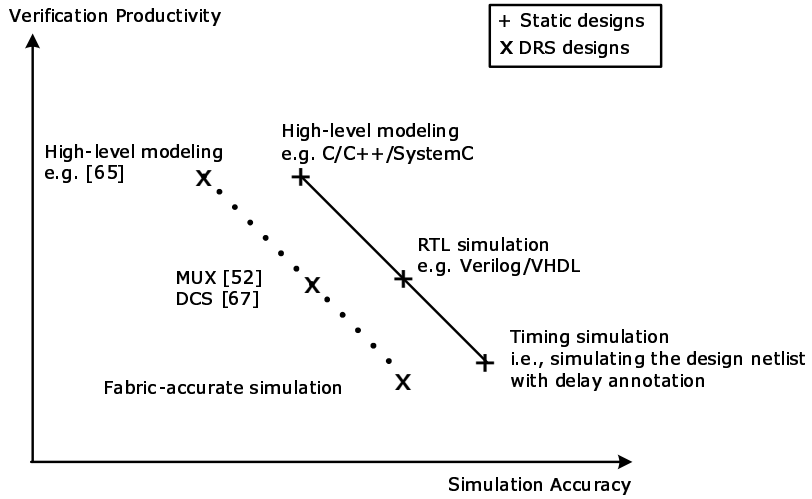


Figure 2.7: Tradeoff between simulation accuracy and verification productivity (for static and DRS designs)

Fabric-accurate simulation is the cycle-accurate simulation of the FPGA device and it can therefore achieve comparable simulation accuracy to traditional RTL simulation. However, since it depends on the FPGA fabric, fabric-accurate simulation is the least productive approach in terms of simulation speed and debug efficiency. It can even be less productive than traditional timing simulation since timing simulation only simulates the netlist and the signal delays of the user design.

Alternatively, by sacrificing accuracy, previous work models DPR without being dependent on the FPGA fabric. For example, no previous work models bitstream traffic and therefore cannot exercise reconfiguration scenarios that need to test the bitstream transfer logic. As another example, both high-level modeling and RTL simulation assume that the reconfiguration delay is a constant. Therefore, simulated modules are swapped after such a constant delay regardless of whether or not there is a bug during reconfiguration. Furthermore, although some previous work models DPR-related activities such as state saving and restoration [69] [67], the original designs need to be changed for simulation purposes. This does not meet the implementation-ready requirement, which states that the user design should not be modified for simulation purposes. As a result, previous simulation-based approaches do not meet the Functional Verification Objective (see Section 1.1).

2.4 Formal and Run-time Verification of DRS Designs

Formal verification mathematically proves that a hardware design meets its formal specification (see Section 2.1). For DRS designs, formal verification can be used to verify the design either when it is operating normally, or when it is being reconfigured. We focus on applying formal methods to verify the reconfiguration process of DRS designs.

With dynamic reconfigurability, a DRS can be designed to be open-ended, which means that hardware RMs are not known at system design time and the system is open to various applications that may be mapped to the system after it has been deployed. For example, DRS computing platforms such as the Erlangen Slot Machine [13], the VAPRES streaming architecture [41], the Sonic video processing system [76] and the MPI-based generic parallel computing accelerator [71] provide basic infrastructure so that future applications can easily be mapped to the platform. Therefore, apart from verifying partial reconfiguration, the designer also needs to verify that newly introduced RMs can be correctly integrated with the existing system. We refer to this type of verification as *run-time verification*.

This section reviews previous formal and run-time verification methods. We cover both methods in one section because previous work often combines them. In particular, by definition, run-time verification verifies a DRS design on its target FPGA under real-world conditions instead of in a simulated environment. Furthermore, since formal verification does not require creating a simulation testbench, it is more suited to being applied at run time compared with simulation. However, formal verification and run-time verification can also be used independently of each other.

Verifying Reconfigurable Cores. Formal methods have been used to verify reconfigurable cores at run-time [81]. The authors propose to run a light-weight, proof system which converts both the reconfigured core and its specification into propositional logic and verifies that the two versions are equivalent. The core to be verified is optimized for the target FPGA while the specification is a general, un-optimized RTL description of the same core.

This approach is limited by the overheads of running the proof system at run time. The original work targeted verifying reconfigurable cores on the XC6200 FPGA. Since only a very small portion of user circuits were modified by partial reconfiguration, the overhead of running the formal prover was acceptable. For modular reconfigurable designs targeting more recent FPGAs, the overhead of proving a complex RM at run time is too large to be practical.

Proof Carrying Hardware. Aiming to reduce the overhead of running the proof system, the proof-carrying-hardware approach [24] proposes to offload the formal proof task to the designer of the RMs. The idea is to record the steps of the formal verification process offline and combine the steps as part of the configuration file of the RM. At run time, the DRS simply checks the proof, which passes if and only if the RM is

configured correctly. The idea was extended from the proof carrying code techniques proven effective in validating the trustworthiness of downloaded software.

This approach saves the overhead of formal verification at run time. Although the idea of the paper is interesting, the approach is described as a proof of concept and needs to be strengthened. For example, the approach only supports combinational equivalence checking, while the majority of hardware designs are sequential.

Pi-calculus. Formal verification can also be used at design time in addition to at run time. Pi-calculus has been used in the modeling, analysis and verification of DRS designs [77, 78]. The Pi-calculus is a process algebra originally designed for modeling communicating systems. By restriction to a subset of the calculus, the authors propose a tiny pi-calculus to formally model the "reconfiguration rules/directives" (i.e., when to reconfigure what) of a DRS design [77]. Pi-calculus is able to model the start and end of an RM's execution, and the communication requirements between multiple RRs [78]. To verify the design, the designers prove that the formal model described by pi-calculus has or does not have a set of required properties.

As described in Section 2.1, the formal specification may only capture a subset of desired design behavior. In terms of the pi-calculus method, it assists in verifying the reconfiguration rules that are explicitly modeled. However, the reconfiguration rules do not support modeling a bitstream transfer by a user-defined reconfiguration controller. Another limitation is that the authors assume that RMs are only deactivated after they have finished (i.e., self-deletion), which is not true for preemptive run-time management systems (e.g., [70]).

Verifying Streaming Hardware. Formal verification has been applied to verify DRS designs targeting streaming applications [85]. The idea is to use symbolic simulation techniques to verify the equivalence of an executable software specification of a design and an optimized, partitioned, software/hardware system. In particular, the proposed method compiles the specification and the optimized target design into an intermediate representation for symbolic simulation. During simulation, symbolic inputs are driven to both models and the responses are checked. The authors model partial reconfiguration using Virtual-Multiplexing [54], which selects one active RM at a time. Furthermore, the authors introduce a new language construct, the `reconfigure_if` conditional statement, to the MaxJ language so as to explicitly capture the virtual multiplexer in the formal specification.

Since the reconfiguration process is modeled by Virtual-Multiplexing, the formal specification only captures the module swapping operation of DRS designs. It fails to capture other aspects of DPR (e.g., bitstream traffic). Furthermore, this approach is limited by the assumed architecture of the system, in which software invokes hardware modules (static or run-time reconfigurable) via APIs and the software is blocked until the end of hardware execution. Such a model restricts the style of the DRS design that can be verified using the method.

On-chip Debugging. As described in Section 2.1, on-chip debugging using vendor tools such as ChipScope [95] is a time-consuming step for FPGA-based designs. Unfortunately, due to the limitation of the clocking circuitry, ChipScope [95] can only be used to monitor signals from the static region of a DRS design [83]. Xilinx has proposed a workaround to resolve the problem [1], but the proposed solution requires expert knowledge of the ChipScope tool, thereby introducing additional complications to the on-chip debugging of DRS designs.

FPGA-in-the-Loop Simulation. The FPGA-in-the-Loop approach proposes to test a DRS design on the target device while driving the stimuli from simulators such as ModelSim [57] or Simulink [55]. To assist in debugging the reconfiguration process, this approach visualizes the placement of RMs on the FPGA and updates the visualized placement information upon reconfiguration [61].

Visualizing module placement could assist in debugging designs that allow module relocation. In particular, a bug in placing/relocating the module could easily be identified by visualizing the placement of RMs on the target device. However, for modular reconfigurable DRS designs, each RM can only be assigned to a single, fixed location. Therefore, visualizing RM placement offers limited assistance in debugging the reconfiguration process. Furthermore, since the visualization technique relies on parsing configuration bitstreams in order to extract placement information, the infrastructure to parse bitstreams needs to be modified for each FPGA family.

Self-checking Circuitry. A car cabin controller uses self-checking circuitry to test the system functionality at run time [63]. In order to save system resources, the testbench modules can be dynamically allocated to reconfigurable slots that are not occupied by normal RMs. Alternatively, reconfigurable slots can be configured with three identical copies of an RM so as to detect malfunctioning transistors in the FPGA device.

Limited by the capabilities of the checker and the checking method, self-checking circuitry can not thoroughly test all functionality of the target system at run time. For example, since the system in [63] uses DPR to allocate testbench modules and triplicated RMs, the reconfiguration machinery itself can not thereby be tested.

* * * * * * * * * * * *

This section reviews previous work on formal and run-time verification of DRS designs. Formal verification has been used to verify that RMs meet their specification [81] [24] and to verify that the high-level operations of a DRS design are correct [77] [85]. However, no existing method verifies the low-level details of a DRS design. In particular, existing methods only capture the module swapping operation in the formal specification and fail to model other DPR-related activities. For example, no existing method formally verifies that a bitstream is correctly transferred to the configuration port by the user-defined reconfiguration controller, and that the desired RM is subsequently swapped in accordingly.

DRS designs can also be validated using field testing by ChipScope [95], customized tools (e.g., [61]) or self-checking circuitry (e.g., [63]). However, existing methods for run-time verification of DRS designs are still ad-hoc.

2.5 Summary

The architectural flexibility of DRS introduces challenges in functionally verifying such systems. The challenges arise from both the user design and the physical layers that can be identified for a DRS design. For the user design layer, DPR introduces new scenarios that need to be covered in verification. The reconfiguration scenarios can be categorized according to the phase of the partial reconfiguration process during which they occur, i.e., BEFORE, DURING, or AFTER reconfiguration. These reconfiguration scenarios need to be properly exercised during simulation in order to verify the correct operation of the DRS during reconfiguration. DPR involves interactions between the user design layer and the physical layer (i.e., inter-layer interactions), and it is challenging to accurately model such inter-layer interactions without relying on the details of the FPGA fabric. Furthermore, the modeling of the interactions should not change the user design lest a variation of the user design is simulated.

Previous simulation-based approaches, either using high-level modeling or RTL simulation, abstract away the details of the FPGA fabric but sacrifice simulation accuracy. In theory, fabric-accurate simulation can achieve full accuracy but suffers from low verification productivity. In reality, the simulation model of the FPGA fabric is not available. It should be noted that DRS designs can also be verified using formal methods and field testing. Previous formal methods only successfully capture the very high-level behavior of DRS designs. Run-time verification approaches can visualize partial reconfiguration activities, but they are still ad-hoc.

From a methodology perspective, it is desirable to provide DRS designers with guidelines and examples of exercising reconfiguration scenarios and identifying DPR-related bugs. However, the literature only focuses on presenting simulation/verification tools without reporting on the verification effort or bugs identified during the development of DRS designs.

We therefore conclude that there is a significant gap between the ability to design a DRS and the ability to verify a DRS.

Chapter 3

Modeling Dynamic Partial Reconfiguration

As discussed in Section 2.2.3, the fundamental challenge for functional verification of DRS designs is the conflicting requirements of modeling accuracy and verification productivity. For simulation-based verification, the designer needs to strike the right balance between the level of simulation accuracy and the level of detail about the fabric being simulated. Furthermore, this balance is constrained by the desire for the simulated design to be implementation ready. Thus in this chapter, we derive a general approach to modeling DPR and show that the proposed approach meets the above-mentioned requirements. Following the Mainstream Objective of this thesis, we focus on simulation-based functional verification, and since RTL simulation is the most common method for verifying hardware design functionality (see Section 2.1), we describe the modeling methodology at the RTL level.

The core idea of our approach is to use a simulation-only layer to *emulate* the physical fabric of FPGAs so as to achieve the desired balance between accuracy and physical independence. Since the simulation-only layer does not change the user design, the DUT is not modified during verification and is therefore implementation-ready. Section 3.1 illustrates the details of the physical layer, from which we extract 6 characteristic features of DPR (see Section 3.1.2) which capture the inter-layer interactions. Section 3.2 describes the simulation-only layer. By abstracting away the details of the FPGA fabric, the simulation-only layer shields the simulated design from the physical details of the fabric (Section 3.2.1). By modeling the characteristic features of DPR, the simulation-only layer is able to accurately mimic the impact of the FPGA fabric on the user design (Section 3.2.2). Section 3.3 analyzes the capabilities and limitations of using the simulation-only layer and the last section (Section 3.4) summarizes this chapter.

3.1 The Physical Layer

As illustrated in Section 2.2, a typical DRS design has two conceptual layers (see Figure 3.1, which is the same as Figure 2.5) and the two layers interact with each other during

reconfiguration. Therefore, in order to accurately simulate the inter-layer interactions, it is essential to model the physical layer and to capture the inter-layer interactions during simulation.

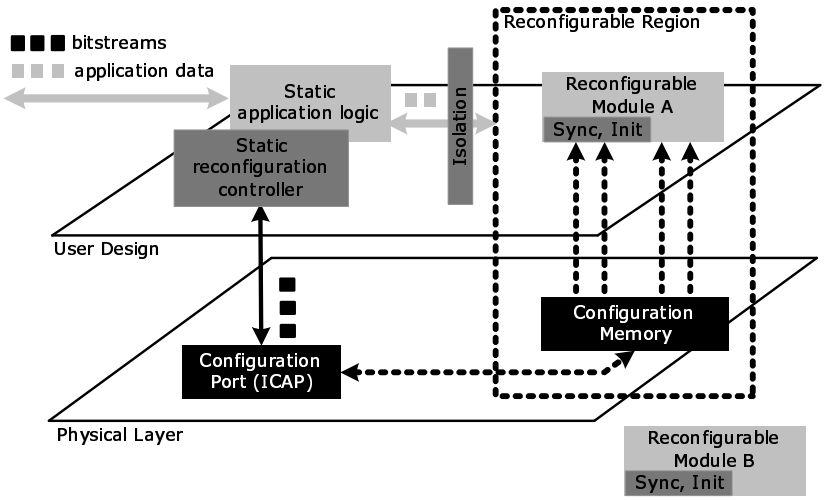


Figure 3.1: Conceptual diagram of a DRS design (Same as Figure 2.5)

3.1.1 The Configuration Mechanisms

As Xilinx is the only vendor to currently support partial reconfiguration in their design tools and their FPGA devices[1], we use Xilinx Virtex series of FPGA families as examples to describe the physical layer of DRS designs.

Configuration Port. Partial reconfiguration is performed by transferring a bitstream to the configuration port of the FPGA. The configuration port is the interface between the user design layer and the physical layer. It can be internal, such as the Internal Configuration Access Port (ICAP), which is typically instantiated as part of a user design as if it were a sub-module of the user design hierarchy. It can also be an external port such as SelectMap, which is driven by an external reconfiguration controller while the DRS design is running. The user design writes bitstreams to and reads bitstreams from the configuration port according to the interfacing protocol specified by the FPGA document (e.g., [94, 100, 101, 104]).

Configuration Memory. The configuration memory is organized into *frames*, and each configuration frame is indexed by a Frame Address (FA) including the Row Address (RA), Column Address (CA) and Minor Address (MNA). A frame is the smallest addressable storage unit in the configuration memory, a frame contains a number of words and each configuration bit is indexed by the Bit Offset (BO) within the frame. For Virtex-5 FPGAs (see Figure 3.2), as an example, the basic logic block illustrated in Figure 2.6 is known as a Configurable Logic Block (CLB). The intersection of a given

[1]Although the Stratix V device supports partial reconfiguration, Altera is yet to incorporate partial reconfiguration into its tool flow

RA and CA contains an array of 20 * 1 CLBs, and their configuration settings are stored in 36 contiguous frames, each of which contains 41 32-bit words. Thus, in order to locate the configuration bit of a particular configuration setting, the designer needs to know the RA, CA, MNA and BO of that configuration bit.

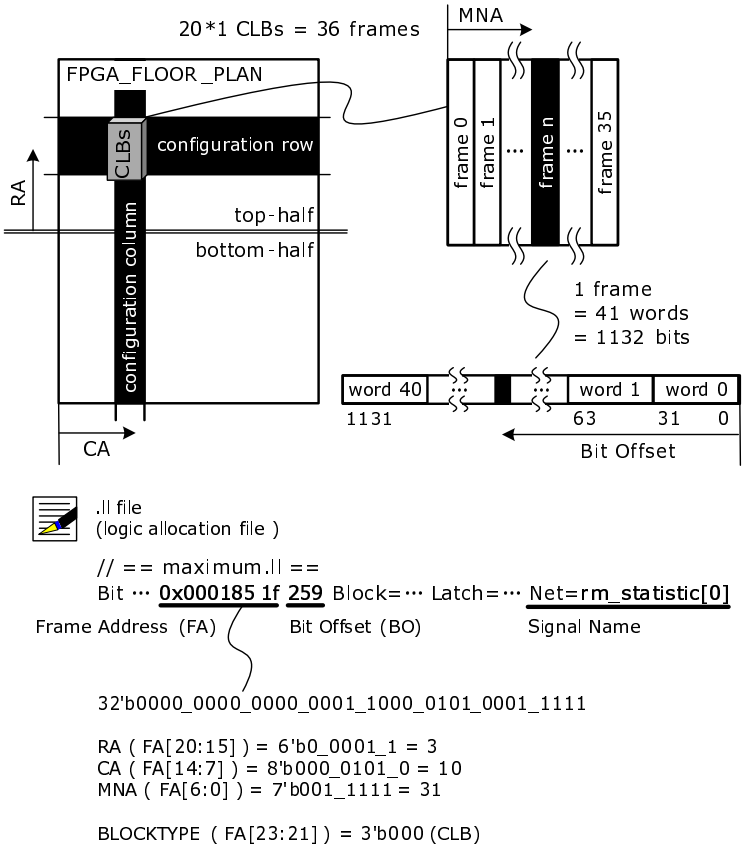


Figure 3.2: Configuration memory of the Virtex-5 FPGA family, after [100]

Figure 3.2 also shows an example of an entry in the `.ll` file generated by the vendor tool. A `.ll` file indicates the allocation (i.e., RA, CA, MNA, BO) of configuration bits in the configuration memory. Although it does not contain allocation information for all bits, it does include all configuration bits that control flip-flop register values (i.e., P3 in Figure 2.6) and memory values (e.g., Block RAM or BRAM). In particular, the `rm_statistic[0]` flip-flop is allocated to Frame Address (FA) 0x0001851f, from which the designer can decompose the RA, CA and MNA (see the bit composition of the frame address in the figure). The example also indicates that the flip-flop is mapped to the 259th bit of the frame (indexing from 0). A `.ll` file also indicates the allocation of memory cells such as BRAM and their corresponding frame address, although different key words are used for BRAM entries (see the configuration guides, e.g., [94, 100, 101, 104]).

Configuration Bitstream. Configuration bitstreams are data items of partial reconfiguration and are also known as partial bitstreams for DRS designs. They are transferred by the user design in the same way as normal data while they contain configuration settings of the physical layer that can only be interpreted by the FPGA device. Table 3.1

lists an example bitstream[2] that indicates a write operation (see entries 3-5 in the table) to the configuration memory. By overwriting the configuration frames to which the old RM is mapped, the configuration settings are updated and the new RM is swapped in. It should be noted that one extra pad frame is typically required by the FPGA. A bitstream starts with a `SYNC` word (entry 1 in the table) and ends with a `DESYNC` command (entry 7 in the table).

Table 3.1: An example bitstream for partial reconfiguration

Entry	Bitstream	Explanation	Actions Taken
1	`0xAA995566`	SYNC Word	Start of bitstream
2	`0x20000000`	NOP	–
3	`0x30002001` `0x0001851f`	Type 1 Write FAR FA = `0x0001851f`	Write configuration data starting from frame with FA=`0x0001851f` (RA = 3; CA = 10; MNA = 31)
4	`0x30008001` `0x00000001`	Type 1 Write CMD WCFG	
5	`0x30004000` `0x500005ED`	Type 2 Write FDRI Size=1517	Pad frame: 41 words Configuration data: 1476 words (36 frames)
6	...	Configuration data & pad data Words 0 - 1516	Configuration data contain the logic settings of the new RM
7	`0x30008001` `0x0000000D`	Type 1 Write CMD DESYNC	End of bitstream

Apart from configuring module logic, some DRS designs also save and restore module state (i.e., flip-flop and memory values) via the configuration port (CP-based SSR). To save the state data of user flip-flops, the user design needs to read back configuration data through the configuration port. Using the logic allocation information specified in the `.ll` file, the user design then filters out configuration bits (i.e., the `INT0/INT1` bits) that represent storage elements of user flip-flops (e.g., P3 of Figure 2.6). The readback and the filtering operations of the user design can be performed by a reconfiguration controller that is either on-chip or off-chip. Table 3.2 lists an example bitstream that reads the frame containing the `rm_statistic[0]` flip-flop mentioned in Figure 3.2[3]. After the default `SYNC` word (`0xAA995566`), the bitstream contains a `GCAPTURE` command to copy flip-flop values to the configuration memory. This is required because flip-flops are not synchronized with their corresponding `INT0/INT1` bits in the configuration memory until such a command is issued. The following 6 words (see entries 3-5 in Table 3.2) indicate the configuration frame to be read. Then the configuration port is switched to read mode, during which the 83 words of configuration data are returned to the user design. The example reads more than a frame because an additional pad frame and a pad word are returned before the actual configuration data are returned. The bitstream ends with a `DESYNC` command.

The restoration process reverses the state saving procedure. The saved or modified state data is merged into the configuration data at the position specified by the `.ll` file, and

[2]To focus on partial reconfiguration, this example doesn't include irrelevant sections.
[3]To focus on state saving, this example doesn't include irrelevant sections.

Table 3.2: An example bitstream for state saving

Entry	Bitstream	Explanation	Actions Taken
1	0xAA995566	SYNC Word	Start of bitstream
2	0x30008001 0x0000000C	Type 1 Write CMD GCAPTURE	Copy the state bits from the flip-flops to the configuration memory
3	0x30002001 0x0001851f	Type 1 Write FAR FA = 0x0001851f	Read configuration data starting from frame with FA=0x0001851f (RA = 3; CA = 10; MNA = 31)
4	0x30008001 0x00000004	Type 1 Write CMD RCFG	
5	0x28006000 0x50000053	Type 2 Read FDRO Size=83	Pad frame: 41 words Pad word: 1 word Configuration data: 41 words
6	...	Configuration data & pad data Words 0 - 82	Configuration data are returned to the user design
7	0x30008001 0x0000000D	Type 1 Write CMD DESYNC	End of bitstream

a GRESTORE command is used to copy state data to the flip-flops. However, as GRESTORE performs a chip-wide restoration, a mask must be set up before using the GRESTORE so that only the frames of interest are restored. Setting up the mask is performed by writing a mask bitstream to the configuration port. Apart from the GRESTORE-based flow, flip-flop values can als be restored using a a reset-based approach [80, 44]. The general procedure for a reset-based approach is similar to the GRESTORE-based approach except that the state data are merged with the SET/RESET bits of the configuration bitstream instead of the INIT0/INIT1 bits. The values of the SET/RESET bits are copied to their corresponding flip-flops when the reconfigured module is reset. Saving and restoring BRAM cells is similar to saving and restoring flip-flops. However, since BRAM data of the user design are stored directly in the configuration memory, it is not necessary to issue the GCAPTURE/GRESTORE commands. Please refer to the configuration guides for further details about mask bitstreams, reset-based restoration and saving and restoring BRAM cells [94, 100, 101, 104].

3.1.2 The Characteristic Features of Reconfiguration

Although the details of FPGA families vary, we extract the characteristic features of DPR that are common to ALL FPGA families. Since we focus on the functional verification of the user design layer of a DRS design, we only extract features that are essential to exercise the reconfiguration scenarios described in Section 2.2.1, and to test the reconfiguration machinery of a DRS design (see Figure 2.4 of Section 2.2.1).

- **Module Swapping:** From the user design's perspective, DPR can be considered as swapping modules at run time. Furthermore, since module swapping could be

delayed by, for example, traffic contention on a shared datapath, the reconfiguration delay is dependent on the size of the bitstream as well as run-time circumstances. Therefore, switching modules instantaneously [54], or after a compile-time defined delay [69], ignores the real timing of module swapping.

- **Triggering Condition:** On FPGAs, module swapping is triggered if and only if *all* bytes of the bitstream are *successfully* written to the configuration port. Without correctly modeling the tiggering condition, the simulation module swapping would be independent of bitstream transfer and a simulated module could be swapped even if the bitstream transfer were to fail. Simulating the triggering condition using any other mechanism (e.g., the *reconfiguration condition detector* component of [69]) therefore doesn't correctly emulate what actually occurs.

- **Bitstream Traffic:** The process of transferring bitstreams, the interaction between bitstream transfers and other activities of the system cannot be verified without simulating the bitstream traffic, which isn't available until after the time-consuming implementation step. A bug on the bitstream transfer datapath (e.g., FIFO overflow/underflow) may not therefore be detected in simulation.

- **Bitstream Content:** The configuration data of bitstreams contain bit-level logic settings, data for state elements (i.e., flip-flops, memory cells) of the target RR, and placement information of the module. Since such configuration data only affect resource blocks in real FPGAs, traditional simulation methods cannot make use of the contents of bitstreams and therefore cannot simulate the impact of the configuration data on the user design. In other words, traditional simulation cannot correlate the contents of the bitstreams with the logic or state of modules that are being simulated.

- **Spurious Outputs:** On FPGAs, a module undergoing reconfiguration may produce spurious outputs to the static region. However, the simulated modules won't produce spurious outputs when being reconfigured and the isolation logic of the DRS design cannot therefore be tested in simulation. Moreover, modeling such spurious outputs as a constant such as the undefined "x" value in [69] lacks the flexibility required to thoroughly test the isolation logic.

- **Undefined Initial State:** The initial states of RMs are undefined AFTER reconfiguration. However in simulation, if, for example, an RM is swapped out and then swapped in again later, the internal signals of the RM are preserved by the simulator as if the module had never been swapped out. The simulated RM would continue to operate even if there is a bug in the initialization process of the user design. The initialization logic of the DRS design cannot therefore be properly tested.

3.2 The Simulation-only Layer

In order to achieve the desired balance between simulation accuracy and physical independence, we propose to substitute the FPGA fabric with a simulation-only layer

in simulation-based functional verification of DRS designs. Figure 3.3 redraws Figure 3.1 with all the physically dependent blocks (darkly shaded boxes) replaced by their corresponding simulation-only artifacts (open boxes). In particular, *simulation-only bitstreams* (SimB, see Section 3.2.1) are used to replace configuration bitstreams so as to model the bitstream traffic, the media of the inter-layer interactions of DRS designs. The part of configuration memory to which each RR is mapped is substituted by an Extended Portal, which models the characteristic features of DPR (see Section 3.2.2). Without loss of generality, possible configuration ports, either internal or external to the design, are represented by an ICAP artifact.

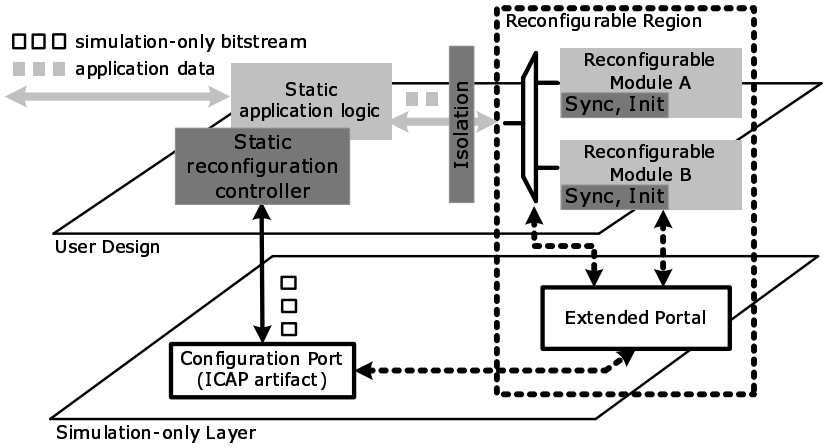


Figure 3.3: Using the simulation-only layer

Using a simulation-only layer, reconfiguration is simulated as follows: the user-defined reconfiguration controller transfers a SimB instead of a real bitstream to the ICAP artifact. While the SimB is being written, the Extended Portal injects errors to both the static region and the RM. Errors injected to the static region model the spurious outputs of the module undergoing reconfiguration and help to test the isolation logic of the user design. Errors injected to the reconfigurable modules model the undefined initial state of the RM and help to test the initialization mechanism of the design. After the SimB has been *completely* written to the ICAP artifact, the Extended Portal checks the signature of the SimB, extracts the numerical ID of the new RM, and drives module swapping according to the extracted ID.

The more complex processes of sate saving and restoration, as can be achieved by reading, writing and processing configuration bitstreams in real FPGAs, can also be emulated using the simulation-only layer. For example, saving the state of a RM is simulated as follows: the static module transfers a SimB instead of a real bitstream to the ICAP artifact. By parsing the SimB, the ICAP artifact extracts the readback parameters, and controls the Extended Portal. The Extended Portal probes the RM state (i.e., values of RTL signals that represent state elements such as flip-flops and memory cells), and returns the state data to the ICAP artifact. Finally, the ICAP artifact returns the retrieved state data to the user design as a readback SimB. Restoring the state of a RM also utilizes the artifacts and SimBs to mimic the behavior of the FPGA fabric. However, a different SimB from the one for state saving is used, and the state data are copied back to the RTL signals of the simulated RM.

The simulation-only layer approach is, to some extent, analogous to the Bus Functional Model (BFM) simulation, in which a BFM is used to emulate the behavior of a microprocessor so as to test and verify peripheral logic attached to a microprocessor bus [96]. Although the BFM model is not a completely accurate representation of the microprocessor, it is accurate enough to capture the interactions between the microprocessor and the bus peripheral to be tested. Furthermore, the verification productivity achieved using BFM is significantly higher than that achieved simulating the netlist of a processor since the BFM approach abstracts away the internal behaviors of a processor such as pipelines. Similarly, the simulation-only layer emulates the FPGA device and captures the inter-layer interactions required to exercise reconfiguration scenarios. Furthermore, the simulation-only layer abstracts away the details of the FPGA fabric and significantly improves the verification productivity compared with fabric-accurate simulation.

The concept of a simulation-only layer is also, to some extent, analogous to a virtual FPGA [51]. The idea of a virtual FPGA was proposed as a means of creating a device-independent FPGA fabric any design could be mapped to. By efficiently mapping the virtual FPGA to each real FPGA type, the user design could be ported between devices without remapping. Although the original idea suffered from significant performance overheads, the concept of a fabric-independent FPGA device is here adopted to facilitate simulation-based functional verification. In particular, we propose performing functional verification of DRS designs using a generic FPGA model, namely the simulation-only layer, which is independent of the target FPGA device. Referring to the Functional Verification Objective of this thesis, we aim to assist designers in detecting functional bugs. Since functional bugs are defined as bugs introduced to the design specification and the captured design (e.g., RTL design, see Section 2.1), they should be exposed regardless of the target FPGA the design is mapped to. The design should be tested for implementation-dependent bugs on the target device. We will describe the capabilities and limitations of this approach in Section 3.3.

