Cairo University

DYNAMIC PARTIAL RECONFIGURATION VERIFICATION AND APPLICATIONS ON FPGA DEBUGGING

By

Islam Osama Ahmed Mounir Mostafa

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Electrical Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2018

DYNAMIC PARTIAL RECONFIGURATION VERIFICATION AND APPLICATIONS ON FPGA DEBUGGING

By

Islam Osama Ahmed Mounir Mostafa

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Electrical Communications Engineering

Under the Supervision of

Prof. Dr. Ahmed Nader Mohieldin

………………………………..

Associate Professor of Electronics and Communications
Department of Electronics and Electrical Communications Engineering
Faculty of Engineering, Cairo University

Dr. Hassan Mostafa Hassan

………………………………..

Assistant Professor of Nanoelectronics, Bioelectronics and Optoelectronics
Department of Electronics and Electrical Communications Engineering
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2018

DYNAMIC PARTIAL RECONFIGURATION VERIFICATION AND APPLICATIONS ON FPGA DEBUGGING

By

Islam Osama Ahmed Mounir Mostafa

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Electrical Communications Engineering

Approved by the
Examining Committee

____
Prof. Dr. First S. Name, External Examiner

____
Prof. Dr. Second E. Name, Internal Examiner

____
Prof. Dr. Third E. Name, Thesis Main Advisor

____
Prof. Dr. Fourth E. Name, Member

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2018

Engineer's Name:	Islam Osama Ahmed Mounir Mostafa
Date of Birth:	10/03/1990
Nationality:	Egyptian
E-mail:	islam.osama.ahmed@gmail.com
Phone:	01115037902
Address:	
Registration Date:	01/10/2013
Awarding Date:	
Degree:	Master of Science
Department:	Electronics and Electrical Communications Engineering

Supervisors:

Prof. Ahmed Nader Mohieldon
Dr. Hassan Mostafa Hassan

Examiners:

Prof. …………………	(External examiner)
Prof. …………………	(Internal examiner)
Porf. …………………	(Thesis main advisor)
Porf. …………………	(Member)

Title of Thesis:

Dynamic Partial Reconfiguration Verification and Applications on FPGA Debugging.

Key Words:
Dynamic Partial Reconfiguration (DPR); Verification; Debugging; Software Defined Radio (SDR); Field Programmable Gate Arrays (FPGA); Reconfigurable Systems.

Summary:

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………

Acknowledgments

Alhamdulillah, all praises and gratitude to Allah, the Almighty, for all his blessings and support me with the strength and health to complete this thesis. I would like to thank my academic supervisors Prof. Ahmed Nader and Dr. Hassan Mustafa for their guidance and help during the thesis work.

I would like to thank as well Eng. Ahmed Kamal (Former Research Assistant at ONE Lab) for helping me with working on the Dynamic Partial Reconfiguration flow. I would like as well to thank my managers and colleagues in Mentor Graphics for their support (Eman El-Mandouh, Haytham Shoukry, Khaled Nouh and Amr Abbas).

Last but absolutely not least, I want to extend my deepest and most sincere gratitude and thanks to my family for their support throughout my study years, and for Hala Ibrahim, my fiancé, for her support and help to write this thesis.

Table of Contents

List of Tables

List of Figures

Nomenclature

Abbreviation	Description
3G	Third Mobile Generation.
ABV	Assertion Based Verification
ASIC	Application Specific Integrated Circuit
BLE	Basic Logic Element
CAD	Computer Aided Design
CDC	Clock Domain Crossing
CLB	Configurable Logic Block
DAG	Directed Acyclic Graph
DFT	Discrete Fourier Transform
DPR	Dynamic Partial Reconfiguration
DRS	Dynamically Reconfigurable Systems
DSP	Digital Signal Processing
FF	Flip Flop
FIFO	First Input First Output
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
GPP	General Purpose Processor
HLS	High-level Synthesis
HDL	Hardware Description Language
HVL	Hardware Verification Language
ICAP	Internal Configuration Access Port
IFFT	Inverse Fast Fourier Transform

