ReSim User Guide

A Guide to RTL *Sim*ulation of
Dynamically *Re*configurable 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

Revision History i

List of Figures v

List of Tables vii

List of Abbreviations viii

Preface 1

0.1 System Requirements . . . 2

0.2 Expected Audience . . . 2

0.3 Documentation . . . 2

0.4 Acknowledgments . . . 3

0.5 Contact Us . . . 3

1 Introduction to ReSim 4

1.1 Dynamic Partial Reconfiguration . . . 4

1.2 Simulating Dynamic Partial Reconfiguration . . . 5

2 Common Usages of ReSim 9

2.1 Simulating Various DRS Design Styles . . . 9

2.2 Detecting DPR-related Bugs . . . 12

2.3 Success Stories . . . 13

2.4 Current Limitations . . . 14

3 Getting Started with ReSim **16**

3.1 Design Flow Overview . . . 16

3.2 Preparing the Design/Testbench . . . 17

3.3 Generating Artifacts . . . 19

3.3.1 Modifying the Generated Artifacts (Optional) . . . 20

3.4 Running Simulation . . . 21

4 Reference Designs **23**

4.1 XDRS.QUICKSTART . . . 23

4.2 XDRS.SINGLE . . . 28

4.3 XDRS.MULTIPLE . . . 33

4.4 Fast PCIe Example Design . . . 33

4.5 State Migration Example Design . . . 36

5 Inside the ReSim Library **42**

5.1 Library Overview . . . 42

5.1.1 Transaction-based Communication . . . 44

5.2 ReSim Built-in Artifacts . . . 46

5.2.1 CP Class . . . 46

5.2.2 RR Class . . . 47

5.2.3 Simulation-only Layer Class . . . 51

5.2.4 Scoreboard Class . . . 51

5.3 ReSim-Generated Artifacts . . . 52

5.3.1 CP Wrapper . . . 52

5.3.2 RR Wrapper . . . 53

5.3.3 Simulation-only Layer Wrapper . . . 55

5.3.4 Simulation-only Bitstreams . . . 55

5.3.4.1 Simulation-only Bitstream for Configuring a New Module . . . 55
5.3.4.2 Simulation-only Bitstream for Saving and Restoration . 57
5.3.5 Simulation-only Logic Allocation Files . . . 59
5.3.6 SystemVerilog Package . . . 59
5.3.7 SystemVerilog Interfaces . . . 60
5.3.8 TODO-LIST File . . . 60
5.3.9 Report File . . . 60
5.3.10 Derived Class . . . 60
5.4 ReSim APIs . . . 61

Bibliography **68**

List of Figures

1.1 Conceptual diagram of a DRS design 4

1.2 Using the simulation-only layer 6

2.1 Simulating partial reconfiguration 9

2.2 Simulating configuration readback 10

3.1 ReSim tool flow 16

3.2 The simulation environment for ReSim-based simulation - (1) 17

3.3 The simulation environment for ReSim-based simulation - (2) 18

3.4 An example of a parameter script 20

4.1 The XDRS reference design 24

4.2 Simulating the XDRS.QUICKSTART reference design 27

4.3 Selected testplan sections of the XDRS.SINGLE reference design 29

4.4 Coverage results of the XDRS.SINGLE reference design 32

4.5 Fast PCIe configuration reference design 34

4.6 Simulating the FPCIe reference design 35

4.7 The State Migration reference design 36

4.8 Simulating the State Migration reference design 39

5.1 ReSim-based RTL simulation (similar to Figure 1.2 but with more details) 42

5.2 ReSim transactions 44
5.3 Timing of ReSim transactions (partial reconfiguration) 45
5.4 Timing of ReSim transactions (configuration readback) 45
5.5 The configuration port class 46
5.6 The portal controller and the error injector class 48
5.7 The state spy class 50
5.8 The scoreboard class 51
5.9 The CP wrapper generated by ReSim 52
5.10 The RR wrapper generated by ReSim 53
5.11 The organization of the spy memory 55
5.12 The simulation-only layer wrapper generated by ReSim 56

List of Tables

2.1 Differences between the Virtex-5 FPGA fabric and the simulation-only layer 13

2.2 Summary of case studies 14

5.1 An example SimB for configuring a new module 56

5.2 An example SimB for state saving 57

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 [25][2]. 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 [21].

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 [25]. 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 [3]. 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 the user guide of 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
- Common Usages of ReSim (see Chapter 2): Describes the applications and the limitations of ReSim
- Getting Started with ReSim (see Chapter 3): A step by step guide on using the ReSim library
- Reference Designs (see Chapter 4): Describes a few reference designs/testbenches
- Inside the ReSim library (see Chapter 5): Describes the implementation details of the ReSim library

0.1 System Requirements

ReSim has been tested using QuestaSim/ModelSim 6.5g [16] 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) [15]. The State Migration has only been tested on EDK 12.4 [24].

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:

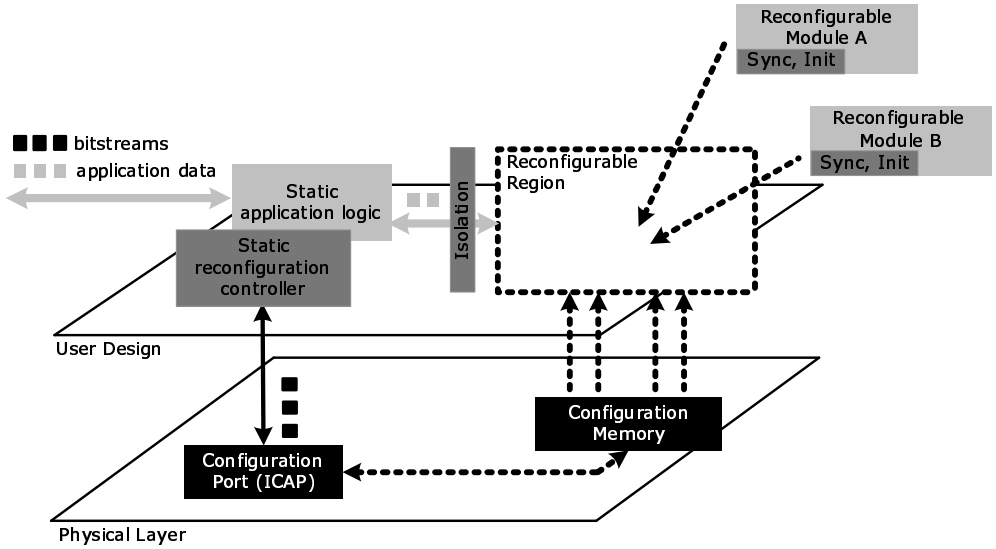


Figure 1.1: Conceptual diagram of a DRS design

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,

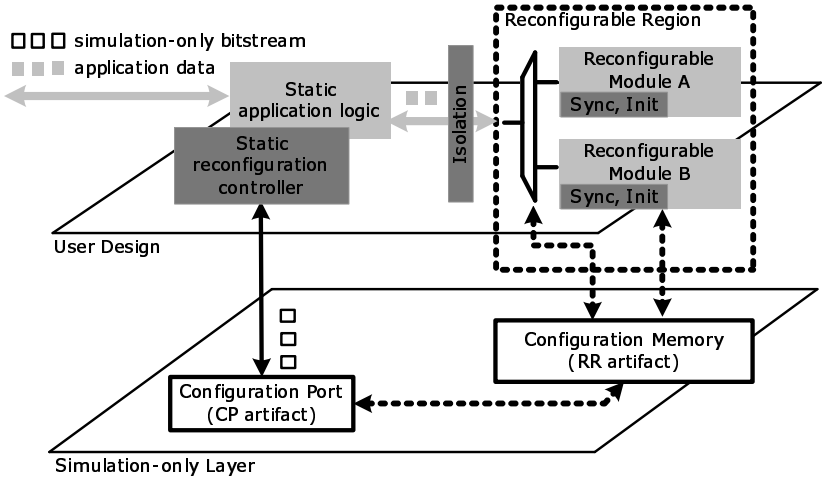


Figure 1.2: Using the simulation-only layer

- 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., [23, 26, 27, 28]).
  - 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 [24]. 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

Common Usages of ReSim

2.1 Simulating Various DRS Design Styles

Using a simulation-only layer, modular reconfiguration and configuration readback can be simulated in the desired cycle-accurate manner. The user-defined reconfiguration controller reads/writes SimBs from/to the simulated CP artifact as if it were reading/writing a real configuration port. The RR artifact uses RRIDs/RMIDs instead of Frame Addresses to drive module swapping and to save and restore state data of simulated RMs. ReSim can also be used to simulate non-modular reconfigurable DRS designs. We describe the use cases of ReSim as follow:

Simulating Modular Reconfiguration. Using a simulation-only layer, reconfiguration is simulated as follows (see Figure 2.1):

Entry	SimB	Explanation	Actions Taken
1	0xAA995566	SYNC Word	Start the **DURING Reconfiguration** phase
2	0x20000000	NOP	–
3	0x30002001 0x01020000	Type 1 Write FAR FA=0x01020000	Select the module id=0x02 to be the next active module in reconfigurable region id=0x01
4	0x30008001 0x00000001	Type 1 Write CMD WCFG	
5	0x30004000 0x50000010	Type 2 Write FDRI Size=16	Pad frame: not required Configuration data: 16 words (4 frames)
6	0x5650EEA7 0xF4649889 ... 0xA9B759F9 0x4E438C83	SimB Frame 0 Word 0 SimB Frame 0 Word 1 ... SimB Frame 3 Word 2 SimB Frame 3 Word 3	Module signature; Module state data; Enable/Disable error injection
7	0x30008001 0x0000000D	Type 1 Write CMD DESYNC	End the **DURING Reconfiguration** phase

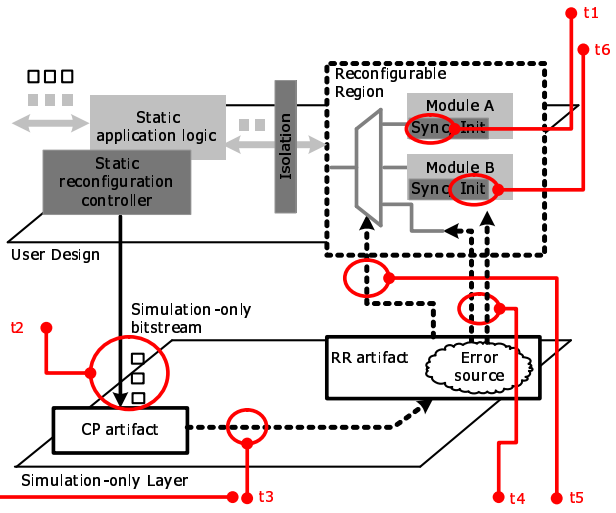


Figure 2.1: Simulating partial reconfiguration

@t1: The static region *synchronizes/pauses/disables* the outgoing RM (i.e., module A) so as to ensure no pending operations between the RM and the static region.

@t2: The user-defined reconfiguration controller *transfers* a partial SimB instead of a real bitstream to the CP artifact.

@t3: The CP artifact parses the SimB and extracts the numerical ID of the module to be swapped in.

@t4: While the SimB is being written, the RR artifact injects errors to both the static region and the RM. Errors injected to the static region model the spurious outputs of the module undergoing reconfiguration and help to test the *isolation* logic of the user design. Errors injected to the reconfigurable modules model the undefined initial state of the RM and help to test the initialization mechanism of the design.

@t5: After the SimB has been completely written to the CP artifact, the RR artifact checks the signature of the SimB and drives module swapping according to the numerical ID of the target module (see t3).

@t6: The static region *initializes* the new RM (i.e. module B) so as to clear the undefined initial RM state.

Simulating State Saving and Restoration. The more complex processes of sate saving and restoration, as can be achieved by reading, writing and processing configuration bitstreams in real FPGAs, can also be emulated using the simulation-only layer. For example, saving the state of a RM is simulated as follows (see Figure 2.2):

Entry	SimB	Explanation	Actions Taken
1	0xAA995566	SYNC Word	Start the **DURING Reconfiguration** phase
2	0x30008001 0x0000000C	Type 1 Write CMD GCAPTURE	Copy the state bits from HDL signals to the spy memory
3	0x30002001 0x00010002	Type 1 Write FAR FA = 0x00010002	Read configuration data starting from frame with FA=0x00010002 (RRid=0; RMid=1; MNA=2)
4	0x30008001 0x00000004	Type 1 Write CMD RCFG	
5	0x28006000 0x50000004	Type 2 Read FDRO Size=4	Pad frame: not required Configuration data: 4 words (1 frame)
6	0xE31DAA1B 0x00D00030 0x0000000F 0x00000000	SimB Frame 2 Word 0 SimB Frame 2 Word 1 SimB Frame 2 Word 2 SimB Frame 2 Word 3	Configuration data are returned to the user design
7	0x30008001 0x0000000D	Type 1 Write CMD DESYNC	End the **DURING Reconfiguration** phase

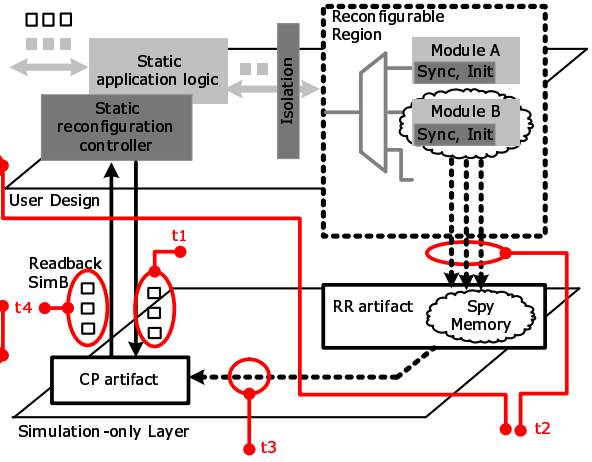


Figure 2.2: Simulating configuration readback

@t1: The static module transfers a readback SimB instead of a real bitstream to the CP artifact. By parsing the SimB, the CP artifact extracts the readback parameters such as the numerical ID of the module to be read.

@t2: The readback SimB has a GCAPTURE command, which informs the RR artifact to probe the RM state (i.e., values of RTL signals that represent state elements such as flip-flops and memory cells) and to store the state data in a `spy_memory`. RTL signals of simulated modules are mapped to simulation-only frames of the RR artifact using a user-defined simulation-only logic allocation file (i.e., a `.sll` file), which is functionally equivalent to the logic allocation file (`.ll` file) generated by vendor tools.

@t3: The state data of the RM of interest are returned to the CP artifact.

@t4: The configuration port is switched to read mode and the CP artifact returns the retrieved state data to the user design as a readback SimB.

Restoring the state of a RM also utilizes the artifacts and SimBs to mimic the behavior of the FPGA fabric. However, a different SimB from the one for state saving is used, and the state data are copied back to the RTL signals of the simulated RM.

The simulation-only layer only maps RTL signals of RMs to simulation-only frames. To simulate state saving and restoration of flip-flops/memory cells of a static module, the designer has to change the static module to be a reconfigurable region for simulation purposes. As long as the modified system does not perform reconfiguration, it is functionally equivalent to the original one.

Simulating Module Relocation. To simulate DRS designs that perform module relocation, the simulated user design relocates a simulated RM by updating the RRID/RMID fields of a SimB. In particular, the simulated user design reads a readback SimB from the CP artifact, updates the RRID/RMID fields, and writes the updated SimB to the CP artifact. The simulated RM instantiated in the target RR is then activated accordingly, as if it were relocated from the original location.

Simulating Fine-grained Reconfiguration. To simulate the process of updating LUTs for fine-grained reconfigurable designs, the designer can model the content of a LUT as a memory cell and map the memory cell to the simulation-only frame using the `.sll` file. During simulation, the simulated user design updates the LUT by reconfiguring a SimB that changes the memory cell of the LUT.

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

2.2 Detecting DPR-related Bugs

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

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

However, the simulation-only layer is not exactly the same as the FPGA fabric. Table **??** lists the differences between a Virtex-5 FPGA, as an example of a target device, and our simulation-only layer, representing a fabric-independent FPGA. The mismatches between the two can lead to bugs that remain undetected using the simulation-only layer (i.e., False Negative bugs) and bugs that are incorrectly reported using the simulation-only layer (i.e., False Positive bugs). These bugs, which could be categorized as being *fabric-dependent*, include:

- Errors in the bitstream itself (e.g., setting an incorrect Frame Address in the bitstream, single or multiple bit flips in the bitstream), and
- Errors in interpreting the content of the bitstream (e.g. accessing an incorrect state bit in a bitstream).

Table 2.1: Differences between the Virtex-5 FPGA fabric and the simulation-only layer

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

In theory, fabric-dependent bugs would not be introduced to modular reconfigurable DRS designs created by vendor tools, and ReSim can thereby provide assistance in verifying modular reconfigurable DRS designs. On the other hand, if a system is designed to directly modify or generate bitstreams at run time, ReSim can only offer limited help to test and verify it. Furthermore, ReSim does not nor does it aim to provide assistance in verifying implementation-related bugs, such as:

- Timing violation errors in the placed and routed design, and
- Possible short or open circuits, if any, caused by partial reconfiguration [4].

2.3 Success Stories

We demonstrate the value of ReSim and ReSim-based functional verification via a number of case studies. The first is an *in-house*, processor-based DRS computing platform, which is similar to existing generic slot-based DRS platforms such as the Erlangen Slot Machine [5], the VAPRES streaming architecture [13] and the Sonic video processing system [19]. 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 [7], 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 two vendor reference designs: The fast PCIe configuration reference design applies partial reconfiguration to meet the tight PCIe startup timing requirement [20]. The processor-based reference design dynamically reconfigures

Table 2.2: 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	–

its peripheral modules for various math tasks [22]. Overall, we aim to demonstrate that ReSim is flexible enough to simulate various DRS design styles.

Table 2.2 summarizes the case studies. We summarize these case studies in terms of the development workload[1], simulation overhead[2] and DPR-related 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. The table also does not include false positive bugs 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. We analyzed the bugs detected by ChipScope and found that all 4 bugs could easily have been detected via ReSim-based simulation. Details of these case studies can be found in [10].