3.2.1 Modeling the Configuration Mechanisms

The simulation-only layer models the behavior of the FPGA fabric at RTL level, or at a higher level, and hides the details of the physical layer. In particular, we define the organization of the configuration memory and the configuration bitstream, and migrate concepts of the physical layer that cannot be effectively used during simulation to equivalent or similar concepts that can be used during simulation.

Configuration Port. The simulated configuration port is an artifact that interfaces the user design with the simulation-only layer. For DRS designs that perform internal reconfiguration, the simulated configuration port is instantiated as part of the user design. In simulation, it interacts with the user-defined reconfiguration controller according to the interfacing protocol specified by the device configuration guide (e.g., [94, 100, 101, 104]). However, the simulated configuration port takes a simulation-only bitstream instead of a real bitstream as input, and extracts simulation-only information (e.g., numerical IDs of modules) that can be processed by the rest of the simulation-only

layer. To simulate configuration readback, the simulated configuration port returns a readback simulation-only bitstream to the user design.

It should be noted that Figure 3.3 uses an ICAP artifact to represent possible configuration ports. To simulate external reconfiguration, the simulated configuration port is instantiated as part of a simulation testbench and parses SimBs during simulation. To simulate DRS designs mapped to FPGAs from other vendors, the general concept of using a simulated configuration port still applies.

Configuration Memory. We derive the organization of the configuration memory of the simulation-only layer (see Figure 3.4) from the organization of the configuration memory of real FPGAs (see Figure 3.2). Instead of being organized into rows and columns like the FPGA fabric, the configuration memory of the simulation-only layer is organized according to the affiliation of RMs to RRs. In the simulation-only layer, each RR and RM is given a numerical ID (RRID & RMID). The example shown in Figure 3.4 has 3 RRs with RRIDs = {0,1,2} and RR0 has 2 RMs with RMID = {0,1}. Note that the RMIDs commence at 0 for each RR. Each RR is composed of N frames where N is a user-defined parameter specified by a parameter script (see Section 4.3). Within one RR, a configuration frame is indexed by a minor address (MNA) and each frame contains 4 words.

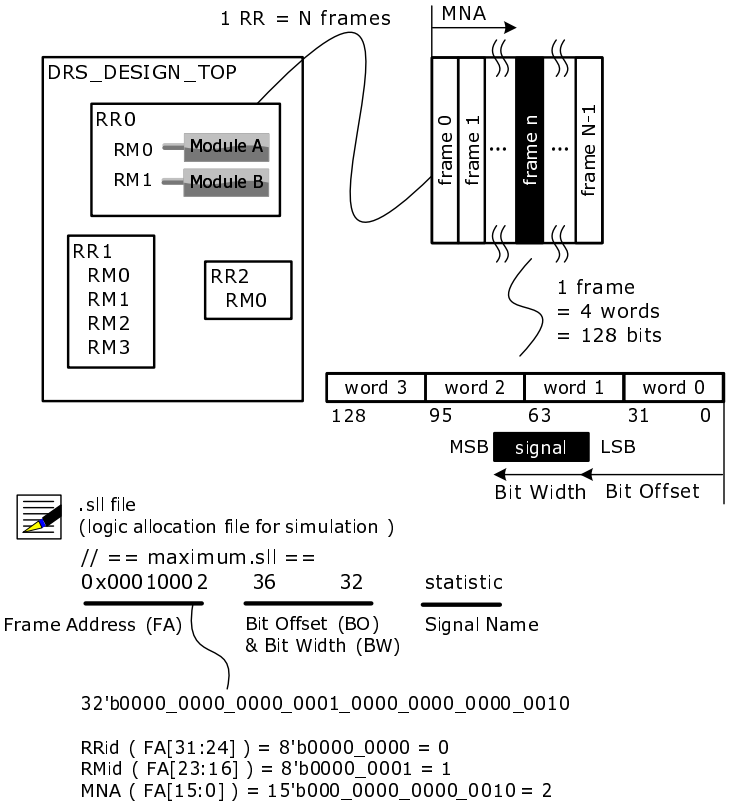


Figure 3.4: Configuration memory of the simulation-only layer

Instead of containing configuration settings that define the logic and routing of the user design, the configuration frames of the simulation-only layer, i.e., the simulation-only frames, store information that can be used in simulation. In particular, the WORD 0 of

all frames of an RR form a sequence of 32-bit signatures assigned by the simulation-only layer. The signature data can identify a unique RM in the user design and are expected to be kept unchanged throughout a simulation run. WORDs 1-3 of a configuration frame are used to store the value of RTL signals that represent the state elements (i.e., flip-flops and memory cells) of the simulated module. If the simulation exercises a `GCAPTURE/GRESTORE` operation, the values stored in the simulated configuration memory are synchronized with the RTL signals of the simulated module. Such a synchronization operation mimics the `GCAPTURE/GRESTORE` of real FPGAs (see Section 3.1.1) and can be used to simulate state saving and restoration.

To map RTL signals to the configuration memory, the simulation-only layer treats the simulation-only logic allocation file (`.sll` file) as being equivalent to the logic allocation file (`.ll` file) generated by the FPGA vendor tools. On real FPGAs, the state of a multi-bit register is flattened into individual flip-flops that are mapped to the configuration memory bit-by-bit (see Section 3.1.1). However in the simulation-only layer, a multi-bit register (typically represented by RTL signals in the HDL code) is stored contiguously. Since signals are typically grouped in the RTL user design, storing the state bits contiguously improves debugging productivity. Thus, in order to locate one user signal in the configuration memory, the designer needs to know the Bit Width (BW) of the signal, in addition to the RRID, RMID, MNA and Bit Offset (BO). Figure 3.4 provides an example of the `.sll` file. The `statistic` register is located at Frame Address `0x00010002`, which is decomposed into `RRID` = 0, `RMID` = 1 and `MNA` = 2. The BO and BW fields indicate that the `statistic` register starts at bit 36 and is 32 bits wide. Similarly, RTL signals that correspond to memory cells (e.g., 2-D arrays in Verilog/VHDL) can also be mapped to the configuration memory of the simulation-only layer using the `.sll` file.

By addressing RMs/RRs using numerical IDs, constructing configuration frames with signatures and state data, and mapping RTL signals using `.sll` files, our approach models the configuration memory of the simulation-only layer using information extracted from the RTL design and yet is physically independent. Note that it does not matter how many words a simulation-only frame has, nor how many bits the module signature or module state data should contain. This thesis defines 4 words per frame, 32 bits per signature, and 3*32 bits per state value for our simulation-only layer. Although we believe it is unnecessary to do so, a designer can choose a different set of parameters to define a different configuration memory layout for the simulation-only layer.

Simulation-only Bitstream. As simulation cannot effectively make use of a real bitstream (see the "Bitstream Content" item in Section 3.1.2), the simulation-only layer substitutes a *simulation-only bitstream* (SimB) to model the bitstream traffic. The SimB captures the essence of a real bitstream but its size is significantly reduced. Table 3.3 provides an example of a SimB that configures a new module. Similar to a real bitstream, a SimB starts with a `SYNC` word (entry 1 in Table 3.3) and ends with a `DESYNC` command (entry 7 in Table 3.3). However, a SimB differs from a real bitstream in the Frame Address and the configuration data field.

Instead of containing the Frame Addresses of the configuration data to be written, as found in a real bitstream, a SimB contains the RRID and RMID of the module to be

Table 3.3: An example SimB for configuring a new module

Entry	SimB	Explanation	Actions Taken
1	`0xAA995566`	SYNC Word	Start the **DURING Reconfiguration** phase
2	`0x20000000`	NOP	–
3	`0x30002001` `0x01020000`	Type 1 Write FAR FA=`0x01020000`	Select the module id=0x02 to be the next active module in reconfigurable region id=0x01
4	`0x30008001` `0x00000001`	Type 1 Write CMD WCFG	
5	`0x30004000` `0x50000010`	Type 2 Write FDRI Size=16	Pad frame: not required Configuration data: 16 words (4 frames)

6	`0x5650EEA7`
	`0xF4649889`
	...
	`0xA9B759F9`
	`0x4E438C83`
		SimB Frame 0 Word 0
		SimB Frame 0 Word 1
		...
		SimB Frame 3 Word 2
		SimB Frame 3 Word 3
			Module signature; Module state data; Enable/Disable error injection

7	`0x30008001` `0x0000000D`	Type 1 Write CMD DESYNC	End the **DURING Reconfiguration** phase

reconfigured. In the example SimB, 6 consecutive words (entries 3-5 in Table 3.3) request that the current module in the RR with ID = 0x1 be replaced by the new module with ID = 0x2. The example SimB also indicates that the target RR contains 4 simulation-only layer frames, as explained next.

Instead of containing bit-level configuration settings for the module to be configured, as found in a real bitstream, the configuration data section of a SimB contains fields that are to be written to the configuration memory of the simulation-only layer (entry 6 in Table 3.3). In particular, WORD 0 is the module signature field and WORDs 1-3 store the values of state elements (i.e., flip-flops and memory cells) of the RM to be configured. The configuration data section of a SimB also indicates the start and end of error injection.

SimBs can also be used to simulate state saving and restoration. Table 3.4 provides an example of a readback SimB that saves the state of the `statistic` register of Figure 3.4. Similar to the previous example, the readback SimB also starts with a `SYNC` word and ends with a `DESYNC` command (entries 1 and 7 in Table 3.4). In addition, the Frame Address of a real bitstream is replaced by the RRID and RMID of the target RM.

However, there are a couple of differences between the readback SimB in Table 3.4 and the partial SimB of Table 3.3. Firstly, the readback SimB has an extra `GCAPTURE` command to copy the values of the RTL signal (i.e., the value of the `statistic` register) to the configuration memory of the simulation-only layer. Secondly, since the `statistic` register is mapped to the 2nd frame of the target RR (i.e., MNA=2, see the `.sll` file entry of Figure 3.4), the example readback SimB only reads the 2nd frame instead of all 4 frames of the target RR. Thirdly, during state saving, the contents of the configuration data section of the readback SimB (the 4 words of entry 6 of Table 3.4) are *returned*

Table 3.4: An example SimB for state saving

Entry	SimB	Explanation	Actions Taken
1	`0xAA995566`	SYNC Word	Start the **DURING Reconfiguration** phase
2	`0x30008001` `0x0000000C`	Type 1 Write CMD GCAPTURE	Copy the state bits from HDL signals to the spy memory
3	`0x30002001` `0x00010002`	Type 1 Write FAR FA = `0x00010002`	Read configuration data starting from frame with FA=`0x00010002` (RRid=0; RMid=1; MNA=2)
4	`0x30008001` `0x00000004`	Type 1 Write CMD RCFG	
5	`0x28006000` `0x50000004`	Type 2 Read FDRO Size=4	Pad frame: not required Configuration data: 4 words (1 frame)

6	`0xE31DAA1B`
	`0x00D00030`
	`0x0000000F`
	`0x00000000`
		SimB Frame 2 Word 0
		SimB Frame 2 Word 1
		SimB Frame 2 Word 2
		SimB Frame 2 Word 3
			Configuration data are returned to the user design

7	`0x30008001` `0x0000000D`	Type 1 Write CMD DESYNC	End the **DURING Reconfiguration** phase

from instead of *being written to* the simulated configuration port. In particular, after entries 1-5 are written, the ICAP artifact is switched to read mode and responds with the 4 words of configuration data. WORDs 1-3 of the configuration data section contain state data of the `statistic` register. After reading back the 4 words of entry 6, the application logic of the user design can extract the value of the `statistic` register. At the end of the readback, the ICAP artifact is switched back to write mode when the `DESYNC` command is issued. We use this readback SimB as an example in a case study in Section 5.1.1.

The restoration process reverses the state saving procedure. To restore the value of a register, the user design merges the previously saved state data with the SimB at the position specified by the `.sll` file. During simulation, a `GRESTORE` SimB (for `GRESTORE`-based restoration method) or a module reset (for reset-based restoration method) copies the state data to the simulated module. Since the `.sll` file also maps RTL signals that correspond to memory cells to the simulation-only layer, saving and restoring memory cells can be performed in a way similar to that of registers, except that the SimBs does not include the `GCAPTURE`/`GRESTORE` commands.

3.2.2 Modeling Characteristic Features of Reconfiguration

In order to accurately simulate the interaction between the physical layer and the user design, the simulation-only layer uses an Extended Portal to model the characteristic features of DPR (i.e., module swapping, bitstream traffic, bitstream content, triggering condition, spurious outputs and undefined initial RM state, see Section 3.1.2). The term portal is adopted from ReChannel, which views a portal to be a special switch

that interfaces the static region with a group of RMs [67]. In our work, the Extended Portal of the the simulation-only layer extends the ReChannel portal concept to model characteristic features of DPR as described in Section 3.1.2:

- **Module Swapping and Triggering Condition:** The Extended Portal connects all RMs in parallel and, like most existing approaches (e.g., [54]), selects the active RM via virtual multiplexers. However, in our simulation-only layer approach, the selection of the active RM is triggered by the SimB and any run-time dependent events that delay the bitstream transfer also delay the module swapping. Therefore, the delay of module swapping is more accurately modeled instead of being approximated by a constant value. Furthermore, since failing to transfer the SimB correctly prevents the new RM from being swapped in, bugs in the bitstream transfer logic can quickly be identified.

- **Bitstream Traffic:** Since we use a SimB insteal of real bitstreams, the bitstream transfer datapath (i.e., the transfer, compression, decryption and arbitration of bitstreams) is exercised in simulation. Furthermore, the length of a SimB is significantly shorter than a real bitstream and can be adjusted to exercise various scenarios of the bitstream transfer mechanism (e.g., FIFO full/empty, see the Fast PCIe cast study in Section 5.4.1).

- **Bitstream Content:** The Extended Portal parses the content of a SimB and performs simulation-only tasks accordingly. The configuration data of a SimB contains the signature of the RM to be reconfigured. The signatures are checked according to the RRID & RMID fields contained in the SimB. In particular, if any bit of the signature data or the RRID/RMID fields of a SimB are corrupted because of a bug in the design, the signature check fails and is reported during simulation. Therefore, checking the signature verifies that a correct bitstream is written to the configuration port.

  Furthermore, the configuration data of a SimB also contain data for state elements (i.e., user flip-flops, memory cells). Since the simulation-only layer correlates the stored state data with the state of the simulated module, incorrect state data, typically caused by bugs in the design, can propagate to RMs of the simulated user design after state restoration, and can therefore be quickly identified in simulation.

- **Spurious Outputs and Undefined Initial State:** The Extended Portal has two error injection mechanisms. These inject error values to interfacing signals of both the static and the reconfigurable regions while the SimB is being written to the ICAP artifact. The isolation and the initialization logic can thereby be exercised and tested in simulation. In particular, failing to isolate the RR correctly can be quickly identified as a consequence of the injected errors propagating to the static region. Failing to initialize the newly configured module correctly can be detected since the injected errors are not cleared. Compared with DCS, which only drives undefined "x" values to the static region [69], our approach injects errors, for testing purposes, to both the static and the reconfigurable regions. Furthermore, in our approach, the start and end of error injection is also triggered by the SimB, which more accurately models the timing of the error injection operation.

By modeling the characteristic features of DPR, the simulation-only layer captures the interaction between the user design and the FPGA fabric (e.g., the transfer of bitstream data and the timing of reconfiguration events), and increases the modeling accuracy compared with previous MUX-based modeling approaches.

3.3 Capabilities and Limitations

We describe the capabilities and limitations of the simulation-only layer according to the DPR-related bugs it is capable of exposing and the DRS design styles it is able to simulate.

Detecting DPR-Related Bugs. As the simulation-only layer abstracts away the details of the FPGA fabric, it can be regarded as a *fabric-independent* FPGA device, and simulation can be thought of as functionally verifying the *user design layer* of a DRS on such a fabric-independent FPGA. Therefore, simulation using the simulation-only layer aims to assist designers in detecting fabric-independent bugs of a DRS design. These bugs include, but are not limited to,

- System integration bugs related to the sequencing or timing of reconfiguration events, such as, deadlock due to the unavailability of a module undergoing reconfiguration, illegal reconfiguration requests to reconfigure a module still undergoing reconfiguration (e.g., BUG-Example.XDRS.3 in Section 5.1), ...
- Software bugs in controlling partial reconfiguration, such as, setting an incorrect bitstream transfer size, passing an invalid bitstream storage buffer pointer (e.g., BUG-Example.XDRS.6 in Section 5.1.1), memory/cache coherence problem of bitstream storage buffers (e.g., BUG-Example.XDRS.2 in Section 5.1) ...
- Bugs in synchronization of the static region and RMs before reconfiguration, such as, failing to block a reconfiguration request until the RM is idle (e.g., BUG-Example.XDRS.5 in Section 5.1) ...
- Bugs in the bitstream transfer logic, such as, cycle mismatch (e.g., BUG-Example. AUTO.2 in Section 5.3), FIFO overflow, errors in driving the ICAP signals, ...
- Bugs in isolating the region undergoing reconfiguration, such as, isolating the reconfigurable region too early or too late (e.g., BUG-Example.XDRS.4 in Section 5.1), ..., and
- Bugs in initializing the newly configured module, such as, resetting the new RM while the reconfiguration is still ongoing (e.g., BUG-Example.AUTO.4 in Section 5.3), failing to feed the RM pipelines with correct data (e.g., BUG-Example.FT.2 in Section 5.2), ...

However, the simulation-only layer is not exactly the same as the FPGA fabric. Table 3.5 lists the differences between a Virtex-5 FPGA, as an example of a target device, and our

Table 3.5: Differences between the Virtex-5 FPGA fabric and the simulation-only layer

	Virtex-5 FPGA	**Simulation-only Layer**
Configuration memory	* Frame Address is composed of RA, CA and MNA * A frame has 41 words * Frame organization is not open	* Frame Address is composed of RRID, RMID and MNA * A frame has 4 words * Word 0: signature data * Words 1-3: state data
Bitstream	* Normal Frame Address * Pad words and frames * Size is determined by the resources used	* Modified Frame Address * No pad word or frame * Reduced size that is determined by design-/test-specific needs (e.g., the amount of state data)
Logic allocation	* State bits are sparsely distributed in a frame	* State bits are grouped and stored contiguously * Has a bit-width field

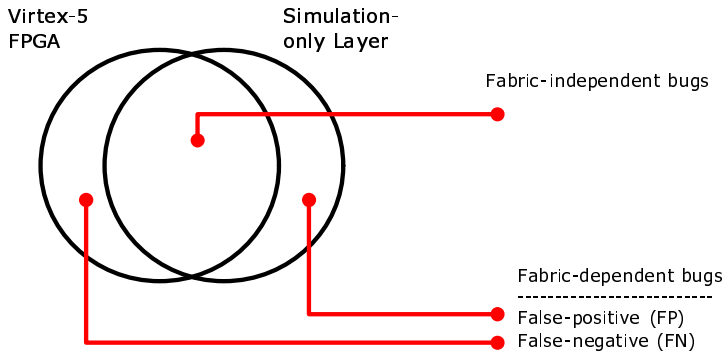


Figure 3.5: Fabric-independent and fabric-dependent bugs

simulation-only layer, representing a fabric-independent FPGA. For example, the size of a real bitstream is determined by the resources used whereas the size of a SimB can be adjusted for design-/test- specific needs. In particular, during simulation, the size of a SimB can be adjusted to increase verification coverage (e.g., see Section 5.4.1), to exercise corner cases of a design (e.g., see BUG-Example.FT.3), to increase/decrease the period when errors are being injected, and to store more or fewer state data of simulated RMs. The mismatches between the two can lead to bugs that remain undetected using the simulation-only layer (i.e., False Negative bugs) and bugs that are incorrectly reported using the simulation-only layer (i.e., False Positive bugs). These bugs are referred to as *fabric-dependent* bugs, and fabric-dependent bugs exposed by using the simulation-only layer can only be used as a reference to identify a real bug in the implemented design. For example,

- Since the simulation-only layer interprets and checks the module signatures of SimBs, simulation can assist in identifying bugs such as transferring an incorrect bitstream. Since the simulation-only layer correlates SimB contents with the state of simulated RMs, simulation also assists in identifying bugs such as accessing an

incorrect state bit in a bitstream. However, since the format of a SimB is significantly different from that of a real bitstream, in general, bugs in the configuration data of a real bitstream may not be exposed using a SimB. For example, flipping a bit in a real bitstream can result in the wrong logic being configured whereas flipping a SimB bit may result in an incorrect value being restored to an RTL signal.

- Since the simulation does not use pad words/frames in the SimB, the simulation-only layer cannot expose bugs related to pad words/frames (e.g., using an incorrect number of pad words/frames, see BUG-Example.XDRS.7 in Section 5.1.1).
- Since the SimB uses RRID/RMID to replace placement information of the simulated RM, a SimB cannot expose bugs related to module placement (e.g., rellocating an RM to a position that overlaps another RM).
- Since each RM is viewed as a single entity in the simulation-only layer, our approach only supports simulating modular reconfiguration. The simulation-only layer cannot be used to simulate fine-grained reconfiguration, which directly modifies individual configuration bits instead of an entire RM. For example, writing a SimB to the simulated configuration port cannot change an arbitrary AND gate of the user design to an OR gate, which could be achieved on the target FPGA using fine-grained reconfiguration.

As described in Section 2.1, design bugs can also be introduced in the implementation stage. Since the simulation-only layer does not contain any implementation-related information, it does not and does not aim to provide assistance in verifying implementation-related bugs, such as:

- Timing violation errors in the placed and routed design, and
- Possible short or open circuits, if any, caused by partial reconfiguration [11].

Referring to the Functional Verification Objective (see Section 1.1), this thesis aims to assist designers in detecting functional bugs, which are, by definition, not related to the FPGA fabric. Therefore, although the simulation-only layer can only be used as a reference to indicate fabric-dependent bugs and cannot expose implementation-related bugs, it still fulfills the Functional Verification Objective of the thesis.

Simulating Various DRS Design Styles. The simulation-only layer can be used to simulate and verify modular reconfigurable DRS designs. In particular, RMs that are mapped to the same RR are virtually multiplexed according to the SimBs written to the simulated configuration port. Since modules are swapped as atomic entities, RMs/RRs can be selected using numerical IDs contained in a SimB. For testing purposes, errors are injected to both static and reconfigurable regions during reconfiguration to mimic the spurious RM outputs and undefined RM state. For modular reconfigurable DRS designs, since the user design does not interact with the configuration memory, bitstream contents and logic allocation information, the mismatches in Table 3.5 are not, in general,

exposed. The simulation-only layer is therefore able to accurately model the modular reconfiguration process. The simulation-only layer can also be used to simulate and verify more flexible DRS design styles.

- State saving and restoration are simulated by mapping RTL signals to simulation-only layer frames and merging RTL signal values to SimBs. Using the simulation-only layer, the simulated user design saves and restores module state by reading and writing SimBs. However, the allocation of state bits is different between the simulation-only layer and the target FPGA.

- The simulation-only layer only maps RTL signals of RMs to simulation-only frames. To simulate state saving and restoration of flip-flops/memory cells of a static module, the designer has to change the static module to be a reconfigurable region for simulation purposes. As long as the modified system does not perform reconfiguration, it is functionally equivalent to the original one. The designers can then create `.sll` files to map RTL signals of the newly created RM, which is originally a static module, into simulation-only frames. However, the design has to be modified for simulation purposes, which incurs additional overheads.

- To simulate DRS designs that perform module relocation, the simulated user design relocates a simulated RM by updating the RRID/RMID fields of a SimB. In particular, the simulated user design reads a readback SimB from the ICAP artifact, updates the RRID/RMID fields, and writes the updated SimB to the ICAP artifact. The simulated RM instantiated in the target RR is then activated accordingly, as if it were relocated from the original location. However, the simulated module relocation process does not accurately model module placement information.

- To simulate the process of updating LUTs for fine-grained reconfigurable designs, the designer can model the content of a LUT as a memory cell and map the memory cell to the simulation-only frame using the `.sll` file. During simulation, the simulated user design updates the LUT by reconfiguring a SimB that changes the memory cell of the LUT. However, the configuration bits of a simulated LUT may not match those of real FPGAs, and the simulation therefore does not accurately represent what is implemented.

- Since the simulation-only layer mimics the behavior of off-the-shelf FPGAs, it cannot be used to simulate coarse-grained reconfiguration that targets customized reconfigurable architectures.

- The simulation-only layer can be used to simulate DRS designs that perform external reconfiguration, of either a modular or a non-modular nature. In particular, both the external reconfiguration controller and the ICAP artifact can be instantiated in the simulation testbench. DPR can be modeled by transferring SimBs in a similar manner to that used for internally reconfigured DRS designs. Since the reconfiguration controller is not part of the FPGA-based user design, it can be modeled in a behavioral style.

Since the simulation-only layer abstracts away the physically dependent information of DRS designs, it does not model the FPGA fabric at the level of detail required to accurately simulate more flexible DRS design styles such as module relocation and fine-grained reconfiguration. However, it can still be used to detect fabric-independent bugs of non-modular DRS designs. Referring to the Mainstream Objective (see Section 1.1), this thesis focuses on DRS design styles supported and recommended by vendor tools, i.e., modular reconfiguration. Therefore, the simulation-only layer is not able to and does not aim to provide complete support to the simulation-based verification of module relocation or fine-grained reconfiguration.

3.4 Summary

This chapter has proposed a general modeling methodology to facilitate simulation-based functional verification of DRS designs. Our approach introduces a simulation-only layer to emulate the behavior of the FPGA fabric and assist in verifying the correctness of the user design. Figure 3.6 redraws Figure 2.7 and adds a third, dashed line indicating simulation-based functional verification using the simulation-only layer. Compared with fabric-accurate simulation (e.g., [56]), the simulation-only layer abstracts away the physical layer of a DRS design and therefore enhances the verification productivity. Compared with traditional MUX-based approaches (e.g., [54, 69]), the simulation-only layer models characteristic features of DPR and the simulation accuracy has been improved significantly.

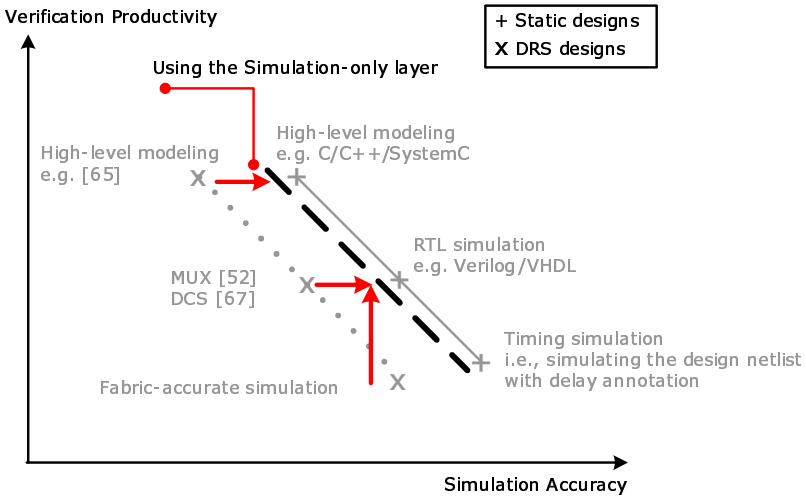


Figure 3.6: Tradeoff between simulation accuracy and verification productivity (using the simulation-only layer approach)

The simulation-only layer itself can be viewed as a fabric-independent FPGA device and using the simulation-only layer assists designers in detecting fabric-independent functional bugs of modular reconfigurable DRS designs. Our approach fulfills the Functional Verification Objective and the Mainstream Objective of this thesis. Furthermore, we are able to provide some assistance in verifying DRS designs that perform module relocation and fine-grained reconfiguration. Unfortunately, since there are mismatches

between the simulation-only layer and the target FPGA device, fabric-dependent bugs could be missed by simulation and can only be tested on the target FPGA.

Even if the simulation-only layer is not completely accurate, it can still be valuable to simulate a DRS design using the simulation-only layer. As described in Section 2.1, for verifying static designs, it is essential to thoroughly simulate a design so as to detect as many bugs as possible before field testing. Similarly, for DRS designs, simulation does not aim to replace on-chip debugging. Following the Tools and Methodologies Objective of this thesis (see Section 1.1), our simulation-only layer assists in identifying fabric-independent bugs in the early stage of the design cycle, and designers can thereby leave the fabric-dependent part of the design to be tested on the target FPGA.

Although we have focused our discussion at RTL level in this chapter, the proposed simulation-only layer can also be applied to high-level modeling. High-level modeling can ignore characteristic features that are related to RTL signals (e.g., the mapping between state data contained in a SimB and the value of RTL signals) and focus just on features that are essential to high-level modeling (e.g., module swapping, bitstream traffic). These ideas are presented in Chapter 4 which discusses the application of the simulation-only layer concept to high-level modeling and RTL simulation.

Chapter 4

ReSim-based Simulation

This chapter describes two possible ways of implementing the simulation-only layer (see Figure 4.1, which is the same as Figure 3.3). The simulation-only layer can be used to model and simulate DRS designs at various levels of abstraction (see Figure 4.2). For high-level modeling, SystemC [40] is the de-facto modeling language. Our work extends a previous open source SystemC library, ReChannel [67] with concepts of the simulation-only layer, and we refer to this library as Extended ReChannel (see Section 4.1). We created a ReSim library from scratch for RTL simulation of DRS designs (see Section 4.2). The ReSim library is implemented in a Hardware Verification Language (HVL), SystemVerilog [36], which integrates better with RTL designs compared to SystemC.

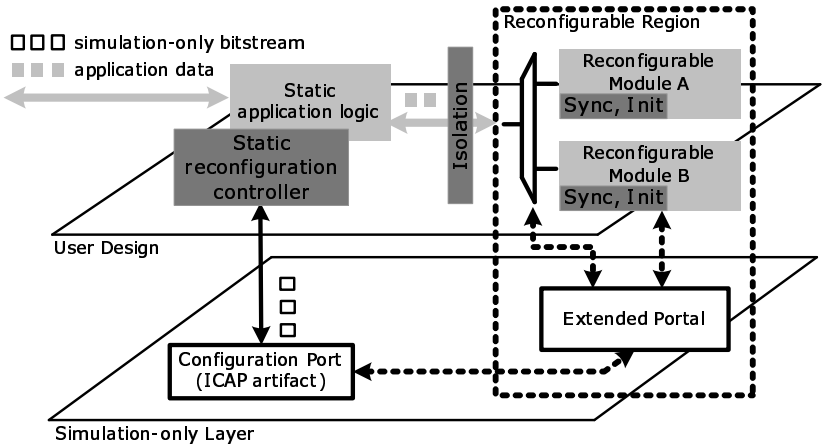


Figure 4.1: Using the simulation-only layer (Same as Figure 3.3)

Using ReSim and Extended ReChannel, the designer needs to create a Tcl script describing DPR-related parameters in addition to his/her user design files (see Section 4.3). DPR-related parameters include signals that cross the boundary of static and reconfigurable regions, the affiliation of RMs to RRs and the target FPGA family (see the example in Figure 4.2). ReSim then automatically generates the source code for artifacts (e.g., SimB files as indicated by small open boxes in Figure 4.2) based on this Tcl script. The generated artifacts can be directly used with the Extended ReChannel and the ReSim libraries. Alternatively, the designer can edit the generated artifacts for design-/test-specific needs, such as error injection or state restoration.

