ReSim Case Studies

A Report on Design Verification Experience of
Dynamically Reconfigurable FPGA-based Systems

Version 2.3b

May 2013

Revision History

The following table shows the revision history for this document.

Version	Time	Description
2.3b	May. 2012	Renamed `sbt_trans` to `simop_trans` Renamed `rsv_monitor` to `rsv_region_recorder` Added `rsv_configuration_port`, which instantiates `rsv_icap_virtex` & `rsv_sbt_parser`
2.3a	Nov. 2012	Changed the error injection mechanism, use `rsv_ei_trans` to trigger error injection Removed user-defined monitor
2.2a	Sep. 2012	First public release

Contents

List of Figures

List of Tables

List of Abbreviations

ASIC Application-Specific Integrated Circuit

BRAM Block Random Access Memory

CLB Configuration Logic Block

CP Configuration Port

DPR Dynamic Partial Reconfiguration

DRS Dynamically Reconfigurable System

DUT Design Under Test

FA Frame Address

FPGA Field-Programmable Gate Array

ICAP Internal Configuration Access Port

IP Intellectual Property

HDL Hardware Description Language

HVL Hardware Verification Language

LUT Look-Up Table

PLB Processor Local Bus

RM Reconfigurable Module

RR Reconfigurable Region

RTL Register Transfer Level

SimB Simulation-only Bitstream

SSR State Saving and Restoration

TB Testbench

TLM Transaction Level Modeling

Preface

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 Field Programmable Gate Arrays (FPGAs) as mainstream computing platforms. An FPGA is a special type of integrated circuit that can be programmed and reprogrammed with arbitrary logic function. Traditionally, an FPGA device is programmed when the system is powered up. Using Dynamic Partial Reconfiguration (DPR), the programming and reprogramming of the FPGA can occur at system run time and can be controlled by the system itself (i.e., self-reconfiguration). In particular, Dynamically Reconfigurable Systems (DRS) implemented on FPGAs can reprogram/reconfigure part of their circuits *at run time* to adapt to changing execution requirements [38][3]. By mapping multiple reconfigurable hardware modules (RM) to the same physical reconfigurable region (RR) of the FPGA, the system can time-multiplex its submodules at run time and the design density of a DRS is increased [33].

Compared with traditional static FPGA designs, DPR has introduced additional flexibility for system designers but has also introduced challenges to the verification of 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 [38]. 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 [1]. Although Altera proposed to support behavioral simulation of the reconfiguration process, it has not yet incorporated such simulation support into its tool flow [4]. As a result, new simulation approaches need to extend traditional simulation techniques to assist designers in testing and debugging DRS designs while part of the design is undergoing reconfiguration.

ReSim is a reusable simulation library to support *cycle-accurate* simulation of modular reconfigurable DRS designs. It assists designers[1] in verifying integrated DRS designs BEFORE, DURING and AFTER partial reconfiguration, including the transfer of configuration bitstreams and the subsequent module swapping. This document presents case studies of using the ReSim library, and includes

[1]Since design engineers also spend significant time in testing and debugging, this document does not explicitly distinguish between design engineers and verification engineers but refers to both as designers.

- Introduction to ReSim (see Chapter 1): Describes the core ideas of ReSim
- Case Studies (see Chapter 2): Describes the development process and example bugs detected in case studies.
- Bugs Detected in Case Studies (see Chapter 3): Describes a list of all bugs detected in case studies.

0.1 System Requirements

ReSim has been tested using QuestaSim/ModelSim 6.5g [22] on Windows XP Professional SP2 machine and should work on other platforms. The tool also require Tcl 8.4 (or later) [6] and Python 2.5 (or later) [21]. The State Migration has only been tested on EDK 12.4 [37].

0.2 Expected Audience

It is expected that the readers have basic knowledge about FPGAs and Dynamic Partial Reconfiguration. It is also expected that the readers understand RTL simulation and simulation-based functional verification of FPGA-based systems.

0.3 Documentation

For Engineers.

- **Lingkan Gong**, "ReSim User Guide - A Guide to RTL Simulation of Dynamically Reconfigurable FPGA-based Systems " (2.3b), 2013
- **Lingkan Gong**, "ReSim Case Studies - A Report on Design Verification Experience of Dynamically Reconfigurable FPGA-based Systems" (2.3b), 2013

For Researchers.

- **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, In press

- **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.

0.4 Acknowledgments

The researchers would like to thank Mr. Jens Hagemeyer from the University of Paderborn for his guidance on state restoration on Virtex-5 FPGAs. We also appreciate Mr. Johny Paul and Prof. Walter Stechele from the Technical University of Munich for providing the AutoVision design as an important case study for assessing ReSim. Lastly, we would like to thank Xilinx for their generous donations.

0.5 Contact Us

Mr. Lingkan Gong, george.gong.47@gmail.com

Dr. Oliver Diessel, odiessel@cse.unsw.edu.au

http://code.google.com/p/resim-simulating-partial-reconfiguration/

Chapter 1

Introduction to ReSim

1.1 Dynamic Partial Reconfiguration

Using DPR, hardware modules that do not need to run simultaneously can be time-multiplexed (see Figure 1.1). In particular, Reconfigurable Modules (RM), whose temporal activities are mutually exclusive, are mapped to the same Reconfigurable Region (RR) and are loaded on demand by partial reconfiguration of the FPGA device. 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. Figure 1.1 indicates that DRS designs have two conceptual layers:

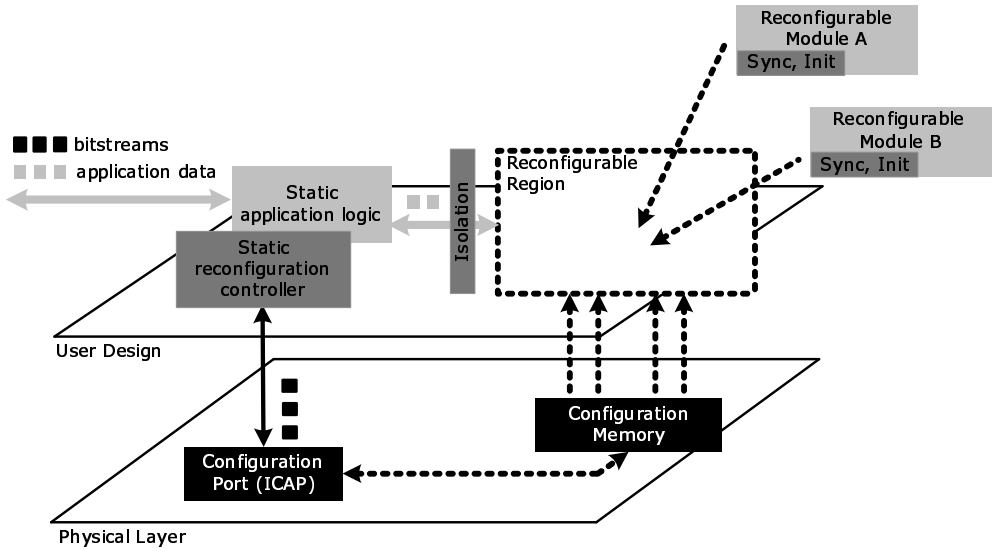


Figure 1.1: Conceptual diagram of a DRS design

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.

Compared with the static designs, DRS designs need to *synchronize/pause/disable* and *initialize/restart* the outgoing and the incoming RMs before and after reconfiguration. During reconfiguration, a reconfiguration controller *transfers* 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 (CP). 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.

Physical Layer. The physical layer (darkly shaded blocks) 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, the configuration distribution network commencing with the configuration port (e.g., ICAP or SelectMap), and the configuration bitstream used to program/reprogram the device. The implementation of the physical layer is proprietary to the FPGA vendors.