2.4 Current Limitations

As described in Section 2.2, the primary limitation of ReSim is that ReSim can only offer limited assistance in detecting *fabric-dependent* bugs. Meanwhile, limited by the complexity of implementing ReSim, ReSim only supports

- ModelSim/QuestaSim 6.5g (thoroughly tested) or above (in theory),

[1]The Lines of Code (LOC) of designs only includes the reconfiguration machinery.
[2]Simulation overhead was measured from the profiling results of ModelSim.

- Virtex 4,5,6 FPGAs,
- The basic operations of ICAP: 32-bit ICAP, configuration write, configuration readback (including readback busy), basic ICAP registers (i.e., CRC, CMD, FAR, FDRI, FDRO, IDCODE) and basic ICAP commands (NULL, WCFG, RCFG, RCRC, GRESTORE, GCAPTURE, DESYNC), and
- For VHDL designs, errors can only be injected to RMs via their IO signals. For Verilog designs, errors can be injected to internal signals of RMs directly.
- Saving and restoring flip-flop/register values via the configuration port.
- State restoration by modifying the state bit (i.e. `INIT0/INIT1` bit) of a bitstream.

ReSim does not yet support,

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

Chapter 3

Getting Started with ReSim

3.1 Design Flow Overview

Figure 3.1 illustrates the development flow of ReSim-based simulation. Files that are created by the designer are marked by a "pen" symbol. ReSim-based simulation can be performed by three steps: preparing the design/testbench, generating artifacts and running simulation.

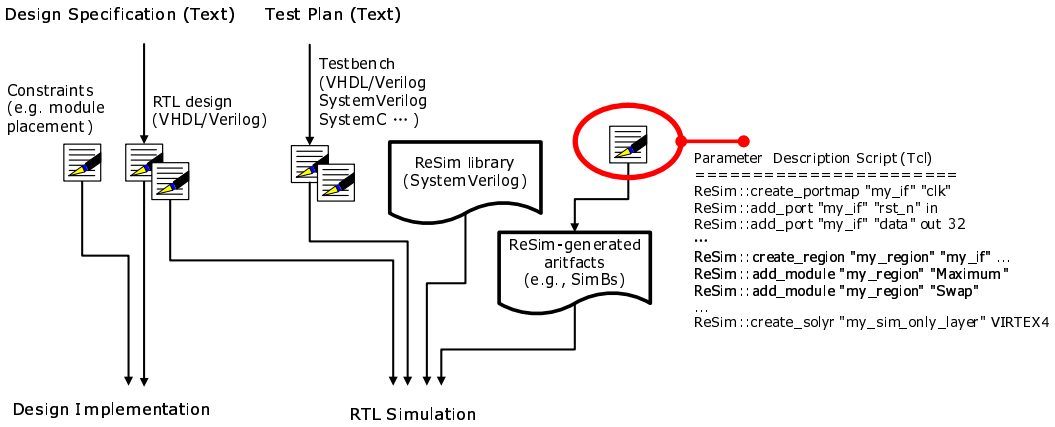


Figure 3.1: ReSim tool flow

- Preparing the design/testbench: Using ReSim-based simulation, the designers need to reserve a few "holes" in the design and the testbench so that the simulation-only artifacts generated by step 2 can be correctly "plugged" into the simulation environment (see Section 3.2).
- Generating artifacts: The second step is to create a parameter script that describes DPR-related parameters of the system. ReSim then automatically generates the source code for artifacts based on this script. The generated artifacts can be directly used with the ReSim libraries. Alternatively, the designer can edit the generated artifacts for design-/test-specific needs (see Section 3.3).

- Running simulation: After generating and modifying the artifacts, the designers can compile the source code for the design/testbench, the ReSim library and the generated artifacts to run simulation (see Section 3.4).

3.2 Preparing the Design/Testbench

Figures 3.2 illustrates a typical top-level testbench for ReSim-based simulation. The testbench instantiates

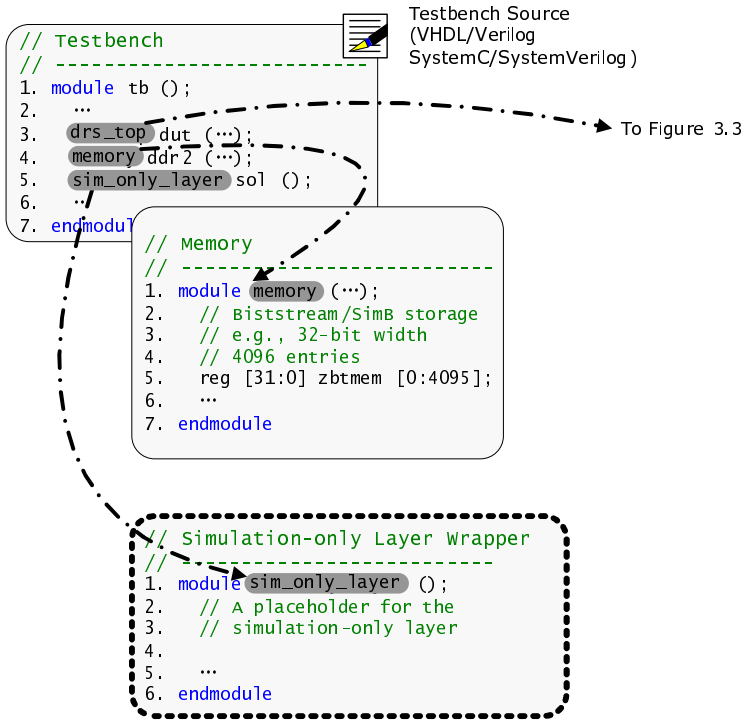


Figure 3.2: The simulation environment for ReSim-based simulation - (1)

- A DRS design (Line 3 of `tb`): See Figure 3.3.
- A simulation model for the off-chip memory (i.e., the `memory` module @ Line 4 of `tb`): Used to store bitstreams and/or application data, and can, in general, be modeled by a two-dimension array. The memory model is, in general, provided by the memory vendor but can also be created from scratch.
- The simulation-only layer wrapper (i.e., the `sim_only_layer` module @ Line 5 of `tb`): A top-level container for code that parameterizes the simulation-only layer. It has no input or output, and is automatically generated by ReSim.

A more comprehensive testbench can also include complex components such as stimuli generator, response checker and coverage analysis modules, but is out of the scope of this document.

Figure 3.3 illustrates an example of the DRS design. The DRS has

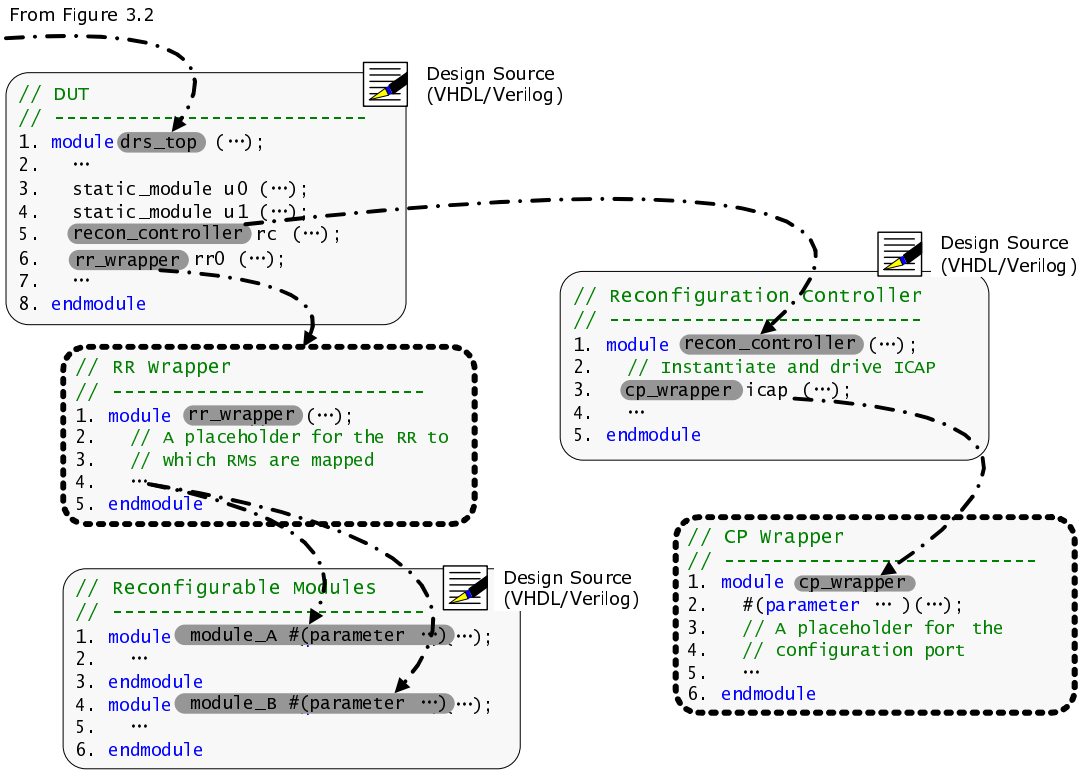


Figure 3.3: The simulation environment for ReSim-based simulation - (2)

- Static modules `u0`, `u1` ... (Line 3-4 of `drs_top`)
- A reconfiguration controller (i.e., the `recon_controller` module @ Line 5 of `drs_top`): Used to transfer bitstreams to the configuration port. It instantiates
  - A CP wrapper (i.e., the `cp_wrapper` module @ Line 3 of `recon_controller`): Used as a placeholder for the configuration port (e.g., ICAP). It has the same IO signals as the configuration port. ReSim automatically generates the source code for the CP wrapper.
- An RR wrapper (i.e., the `rr_wrapper` module @ Line 6 of `drs_top`): Used as a placeholder for the RR to which RMs (i.e., `module_A` and `module_B`) are mapped. It has the same IO signals as the RMs mapped to it. ReSim automatically generates the source code for the RR wrapper.

A DRS design can also instantiate its reconfiguration controller deep down in the design hierarchy or use external ones, and it may also have multiple reconfigurable regions.

As illustrated by Figures 3.2 and 3.3, it is essential for designers to keep the placeholders for ReSim-generated artifacts so that the artifacts can be correctly "plugged" into. The next step is to create a parameter script in order to generate these artifacts.

3.3 Generating Artifacts

In order to reuse the ReSim library for various DRS design styles, designers need to parameterize the library with design-specific information. DPR-related parameters include,

- The interfacing signals crossing the RR boundary, which are used to generate an RR wrapper that can correctly interleave these signals.
- The topology (i.e., the affiliation of RMs and RRs) of the user design, which are required to assign numerical IDs to each RM and RR and to parameterize SimBs with the RRID/RMID of target modules.
- The number of simulation-only frames for each RR, which is used to determine the length of the generated SimBs.
- Derived artifacts associated with each RR, which are used to generate templates for derived classes (see Section 3.3.1).
- The target FPGA family, which is used to generate a correct CP wrapper.

ReSim provides a set of Tcl APIs to help designers integrate the simulation-only layer into their design-specific simulation environment. Figure 3.4 illustrates an example of a parameter script. By calling APIs in a Tcl script, the user describes the interfacing signals crossing the RR boundary (Line 3-6), the affiliation of RMs and RRs (Line 8-10), the size and the derived artifacts of RRs (Line 8), and the FPGA family used by the design (Line 12). ReSim generates the parameterized artifacts that can then be correctly instantiated in the design hierarchy. The artifacts include:

- The CP wrapper (e.g., the `cp_wrapper` module, see Section 3.2)
- The RR wrapper (e.g., the `rr_wrapper` module, see Section 3.2)
- The simulation-only layer wrapper (e.g., the `sim_only_layer` module, see Section 3.2)
- Simulation-only Bitstream files: In both binary format and memory format loadable to ModelSim, in the `sbt` directory (see Figure 3.4).
- Simulation-only logic allocation files: In the `spy` directory (see Figure 3.4).
- A SystemVerilog Package file
- A SystemVerilog Interface file
- A TODO-LIST file
- A Report file
- Templates for derived classes: See Section 3.3.1.

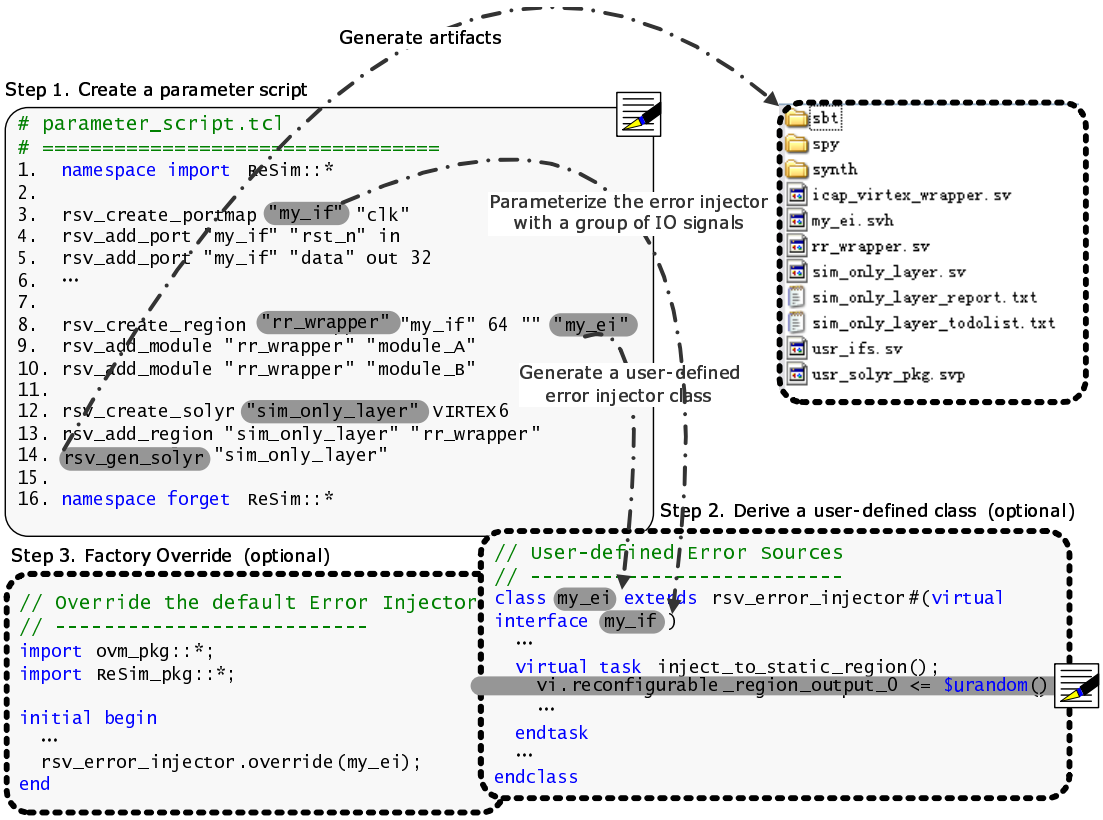


Figure 3.4: An example of a parameter script

3.3.1 Modifying the Generated Artifacts (Optional)

For ReSim-based RTL simulation, the generated artifacts are ready to be used directly. As an option, the designer can then edit the generated artifacts for design-/test-specific needs. There are two typical usage scenarios that require modifying the generated artifacts for ReSim-based RTL simulation.

Scenario 1: Overriding the default class-based artifacts. The example (see Step 2 in Figure 3.4) illustrates changing the default "x" injection to a user-defined error injector that drives random values to the static region. Such extension is implemented by redefining the *virtual* functions in the derived classes (see Figure 3.4). Using the factory mechanism provided by the OVM library [9], the default error injector can be easily overridden with the derived `my_ei` class. Furthermore, the Tcl script automatically generates code (see Step 3 in Figure 3.4) that performs the factory override operation. Designers can thus focus on defining the virtual functions in their derived classes without bothering about how to integrate the derived classes. Artifacts that can be overridden include:

- Device-specific reconfiguration interface: Can be derived to model various interfacing protocols of possible configuration ports.

- Generic SimB parser: Can be derived to parse real bitstreams.
- Error injector: Can be derived to define design-/test-specific error sources.
- Scoreboard: Can be derived to define design-/test-specific coverage items.

Scenario 2: Mapping RTL signals to simulation-only configuration frames. The parameter script generates simulation-only logic allocation files (`.sll` files). The `.sll` files are used to map RTL signals of the user design to the `spy_memory`, which implements simulation-only frames that mimic the configuration frames of the FPGA fabric. The generated `.sll` files are empty by default. For DRS designs that saves and restores module state via the configuration port, the designer needs to modify the generated `.sll` files and explicitly specify the names of RTL signals and the simulation-only frame addresses the signals are mapped to.

3.4 Running Simulation

Since the user design/testbench has kept placeholders for ReSim-generated artifacts, the generated/modified artifacts are ready to be integrated to the simulation environment. As described by Section 3.2, the designer needs to

- Instantiate the generated CP wrapper as part of the reconfiguration controller.
- Instantiate the generated RR wrappers as part of the top-level user design.
- Instantiate the simulation-only layer wrapper in the testbench.

The designer then compiles all design source and starts simulation as follow,

- Compile the ReSim library and the generated artifacts

```
# Compile ReSim library
ModelSim> vlog ...  $RSV_HOME/src/rsv_ifs.sv
ModelSim> vlog ...  $RSV_HOME/src/rsv_solyr_pkg.svp
# Compile ReSim-generated artifacts
ModelSim> vlog ...  ./artifacts/usr_ifs.sv
ModelSim> vlog ...  ./artifacts/usr_solyr_pkg.svp
ModelSim> vlog ...  ./artifacts/rr_wrapper.sv
ModelSim> vlog ...  ./artifacts/cp_wrapper.sv
ModelSim> vlog ...  ./artifacts/sim_only_solyr.sv
```

- Run simulation

```
# Load simulation
ModelSim> vsim -L mtiReSim -permit_unmatched_virtual_intf ...
```

- Load the generated SimBs into the memory of the simulation environment.

```
# Load SimBs into the bitstream storage memory
ModelSim> mem load -infile ./artifacts/sbt/xxx_bank0.txt ...
```

- Visualize transactions (Optional)

```
# Visualize transactions after the simulation starts
ModelSim> add wave sim:/solyr/rr0/rec/usr_trans
ModelSim> add wave sim:/solyr/rr0/rec/sbt_trans
```

The simulation commands vary from design to design, and the designer can refer to the TODO-LIST file generated by ReSim for more detailed information about the simulation commands.