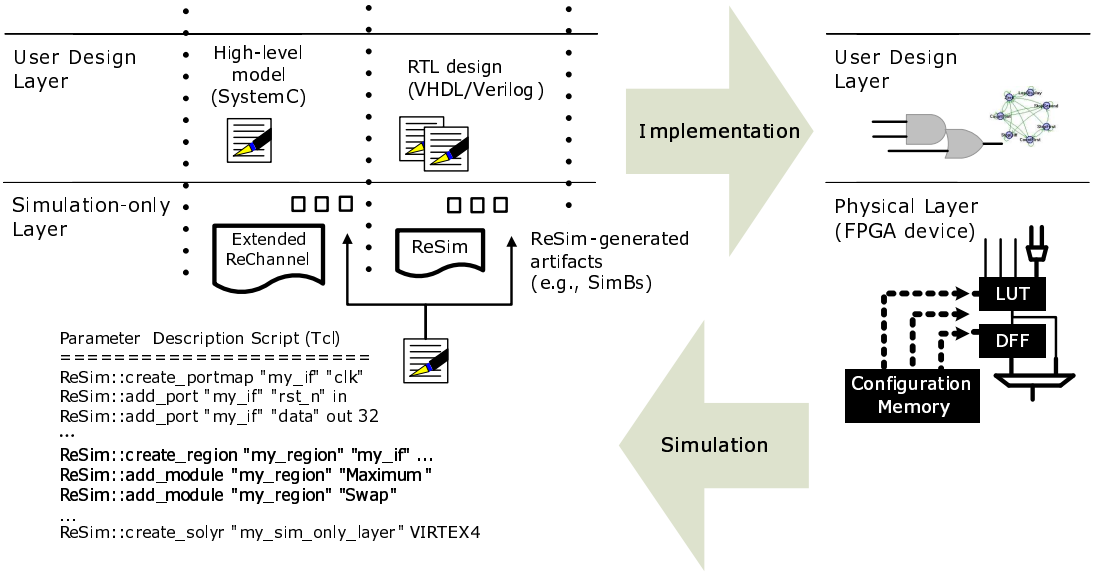


Figure 4.2: Simulating a DRS at various levels of abstraction

The ReSim-based design flow is fully compatible with existing and mainstream EDA tools. High-level models with the Extended ReChannel can be compiled and run on any machine with a C++ compiler such as Visual Studio. The ReSim library, on the other hand, uses an open source SystemVerilog class library known as the Open Verification Methodology (OVM) [27], and can be simulated with existing HDL simulators such as ModelSim. The verified RTL design can be synthesized and implemented using standard FPGA vendor tools such as PlanAhead.

4.1 High-level Modeling

By offering 1000x speedup compared with cycle-accurate RTL simulation, high-level modeling approaches assist designers in exploring design options, evaluating system performance, and verifying hardware architecture and embedded software [9]. By performing early system integration, the designer can detect system-wide bugs before the detailed RTL design is available. This can significantly reduce the number of design iterations for the project.

4.1.1 Behavioral Level Modeling

High-level modeling can be further classified as being performed at behavioral level or at transaction level. The behavioral level model of DRS only assists in verifying the very abstract operations of the application and hides all the details of reconfiguration (see Figure 4.3[1]). To focus on proof-of-concept designs, behavioral modeling only needs to capture the module swapping operations of partial reconfiguration. In particular, the static region communicates with the RM through an `rc_portal`, a MUX-like class of

[1]Similar to Figure 3.3, the application logic is lightly shaded, reconfiguration machinery is moderately shaded, and simulation-only artifacts are indicated by open boxes.

the ReChannel library, and it is acceptable to use a compile-time defined reconfiguration delay. Furthermore, in order to increase development productivity, there is no clear boundary between the user design and the simulation-only artifacts (illustrated by half-open half-shaded blocks). For example, both the user-defined reconfiguration controller and the configuration port of the FPGA could be modeled with a single `rc_control` class of the ReChannel library [67].

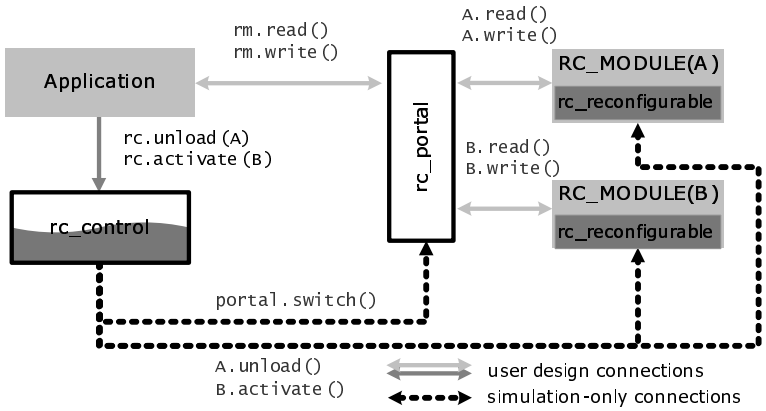


Figure 4.3: Behavioral modeling using ReChannel, after [67]

Upon receiving a request to reconfigure a region, the static region reconfigures the RM via two function calls (i.e., `rc.unload(A)` & `rc.activate(B)`) to the `rc_control` class, whereas the bitstream traffic is not explicitly modeled. Internal to the ReChannel library, the `rc_control` class switches the the `rc_portal` and selects the new RM after a compile-time defined reconfiguration delay.

Despite making only limited contributions to detecting DPR-related bugs, behavioral level modeling is useful for design space exploration to see whether it is worthwhile adopting dynamic reconfiguration as a strategy and for evaluating the available reconfiguration options BEFORE, DURING and AFTER reconfiguration as early as possible. These include, for example, whether the system should wait for the current RM to finish processing, kill the current RM, or save the current RM state BEFORE reconfiguration. After the desired options have been chosen, the behavioral model can be considered to be an executable, non-ambiguous specification of the DRS design.

4.1.2 Transaction Level Modeling

Transaction-level Modeling (TLM) aims to verify system integration between hardware modules and hardware/software [59]. It requires clearly defined module execution and communication flows and therefore must model the distinction between the user design and the simulation-only layer. While the original ReChannel library models module swapping, we have extended ReChannel with new classes and data structures to model the characteristic features of DPR (see Section 3.2.2) such as bitstream traffic and triggering condition. However, since TLM modeling abstracts away the signal level details of individual modules, our extensions do not include signal-level reconfiguration activities such as spurious RM outputs and undefined initial RM state.

By incorporating bitstream traffic and the triggering condition, TLM models could be used in the following example scenarios:

- For design space exploration or performance evaluation purposes, the bitstream traffic should be simulated to decide whether to use a dedicated or a shared datapath for bitstream transfer.
- System-wide integration bugs related to a sequence of reconfiguration events can be detected in the TLM model. For example, deadlocks caused by circular waiting of multiple swapped out modules can be detected by accurately modeling the timing and triggering of module swapping. As another example, illegal reconfiguration requests to reconfigure a module still undergoing reconfiguration can be detected by more accurately modeling the reconfiguration delay.
- In order to test and debug any embedded software that could be in control of the reconfiguration process, the TLM model should be functionally equivalent to the implemented design from the programmer's point of view. The designer therefore needs to simulate the bitstream traffic byte-by-byte if the software buffers and transfers bitstream data byte by byte. Software bugs such as transferring an incorrect amount of bitstream data could be detected in simulation.

Figure 4.4 illustrates TLM modeling of DRS designs, which can be viewed as the TLM representation of Figure 4.1. Components of the simulation-only layer are implemented by extending the ReChannel library. Since we reused most of the source code from ReChannel, the workload of our extension was trivial. To facilitate the modeling of bitstream traffic, we created an `rc_icap_wrp` class. The `rc_icap_wrp` class has two member functions, `write_cdata()` and `read_cdata()`, which are to be called by the user-defined reconfiguration controller to write and read back SimBs. We have also derived an `rc_portal_controller` class from the built-in `rc_control` class to more accurately model the triggering condition of module swapping. In particular, the `reconfigure()` member function of the `rc_portal_controller` class is not called until all bytes of the SimB have been successfully written to the `rc_icap_wrp` class.

Using our extensions to ReChannel, the designer models both static and reconfigurable modules in the user design layer. It should be noted that the designer should explicitly model each and every building block of the system so as to verify the design intent. In other words, all user-defined modules in Figure 4.4 (lightly and moderately shaded blocks) are modeled according to the design intent described by the design specification. For example, instead of using the built-in `rc_control` class, the user-defined reconfiguration controller is explicitly modeled as a real entity in the static region.

By explicitly separating the user design and the simulation-only layer, the designer can focus on the verification of the execution flow of modules and the inter-module communication. Upon receiving a request to reconfigure a region, the static design deactivates the current module (`rm.deactivate()`), and transfers the SimB, word by word to the `rc_icap_wrp` class (`icap.write_cdata(SimB)`). Internal to the extended ReChannel library, the `rc_icap_wrp` class calls the `rc_portal_controller`, which unloads the current RM and loads the new RM after the successful transfer of all SimB data. Finally,

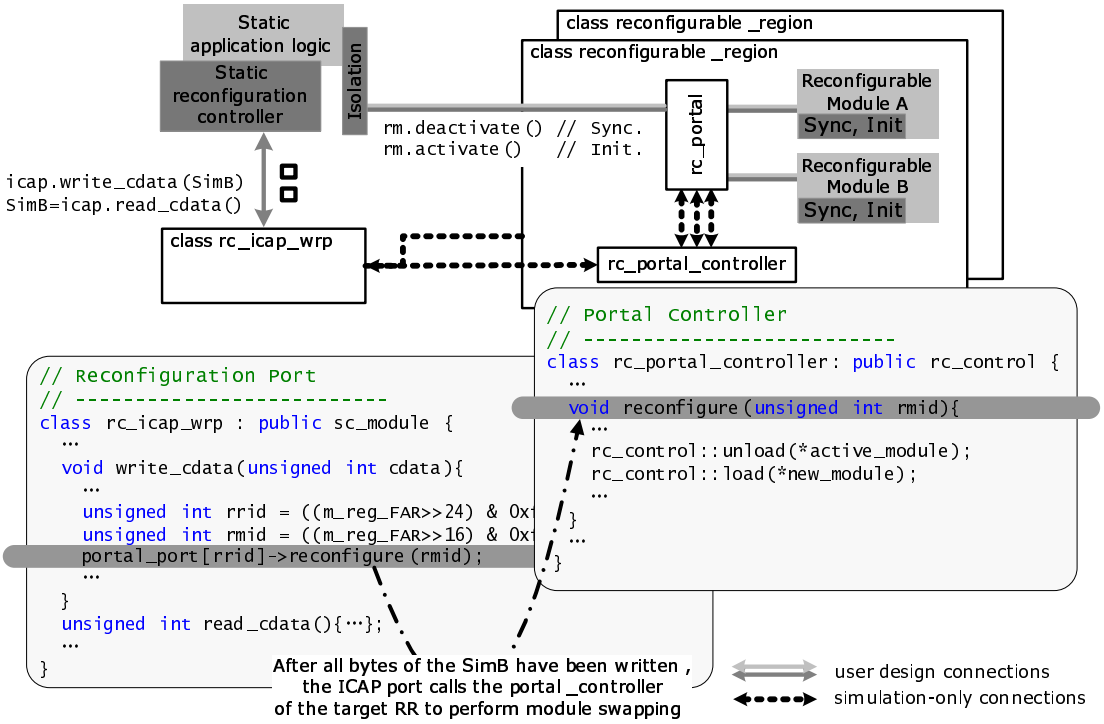


Figure 4.4: TLM modeling using Extended ReChannel

the static design initializes the new RM (`rm.activate()`). Such execution and communication flow are the same as in the implemented DRS, except that reconfiguration events are modeled by function calls instead of signals transitions.

4.2 RTL Simulation

In RTL simulation, the designer refines and maps the verified execution and communication flows of the TLM model to a signal-level RTL design. In addition to modeling bitstream traffic, module swapping and the triggering condition, RTL simulation also needs to capture DPR features such as Bitstream Content, Spurious Outputs and Undefined Initial State as illustrated in Section 3.2.2. In particular,

- Simulating the bitstream traffic can assist in identifying bugs in the bitstream transfer datapath (e.g., cycle-mismatch, FIFO overflow/underflow, errors in driving the ICAP signals, etc ...)
- Simulating the module swapping and its triggering condition can assist in identifying bugs related to the timing and sequence of reconfiguration events (e.g., failing to finish partial reconfiguration before resetting the RM)
- Modeling the spurious RM outputs can assist in identifying bugs in isolating the region undergoing reconfiguration (e.g., failing to isolate the RR and isolating the RR too early or too late)

- Modeling the undefined initial RM state can assist in identifying bugs in initializing the newly configured RM (e.g., failing to feed pipelines of the RM with known data, failing to setup the new RM with the correct initial state)
- Modeling the bitstream content can assist in identifying bugs in the configuration data (e.g., transferring an incorrect bitstream, extracting incorrect state data from the bitstream)

Figure 4.5 illustrates RTL simulation of DRS designs using the ReSim library, which can be viewed as the RTL representation of Figure 4.1. We implemented the simulation-only layer using a HVL, SystemVerilog, which more seamlessly integrates with RTL designs than SystemC-based ReChannel does. Our ReSim library adopts the Open Verification Methodology (OVM) [27], and we applied the transaction-based communication and the reusability guidelines of OVM to the implementation of the ReSim library.

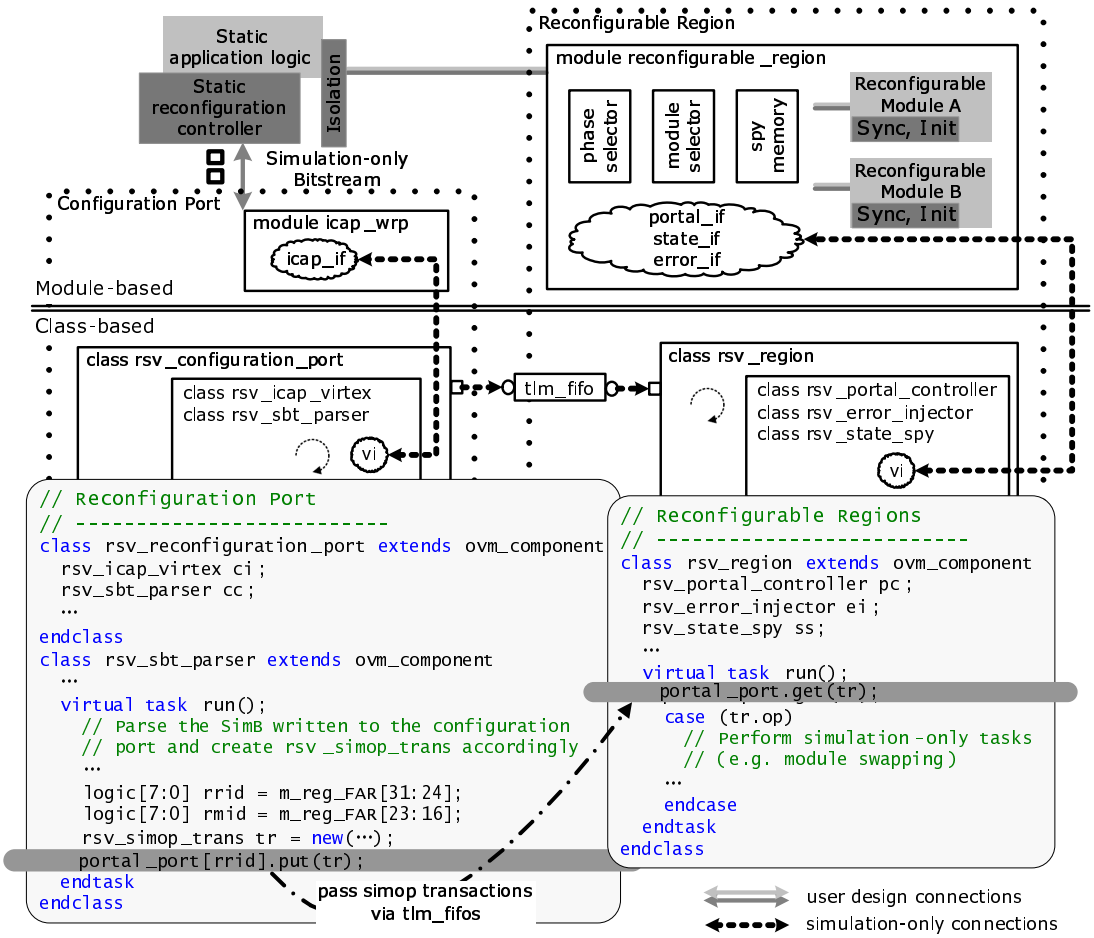


Figure 4.5: ReSim-based RTL simulation

According to the modeling guidelines of OVM, artifacts of ReSim communicate with each other via transactions, i.e., collections of attributes similar to the `struct` data aggregation mechanism of the C programming language. For example, the transaction that reconfigures a new module has an operation attribute set to "write module" and a data attribute set to the numerical ID of the new module. Transactions are *produced* and *consumed* by artifacts running in parallel. In particular, a producer of transactions is

analogous to a function caller in SystemC-based Extended ReChannel, and a transaction consumer to a function callee. Using the notations for OVM transactions (see Figure 4.5), the initiator of a transaction channel is depicted with a ▫ symbol whereas the target is depicted with a ○ symbol. The arrows indicate the directions of the transaction data flow and the ↻ symbol indicates a standalone thread running in a component.

According to the reusability principles of OVM, the simulation-only layer of ReSim is separated into a *module-based* part and a *class-based* part, which are connected via *SystemVerilog Interfaces* (shown as clouds and the Ⓥ symbols). The module-based part instantiates the artifacts as part of the user design and interacts with the user design via RTL signals. In particular, the `icap_wrp` module is instantiated in the user design hierarchy as if it were a real configuration port, and it groups the IO signals of the ICAP port into an `icap_if` interface. The `reconfigurable_region` module instantiates the user-defined reconfigurable modules, a MUX-like `module_selector`, a switch-box like `phase_selector` module and a `spy_memory` module. Similar to the Virtual Multiplexer in [54], the `module_selector` interleaves all IO signals of RMs connected in parallel and selects one active module. Similar to the *task modeler* and the *task selector* in [69], the `phase_selector` connects the static region with RMs in normal operation and connects error sources (e.g., undefined "x" values) to the user design *during* reconfiguration. The `spy_memory` module implements the configuration memory and the simulation-only frames as described by Section 3.2.1

The class-based part controls the operation of the module-based part. It implements the functionality of the Extended Portal and mimics the characteristic features of DPR (see Section 3.2.2). In particular, to facilitate the modeling of bitstream traffic, the `rsv_icap_virtex` class interacts with the user-defined reconfiguration controller via the `icap_if` interface and the `rsv_sbt_parser` class parses SimBs. To model module swapping and its triggering condition, the `rsv_portal_controller` class does not select a new RM (i.e., does not switch the `module_selector`) until a SimB is written to the ICAP artifact, and does not connect the newly selected RM (i.e., does not switch the `phase_selector`) until all bytes of the SimB has been successfully written to the ICAP artifact. To model spurious module outputs and undefined module state, an `rsv_error_injector` class drives error sources to both the static region and the RMs. Last but not least, to correlate bitstream content with simulated RMs, the `rsv_state_spy` class checks module signatures and performs state saving and restoration of simulated modules. By modeling characteristic features of DPR, ReSim increases the simulation accuracy compared with previous work in [54] and [69].

Although the module- and class-based implementation of ReSim is more complex than the implementation of the Extended ReChannel, applying the OVM library provides two significant benefits. Firstly, using the object-oriented techniques of SystemVerilog and OVM, the class-based part can be flexibly overridden and reused for design- and test-specific needs. For example, ReSim can be ported to a different FPGA family by deriving a new class overriding the existing `rsv_configuration_port` class. As another example, the default behavior of the `rsv_error_injector` class is to drive undefined "x" values as error sources. The default errors can be overridden with other error values such as logic "1", logic "0", random value, last known value, and even design-/test-specific error sequences.

Secondly, by separating the implementation into module- and class-based parts, ReSim is able to model the simulation-only layer without affecting the user design. Since the module-based part of artifacts (i.e., the `icap_wrp` module and the `reconfigurable_region` module) are instantiated as part of the user design, user-defined modules (lightly and moderately shaded blocks in Figure 4.4) can be constructed and connected just as intended by the design specification. Furthermore, in order to use SimB as a trigger for module swapping, the simulation-only layer needs to connect the ICAP artifact with multiple reconfigurable regions. In Extended ReChannel, such connections are implemented by assigning pointers. In particular, pointers of the `rc_icap_wrp` class are assigned with instances of reconfigurable regions, which does not affect SystemC models of the user design. However, at RTL level, if the simulation-only layer is described by traditional HDL languages, the ICAP artifact and the reconfigurable regions would only have module-based parts and could only be connected via RTL signals, which would require undesirable modifications to the user design. Using the SystemVerilog language and the OVM library, such connection is implemented in the class-based part of the simulation-only layer. In particular, the class-based parts of the ICAP artifact and the reconfigurable regions are connected via `tlm_fifos`, which do not affect the module-based user design components (See Figure 4.5).

Since ReSim captures the characteristic features of DPR, the designer is able to model and verify the cycle-accurate description of the user design. Furthermore, since ReSim-based simulation does not require changing the design for simulation purposes, the simulated design is *implementation ready* and can be mapped to the target device without changing the design source code.

4.3 Parameter Script

In order to reuse the Extended ReChannel and ReSim libraries for various DRS design styles, designers need to parameterize the library with design-specific information. For example, for both high-level modeling and ReSim-based RTL simulation, the libraries need to know the topology (i.e., the affiliation of RMs and RRs) of the user design so as to correctly assign numerical IDs to each RM and RR and to parameterize SimBs with the RRID/RMID of target modules. For ReSim-based RTL simulation, the `module_selector` needs to be parameterized with the interfacing signals crossing the RR boundary so as to correctly interleave these signals. Optionally, designers may also wish to extend the ReSim library for design- or test-specific needs (e.g., user-defined error sources). In order to hide the details of the library implementation and to improve flexibility, ReSim provides a set of Tcl APIs to help designers integrate the simulation-only layer into their design-specific simulation environment.

Figure 4.6 illustrates an example of a parameter script. By calling APIs in a Tcl script, the user describes the interfacing signals crossing the RR boundary (Line 3-6), the affiliation of RMs and RRs (Line 8-10), and the FPGA family used by the design (Line 12). ReSim generates the parameterized artifacts that can then be correctly instantiated in the design hierarchy.

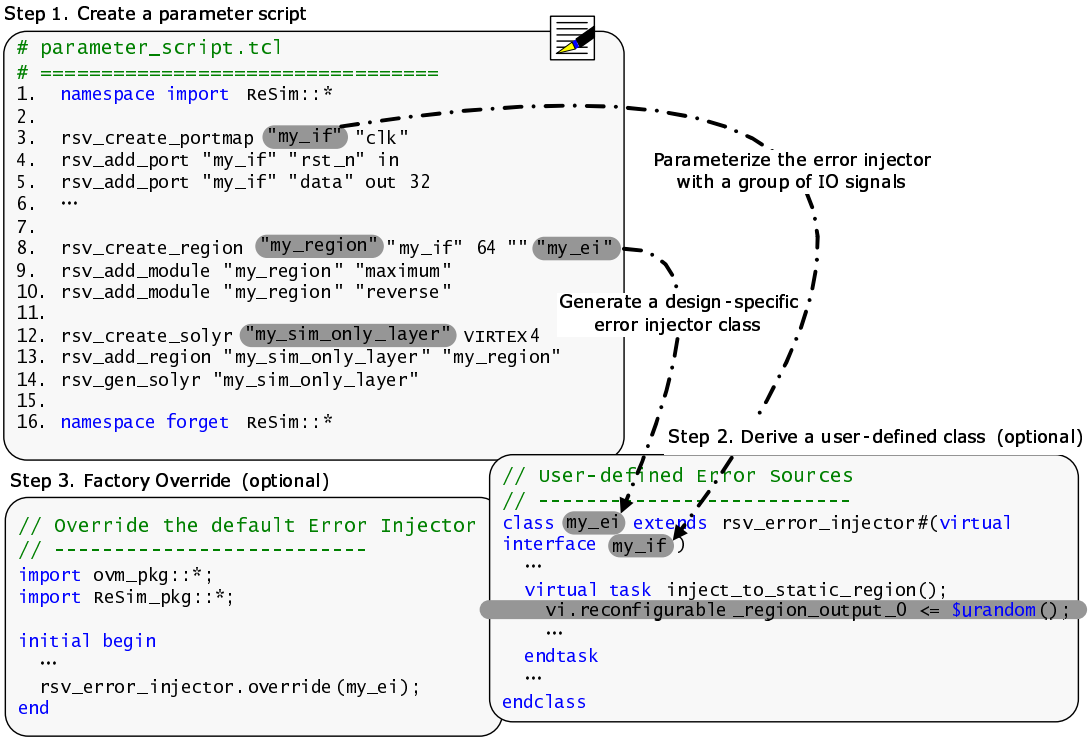


Figure 4.6: An example of a parameter script

For high-level modeling using Extended ReChannel, the designer needs to instantiate the parameterized configuration port (i.e., the `rc_icap_wrp` class) in the user design and load the generated SimBs into the simulation environment. Because of the variety of coding-styles for ReChannel-based modeling, we are only able to generate templates for the `reconfigurable_region` class. The designer needs to modify these templates and instantiate the reconfigurable regions in the user design.

For ReSim-based RTL simulation, the generated artifacts are ready to be used directly. The designer needs to instantiate the configuration port (i.e., the `icap_wrp` module) and one or more `reconfigurable_region` modules (e.g., the `my_region` module defined in Line 8 of Figure 4.6) as part of the user design. The designer also needs to instantiate the top-level of all class-based artifacts (e.g., the `my_sim_only_layer` class in Line 12 of Figure 4.6), and load the generated SimBs into the simulation environment.

As an option, the designer can then edit the generated artifacts for design-/test-specific needs. There are two typical usage scenarios that require modifying the generated artifacts for ReSim-based RTL simulation. The example (see Step 2 in Figure 4.6) illustrates changing the default "x" injection to a user-defined error injector that drives random values to the static region. Such extension is implemented by redefining the *virtual* functions in the derived classes (see Figure 4.6). Using the factory mechanism provided by the OVM library [27], the default error injector can be easily overridden with the derived `my_ei` class. Furthermore, the Tcl script automatically generates code (see Step 3 in Figure 4.6) that performs the factory override operation. Designers can thus focus on defining the virtual functions in their derived classes without bothering about how to integrate the derived classes. In contrast, the error injection used by [69] only support a constant "x" error value.

The other usage scenario involves simulating a DRS design that saves and restores module state. In particular, the parameter script generates a template logic allocation file for simulation (`.sll` file). The `.sll` file is used to map HDL signals of the user design to the `spy_memory` module, which implements the simulation-only frames as described in Section 3.2.1. The generated `.sll` file is empty by default. For DRS designs that saves and restores module state via the configuration port, the designer needs to modify the generated `.sll` file and explicitly specify the names of RTL signals and the simulation-only frame addresses the signals are mapped to (see Figure 3.4 for an example).

4.4 Capabilities and Limitations

The Extended ReChannel and ReSim libraries are two implementations of the simulation-only layer. Therefore, they have the same capabilities and limitations described in Section 3.3. In particular, the two libraries can only assist designers in detecting fabric-independent bugs and fabric-dependent bugs can only be identified on the target FPGA.

Limited by the complexity of implementation, ReSim has not yet implemented all functionality of the simulation-only layer described in Section 3.3. In particular, the ReSim library only supports

- ModelSim/QuestaSim 6.5g (thoroughly tested) or above (in theory),
- Virtex 4,5,6 FPGAs,
- The basic operations of ICAP: 32-bit ICAP, configuration write, configuration readback (including readback busy), basic ICAP registers (i.e., CRC, CMD, FAR, FDRI, FDRO, IDCODE) and basic ICAP commands (NULL, WCFG, RCFG, RCRC, GRESTORE, GCAPTURE, DESYNC), and
- Saving and restoring flip-flop/register values via the configuration port.
- State restoration by modifying the state bit (i.e. `INT0/INT1` bit) of a bitstream.

ReSim does not yet support,

- Other HDL simulators (because ReSim uses ModelSim/QuestaSim built-in commands)
- Xilinx 7 series FPGAs or Altera FPGAs
- Advanced operations of ICAP: abort sequence, status register readback, etc
- Saving and restoring BRAM values via the configuration port.
- State restoration by modifying the SET/RESET bit of a bitstream.

It should be noted that these are only limitations imposed by the current implementation of the library.

4.5 Summary

Table 4.1 compares the characteristic features of DPR with the proposed simulation-only layer and two possible implementations (i.e., Extended ReChannel, ReSim).

Table 4.1: Summary of Extended ReChannel and ReSim

Characteristic features of DRS designs	Simulation-only Layer	Extended ReChannel	ReSim
Module Swapping and Triggering Condition	Virtual Multiplexing & use of SimBs to trigger the multiplexing	Yes	Yes
Bitstream Traffic	SimB	Yes	Yes
Bitstream Content	Check module signature. Update module state. Read/write the spy memory.	N/A	Yes
Spurious Outputs and Undefined Initial State	Inject errors to both the static region and reconfigurable modules	N/A	Yes

To facilitate high-level modeling of DRS designs, Extended ReChannel captures the "module swapping and triggering condition" and the "bitstream traffic" features while abstracting away the impact of the "bitstream content" and the "spurious outputs and undefined initial state" features of DRS designs. A high-level model of a DRS design can assist in design space exploration, detecting system-wide integration bugs and validating embedded software that controls the reconfiguration process.

To facilitate the verification of an *implementation-ready* DRS design, the ReSim library covers all characteristic features of DPR and therefore supports cycle-accurate RTL simulation of partial reconfiguration. ReSim-based RTL simulation can assist in detecting device-independent functional bugs of an integrated DRS while part of it is being reconfigured. Furthermore, ReSim does not require changing the design for simulation purposes and thus verifies the design intent.

To simplify the use of the Extended ReChannel and ReSim libraries, we have created a set of Tcl-based APIs. By describing DPR-related parameters (e.g., the affiliation of RMs and RRs) of a design, the Tcl APIs can automatically generate simulation-only artifacts (e.g., SimBs) to be integrated into a design-specific simulation environment. The generated artifacts can also be flexibly extended for design-/test-specific needs.

The ReSim library has been released as an open source tool under the BSD license, and is available from the project website at `http://code.google.com/p/resim-simulating-partial-reconfiguration`. The project website also includes the following documentation:

- The ReSim library: The library source code and three DRS design and simulation/verification examples.
- User Guide [29]: The user guide contains a step-by-step guide on using/extending the library, runing examples, Tcl APIs, and explains the code structure of the library.

- Case Studies [28]: The case study report contains design and verification experience of using the ReSim library.

Chapter 5

Case Studies

We demonstrate the value of ReSim and ReSim-based functional verification via a number of case studies. The first is a generic DRS computing platform, through which we aim to illustrate the process and results of ReSim-based verification on an *in-house*, processor-based DRS design. The second, fault-tolerant application uses DPR to recover from circuit faults introduced by radiation, and we aim to demonstrate verifying an *in-house*, non-processor based DRS system. Using a third-party design, a video-based driver assistance system [18], as our third case study, we then study the use of ReSim to perform functional verification of cutting-edge, complex, real world DRS applications. Finally, we present the application of ReSim to vendor reference designs.