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. 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 the *configuration port*. Internal to the physical layer, bitstreams are distributed 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). 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 1.1) and the configuration bitstream is the *medium* for such inter-layer interactions.

1.2 Simulating Dynamic Partial Reconfiguration

Register Transfer Level (RTL) simulation is the most common method of verifying hardware (either ASIC- or FPGA-based) design functionality. Since DPR is the process of reprogramming the configuration memory of the FPGA fabric with a configuration bitstream, cycle-accurate simulation of the reconfiguration process involves modeling the FPGA fabric (i.e., *fabric-accurate simulation*). However, since the organization of the

FPGA fabric, including the configuration memory and the configuration bitstream, is proprietary to the FPGA vendors, it is non-trivial for designers to *accurately* simulate the inter-layer interactions of the reconfiguration process. Furthermore, even if the simulation model of the FPGA fabric were available, fabric-accurate simulation would include a multitude of unnecessary details for verification and would significantly reduce verification *productivity*. An effective simulation method therefore needs to strike a balance between simulation accuracy and verification productivity. Furthermore, this balance is constrained by the desire for the simulated design to be *implementation ready*. That is, the captured design should not be changed for simulation purposes.

Simulation-only Layer. The core idea of ReSim is to use a simulation-only layer to *emulate* the physical fabric of FPGAs so as to assist designers in testing/debugging/verifying the user design. Figure 1.2 redraws Figure 1.1 with all the physically dependent blocks (darkly shaded boxes) replaced by their corresponding simulation-only artifacts (open boxes). In ReSim-based simulation,

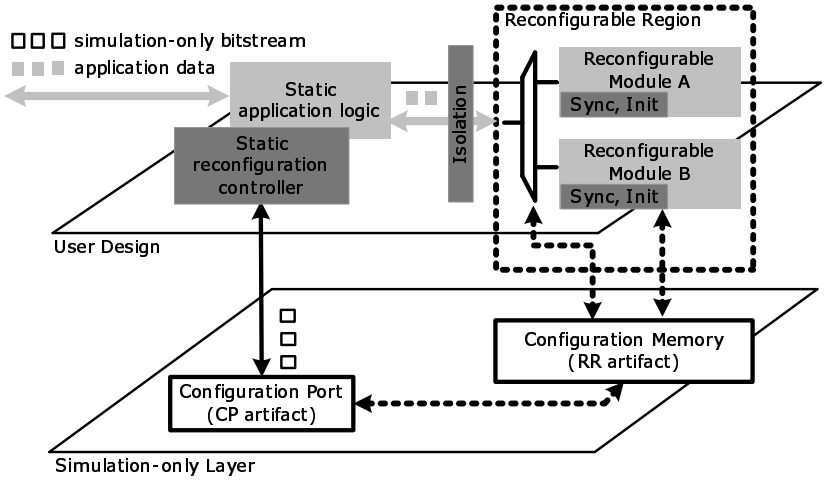


Figure 1.2: Using the simulation-only layer

- The configuration bitstreams are replaced by simulation-only bitstreams (SimB) which are used as the media of modeling the inter-layer interactions of DRS designs. Since the configuration bits found in a real bitstream can only be interpreted by the target FPGAs, a SimB re-defines the format of a bitstream and contains simulation-only fields. For example,
  - Instead of containing the Frame Addresses of the configuration data to be written, a SimB contains the numerical IDs (RMIDs and RRIDs) of the module to be reconfigured.
  - Instead of containing bit-level configuration settings for the module to be configured, the configuration data section of a SimB contains signature data and state data of the simulated module to be swapped in.

  - The size of a real bitstream is dependent on the FPGA resource used by a module whereas the size of a SimB is a user-defined parameter which can be adjusted for test purposes.

- Without loss of generality, possible configuration ports, either internal or external to the design, are represented by a CP artifact[1], which

  - Interact with the RTL user design according to the interfacing protocol specified by the device configuration guide (e.g., [35, 39, 40, 41]).
  - Takes a SimB instead of a real bitstreams 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.
  - Returns a readback SimB instead of readback bitstream so as to simulate configuration readback.

- The part of configuration memory to which each RR is mapped is modeled by an RR artifact[2], which

  - Controls module swapping by selecting a MUX-like `module_selector`, which connects one and only one active RM to the static region.
  - Triggers module swapping according to the numerical IDs of modules.
  - Injects errors to the static region so as to mimic the spurious RM outputs during reconfiguration.
  - Injects errors to the RMs so as to mimic the undefined initial RM state after reconfiguration.
  - Synchronize state data of the simulated RMs with the contents of the `spy_memory`.

We summarize the benefits of using the simulation-only layer from three aspects:

- ReSim-based simulation is *cycle accurate*. Using the simulation-only layer, the reconfiguration process, including the transfer of configuration bitstreams and the subsequent module swapping can be emulated, except that simulation-only bitstreams instead of real bitstreams are used and interpreted. Although the simulation-only layer is not a completely accurate model of the FPGA device, the user design, which is the focus of verification, is simulated in the desired cycle-accurate manner.

- ReSim-based simulation is *physically independent*. Instead of modeling the configuration bits of the FPGA fabric, the simulation-only layer only utilizes user design parameters (e.g., a list of interfacing signals that crosses RR boundaries, the affiliation of RMs and RRs and the target FPGA family) to model reconfiguration.

[1] The CP artifact was called "ICAP artifact" in previous publications
[2] The RR artifact was called "Extended Portal" in previous publications

Therefore, the productivity of verifying a DRS design is not compromised for the level of simulation accuracy. One significant extra benefit of physical independence is that FPGA vendors do not need to disclose the details of the FPGA device in order to support simulation of partial reconfiguration.

- The simulated design is *implementation ready.* Since the simulation-only layer emulates the target FPGA, the user design does not need to be changed for simulation purposes. As a result, a user design bug exposed by ReSim-based simulation reveals an actual bug in the implemented design, and the design as implemented instead of some variation of it is simulated and verified.

Use of a simulation-only layer to emulate an FPGA device is analogous to the use of a Bus Functional Model (BFM) to emulate a microprocessor when testing and verifying peripheral logic attached to a microprocessor bus [37]. Although the BFM 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, since the BFM approach abstracts away the internal behaviors of a processor such as pipelines, BFM-based simulation is more productive than simulating a completely accurate processor model. Similarly, the simulation-only layer emulates the FPGA device and captures the interactions between the physical device and the user design to be tested. Furthermore, the simulation-only layer abstracts away the details of the FPGA fabric and significantly improves the verification productivity compared with fabric-accurate simulation.

Chapter 2

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 [9], 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.

2.1 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 2.1). This platform is similar to existing generic slot-based DRS platforms such as the Erlangen Slot Machine [5], the VAPRES streaming architecture [17] and the Sonic video processing system [30]. 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 [28]. After thoroughly verifying the platform, the verification effort of applications customized/derived from the platform can be significantly reduced [26]. 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. Table 2.1 provides a list of facts of the streaming application of this cast study.

Table 2.1: Case study I fact sheet - (1)

Case Study	Facts
Case Study Name	XDRS (Streaming Application)
DUT Features	Processor+reconfigurable cores Generic DRS computing platform Streaming and periodic applications
Similar Designs	Erlangen Slot Machine [5] VAPRES streaming architecture [17] Sonic video processing system [30]
Target Board/FPGA	ML507/Virtex-5 FX70T
Verification Methods	Platform-based design, Top-down modeling, HW/SW co-simulation, Coverage analysis
Verification Tools	ReSim, Extended-ReChannel
Reference	Not yet published[a].
Released?	A simplified version of the XDRS core logic has been released (see the XDRS.QUICKSTART, XDRS.SINGLE, XDRS.MULTIPLE examples)

[a] It should be noted that the XDRS platform is not the same as the one illustrated in [11, 12]. In particular, the XDRS platform of this document is a processor-based embedded system targeting a ML507 board whereas the previous design was a hardware accelerator targeting an ADM-XRC-4 board. Furthermore, in order to systematically study the top-down modeling and the coverage-driven verification of the platform, the system was re-designed from scratch.