Chapter 4

Reference Designs

The ReSim library includes 3 examples: XDRS, Fast PCIe configuration, and State Migration. The XDRS example further includes three versions: XDRS.QUICKSTART XDRS.SINGLE XDRS.MULTIPLE. It is recommended that you run these examples in the following order:

- XDRS.QUICKSTART: This example demonstrates the standard effort to use ReSim.
- XDRS.SINGLE: This example demonstrates the advanced concepts of ReSim. The concepts include, modifying artifacts for test-specific purposes (e.g. monitor, error injection) and using coverage-driven verification on DRS designs.
- XDRS.MULTIPLE: This example demonstrates the use of ReSim on designs that have multiple reconfigurable regions.
- Fast PCIe configuration: This example demonstrates the use of ReSim on a Xilinx reference design.
- State Migration: This example demonstrates using ReSim to simulate a processor-based design with EDK tools. It also demonstrates designing and simulating a system that accesses module state (FFs, memory cells) via the configuration port.

4.1 XDRS.QUICKSTART

The XDRS reference design (see Figure Figure 4.1) is similar to the ones reported in [11, 12, 10], and is also similar to the core logic of the XDRS platform reported in [10]. However, the original designs have been slightly modified in the released library. In particular, in order to focus on ReSim-based simulation, some of the user design modules are described using un-synthesizable code to improve readability. The testbench is also described using un-synthesizable code.

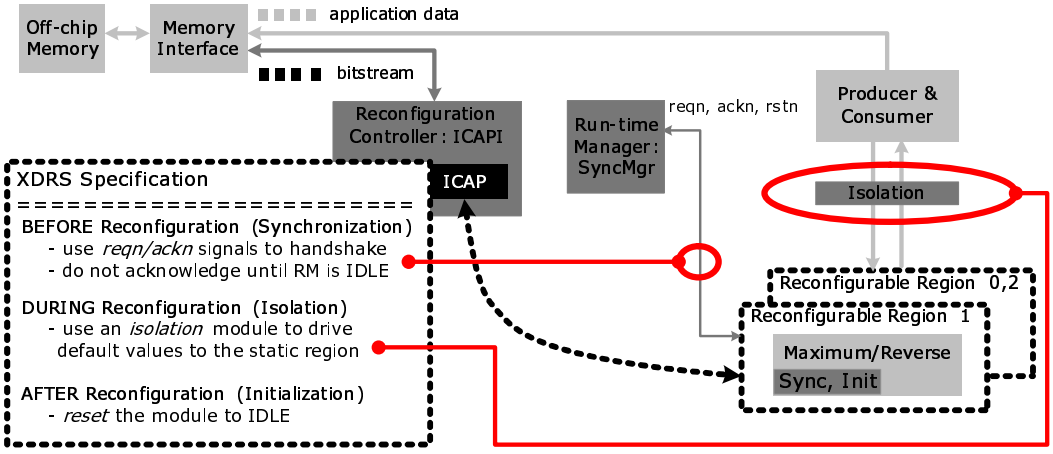


Figure 4.1: The XDRS reference design

- The XDRS application. The XDRS runs a producer-consumer application, in which the Producer-Consumer module (`./xdrs/prodcons.sv`) communicates with the RMs (`./xdrs/cores/maximum.v` & `./xdrs/cores/reverse.v`) to perform mathematical computation. Upon reconfiguration is requested, the RM block the request until it finishes the current data processing operation (`./xdrs/cores/intern_sync.v` & `./xdrs/cores/filter_sync.v`).

- The reconfiguration machinery. Reconfiguration machinery includes all logic that enables partial reconfiguration. The reconfiguration machinery (moderately shaded blocks in the figure) of the design is composed of the `ICAP-I` reconfiguration controller (`./xdrs/icapi.v`), the `SyncMgr` module (`./xdrs/manager.sv`) and the `Isolation` module (`./xdrs/isolator.v`). BEFORE reconfiguration, as listed in the specification, 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. On the other hand, the `ICAP-I` controller *transfers* a partial bitstream to the ICAP. AFTER reconfiguration, the newly configured module is *reset* to IDLE by the `SyncMgr` module.

- The system backplane (memory and bus). Both bitstream data and application data (i.e., data accessed by RMs) are stored in the same memory (`./xdrs/memctrl.sv`). During reconfiguration, a bus arbiter (`./xdrs/xbuscore.v`) schedules the two traffic streams, where the application traffic has a higher priority.

Preparing the Design/Testbench. According to Section 3.2, the testbench of the XDRS.QUICKSTART reference design instantiates the DUT, a bitstream storage memory and the simulation-only layer wrapper. Since the DUT is flatterned in the testbench, RR wrappers, as well as the `ICAP-I` reconfiguration controller, are directly instantiated in the testbench (`./xdrs/xdrs.sv`). Furthermore, the CP wrapper is instantiated in the `ICAP-I` reconfiguration controller.

```
1  // ./xdrs/xdrs.sv
2  // ============================
3  module xdrs ();
4      // The DUT Part:  Instantiate the RR wrapper & the ICAP-I module
5      // The XDRS.QUICKSTART reference design only has one RR
6      my_region region_0 (...);
7      icapi icapi_0 (...); // Instantiates the CP wrapper
8
9      // The TB Part:  Instantiate a memory & the simulation
10     // -only layer wrapper
11     memctrl mem_0(...); // A "32-bit x 4096" array
12     my_solyr sol_0();
13
14     // The TB Part:  Include tests
15     `include "tdpr_quick_start.sv"
```

Tests are "included" as part of the top-level testbench. This example only has one test (`./xtests/tdpr_quick_start.sv`), which performs two reconfiguration operations. The test assume that partial bitstreams for the `maximum` and the `reverse` modules are stored at memory address 0x100 and 0x200.

```
1  // ./tdpr_quick_start.sv
2  // ============================
3  // Producer-consumer thread
4  // Producing data by calling pc_0.produce_data()
5  initial begin
6      pc_0.produce_data(32'ha4a5a6a7);
7      pc_0.produce_data(32'ha0a1a2a3);
8
9  // Partial reconfiguration thread
10 // Performing partial reconfiguration by calling
   mgr_0.reconfigure_module()
11 initial begin
12     // Load bitstream from 0x100, 32 words
13     mgr_0.reconfigure_module(8'b0000_0001, 32'h100, 32);
```

Generating Artifacts. Create a `settings.do` file in the working folder. This file is required to setup environment variables. Here is an example of the file.

```
1  // ./settings.do
2  // ============================
3  // The following is an example of the settings.do file
4
5  set RSV_HOME "C:/DPR_TOOLS/ReSim"
6  set OVM_HOME "C:/ModelSim6.5g/verilog_src/ovm-2.1.1"
7  source "$RSV_HOME/scripts/resim.do"
```

Run the provided parameter script `auto_generation.tcl`. This step automatically generates simulation-only artifacts that should be included in the simulation testbench. The following are selected lines of the parameter script of XDRS.QUICKSTART.

```
1  // ./auto_generation.tcl
2  // ============================
3  // The parameter script for XDRS.QUICKSTART
4
5  source settings.do; // Setup library path
6  import ReSim::* // Import ReSim library
7
8  // Create a portmap and add ports to it.
9  rsv_create_portmap "my_if" "clk"
10 rsv_add_port "my_if" "rstn" in
11 ...
12
13 // Create reconfigurable regions and add RMs to it.
14 rsv_create_region "my_region" "my_if" 4 "" "my_ei"
15 rsv_add_module "my_region" "maximum" ""
16 ...
17
18 // Create the simulation-only layer and add RRs to it.
19 rsv_create_solyr "my_solyr" VIRTEX4 ""
20 rsv_add_region "my_solyr" "my_region"
21 rsv_gen_solyr "my_solyr" // Generate the simulation-only layer
22 ...
23
24 // Convert SimBs to ModelSim memory format
25 rsv_create_memory "zbt" 4 1 be
26 rsv_add_2_memory "zbt" "./artifacts/sbt/my_region_rm0.sbt" 0x100
27 rsv_add_2_memory "zbt" "./artifacts/sbt/my_region_rm1.sbt" 0x200
```

The generated SimBs are in binary format (See `./artifacts/sbt/xxxx.sbt`) and can not be used by the ModelSim simulator directly. The `rsv_create_memory` and the `rsv_add_2_memory` APIs (i.e., Lines 26-27) convert the binary SimBs into a "memory format" that are ready to be loaded into ModelSim. In this example, the memory is a 32bit-addressable, single bank memory, and the two SimBs are loaded to address 0x100 and 0x200 respectively. Note, the conversion of SimB is optional. The designer can create their own script to convert binary format SimB to whatever format that meets can be understood by the simulation environment.

Running Simulation. In this example, all ModelSim commands required to start simulation have been included in a script. Apart from compiling the generated and the built-in ReSim artifacts, the script also loads the generated SimBs into the memory of the testbench.

```
1  // ./simulate_mti.do
2  // ============================
3  // The simulation script for XDRS.QUICKSTART
4
5  ...
6
7  vsim -t ps -permit_unmatched_virtual_intf -L mtiReSim xdrs
8  mem load ... "/xdrs/mem_0/zbtmem"
9  mem load -infile ".../zbt_bank0.txt" ... "/xdrs/mem_0/zbtmem"
10
11 run -all
```

Figure 4.2 is a screen shot of simulating the XDRS.QUICKSTART reference design.

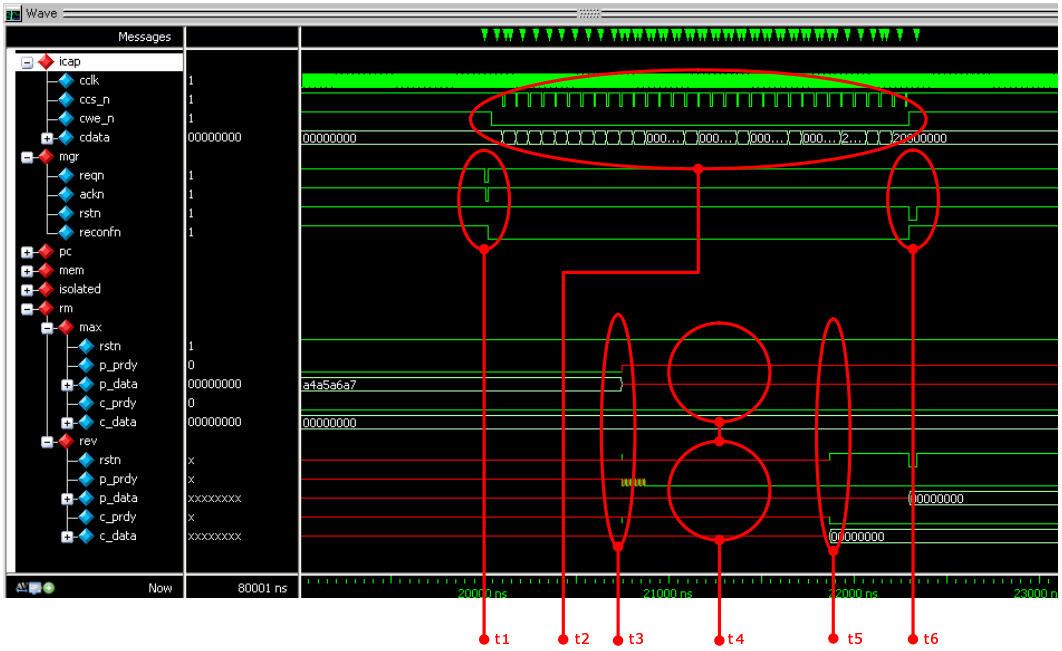


Figure 4.2: Simulating the XDRS.QUICKSTART reference design

@t1: The `SyncMgr` module disables the outgoing RM (i.e., the `maximum` module) so as to prevent the RM from starting a new operation (see the `mgr/reqn` and the `mgr/ackn` signals).

@t2: The `ICAP-I` reconfiguration controller transfers a SimB to the CP artifact (see the `icap` signal group).

@t3: The CP artifact parses the SimB and extracts the numerical ID of the module to be swapped in.

@t4: While the SimB is being written, the RR artifact injects errors to both the static region and the RM (see the `max` and `rev` signal groups), and we can check that the injected errors are cleaned by the `Isolation` module (see the `mgr/reconfn` signal, which controls the `Isolation` module).

@t5: After the SimB has been completely written to the CP artifact, the RR artifact checks the signature of the SimB and drives module swapping according to the numerical ID of the target module (see t3).

@t6: The static region initializes the new RM (i.e., the `reverse` module) so as to clear the undefined initial RM state (see the `mgr/rstn` and the `rev/rstn` signals).

4.2 XDRS.SINGLE

This example demonstrates the advanced concepts of ReSim. The concepts include, modifying artifacts for test-specific purposes (e.g. error injection) and using coverage-driven verification on DRS designs.

Preparing the Design/Testbench. The DUT of this example is exactly the same as the XDRS.QUICKSTART example. However, this example includes more tests (see `./xtests`). These tests examine various aspects of the XDRS system.

- TEST_DPR_SIMPLE: A simple test that performs two partial reconfiguration operations. For the 2nd partial reconfiguration, a high priority memory traffic occurs on the shared bus and the bitstream traffic is thereby delayed.
- TEST_DPR_READBACK: A test that performs configuration readback and state restoration. In particular, the registers of the `maximum` module is saved and restored by reading and writing the configuration port (see the `.sll` file `./artifacts/spy/my_region_rm0.sll`) .
- TEST_DPR_ISOLATION and TEST_DPR_RETRY: Two tests that exercise corner cases of the Isolation module.
- TEST_DPR_RANDOM: A test that drives random stimuli to the XDRS.
- TEST_DPR_DEMO: A test that is a mixture of all above tests. It is used for demonstration purposes.

Figure 4.3 illustrate selected testplan sections of the XDRS.SINGLE reference design (see `./testplan.xml` for the complete testplan). The example testplan includes code coverage items, which are automatically tracked by the simulator, and functional coverage items, which are manually modeled using SystemVerilog coverage groups (e.g. item 7.2), coverage directives (e.g. item 5.4) and assertions (e.g. item 6.6).

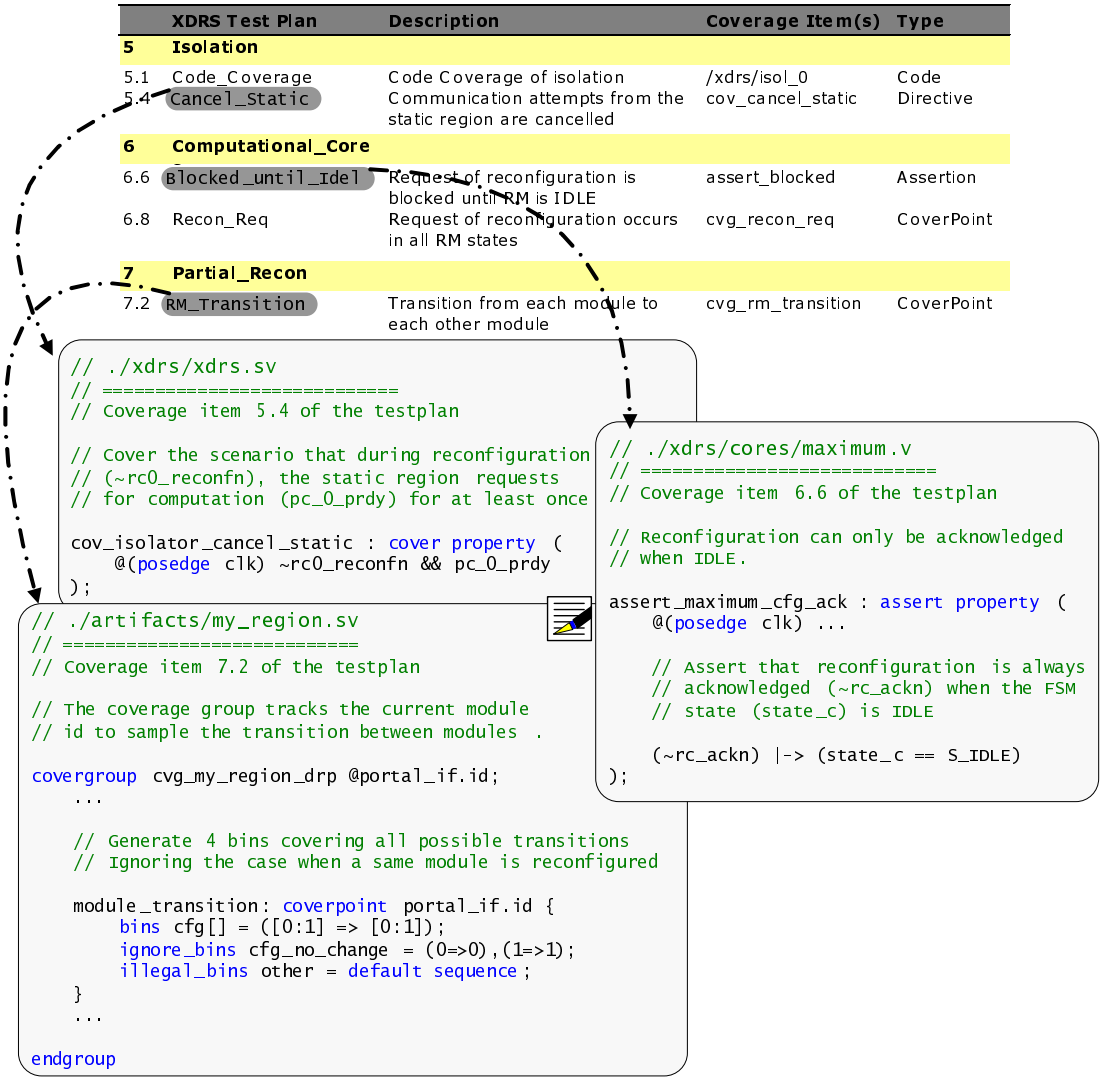


Figure 4.3: Selected testplan sections of the XDRS.SINGLE reference design

Generating Artifacts. The parameter script for this example is similar to the one in the XDRS.QUICKSTART reference design. However, this reference design modifies the default error injector (`./artifacts/my_ei_edited.sv`) and the generated simulation-only logic allocation files. This design also uses a scoreboard (`./artifacts/my_scb.sv`).