Overall, we aim to demonstrate that ReSim is flexible enough to simulate various DRS design styles. It should be noted that, unless explicitly described, all bugs detected in our case studies were *real* bugs exposed during the project development. Furthermore, unless explicitly described, all bugs revealed design flaws in the *user designs* instead of in the simulation-only layer or in any simulation testbench. Therefore, our case studies can be used as examples to guide future designers in verifying their DRS designs.

5.1 Case Study I: In-house DRS Computing Platform

The Design Under Test (DUT) of our first case study is a generic DRS computing platform known as XDRS (see Figure 5.1). This platform is similar to existing generic slot-based DRS platforms such as the Erlangen Slot Machine [13], the VAPRES streaming architecture [41] and the Sonic video processing system [76]. Using a platform-based design methodology, a designer can map various hardware tasks to pre-defined reconfigurable regions and thereby customize the platform for various applications [73]. After thoroughly verifying the platform, the verification effort of applications customized/derived from the platform can be significantly reduced [68]. This section aims to use XDRS as a representative system to study the functional verification of generic DRS computing platforms. In particular, we applied top-down modeling to verify system integration and to test system software, used coverage-driven verification to test hardware peripherals attached to the processor bus, and used the platform-based verification method to test

a second DRS application mapped to XDRS. We demonstrate that ReSim can be seamlessly integrated into the verification of the DRS computing platform and of the DRS applications mapped to the platform.

The XDRS computing platform has a control-centric processor and a computation-centric accelerator. The `xps_xdrs` accelerator module has 3 dynamically reconfigurable regions (i.e., RRs) and a `Producer/Consumer` module that interfaces the RRs with the rest of the system via the PLB bus. The in-house designed, bus-based reconfiguration controller, `xps_icapi`, is similar to customized controllers such as [17] [34], and the Xilinx `xps_hwicap` IP core [93]. Apart from configuring module logic, XDRS also supports saving and restoring module state via the configuration port in a way similar to [80, 44]. The case study runs a demo streaming application in which the RRs are reconfigured with simple computational cores (e.g., a `Maximum` module, a `Reverse` module) and are chained with the `Producer/Consumer` module to form a processing loop.

The primary focus of this case study is to verify that the reconfiguration machinery and its software driver (moderately shaded parts of Figure 5.1) are correct and are correctly *integrated* with the rest of the system hardware and software. In particular, the in-house designed reconfiguration controller `xps_icapi` transfers bitstreams from the DDR2 memory to the ICAP according to the parameters (e.g., bitstream address, length, etc) set by the software. The `xps_xdrs` accelerator uses a `SyncMgr` module to synchronize the RMs in the processing loop before and after reconfiguration, and uses an `Isolation` module to avoid the propagation of erroneous signals from the region undergoing reconfiguration.

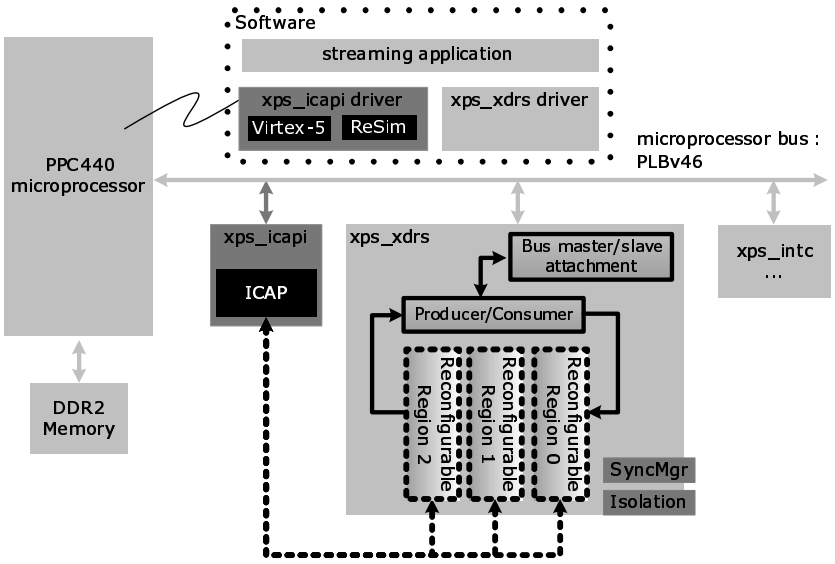


Figure 5.1: The XDRS demonstrator

Figure 5.2 illustrates the progress of designing and verifying the XDRS system in terms of Lines of Code (LOC) changed and bugs detected per week[1]. The LOC numbers were reported by a version control tool and included design source such as HDL, scripts, software (*.c, *.h), constraint files, development log files and project files. It should be noted that since we used the Embedded Development Kit (EDK) framework [96], some

[1]One Week = 35 hours full-time work by a designer with 3 years of FPGA design experience

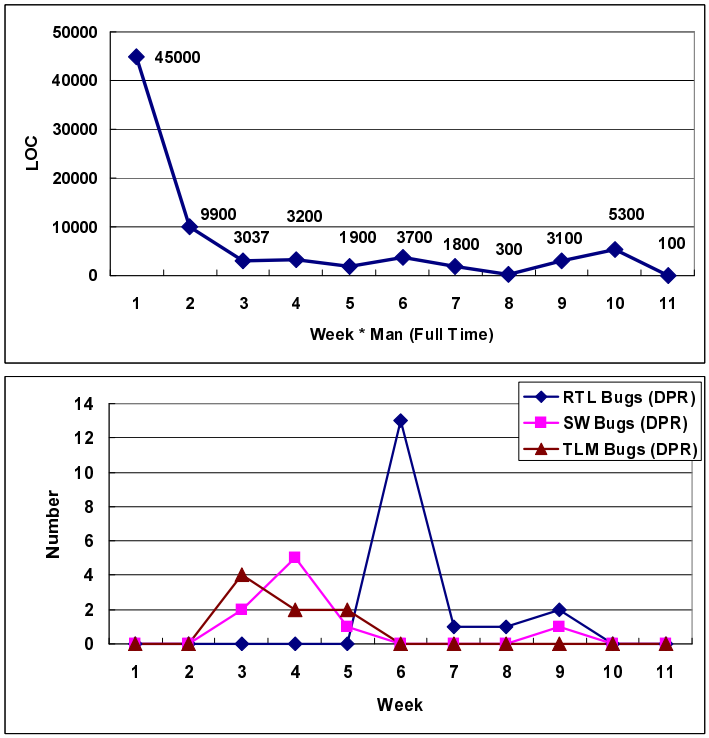


Figure 5.2: Development progress of the XDRS system

of the design source files were generated by the tool and also contributed to the LOC numbers. As a result, the LOC numbers should be thought of as reference data that indicate the relative development effort. The bugs recorded are all DPR-related bugs, which include hardware bugs in the reconfiguration machinery (either found at TLM level or RTL level) and software bugs in controlling partial reconfiguration.

- By the end of Week 1, the designer had finished setting up the workspace. The relatively high LOC numbers were contributed to by code generated by vendor tools and IP provided by the vendor. The designer had created a testbench to simulate the "hello world" software program as a sanity check and had not yet started working on the `xps_icapi` and `xps_xdrs` modules

- We applied ReSim to well-established static design verification methodologies, both FPGA-based and ASIC-based, to verify XDRS. In particular, we used a top-down modeling methodology [68] to verify XDRS at three abstraction levels, namely, behavioral level (Week 2), TLM level (Week 3-5) and RTL level (Week 6-9). At transaction-level, HW/SW co-simulation [68] was used to debug system software and the integration of software and hardware. At RTL level, we applied coverage-driven verification [66] to thoroughly test the XDRS platform.

- In the last 2 weeks, the design was implemented and tested on an ML507 board with a Virtex 5 FX70T FPGA and no more bugs were detected. The LOC were contributed to by changes to the design for optimization purposes, and implementation related files (e.g., constraint files). Regression tests were performed to verify changes to the design.

Transaction Level Modeling. Using extended ReChannel, the designer performed TLM modeling in Weeks 3-5 and verified the system integration of the XDRS platform. In particular, the designer created SystemC models for both the `xps_icapi` and `xps_xdrs` modules, and built a virtual platform for the whole XDRS system. The virtual platform was co-simulated with both the hardware modules and the target streaming application software.

Since the virtual platform was implemented before the RTL design was ready, TLM modeling assisted in detecting software bugs and software/hardware integration bugs at a very early stage, which significantly reduced the overall project time-line. Figure 5.3 is a screen shot of the co-simulation environment. The upper left window is the source code debugger for embedded software (C), the bottom window is the source code for the virtual platform (SystemC) and the upper right corner is the standard output of the virtual platform. Co-simulation allows designers to move back and forth between software and hardware to, for example, view software variables while tracing the hardware model.

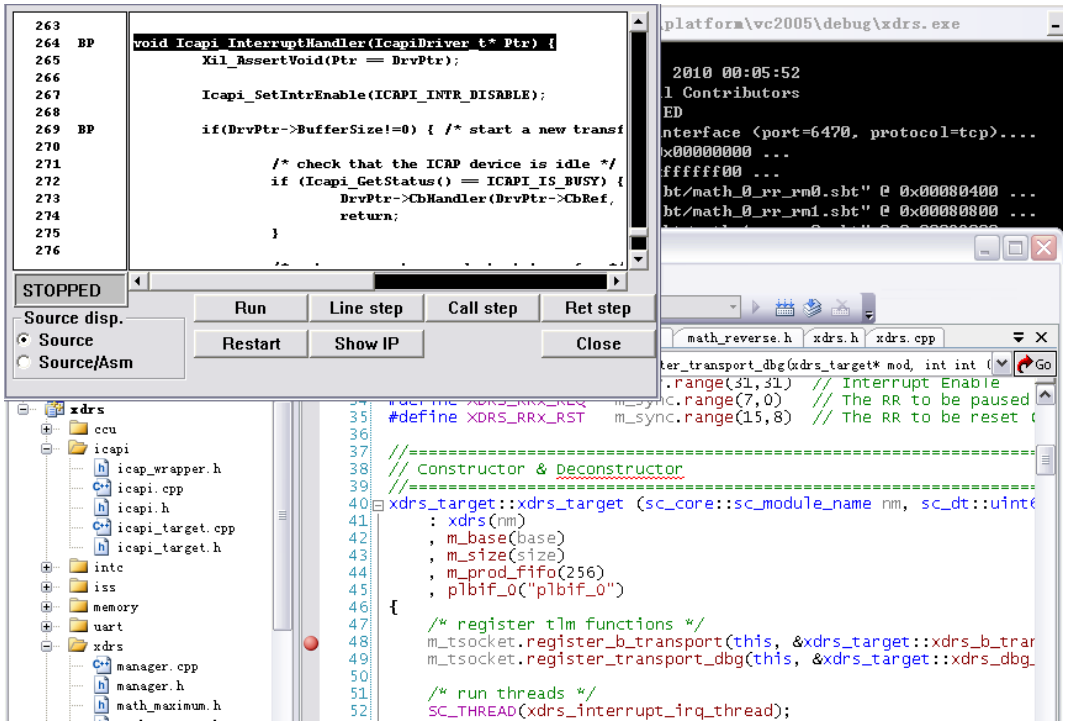


Figure 5.3: A Screen-shot of the co-simulation environment for the XDRS system

TLM modeling assisted in identifying and fixing 16 DPR-related bugs in the system (see Figure 5.2). Some DPR bugs captured in the TLM model revealed potential design defects in the system hardware (e.g., BUG-Example.XDRS.1, see page 68). For some DPR-related bugs, the source of error was the software running on the microprocessor (e.g., BUG-Example.XDRS.2). For DPR bugs such as BUG-Example.XDRS.3, the bugs were caused by both system software and hardware. These bugs could be more difficult to trace when simulating the reconfiguration process at the more detailed RTL level. Furthermore, DPR-related bugs such as BUG-Example.XDRS.2 and BUG-Example.XDRS.3 were detected since the Extended ReChannel accurately models the bitstream traffic and the triggering condition of module swapping in the TLM model.

Therefore, it was found to be highly desirable to have a simulation environment to test the integrated SW/HW system, including when it is being reconfigured.

BUG-Example.XDRS.1: The `xps_icapi` module uses an `ICAPI_DONE` flag to indicate the end of bitstream transfer. Such a flag should be set to 1 at power up but was incorrectly initialized to 0. This bug was detected in the TLM model of the system, but it revealed a potential bug that could also exist in the RTL design.

BUG-Example.XDRS.2: To accelerate bitstream transfer, the system software loads bitstream data from a slow flash memory and buffers the bistreams in a fast DDR2 memory. However, the software failed to flush the bitstream data from the processor cache to the DDR2 memory and the `xps_icapi` module transferred incorrect bitstream data from the DDR2 memory during reconfiguration. The bug was easily identified since unflushed SimB data was transferred to the simulated ICAP port but the new RM was not swapped in during simulation, and would not have been identified without modeling bitstream traffic.

BUG-Example.XDRS.3: The `SyncMgr` mistakenly requested a second reconfiguration before the first one had finished (i.e., an illegal reconfiguration sequence). The bug was detected because the delay associated with the first reconfiguration was accurately modeled by transferring the SimB using Extended ReChannel. The designer fixed the bug by modifying the hardware to report such illegal reconfiguration requests and by adding a `reconfiguration_in_progress` flag to the software driver.

TLM modeling offered reasonable simulation throughput while running the target software application on the virtual platform, and Extended ReChannel-based simulation assisted in detecting system integration bugs. However, the TLM model of XDRS contains 2000 LOC (SystemC), which is a significant overhead that can also introduce bugs. In this case study, we detected 7 bugs in the TLM model (not shown in Figure 5.2) that were introduced by the SystemC code and that were subsequently identified as false positive bugs.

Running Simulation. From Week 6 to Week 9, the verified virtual platform was manually refined to RTL. Since the demo streaming application does not exercise all possible scenarios of the XDRS platform, we applied coverage-driven verification to systematically test the platform. Since the microprocessor is a hard core on the FPGA, we focused on the verification of the `xps_icapi` and `xps_xdrs` modules. Since the XDRS system was created using the EDK development framework, the software-accessible registers and the bus interface logic are standard components and the designer could therefore focus on the testing of the core logic of the XDRS system (Week 6-8). In Week 9, the designer used the EDK framework to generate the software-accessible registers and bus interface logic for the `xps_icapi` and `xps_xdrs` modules. Although the generated logic accounts for a relatively high LOC count, it can be expected to be correct. The designer connected the generated logic with the core logic verified by Week 8, and fixed a few bugs in the interconnection of these modules.

We created a separate testbench (see Figure 5.4) to systematically test the core logic of the XDRS platform. The core logic of the XDRS platform has the same architecture as the original XDRS system but does not include any software-accessible registers or the bus interface. In particular, the reconfiguration machinery (moderately shaded blocks in Figure 5.4) is composed of the core logic of the `xps_icapi` module, i.e., the `ICAP-I` reconfiguration controller [52], the `SyncMgr` module and the `Isolation` module. The application logic is composed of the core logic of the `xps_xdrs` module, which contains the `Producer/Consumer` module and the RRs. BEFORE reconfiguration, as listed in the specification (see Figure 5.4), reconfiguration requests are blocked (not acknowledged) until the RM becomes idle. DURING reconfiguration, the `Isolation` module drives default values to the static region and isolates the RR undergoing reconfiguration. AFTER reconfiguration, the newly configured module is reset to IDLE by the `SyncMgr` module.

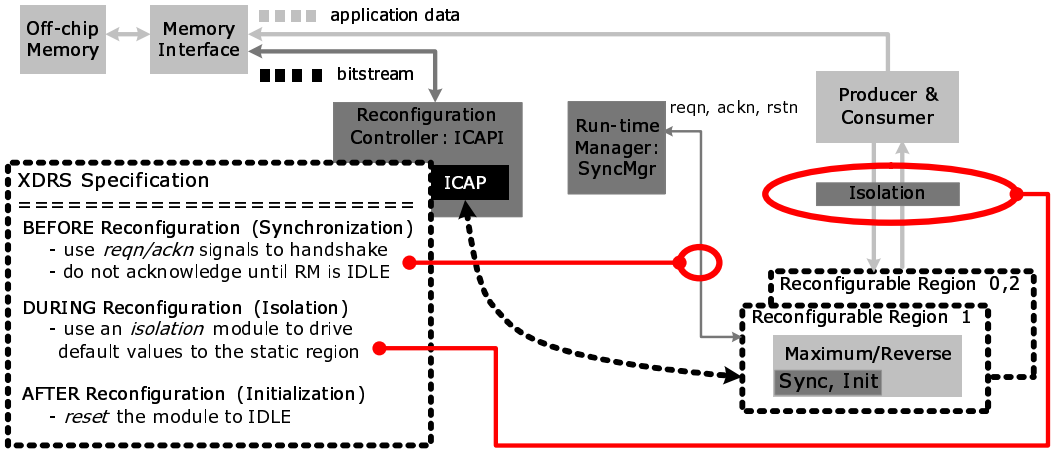


Figure 5.4: The core logic of the XDRS demonstrator

The use of ReSim enables the designer to debug the cycle-accurate behavior of the XDRS system while it is performing partial reconfiguration. Figure 5.5 is a waveform obtained by simulating the core logic of the XDRS system using ModelSim 6.5g. Here, an old `Maximum` module in Reconfigurable Region 1 of the XDRS system is reconfigured to a new `Reverse` module. The figure links the events on the waveform with the block diagram of the core logic of the XDRS system.

- **@t1:** `SyncMgr` starts the reconfiguration and unloads the current module by asserting the request (the `xctrl/req_n` signal). As the `Maximum` module is busy, it blocks the `SyncMgr` and doesn't acknowledge (the `xctrl/ack_n` signal) until a few cycles later. Simulating these handshake signals assists the verification of the synchronization mechanism of the design.
- **@t2:** The first word of the SimB (`0xAA995566`) is transferred from the memory interface (the `mem/data_o` signal) to the ICAP port (the `icap/cdata` signal), thereby marking the start of the "DURING reconfiguration" phase.
- **Triggering module swapping @t3:** After parsing the header section of the SimB, the new `Reverse` module is selected by the `module_selector`. Although such a selection is performed instantaneously, the new module is not connected to

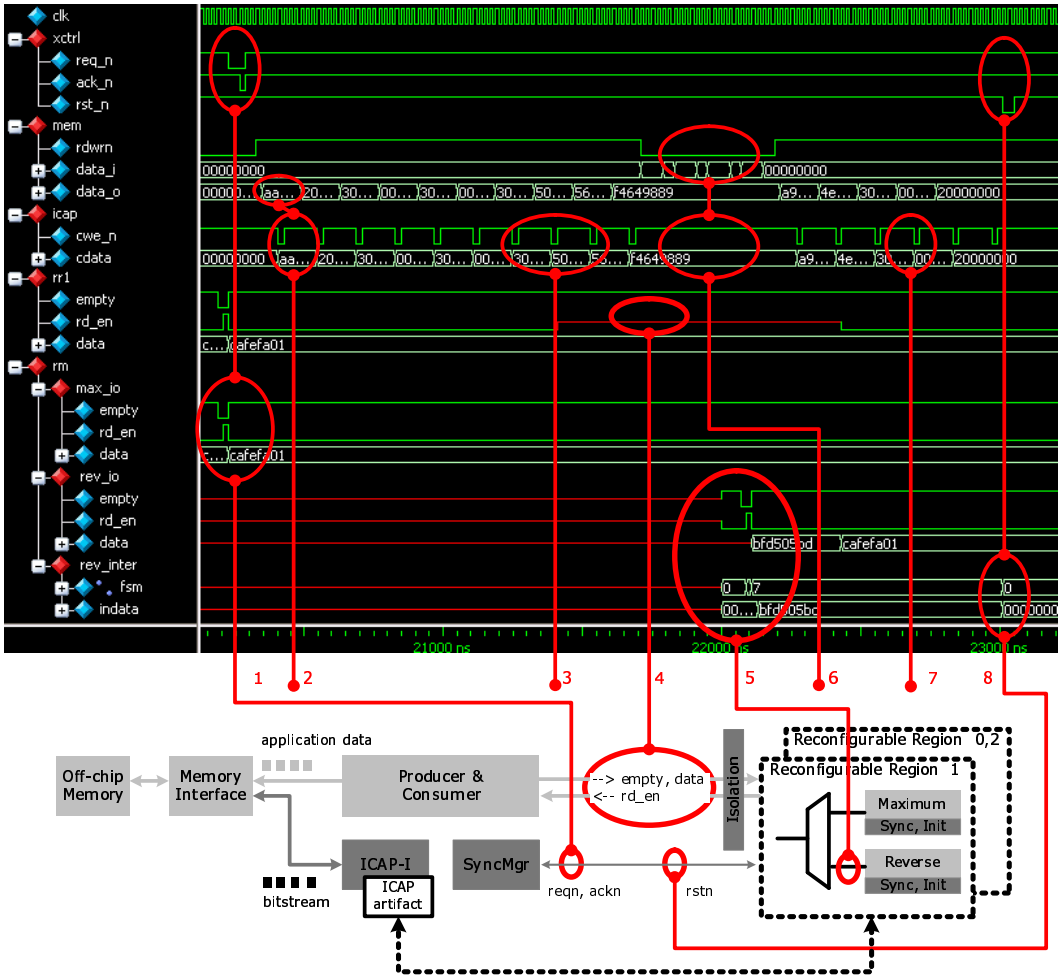


Figure 5.5: Waveform example for partial reconfiguration

the static part until all configuration data of the SimB are written to the ICAP artifact. Thus the delay of such swapping is determined by the bitstream traffic.

- **Mimicking spurious outputs @t4:** When the configuration data section of the SimB is being written to the ICAP, a spurious error source is connected to the static region so as to test the robustness of the isolation module. The error source drives undefined "x" values to all RR outputs (e.g., the `rr1/rd_en` signal). Note that the 3 versions of IO signals (the `rr1`, `rm/max_io` & `rm/rev_io` signal groups) represent signals of the static side, the `Maximum` and the `Reverse` modules respectively.

- **Mimicking undefined initial state @t5:** Apart from the "x" injection to the static side, the Extended Portal also injects a sequence of *user-defined errors* to the `Reverse` module so as to mimic its undefined initial state. In particular, the Extended Portal starts the error injection sequence by de-asserting the `rev_io/empty` signal. The `Reverse` module responds to the `empty` signal by reading random data (`0xbfd505bd` on the `rev_io/data` signal) into its internal register `rev_inter/indata`.

- **Modeling bitstream traffic @t6:** The `Producer/Consumer` module requests memory access. The `Memory-Interface` therefore pauses the reconfiguration until

the requested data are written to memory (see the long delay on the `icap` signal group **@t6**).

- **@t7:** The `DESYNC` command is written to the ICAP thereby ending the "DURING reconfiguration" phase. Until the next `SYNC` word, all subsequent signal transitions on the ICAP interface are ignored.
- **@t8:** The `SyncMgr` module asserts the reset signal (the `xctrl/rst_n` signal) to the RM, which cleans all errors (i.e., the random data in the `rev_inter/indata` register) injected to the `Reverse` module.

By accurately simulating the synchronization, isolation and initialization mechanisms of the XDRS system BEFORE, DURING and AFTER reconfiguration, we detected dozens of cycle-mismatch bugs in our design. For example, the isolation bug, BUG-Example.XDRS.4, was easily identified since ReSim models the spurious outputs of RMs during reconfiguration. Although such "x" injection is similar to [69], ReSim allows cycle-accurate simulation of the transition from DURING to AFTER reconfiguration, which is also essential to detect this type of bug.

BUG-Example.XDRS.4: The `Isolation` module, which disconnects the RM DURING reconfiguration, resumed such connection one cycle too early AFTER reconfiguration. The bug was identified because the undefined "x" values injected by ReSim propagated to the static part in the mismatched cycle.

Coverage Analysis. In order to thoroughly test the core logic of XDRS, we applied coverage analysis to the RTL simulation. According to the specification (see Figure 5.4), we created a test plan, which described the list of scenarios and coverage items that were to be tested (see Figure 5.6 for an extract of the test plan). The test plan includes both code coverage items and functional coverage items. Code coverage items (e.g., item 5.1 `Code_Coverage`) are automatically tracked by the simulator. In particular, the simulator automatically reports which lines were not exercised over all simulation runs.

Functional coverage items are modeled using SystemVerilog coverage groups (e.g., items 7.2 `RM_Transition` & 6.8 `Recon_Req`), coverage directives (e.g., item 5.4 `Cancel_Static`) and assertions (e.g., item 6.6 `Blocked_until_Idle`). For example, the item `Recon_Req` is not considered to be covered until reconfiguration is requested in each legal state of the RM. The corresponding `cvg_recon_req` coverage group is modeled by sampling the FSM state when reconfiguration is requested (@negedge `reqn`), and is deemed covered when the sampled FSM state touches all five possibilities (`Rd1`, `Rd2`, `Wr`, `ReTry` and `IDLE`). As an example of using assertions, the item `Blocked_until_Idle` verifies that the FSM must be in the IDLE state when the reconfiguration is acknowledged. The corresponding `assert_blocked` assertion is checked throughout simulation, and is considered covered if it passes.

The use of ReSim assists the designer in modeling DPR-specific coverage items. For example, the item `RM_Transition` requires all legal transitions between any two RMs to

	XDRS Test Plan	Description	Coverage Item(s)	Type
5	**Isolation**			
5.1 5.4	Code_Coverage Cancel_Static	Code Coverage of isolation Communication attempts from the static region are cancelled	/xdrs/isol_0 cov_cancel_static	Code Directive
6	**Computational_Cor**			
6.6	Blocked_until_Idle	Request of reconfiguration is blocked until RM is IDLE	assert_blocked	Assertion
6.8	Recon_Req	Request of reconfiguration occurs in all RM states	cvg_recon_req	CoverPoint
7	**Partial_Recon**			
7.2	RM_Transition	Transition from each module to each other module	cvg_rm_transition	CoverPoint

```
Examples of functional coverage items
=====================================================
cov_cancel_static : cover property ( ~reconfn && static_start_communication );
assert_blocked : assert property ( (~ackn) |-> (fsm_state == IDLE) );

covergroup cvg_recon_req @(negedge reqn);
   coverpoint fsm_state = {Rd1,Rd2,Wr,ReTry,IDLE};
endgroup
covergroup cvg_rm_transition @(current_module_id);
   coverpoint current_module_id = { [0:1] => [0:1] };
endgroup
```

Figure 5.6: Extract of test plan and selected coverage items

be exercised. The corresponding `cvg_rm_transition` coverage group creates four bins for the 4 possible transitions, and is tracked by sampling the RM IDs extracted from the SimB.

Figure 5.7 illustrates the progress of coverage-driven verification in terms of Lines of Code (LOC) changed, coverage, and bugs detected per day[2] *in the reconfiguration machinery.* Generally speaking, the four plots are related to each other.

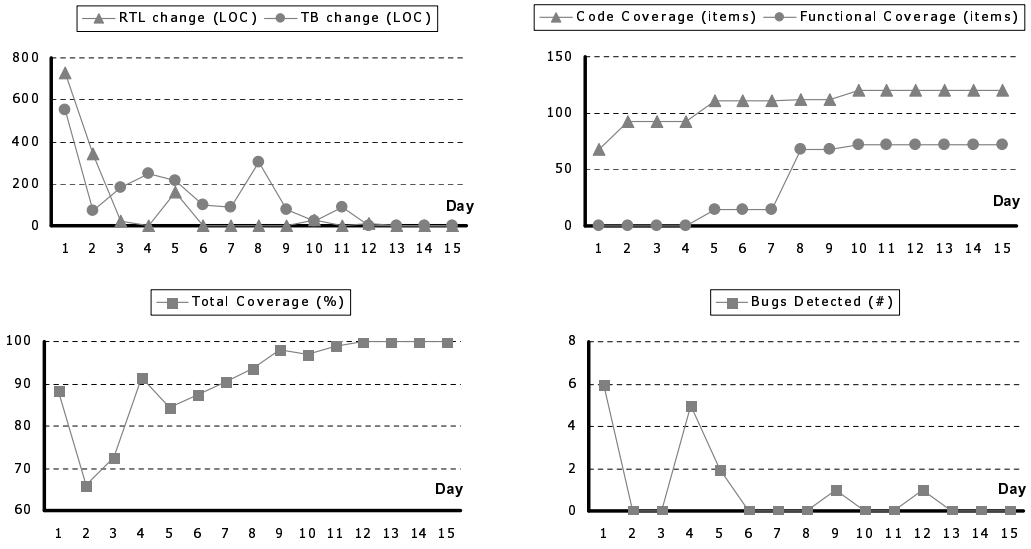


Figure 5.7: Coverage-driven verification progress with the XDRS system

- The changes in the design RTL and the simulation testbench (TB) indicate the development workload attributed to the core logic of XDRS.

[2]One Day = 7 hours full-time work by a designer with 3 years of FPGA design experience

- Since Code Coverage items are automatically generated from the design source, they increased with increases in the amount of RTL design source code. Although Functional Coverage items also increased with design complexity, they were only widely adopted after Day 8 in this project.
- The total coverage dropped as new coverage items were added (e.g., Days 2 & 5) and increased as more tests were created (e.g., Days 4 & 11). However, the trend for the total coverage is not that easy to predict. On Day 8, for example, although the number of coverage items increased significantly, a few more tests were added to cover the newly added coverage items as well as previously uncovered items. Therefore, the coverage increased.
- The bug discovery rate was relatively high at the beginning of the verification (Days 1-5) and decreased as the design began to mature (Days 6-10). We randomized the test stimuli to exercise complex scenarios in the final development phase (Days 11-15), and achieved 100% coverage of the test plan items.

Coverage analysis assisted in detecting 2 corner case bugs in Days 9 and 12. For example, in order to cover the `Recon_Req` item (see Figure 5.6), we randomized the time at which reconfiguration was requested and detected BUG-Example.XDRS.5. This bug was only exposed when reconfiguration was requested in the same cycle that the RM left the IDLE state, and was very likely to be left undetected without analyzing the missed coverage item `Recon_Req`. Furthermore, since the `Blocked_until_Idle` assertion failed in simulation, we were able to quickly localize the source of the failure instead of randomly tracing signals in the waveform. The use of assertions therefore simplified the debugging process.

BUG-Example.XDRS.5: If a reconfiguration request arrived precisely when the RM had just started processing the next input sample, the RM failed to block the reconfiguration request, which violated the `Blocked_until_Idle` item in the test plan. The bug was detected since the `Blocked_until_Idle` assertion failed.

Code coverage was widely used in the early stages of the project and assisted in ensuring a reasonable level of confidence without extra development overhead. Functional coverage items assisted in detecting two critical bugs at the cost of manually modeling the items. In particular, the development workload for modeling functional coverage items involved 100 LOC for modeling coverage groups and directives and 200 LOC for assertions which assisted in debugging the design in addition to indicating coverage. The regression time for running all tests and generating the coverage report was 5 minutes on a Windows XP, Intel 2.53G Dual Core machine.