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 [8] [15], and the Xilinx `xps_hwicap` IP core [34]. Apart from configuring module logic, XDRS also supports saving and restoring module state via the configuration port in a way similar to [31, 18]. 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 2.1) 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.

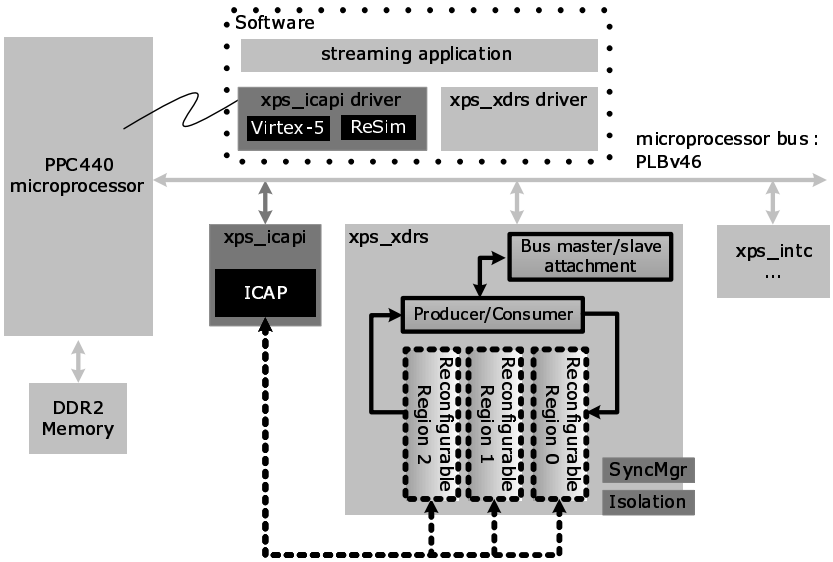


Figure 2.1: The XDRS demonstrator

Figure 2.2 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 [37], some 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

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

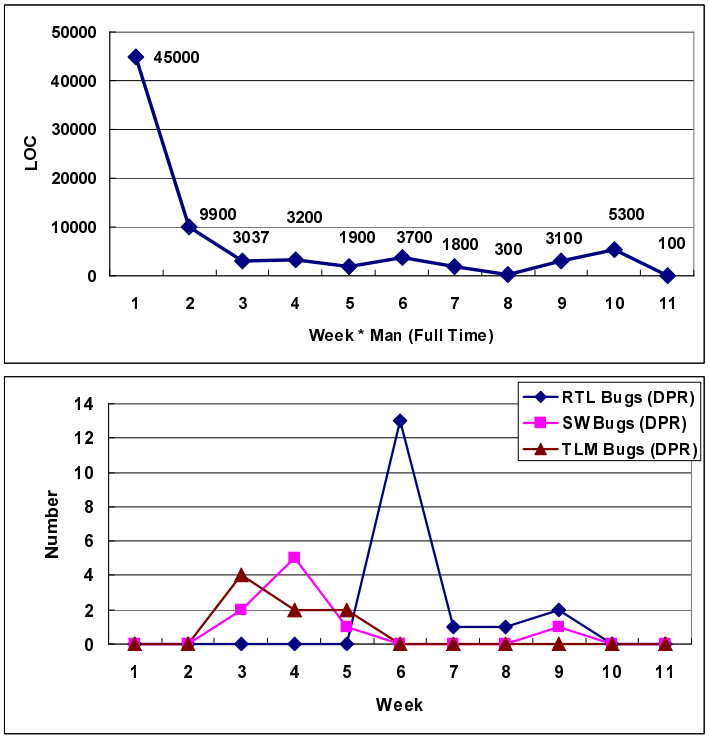


Figure 2.2: Development progress of the XDRS system

- 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 [26] 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 [26] was used to debug system software and the integration of software and hardware. At RTL level, we applied coverage-driven verification [24] 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 2.3 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.

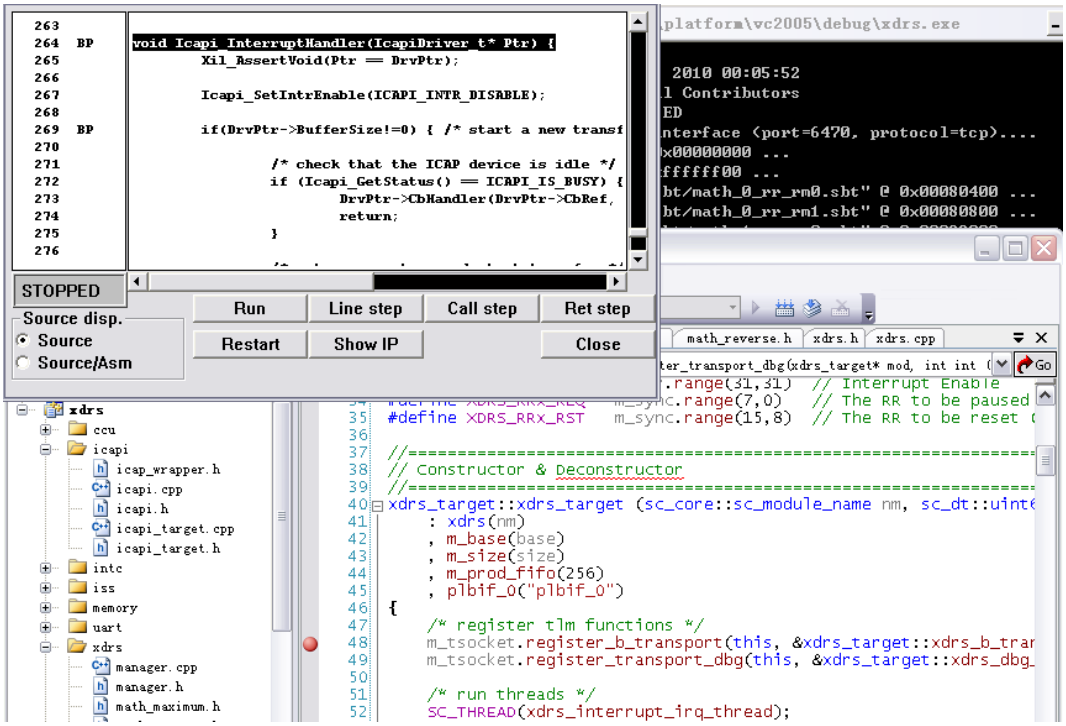


Figure 2.3: A Screen-shot of the co-simulation environment for the XDRS system

TLM modeling assisted in identifying and fixing 16 DPR-related bugs in the system (see Figure 2.2). Some DPR bugs captured in the TLM model revealed potential design defects in the system hardware (e.g., BUG-Example.XDRS.1, see page 14). 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.

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.

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.

In this case study, we detected 7 bugs in the TLM model (not shown in Figure 2.2) 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 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 2.4) 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 2.4) is composed of the core logic of the `xps_icapi` module, i.e., the `ICAP-I` reconfiguration controller [19], 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 2.4), 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.