- The simulation-only logic allocation files (`./artifacts/spy/xxx.sll`) has been modified so as to map RTL signals to the simulation-only frames. These `.sll` files are used by tests that performs configuration readback (`tdpr_demo.sv` & `tdpr_readback.sv`)
- The modified error injector (`./artifacts/my_ei_edited.sv`) derives and overrides the base class so that it can inject design-specific errors to mimic spurious RM outputs and undefined RM state. In particular, the `inject_to_static_module`, `inject_to_reconfigurable_module` and the `inject_to_internal_signals` tasks have been re-defined.

The modified artifacts of this reference design are in the `./artifacts.edited` folder and they are all copied to overwrite the default artifacts generated by the Tcl script. It should be noted that modifying the generated artifacts is ***ONLY*** required are for advanced users who wish to change the default artifacts for test purposes.

```
1  // ./auto_generation.tcl
2  // ============================
3  // The parameter script for XDRS.SINGLE
4
5  ...
6
7  // Create the reconfigurable region and add modules
8  // Attach an error injector to the region
9  // Pass non-default parameter values to RMs
10
11 rsv_create_region "my_region" "my_if" 4 "" "my_ei_edited"
12 rsv_add_module "my_region" "maximum" "#(48)"
13 rsv_add_module "my_region" "reverse" "#(24,24)"
14
15 // Create the simulation-only layter and attach a scoreboard to it
16 rsv_create_solyr "my_solyr" VIRTEX4 "my_scb"
```

```
1  // ./artifacts/spy/my_region_rm0.sll
2  // ============================
3
4  // The data1 signal is allocated to
5  // frame_address = 0x00000000, offset = 32, bitwith = 32.
6  0x00000000 32 32 data1
7  ...
8
9  // The stat_0/incnt signal is allocated to
10 // frame_address = 0x00000002, offset = 64, bitwith = 48.
11 0x00000002 64 48 stat_0/incnt
```

```
1  // ./artifacts/my_ei_edited.sv
2  // ============================
3
4  // Inject errors to the ***outputs***to the static region
5  // Use the ei_vi interface to enable/disable error injection
6  // Use the sei_vi interface to drive signals of the static region
7
8  task my_ei_edited::inject_to_static_module ();
9      ei_vi.sei_en <= 1;
10     ...
11     sei_vi.rc_ackn <= x;
12     sei_vi.p_prdy <= x;
13     ...
14 endtask : my_ei_edited::inject_to_static_module
15
16 // Inject errors to the ***inputs***to the RM
17 // Use the ei_vi interface to enable/disable error injection
18 // Use the dei_vi interface to drive signals of the RM
19
20 task my_ei_edited::inject_to_reconfigurable_module ();
21     ei_vi.dei_en <= 1;
22     ...
23 endtask : my_ei_edited::inject_to_reconfigurable_module
24
25 // Inject errors to the ***internal signals***of the RM
26 // Use the Tcl API "rsv_iei_hdl_state" to inject to internal signals
27
28 task my_ei_edited::inject_to_internal_signals ();
29     // Use the "rsv_execute_tcl" macro to evaluate
30     // Tcl APIs other ModelSim commands (e.g., force, mem load)
31     // in the SystemVerilog code.  The "interp" variable
32     // points to the ModelSim Tcl interpreter
33     rsv_execute_tcl(interp, $psprintf("ReSim::rsv_iei_hdl_state
...))
34 endtask : my_ei_edited::inject_to_internal_signals
```

Running Simulation. This reference design has two ModelSim simulation scripts. The `simulate_mti.do` script is used to run a single test while the `simulate_coverage.do` script is used to run all tests and analyze coverage results.

```
# Run a single test
ModelSim> do simulate_mti.do
# Run all tests and view coverage reports
ModelSim> do simulate_coverage.do
```

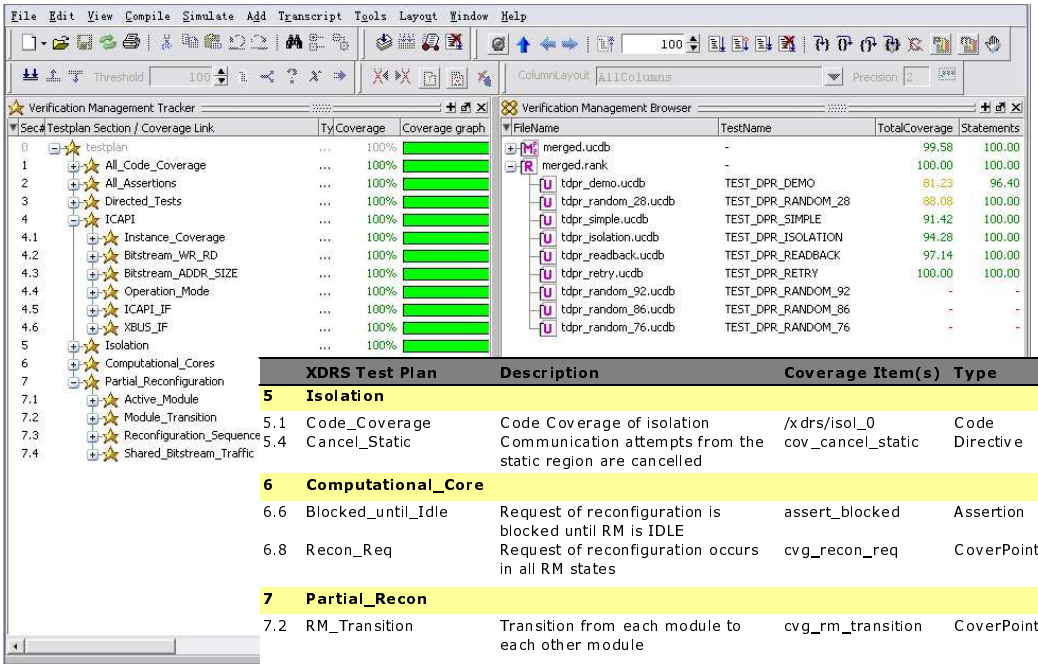
The `simulate_mti.do` script of this reference design is similar to XDRS.QUICKSTART, apart from the following two main differences.

- Need to define a test name before compiling the testbench

- Can optionally add ReSim transactions to the waveform window

```
1  // ./simulate_mti.do
2  // ============================
3  // The simulation script for XDRS.SINGLE
4
5  ...
6
7  // Change the name of the test in simulate_mti.do
8  vlog ... +define+TEST_DPR_RANDOM "./xdrs/xdrs.sv"
9
10 // Add ReSim transactions to the waveform
11 // Can only add to the waveform after the simulation has started.
12 run 10ns
13 add wave ... //solyr/rr0/rec/sbt_trans
14
15 run -all
```

Figure 4.4 shows the coverage results of the XDRS.SINGLE reference design. After running all tests, the `simulate_coverage.do` script merges the coverage data of individual tests and generates a report (see the `./coverage` folder). Please refer to ModelSim User Guide [16] for more information on coverage-driven verification.



	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_Core**			
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

Figure 4.4: Coverage results of the XDRS.SINGLE reference design

4.3 XDRS.MULTIPLE

This example demonstrates the use of ReSim on designs that have multiple RRs. In particular, the XDRS.MULTIPLE reference design has 3 RRs. RR0 & RR1 contains the `maximum` & `reverse` modules that are the same as previous examples but with different parameters, whereas RR2 is mapped with two low pass filters (`./xdrs/lpfilter/...`). Three regions have the same set of IO signals.

The filters use a `pipeline_sync` module (`./xdrs/lpfilter/pipeline_sync.v`) to perform synchronization. In particular, upon a reconfiguration is requested, the `pipeline_sync` module blocks the request and push zeros to the filter to flush the partial results that are still in the filter. Reconfiguration is acknowledge after all pipeline data are flushed out of the filter. This reference design has two tests:

- TEST_DPR_STREAMING: This test reconfigures RR2 (i.e., the filter) twice.
- TEST_DPR_RANDOM: A test that drives random stimuli to the XDRS system.

Run the example according to the following step:

- Generate the artifacts by running the parameter script (`./auto_generation.tcl`).
- Change the name of the test to be compiled in the `simulate_mti.do` script.
- Start ModelSim and run simulation.

4.4 Fast PCIe Example Design

This reference design demonstrates using ReSim with XAPP883, the Fast PCIe configuration system [20]. It is strongly recommended that the readers run the original design (XAPP883) before running this example.

Figure 4.5 illustrates the block diagram of this reference design. At startup, the Fast PCIe configuration (FPCIe) 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.

Preparing the Design/Testbench.

- Download the Fast PCIe configuration reference design

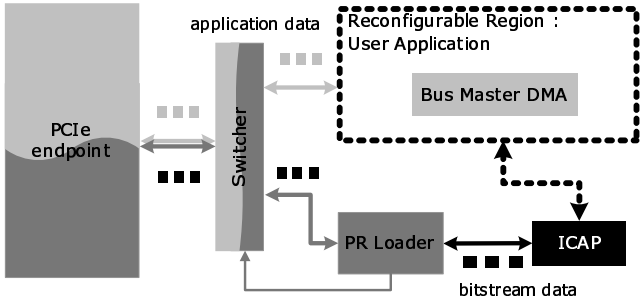


Figure 4.5: The Fast PCIe configuration reference design, after [20]

  - Go to Xilinx website and download XAPP883
  - Extract and copy the design files to `./xapp883`

- Generate the PCIe endpoint logic according to the descriptions from XAPP883.
  - Use Coregen to generate the PCIe endpoint logic
  - Copy the endpoint code to `./xapp883/hw/Source/PCIe_x8_gen1/source`
  - Use the files in the custom folder to overwrite the same files in `./xapp883/hw/Source/PCIe_x8_gen1/source`

- Modify the design so as to integrate with ReSim
  - Start bash (e.g., cygwin, mingw, or xilinx-bash) and apply a patch

```
# Modify the original Fast PCIe system
bash> patch -p0 -i board_patch.txt
```

  - The patch changes the original design according to the design guidelines outlined in Section 3.2. In particular,
    * It changes the RM name from `pcie_app_v6` to `pcie_bmdma` (i.e., RM name should be different from the RR name)
    * It changes the `pr_loader` module to instantiate the CP wrapper instead of the ICAP primitive.
    * It changes the test to send SimBs to the FPCIe system.
    * It adds code to instantiate the simulation-only layer wrapper.

Generating Artifacts.

- Create a settings.do file to setup environment variables
- Run auto_generation.tcl

Running Simulation. Start simulation using the `simulate_mti.do` script. Apart from compiling the design source and the ReSim built-in and generated artifacts, the script loads the generated SimBs to the memory and visualizes SimB transactions. It should be noted that the memory is instantiated in the testbench and is added by the batch file.

```
1  // ./simulate_mti.do
2  // ============================
3  // The simulation script for FPCIe
4
5  ...
6
7  vsim ... work.board
8  mem load ... "/xdrs/mem_0/zbtmem"
9  mem load -infile ".../mem_bank0.txt" ... "/board/.../sbtmem"
10
11 run 10ns
12 add wave ... //solyr/rr0/rec/sbt_trans
13 run 80us
```

Figure 4.6 illustrates a screenshot of ReSim-based simulation.

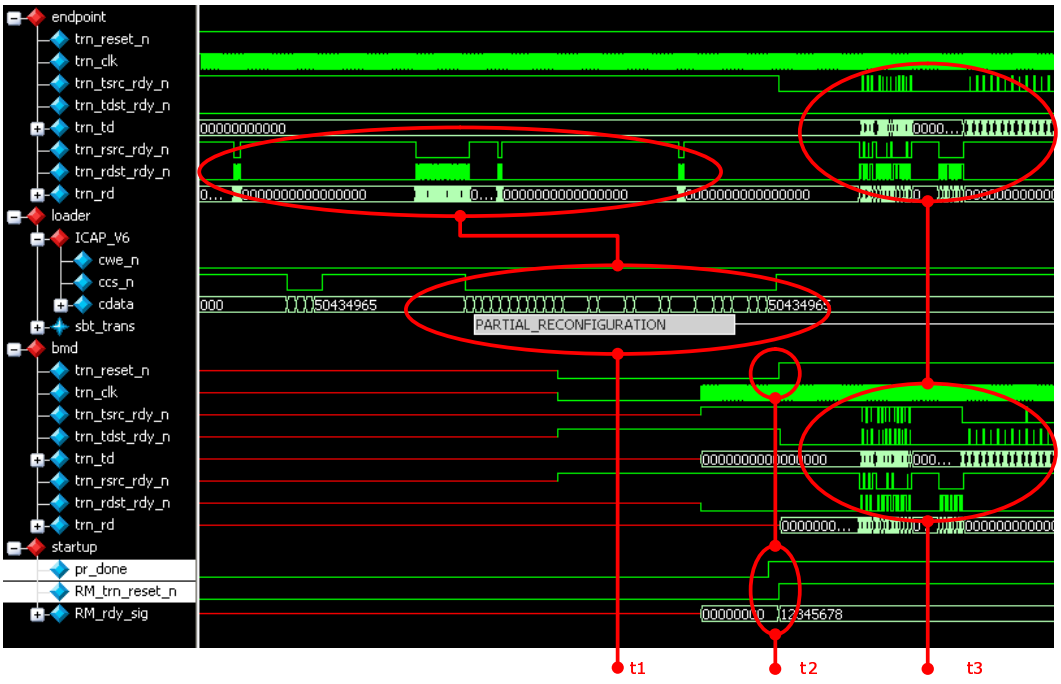


Figure 4.6: Simulating the FPCIe reference design

@t1: The testbench sends a SimB over the PCIe link. When the SimB is being transferred, the Bus Master DMA module is injected with error signals (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.

4.5 State Migration Example Design

This reference design demonstrates using ReSim in simulating a microprocessor-based system. The design is based on UG744 [22] whereas the simulation environment is referenced from XAPP1111 [18]. It is strongly recommended that the readers run UG744 and XAPP1111 before running this example. Figure 4.7 illustrates the block diagram of this reference design. Compared with UG744, the STATE_MIGRATION reference design

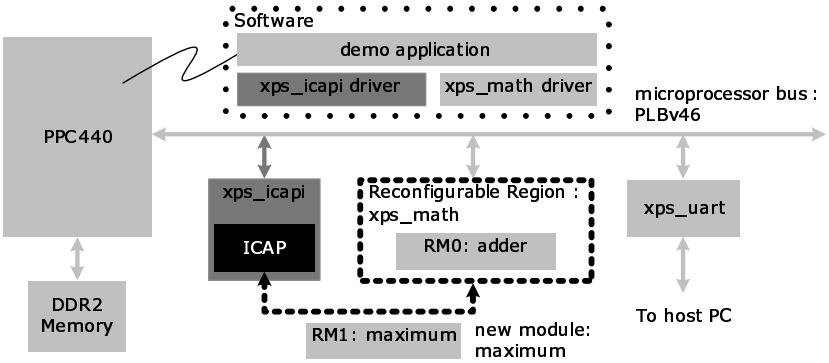


Figure 4.7: The State Migration reference design

- Uses an in-house reconfiguration controller, the `xps_icapi` module, as opposed to the `xps_hwicap` IP core released by Xilinx.
- Targets a ML507 board instead of an ML605 board. Therefore, it uses PowerPC as the microprocessor as opposed to MicroBlaze.
- Apart from partial reconfiguration, the design also reads RM state (i.e., values of a `result` register and a `statistic` register of the RMs) via the ICAP port.

It should be noted that the design has only been tested with EDK 12.4 [24].

Preparing the Design/Testbench. According to Section 3.2, the design/testbench of this reference design has been prepared for ReSim-based simulation. Since this reference design involves software, we use the software driver for the `xps_icapi` module as an example to explain how to use a simulation/verification-friendly software coding style. The software driver for the `xps_icapi` module is organized into two layers:

- `icapi.c` & `icapi.h`: These two files contain functions and macros that access the memory-mapped registers of the `xps_icapi` module. These functions interact with the hardware to read/write a particular length of bitstreams from/to the ICAP.
- `icapi_configuration.c` & `icapi_configuration.h`: These two files utilize `icapi.c` & `icapi.h` access the configuration mechanism of the FPGA device. It provides routines such as reading one frame and extract the value of one multi-bit signal.
- `fpga_family.h`: This file defines the target FPGA family (i.e., either Virtex-5 or ReSim).
- `logic_allocation.h`: This file contains logic allocation information extracted from vendor generated `.ll` files and user-defined `.sll` files. For each RTL signal to be saved, it defines a buffer to store the logic allocation information of each individual bit.

The software driver has two versions (i.e., either Virtex-5 or ReSim). In particular, for each mismatch between Virtex-5 and ReSim, the software use conditional compilation directive to separate the two versions. By doing this, we are able to localize fabric-dependent code and ReSim-based simulation can still assist in testing the fabric independent part of the driver (i.e., the code that are not encapsulated by #ifdef).

Generating Artifacts.

- Create an `./edk/state_migration_incl.make` file. This file sets the paths which are used by the makefile.

```
XILINX_HOME=C:/Xilinx/12.4/ISE_DS/ISE
XLIB_HOME=C:/Xilinx/12.4/Users/xlib/mti_se/6.5g/nt
```

- Start Xilinx Bash Shell, change to `./edk` directory and run makefile, which generates simulation files in `./simulation/behavioral`.

```
bash> cd C:/DPR_TOOLS/ReSim/examples/state_migration/edk
bash> make -f state_migration.make sim
```

  - NOTE: Since the makefile calls Xilinx binary utilities (e.g. `simgen`), it can only be executed in Xilinx bash. In particular, it is recommended to run EDK in batch mode so as to get more "predictable" results. The GUI may unexpectedly change project settings.
  - NOTE: The directory structure of this example is not the same as the default layout of EDK projects.