ILA	Integrated Logic Analyzer
IP	Intellectual Property
JTAG	Joint test Action Group
LTE	Long Term Evolution
LUT	Look Up Table
PC	Personal Computer
PCAP	Processor Configuration Access Port
PL	Programmable Logic
PS	Processing System
RM	Reconfigurable Module
RR	Reconfigurable Region
RTL	Register Transfer Level
Rx/Tx	Receiver/Transmitter
SC-FDMA	Single carrier-Frequency Division Multiple Access
SDR	Software Defined Radio
SoC	System on Chip
SVA	System Verilog Assertion

Abstract

Dynamic Partial Reconfiguration (DPR) on Field Programmable Gate Arrays (FPGAs) allows a portion of the logic to be reconfigured at runtime while the rest of the logic keeps operating. Such category of designs called Dynamically Reconfigurable Systems (DRS) designs. This feature enables the designers to build complex systems such as Software Defined Radio (SDR) in a reasonable area. Despite of the flexibility provided by the DPR, there are new challenges to design and verify the designs which utilize the DPR technique when it is compared to static FPGA systems.

In this thesis, a new verification methodology for DPR is presented. The new methodology addresses DPR specific logic and issues such as guaranteeing proper connections for the ports of the Reconfigurable Modules (RMs) which share the same Reconfigurable Region (RR) on the FPGA, waiting for running computations on a module before reconfiguring it, isolation of the reconfigurable modules during the reconfiguration process, and initialization of the reconfigurable module after the reconfiguration process is done. This DPR logic is verified using Assertion Based Verification (ABV) by modeling its functionality using System Verilog Assertion (SVA) properties, then instrument the design with these properties. Following that, these properties are verified using simulation or formal methods to check the correctness of the DPR logic. Also, this thesis presents an automated flow for Clock Domain Crossings (CDC) verification for DRS designs.

In addition, this thesis demonstrates the power of utilizing the DPR technique to minimize the cost of designing applications which perform time multiplexing of the digital logic, such as debugging of FPGAs. Because of the limited accessibility to the internal signals of the designs implemented on FPGAs, the debugging of FPGAs is a hard task. Embedded logic analyzers enhance the signal observability for FPGAs. These analyzers are implemented on the FPGA resources, and they use the embedded memory blocks as trace buffers, so a limited number of signals can be observed using these analyzers due to resources constraints. Changing the traced set of signals requires re-synthesis, placement and routing of the whole design. In this thesis, a new methodology for FPGA debugging is proposed to change dynamically the set of signals to be observed at runtime, and consequently, minimize the time required for debugging. The proposed methodology utilizes the DPR technique to dynamically switch between different sets of signals. DPR creates a reconfigurable module (RM) to route each set of signals to an embedded logic analyzer. The proposed approach is demonstrated using Xilinx FPGA tools, finding that changing the set of signals to be observed requires only few milli-seconds to re-program the reconfigurable region (RR). The area overhead of the proposed methodology is lower than other traditional methods of using multiplexers as the DPR allows the routing module to only use buffers to connect a set of signals to the embedded logic analyzer.

Chapter 1 : Introduction

Design and verification of Integrated Circuits (ICs) have become a complex task during the last two decades due to the need to integrate extra functionalities and applications into a single chip. Consequently, the costs of developing modern ICs have been multiplying. Such increase in the costs is representing a threat to the continuance of the semiconductor evolution [1]. The development cost has been estimated to reach over 0.17 billion US dollars for a chip at 28 nm technology node [2]. Moreover, the significant engineering efforts and investments do not minimize the possibility of the failure of the project. The cycle of development of the chips takes from few months to years with high uncertainty [1], and it includes a lot of testing and verification efforts to ensure the correctness of its functionality when it is fabricated.

The development of customized IC solutions is accompanied with huge risks and costs. Therefore, it is only justified for a small number of ultra-high volume electronic products. As a second choice, the electronics industry has started moving into using reconfigurable platforms such as FPGAs as computing platforms. The major advantage of an FPGA is that it can be configured at the design time of the system to implement a logic application, also it can be reconfigured at runtime and after deployment. The FPGA is considered a programmable type of integrated circuits. Compared with custom chips, the programmability of reconfigurable devices has enlarged the ability of easily modifying the designs while inserting acceptable overheads in performance, area, and power. The systems (either hardware or software) can realize shorter time to market when they are implemented on reconfigurable devices. Also, they are more responsive to bug fixes or upgrades throughout the product life cycle. By 2024, it is expected that, on average 70% of the chip functionalities will be reprogrammable [1].