5.1.1 Targeting a Second Application

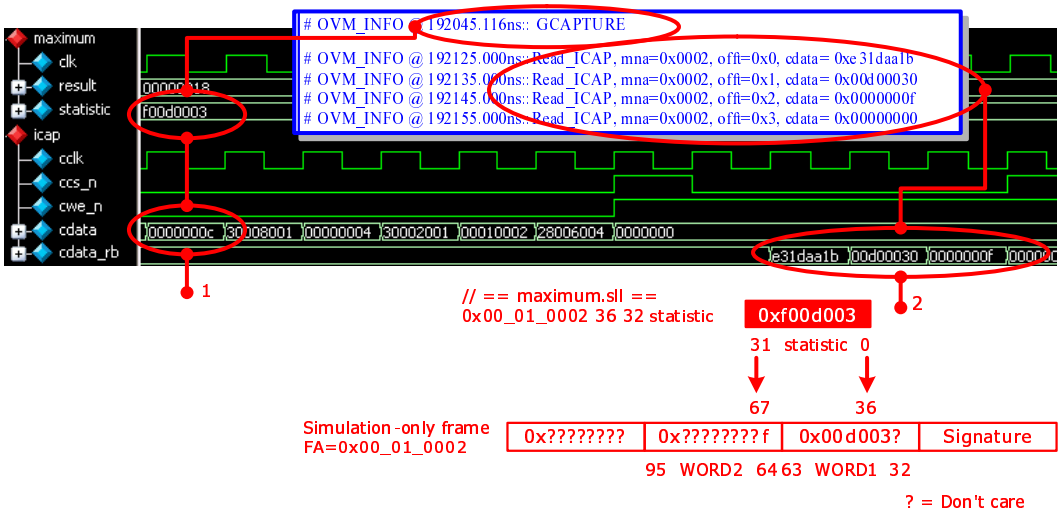
To demonstrate the benefit of applying the platform-based design methodology to the design and verification of the XDRS system, we mapped a second DRS application to XDRS. This second application demonstrated the use of ReSim to verify a DRS design

that saves and restores module state in addition to reconfiguring its logic. The second application periodically reconfigures one slot of the `xps_xdrs` accelerator with either an `Adder` core or a `Maximum` core as two alternative RMs. Apart from computation, each core maintains a `statistic` register, the value of which is copied across configuration periods, and the saving and restoration of the `statistic` register is performed via the ICAP. We updated the embedded software to support state saving and restoration. Since the RTL design of the XDRS platform was already available, we skipped high-level modeling and used RTL simulation to test and debug the new application mapped to the platform.

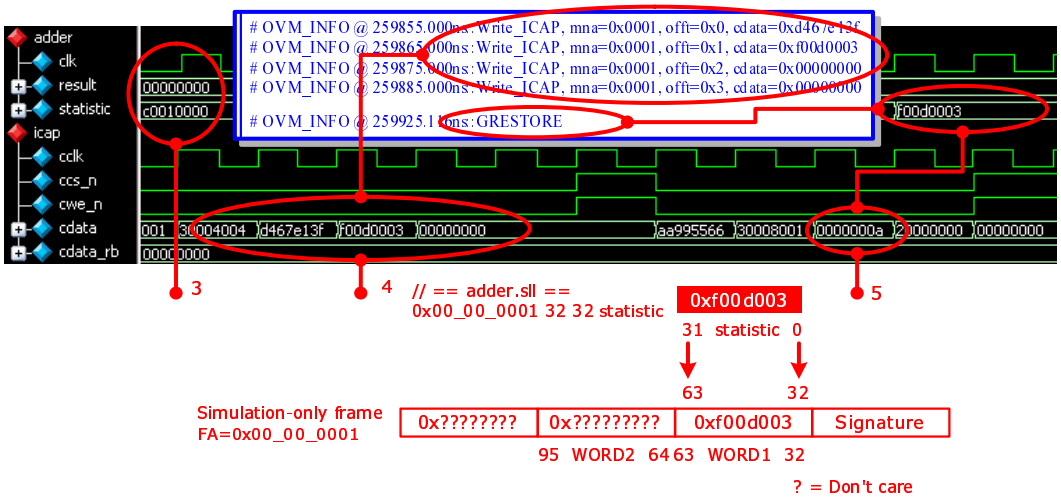
Figure 5.8 shows two waveform segments obtained by simulating the design using ModelSim 6.5g. Here, an old `Maximum` core is reconfigured to a new `Adder` core, and the value of the `statistic` register is copied from the old core to the new core. Figure 5.8-(a) illustrates saving the `maximum/statistic` register by reading back a SimB via the ICAP artifact. The figure contains a screen shot of the simulation waveform and the printout messages (i.e., messages start with `OVM_INFO`). To improve the readability, Figure 5.8-(a) does not include signals of the `Adder` core, which are all injected with undefined "x" values. Similarly, Figure 5.8-(b) illustrates restoring the saved value to the newly configured `Adder` module.

In order to simulate state saving and restoration, the designers need to manually map RTL signals to simulation-only frames by modifying `.sll` files (see Section 4.3). It should be noted that the `statistic` registers of the `maximum` and the `adder` modules are treated as different signals and are mapped to two different configuration frame addresses both on the target FPGA and in the simulation-only layer. In particular, the `maximum/statistic` register starts from bit 36 of frame 0x00010002, and spans 32 bits (see the `.sll` file entry at the bottom of Figure 5.8-(a)), and the `adder/statistic` register is mapped to bit 63-32 of frame 0x00000001 (see Figure 5.8-(b)). The saving and restoration of the `statistic` register proceeds as follow:

- **Modeling bitstream traffic @t1**: A readback SimB (see the `icap/cdata` signal) with a `GCAPTURE` command (i.e., 0x0000000c) is written to the ICAP artifact requesting for the frame containing the `maximum/statistic` register. By the time the `GCAPTURE` command is written to the ICAP artifact, the value of the `maximum/statistic` register is 0xf00d0003. Figure 5.8-(a) illustrates mapping 0xf00d0003 to bits 67-36, which spans two consecutive words (i.e., WORD1 and WORD2) of the simulation-only frame.
- **Modeling bitstream content @t2**: The ICAP artifact is switched to read mode and returns a SimB (see the `icap/cdata_rb` signal) that contains the retrieved state data. In particular, the signature word, WORD1, WORD2 and WORD3 are returned on four consecutive cycles. The waveform only shows part of the readback SimB; please refer to Table 3.4 in Section 3.2.1 for the full SimB.
- **@t3**: After reconfiguration, the `Maximum` core is reconfigured with the `Adder` core and the initial value of the `adder/statistic` register is 0xc0010000.
- **Modeling bitstream traffic @t4**: A restoration SimB containing the saved value (i.e., 0xf00d0003) is written to the ICAP artifact (see the `icap/cdata` signal).



(a) Read back a SimB to save the `maximum/statistic` register



(b) Write a SimB to restore the `adder/statistic` register

Figure 5.8: Waveform example for state saving and restoration

Since the `adder/statistic` register is mapped to bits 63-32, it only spans one word (i.e., WORD1) of the simulation-only frame.

- **Modeling bitstream content @t5**: A `GRESTORE` command (i.e., 0x0000000a) is sent to the ICAP artifact (see the `icap/cdata` signal) and the desired value is restored to the `adder/statistic` register in the following cycle.

We detected 5 software bugs (e.g., BUG-Example.XDRS.6, see page 76) while simulating the periodic application. The simulated design was subsequently tested on an ML507 board with a Virtex-5 FX70T FPGA and we detected one *fabric-dependent* bug (i.e., BUG-Example.XDRS.7). Since ReSim failed to mimic the exact behavior of the target device, a limitation discussed in Section 3.3, BUG-Example.XDRS.7 was missed by ReSim-based simulation. By inserting probing logic using ChipScope [95], we were able to identify this bug on the target FPGA. However, as the probing logic was only able to visualize a limited number of signals for a limited period of time, we used 5 iterations to trace the cause of one bug. Each iteration involved inserting new probing logic and

re-implementing the design, and took 59 minutes to complete on the same machine used for RTL simulation. Debugging the design using ChipScope was therefore found to be quite time consuming.

BUG-Example.XDRS.6: The restoration routine of the software driver uses a pointer as an argument and the pointer is expected to point to the logic allocation information of the signal to be restored. However, the application software program passed an incorrect pointer to the restoration routine. The bug was detected as a consequence of incorrect values being restored to the simulated `statistic` register.

BUG-Example.XDRS.7: The number of pad words returned from ICAP was not the same as the designer had expected. The software attempted to extract state bits from the wrong bit positions and the extracted data were therefore incorrect.

Although the first streaming application did not save and restore module state, the state saving and restoration capability of XDRS was thoroughly verified via the coverage-driven verification process of the platform. The verification effort for this second, periodic application was therefore reduced due to the platform-based design methodology.

5.2 Case Study II: In-house Fault-Tolerant Application

The second case study involves the design of signal processing circuits to be used in space-based applications (e.g., Synthetic Aperture Radar, SAR) [16]. To improve the tolerance of radiation-induced errors such as Single Event Upsets (SEU), Triple Modular Redundancy (TMR) has been used to protect the signal processing circuits and a module suffering from permanent SEUs is dynamically reconfigured with a correct copy. DPR has been used in such fault-tolerant applications such as [63, 35, 16] and this case study aims to apply ReSim and coverage analysis to verifying the DPR activities in fault-tolerant applications. We also compared the results of coverage-driven verification with ad-hoc on-chip debugging.

Comparing the system architecture between this case study and the XDRS platform (see Section 5.1), one significant difference is that the DUT of this case study does not contain microprocessors, buses or software. As illustrated by Figure 5.9, the DUT is composed of two computing nodes (i.e., nodes 0 & 1) and a reconfiguration controller node (RC-node, i.e., node 2). Each computing node contains three identical copies of specific signal processing circuits, either Finite Impulse Response (FIR) filters or Block Adaptive Quantizers (BAQ), and are checked by a voter [16]. The RC-node reconfigures a module that is suffering from a permanent SEU error by transferring a bitstream from off-chip flash memory to the ICAP port. The computing nodes and the RC-node are connected via a self-timed ring network, so that nodes can execute at independent clock frequencies. Nodes exchange information (e.g., requests and acknowledgments of reconfiguration) by sending and receiving messages using Network Interfaces (NI).

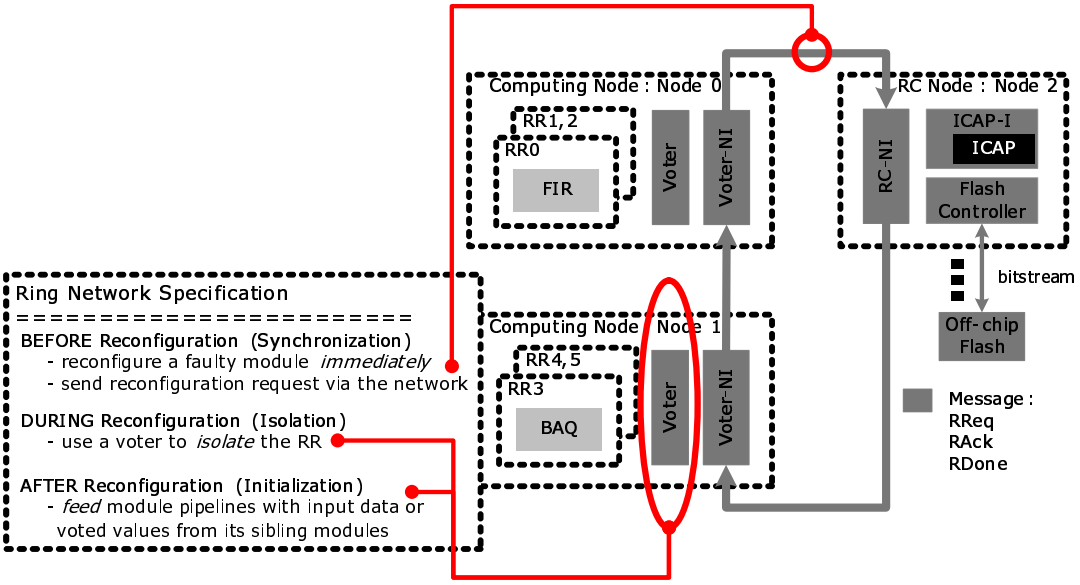


Figure 5.9: The hardware architecture of the fault-tolerant DRS

This case study does not aim to explain in detail the robustness of this design with respect to SEUs since our primary interest is in the functional verification of the reconfiguration machinery (moderately shaded parts of Figure 5.9). In particular, if a voter detects that one copy of the FIR filters or BAQs is permanently affected by an SEU, the voter initiates a reconfiguration request message (i.e., RReq) over the network. Upon receiving the request, the RC-node replies with an acknowledgment message (i.e., RAck) indicating the start of reconfiguration and sends a done message (i.e., RDone) at the end of bitstream transfer. DURING reconfiguration, the module undergoing reconfiguration may produce spurious outputs, which are ignored by the voter. Thus, the voter works as an isolation module during reconfiguration and is therefore considered to be a component of the reconfiguration machinery. After reconfiguration, the execution state of the newly reconfigured module is recovered by feeding the pipelines of the newly configured module with correct values either from the input data stream or from the checked feedback of its sibling modules [16].

Figure 5.10 illustrates part of the test plan for the voter, which describes the scenarios that need to be covered to test the voter. For example, as described above, the voter initiates reconfiguration requests and isolates the RM undergoing reconfiguration. These two operations are specified by coverage items 4.7 and 4.8 respectively in the test plan. The two items are further linked to assertions and coverage directives in the design source code.

Figure 5.11 illustrates the progress of coverage-driven verification in terms of Lines of Code (LOC) changed, coverage, and bugs detected per day *in the reconfiguration machinery*. In particular, the figure does not take into account the statistics from the application logic (i.e., the FIR and the BAQ modules) or from proven IP (i.e., the flash controller IP provided by the vendor and the ICAP-I IP verified in the first case study, see Section 5.1). Similar to the coverage-driven verification of the XDRS core logic (see Section 5.1), the total coverage of the fault-tolerant DRS also dropped as new design

	Title	Description	Link	Type
4	**Voter**			
4.1	Code_Coverage	Code coverage of the voter	.../node_0/voter_0 .../node_1/voter_0	Code
4.2	Majority_Voter_Op	The Majority Voter does not detect error; detect one error; detect multiple errors;	assert_mv_no_error assert_mv_one_error_* assert_mv_more_error	Assertion
4.3	Voter_PermErr	The Voter detects permanent error and starts recon.	assert_vtr_errcnt_*_sat	Assertion
4.4	Voter_TranErr	The Voter detects transient error and does not start recon.	cov_vtr_errcnt_*_non_sat	Directive
4.6	Voter_OngoingErr	The Voter detects ongoing error during recon.	assert_vtr_oge_*	Assertion
4.7	**Voter_Isolate_Err**	**The Voter isolates errors from module undergoing recon.**	**assert_isol_at_recon assert_isol_at_feeding**	**Assertion**
4.8	**Voter_NI_Recon**	**The Voter starts recon. and ends by (1) Ack-Done (2) ...**	**cov_ni_recon_done ...**	**Directive**

Figure 5.10: Extract of the test plan section for the voter

features were added (e.g., Days 2-4 and Day 8) and increased as more tests were created (e.g., Days 5-6 and Days 9-12).

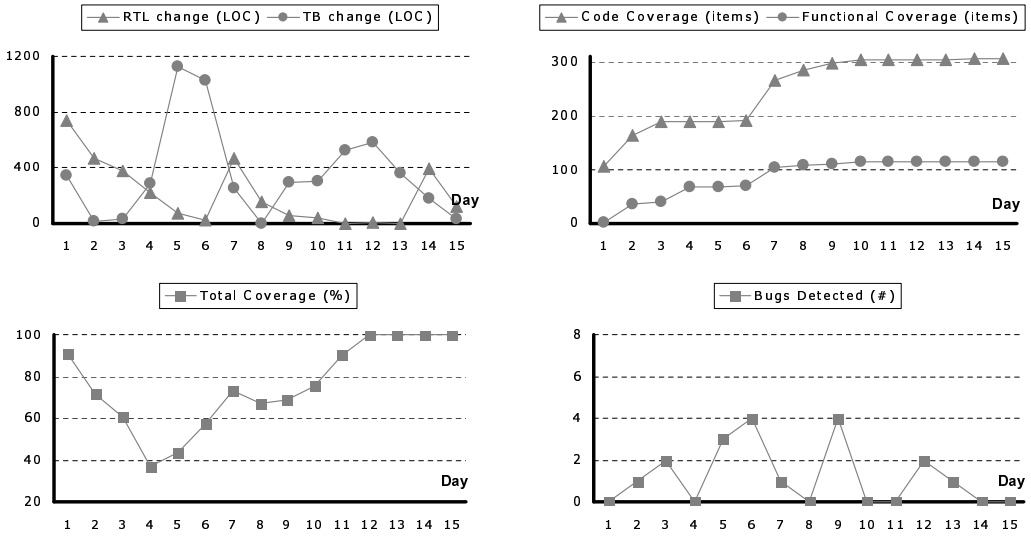


Figure 5.11: Simulation-based verification of the fault-tolerant DRS

- In order to compare simulation with on-chip debugging, we deliberately separated the development into two streams. At the beginning of the verification (Days 1-6), designer A worked on the two computing nodes (i.e., nodes 0 & 1) using simulation-based verification while designer B designed and tested the RC-node (i.e., node 2) using on-chip debugging. Since on-chip debugging does not collect coverage data, Days 1-6 of Figure 5.11 only track the progress of simulation-based verification, which detected 10 bugs during the period. On the other hand, using on-chip debugging, designer B identified 4 bugs (not shown in the figure) in the RC-node.

- On Day 7, the FPGA-proven RC-node was integrated with the computing nodes. From designer A's perspective, the RC-node was viewed as an IP, and 1 bug was captured in integrating the IP with the rest of the design. Designer A continued

testing the integrated design using simulation, and created tests to cover scenarios that had not yet been tested in the integrated system. In particular, ReSim-based simulation was applied starting from Day 9, when the design was stable and mature enough.

- The coverage reached 100% on Day 13[3], after which the designer started optimizing the design. For each change made, regression tests were applied to the design and we detected 1 more bug during the optimization stage.

On-Chip Debugging. Comparing the on-chip debugging stream with the simulation stream, the progress of both designers was found to be similar. Using ChipScope [95], designer B tested the RC-node on the target FPGA without using any simulation techniques. Since the reconfiguration controller (i.e., the `ICAP-I` module) was reused from the first case study (see Figure 5.4 in Section 5.1) and the flash controller is an IP provided by the vendor, the workload of designer B involved integrating the two IPs and creating the network interface (i.e., the `RC-NI` module). In the same period, designer A designed and tested the two computing nodes using simulation. Since the signal processing circuits had already been created and verified beforehand, the workload of designer A involved creating a voter and a network interface (i.e., the `Voter-NI` module).

The simulation testbench created by designer A was substantially reused later in the project. In particular, using the OVM framework [27], designer A created a library of stimuli patterns (e.g., transient or permanent SEU sequences), trackable coverage items and assertions. This library was later extended to exercise corner cases and scenarios of the reconfiguration process. However, designer B spent the majority of time tracing errors using ChipScope. For example, designer B accidentally introduced BUG-Example.FT.1 and thought it was a bug in the logic, it turned out to be a typo in the clocking circuitry, and it took 5 hours to trace the bug. Such a bug would have been identified very quickly in simulation.

BUG-Example.FT.1: The designer accidentally created a level-sensitive clock instead of an edge-sensitive one, as desired.

ReSim-based Simulation. Since testing individual modules such as the network interface and the voter do not require modeling any characteristic features of DPR, designer A used traditional simulation techniques to test the computing nodes in Days 1-6 and the integrated design in Days 7-8. We applied ReSim-based simulation to test the reconfiguration process of the integrated design from Day 9 and detected 7 more bugs. By analyzing the causes of the bugs, we determined that 4 out of the 7 bugs could have been detected by prior simulation methods such as Virtual Multiplexing [54] or DCS [69]. In particular, since the RC-node was already FPGA-proven, and bitstreams are transferred over a dedicated datapath, reconfiguration could be modeled with a constant delay. However, 3 bugs could only have been identified using ReSim, and we describe two of these bugs (i.e., BUG-Example.FT.2, BUG-Example.FT.3) in detail.

[3]The coverage at Day 12 was 99.95%

BUG-Example.FT.2: After reconfiguration, the system failed to feed enough data to flush the internal pipeline of the FIR filter. Thus the undefined initial values of the pipeline were not properly cleared. This bug was exposed since the undefined initial values injected to the pipeline registers of the simulated FIR filter propagated to the static region after reconfiguration, and the undefined values were detected by an `assert_isolate_at_feeding` assertion as described by item 4.7 in the test plan (see Figure 5.10).

BUG-Example.FT.3: Typically, reconfiguration is slower than transferring a message between two nodes. The expected operation of the RC-node is therefore: start reconfiguration; transfer an RAck message to a computing node; end reconfiguration; transfer an RDone message to a computing node. However, when the bitstream is very small and when the computing node is executing with a very slow clock, reconfiguration can finish before the RAck message is transferred. Under such a circumstance, the RC-node did not correctly send the RDone message.

Coverage analysis assisted in detecting a number of corner case bugs. For example, we randomized the SimB size so as to exercise the design with various SimBs and randomized the operating frequencies of nodes so as to cover various clock frequency-related coverage items. We were able to identify a corner case bug in the RC-node (i.e., BUG-Example.FT.3), which had already been tested on the target FPGA. The bug is only exposed when the system reconfigures a very small RM whose bitstream is short. Unfortunately, such a scenario was not tested on the FPGA, and it is difficult to test since the size of a real bitstream cannot easily be adjusted for test purposes. Therefore, although on-chip debugging represents real world conditions, it can sometimes only validate a limited number of the possible execution scenarios for a design. Previous simulation methods (e.g., [74]) tend to annotate the reconfiguration delay according to the size of a real bitstream, which may also limit the possible scenarios that can be exercised using them. In order to create a robust design, it is highly desirable to identify and exercise all corner cases of the system. Since a SimB is not dependent on the FPGA fabric, the size of a SimB can easily be adjusted for test purposes. ReSim-based simulation is therefore better able to exercise some of the corner cases of a design.

In this case study, we used both code coverage and functional coverage throughout the verification process. Furthermore, we used constrained random stimuli generation techniques to assist in covering corner cases of the design. As illustrated by Figure 5.11, the last version of the design contains 304 code coverage items and 114 functional coverage items. The regression time for running all tests and generating the coverage report was 8 minutes on a Windows XP, Intel 2.53G Dual Core machine.

5.3 Case Study III: Third-Party Video-Processing Application

The third case study involved the design and verification of a cutting-edge, dynamically reconfigurable driver assistance demonstrator from the AutoVision project [18]. A more

recent design of the project uses an Optical Flow algorithm to determine the speed and distance of moving objects (e.g., cars) on the road so as to identify potentially dangerous driving conditions [4]. As illustrated in Figure 5.12, the demonstrator uses two video processing engines to accelerate the Optical Flow algorithm. In particular, each input video frame is first processed by a Census Image Engine (CIE) to generate a feature image. The CIE engine is then reconfigured with a Matching Engine (ME) to compare two consecutive feature images and compute the motion vectors. Finally, the embedded software running on an on-chip PowerPC processor outputs the video frames that are annotated with motion vectors.

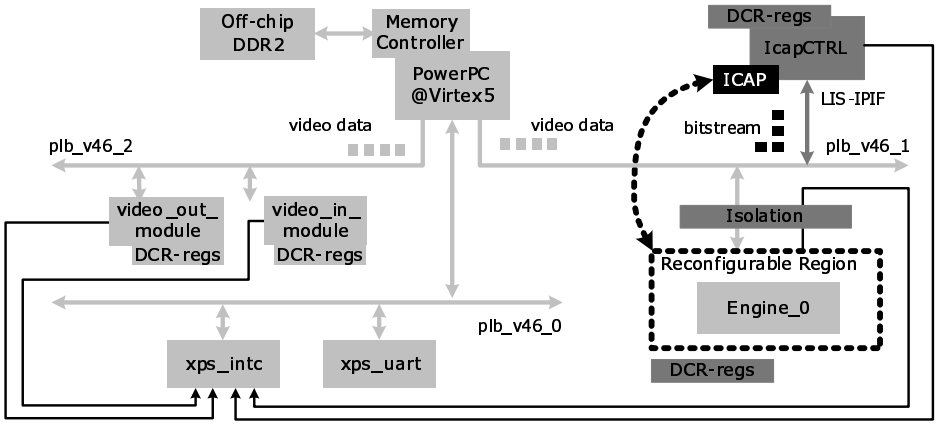


Figure 5.12: The hardware architecture of the Optical Flow Demonstrator

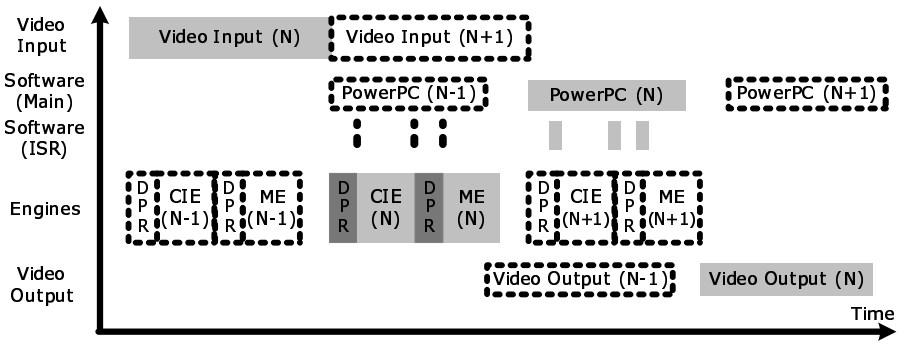


Figure 5.13: The processing flow of the Optical Flow Demonstrator

Instead of building the design from scratch, as was done for the first and the second case studies, this case study aimed to study the application of an IP-reuse methodology to the design and verification of DRS designs. In particular, both the hardware and the software of the Optical Flow Demonstrator were slightly modified from the original design, but could be viewed as a re-integration of the original design modules. Compared with the original design, the re-integrated design (see Figure 5.12) connects the `IcapCTRL` controller to the PLB bus instead of the NPI interface of the memory controller. The processing flow of the re-integrated design was pipelined to better exploit parallelism in the system (see Figure 5.13). In particular, the PowerPC processor draws motion vectors for the previous frame while the engines are processing the current frame. The start, end and reconfiguration of the video processing engines are controlled by Interrupt Service Routines (ISR) that are independent of the main software functions.

The primary focus of this case study was the verification of the reconfiguration machinery (moderately shaded parts of Figure 5.12). In particular, the DUT instantiates a reconfiguration controller (`IcapCTRL`) that transfers bitstreams from off-chip memory to the ICAP port. To avoid the propagation of erroneous signals from the region undergoing reconfiguration, an `Isolation` module was used to isolate the engines during reconfiguration. Furthermore, the DCR registers were moved from inside the engines to the outside so as to avoid breaking the DCR daisy chain during reconfiguration. Since the DUT is an integration of various proven IP blocks from the original design, we focused on the verification of system *integration*. In particular, we aimed to verify that the modified reconfiguration machinery (i.e., the PLB-based `IcapCTRL`) and its software driver (i.e., the interrupt-driven reconfiguration of the pipelined processing flow) were correct and were correctly *integrated* with the rest of the system hardware and software.

Since individual hardware and software building blocks were already FPGA-proven, we didn't create high-level models of the system, but rather used RTL simulation directly. Figure 5.14 illustrates the progress of development in terms of Lines of Code (LOC) changed and bugs detected per week.

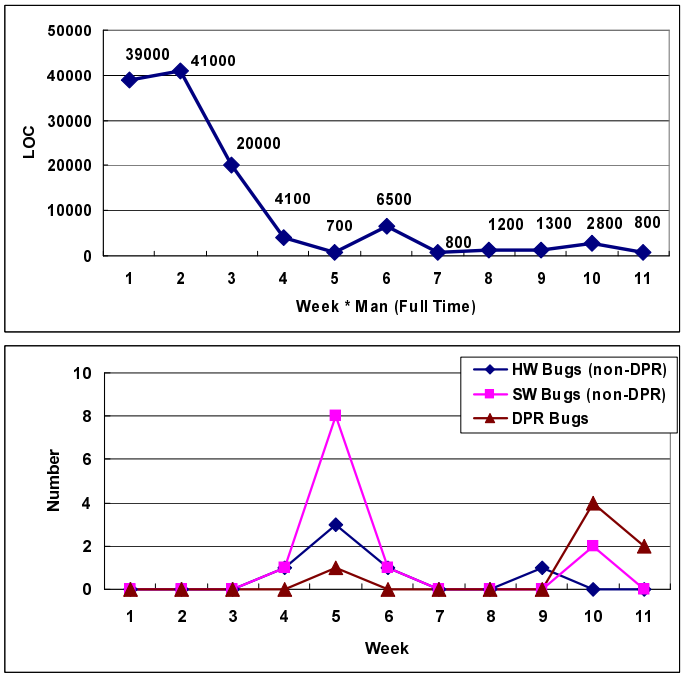


Figure 5.14: Development progress of the Optical Flow Demonstrator

- By the end of Week 3, the designer finished assembling the modified Optical Flow Demonstrator (see Figures 5.12 and 5.13) as well as an initial testbench. Since design files of the Optical Flow Demonstrator were initially added to version control, the reported LOC number is very high. However, most design files were reused from previous projects, and the designer's workload involved re-integrating legacy components and simulating sanity checks such as a "hello world" program and a "camera to VGA display" application.

- The real verification work began in Week 4. In order to compare ReSim-based simulation with Virtual Multiplexing, we deliberately used Virtual Multiplexing

to simulate the system at the beginning of the case study. While only being able to identify 1 DPR-related bug, Virtual Multiplexing was helpful in detecting most of the bugs in the static design. In particular, between Week 6 and Week 9, the designer fixed 3 extremely costly bugs in the static region and Virtual Multiplexing-based simulation passed. Apart from bug fixes, the design itself was stable during this period. The LOC numbers were contributed to by changes to the testbench aimed at improving the simulation throughput and at reducing debug turnaround time.

- In the last 2 weeks, the designer used ReSim to simulate DPR and detected 2 software bugs and 6 DPR bugs in the system. The simulation passed at Week 11, after which no more bugs were detected.

Virtual Multiplexing. Using the virtual multiplexing method, as reported in [54], we inserted a multiplexer to switch the currently active engine with the selection being performed by writing to a software-accessible `engine_signature` register. Therefore, both the hardware and software of the system had to be modified during simulation to insert and use the multiplexer. Virtual Multiplexing was able to detect 1 bug in the reconfiguration machinery in Week 5.

Although Virtual Multiplexing was able to mimic the intra-frame reconfiguration of the DUT, in simulation, DPR was triggered by software instead of by transferring a bitstream to the configuration port. Therefore, the software and hardware tested in simulation did not match what was actually implemented, and we even detected a *false positive* bug (i.e., BUG-Example.AUTO.1) in the `engine_signature` register in Week 4[4]. Furthermore, since the `IcapCTRL` module was instantiated in the design but was not used in simulation, Virtual Multiplexing was unable to detect bugs in the bitstream transfer datapath (e.g., BUG-Example.AUTO.2, see page 84). Last but not least, since multiplexing the simulated engines did not generate erroneous signals, as implemented designs might do, the isolation mechanism (i.e., the `Isolation` module) was not tested in simulation. Using Virtual Multiplexing to simulate DPR therefore only provided limited assistance in debugging DPR of the AutoVision system.

BUG-Example.AUTO.1: The CIE/ME was not reset correctly. This was because the `engine_signature` register was not correctly initialized and no engine was selected to be active. Since the `engine_signature` register only exists in Virtual Multiplexing-based simulation, this bug was a *false alarm*. Since ReSim does not change the user design, this bug would NOT have been introduced with ReSim-based simulation.