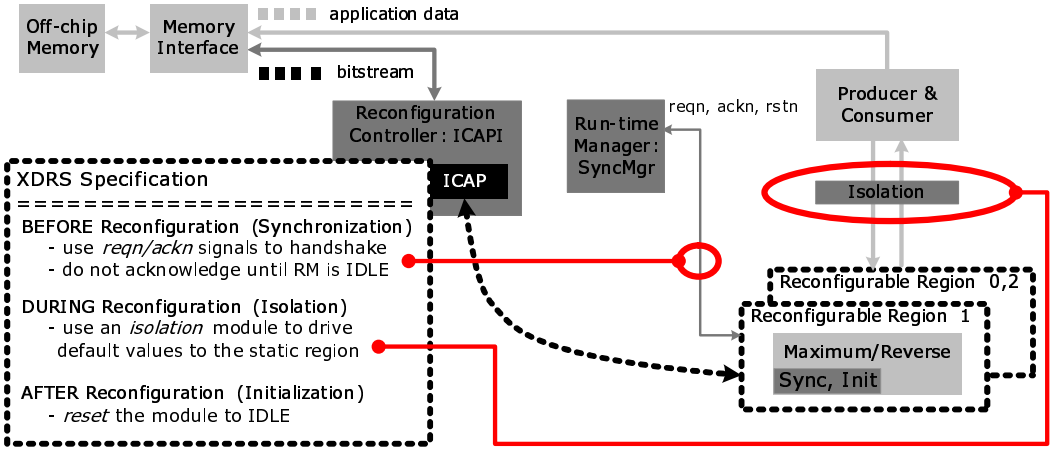


Figure 2.4: The core logic of the XDRS demonstrator

The use of ReSim enables the designer to debug the cycle-accurate behavior of the XDRS system while it is performing partial reconfiguration. Figure 2.5 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 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.

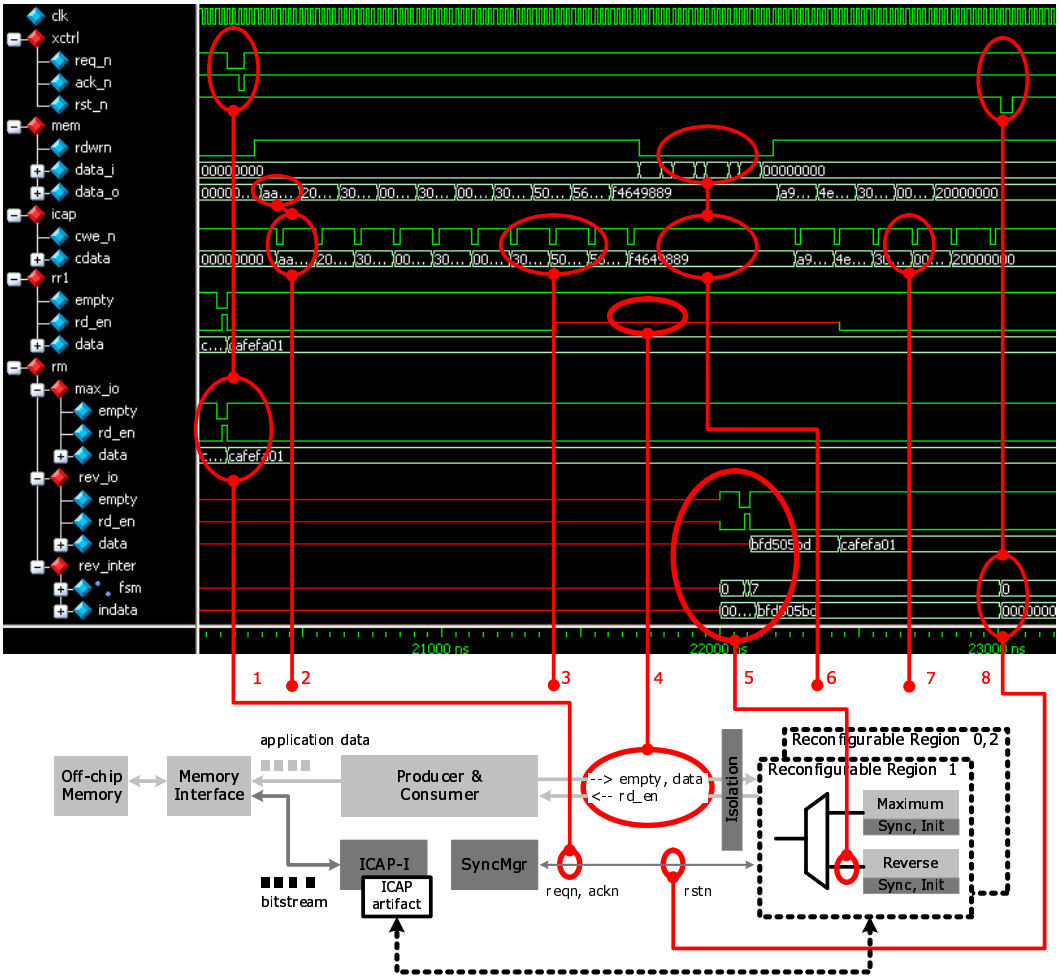


Figure 2.5: Waveform example for partial reconfiguration

- **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 [27], 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 2.4), we created a test plan, which described the list of scenarios and coverage items that were to be tested (see Figure 2.6 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.

	XDRS Test Plan	Description	Coverage Item(s)	Type
5	Isolation			
5.1	Code_Coverage	Code Coverage of isolation	/xdrs/isol_0	Code
5.4	Cancel_Static	Communication attempts from the static region are cancelled	cov_cancel_static	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 2.6: Extract of test plan and selected coverage items

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 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 2.7 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.

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

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

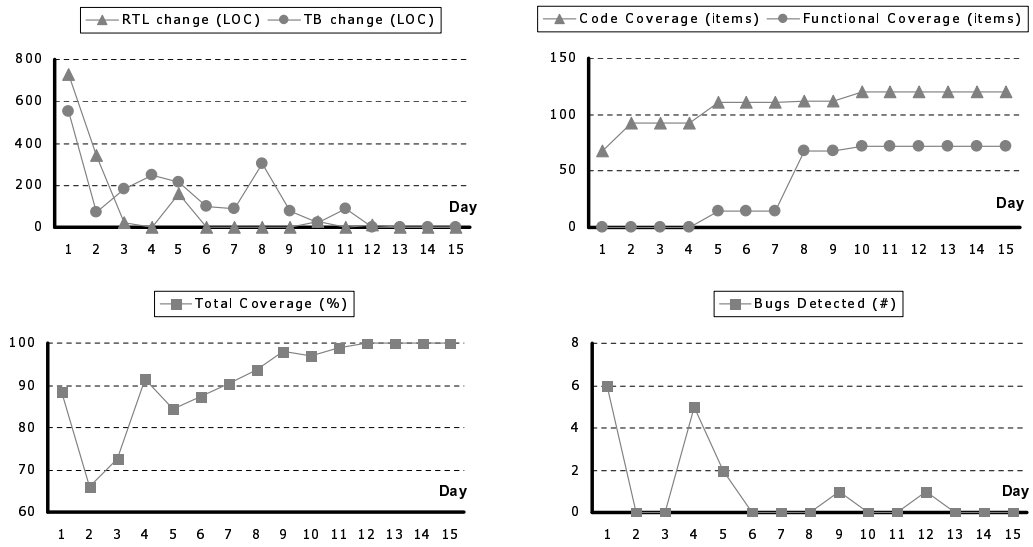


Figure 2.7: Coverage-driven verification progress with the XDRS system

- 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 2.6), 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.