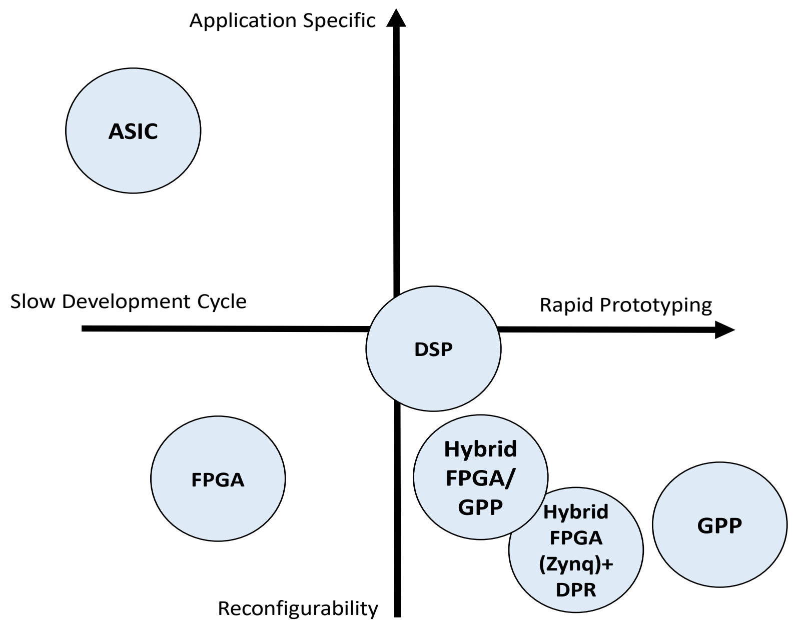


Figure 1.1: Trade-off between Different SDR Hardware Platforms [12]

In the recent years, the FPGAs capabilities are enhanced and developed to be more flexible and reconfigurable at runtime [10,11] by the introduction of the concept of DPR. DPR allows the FPGA to be reconfigured at runtime by reconfiguring a specific part on the FPGA without turning off the rest of FPGA. DPR pushes the FPGAs to become a promising reconfigurable hardware platform with a high degree of flexibility that allows it to be used as the target hardware platform for the implementation of complex systems such as SDR. Figure 1.1 shows the trade-off between design time and reconfigurability for different hardware platforms suitable for the hardware implementation of SDR [12]. As shown in Figure 1.1, applying DPR on the FPGA platform increase the reconfigurability of the FPGA to be more reconfigurable than traditional software programmable platforms such as the Digital Signal Processors (DSPs) and General Purpose Processors (GPPs). DPR offers the benefits of efficient resources utilization for the FPGA hardware resources as well as low power consumption for the SDR system.

Currently, there are more designs start targeting FPGAs while the amount of designs that target Application-Specific Integrated Circuits (ASICs) is in decline [4]. The vendors of FPGAs are now fabricating programmable System-on-Chip platforms, they are switching into ASIC markets (e.g., [3, 4]). Recently, there are new FPGA systems which permit sub-modules of hardware to be reconfigured partially at runtime while the rest of the system components keep operating, such FPGA systems are called Dynamically Reconfigurable Systems (DRS). The flexibility of the design is extended in DRS designs relative to traditional statically configured FPGA systems**:**

- By allowing the same physical reconfigurable region (RR) of the FPGA to serve and accommodate multiple reconfigurable hardware modules (RM), the required modules are being loaded on demand by the system, the switching can be automatically triggered or by user interference, which saves resource usage significantly, maximizes design density, and minimizes system cost [6].
- At runtime, the modules can be time-multiplexed to respond to the changes in the operation requirements of an application. For example, a networked multiport switch [7] and an SDR [8] reconfigure the processing logic of their protocol according to the protocol of the incoming traffic.
- The functionality of a system can be expanded at runtime, by reconfiguring the design with new modules. For example, when identifying suspected attacks for network flow analysis application, the application reconfigures one of its unused modules to implement an intruder detection module [9].