- Create a settings.do file in `./simulation` to setup environment variables. It should be noted that this file also need to set up Xilinx libraries

```
1 // ./settings.do
2 // ============================
3 // Set up Xilinx libraries in the settings.do file
4 ...
5 set XILINX_HOME "C:/Xilinx/12.4/ISE_DS/ISE"
6 set XLIB_HOME "C:/Xilinx/12.4/Users/xlib/mti_se/6.5g/nt"
```

- Copy the data2mem utility to `./simulation/tb`
  Copy the edit_mem.pl utility to `./simulation/tb`
  Both utilities are used to convert software executables to text format recognized by ModelSim.
  Both utilities can be found in XAPP1111.
- Run `auto_generation.tcl`.
  - It should be noted that the generated error injectors and `.sll` files are overwritten by modified files from `./simulation/artifacts.edited`

Running Simulation. Start simulation using the `simulate_mti.do` script. Apart from compiling the design source and the ReSim built-in and generated artifacts, the script loads the pre-compiled software (`./sdk/tapp_dpr_0/Debug/tapp_dpr_0_sim.elf`) as well as generated SimBs to the memory, and visualizes SimB transactions. Figure 4.8 illustrates a screenshot of the simulation.

@t1: The initialization routine of the `xps_icapi` driver reads the IDCODE register of ICAP to check the target device. This routine can be simulated without change since the CP artifact supports configuration readback. In particular, the IDCODE of ReSim, 0xc1b2011 is returned to the user design from the CP artifact.

@t2: A GCAPTURE command is issued to the CP artifact. The values of the `rm1/result` and the `rm1/statistic` registers are copied to the `spy_memory`.

@t3: The user design reads the CP artifacts and the values of the `rm1/result` or the `rm1/statistic` registers is returned to the user design as a simulation-only frame in the readback SimB.

Run Pre-compiled Bitstreams and Binaries

- You can find the pre-compiled bitstream in `./edk/images/...`
- You can find the pre-compiled software executable in `./sdk/tapp_dpr_0/Debug/tapp_dpr_0_fpga.elf`

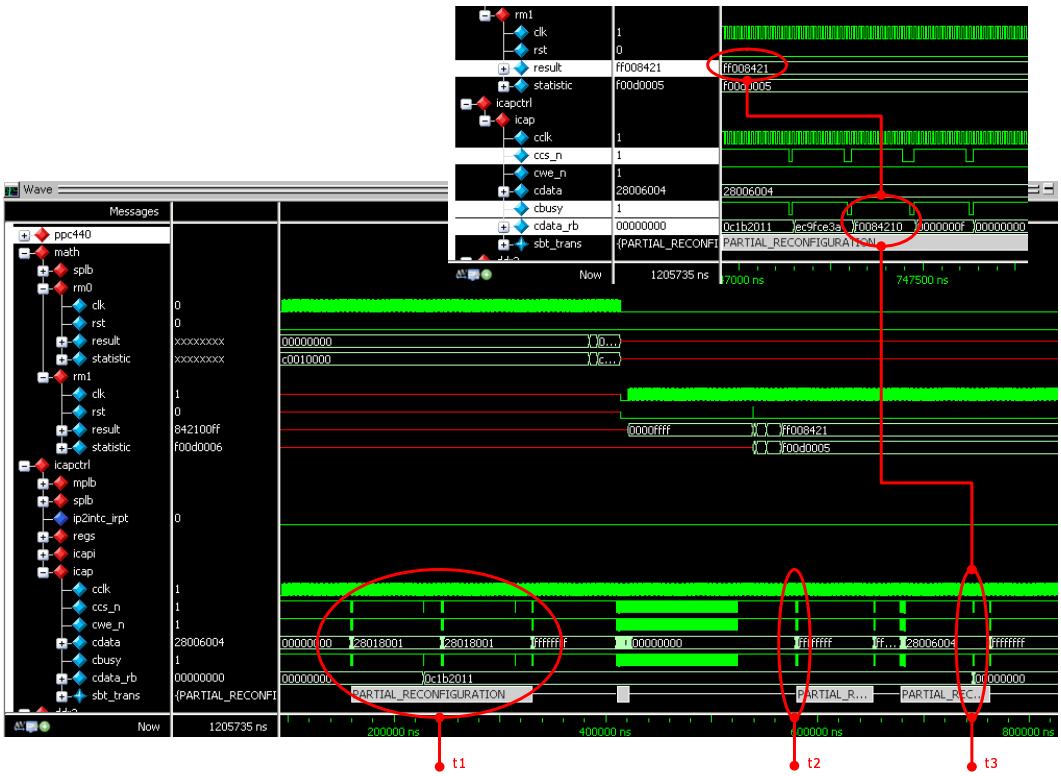


Figure 4.8: Simulating the State Migration reference design

- Copy the generated partial bitstream to SD Flash
  FLASH_DISK:/ML507/george/adder.bin
  FLASH_DISK:/ML507/george/maximum.bin
  - NOTE: The path is hard coded in the software and can be changed if desired.
  - NOTE: The software do not process the bitstream header so you need to copy the `.bin` file instead of the `.bit` file to the SD flash. To generate the bin file, please add the "-d -l -m -g Binary:Yes" options when generating the bitstreams.
- Download the design and the software in SDK.
  You need to use the `./edk/images/system_bd.bmm` file in order to download software correctly.

Implement the Design.

- Start Xilinx Bash Shell, change to `./edk` directory and run makefile, which synthesize of the static and reconfigurable modules.

```
bash> cd C:/DPR_TOOLS/ReSim/examples/state_migration/edk
bash> make -f state_migration.make netlist
```

- Use the planahead project in `./edk/planahead` to implement the design.
  - NOTE: State saving needs to locate the FF bits on the FPGA fabric. The location information is provided by the logic allocation file (`*.ll`) generated by the `bitgen` utility, and the software need to use the `.ll` file to extract correct bits from the readback frames. It is desirable that such allocation information does not change so that the software (i.e., `./sdk/tapp_dpr_0/src/logic_allocation.h`) does not have to be updated after each implementation run. Therefore in this reference design, the placement of RM registers is locked down in the ucf files (`./edk/netlist/math_core/adder/math_core_i_adder.ucf` and `xxxxxx/math_core_i_maximum.ucf`).
  - NOTE: If the placement and routing may fail due to the constraints that lock RM registers, the designer needs to remove the constraints. After implementation, the designer needs to update the `logic_allocation.h`.
  - NOTE: To use partial reconfiguration with EDK, the designer needs to instruct PlanAhead to generate a `*.bmm` file in the `ngdbuild` step. This step is illustrated in UG744.
  - NOTE: Please add the "-d -l -m -g Binary:Yes" options in the `bitgen` step of implementation.

Build the Software.

- Export to SDK

```
bash> cd C:/DPR_TOOLS/ReSim/examples/state_migration/edk
bash> make -f state_migration.make sdk
```

- Create SDK project
- Choose compilation target (ReSim or Virtex5)
  - In `./sdk/tapp_dpr_0/src/fpga_family.h`, change the compilation target to either Virtex-5 or ReSim.

```
1  // ./sdk/tapp_dpr_0/src/fpga_family.h
2  // ============================
3  // The header that defines the target FPGA family
4  ...
5
6  // Using ReSim
7  #define XHI_FAMILY       XHI_DEV_RESIM
8  #define XHI_FPGA_IDCODE  XHI_RESIM_1_00A
9
10 // Using ML507
11 #define XHI_FAMILY       XHI_DEV_FAMILY_V5
12 #define XHI_FPGA_IDCODE  XHI_XC5VFX70T
```

- Compile the software

Chapter 5

Inside the ReSim Library

5.1 Library Overview

Figure 5.1 illustrates ReSim-based RTL simulation of a DRS design (similar to Figure 1.2 but with more details). The ReSim library adopts the Open Verification Methodology (OVM) [9] and, according to the reusability principle of OVM, the simulation-only layer is separated into a *module-based* part and a *class-based* part, which are connected via *SystemVerilog Interfaces* (shown as clouds and the (vi) symbols). In particular, a ReSim-based simulation environment contains the following components:

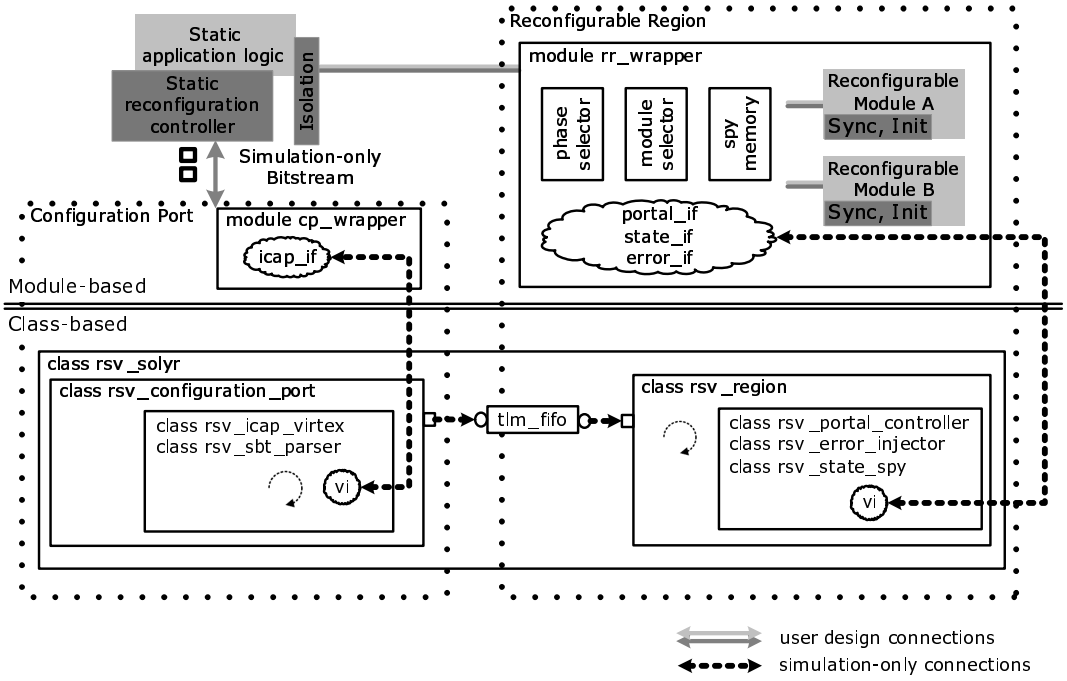


Figure 5.1: ReSim-based RTL simulation (similar to Figure 1.2 but with more details)

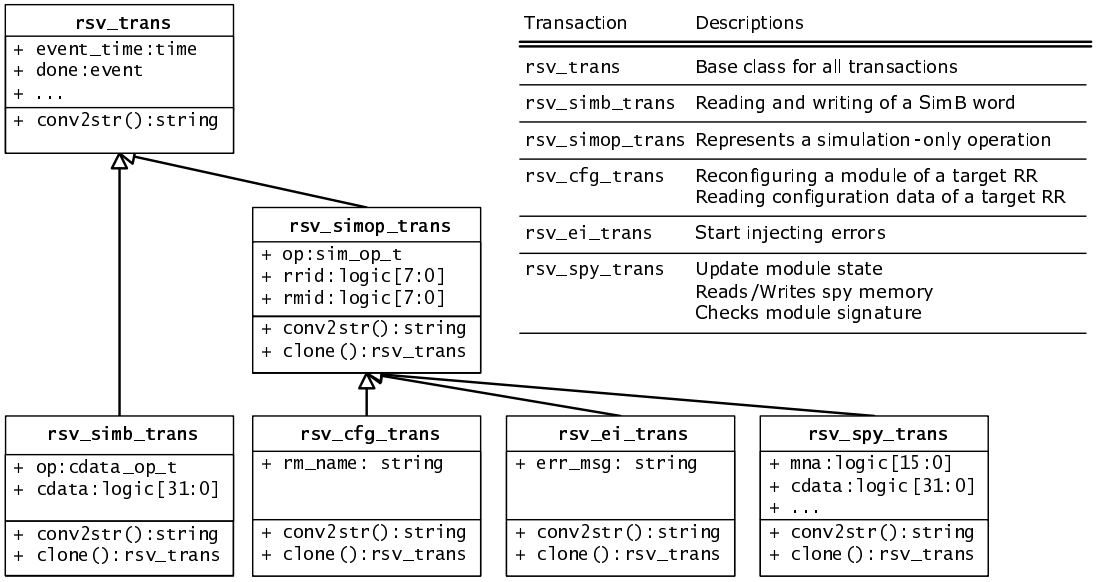
- The CP artifact
  - The CP wrapper (i.e., the `cp_wrapper` module), which is instantiated in the user design hierarchy as the configuration port.
  - The CP class (i.e., the `rsv_configuration_port` class), which controls the operation of the `cp_wrapper` module. It instantiates,
    - The `rsv_icap_virtex` class: Drives the `icap_if` interface
    - The `rsv_sbt_parser` class: Parses SimBs and extracts RRIDs/RMIDs. Returns readback SimBs.
- The RR artifact
  - The RR wrapper (i.e., the `rr_wrapper` module), which is instantiated in the user design hierarchy as the top-level of the reconfigurable region. It instantiates the RMs and artifacts such as,
    - The `module_selector` module: Connects and interleaves RMs
    - The `phase_selector` module: Connects and interleaves the error sources
    - The `spy_memory` module: Synchronizes module state data.
  - The RR class (i.e., the `rsv_region` class), which controls the operation of the `rr_wrapper` module. It instantiates,
    - The `rsv_portal_controller` class: Triggers module swapping
    - The `rsv_error_injector` class: Provides error sources
    - The `rsv_state_spy` class: Reads/writes the `spy_memory`. Synchronize state data of simulated RMs.
- The simulation-only layer artifact
  - The simulation-only layer wrapper[1], which is instantiated in the testbench as the module-based part of the simulation-only layer.
  - The simulation-only layer class (i.e., the `rsv_solyr` class), which instantiates all class-based artifacts.
- The scoreboard artifact
  - The scoreboard class (i.e., the `rsv_scoreboard` class)[2], which collects coverage data in simulation

The module-based part instantiates the artifacts as part of the user design and interacts with the user design via RTL signals whereas the class-based part controls the operation of the module-based part. The class-based artifacts are parameterizable OVM components and are built-in to the ReSim library whereas the module-based artifacts are automatically generated by ReSim to parameterize the class-based artifacts. We describe the class-based, built-in artifacts in Section 5.2, the module-based, ReSim-generated artifacts in Section 5.3, and the ReSim APIs in Section 5.4.

[1] Not shown in the figure
[2] Do not have a corresponding module-based part

5.1.1 Transaction-based Communication

ReSim artifacts communicate with each other via transactions. Figure 5.2 illustrates the definition and the relationship of ReSim transactions. From a language perspective, a transaction is a SystemVerilog class, which has data fields and member functions. For example, in case of reconfiguring a new module, the `op` field of the `rsv_cfg_trans` transaction is set to "write module" and the `rrid` and `rm_name` fields contain the numerical ID and the name of the new module, respectively. Note that the `op` and the `rrid` fields of the `rsv_cfg_trans` transaction are inherited from its parent `rsv_simop_trans` transaction. In ReSim, transactions are used as objects that are passed around between artifacts.



Transaction	Descriptions
rsv_trans	Base class for all transactions
rsv_simb_trans	Reading and writing of a SimB word
rsv_simop_trans	Represents a simulation-only operation
rsv_cfg_trans	Reconfiguring a module of a target RR Reading configuration data of a target RR
rsv_ei_trans	Start injecting errors
rsv_spy_trans	Update module state Reads/Writes spy memory Checks module signature

Figure 5.2: ReSim transactions

Using transactions, artifacts cooperate with each other to perform simulation-only tasks. Transactions are produced by artifacts requesting simulation-only tasks and are consumed by artifacts that perform the operations. In particular,

- The `rsv_icap_virtex` class wraps the collected SimBs into `rsv_simb_trans` transactions and passes them to the SimB parser.
- By parsing the SimB, the `rsv_sbt_parser` class produces `rsv_simop_trans` transactions (and derived transactions) to target RRs.
- The target `rsv_region` class performs the specified simulation-only tasks accordingly. It calls the `rsv_portal_controller` class to select a new RM, calls the `rsv_error_injector` class to inject errors that mimic spurious module outputs and undefined module state, and calls the `rsv_state_spy` class to interpret the configuration data section of a SimB by checking the module signature and updating module state.