ReSim-based Simulation. After the static aspects of the design had matured, we used ReSim-based simulation to test the reconfiguration machinery of the system (Weeks 10-11). Compared with Virtual-Multiplexing, ReSim-based simulation more accurately models partial reconfiguration and more thoroughly tests the Optical Flow Demonstrator. In particular,

[4]Since BUG-Example.AUTO.1 was a false positive DPR bug, it is not counted in Figure 5.14

- Since ReSim uses a SimB to replace real bitstreams, the bitstream transfer datapath (e.g., the RTL code of the `IcapCTRL`) was verified in simulation. The reconfiguration delay was determined by bitstream transfer time instead of being zero or a constant. A bug in the bitstream transfer datapath would have prevented the new engine from being swapped in simulation.
- Although ReSim connected all engines in parallel, like Virtual Multiplexing, the selection of engines was triggered by the SimB. Therefore, ReSim did not use the indirect mechanism of an `engine_signature` register, and the software driver that controlled the reconfiguration process did not have to be changed for simulation purposes.
- Since errors were injected into the static region when the SimB was being written to the ICAP, the isolation logic and the software driver that controlled such logic was verified in simulation. If, for example, the designer failed to move the DCR registers out of the engines, the DCR daisy chain would break as a consequence of the injected errors propagating to the DCR bus of the static region.

ReSim-based simulation assisted in detecting 6 DPR-related bugs in the system (e.g., BUG-Example.AUTO.2, BUG-Example.AUTO.3 and BUG-Example.AUTO.4).

BUG-Example.AUTO.2: The `IcapCTRL` module was used in point-to-point mode in the original system, and it failed to work with the shared PLB bus in the modified Optical Flow Demonstrator. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation. This bug was introduced by changing the way the IP was integrated.

BUG-Example.AUTO.3: After changing a parameter of the `IcapCTRL` module, the software driver was not updated accordingly and the SimB was not successfully transferred. This bug was detected when a new engine was not swapped in due to the bug in the bitstream transfer process. The bug was introduced by a mismatch between hardware and software.

BUG-Example.AUTO.4: The system software failed to wait until the completion of bitstream transfer before resetting the engines. This bug was introduced when the design was modified to use a different clocking scheme that slowed down the bitstream transfer and the software was not updated to slow down the reset operations accordingly. Since ReSim more accurately modeled the timing of reconfiguration events, this bug could ONLY be detected by ReSim-based simulation.

Although individual engines and their software drivers were already FPGA-proven from the original design, bugs were introduced through mismatches between module parameters (e.g., BUG-Example.AUTO.2) and software/hardware parameters (e.g., BUG-Example.AUTO.3). Apart from detecting bugs introduced by modifying the original design, we were able to identify 3 potential bugs in parts of the system that were the same as the original one. For example, BUG-Example.AUTO.4 was not exposed before because the original design used a faster configuration clock. This bug was identified because ReSim did not activate the newly configured module until all words of the SimB

were successfully written to the ICAP. Therefore, the use of SimBs more accurately modeled the timing associated with partial reconfiguration, and ReSim-based simulation was effective in testing the integrated Optical Flow Demonstrator.

We ran the simulations using ModelSim 6.5g on a Windows XP, Intel 2.53GHz Dual Core machine. Figure 5.15 is a screen shot of one frame of a sample video. The simulated design identifies the moving car and draws motion vectors accordingly. The movement of the matched pixels is represented by various colors. In particular, red and yellow dots indicate slow moving and distant objects whereas green and blue dots indicate fast moving and close objects.

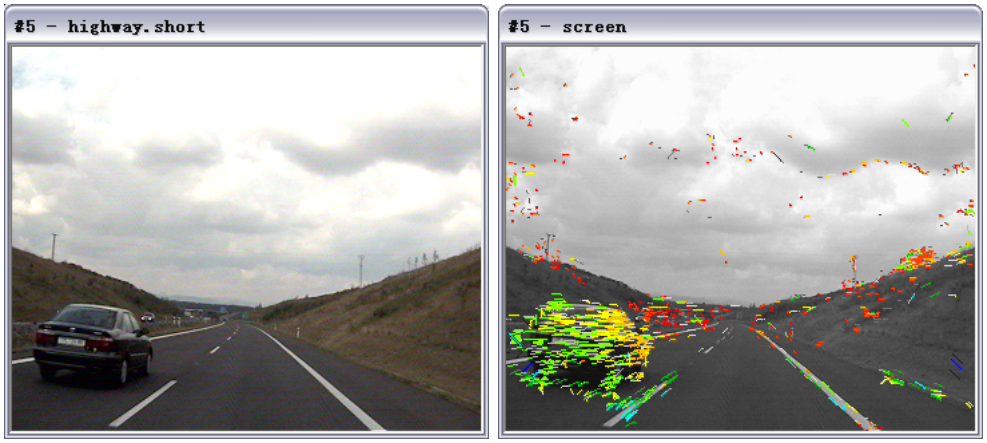


Figure 5.15: One frame of the input and the processed video (sensitivity parameter ε =20)

Table 5.1: Time to simulate one video frame

	Simulated Time (ms)	Elapsed Time (min)
Census Image Engine	1.1	6
Matching Engine	1.4	4.5
PowerPC Interrupt Handler	0.5	0.5
Dynamic Partial Reconfiguration	< 0.1	negligible
Overall	3.0	11

The simulation performance of the testbench is summarized in Table 5.1. In the pipelined processing flow, the video engines are the bottleneck of the system throughput (see Figure 5.13). The time required to process one frame is determined by the video engines and is therefore the sum of the execution time of CIE (1.1 ms), ME (1.4 ms), 2 DPR intervals (<0.1 ms) and 3 ISR intervals (0.5 ms). Overall, it took 11 minutes to simulate the processing of one frame. The Elapsed Time of each execution stage increases with the Simulated Time and also increases if the simulated design has more signal activity. For example, since the CIE engine has more signal toggling activity, the Elapsed Time to simulate 1.1 ms of CIE operations (6 min) is longer than the time to simulate 1.4 ms of ME operations (4.5 min). Since all bugs identified in this study were detected within the first 2-4 frames, the debug turnaround time for simulation was therefore at most 44

minutes per iteration. It should be noted that since the length of a SimB (4K words) was significantly smaller than that of the real bitstream (129K words), the Elapsed Time for simulating DPR could be ignored.

The original AutoVision project tested and debugged the *integrated* Optical Flow Demonstrator using ChipScope [4]. For this case study and our host machine, the implementation and bitstream generation iteration took 52 minutes, and, as described in Section 2.1, the debug turnaround time for on-chip debugging would therefore have been at least that much time per iteration, which is longer than the longest debug turnaround time for simulation (i.e., 44 minutes as described above). Furthermore, it could be anticipated that complex bugs such as BUG-Example.AUTO.4 would require several iterations to trace using ChipScope, and would therefore have been extremely time consuming.

5.4 Case Study IV & V: Vendor Reference Designs

These case studies demonstrate the use of ReSim to verify two vendor reference designs. In the first case study, a fast PCIe configuration reference design applies partial reconfiguration to meet the tight PCIe startup timing requirement [83]. In the second case study, a processor dynamically reconfigures its peripheral modules for various math tasks [93]. Both reference designs target an ML605 board with a Virtex 6 LX240T FPGA.

Since the reference designs had already been proven, it is not surprising that we were not able to detect bugs using ReSim-based simulation. However, simulating proven designs from the vendor demonstrates the robustness and the flexibility of ReSim. Furthermore, we use these reference designs to assess the *additional* workload required to apply ReSim to designs that were not initially prepared for ReSim-based simulation.

5.4.1 Case Study IV: Fast PCIe Reference Design

Figure 5.16 illustrates the block diagram of the Fast PCIe configuration (FPCIe) reference design [83]. At startup, the reference design loads a light-weight PCIe endpoint logic block within the required time. The rest of the FPGA is then dynamically reconfigured with the core application logic (i.e., the Bus Master DMA module) via the established PCIe link. To further save resource utilization of the design, we used a 64-entry FIFO to buffer the bitstream instead of a 1024-entry one. We compare the original simulation testbench with ReSim-based simulation. In particular:

- The original testbench provided with the design also used a dummy bitstream to verify the bitstream datapath. However, the contents of this dummy bitstream were meaningless and were therefore discarded, after being written to the ICAP, without triggering the swapping of the core application logic. As a result, the simulated application logic operated whether or not the dummy bitstream was loaded correctly, and potential bugs on the bitstream datapath could not be detected. ReSim modeled the triggering condition more accurately by swapping in the Bus Master DMA module according to the SimB.

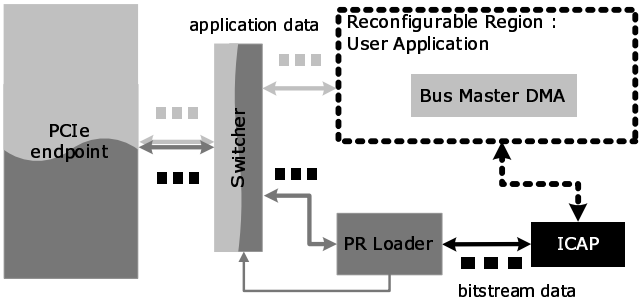


Figure 5.16: Fast PCIe configuration reference design, after [83]

- Moreover, ReSim supported the error injection required to test the isolation and initialization mechanisms of the design. The `Switcher` isolated the RR from the static part, so that DURING reconfiguration, the spurious outputs of the application logic did not affect the PCIe endpoint. The robustness of the `Switcher` as an isolation module was tested against the errors injected to it. Furthermore, ReSim injected errors to the core application logic so as to mimic its undefined initial state and to check whether the `PR Loader` correctly reset the core application logic AFTER reconfiguration.

Figure 5.17 illustrates a screenshot of ReSim-based simulation.

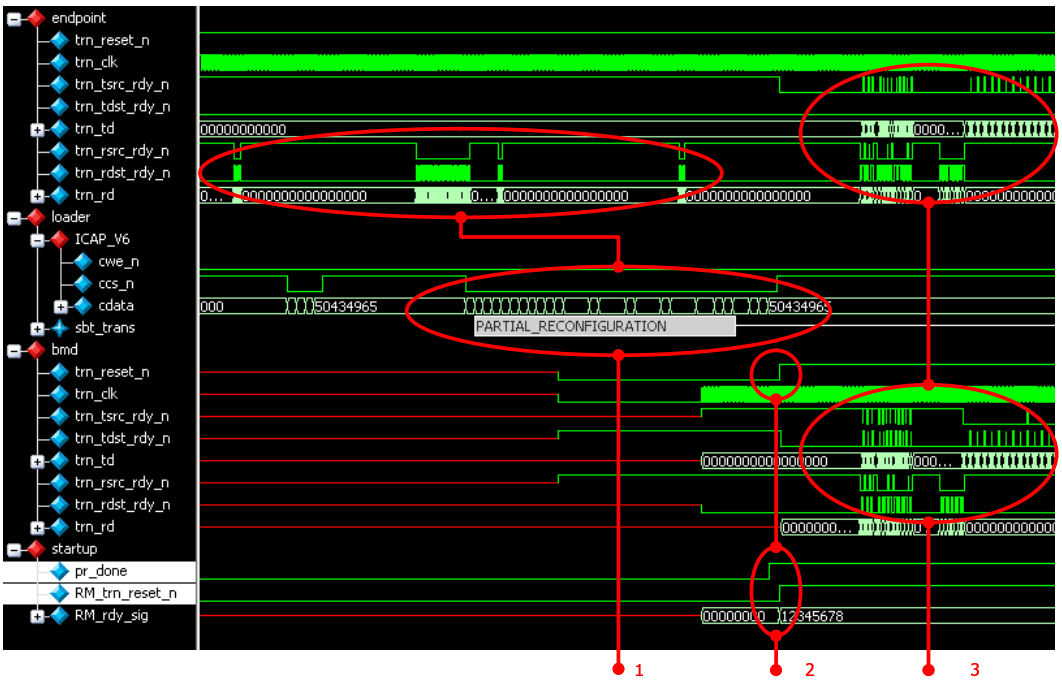


Figure 5.17: Simulating the FPCIe reference design

@t1: The testbench sends a SimB over the PCIe link. When the SimB is being transferred, errors are injected to the Bus Master DMA module (see the `bmd` signal group).

@t2: At the end of bitstream transfer, the Bus Master DMA module is swapped in. At the same time, the `pr_loader` module enables the RM.

@t3: After reconfiguration, the Bus Master DMA module starts operating. Subsequent PCIe transactions are thus routed to the application logic instead of to the `pr_loader` module.

Apart from performing cycle-accurate simulation on the reference designs, we tuned the SimB to analyze the coverage of the Fast PCIe reference design. In particular, we adjusted the size of the SimB to exercise various scenarios of the bitstream datapath (see Table 5.2).

Table 5.2: The effect of SimB size on verification coverage

Test ID	SimB Size (B)	Individual Coverage (%)	Accumulated Coverage (%)	CPU Time (s)
1	32	56.7	56.7	253
2	48	68.62	68.85	251
3	80	69.01	69.64	257
4	144	69.01	69.64	264
5	272	69.01	69.64	310

To exercise the FIFO_FULL scenario, we increased the SimB size from 48 to 80 bytes, which is beyond the 64-byte FIFO capacity, and the coverage increased accordingly. The table also assists in selecting an optimal test set for regression. Considering Test 3, the individual coverage using 80 bytes of SimB was less than the accumulated coverage for Tests 1, 2 and 3. This increase in coverage was contributed by the shorter tests, which exercised scenarios that were missed by the longer test. Therefore, Tests 1, 2 and 3 form an optimal set of "golden" tests which contribute the highest accumulated coverage with the least simulation time.

The development workload for integrating ReSim with the released testbench [83] included 150 LOC (Tcl) describing parameters for generating the artifacts and 10 LOC (Verilog) to instantiate the generated artifacts. We also changed 110 LOC (Verilog) in the original testbench to send SimBs instead of the original dummy bitstream to the FPCIe design. This case study only used code coverage and there was no extra workload for applying coverage-driven verification. Compared to the complexity of the design (3500 LOC excluding code generated by CoreGen), the overheads of using ReSim and performing code coverage analysis were trivial.

5.4.2 Case Study V: Reconfigurable Peripheral Reference Design

Figure 5.18 illustrates the block diagram of the Reconfigurable Peripheral reference design [93]. The design instantiates the bus-based `xps_hwicap` IP core as a reconfiguration controller in order to swap peripheral modules (e.g., the `xps_math` module) attached to the bus. By modeling the bitstream traffic, ReSim-based simulation tests the hardware logic and the software driver of the `xps_hwicap` IP core, and verifies that the core is correctly integrated with the rest of the system.

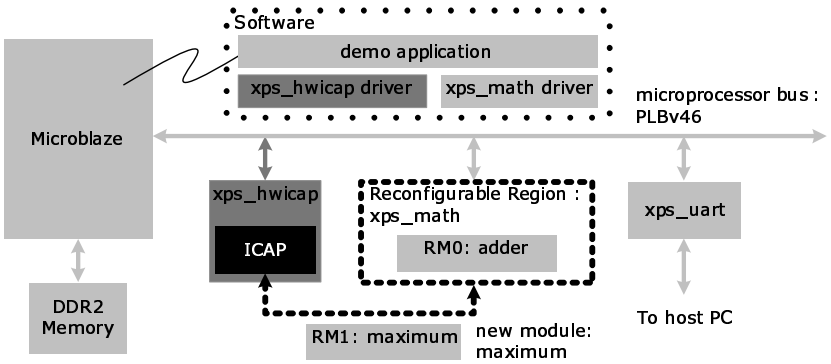


Figure 5.18: Reconfigurable Processor Peripheral reference design, after [93]

In order to simulate the system, we changed the reference design slightly. Since the original reference design did not provide source code for the math cores, we designed our own simple math cores (i.e., a `maximum` module and an `adder` module, which were similar to those used in Section 5.1.1). Since we did not have a simulation model for the provided SD card, we were not able to simulate the process of copying bitstreams from the SD card to the DDR2 memory. We therefore assumed that the copy was performed correctly and directly loaded SimBs to the DDR2 memory using ModelSim commands. It should be noted that the above changes were required since the original reference design was not provided in a simulatable form, and were not requirements of the ReSim library. To integrate ReSim with the simulation environment, the designer created 50 LOC (Tcl) describing parameters for generating the artifacts and 10 LOC (Verilog) to instantiate the generated artifacts.

Although the design was proven, we were able to expose a potential isolation bug in the system (i.e., BUG-Example.UG744.1). This isolation bug was not exposed in the reference design because RMs were not accessed during reconfiguration. To expose this bug, we deliberately modified the reference design so that the software attempted to access the RM during reconfiguration. In particular, instead of using the software to copy the bitstream data byte-by-byte to the `xps_hwicap` module, we added a Direct Memory Access (DMA) module to the system to perform bitstream transfer. Using the DMA module, the software was relieved from performing bitstream transfer and was thereby able to read software-accessible registers of the RM undergoing reconfiguration. It should be noted that we deliberately introduced the above-mentioned changes so as to expose the isolation bug. Figure 5.19 illustrates a simulation run that exposes the isolation bug.

BUG-Example.UG744.1: The Reconfigurable Peripheral reference design did not isolate the RR undergoing reconfiguration. Spurious outputs could therefore propagate from the RR to the static part of the system

- **BEFORE Reconfiguration @t1**: The value of the `rm1/statistic` register is 0xf00d003.
- **DURING Reconfiguration @t2**: The DMA starts transferring a SimB (see the `icap/cdata` signal) to the `xps_hwicap` module. As a result, RM1 is swapped out

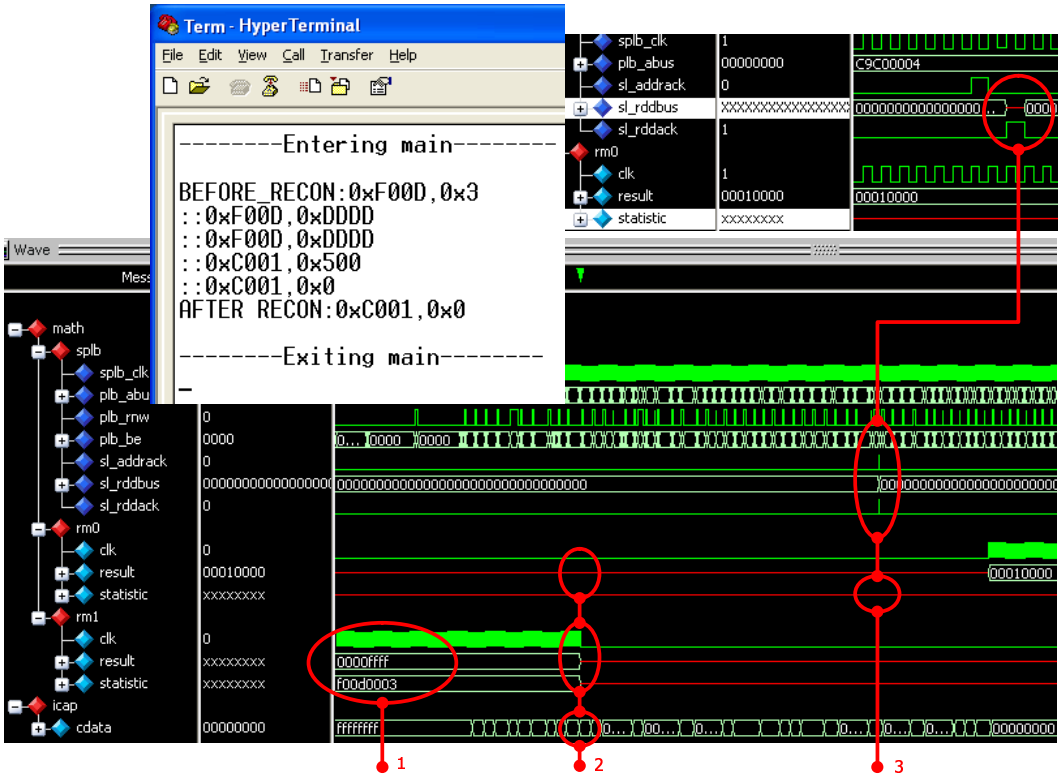


Figure 5.19: A *potential* isolation bug in the reference design

immediately and RM0 is not swapped in until the end of SimB transfer. Errors have been injected to the output signals of both RM0 and RM1 so as to mimic the spurious outputs of the RR during reconfiguration.

- **DURING Reconfiguration @t3**: The software attempts to access the `rm0/statistic` register. However, since reconfiguration is still in progress, the new module, RM0, has not yet been swapped in. As a result, an undefined "x" value is returned to the processor via the PLB bus (see the `sl_rddbus` signal in the upper right part of the figure).

This bug was easily identified because the undefined "x" values injected by ReSim propagated to the registers of the microprocessor. Furthermore, ReSim accurately simulates the timing of reconfiguration events (i.e., bitstream transfer, start and end of error injection) and activities of the static design (i.e., reading registers), which are all essential to detect this bug.

Using the slightly modified reference design, we were also able to expose the bug on the target FPGA. The upper left half of Figure 5.19 illustrates the standard output of the software. The software printed the value of the `statistic` register BEFORE, DURING and AFTER reconfiguration. In particular, the value of the `statistic` register was undefined and kept changing DURING reconfiguration. Exposing this bug on the FPGA demonstrates that ReSim-based simulation can accurately reveal DPR-related bugs in DRS designs.

5.5 Summary

Our case studies demonstrate that ReSim can be applied to a range of DRS design styles. In particular, we demonstrated the use of ReSim with in-house designs (i.e., the XDRS platform and the fault-tolerant DRS), a third-party design (i.e., AutoVision) and reference designs; with hardware-only designs (i.e., the FPCIe reference design and the fault-tolerant DRS) and with microprocessor-based HW/SW designs; with demo applications (i.e., streaming and periodic applications) and with real-world applications (i.e., a fault-tolerant application and a video processing application); a design that saves and restores module state via the configuration port (i.e., the periodic application on XDRS); with designs that use customized reconfiguration controllers (i.e., `xps_icapi`, `IcapCtrl`) and vendor IP (i.e., the `xps_hwicap` core); as well as with designs that were mapped to Virtex-5 (i.e., XDRS, fault-tolerant DRS, AutoVision) and to Virtex-6 (i.e., the two reference designs).

Our case studies also demonstrate that ReSim can be seamlessly integrated with well-established verification methodologies and extend such methodologies to DRS designs. For complex DRS designs such as the XDRS platform and the Optical Flow Demonstrator, it is not possible, in general, to distinguish a software bug from a hardware bug when the system is not working (e.g., BUG-Example.XDRS.3, BUG-Example.AUTO.4). HW/SW co-simulation has the advantage of allowing the designer to cross-debug variables and functions of the software and signals of the RTL hardware (see Sections 5.1 and 5.3). By creating a virtual platform, co-simulation can be performed at transaction level, which is fast enough to run the target software application (see Section 5.1). Since ReSim and/or Extended ReChannel-based simulation can accurately simulate the reconfiguration process, the designer can debug the software that controls the start and end of reconfiguration as if the software were running on a target FPGA.

Coverage analysis helps to track the progress of verification and to identify corner cases (see Sections 5.1, 5.2 and 5.4.1). Code coverage assisted in ensuring a reasonable level of confidence in a design without extra development overhead. Functional coverage further quantifies the verification progress at the cost of manually modeling coverage items. In particular, simulation artifacts such as SimBs can be used in the modeling of coverage items (see Section 5.1). Coverage analysis is typically combined with constrained random stimuli generation techniques to maximize the possibility of exercising corner cases and to improve coverage. In the verification process, the size of SimBs can be used as a random variable so as to exercise a range of execution scenarios of the system (see Sections 5.2 and 5.4.1).

We used ReSim-based simulation in the context of platform-based design (see Section 5.1.1) and IP-reuse methodology (see Section 5.3). For the in-house designed XDRS platform, although the first streaming application did not save and restore module state, the state saving and restoring capability of XDRS was thoroughly verified in the ReSim-based verification of the platform. As a result, the verification effort of the second periodic application was reduced due to the use of a platform-based design methodology. The verification progress of the AutoVision system indicated that IP *integration* could be much more difficult than a designer might expect, even when the individual modules

and software functions are FPGA-proven and reused. ReSim enabled the simulation of an integrated DRS design, and in doing so detected system integration bugs and bugs that were missed by Virtual Multiplexing.

Table 5.3 summarizes the case studies. We summarize these case studies in terms of the development workload, simulation overhead and bugs detected. The bugs in the table were all exposed in verifying the designs, i.e., we did not deliberately introduce any bug to a design. In particular, the table does not include the isolation bug (i.e., BUG-Example.UG744.1) that was deliberately introduced to the Processor Peripheral reference design. The table also does not include false positive bugs (i.e., BUG-Example.AUTO.1, and the 7 false positive bugs in the TLM model of the XDRS platform) identified in the case study. It should be noted that for the fault-tolerant DRS case study, we used ChipScope at the beginning of the project in order to compare ChipScope-based debugging with ReSim (see Section 5.2). We analyzed the bugs detected by ChipScope and found that all 4 bugs could easily have been detected via ReSim-based simulation.

Table 5.3: Summary of case studies

Case Study	Complexity of the reconfiguration machinery (LOC)	Parameter Script (LOC)	Simulation Overhead (%)	DPR-related Bugs (ReSim/Others)
XDRS (Streaming Application)	1300 (Verilog, excluding EDK code) + 1150(C)	50 (Tcl)	8.3	34/0
XDRS (Periodic Application)	1300 (Verilog, excluding EDK code) + 1750(C)	50 (Tcl)	6.8	5/1
Fault-tolerant DRS	2150 (Verilog, excluding ICAP-I & Flash controller)	60 (Tcl)	20.9	18/4
AutoVision	1250 (VHDL) + 400(C)	80 (Tcl)	1.7	7/0
Fast PCIe Configuration (XAPP883)	3500 (Verilog, excluding CoreGen code)	150 (Tcl)	0.3	–
Processor Peripheral (UG744)	2400 (VHDL) + 3200(C)	50 (Tcl)	0.7	–

Development Workload. The extra development workload for using ReSim involved creating the parameter scripts outlined in Figure 4.2, which ranged from 50-150 LOC (Tcl) for these case studies. Since we planned for simulation-based verification at the very beginning of the two in-house case studies, the workload of building the simulation environment and applying ReSim-based simulation was more predictable. However, in the AutoVision case study (see Section 5.3), we spent significant effort in setting up a baseline simulation environment. For the two reference designs (see Section 5.4), the workload to build a simulation environment or applying ReSim to an existing simulation environment was not significant compared with the complexity of the designs. Generally

speaking, the workload of using ReSim is trivial compared to the effort spent creating a DRS design and a testbench.

Admittedly, simulation involves overheads to develop testbenches, create high-level reference models (see Section 5.1) and model functional coverage items (see Sections 5.1 and 5.2). Referring to the Tools and Methodologies Objective of this thesis (see Section 1.1), a designer needs to trade-off between various methodologies and apply them to specific DRS design projects. It should be noted these overheads are not introduced by ReSim. As described by the Tools and Methodologies Objective of this thesis (see Section 1.1), ReSim aims to bridge the gap between verifying DRS designs and verifying traditional static designs; it does not aim to reduce the effort of functional verification itself.

Simulation Overhead. For each case study, we used the ModelSim profiling tool to evaluate the simulation overhead of ReSim. We found that 0.3–20.9% of simulation time was spent in ReSim (including both in generated artifacts and in library components). The simulation overhead of ReSim is proportional to the number of signals crossing the RR boundary since all boundary signals are multiplexed as opposed to being connected to the static part directly. The overhead is also proportional to the frequency of reconfiguration in a specific simulation run since each reconfiguration involves costs to swap modules and inject errors, and a scenario-dependent delay to transfer the SimB. Overall, the simulation overhead was lower for more complex designs (e.g., AutoVision, FPCIe, etc) and was higher for simpler designs (e.g., the fault-tolerant DRS).

Bugs Detected. From the development progress and the bugs detected from various case studies, we notice that the reconfiguration process could be much more difficult to verify than a designer might expect. Correctly verifying the RMs and the static region is essential but does not guarantee the correct transition from one configuration to another. Table 5.4 lists all example bugs described in this Chapter (including false positive bug(s) and deliberately introduced bug(s)). Apart from citing the Bug ID, we add to each bug a Tag indicating the key words associated with each bug so as to remind readers of the details of these bugs. Bugs could be introduced immediately BEFORE (e.g., BUG-Example.XDRS.2, BUG-Example.XDRS.5, ...), DURING (e.g., BUG-Example.XDRS.4, BUG-Example.AUTO.2, ...) and AFTER (e.g., BUG-Example.FT.2, BUG-Example.AUTO.4, ...) reconfiguration. Bugs could be introduced to hardware, software (e.g., BUG-Example.XDRS.2, BUG-Example.XDRS.6, BUG-Example.AUTO.3) and a mixture of hardware and software (e.g., BUG-Example.XDRS.3, BUG-Example.AUTO.4). Even if individual modules and software functions were FPGA-proven and reused, bugs could still be introduced due to mismatches between module parameters (e.g., BUG-Example.AUTO.2), inconsistencies between software and hardware (e.g., BUG-Example.AUTO.3) and using proven IP in a different scenario (e.g., BUG-Example.FT.3, BUG-Example.AUTO.4). Therefore, it is highly desirable to test and debug an *integrated* DRS design including the process of reconfiguring the design with a range of configuration bitstream sizes that could be expected at run time.

Table 5.4: Summary of example bugs described in Chapter 5

Bug ID	Bug Tag	Section ...	Detected by ...
BUG-Example.XDRS.1	ICAPI_DONE	5.1	Extended ReChannel
BUG-Example.XDRS.2	Flush_Cache	5.1	**Extended ReChannel**
BUG-Example.XDRS.3	Multiple_Recon	5.1	**Extended ReChannel**
BUG-Example.XDRS.4	Isolation_Mismatch	5.1	**ReSim**
BUG-Example.XDRS.5	Block_Until_Idle	5.1	ReSim
BUG-Example.XDRS.6	Restoration_Pointer	5.1.1	**ReSim**
BUG-Example.XDRS.7	Wrong_Pad_Word	5.1.1	On-chip Debugging
BUG-Example.FT.1	Level_Clock	5.2	On-chip Debugging Could have been detected by ReSim
BUG-Example.FT.2	Feed_Pipeline	5.2	**ReSim**
BUG-Example.FT.3	Too_Quick_Recon	5.2	**ReSim**
BUG-Example.AUTO.1	Sig_Reg	5.3	Virtual Multiplexing
BUG-Example.AUTO.2	IcapCtrl_PLB	5.3	**ReSim**
BUG-Example.AUTO.3	Driver_Update	5.3	**ReSim**
BUG-Example.AUTO.4	Engine_Reset	5.3	**ReSim**
BUG-Example.UG744.1	Bus_Isolation	5.4.2	ReSim On-chip Debugging

Bugs that can ONLY be detected by ReSim/Extended ReChannel-based simulation are marked in bold in the "Detected by ..." column in Table 5.4 (see detailed descriptions of each bug in relevant Sections). Other bugs could also be detected with previous methods such as Virtual Multiplexing [54] and DCS [69]. For example, BUG-Example.XDRS.1 and BUG-Example.XDRS.5 could be detected by simulating the *static* part of the design in a module-level testbench using conventional simulation techniques, because exposing the two bugs did not require modeling any characteristic feature of DPR. However, since ReSim enables simulating an integrated DRS design before, during and after reconfiguration, it significantly reduces the system integration effort. As another example, BUG-Example.UG744.1 could also be detected by DCS, because it also supports injecting errors to the static region while the RM is being reconfigured [69]. However, since ReSim models characteristic features of DPR, it is able to detect more bugs missed by previous methods (see Section 5.3). Furthermore, since ReSim does not require changing the design for simulation purposes, it does not introduce false positive bugs (e.g., BUG-Example.AUTO.1).