Figure 1.2 shows the idea of the DPR technique which is supported in the modern FPGAs. Figure 1.2.a shows the full configuration of the FPGA in which the application consumes big area. Figure 1.2.b shows that the size of the application can be decreased by utilizing the DPR technique, i.e. if this application has some blocks that not operating at the same time so such modules can be time multiplexed. Each module can be loaded to operate for a certain period of time then another module to be loaded. Figure 1.2.c shows that using DPR increases the size of the FPGA theoretically to realize more applications than regular FPGA configuration, this leads to a better utilization of the FPGA resources. This concept may also be generalized to different fields of study, in this thesis it is demonstrated on runtime debugging of FPGAs.

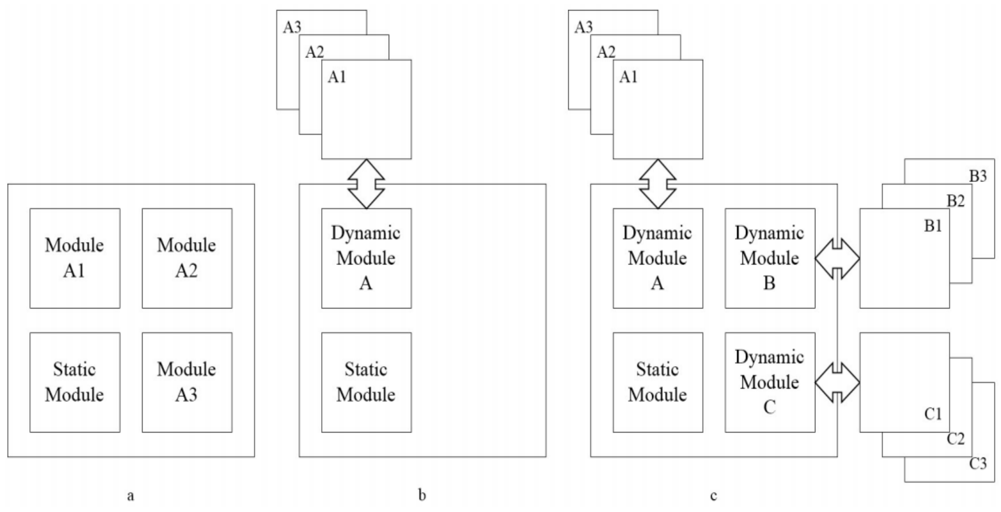


Figure 1.2: (a) Shows FPGA full configuration; (b) DPR technique is utilized to get the same system; (c) Shows how the FPGA size increased theoretically

Despite the flexibility provided by the DRS designs, there are more challenges to design and validate a DRS design compared with static conventional FPGA systems.

1.1. Design Flow of FPGA and DRS Designs

The typical flow of design is shown in Figure 1.3 for hardware systems targeting reconfigurable devices. The designer creates a specification document to fulfill and describe the functionalities of the design intent. After that, the designer uses a Hardware Description Languages (HDL) to translate the specification document into a Register Transfer Level (RTL) representation. Such translation process could also include re-using modules from previous projects or instantiation of Intellectual Property (IP) from third parties, and the IP is modeled as synthesized macros or HDL code. After that, the design is constrained by the designer, then it synthesized and implemented using Computer Aided Design (CAD) FPGA tools (e.g., Xilinx ISE [13]).

Also, high-level description languages such as SystemC [14] can be used to represent the design. In such case, High-Level Synthesis (HLS) tools (e.g., Vivado-HLS [15]) are used to synthesize the design to the target FPGA device. After this sequence of translations and design activities, the implemented design is programmed and downloaded on the target FPGA device and it is ready to run. In order to make sure of the correctness of the design and its functionality, each translation step should be verified and any change in the behavior or inconsistency in the representations between two successive steps is considered as a bug. Such errors or bugs should be fixed as early as they are identified, because the cost of the fixing an error or a bug is increased as designers go through the design flow. The bugs or errors that are introduced in the process of implementing the design such as timing violations and bad design constraints can be caught and identified using the vendor FPGA tools [13]. The errors and bugs injected into the specification and the translated design (i.e. human bugs) are called functional bugs. The process of identifying and fixing functional bugs to guarantee that the captured design fulfills and meets the intent of the design, is called functional verification [16].