2.1.1 Targeting a Second Application

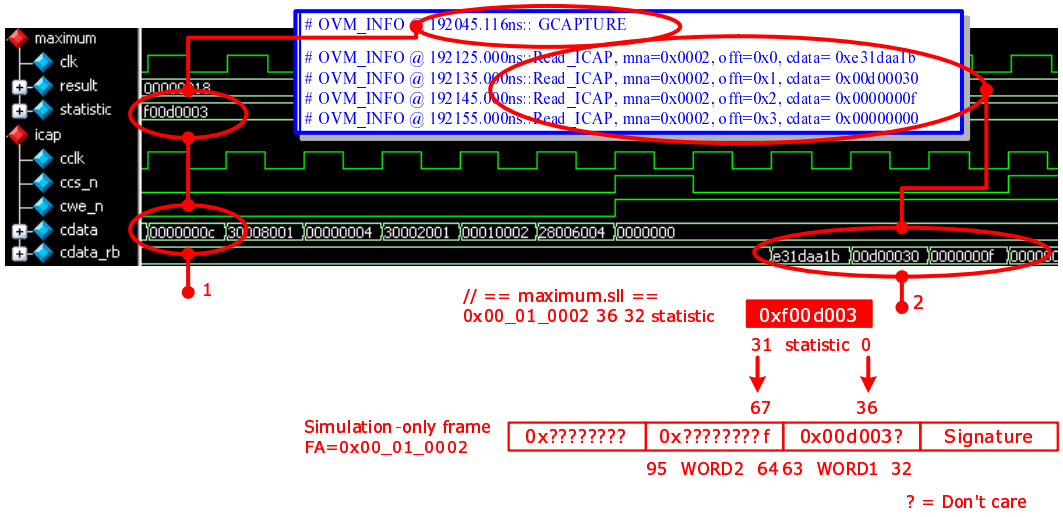
To demonstrate the benefit of applying the platform-based design methodology to the design and verification of the XDRS system, we mapped a second DRS application to XDRS. This second application demonstrated the use of ReSim to verify a DRS design that saves and restores module state in addition to 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. Table 2.2 provides a list of facts of the periodic application.

Table 2.2: Case study I fact sheet - (2)

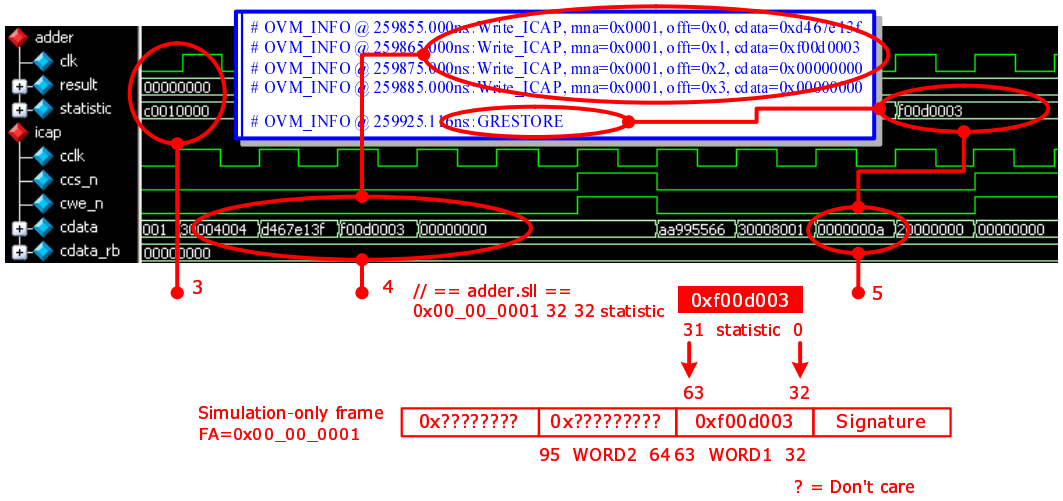
Case Study	Facts
Case Study Name	XDRS (Periodic Application)
DUT Features	Processor+reconfigurable cores Generic DRS computing platform Streaming and periodic applications
Similar Designs	Task Preemption [31] Module Relocation [18]
Target Board/FPGA	ML507/Virtex-5 FX70T
Verification Methods	Platform-based design, RTL simulation, HW/SW co-simulation, On-chip debugging
Verification Tools	ReSim, ChipScope
Reference	Not yet published[a]
	[a]It should be noted that the periodic application was derived from [13], and was re-designed in order to target the generic XDRS platform.
Released?	A simplified version of the XDRS core logic has been released (see the XDRS.QUICKSTART, XDRS.SINGLE, XDRS.MULTIPLE examples)

Figure 2.8 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

2.8-(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 2.8-(a) does not include signals of the `Adder` core, which are all injected with undefined "x" values. Similarly, Figure 2.8-(b) illustrates restoring the saved value to the newly configured `Adder` module.



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



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

Figure 2.8: Waveform example for state saving and restoration

In order to simulate state saving and restoration, the designers need to manually map RTL signals to simulation-only frames by modifying `.sll` files. 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 2.8-(a)), and the `adder/statistic` register is mapped to bit 63-32 of

frame 0x00000001 (see Figure 2.8-(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 2.8-(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.
- **@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). 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 23) 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, BUG-Example.XDRS.7 was missed by ReSim-based simulation. By inserting probing logic using ChipScope [36], 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.

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.

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.

2.2 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) [7]. 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 [23, 16, 7] 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. Table 2.3 provides a list of facts of the fault-tolerant DRS design.

Table 2.3: Case study II fact sheet

Case Study	Facts
Case Study Name	Fault-Tolerant DRS
DUT Features	Hardware-only system Fault-tolerant application
Similar Designs	Fault-tolerant car controller [23] Fault-tolerant soft processor core [16]
Target Board/FPGA	ML507/Virtex-5 FX70T
Verification Methods	RTL simulation, Coverage analysis On-chip debugging
Verification Tools	ReSim, ChipScope
Reference	Not yet published.
Released?	No

Comparing the system architecture between this case study and the XDRS platform (see Section 2.1), one significant difference is that the DUT of this case study does not contain microprocessors, buses or software. As illustrated by Figure 2.9, 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 [7]. 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).

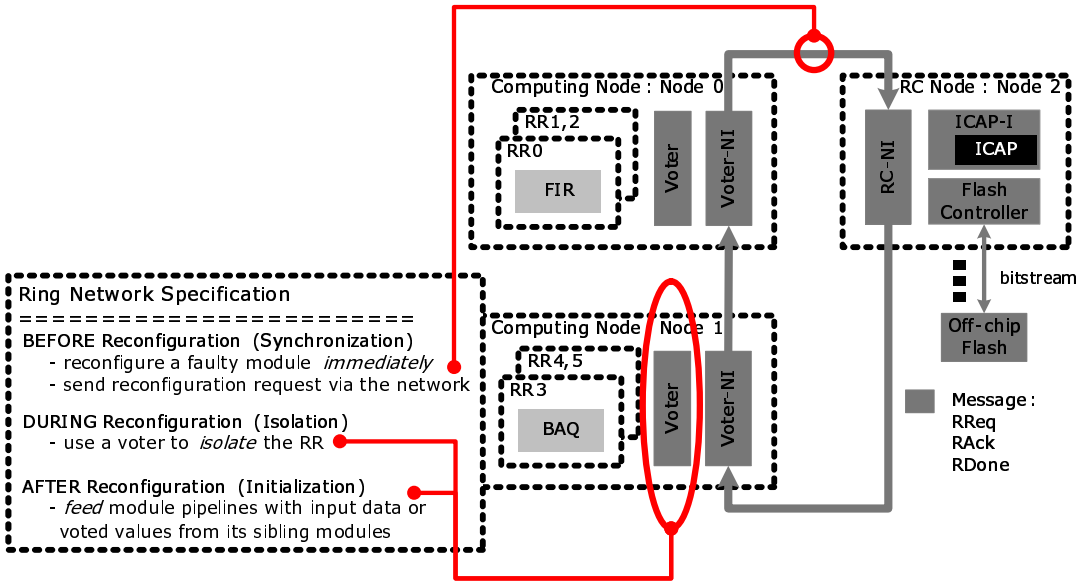


Figure 2.9: The hardware architecture of the fault-tolerant DRS

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 2.9). 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 [7].