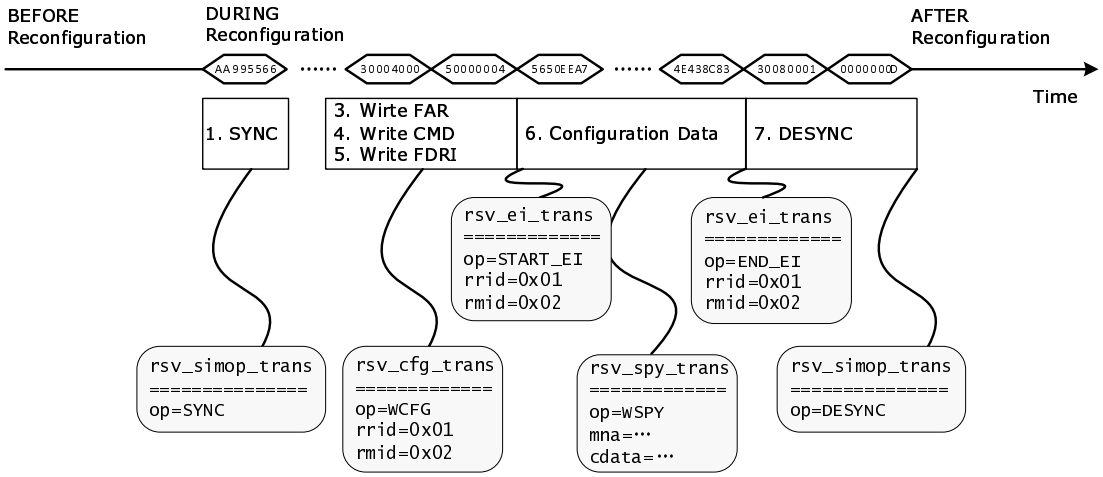


Figure 5.3: Timing of ReSim transactions (partial reconfiguration)

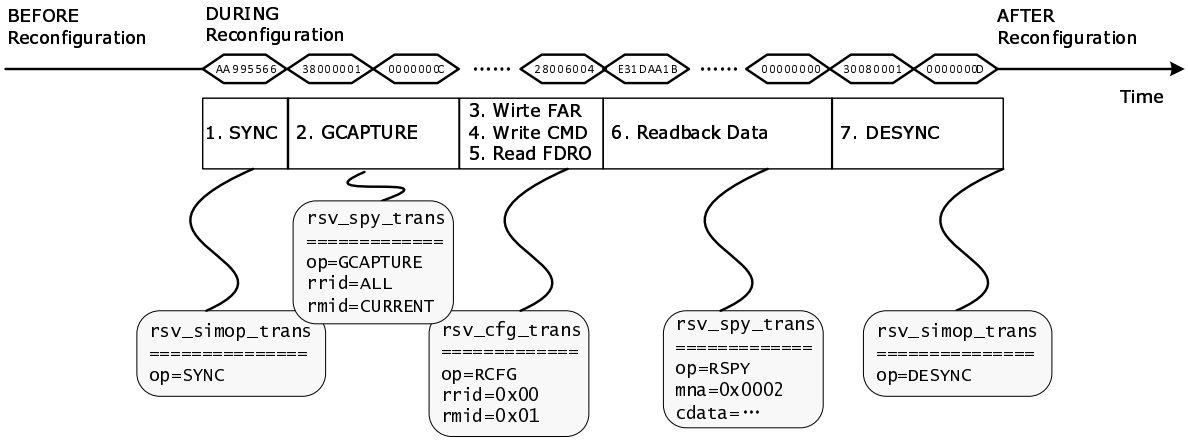


Figure 5.4: Timing of ReSim transactions (configuration readback)

Figure 5.3 illustrates the relationship of partial reconfiguration and the passing of transactions. The reconfiguration starts with a SYNC, followed by an `rsv_cfg_trans` transaction indicating "reconfiguring a module", followed by an `rsv_ei_trans` transaction to start error injection, followed by a number of `rsv_spy_trans` transactions corresponding to each word of the configuration data, followed by another `rsv_ei_trans` transaction that ends error injection, and ends with a DESYNC.

Similarly, Figure 5.4 illustrates the configuration readback operation using transactions. Apart from setting the `op` field of `rsv_cfg_trans` and `rsv_spy_trans` transactions to READ, configuration readback creates an extra `rsv_spy_trans` transaction for `GCAPTURE`, and does not produce `rsv_ei_trans` transactions. The SimBs for Figures 5.3 and 5.4 are from Tables 5.1 and 5.2, respectively.

5.2 ReSim Built-in Artifacts[3]

5.2.1 CP Class

The CP class, i.e., the `rsv_configuration_port` class, controls the operations of the CP wrapper (see Section 5.3.1), and is composed of a device-specific reconfiguration interface and a generic SimB parser (see Figure 5.5).

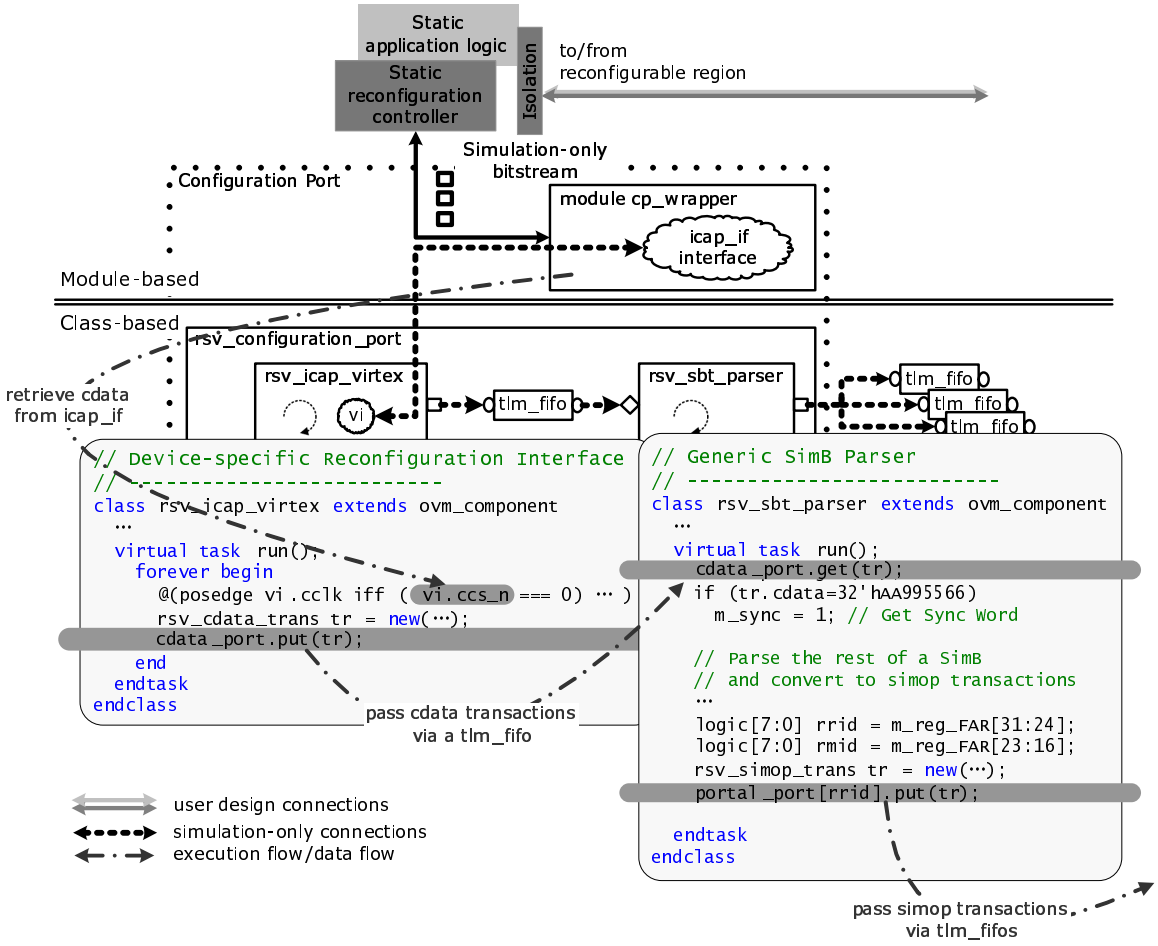


Figure 5.5: The configuration port class

Device-specific Reconfiguration Interface. The `rsv_icap_virtex` class models the device-specific reconfiguration interface. It drives the `icap_if` interface as described by the configuration guides of the target FPGA families (e.g. [23, 26, 27, 28]). For example,

- It collects SimBs written to it and returns readback SimBs.

[3]To focus on the main idea of ReSim, some of the code shown in this chapter has been modified for readability purposes.

- It checks that `RDWRB`, the read write qualifier signal, does not change its value when `CSB`, the enable signal, is asserted.
- It drives the `BUSY` signal in the process of configuration readback.

Since the interfacing protocol of configuration ports varies between FPGA families and FPGA vendors, this class is device-specific. SimB data collected by this class are passed to the generic SimB parser using `rsv_simb_trans` transactions.

Generic SimB Parser. By consuming `rsv_simb_trans` transactions, the `rsv_sbt_parser` class interprets the SimB stream and requests simulation-only tasks accordingly. In particular,

- It produces `rsv_cfg_trans` transactions to indicate RRIDs/RMIDs of reconfiguration.
- It produces `rsv_ei_trans` transactions to indicate the start and end of error injection.
- It produces `rsv_spy_trans` transactions to read and write configuration data.

The example code in Figure 5.5 illustrates the identification of a SYNC word in the SimB stream. It also implements

- Basic ICAP registers (i.e., CRC, CMD, FAR, FDRI, FDRO, IDCODE)
- Basic ICAP commands (NULL, WCFG, RCFG, RCRC, GRESTORE, GCAPTURE, DESYNC).

5.2.2 RR Class

The RR class, i.e., the `rsv_region` class, controls the operations of the RR wrapper (see Section 5.3.2), and is composed of a `rsv_portal_controller`, a `rsv_error_injector`, and a `rsv_state_spy` class (see Figures 5.6 and 5.7).

Portal Controller. In ReSim-based simulation, module swapping is modeled by a `module_selector` module, which interleaves the communication between RMs and the static region so that only one module is selected as the currently active module (see Figure 5.6). Furthermore, reconfiguration is not triggered until after the successful transfer of a SimB. In particular, the `reconfigure()` member function (see Figure 5.6) of the `rsv_portal_controller` class is not called until the `rsv_region` class receives `rsv_cfg_trans` transactions from the SimB parser.

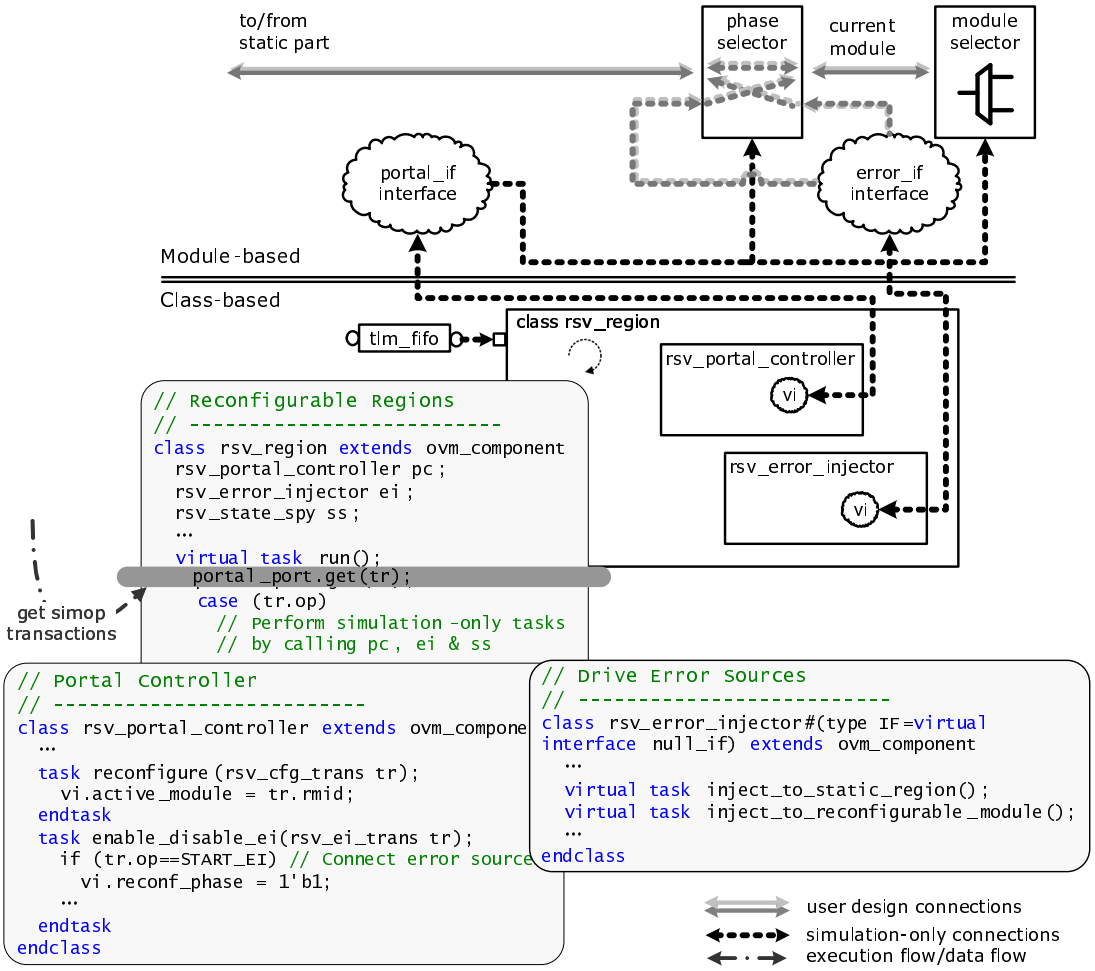


Figure 5.6: The portal controller and the error injector class

Modeling Module Swapping and Triggering Condition. Similar to previous simulation approaches (e.g., [14], [17]), ReSim also connects RMs in parallel and selects one active RM with a MUX-like `module_selector` module. However, in our simulation-only layer approach, the selection of the active RM is triggered by the SimB. Therefore,

- Any run-time dependent events that delay the bitstream transfer also delay the module swapping. The delay of module swapping is more accurately modeled instead of being approximated by a constant value.
- Failing to transfer the SimB correctly prevents the new RM from being swapped in. Bugs in the bitstream transfer logic can quickly be identified.

ReSim more accurately simulates the timing of module swapping.

Error Injector. ReSim uses the error injector artifact to mimic the spurious RM outputs during reconfiguration and the undefined RM state after reconfiguration. The `rsv_error_injector` class defines three types of error sources:

- Static error injection (see the `inject_to_static_region()` member function), which injects errors to the static region
- Dynamic error injection (see the `inject_to_reconfigurable_module()` member function), which injects errors via the *input pots* of the currently active RM
- Internal error injection (see the `inject_to_internal_signals()` member function), which inject errors to the *internal* signals of the currently active RM.

By default, these error injection member functions drive undefined "x" values as error sources. Since the error source tasks are both defined as *virtual* tasks, they can be overridden with user-defined errors or error sequences.

The error signals driven by the `rsv_error_injector` class are connected/disconnected to/from the static region/RMs via a switch-box like `phase_selector` module. When the SimB is being written to the CP artifact, the `rsv_portal_controller` class drives the `phase_selector` module to connect the error sources (see the `enable_disable_ei()` member function). By the end of bitstream transfer, the `rsv_portal_controller` class resumes the connection between the static region and the currently active RM and disconnects the error sources.

Modeling Spurious Outputs and Undefined Initial State. Compared with DCS, which only drives undefined "x" values to the static region [17], our approach injects errors to both the static and the reconfigurable regions.

- Errors injected to the static region (i.e. static error injection) model the spurious outputs of the module undergoing reconfiguration and help to test the isolation logic of the user design. Failing to isolate the RR correctly can be quickly identified as a consequence of the injected errors propagating to the static region.
- Errors injected to the reconfigurable modules (i.e., dynamic and internal error injection) model the undefined initial state of the RM and help to test the initialization mechanism of the design. Failing to initialize the newly configured module correctly can be detected since the injected errors are not cleared.

Furthermore, in our approach, the start and end of error injection is also triggered by the SimB, which, compared with DCS, more accurately models the timing of error injection.

State Spy. Figure 5.7 illustrates the simulation of state saving and restoration in ReSim. The `spy_memory` module mimics the behavior of the configuration memory on real FPGAs. The `rsv_state_spy` class controls the operation of the `spy_memory`. In particular,

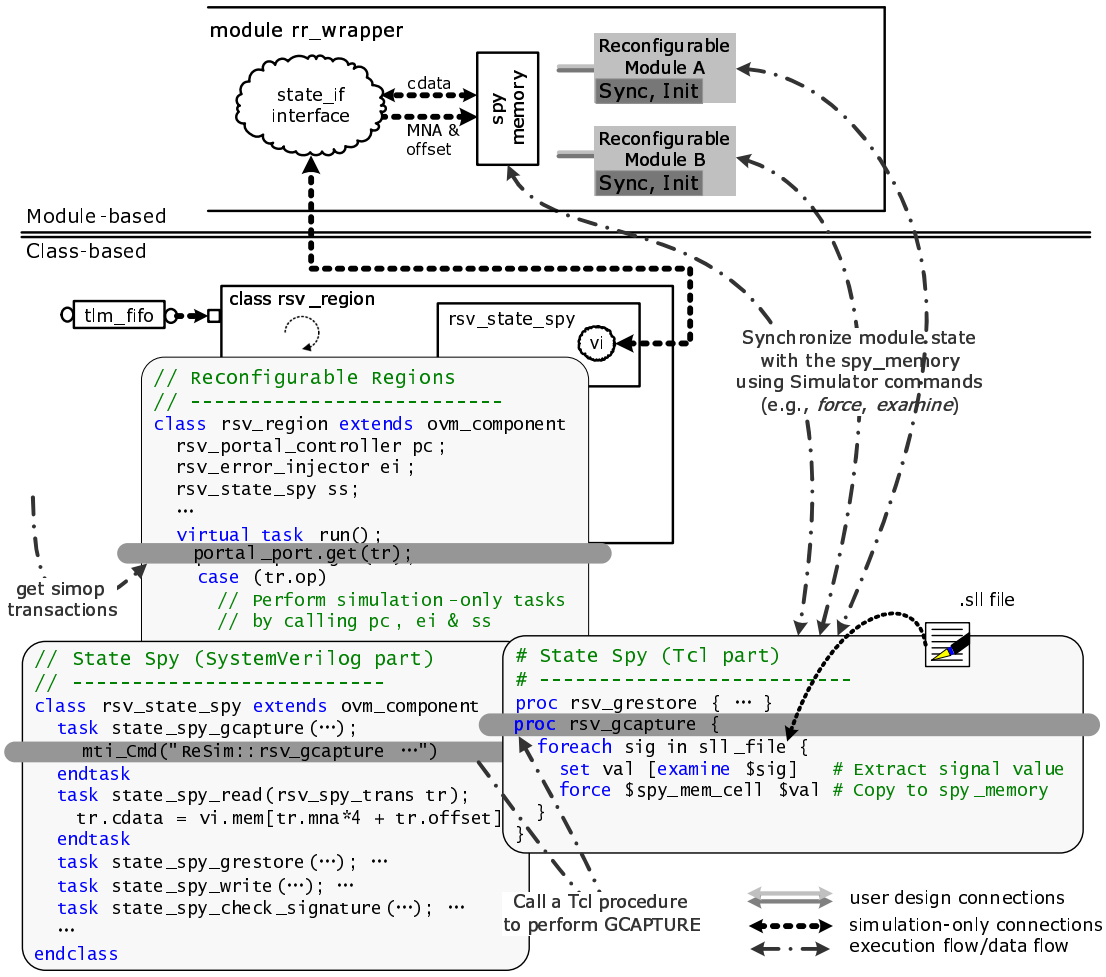


Figure 5.7: The state spy class

- It reads/writes the `spy_memory`. For example, the `state_spy_read()` member function accesses the `spy_memory` via the `state_if` interface (`...=vi.mem[]`). Since each `spy_memory` frame has 4 words, the `state_spy_read()` member function calculates the memory address of the `spy_memory` by multiplying the Minor Address by 4 and adding the offset.