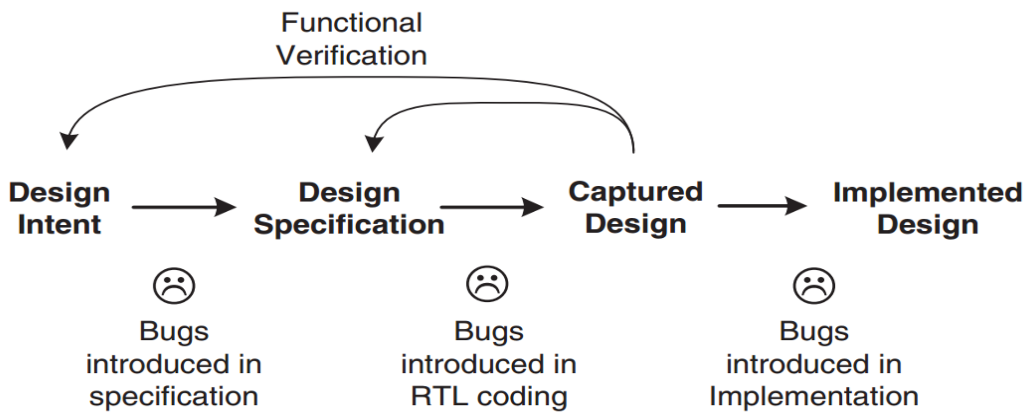

Functional Verification
Design Intent
Design Specification
Captured Design
Implemented Design
Bugs introduced in specification
Bugs introduced in RTL coding
Bugs introduced in Implementation

Figure 1.3: Hardware typical design flow

For the functional verification of ASIC or FPGA systems, simulation, and RTL simulation, especially, is the most widely used methodology. Off the shelf simulators, such as ModelSim [17] and ISim [18], compile and elaborate the captured design source (e.g., RTL code). Designers can review the waveforms which simulate the behavior of the design under some specified design inputs for all the signals in the design in order to debug errors that are injected into the design. A typical simulation environment is shown in Figure 1.4. Since functional verification only focuses on identifying functional bugs, simulation usually only involves the user design and does not include the physical layer.

The flow of design of DRS designs is similar to that of statically configured designs, except for few things. To explain the challenges and extra efforts needed to design a DRS, the modular reconfiguration flow [19,7] is considered as an example:

1. The design should be split into reconfigurable and static parts, and the designer has to design the application logic of the modules of these two parts. The static parts are those parts that operate during all the configuration modes of the design (i.e. they are needed all the time and cannot be shut down). Also, the reconfiguration mechanism of the system has to be added into the design to control and manage the process of reconfiguration. Such mechanism can be only hardware or a combination of both software and hardware.
2. The designer has to specify the border of the reconfigurable and static regions in order to lock down the signals traversing such borders. The designer also has to add placement constraints for modules, assign RRs to RMs, and generate partial bitstreams to configure the RR according to the modes of its associated RMs.