For a complete list of all DPR-related bugs detected in the case studies, please refer to Appendix A.

Our case studies indicate that it is non-trivial to insert probe logic and to debug the implemented DRS design using ChipScope (see, BUG-Example.XDRS.7, BUG-Example.FT.1). For the AutoVision case study, even the shortest debug turnaround

time for on-chip debugging is longer than the longest debug turnaround time for ReSim-based simulation (see Section 5.3). Furthermore, on-chip debugging does not collect coverage and can only validate a limited number of scenarios for the implemented design (see BUG-Example.FT.3). Even when a module is FPGA-proven, a bug can still be introduced during system integration (see BUG-Example.AUTO.2). However, on-chip debugging is completely accurate and can detect fabric-dependent bugs (e.g., BUG-Example.XDRS.7), which cannot be detected by ReSim-based simulation (see Section 3.3). Referring to the Tools and Methodologies Objective of this thesis (see Section 1.1), we do not advocate eliminating on-chip debugging. Nevertheless, we believe that it is highly desirable to perform simulation to identify and fix as many fabric-independent bugs as possible in the early stage of the design cycle, and leave the fabric-dependent part of the design to be tested on the target FPGA.

Chapter 6

Conclusions

This chapter concludes the thesis. The chapter starts with concluding remarks on the contributions of this thesis (see Section 6.1). We describe our contributions according to the thesis objectives described in Section 1.1. Section 6.2 describes the future work. With regard to the limitations of the simulation-only layer (see Section 3.3), we illustrate two possible directions to resolve the challenges of eliminating fabric-dependent bugs in DRS designs. We also discuss the possible future trend to verifying DRS designs at run-time.

6.1 Concluding Remarks

Due to the exponential increase in hardware design costs and risks, the electronics industry has begun shifting towards the use of reconfigurable devices such as FPGAs as mainstream computing platforms. Compared with customized ASIC designs, FPGA-based hardware/software systems can be flexibly programmed in the field and are therefore more amenable to upgrades and bug fixes over the product life-cycle. Dynamic Partial Reconfiguration further extends the flexibility of FPGAs by allowing hardware logic to be partially reprogrammed while the rest of a design continues to operate. However, such flexibility introduces significant new challenges to ensuring the functional correctness of DRS designs. The objective of this thesis was to develop a methodology and tools to assist designers in verifying DRS designs while part of their design is undergoing reconfiguration.

For simulation-based functional verification, the primary challenge in verifying DRS arises from the conflicting requirements of simulation accuracy and verification productivity. In particular, in order to exercise and verify reconfiguration scenarios of the user design, the simulation environment needs to model the FPGA fabric so as to accurately simulate the inter-layer interactions between the user design and the physical layers. However, for the sake of productivity, simulation and verification of the user design should abstract away the details of the FPGA fabric to which the design is mapped. For effective simulation-based functional verification of hardware designs, either static

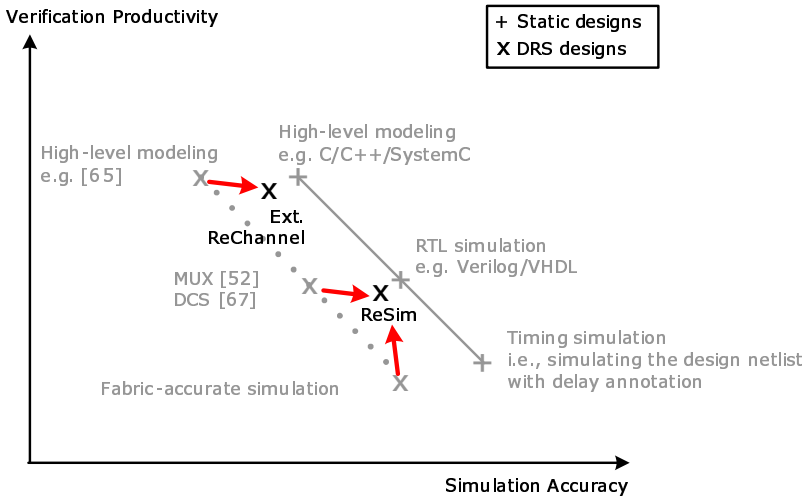


Figure 6.1: Tradeoff between simulation accuracy and verification productivity (using ReSim and Extended ReChannel)

or DRS, designers therefore need to find a balance between accuracy and productivity (see Figure 6.1, similar to Figure 3.6).

For static designs (the solid line in the figure), cycle-accurate RTL simulation is at the middle of the spectrum, and is both cycle-accurate and physically independent. Compared with the simulation of static designs, designers have to sacrifice either accuracy or productivity in simulating DRS designs (the dashed line in the figure). Fabric-accurate simulation can be viewed as a cycle-accurate simulation of the FPGA device together with the configuration-bit representation of the user design. Although its simulation accuracy is comparable to traditional RTL simulation, fabric-accurate simulation suffers from low verification productivity and, more importantly, the simulation model of the FPGA fabric is not openly available to DRS designers. Without being dependent on the fabric, previous high-level modeling (e.g., ReChannel [67]) or RTL simulation (e.g., MUX-based method [54], DCS [69]) approaches achieve the desired level of productivity while sacrificing accuracy in simulating the reconfiguration process.

This thesis balances the conflicting requirements of simulation accuracy and verification productivity by introducing a simulation-only layer. The simulation-only layer can be viewed as a fabric-independent FPGA device that mimics the impact of the FPGA fabric on the user design. We abstracted away the details of the FPGA fabric by extracting and modeling 6 characteristic features of DPR, namely, module swapping, triggering condition, bitstream traffic, bitstream content, spurious outputs and undefined initial state. Using the simulation-only layer, simulation can be thought of as functionally verifying the *user design layer* of a DRS on a fabric-independent FPGA. Simulation using the simulation-only layer is therefore able to assist designers in detecting fabric-independent bugs of a DRS design, thereby fulfilling the Functional Verification Objective of the thesis.

From a CAD tool perspective, we have proposed an Extended ReChannel library for the high-level modeling of DRS designs, and a ReSim library to support RTL simulation of DRS designs. The two libraries are two possible implementations, at different levels of

abstraction, of the simulation-only layer concept. Using a library-based approach, our method does not require designers to change their design for simulation purposes and the simulated designs are therefore implementation-ready. The libraries provide Tcl APIs to help designers in generating parameterizable source code for the simulation-only layer. From a methodology perspective, we have demonstrated the use of ReSim with well-established verification methodologies such as top-down modeling and coverage-driven verification. The case studies demonstrated that ReSim-based simulation assisted in detecting bugs that would have been missed using previous methods. Although not being able to expose fabric-dependent bugs, ReSim-based simulation helps designers to reduce their reliance on costly on-chip debugging. The overall effort of functionally verifying DRS designs has therefore been significantly reduced. We have thereby achieved the Tools and Methodologies Objective of the thesis. Since the ReSim and the Extended ReChannel libraries can be used with the mainstream DRS design flow, we have also been able to achieve the Mainstream Objective of the thesis.

By mimicking the FPGA fabric using a simulation-only layer, ReSim significantly improves simulation accuracy over previous RTL level approaches with negligible productivity penalties (i.e., development and simulation overheads), and significantly improves simulation productivity compared with fabric-accurate simulation (see Figure 6.1). Using similar concepts, Extended ReChannel also improves the accuracy of high-level modeling without sacrificing much productivity. ReSim has been released as an open source tool under the BSD license, and is available via `http://code.google.com/p/resim-simulating-partial-reconfiguration` .

6.2 Future Work

The simulation-only layer proposed in this thesis can offer limited assistance in detecting fabric-dependent bugs. It is also not straightforward to apply our approach to verify module relocation or fine-grained reconfiguration (see Section 3.3). Since module relocation and fine-grained reconfiguration improve the design flexibility compared with the modular reconfiguration flow proposed by FPGA vendors, it would be desirable to support simulation-based functional verification of non-modular DRS designs as well. We propose two possible directions to resolve such a verification gap but leave both directions as future work.

One direction is to further improve the simulation accuracy of the simulation-only layer by capturing more details of the FPGA fabric (e.g., placement information, configuration bits). For example, instead of simulating the entire FPGA fabric, it is possible to simulate a small portion of the design (e.g., the LUTs that are fine-grained reconfigured) in a fabric-accurate manner, which can significantly improve the simulation accuracy for fine-grained reconfigurable designs. On the other hand, to simulate module relocation, it is possible to keep track of module placement information in the simulation-only layer, so as to ensure modules do not overlap with each other. The more accurate the simulation-only layer is, the less mismatches exist between the simulation-only layer and the target FPGA, and the fewer false positive and negative bugs are introduced. However, in order to maintain verification productivity, the simulation-only layer cannot include all details

of the FPGA fabric. The modeling should focus on the inter-layer interactions between the physical layer and the user design instead of on activities within the physical layer.

On the other hand, it should be noted that with the increased number and variety of inter-layer interactions, it is more and more difficult to simulate a DRS design without being dependent on the physical layer. The mismatches between the simulation-only layer and the target FPGA fabric could cause more false positive bugs and false negative bugs to be reported by simulation. For these DRS designs, it is inevitable to rely on on-chip debugging to identify and fix fabric-dependent bugs. Therefore, the other way to resolve the verification gap is to improve the verifiability of a DRS design by reducing the degree of inter-layer interactions within a DRS design. In particular, it is desirable that:

- The designer keeps the fabric-dependent part to a minimum or to avoid using fabric-dependent features in the design. This can also improve the portability of the design while reducing verification difficulty.
- The designer separates the fabric-dependent and fabric-independent parts so as to localize potentially unidentified bugs to the fabric-dependent part of the design. By carefully partitioning the design, the designers can maximize the number of fabric-independent bugs that could be exposed using the simulation-only layer and field testing can focus on debugging the fabric-dependent part, which requires less effort than debugging the entire design.
- For DRS designs that perform module relocation or fine-grained reconfiguration, it is desirable that designers reuse existing proven design tools and IP that support non-modular reconfiguration so as to avoid creating fabric-dependent components from scratch.

The simulation-only layer can also be extended to non-simulation-based verification such as formal verification. As described in Section 2.2.3, the designer needs to capture the intended interaction between the user design and the FPGA fabric in a formal specification, and use a formal verification tool to verify that a design meets its formal specification. Ideally, the formal specification needs to capture the inter-layer interactions without modeling the details of the FPGA fabric. It would be possible to model the inter-layer interactions using artifacts that are similar to the simulation-only layer, except that the artifacts should be implemented such that they can be easily used by formal verification tools. However, the general concepts of the simulation-only layer still apply.

As described in Section 4.4, the current implementation of the library also imposes some limitations. However, since ReSim applies object-oriented techniques and provides Tcl APIs, users can extend the library for their specific test purposes. Furthermore, since the library is released as an open source tool, users can, if they wish, modify the library to address the implementation-related limitations. Last but not least, we aim to address the implementation-related limitations of ReSim in future releases of the library.

Looking ahead, dynamic reconfiguration has changed the way engineers design hardware and will also change the way hardware is verified. As described in Section 2.4, since a

DRS can be designed to be open-ended, it requires the capability to verify its correct reconfiguration and execution at run time. Despite limited research being done, run-time verification is a long term trend for future DRS designs. Compared with design-time verification, run-time verification of open-ended DRS designs introduces two main challenges:

- Hardware modules/tasks sourced externally to the system may not have been thoroughly verified. Functional bugs could exist in the module itself or can be introduced when integrating the module with the rest of the system. Even for thoroughly verified modules, the reconfiguration process can introduce bugs to the system. For example, configuration bits can be corrupted either externally or internally to the design. If the Frame Address bits have been corrupted, reconfiguration could incorrectly change the static user logic.
- Apart from checking bugs that are unintentionally introduced to the design, the system also needs to identify malicious hardware tasks that could damage the system. Therefore, run-time verification needs to verify the credibility of the source of configuration bitstreams and reconfiguration requests. Unauthorized hardware tasks should not be allowed to run. The verification problem then morphs into a security problem.

Furthermore, even for traditional static designs, there is a growing need to perform run-time verification so as to detect and mitigate configuration bit errors introduced by radiation or device aging. FPGA-based systems use configuration scrubbing and/or partial reconfiguration to fix run-time configuration errors [92] [16]. In such case, the verification problem morphs into a fault-tolerance problem.

Since it is not practical to run simulation at system run time, the proposed simulation-only layer approach cannot be used for run-time verification. Furthermore, since run-time verification directly tests hardware tasks on the target FPGA, it is not necessary to provide designers with a simplified simulation model of the FPGA device and run-time verification should therefore be 100% accurate. However, run-time verification suffers from low visibility and controllability. Furthermore, the system may not be able to afford many resources to perform the verification. Existing run-time verification techniques (e.g., run-time equivalence checking [81], proof carrying hardware [24], FPGA-in-the-loop simulation [61] and self-checking circuitry [63], see Section2.4) do not systematically or thoroughly verify DRS designs and significant effort is needed to address these issues.

Appendix A

Bugs Detected in Case Studies

This appendix describes DPR-related bugs that were detected in the case studies described in Chapter 5 (see Table 5.3). Our discussion does not include any bug that was deliberately introduced to the case studies (e.g., BUG-Example.UG744.1 bug). We also do not include false positive bugs (i.e., BUG-Example.AUTO.1, and the 7 false positive bugs in the TLM model of the XDRS platform) identified in the case study. We describe each bug in the following format

- Bug ID @ Time detected (as described in the corresponding case study)
  - Synopsis: A brief description of the cause of the bug
  - Description: A more detailed explanation of the cause of the bug
  - Method: The method we used to detect the bug. If Extended ReChannel or ReSim is the only method to detect the bug, we also explain the reason why the bug cannot be detected using previous methods.
  - Alternative Method(s): By analyzing the cause of the bug, we determine alternative simulation-based methods, such as ReChannel [67], Virtual Multiplexing [54] and DCS [69], which could be used to detect the bug. It should be noted that all of these bugs can, in general, be detected by on-chip debugging at the cost of significantly greater debugging time. We therefore do not explicitly list on-chip debugging as an alternative method.

By comparing Extended ReChannel and ReSim with previous simulation-based approaches, we aim to demonstrate that the proposed simulation-only layer approach is effective in assisting designers in identifying DPR-related bugs. The bugs described here can also be used as a reference for future designers to understand the potential sources of DPR-related bugs.

A.1 Case Study I: In-house DRS Computing Platform

For the XDRS case study (see Section 5.1), simulation-based approaches detected 34 DPR-related bugs in the streaming application and 5 DPR-related bugs in the periodic

application. We detected 1 fabric-dependent bug in the periodic application using ChipScope. We describe the bugs according to the time at which they were detected during the project development (see Figure 5.2).

Weeks 3-5: TLM modeling using Extended ReChannel.

- BUG.XDRS.TLM.1 @ Week 3
  - Synopsis: Address map mismatch in SW/HW
  - Description: The status register of the `xps_icapi` module was mapped to offset 0x4 in software and to offset 0x0 in hardware.
  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel
- BUG.XDRS.TLM.2 (Same as BUG-Example.XDRS.1) @ Week 3
  - Synopsis: Incorrect reset value for the `ICAPI_DONE` flag
  - Description: The `xps_icapi` module uses an `ICAPI_DONE` flag to indicate the end of bitstream transfer. Such a flag should be set to 1 at power up but was incorrectly initialized to 0. This bug was detected in the TLM model of the system, but it revealed a potential bug that could also exist in the RTL design.
  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel
- BUG.XDRS.TLM.3 @ Week 3
  - Synopsis: Software reset register
  - Description: HW/SW co-simulation indicated that a software reset register had to be added to the `xps_icapi` module.
  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel
- BUG.XDRS.TLM.4 @ Week 3
  - Synopsis: Incorrect PLB connection
  - Description: The master PLB interface of the `xps_icapi` module was incorrectly left unconnected.
  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel
- BUG.XDRS.SW.1 @ Week 3
  - Synopsis: A bug in an *`if`* condition

  - Description: A logic error in polling the status of the `xps_icapi` module.

```
while ( Icapi_GetStatus() == ICAPI_IS_BUSY ) { ... }
/* WAS. Icapi_GetStatus() !=ICAPI_IS_BUSY */
```

  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel

- BUG.XDRS.SW.2 @ Week 3

  - Synopsis: A bug in an assertion
  - Description: A logic error in the software driver that checks the `IDCODE`.

```
Xil_AssertNonvoid(IcapiReadIdCode() == FPGA_IDCODE);
/* WAS. Xil_AssertNonvoid(IcapiReadIdCode() != FPGA_IDCODE); */
```

  - Method: Extended ReChannel. Since ReSim supports configuration readback, software can read and check the IDCODE of the simulation-only layer in the same way as reading the `IDCODE` of the target FPGA. The above software code can therefore be tested in ReSim-based simulation
  - Alternative Method(s): None

- BUG.XDRS.SW.3 @ Week 4

  - Synopsis: A bug in setting a control register
  - Description: The software driver didn't correctly set the `start` bit of the `ctrl_reg` register and the bitstream transfer didn't start.

```
void Icapi_SetCfgStart() {
    ...
    ctrl_reg |= 0x1;
    /* WAS. ctrl_reg &= 0x1 */
```

  - Method: Extended ReChannel. The bug was detected because bitstream transfer was not started in simulation. Since Extended ReChannel explicitly models bitstream traffic, the start and end of the bitstream traffic can be simulated and verified.
  - Alternative Method(s): None

- BUG.XDRS.TLM.5 @ Week 4

  - Synopsis: Transferring the same bitstream twice.
  - Description: The software starts reconfiguration by writing the `start` bit of the control register of the `xps_icapi` module. However, it incorrectly set the `start` bit twice. Although this bug was caused by software, we modified the hardware to ignore the second *start*.
  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel

- BUG.XDRS.TLM.6 @ Week 4

  - Synopsis: Fail to enable interrupts before reconfiguration
  - Description: The software needs to enable interrupts before reconfiguration. Following the previous bug, ignoring multiple *starts* in the `xps_icapi` module also incorrectly ignored the operation that enabled interrupts.
  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel

- BUG.XDRS.SW.4 @ Week 4
  - Synopsis: Typo in asserting function argument
  - Description: An assertion of the software driver checked a wrong argument.

```
void Icapi_SetCbHandler(IcapiCbHandler_t FuncPtr, void* CbRef) {
    Xil_AssertVoid(FuncPtr != NULL);
    /* WAS. Xil_AssertVoid(CbRef != NULL); */
```

  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel

- BUG.XDRS.SW.5 @ Week 4
  - Synopsis: Fail to disable interrupts after reconfiguration
  - Description: Failed to disable interrupts of the `xps_icapi` module in the software driver. As a result, the interrupts were incorrectly asserted twice.

```
void Icapi_InterruptHandler(){
    ...
    Icapi_SetIntrDisable(); /* WAS. no such call */
    DrvPtr->CbHandler(DrvPtr->CbRef, Icapi_GetStatus());
```

  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel

- BUG.XDRS.SW.6 (Same as BUG-Example.XDRS.2) @ Week 4
  - Synopsis: Fail to flush cache
  - Description: To accelerate bitstream transfer, the system software loads bitstream data from a slow flash memory and buffers the bitstreams in a fast DDR2 memory. However, the software failed to flush the bitstream data from the processor cache to the DDR2 memory and the `xps_icapi` module transferred incorrect bitstreams from the DDR2 memory during reconfiguration.
  - Method: Extended ReChannel. The bug was easily identified since un-flushed SimB data was transferred to the simulated ICAP port but the new RM was not swapped in during simulation, and would not have been identified without modeling bitstream traffic.
  - Alternative Method(s): None

- BUG.XDRS.SW.7 @ Week 4
  - Synopsis: Bug in setting bitstream transfer size

  - Description: The transfer size register of the `xps_icapi` module uses size in WORDs while software incorrectly set size in BYTEs.

```
void Icapi_DoConfigurationIntr() {
    ...
    DrvPtr->BufferSize=NumWords
    /* WAS. DrvPtr->BufferSize=NumWord*4 */
```

  - Method: Extended ReChannel. Since Extended ReChannel models bitstream traffic, this bug was easily identified. In particular, since the `xps_icapi` module incorrectly attempted to transfer more data than required, it accessed un-defined memory space and the simulation failed.
  - Alternative Method(s): None

- BUG.XDRS.TLM.7 @ Week 5
  - Synopsis: Tuning register definitions.
  - Description: HW/SW co-simulation indicated that the `xps_icapi` module should use one ACK field in the status register for all three reconfigurable regions instead of using three.
  - Method: Extended ReChannel
  - Alternative Method(s): ReChannel

- BUG.XDRS.TLM.8 (Same as BUG-Example.XDRS.3) @ Week 5
  - Synopsis: Illegal reconfiguration sequence
  - Description: The `SyncMgr` mistakenly requested a second reconfiguration before the first one had finished (i.e., an illegal reconfiguration sequence). The designer fixed the bug by modifying the hardware to report such illegal reconfiguration requests and by adding a `reconfiguration_in_progress` flag to the software driver.
  - Method: Extended ReChannel. The bug was detected because the delay associated with the first reconfiguration was accurately modeled by transferring the SimB using Extended ReChannel.
  - Alternative Method(s): None

- BUG.XDRS.SW.8 @ Week 5
  - Synopsis: RM is not reset after reconfiguration
  - Description: A software reset function should be called by the interrupt service routine as a user-defined call-back function at the end of reconfiguration. However, the software reset function was not correctly called.
  - Method: Extended ReChannel. In the TLM model, RMs need to be activated by the software writing to a software reset register of the `SyncMgr` module. Since the software reset function was not correctly called, the software reset register was not written and the RMs were not activated.
  - Alternative Method(s): ReChannel

Weeks 6-8: Coverage-driven Verification using ReSim.

- BUG.XDRS.RTL.1 @ Week 6
  - Synopsis: Cycle mismatch in asserting the *write enable* signal of the ICAP.
  - Description: Should assign the `wr_en` signal from `state_c` not `state_n` .
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None
- BUG.XDRS.RTL.2 @ Week 6
  - Synopsis: Incorrect bitstream size.
  - Description: The `ICAP-I` module loaded one more word of bitstream data than needed.
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None
- BUG.XDRS.RTL.3 @ Week 6
  - Synopsis: Active-high vs. active-low enable signals.
  - Description: The *chip select* signal of the ICAP port is active-low but was incorrectly asserted high during reconfiguration.
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None
- BUG.XDRS.RTL.4 @ Week 6
  - Synopsis: Fail to reverse the order of ICAP signals.
  - Description: The ICAP signals on Virtex-5, Virtex-6 and Virtex-7 have been bit-reversed. The designer failed to reverse the bit order of the `ICAP-I` reconfiguration controller, which was previously designed for Virtex-4 FPGAs.
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None
- BUG.XDRS.RTL.5 @ Week 6
  - Synopsis: Active-high vs. active-low reset signals
  - Description: The control signal of the `Isolation` module is active-low but was incorrectly assigned with an active-high source. As a result, RMs were incorrectly isolated when the system was not reconfiguring.
  - Method: ReSim.

  - Alternative Method(s): Virtual Multiplexing, DCS

- BUG.XDRS.RTL.6 @ Week 6
  - Synopsis: A bug in connecting the `Isolation` module.
  - Description: Incorrectly connected an non-isolated signal to the static region and left the isolated signal unused.
  - Method: ReSim. The bug was identified because the undefined "x" values injected by ReSim propagated to the static region.
  - Alternative Method(s): DCS

- BUG.XDRS.RTL.7 @ Week 6
  - Synopsis: Bug in driving the *chip select* signal of ICAP
  - Description: The `ICAP-I` module should assert the chip select signal of ICAP when reading from and writing to the ICAP. However, the chip select was not asserted for configuration readback.

```
assign cfg_cen = ~((state_c == WRCFG) || (state_c == RDCFG));
/* WAS. assign cfg_cen = ~((state_c == WRCFG)); */
```

  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.XDRS.RTL.8 @ Week 6
  - Synopsis: Typo
  - Description: Incorrectly asserted a write flag for a read operation of the `ICAP-I` module.

```
... begin ma_select = 1'b1; ma_rnw = 1'b1; end
/* WAS. ... begin ma_select = 1'b1; ma_rnw = 1'b0; end */
```

  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.XDRS.RTL.9 @ Week 6
  - Synopsis: Bug in requesting bus access.
  - Description: The `ICAP-I` module should request bus access for each word of the readback SimB. The system incorrectly asserted the request signal high throughout the configuration readback and only one word of the readback SimB was returned.
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.XDRS.RTL.10 @ Week 6
  - Synopsis: Bug in updating the `bsize` & `baddr` registers in the `ICAP-I` module.
  - Description: For configuration readback, both registers should be updated during the *last* cycle of the `WRMEM` state of the FSM.

```
end else if (...((state_c == WRMEM)&&(xbm_ack == 1'b1)) ) begin
/*WAS. ((state_c == WRMEM)) */
    baddr <= baddr + 32'h1;
    bsize <= bsize - 32'h1;
```

  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.XDRS.RTL.11 @ Week 6
  - Synopsis: Cycle mismatch in reading back configuration data
  - Description: The `ICAP-I` module incorrectly missed the cycle when the read-back bitstream/SimB data was returned from the ICAP.
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.XDRS.RTL.12 @ Week 6
  - Synopsis: Monitoring incorrect signal
  - Description: Fail to keep track of pipeline status.
  - The `pipeline_sync` module is used to block reconfiguration requests until the pipelines of the RMs are drawn. However, the module failed to track the correct number of valid data in the pipelines and incorrectly asserted the `pipeline_empty` flag.
  - Method: ReSim.
  - Alternative Method(s): Virtual Multiplexing, DCS

- BUG.XDRS.RTL.13 @ Week 6
  - Synopsis: Bug in resetting the RM
  - Description: The `enable` input signal of a reconfigurable filter was incorrectly asserted when resetting the RM.

```
lpfirTF #(...) fir (
    ...
    .en ( fir_nd & rstn ), /* WAS. .en ( fir_nd ) */
```

  - Method: ReSim.
  - Alternative Method(s): Virtual Multiplexing, DCS

- BUG.XDRS.RTL.14 (Same as BUG-Example.XDRS.4) @ Week 7

  - Synopsis: Cycle mismatch in isolation.
  - Description: The `Isolation` module, which disconnects the RM DURING reconfiguration, resumed such connection one cycle too early AFTER reconfiguration.
  - Method: ReSim. The bug was identified because the undefined "x" values injected by ReSim propagated to the static part in the mismatched cycle. Although such "x" injection is similar to [69], ReSim allows cycle-accurate simulation of the transition from DURING to AFTER reconfiguration, which is also essential to detect this type of bug.
  - Alternative Method(s): None

- BUG.XDRS.RTL.15 (Same as BUG-Example.XDRS.5) @ Week 8
  - Synopsis: Fail to block a reconfiguration request.
  - Description: If a reconfiguration request arrived precisely when the RM had just started processing the next input sample, the RM failed to block the reconfiguration request, which violated the `Blocked_until_Idle` item in the test plan. The bug was detected since the `Blocked_until_Idle` assertion failed.
  - Method: ReSim.
  - Alternative Method(s): Virtual Multiplexing, DCS

Week 9: Integrating the XDRS core logic with the bus interfacing logic.

- BUG.XDRS.RTL.16 @ Week 9
  - Synopsis: Active-high vs. active-low reset signals
  - Description: The `SyncMgr` module uses an active-low reset whereas the PLB bus uses an active-high reset. The designer failed to invert the reset signal when connecting the core logic of XDRS with the PLB bus.
  - Method: ReSim.
  - Alternative Method(s): Virtual Multiplexing, DCS

- BUG.XDRS.RTL.17 @ Week 9
  - Synopsis: Bug in the bus interface logic
  - Description: In the bus interface logic of the `xps_icapi` module, the data signal may be acknowledged either before or after the command signal. The FSM of the bus interface logic failed to wait for the acknowledgments from both the data and the command signals before entering the next state.
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.XDRS.SW.9 @ Week 9
  - Synopsis: Fail to update software
  - Description: The `m_sync` register of the `SyncMgr` module had been added with a `clk_en` field which did not exist in the TLM model. However, the software that was verified in TLM modeling was not correctly updated. This bug was caused by a mismatch between the RTL and TLM models.
  - Method: ReSim.
  - Alternative Method(s): Virtual Multiplexing, DCS

Weeks 12: Targeting a Second Application

- BUG.XDRS.SSR.1 @ Week 12
  - Synopsis: Incorrect bitstream/SimB header type.
  - Description: The software generated a bitstream/SimB with an incorrect bitstream header type. In particular, a type 2 header was incorrectly used with a type 1 data packet.
  - Method: ReSim. The incorrect bitstream header type was detected by the ICAP artifact, which printed an error message. Since the SimB has a similar structure as a real bitstream, some bugs in the bitstream itself can still be detected by ReSim.
  - Alternative Method(s): None.
- BUG.XDRS.SSR.2 @ Week 12
  - Synopsis: Uninitialized software variable.
  - Description: The `IcapiGcapture()` software function is used to assemble a `GCAPTURE` bitstream/SimB packet at run time and to send the packet to the ICAP. However, the size of the packet was incorrectly left uninitialized. At run time, the uninitialized packet size caused the software to generate an incorrect `GCAPTURE` packet.
  - Method: ReSim. Since ReSim also uses a `GCAPTURE` command to synchronize storage elements of simulated RMs with the state data stored in the simulation-only layer, a bug in assembling the `GCAPTURE` packet (like this bug) can readily be exposed during simulation.
  - Alternative Method(s): None.
- BUG.XDRS.SSR.3 (Same as BUG-Example.XDRS.6) @ Week 12
  - Synopsis: Invalid software pointer.
  - Description: The restoration routine of the software driver uses a pointer as an argument and the pointer is expected to point to the logic allocation information of the signal to be restored. However, the application software program passed an incorrect pointer to the restoration routine.

```
switch(Math_GetId(&MathDrv)) {
    case MATH_ADDER_ID: result_ll_ptr = &mca_result_ll; break;
/* WAS. case MATH_ADDER_ID: result_ll_ptr = &mcm_result_ll; */
```

  - Method: ReSim. The bug was detected as a consequence of incorrect values being restored to the simulated `statistic` register.
  - Alternative Method(s): None.

- BUG.XDRS.SSR.4 @ Week 12
  - Synopsis: Missing a `break` statement in a `case` statement.
  - Description: After fixing BUG.XDRS.SSR.3, the value of the simulated `statistic` register was still incorrect. The `case` statement shown above missed a `break` statement and the assignment to the restoration pointer was incorrectly overwritten.
  - Method: ReSim. The bug was detected as a consequence of incorrect values being restored to the simulated `statistic` register.
  - Alternative Method(s): None.

- BUG.XDRS.SSR.5 @ Week 12
  - Synopsis: Incorrect logic allocation information specified by software.
  - Description: To save and restore module state, the software needs to extract the state bits from bitstreams/SimBs according to the logic allocation information specified in the `.ll`/`.sll` file. However, due to a cut-and-paste error, the periodic application incorrectly specified the logic allocation of one bit of the `statistic` register.
  - Method: ReSim. The bug was detected as a consequence of incorrect values being restored to the simulated `statistic` register.
  - Alternative Method(s): None.