Figure 2.10 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.

	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 2.10: Extract of the test plan section for the voter

Figure 2.11 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 2.1). Similar to the coverage-driven verification of the XDRS core logic (see Section 2.1), the total coverage of the fault-tolerant DRS also dropped as new design 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).

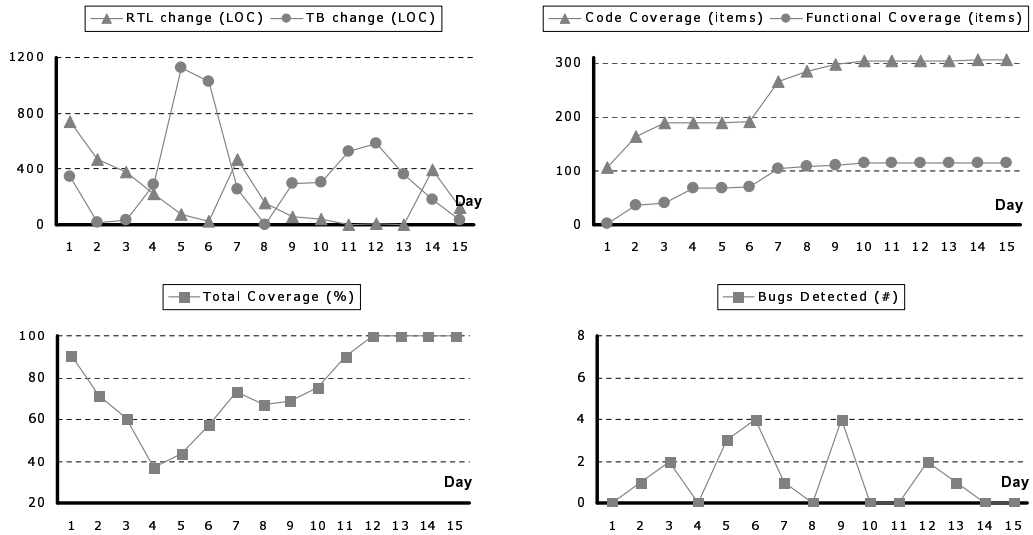


Figure 2.11: Simulation-based verification of the fault-tolerant DRS

- 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 2.11 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 [36], 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 2.4 in Section 2.1) 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 [10], 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.

[3]The coverage at Day 12 was 99.95%

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 [20] or DCS [27]. 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.

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 2.10).

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., [29]) 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 2.11, 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.

2.3 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 [9]. 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 [2]. As illustrated in Figure 2.12, 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.

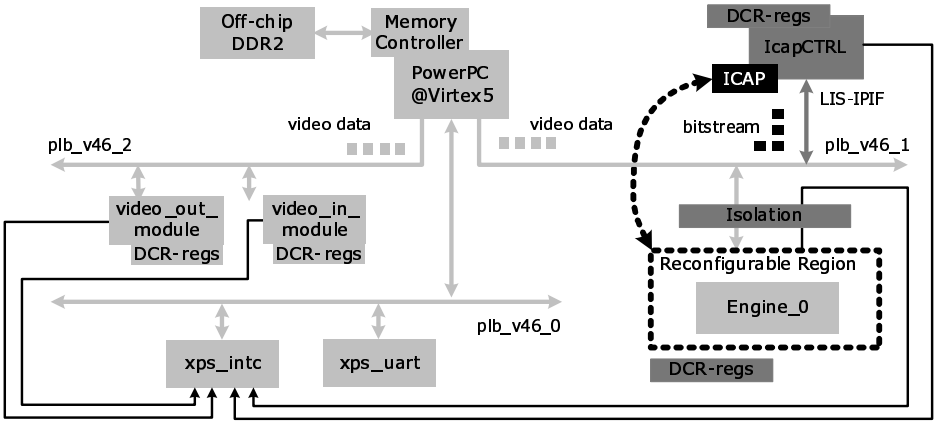


Figure 2.12: The hardware architecture of the Optical Flow Demonstrator

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 2.12) 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

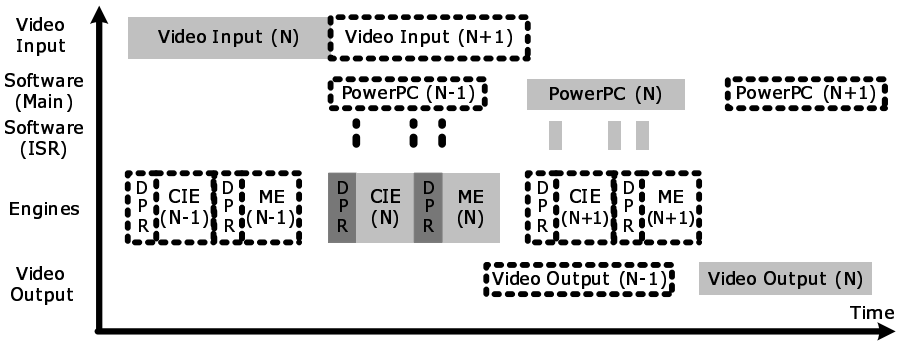


Figure 2.13: The processing flow of the Optical Flow Demonstrator

in the system (see Figure 2.13). 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 2.12). 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. Table 2.4 provides a list of facts of this case study.

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 2.14 illustrates the progress of development in terms of Lines of Code (LOC) changed and bugs detected per week.

- By the end of Week 3, the designer finished assembling the modified Optical Flow Demonstrator (see Figures 2.12 and 2.13) 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

Table 2.4: Case study III fact sheet

Case Study	Facts
Case Study Name	AutoVision
DUT Features	Processor+reconfigurable cores Video-based driver assistance system
Similar Designs	AutoVision [9] Optical Flow Demonstrator [2]
Target Board/FPGA	ML507/Virtex-5 FX70T
Verification Methods	RTL simulation, HW/SW co-simulation, IP-reuse
Verification Tools	ReSim, Virtual-Multiplexing
Reference	The case study has been published in [14]
Released?	No

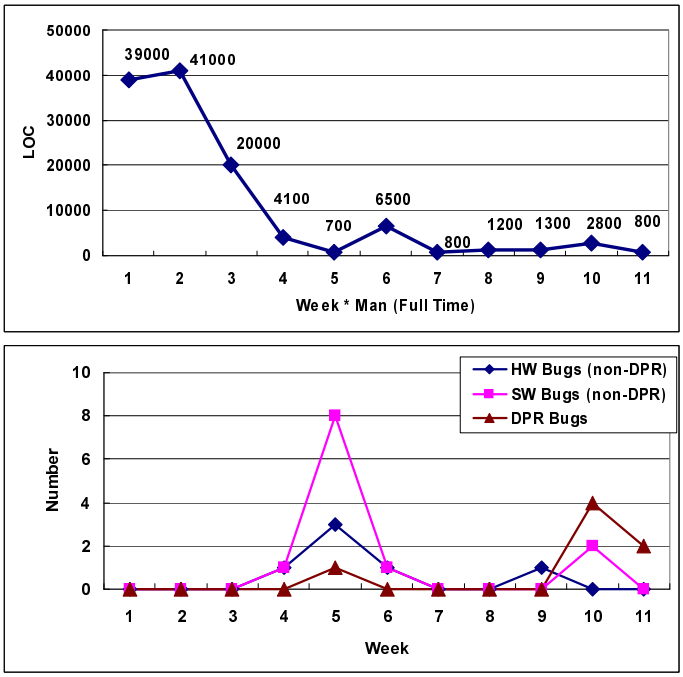


Figure 2.14: Development progress of the Optical Flow Demonstrator

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 [20], 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 32). 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,

- 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.