- It synchronizes the buffered state data with the RTL signals of the simulated RMs. For example, the `state_spy_gcapture()` member function (see Figure 5.7) calls a Tcl script of the ReSim library, which loops through all signals in the `.sll` file, extracts the signal values using the ModelSim `examine` command, and copies the values to the `spy_memory` using the ModelSim `force` command.

- It checks module signature. The `state_check_signature()` compares the signature computed from each simulation-only frame with the expected signature, and reports an error if mismatch.

The member functions of the `rsv_state_spy` class are called when `rsv_spy_trans` transactions are received from the SimB parser (see Figure 5.3 and Figure 5.4). Therefore, the state saving and restoration of RMs are triggered by the transfer of SimBs.

5.2.3 Simulation-only Layer Class

The simulation-only layer artifact is the container for ReSim built-in artifacts (see Figure 5.8). It is the root of class-based artifacts. It instantiates the CP class and the RR classes, as well as a scoreboard artifact.

5.2.4 Scoreboard Class

Figure 5.8 illustrates the scoreboard component in ReSim. The `rsv_scoreboard` class is used to collect coverage data in simulation. The default scoreboard collects the coverage of the RRID/RMID fields of SimBs. In particular,

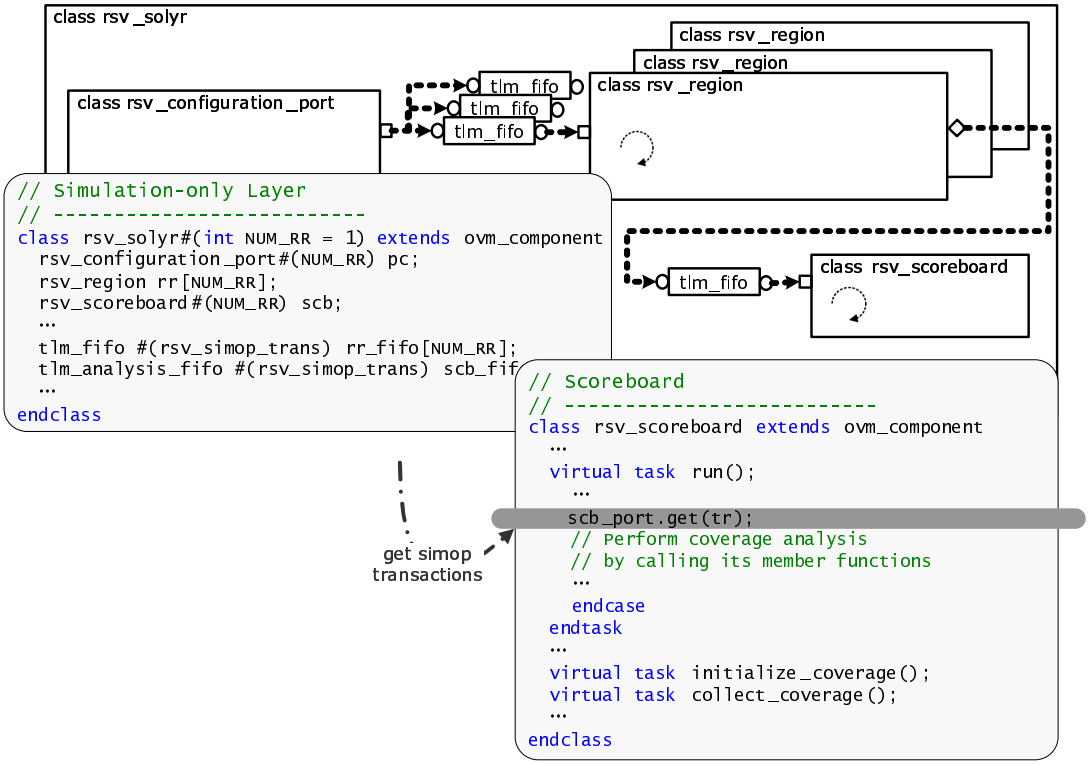


Figure 5.8: The scoreboard class

- It collects whether all RMs have been exercised (i.e., to cover all RMs)
- It collects whether each RM has been reconfigured to every other RM (i.e., to cover all possible transitions between RMs)

5.3 ReSim-Generated Artifacts

5.3.1 CP Wrapper

The CP wrapper is instantiated in the user design hierarchy as the configuration port. It has the same IO signals as the real configuration port. As illustrated by Figure 5.9, ReSim generates two versions of the CP wrappers for simulation and implementation, respectively

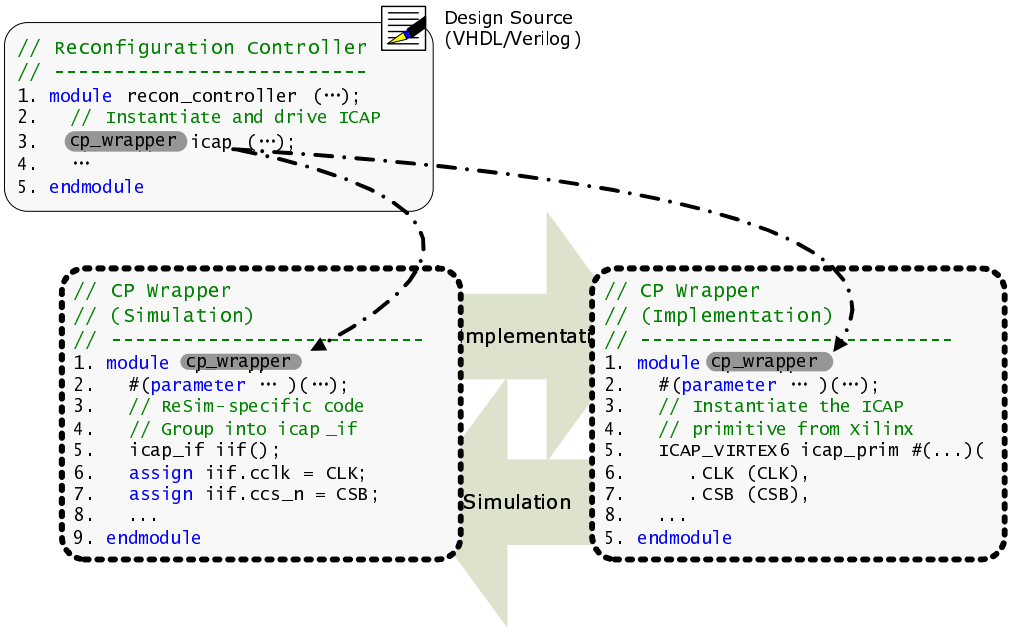


Figure 5.9: The CP wrapper generated by ReSim

- For simulation (using ReSim): The simulation version of the CP wrapper groups IO signals of the configuration port into an `icap_if` interface (Line 5-7 of the simulation version of the `cp_wrapper` module in Figure 5.9).
- For implementation (using e.g., PlanAhead [25]): The implementation version of the CP wrapper instantiates the ICAP primitive from Xilinx [23, 26, 27, 28] (Line 5-7 of the implementation version of the `cp_wrapper` module in Figure 5.9).

5.3.2 RR Wrapper

The RR wrapper for each RR is used as a placeholder for RMs mapped to it. ReSim automatically generates the source code for the RR wrapper (see Figure 5.10).

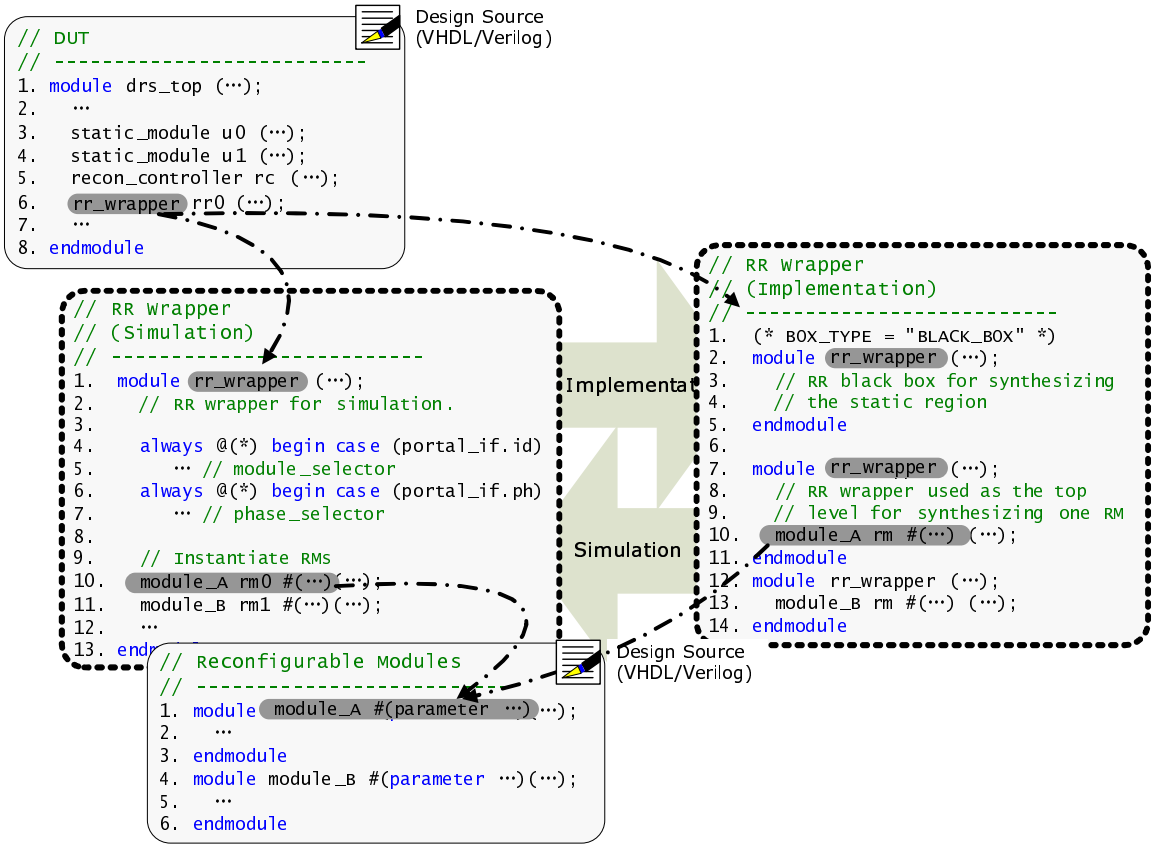


Figure 5.10: The RR wrapper generated by ReSim

- For simulation (using ReSim): The simulation version instantiates *both* RMs (Line 10-11) as well as other simulation-only artifacts (Line 4-7).
- For implementation (using e.g., PlanAhead [25]): Since the static region and the RMs are implemented separately, ReSim generates one version for synthesizing static region and one for *each* RM.
  - The RR wrapper for synthesizing the static design is a *black box* (Line 1 of the implementation version of the `rr_wrapper` module in Figure 5.10),
  - The RR wrapper for synthesizing each RM directly *wraps* the RM (Line 7-14 of the implementation version of the `rr_wrapper` module in Figure 5.10). In particular, the parameters of each RM should be hard coded and the RR wrapper should not contain any glue logic.

We describe the artifacts instantiated in the RR wrapper as follow.

Module Selector.[4] The `module_selector` module selects one active module according to the currently active RMID (Line 4-5 of the simulation version of the rr_wrapper module in Figure 5.10).

Phase Selector.[5] The `phase_selector` module connects/disconnects the error sources to/from the currently active module and the static region according to whether or not the configuration data section of a SimB is being written to the CP artifact (Line 6-7 of the simulation version of the rr_wrapper module in Figure 5.10).

Spy Memory.[6] The `spy_memory` module models the configuration memory of the simulation-only layer (see Figure 5.11). Instead of being organized into rows and columns like the FPGA fabric, the `spy_memory` is organized according to the affiliation of RMs to RRs. In particular,

- Each RR and RM is given a numerical ID (RRID & RMID). The example shown in Figure 5.11 has 3 RRs with RRIDs = {0,1,2} and RR0 has 2 RMs with RMID = {0,1}. Note that the RMIDs commence at 0 for each RR.
- Each RR is composed of N simulation-only frames where N is a user-defined parameter specified by a parameter script (see Section 3.3).
- Within one RR, a configuration frame is indexed by a minor address (MNA) and each frame contains 4 words.

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

- The WORD 0 of all frames of an RR form a sequence of 32-bit signatures assigned by the simulation-only layer. The signature data can identify a unique RM in the user design and are expected to be kept unchanged throughout a simulation run.
- WORDs 1-3 of a configuration frame are used to store the value of RTL signals that represent the state elements (i.e., flip-flops and memory cells) of the simulated module. If the simulation exercises a `GCAPTURE/GRESTORE` operation, the values stored in the simulated configuration memory are synchronized with the RTL signals of the simulated module. Such a synchronization operation mimics the `GCAPTURE/GRESTORE` of real FPGAs [23, 26, 27, 28] and can be used to simulate state saving and restoration.

[4] In the generated source code, the `module_selector` is *flatterned* in the RR wrapper.
[5] In the generated source code, the `phase_selector` is *flatterned* in the RR wrapper.
[6] In the generated source code, the spy memory is implemented *inside* the `state_if` interface.

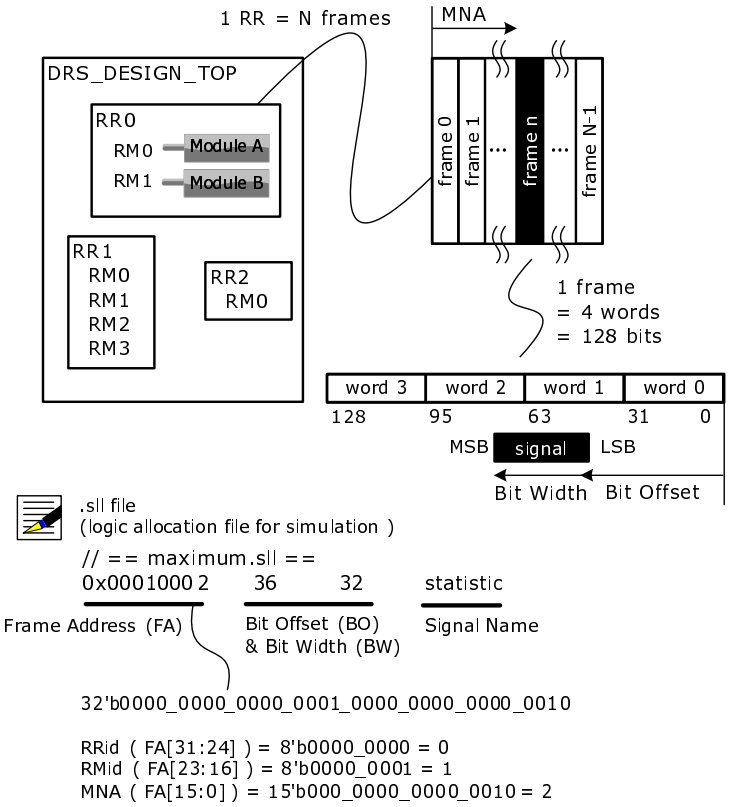


Figure 5.11: The organization of the spy memory

5.3.3 Simulation-only Layer Wrapper

The simulation-only layer wrapper starts the simulation-only artifacts (see Figure 5.12). It is a module without any input or output, and is typically placed in the testbench.

5.3.4 Simulation-only Bitstreams

As simulation cannot effectively make use of a real bitstream, the simulation-only layer substitutes a *simulation-only bitstream* (SimB) to model the bitstream traffic. The SimB captures the essence of a real bitstream but its size is significantly reduced.

5.3.4.1 Simulation-only Bitstream for Configuring a New Module

Table 5.1 provides an example of a SimB that configures a new module.

- A SimB starts with a `SYNC` word (entry 1 in Table 5.1)
- Instead of containing the Frame Addresses of the configuration data to be written, as found in a real bitstream, a SimB contains the RRID and RMID of the module to be reconfigured. In the example SimB,

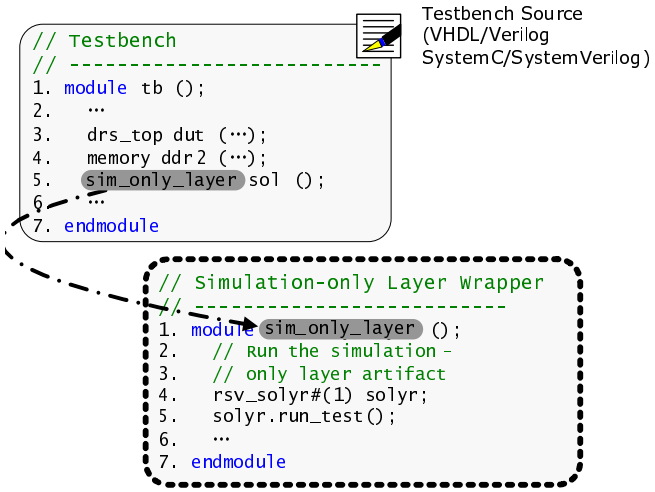


Figure 5.12: The simulation-only layer wrapper generated by ReSim

Table 5.1: An example SimB for configuring a new module

Entry	SimB	Explanation	Actions Taken
1	0xAA995566	SYNC Word	Start the **DURING Reconfiguration** phase
2	0x20000000	NOP	–
3	0x30002001 0x01020000	Type 1 Write FAR FA=0x01020000	Select the module id=0x02 to be the next active module in reconfigurable region id=0x01
4	0x30008001 0x00000001	Type 1 Write CMD WCFG	
5	0x30004000 0x50000010	Type 2 Write FDRI Size=16	Pad frame: not required Configuration data: 16 words (4 frames)
6	0x5650EEA7 0xF4649889 ... 0xA9B759F9 0x4E438C83	SimB Frame 0 Word 0 SimB Frame 0 Word 1 ... SimB Frame 3 Word 2 SimB Frame 3 Word 3	Module signature; Module state data; Enable/Disable error injection
7	0x30008001 0x0000000D	Type 1 Write CMD DESYNC	End the **DURING Reconfiguration** phase

  - 4 consecutive words (entries 3-4 in Table 5.1) request that the current module in the RR with ID = 0x1 be replaced by the new module with ID = 0x2.
  - The size of the configuration data section is set to 4 (entry 5 in Table 5.1), which indicates that the target RR contains 4 simulation-only layer frames.