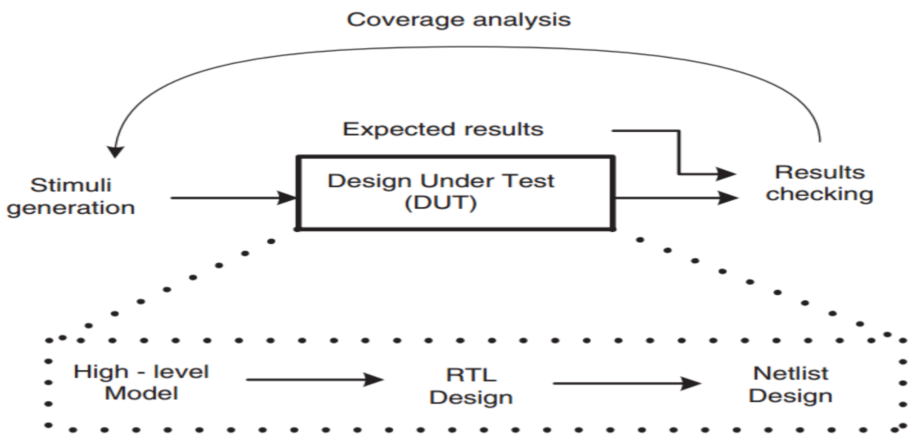


Figure 1.4: Typical simulation environment

The errors and bugs which are related to the implementation of the DRS designs can be detected and excluded by FPGA vendor tools which is similar to the case of the static designs. In particular, the vendors of FPGA tools define a group of rules for the physical layer design constraints, and it will do automatic check and verification for the DRS design against these rules. An example of such rules is that the wires or signals which are traversing the reconfigurable-static border should be assigned to the exact same FPGA resources for all the RMs. On the other hand, functional bugs in DRS designs cannot be automatically identified and caught by the FPGA vendors CAD tools.

It is the responsibility of the designers to check and verify the correctness of the captured design to make sure that it fulfills the intent of the design and meets the specification of the design. Consequently, the designers have to identify functional bugs that are injected into the system, which is similar to the case of the statically configured designs. In particular, since DRS designs include a newly added logic and a machinery for reconfiguration, the designer needs to verify that the reconfiguration logic and machinery are 1) correct which means that the reconfiguration modules needs to be verified standalone to make sure of the correctness of their functionality, and 2) are correctly integrated with the rest of the system, which means the reconfiguration components should be put into the integrated DRS design, and then the DRS design should be verified as a whole to verify and test the interactions of the reconfiguration components with the rest of the design's logic, which means that the testing of the reconfiguration mechanism's units as standalone components is necessary but not sufficient.

1.2. Functional Verification Challenges for DRS Designs

DPR offers a flexibility for designs of digital systems when being compared with static traditional FPGA designs. But, new challenges have been introduced into the functional verification of the design. In conventional simulation methodologies (such as RTL simulation), the hierarchy of the design is assumed to be always defined at compile time, such methods cannot understand the modules swapping during the simulation run. Furthermore, these traditional simulation tools cannot understand or interpret the configuration bitstreams which are used to reprogram the FPGA, only the FPGA device can interpret such configuration bitstreams.

Vendors of FPGA devices and CAD software, such as Xilinx, claim that every valid mode of configuration of a DRS can be tested separately by utilizing conventional simulation methodologies, but the simulation of the process of reconfiguration itself is not supported [7]. While, the behavioral simulation of the DPR process is proposed by Altera, but this simulation support has not been incorporated yet into Altera's tool flow [20]. Previous research works have proposed frameworks to support both high-level and RTL simulation for DRS designs. However, previously proposed frameworks fail to offer the precision needed to check and test the design being reconfigured. Hence, new simulation tools for the functional verification of DRS designs have to be developed.

Even if there are reliable tools available for simulation, it is not guaranteed if the well-established traditional methodologies of verification for statically configured designs are still applicable for usage with DRS designs. Particularly, since the design hierarchy and logic of a DRS design can be modified at runtime of the system, DPR come up with new testing cases that cannot be applied for statically configured designs. For example, in order to test if the RMs are stopped properly when a reconfiguration request arrives, the simulation environment needs to test partial reconfiguration in all possible states of the currently active RM. In order to verify that an ongoing reconfiguration doesn't inject any error (e.g., deadlock) to the rest of the design, the simulation needs to exercise all valid transitions between any two RMs. In this way, new rules and guidelines should be provided to the designers to aid the verification of the scenarios related to DPR in a design and ensure its correctness.