[4]Since BUG-Example.AUTO.1 was a false positive DPR bug, it is not counted in Figure 2.14

- 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 2.15 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.

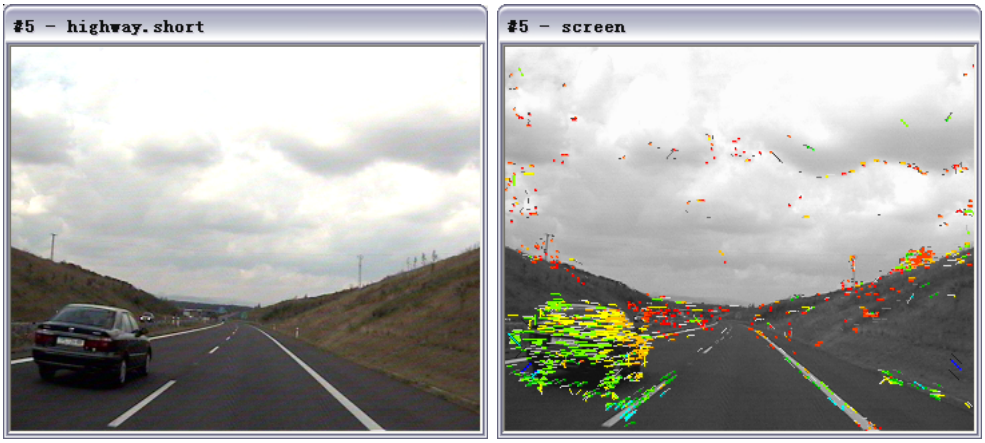


Figure 2.15: One frame of the input and the processed video (sensitivity parameter ε =20)

Table 2.5: 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 2.5. In the pipelined processing flow, the video engines are the bottleneck of the system throughput (see Figure 2.13). 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 [2]. For this case study and our host machine, the implementation and bitstream generation iteration took 52 minutes, and 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.

2.4 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 [32]. In the second case study, a processor dynamically reconfigures its peripheral modules for various math tasks [34]. Both reference designs target an ML605 board with a Virtex 6 LX240T FPGA. Tables 2.7 and 2.8 provide lists of facts of the reference designs.

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.

2.4.1 Case Study IV: Fast PCIe Reference Design

Figure 2.16 illustrates the block diagram of the Fast PCIe configuration (FPCIe) reference design [32]. 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

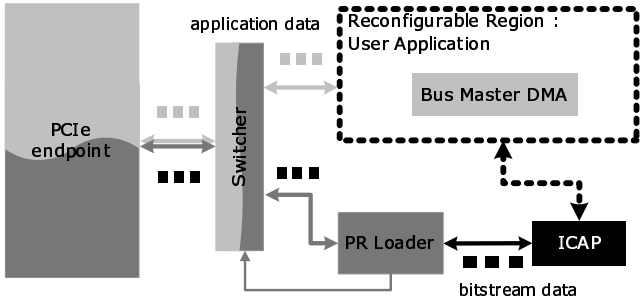


Figure 2.16: Fast PCIe configuration reference design, after [32]

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.

- 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 2.17 illustrates a screenshot of ReSim-based simulation.

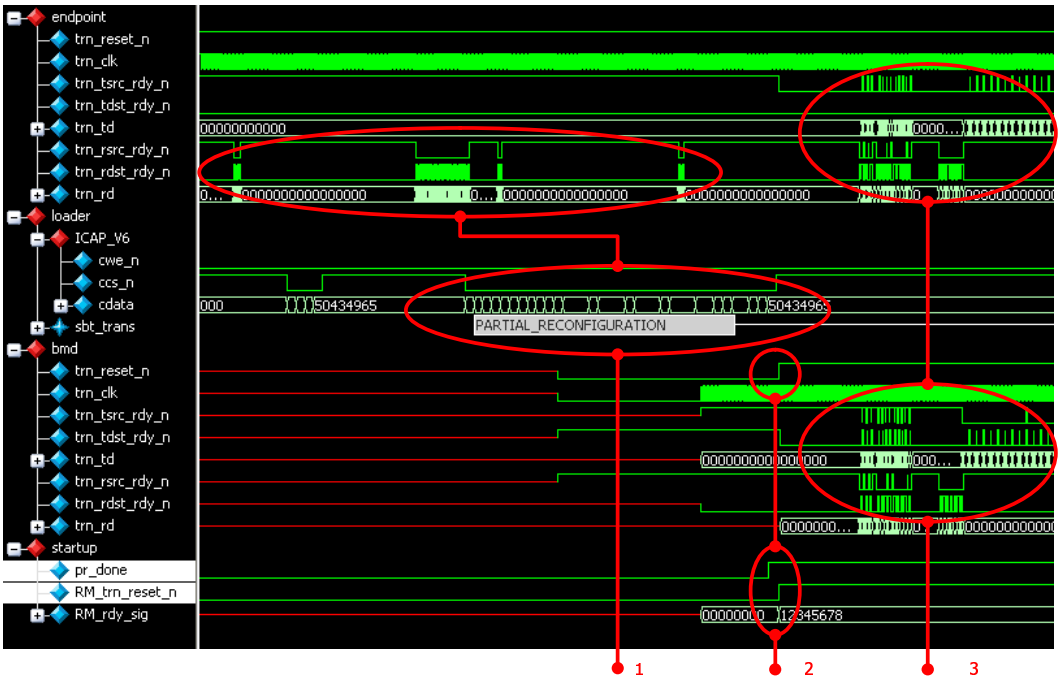


Figure 2.17: Simulating the FPCIe reference design

@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 2.6).

Table 2.6: 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 [32] 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.

Table 2.7: Case study IV fact sheet

Case Study	Facts
Case Study Name	Fast PCIe Configuration
DUT Features	Hardware-only design
Similar Designs	Derived from a vendor reference design
Target Board/FPGA	ML605/Virtex-6 LX240
Verification Methods	RTL simulation, Coverage analysis
Verification Tools	ReSim
Reference	The design is from XAPP883 [32], and the case study has been published in [12]
Released?	Yes (see the Fast_PCIe example)

2.4.2 Case Study V: Reconfigurable Peripheral Reference Design

Figure 2.18 illustrates the block diagram of the Reconfigurable Peripheral reference design [34]. 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.

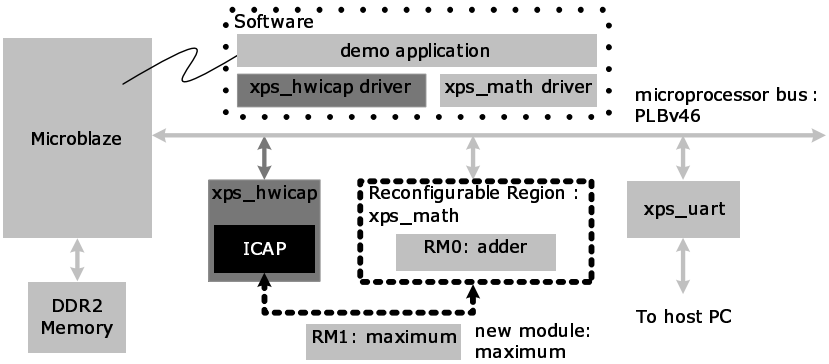


Figure 2.18: Reconfigurable Processor Peripheral reference design, after [34]

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 2.1.1). 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 2.19 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