- Instead of containing bit-level configuration settings for the module to be configured, as found in a real bitstream, the configuration data section of a SimB contains fields that are to be written to the configuration memory of the simulation-only layer (entry 6 in Table 5.1). For each simulation-only frame,

  - WORD 0 is the module signature field
  - WORDs 1-3 store the values of state elements (i.e., flip-flops and memory cells) of the RM to be configured.

  The configuration data section also indicates the start and end of error injection.

- A SimB ends with a `DESYNC` command (entry 7 in Table 5.1).

5.3.4.2 Simulation-only Bitstream for Saving and Restoration

SimBs can also be used to simulate state saving and restoration. Table 5.2 provides an example of a readback SimB that saves the state of the `statistic` register of Figure 5.11. Similar to the SimB for reconfiguring a new module, a readback SimB also

Table 5.2: An example SimB for state saving

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

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

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

- Starts with a `SYNC` word and ends with a `DESYNC` command (entries 1 and 7 in Table 5.2).
- Uses the RRID and RMID of the target RM as the Frame Address (entries 3 in Table 5.2).

However, there are a couple of differences between the readback SimB in Table 5.2 and the partial SimB of Table 5.1. In particular,

- The example readback SimB has an extra `GCAPTURE` command to copy the values of the RTL signal (i.e., the value of the `statistic` register) to the `spy_memory`.
- Since the `statistic` register is mapped to the 2nd frame of the target RR (i.e., MNA=2, see the `.sll` file entry of Figure 5.11), the example readback SimB only reads the 2nd frame instead of all 4 frames of the target RR.
- During state saving, the contents of the configuration data section of the readback SimB (the 4 words of entry 6 of Table 5.2) are *returned from* instead of *being written to* the simulated configuration port. The operational sequence of configuration readback looks as follow:
  - Entries 1-5 are written to the CP artifact
  - The CP artifact is switched to read mode and responds with the 4 words of configuration data. WORDs 1-3 of the configuration data section contain state data of the `statistic` register.
  - The CP artifact is switched back to write mode when the `DESYNC` command is issued.

  After reading back the SimB, the application logic of the user design can extract the value of the `statistic` register.

The restoration process reverses the state saving procedure. To restore the value of a register, the user design merges the previously saved state data with the SimB at the position specified by the `.sll` file. During simulation, a `GRESTORE` SimB (for `GRESTORE`-based restoration method) or a module reset (for reset-based restoration method) copies the state data to the simulated module.

Modeling Bitstream Traffic. Compared with previous simulation approaches (e.g., [14], [17]), ReSim is the first tool to simulate the bitstream traffic. Therefore,

- The bitstream transfer datapath (i.e., the transfer, compression, decryption and arbitration of bitstreams) is exercised in simulation and a bug on the bitstream datapath can be detected.
- Instead of being determined by the resources used, the size of a simulated SimB is defined by the parameter script, and can be adjusted for design-/test- specific needs, such as,
  - To exercise corner cases on the bitstream transfer datapath (e.g., FIFO full/empty),
  - To increase/decrease the period when errors are being injected, and
  - To store more or fewer state data of simulated RMs.

Modeling Bitstream Content. Compared with previous simulation approaches (e.g., [14], [17]), ReSim is the first tool to parse and interpret bitstream content in simulation. Therefore,

- If any bit of the signature data or the RRID/RMID fields of a SimB are corrupted because of a bug in the design, the signature check fails and is reported during simulation. Checking the signature verifies that a correct bitstream is written to the configuration port.

Compared with [17], which modifies the RTL description of all storage elements to simulate state saving and restoration, ReSim more accurately models the actual state saving and restoration process of DRS designs. Therefore,

- Incorrect state data, typically caused by bugs in the design, can propagate to RMs of the simulated user design after state restoration, and can therefore be quickly identified in simulation.

5.3.5 Simulation-only Logic Allocation Files

To map RTL signals to the `spy_memory`, the simulation-only layer treats the simulation-only logic allocation file (`.sll` file) as being equivalent to the logic allocation file (`.ll` file) generated by the FPGA vendor tools. On real FPGAs, the state of a multi-bit register is flattened into individual flip-flops that are mapped to the configuration memory bit-by-bit [23, 26, 27, 28]. However in the simulation-only layer, a multi-bit register (typically represented by RTL signals in the HDL code) is stored contiguously. Since signals are typically grouped in the RTL user design, storing the state bits contiguously improves debugging productivity. Thus, in order to locate one user signal in the configuration memory, the designer needs to know the Bit Width (BW) of the signal, in addition to the RRID, RMID, MNA and Bit Offset (BO).

Figure 5.11 provides an example of the `.sll` file. The `statistic` register is located at Frame Address `0x00010002`, which is decomposed into `RRID` = 0, `RMID` = 1 and `MNA` = 2. The BO and BW fields indicate that the `statistic` register starts at bit 36 and is 32 bits wide. Similarly, RTL signals that corresponds to memory cells (e.g., 2-D arrays in Verilog/VHDL) can also be mapped to the configuration memory of the simulation-only layer using the `.sll` file.

5.3.6 SystemVerilog Package

The ReSim-generated SystemVerilog package file includes ReSim-generated artifacts in a single file. It is used for compilation purposes.

5.3.7 SystemVerilog Interfaces

The parameter script defines the interfacing signals that cross the RR boundary. This list of signals are grouped in the a SystemVerilog interface and its source code is generated by ReSim.

5.3.8 TODO-LIST File

The TODO-LIST file describes the steps involved to start a ReSim-based simulation run. In particular, it contains ModelSim scripts to compile the ReSim library and to run the simulation.

5.3.9 Report File

The report file includes the assignment of IDs to RMs and RRs in the user design.

5.3.10 Derived Class

The default behavior of the generated artifacts and the built-in artifacts can be modified for design-/test-specific needs. The following are typically usage examples of deriving and overriding the default class implementation.

Device-specific reconfiguration interface. The default `rsv_icap_virtex` class models the Virtex-4,5 and 6 FPGA families, which operate on 32-bit SimBs and produce one `rsv_simb_trans` transaction every cycle. The `rsv_icap_virtex` class can be derived to support new FPGA families. For example,

- For DRS designs targeting Spartan-6 devices (e.g. [8]), the designer can derive an `rsv_icap_spartan` class that operates on a half SimB word (16-bits) and produces one `rsv_simb_trans` transaction every other cycle.
- For DRS designs targeting Altera FPGAs, the designer can derive an `rsv_cp_stratix` class that implements the reconfiguration interfacing protocol of Altera devices [2].

Generic SimB Parser. The default `rsv_sbt_parser` class is a generic SimB parser that are independent of the FPGA family and interpret simulation-only bitstreams. The `rsv_sbt_parser` class can be derived and extended to

- Parse and interpret real bitstreams

Error Injector. By default, the `rsv_error_injector` class drives undefined "x" values as error sources. The designer can derive the error injector to define design-/test-specific error sources, such as,

- Stuck-at-zero errors
- Stuck-at-one errors
- Random signal values
- Glitches
- Design-/test-specific error sequence, such as
  - Injecting a spurious interrupt request
  - Initiating a spurious bus master transaction request
  - Injecting a spurious packet to the RM

Scoreboard. The default `rsv_scoreboard` class collects the coverage of the RRID/RMID fields of SimBs. The designers can add design-/test-specific coverage items to a derived scoreboard class. For example

- For designs that have multiple RRs, simulation needs to collect cross-coverage of the combination of RMs mapped to multiple RRs

5.4 ReSim APIs

This section describes the Tcl APIs of the ReSim library. The APIs are arranged in alphabetical order.

rsv_add_2_memory

- Usage: `rsv_add_2_memory` *`memory_name simb_file_name memory_address`*
- Description: The generated SimBs are in binary format (See `./artifacts/sbt/xxxx.sbt`) and can not be used by simulation directly. This API converts the binary SimB file *`simb_file_name`* to memory format loadable by ModelSim. The memory format should be previously defined by the `rsv_create_memory` API. The SimB is expected to be loaded to address *`memory_address`* of the memory, and the address is based the granularity specified in the `rsv_create_memory` API. Fox example, address 0x100 indicates the 0x100th entry of the memory. If the memory granularity is 4, the byte address of SimB is therefore 0x400. The converted SimB files can be found at `./artifacts/sbt/memory_name_bank0,1,2...txt`. Designers can also create in-house tools to parse/use/convert SimBs.

- Examples:

```
rsv_add_2_memory zbt "./artifacts/sbt/my_rr_rm0.sbt" 0x100 // Add generated
SimB to memory starting from 0x100
```

rsv_add_module

- Usage: `rsv_add_module` *region_name module_name optional_module_parameter_list*
- Description: Add a RM named *module_name* to the RR named *region_name*. The argument *module_name* is the design unit name of an RM. In ReSim-based simulation, each RR can have multiple copies of a same RM (i.e., multiple instances of a same design unit). The RM can have same or different parameter list values. However, they must be added separately by multiple `rsv_add_module` calls. As an option, you can pass Verilog parameter/VHDL generic values to the module via the *optional_module_parameter_list* argument, which is passed as a Tcl string directly to instantiate the module and it uses Verilog Syntax.
- Examples:

```
rsv_add_module my_rr module_A "" // Add the module_A module to the my_rr
region
rsv_add_module my_rr module_A "#(24,3)" // Passing non-default parameter
values
rsv_add_module my_rr module_A "#(.p1(3),.p0(24))" // Using name association
rsv_add_module my_rr module_A "#(.p1(5))" // Only pass a non-default value
to parameter p1 and leave p0 with its default value
```

rsv_add_port

- Usage: `rsv_add_port` *portmap_name signal_name signal_direction optional_signal_width*
- Description: Add a signal named *signal_name* to the portmap named *portmap_name*. A portmap must already been defined by `rsv_create_portmap`. The *signal_direction* argument is either *in* or *out*. The *optional_signal_width* argument is 1 by default.
- Examples:

```
rsv_add_port rr_if rstn in // Add a rstn input to the rr_if portmap
rsv_add_port rr_if data out 32 // Add a 32-bit data output to the rr_if
portmap
```

rsv_add_region

- Usage: `rsv_add_region` *`solyr_name region_name`*
- Description: Add the reconfigurable region *`region_name`* to the simulation-only layer *`solyr_name`*.
- Examples:

```
rsv_add_region my_sim_only_layer my_rr // Add the my_rr region to the
simulation-only layer
```

rsv_cleanup

- Usage: `rsv_cleanup`
- Description: Clean up all ReSim related global variables in the Tcl interpreter. It is typically used at the beginning or the end of the parameter description script. This API does not have any argument.

rsv_create_memory

- Usage: `rsv_create_memory` *`memory_name granularity number_of_banks endian`*
- Description: Specify the memory format of the SimB storage memory in ReSim-based simulation. The *`granularity`* argument indicates in bytes the smallest addressable unit of the memory (e.g.,. *`granularity`*=4 means 4-byte addressable or word-addressable). The *`number_of_banks`* argument specifies the number of memory banks that are connected in parallel. The *`endian`* argument can be either *`le`* for little endian or *`be`* for big endian. The arguments specified by this API should match the memory model in the simulation environment.
- Examples:

```
rsv_create_memory zbt 4 1 be // Define a zbt memory that is 4-byte-
addressed, has only 1 bank, and is big endian
rsv_create_memory ddr2 2 4 le // Define a ddr2 memory that is 2-byte-
addressed, has only 4 banks, and is little endian
```

rsv_create_portmap

- Usage: `rsv_create_portmap` *portmap_name clock_name*
- Description: Define a portmap named *portmap_name* with a clock named *clock_name*. In ReSim, a portmap groups the IO signals of a RR. The portmap is used by all RMs mapped to the RR. ReSim only supports RRs with one and only one clock.
- Examples:

```
rsv_create_portmap rr_if clk // Create an rr_if portmap that has a clk
clock signal
```

rsv_create_region

- Usage: rsv_create_region *region_name portmap_name region_size reserved_argument optional_error_injector_name*
- Description: Define a RR named *region_name* which uses the *portmap_name* portmap. The *region_name* is the design unit name of the reconfigurable region. In ReSim-based simulation, RRs must have unique names. Multiple identical RRs (i.e., same IOs, same RMs etc ...) must be created separately by multiple `rsv_create_region` calls with different names. The RR contains *region_size* number of simulation-only frames, which determines the size of the generated SimB. The *reserved_argument* argument is reserved and can only be passed with a null string "". This argument was used to specify a monitor of the reconfigurable region in previous versions of ReSim. Each region can have an optional error injector. In most cases, the automatically generated error injector is enough for simulation. As an option, the designer can modify the generated error injector design-/test- specific purposes. The designer can pass the *optional_error_injector_name* argument with null string "" if he/she decides not to use an error injector in the simulation environment.
- Examples:

```
rsv_create_region my_rr rr_if 8 "" "" // Create an RR and use the rr_if
portmap
rsv_create_region my_rr rr_if 8 "" "my_ei" // Also attach a my_ei error
injector to the RR
```

rsv_create_solyr

- Usage: `rsv_create_solyr` *simulation_only_layer_name target_fpga_family optional_scoreboard_name*
- Description: Define the simulation-only layer named *simulation_only_layer_name* which targets the specified FPGA family. The *target_fpga_family* argument can be either *VIRTEX4*, *VIRTEX5* or *VIRTEX6*. The simulation-only layer can have an optional scoreboard, where designers can add design-specific functional coverage items. In most cases, users do not have to create a scoreboard and can specify a null string "" to the *optional_scoreboard_name* argument. The automatically generated scoreboard collects coverage data for all possible transitions between RMs mapped to each RR.
- Examples:

```
rsv_create_solyr my_sim_only_layer VIRTEX5 "" // For DRS designs on VIRTEX-
5
rsv_create_solyr my_sim_only_layer VIRTEX5 my_scb // Also attach a my_scb
scoreboard to the simulation-only layer
```

rsv_gen_sbt

- Usage: `rsv_gen_sbt` *simb_file_name simb_op simb_fa simb_wc*
- Description: Generate the binary SimB file *simb_file_name*. The `.sbt` extension is automatically pended to the end of the SimB file name *simb_file_name*. The parameter *simb_op* can be either "WCFG" for reconfiguration or "RCFG" for configuration readback. The parameter *simb_fa* is the frame address of the generated SimB. The parameter *simb_wc* specifies the size of the configuration data section (i.e., 4x *region_size*). This API is typically automatically called by ReSim. However, advanced users may wish to use it to generate and customize SimB.
- Examples:

```
rsv_gen_sbt "./my" WCFG 0x01020000 32 // A SimB for DPR
rsv_gen_sbt "./my" RCFG 0x01020001 4 // A readback SimB for one frame
```

rsv_gen_solyr

- Usage: `rsv_gen_solyr` *`solyr_name`*
- Description: Generate source code for the simulation-only layer *`solyr_name`*, and the simulation-only bitstream to be used as test stimuli. It also generate a `TO_DO_LIST` file that explains the steps to instantiate and use the generated artifacts.
- Examples:

```
rsv_gen_solyr my_sim_only_layer // Generate the simulation-only layer
```

rsv_iei_hdl_state

- Usage: `rsv_iei_hdl_state` *`rr_inst rm_inst iei_sig_type iei_mem_type`*
- Description: Inject errors to *all* internal signals of module *`rm_inst`* in reconfigurable region *`rr_inst`*. This API injects errors to both signals and memory cells of the RTL design. The type of error can be ”none”, ”zero” OR. ”x” for signals, and ”none”, ”zero” OR. ”rand” for memory cells. This API is typically called in the error injector during reconfiguration. It can also be used in other places to inject errors. Limited by the implementation of ReSim, this task can only inject errors to Verilog registers (i.e., NOT Verilog nets or VHDL signals). User can use the `rsv_execute_tcl` macro to call such Tcl API (and other Tcl APIs) from SystemVerilog test code.
- Examples:

```
rsv_iei_hdl_state "/tb/dut/rr0" "rm0" "x" "rand" // Inject X and random
value to design object /tb/dut/rr0/rm0
rsv_iei_hdl_state "/tb/dut/rr1" "rm2" "x" "none" // Inject X to signals
only
```

rsv_parse_sbt

- Usage: `rsv_parse_sbt` *`simb_file_name`*
- Description: Parse the binary SimB file *`simb_file_name`* and dump the results to a text file named *`simb_file_name.txt`*. The `.sbt` extension is automatically pended to the end of the SimB file name *`simb_file_name`*. This API is typically automatically called by ReSim. However, advanced users may wish to use it to generate and customize SimB.

- Examples:

```
rsv_parse_sbt "./my" // Parse and dump a SimB ./my.sbt
```

Bibliography

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

[2] *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

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

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

[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] 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.

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

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

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

[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] 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.

[14] 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.

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

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

[17] 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.

[18] L. Sanders, *Simulation of an EDK System Which Uses the PLBv46 Endpoint Bridge for PCI Express (XAPP1111)*, Xilinx Inc., 2009.

[19] 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.

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

[21] 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.

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

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

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

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

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

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

[28] *7 Series FPGAs Configuration User Guide (UG470)*, Xilinx Inc., 2013.