- BUG.XDRS.SSR.6 (Same as BUG-Example.XDRS.7) @ Week 12
  - Synopsis: Incorrect number of pad words.
  - Description: The number of pad words returned from ICAP was not the same as the designer had expected. The software attempted to extract state bits from the wrong bit positions and the extracted data were therefore incorrect.
  - Method: On-chip debugging. Since ReSim failed to mimic the exact behavior of the target device, BUG.XDRS.SSR.6 was missed by ReSim-based simulation.
  - Alternative Method(s): None.

A.2 Case Study II: In-house Fault-Tolerant Application

As described in Section 5.2, ReSim-based simulation detected 18 DPR-related bugs in the fault-tolerant DRS case study. We also used ChipScope at the beginning of the project (see Section 5.2) to test the reconfiguration controller node of the network, and we detected 4 bugs. We describe the bugs according to the time at which they were detected during the project development (see Figure 5.11).

Days 1-6: Module-level testing of the Voter-NI module.

- BUG.FT.VOTER.1 @ Day 2
  - Synopsis: Cycle-mismatch in the network interface
  - Description: The `ready_to_receive_flit` signal should be cleared by the `flit_valid` signal, and should not be asserted until receiving the next flit. However, the handshake between the two signals were designed incorrectly. This bug was detected by an assertion.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim
- BUG.FT.VOTER.2 @ Day 3
  - Synopsis: Glitch from the majority voter.
  - Description: The coding-style of voter was not good and the voter outputs had glitches.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim
- BUG.FT.VOTER.3 @ Day 3
  - Synopsis: Fail to isolate module undergoing reconfiguration.
  - Description: During reconfiguration, the majority voter is also used to isolate spurious outputs from the RM. However, the coding-style of the voter was not good and the X values were not cleared. MUST use the *case equality* statement of Verilog to stop X propagation.

    ```
    assign sig=((sig_0===sig_1) || (sig_0===sig_2))? sig_0:sig_1;
    /* WAS. (... == ...), which produces X when either signal is X */
    ```

  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim
- BUG.FT.VOTER.4 @ Day 5
  - Synopsis: Counter bug.

  - Description: The sender counter of the network interface should count from 0 to 6, each of which corresponds to one bit in the flit to be sent. But the counter didn't stop until it reached 255.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

- BUG.FT.VOTER.5 @ Day 5

  - Synopsis: Counter bug.
  - Description: Due to the way the receiver FSM was designed, the receiver counter should count to 7 instead of 6. The incorrect counter value caused the receiver to miss the last bit on the network link. Since the FSMs in the receiver and the sender were different, the receiver counter was not the same as the sender counter and the sender counter could be reused directly.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

- BUG.FT.VOTER.6 @ Day 5

  - Synopsis: Fail to update a connection.
  - Description: Failed to connect the `filt_received` signal after the design had been slightly modified.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

- BUG.FT.VOTER.7 @ Day 6

  - Synopsis: Multiple signal drivers
  - Description: The `comms_alarm` signal was driven by two sources.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

- BUG.FT.VOTER.8 @ Day 6

  - Synopsis: Bitwidth bug for a signal
  - Description: The `perm_errors_one_hot` signal should be declared as THREE-bit wide but was incorrectly designed as a one bit signal.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

- BUG.FT.VOTER.9 @ Day 6

  - Synopsis: Decoding bug

  - Description: The receiver received a `RReq` (RRID=0x0) message, but decoded the RRID section of the message as a `TOKEN` message. This bug was detected by randomizing the RRID field of the messages in simulation. The bug was fixed by modifying the coding scheme of network messages. See BUG.FT.RN.7 below.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

- BUG.FT.VOTER.10 @ Day 6

  - Synopsis: Time-out on the network link
  - Description: If the current network node was in the `BYPASS` FSM state, it blocked the upstream nodes and the upstream node failed to wait for enough time before it asserted a time-out flag.
  - Method: Module-level testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

Days 1-6: Module-level testing of the RC-NI module.

- BUG.FT.RC.1 @ Day 3

  - Synopsis: Flash initialization bug
  - Description: The flash controller was not correctly initialized and the RC-NI failed to read bitstreams.
  - Method: Module-level testing using ChipScope.
  - Alternative Method(s): ReSim

- BUG.FT.RC.2 (Same as BUG-Example.FT.1) @ Day 5

  - Synopsis: Typo
  - Description: The designer accidentally created a level-sensitive clock instead of an edge-sensitive one, as desired.
  - Method: Module-level testing using ChipScope.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

- BUG.FT.RC.3 @ Day 5

  - Synopsis: Counter bug
  - Description: This bug was exactly the same as BUG.FT.VOTER.5, but it was introduced by a different designer and was identified using on-chip debugging instead of using simulation.
  - Method: Module-level testing using ChipScope.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

- BUG.FT.RC.4 @ Day 6
  - Synopsis: Initial `TOKEN` message bug
  - Description: The power up status for all nodes was set to `BYPASS` and there was no node initiating the `TOKEN` message.
  - Method: Module-level testing using ChipScope.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim

Days 7-15: Integrating the Voter-NI, RC-NI and the TMRed circuits.

- BUG.FT.RN.1 @ Day 7
  - Synopsis: Coding scheme mismatch
  - Description: Mismatch between the coding schemes when integrating modules created by two designers.
  - Method: Integrated testing using conventional RTL simulation.
  - Alternative Method(s): Virtual Multiplexing, DCS, ReSim
- BUG.FT.RN.2 @ Day 9
  - Synopsis: Connectivity bug
  - Description: Failed to connect clock/reset to the BAQ node.
  - Method: ReSim
  - Alternative Method(s): Virtual Multiplexing, DCS
- BUG.FT.RN.3 @ Day 9
  - Synopsis: Incorrect bitstream Look-up Table.
  - Description: The RC-node had an incorrect bitstream Look-up Table. As a result, the bitstream/SimB for the BAQ module was transferred to the ICAP when it was the FIR node that requested reconfiguration.
  - Method: ReSim. Since ReSim modeled bitstream traffic and triggered reconfiguration by SimBs, transferring an incorrect bitstream caused an incorrect module being swapped in and the SEU was not corrected in simulation.
  - Alternative Method(s): None
- BUG.FT.RN.4 @ Day 9
  - Synopsis: Fail to clear the error counter
  - Description: Failed to clear the error counters (i.e., the `errcnt_0,1,2` signals) after the SEU had been recovered.
  - Method: ReSim
  - Alternative Method(s): Virtual Multiplexing, DCS

- BUG.FT.RN.5 (Same as BUG-Example.FT.2) @ Day 9
  - Synopsis: Fail to feed the pipeline correctly
  - Description: After reconfiguration, the system failed to feed enough data to flush the internal pipeline of the FIR filter. Thus the undefined initial values of the pipeline were not properly cleared.
  - Method: ReSim. This bug was exposed since the undefined initial values injected to the pipeline registers of the simulated FIR filter propagated to the static region after reconfiguration, and the undefined values were detected by an `assert_isolate_at_feeding` assertion as described by item 4.7 in the test plan (see Figure 5.10).
  - Alternative Method(s): None
- BUG.FT.RN.6 @ Day 12
  - Synopsis: Bug in the `more_than_one_error` signal
  - Description: The `more_than_one_error` signal is used to indicate multiple event upsets in the system. It is calculated by examining individual bits of the `err_onehot` signal. However, the logic function was not correctly designed.

```
assign more_than_one_error <= (err_onehot[0] && err_onehot[1])
    || (err_onehot[0] && err_onehot[2])
    || (err_onehot[1] && err_onehot[2]);
/* WAS. assign ... <= (err_onehot[0] && err_onehot[1]); */
```

  - Method: ReSim
  - Alternative Method(s): Virtual Multiplexing, DCS
- BUG.FT.RN.7 @ Day 12
  - Synopsis: Decoding bug
  - Description: Following BUG.FT.VOTER.9, the coding scheme bug still hadn't been fixed. In particular, the RRID field of the `RReq` messages were incorrectly interpreted as another message by the receiver. The bug was fixed by adding to the receiving FSM an `RRID_FLAG`, which explicitly distinguished the RRID field of a `RReq` message from other messages.
  - Method: ReSim
  - Alternative Method(s): Virtual Multiplexing, DCS
- BUG.FT.RN.8 (Same as BUG-Example.FT.3) @ Day 13
  - Synopsis: Bug when reconfiguration was faster than message transfer
  - Description: Typically, reconfiguration is slower than transferring a message between two nodes. The expected operation of the RC-node is therefore: start reconfiguration; transfer an RAck message to a computing node; end reconfiguration; transfer an RDone message to a computing node. However, when the bitstream is very small and when the computing node is executing with a very slow clock, reconfiguration can finish before the RAck message is transferred. Under such a circumstance, the RC-node did not correctly send the RDone message.

  - Method: ReSim. In ReSim-based simulation, the size of a bitstream (i.e., the size of a SimB) can be adjusted for test purposes. This bug was exposed since the designer randomized the SimB size and the clock frequencies of the computing nodes. Unfortunately, such a scenario was not tested on the FPGA, and it is difficult to test since the size of a real bitstream cannot easily be adjusted for test purposes.
  - Alternative Method(s): None

A.3 Case Study III: Third-Party Video-Processing Application

As described in Section 5.3, we detected 6 DPR-related bugs in the AutoVision case study. We describe the bugs according to the time at which they were detected during the project development (see Figure 5.14). Our discussion does not include the BUG-Example.AUTO.1 bug, which is a false positive bug.

Weeks 10-11: Virtual Multiplexing-based simulation of the Optical Flow Demonstrator

- BUG.AUTO.1 @ Week 5
  - Synopsis: Coding style bug in the `Isolation` module
  - Description: When the design was *not* performing DPR, the `Isolation` module failed to connect all outputs from the CIE/ME engines to the static region. In particular, since the engines were changed from the point-to-point PLB mode to the shared PLB mode, new output signals were added to the engines. The `Isolation` module failed to connect new engine outputs to the static region.
  - Method: Virtual Multiplexing.
  - Alternative Method(s): DCS, ReSim

Weeks 10-11: ReSim-based simulation of the Optical Flow Demonstrator

- BUG.AUTO.2 @ Week 10
  - Synopsis: Coding style bug in the `Isolation` module
  - Description: To describe combinational logic using VHDL, all input signals of a block MUST be in the sensitivity list. However, the designer failed to put all inputs in the sensitivity list of the `Isolation` module. Although such a bug can automatically be fixed by the synthesis tool, it is still desirable to fix the bug manually.

  - Method: ReSim. Since ReSim modeled spurious outputs of RMs, the `Isolation` module was exercised in simulation and the bug was exposed because the errors injected by ReSim were not correctly isolated.
  - Alternative Method(s): DCS

- BUG.AUTO.3 @ Week 10

  - Synopsis: Connection error between the `IcapCTRL` module and its bus interface module, the `lisipif_master` module.
  - Description: The `IcapCTRL` module has been changed from a point-to-point connection structure to a shared bus-based structure. The `BE` signal of the `lisipif_master` interface should be hardwired to 0xFF.
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.AUTO.4 (Same as BUG-Example.AUTO.2) @ Week 10

  - Synopsis: Connection error on the PLB bus
  - Description: The `IcapCTRL` module was used in point-to-point mode in the original system, and it failed to work with the shared PLB bus in the modified Optical Flow Demonstrator. This bug was introduced by changing the way the IP was integrated.
  - Method: ReSim. Since ReSim modeled bitstream traffic, this bug was readily detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.AUTO.5 (Same as BUG-Example.AUTO.3) @ Week 10

  - Synopsis: Fail to update software driver
  - Description: After changing a parameter of the `IcapCTRL` module, the software driver was not updated accordingly and the SimB was not successfully transferred. The bug was introduced by a mismatch between hardware and software.
  - Method: ReSim. This bug was detected when a new engine was not swapped in due to the bug in the bitstream transfer process.
  - Alternative Method(s): None

- BUG.AUTO.6 (Same as BUG-Example.AUTO.4) @ Week 11

  - Synopsis: Engine reset bug
  - Description: The system software failed to wait until the completion of bitstream transfer before resetting the engines. This bug was introduced when the design was modified to use a different clocking scheme that slowed down the bitstream transfer and the software was not updated to slow down the reset operations accordingly.

  - Method: ReSim. Since ReSim more accurately modeled the timing of reconfiguration events, this bug could ONLY be detected by ReSim-based simulation.
  - Alternative Method(s): None

- BUG.AUTO.7 @ Week 11
  - Synopsis: Null pointer bug
  - Description: After reconfiguring the ME, the software interrupt handler needs to clear the feature list of the previous frame before starting the ME. However, the software failed to check the feature pointer before clearing the list,

```
/* WAS. Didn't check the DrvPtr->FeatureOutPtr pointer */
if((unsigned char*)(DrvPtr->FeatureOutPtr)!=NULL) {
    Mvec_ClearFeatureList((u32)(DrvPtr->FeatureOutPtr));
```

  - Method: ReSim.
  - Alternative Method(s): Virtual Multiplexing, DCS

A.4 Case Study IV & V: Vendor Reference Designs

As described in Section 5.4, we were not able to detect any bug in the vendor reference designs. We were only able to expose a potential bug (i.e., BUG-Example.UG744.1) by deliberately modifying the Processor Peripheral reference design (see Section 5.4.2).

Bibliography

[1] "Partial Reconfiguartion - Can I Insert ChipScope Cores within Reconfigurable Modules?" Xilinx Answer Record No. 42899, 2011. [Online]. Available: http://www.xilinx.com/support/answers/42899

[2] *Market Trends: Worldwide, ASIC and ASSP Design Starts Continue Declining Trend, 2012*, 2012. [Online]. Available: http://www.gartner.com/DisplayDocument?doc_cd=229088&ref=nl

[3] *The International Technology Roadmap for Semiconductors*, 2012. [Online]. Available: http://www.itrs.net/reports.html

[4] F. Altenried, "Time-sharing of Hardware Resources for Image Processing Accelerators using Dynamic Partial Reconfiguration," Bachelor's Thesis, Technical University of Munich, 2009.

[5] *FPGA Run-Time Reconfiguration: Two Approaches (WP01055)*, Altera Corporation, 2010. [Online]. Available: http://www.altera.com/literature/wp/wp-01055-fpga-run-time-reconfiguration.pdf

[6] *Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs (WP01137)*, Altera Corporation, 2010. [Online]. Available: http://www.altera.com/literature/wp/wp-01137-stxv-dynamic-partial-reconfig.pdf

[7] *Design Debugging Using the SignalTap II Embedded Logic Analyzer*, Altera Corporation, 2012. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii53009.pdf

[8] *Arria V Device Handbook*, Altera Corporation, 2013.

[9] J. Aynsley, *OSCI TLM-2.0 Language Reference Manual*, Open SystemC Initiative (OSCI), 2009. [Online]. Available: http://www.systemc.org/home

[10] T. Becker, W. Luk, and P. Y. Cheung, "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2007, pp. 35 – 44.

[11] C. Beckhoff, D. Koch, and J. Torresen, "Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2010, pp. 596 – 601.

[12] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A Self-Reconfiguring Platform," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2003, pp. 565 – 574.

[13] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The Erlangen Slot Machine: Increasing Flexibility in FPGA-based Reconfigurable Platforms," in *Field-Programmable Technology (FPT), International Conference on*, 2005, pp. 37 – 42.

[14] A. V. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. U. K. Melcher, "Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC," in *VLSI (ISVLSI), IEEE Computer Society Annual Symposium on*, 2007, pp. 35 – 40.

[15] F. Campi, A. Deledda, M. Pizzotti, L. Ciccarelli, P. Rolandi, C. Mucci, A. Lodi, A. Vitkovski, and L. Vanzolini, "A Dynamically Adaptive DSP for Heterogeneous Reconfigurable Platforms," in *Design, Automation and Test in Europe (DATE)*, 2007, pp. 1 – 6.

[16] E. Cetin, O. Diessel, L. Gong, and V. Lai, "Towards Bounded Error Recovery Time in FPGA-based TMR Circuits using Dynamic Partial Reconfiguration," *Field-Programmable Logic and Applications (FPL), International Conference on*, 2013, In press.

[17] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker, "A Multi-platform Controller Allowing for Maximum Dynamic Partial Reconfiguration Throughput," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2008, pp. 535 – 538.

[18] C. Claus, J. Zeppenfeld, F. Muller, and W. Stechele, "Using Partial-Run-Time Reconfigurable Hardware to Accelerate Video Processing in Driver Assistance System," in *Design, Automation and Test in Europe (DATE)*, 2007, pp. 1 – 6.

[19] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys (CSUR)*, vol. 34, no. 2, pp. 171 – 210, 2002.

[20] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473 – 491, 2011.

[21] C. Dennl, D. Ziener, and J. Teich, "On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2012, pp. 45 – 52.

[22] L. Devaux, S. Pillement, D. Chillet, and D. Demigny, "R2NoC: Dynamically Reconfigurable Routers for Flexible Networks on Chip," in *Reconfigurable Computing and FPGAs (ReConFig), International Conference on*, 2010, pp. 376 – 381.

[23] R. Drechsler, *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.

[24] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying Hardware: Towards Runtime Verification of Reconfigurable Modules," in *Reconfigurable Computing and FPGAs (ReConFig), International Conference on*, 2009, pp. 189 – 194.

[25] M. Feilen, M. Ihmig, C. Schwarzbauer, and W. Stechele, "Efficient DVB-T2 Decoding Accelerator Design by Time-Multiplexing FPGA Resources," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2012, pp. 75 – 82.

[26] S. P. Fekete, T. Kamphan, N. Schweer, C. Tessars, J. C. Van Der Veen, J. Angermeier, D. Koch, and J. Teich, "Dynamic Defragmentation of Reconfigurable Devices," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 2, pp. 8:1 – 8:20, 2012.

[27] M. Glasser, *Open Verification Methodology Cookbook*, Mentor Graphics Corporation, 2009. [Online]. Available: http://www.mentor.com/cookbook

[28] L. Gong, *ReSim Case Studies*, 2013. [Online]. Available: http://code.google.com/p/resim-simulating-partial-reconfiguration/

[29] ——, *ReSim User Guide*, 2013. [Online]. Available: http://code.google.com/p/resim-simulating-partial-reconfiguration/

[30] K. Goossens, M. Bennebroek, J. Y. Hur, and M. A. Wahlah, "Hardwired Networks on Chip in FPGAs to Unify Functional and Configuration Interconnects," in *Networks-on-Chip, ACM/IEEE International Symposium on*, 2008, pp. 45 – 54.

[31] H. Hinkelmann, P. Zipf, and M. Glesner, "A Domain-Specific Dynamically Reconfigurable Hardware Platform for Wireless Sensor Networks," in *Field-Programmable Technology (FPT), International Conference on*, 2007, pp. 313 – 316.

[32] C. H. Hoo and A. Kumar, "An Area-Efficient Partially Reconfigurable Crossbar Switch with Low Reconfiguration Delay," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2012, pp. 400 – 406.

[33] C.-H. Huang, K.-J. Shih, C.-S. Lin, S.-S. Chang, and P.-A. Hsiung, "Dynamically Swappable Hardware Design in Partially Reconfigurable Systems," in *Circuits and Systems (ISCAS), IEEE International Symposium on*, 2007, pp. 2742 – 2745.

[34] J. Huang and J. Lee, "A Self-Reconfigurable Platform for Scalable DCT Computation Using Compressed Partial Bitstreams and BlockRAM Prefetching," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 11, pp. 1623 – 1632, 2009.

[35] Y. Ichinomiya, S. Tanoue, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, "Improving the Robustness of a Softcore Processor against SEUs by Using TMR and Partial Reconfiguration," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2010, pp. 47 – 54.

[36] *IEEE Standard 1800-2005: SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, The Institute of Electrical and Electronics Engineers, Inc., 2005.

[37] *IEEE Standard 1364-2005: Verilog Hardware Description Language*, The Institute of Electrical and Electronics Engineers, Inc., 2006.

[38] *IEEE Standard 1076-2008: VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., 2009.

[39] *IEEE Standard 1850-2010: Property Specification Language (PSL)*, The Institute of Electrical and Electronics Engineers, Inc., 2010.

[40] *IEEE Standard 1666-2011: SystemC Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., 2012.

[41] A. Jara-Berrocal and A. Gordon-Ross, "VAPRES: A Virtual Architecture for Partially Reconfigurable Embedded Systems," in *Design, Automation and Test in Europe (DATE)*, 2010, pp. 837 – 842.

[42] Y.-C. Jiang and J.-F. Wang, "Temporal Partitioning Data Flow Graphs for Dynamically Reconfigurable Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 12, pp. 1351 – 1361, 2007.

[43] C. Kachris and S. Vassiliadis, "Analysis of a Reconfigurable Network Processor," in *Parallel and Distributed Processing, International Symposium on, Reconfigurable Architectures Workshop (RAW)*, 2006, pp. 1 – 8.

[44] H. Kalte and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfigurable Systems," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2005, pp. 223 – 228.

[45] F. Khan, N. Hosein, S. Vernon, and S. Ghiasi, "BURAQ: A Dynamically Reconfigurable System for Stateful Measurement of Network Traffic," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2010, pp. 185 – 192.

[46] D. Koch, C. Haubelt, and J. Teich, "Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation," in *Field-Programmable Gate Arrays (FPGA), ACM/SIGDA International Symposium on*, 2007, pp. 188 – 196.

[47] ——, "Efficient Reconfigurable On-Chip Buses for FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2008, pp. 287 – 290.

[48] D. Koch and J. Torresen, "FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting," in *Field-Programmable Gate Arrays (FPGA), ACM/SIGDA International Symposium on*, 2011, pp. 45 – 54.

[49] D. Koch, J. Torresen, C. Beckhoff, D. Ziener, C. Dennl, V. Breuer, J. Teich, M. Feilen, and W. Stechele, "Partial Reconfiguration on FPGAs in Practice - Tools and Applications," in *Architecture of Computing Systems (ARCS), International Conference on*, 2012, pp. 1 – 12.

[50] M. Kuehnle, A. Britoy, C. Roth, K. Dagas, and J. Becker, "The Study of a Dynamic Reconfiguration Manager for Systems-on-Chip," in *VLSI (ISVLSI), IEEE Computer Society Annual Symposium on*, 2011, pp. 13 – 18.

[51] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier, "Placing, Routing, and Editing Virtual FPGAs," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2001, pp. 357 – 366.

[52] V. Lai and O. Diessel, "ICAP-I: A Reusable Interface for the Internal Reconfiguration of Xilinx FPGAs," in *Field-Programmable Technology (FPT), International Conference on*, 2009, pp. 357 – 360.

[53] W. Luk, N. Shirazi, and P. Y. Cheung, "Modelling and Optimising Run-Time Reconfigurable Systems," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 1996, pp. 167 – 176.

[54] ——, "Compilation Tools for Run-time Reconfigurable Designs," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 1997, pp. 56 – 65.

[55] *Simulink: Simulation and Model-based Design*, MathWorks, 2012. [Online]. Available: http://www.mathworks.com.au/products/simulink/

[56] S. P. McMillan, B. J. Blodget, and S. A. Guccione, "VirtexDS: a Virtex Device Simulator," in *Proceedings of SPIE*, 2000, pp. 50 – 56.

[57] *ModelSim SE User's Manual (Software Version 6.5g)*, Mentor Graphics Corporation, 2010.

[58] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip," in *Design, Automation and Test in Europe (DATE)*, 2003, pp. 10 986 – 10 991.

[59] M. Montoreano, *Transaction Level Modeling using OSCI TLM 2.0*, Open SystemC Initiative (OSCI), 2007.

[60] F. Nava, D. Sciuto, M. D. Santambrogio, S. Herbrechtsmeier, M. Porrmann, U. Witkowski, and U. Rueckert, "Applying Dynamic Reconfiguration in the Mobile Robotics Domain: A Case Study on Computer Vision Algorithms," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 3, pp. 29:1 – 29:22, 2011.

[61] C. Paiz, C. Pohl, R. Radkowski, J. Hagemeyer, M. Porrmann, and U. Ruckert, "FPGA-in-the-Loop-Simulations for Dynamically Reconfigurable Applications," in *Field-Programmable Technology (FPT), International Conference on*, 2009, pp. 372 – 375.

[62] C. Patterson, P. Athanas, M. Shelburne, J. Bowen, J. Suris, T. Dunham, and J. Rice, "Slotless Module-Based Reconfiguration of Embedded FPGAs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, pp. 6:1 – 6:26, 2009.

[63] K. Paulsson, M. Hubner, M. Jung, and J. Becker, "Methods for Run-time Failure Recognition and Recovery in Dynamic and Partial Reconfigurable Systems Based on Xilinx Virtex-II Pro FPGAs," in *Emerging VLSI Technologies and Architectures, IEEE Computer Society Annual Symposium on*, 2006, pp. 1 – 6.

[64] A. Pelkonen, K. Masselos, and M. Cupak, "System-level Modeling of Dynamically Reconfigurable Hardware with SystemC," in *Parallel and Distributed Processing, International Symposium on, Reconfigurable Architectures Workshop (RAW)*, 2003, pp. 1 – 8.

[65] T. Pionteck, R. Koch, and C. Albrecht, "Applying Partial Reconfiguration to Networks-On-Chips," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2006, pp. 1 – 6.

[66] A. Piziali, *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic Publishers, 2004.

[67] A. Raabe, P. A. Hartmann, and J. K. Anlauf, "ReChannel: Describing and Simulating Reconfigurable Hardware in SystemC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 1, pp. 15:1 – 15:18, 2008.

[68] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-Chip Verification Methodology and Techniques* . Kluwer Academic Publishers, 2002.

[69] I. Robertson and J. Irvine, "A Design Flow for Partially Reconfigurable Hardware," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 2, pp. 257 – 283, 2004.

[70] K. Rupnow, W. Fu, and K. Compton, "Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2009, pp. 63 – 70.

[71] M. Saldana, A. Patel, H. J. Liu, and P. Chow, "Using Partial Reconfiguration in an Embedded Message-Passing System," in *Reconfigurable Computing and FPGAs (ReConFig), International Conference on*, 2010, pp. 418 – 423.

[72] O. Sander, L. Braun, M. Hubner, and J. Becker, "Data Reallocation by Exploiting FPGA Configuration Mechanisms," in *Applied Reconfigurable Computing (ARC), International Symposium on*, 2008, pp. 312 – 317.

[73] A. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design and Test of Computers*, vol. 18, no. 6, pp. 23 – 33, 2001.

[74] A. Schallenberg, W. Nebel, A. Herrholz, and P. A. Hartmann, "OSSS+R: A Framework for Application Level Modelling and Synthesis of Reconfigurable Systems," in *Design, Automation and Test in Europe (DATE)*, 2009, pp. 970 – 975.

[75] P. Sedcole, B. Blodget, J. Anderson, P. Lysaght, and T. Becker, "Modular Partial Reconfiguration in Virtex FPGAs," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2005, pp. 211 – 216.

[76] P. Sedcole, P. Y. K. Cheung, G. A. Constantinides, and W. Luk, "Run-Time Integration of Reconfigurable Video Processing Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 9, pp. 1003 – 1016, 2007.

[77] A. Seffrin, A. Biedermann, and S. A. Huss, "Tiny-n: A Novel Formal Method for Specification, Analysis, and Verification of Dynamic Partial Reconfiguration Processes," in *Specification and Design Languages (FDL), Forum on*, 2010, pp. 1 – 6.

[78] A. Seffrin and S. A. Huss, "Hardware-Accelerated Execution of Pi-Calculus Reconfiguration Schedules," in *Field-Programmable Technology (FPT), International Conference on*, 2011, pp. 1 – 8.

[79] M. Shelburne, C. Patterson, P. Athanas, M. Jones, B. Martin, and R. Fong, "Metawire: Using FPGA Configuration Circuitry to Emulate a Network-on-Chip," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2008, pp. 257 – 262.

[80] H. Simmler, L. Levinson, and R. Manner, "Multitasking on FPGA Coprocessors," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2000, pp. 121 – 130.

[81] S. Singh and C. J. Lillieroth, "Formal Verification of Reconfigurable Cores," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 1999, pp. 25 – 32.

[82] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *Parallel and Distributed Processing, International Symposium on, Reconfigurable Architectures Workshop (RAW)*, 2011, pp. 228 – 235.

[83] S. Tam and M. Kellermann, *Fast Configuration of PCI Express Technology through Partial Reconfiguration (XAPP883)*, Xilinx Inc., 2010.

[84] X. Tian and C.-S. Bouganis, "A Run-Time Adaptive FPGA Architecture for Monte Carlo Simulations," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2011, pp. 116 – 122.

[85] T. Todman, P. Boehm, and W. Luk, "Verification of Streaming Hardware and Software Codesigns," in *Field-Programmable Technology (FPT), International Conference on*, 2012, pp. 147 – 150.

[86] M. Ullmann, M. Hubner, B. Grimm, and J. Becker, "An FPGA Run-Time System for Dynamical On-Demand Reconfiguration," in *Parallel and Distributed Processing, International Symposium on, Reconfigurable Architectures Workshop (RAW)*, 2004, pp. 1 – 8.

[87] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.

[88] T. Vogt and N. Wehn, "A Reconfigurable Application Specific Instruction Set Processor for Convolutional and Turbo Decoding in a SDR Environment," in *Design, Automation and Test in Europe (DATE)*, 2008, pp. 38 – 43.

[89] G. Wigley and D. Kearney, "The Development of an Operating System for Reconfigurable Computing," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2001, pp. 249 – 250.

[90] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Using Run-Time Circuit Reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 2, pp. 247 – 256, 1998.

[91] S. Wray, W. Luk, and P. Pietzuch, "Run-Time Reconfiguration for a Reconfigurable Algorithmic Trading Engine," in *Field-Programmable Logic and Applications (FPL), International Conference on*, 2010, pp. 163 – 166.

[92] *Correcting Single-Event Upsets Through Virtex Partial Configuration (XAPP216)*, Xilinx Inc., 2000.

[93] *PlanAhead Software Tutorial: Partial Reconfiguration of a Processor Peripheral (UG744)*, Xilinx Inc., 2009.

[94] *Virtex-4 FPGA Configuration User Guide (UG071)*, Xilinx Inc., 2009.

[95] *ChipScope Pro 12.1 Software and Cores (UG029)*, Xilinx Inc., 2010.

[96] *EDK Concepts, Tools and Techniques (UG683)*, Xilinx Inc., 2010.

[97] *ISE In-Depth Tutorial (UG695)*, Xilinx Inc., 2010.

[98] *Partial Reconfiguration User Guide (UG702)*, Xilinx Inc., 2010.

[99] *Synthesis and Simulation Design Guide*, Xilinx Inc., 2010.

[100] *Virtex-5 FPGA Configuration User Guide (UG191)*, Xilinx Inc., 2010.

[101] *Virtex-6 FPGA Configuration User Guide (UG360)*, Xilinx Inc., 2010.

[102] *Vivado Design Suite Tutorial - High-level Synthesis (UG871)*, Xilinx Inc., 2012.

[103] *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)*, Xilinx Inc., 2012.

[104] *7 Series FPGAs Configuration User Guide (UG470)*, Xilinx Inc., 2013.

[105] *Xilinx Corporate Overview*, Xilinx Inc., 2013. [Online]. Available: http://www.xilinx.com/aboutus/corporate_overview.pdf

[106] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.

[107] S. Yusuf, W. Luk, M. Sloman, N. Dulay, E. C. Lupu, and G. Brown, "Reconfigurable Architecture for Network Flow Analysis," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, no. 16, 2008, pp. 57 – 65.