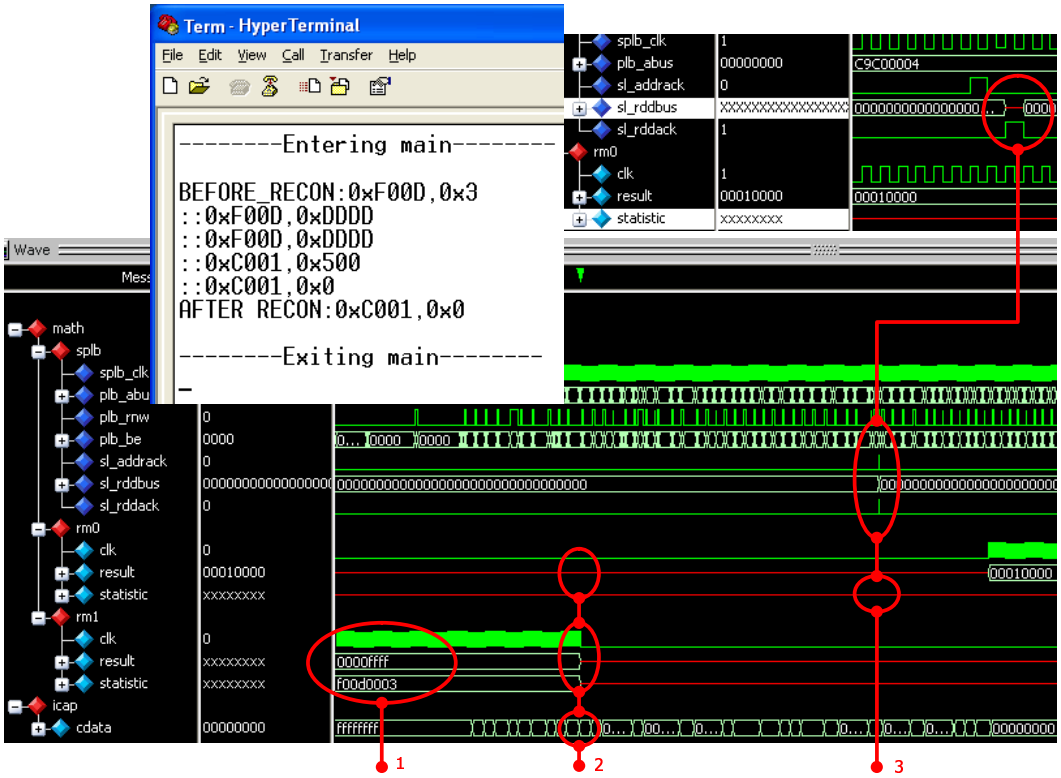


Figure 2.19: A *potential* isolation bug in the reference design

- **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 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 2.19 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.

Table 2.8: Case study V fact sheet

Case Study	Facts
Case Study Name	Processor Peripheral
DUT Features	Processor+reconfigurable cores
Similar Designs	Derived from a vendor reference design
Target Board/FPGA	ML605/Virtex-6 LX240
Verification Methods	RTL simulation, HW/SW co-simulation
Verification Tools	ReSim
Reference	UG744 [34]
Released?	A similar design has been released (see the STATE_MIGRATION example)

2.5 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 2.1 and 2.3). 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 2.1). 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 2.1, 2.2 and 2.4.1). 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 2.1). 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 2.2 and 2.4.1).

We used ReSim-based simulation in the context of platform-based design (see Section 2.1.1) and IP-reuse methodology (see Section 2.3). 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 2.9 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 2.2). We analyzed the bugs detected by ChipScope and found that all 4 bugs could easily have been detected via ReSim-based simulation.

Table 2.9: 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 a parameter script for each project, 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 2.3), we spent significant effort in setting up a baseline simulation environment. For the two reference designs (see Section 2.4), 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 2.1) and model functional coverage items (see Sections 2.1 and 2.2). It should be noted these overheads are not introduced by ReSim. In particular, 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 2.10 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 2.10: Summary of example bugs described in Chapter 2

Bug ID	Bug Tag	Section ...	Detected by ...
BUG-Example.XDRS.1	ICAPI_DONE	2.1	Extended ReChannel
BUG-Example.XDRS.2	Flush_Cache	2.1	**Extended ReChannel**
BUG-Example.XDRS.3	Multiple_Recon	2.1	**Extended ReChannel**
BUG-Example.XDRS.4	Isolation_Mismatch	2.1	**ReSim**
BUG-Example.XDRS.5	Block_Until_Idle	2.1	ReSim
BUG-Example.XDRS.6	Restoration_Pointer	2.1.1	**ReSim**
BUG-Example.XDRS.7	Wrong_Pad_Word	2.1.1	On-chip Debugging
BUG-Example.FT.1	Level_Clock	2.2	On-chip Debugging Could have been detected by ReSim
BUG-Example.FT.2	Feed_Pipeline	2.2	**ReSim**
BUG-Example.FT.3	Too_Quick_Recon	2.2	**ReSim**
BUG-Example.AUTO.1	Sig_Reg	2.3	Virtual Multiplexing
BUG-Example.AUTO.2	IcapCtrl_PLB	2.3	**ReSim**
BUG-Example.AUTO.3	Driver_Update	2.3	**ReSim**
BUG-Example.AUTO.4	Engine_Reset	2.3	**ReSim**
BUG-Example.UG744.1	Bus_Isolation	2.4.2	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 2.10 (see detailed descriptions of each bug in relevant Sections). Other bugs could also be detected with previous methods such as Virtual Multiplexing [20] and DCS [27]. 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 [27]. However, since ReSim models characteristic features of DPR, it is able to detect more bugs missed by previous methods (see Section 2.3). 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 Chapter 3.

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 2.3). 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. Given the conflicting requirements of simulation accuracy and verification productivity, 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 3

Bugs Detected in Case Studies

This appendix describes DPR-related bugs that were detected in the case studies described in Chapter 2 (see Table 2.9). 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 [25], Virtual Multiplexing [20] and DCS [27], 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.

3.1 Case Study I: In-house DRS Computing Platform

For the XDRS case study (see Section 2.1), 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 2.2).

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 readback 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 [27], 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.

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

As described in Section 2.2, 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 2.2) 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 2.11).

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 2.10).
  - 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

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

As described in Section 2.3, 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 2.14). 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

3.4 Case Study IV & V: Vendor Reference Designs

As described in Section 2.4, 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 2.4.2).

Bibliography

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

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

[3] *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

[4] *Quartus II Handbook Version 12.1, Volume 1: Design and Synthesis*, Altera Corporation, 2012. [Online]. Available: http://www.altera.com/literature/lit-qts.jsp

[5] 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.

[6] K. J. Brent Welch and J. Hobbs, *Practical Programming in Tcl and Tk (4th Edition)*. Prentice Hall, 2003.

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

[8] 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.

[9] 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.

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

[11] L. Gong and O. Diessel, "Modeling Dynamically Reconfigurable Systems for Simulation-based Functional Verification," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2011, pp. 9 – 16.

[12] ——, "ReSim: A Reusable Library for RTL Simulation of Dynamic Partial Reconfiguration," in *Field-Programmable Technology (FPT), International Conference on*, 2011, pp. 1 – 8.

[13] ——, "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.

[14] L. Gong, O. Diessel, J. Paul, and W. Stechele, "RTL Simulation of High Performance Dynamic Reconfiguration: A Video Processing Case Study," in *Parallel and Distributed Processing, International Symposium on, Reconfigurable Architectures Workshop (RAW)*, 2013, pp. 1 – 8.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] W. Luk, N. Shirazi, and P. Y. Cheung, "Compilation Tools for Run-time Reconfigurable Designs," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 1997, pp. 56 – 65.

[21] M. Lutz, *Programming Python (4th Edition)*. O'Reilly Media, 2011.

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

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

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

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

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

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

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

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

[41] *7 Series FPGAs Configuration User Guide (UG470)*, Xilinx Inc., 2013.