From the user design's point of view, the process of reconfiguration introduces new scenarios such as transferring partial bitstreams, and isolating, initializing, and synchronizing the RMs. These scenarios should be tested in simulation to ensure the correctness of the reconfiguration machinery, and verify the connections and communications between the whole design and the reconfiguration logic. From a timing perspective, the scenarios of reconfiguration can be classified as per the phase of the process of partial reconfiguration during which these scenarios may happen, i.e., AFTER, DURING, or BEFORE reconfiguration. Before reconfiguration, it is important to synchronize the process of reconfiguration according to the ongoing computations on the RMs of the DRS design, as an example for SDR systems if a packet is being processed for Wi-Fi standard, the computation should be completed before switching to another communication standard. During reconfiguration, it is important to properly isolate the RR being reconfigured in order to guarantee that no erroneous values will be propagated from the RM being reconfigured to the static logic or the output ports of the DRS design. After reconfiguration, the new loaded RM should be initialized to a known state to make sure of the correct operation of the RM, otherwise there will be undefined values or states propagated from the RM to the static part of the design.

1.3. Thesis Objectives

This thesis explores the functional verification of DRS designs that utilize DPR technique, and also explores the usage of the DPR to minimize the cost of runtime debugging for FPGAs as an application for the DPR technique. The main objectives of this thesis are:

1. Provide essential verification guidelines for functional verification of DPR.

2. Modeling the DPR logic and activities using System Verilog Assertion (SVA) [21].

3. Develop a technique to verify DPR using Assertion Based Verification (ABV) [22].

4. Provide a flow for Clock Domain Crossing (CDC) [23] verification for DRS designs.

5. Provide a technique to utilize DPR to minimize the cost of debugging on FPGA devices.

1.4. Organization of the Thesis

The thesis presents functional verification methodologies for DPR and DPR implementation to minimize the cost of FPGA debugging. The thesis is organized as follows.

Chapter 2 presents a summary on the FPGA as well as its construction. The details about DPR is introduced in this chapter as well.

Chapter 3 presents a functional verification methodology for DPR. The common issues for DPR logic are presented such as guaranteeing proper connections for the ports of the Reconfigurable Modules (RMs) which share the same Reconfigurable Region (RR) on the FPGA, waiting for running computations on a module before reconfiguring it, isolation of the reconfigurable modules during the process of reconfiguration, and initialization of the reconfigurable module after the process of reconfiguration is done. A verification methodology for the DPR logic using Assertion Based Verification (ABV) is presented and demonstrated on SDR system which utilizes DPR.

Chapter 4 presents an automated verification approach for Clock Domain Crossing (CDC) verification for DRS designs. A Perl utility is implemented to automate the generation of the RTL code for each operating mode of the design, and then the RTL is provided to a CDC CAD tool to verify the CDC signals in the design, the results of CDC verification of different operating modes of the design are collected and presented in a single report to the designer to ease the CDC verification process.

Chapter 5 presents the usage of DPR to minimize the cost of the FPGA debugging. The traditional FPGA debugging flow is presented as well as its drawbacks. The usage of DPR for FPGA debugging allows the designer to switch between different signals to be traced by the embedded logic analyzers at runtime, which reduce the total time taken for debugging on FPGAs.

Chapter 2 Overview about FPGAs and Dynamic Partial Reconfiguration

FPGAs were introduced almost thirty years ago. Since their first appearance, they have been rapidly-growing as a means of digital circuits' implementation. FPGAs great advantage is their flexibility, which arises from their programmable nature as compared to systems using ASICs [24]. In some cases, where the specifications of the system are time-dependent, not all modules need to operate concurrently. An unused module on the FPGA wastes power, area, and cost. So, it would be beneficial if a module is loaded only when its application is running, and removed when the application is done with the required computations [25]. Accordingly, a new concept has evolved in FPGA industry, which is known as dynamic partial reconfiguration (DPR). This new technology can be exploited in many applications, for example, to fulfill area requirements in small portable systems, to create a system-on-a-chip with a very high degree of flexibility, and to realize adaptive hardware algorithms [26].

In this chapter, various aspects of FPGA and FPGA dynamic partial reconfiguration are covered. First, an introduction of FPGA basics is presented to cover FPGA programming technologies, routing architecture, and software flow. Then, the FPGA reconfiguration technology is presented, such as reconfigurable logic and routing techniques, benefits of using partial reconfiguration, and partial reconfiguration in space and time.

2.1. FPGA Overview

FPGAs are pre-made silicon devices that can be electrically programmed to build any intended type of digital circuits or systems. They offer a number of competing advantages over ASIC technologies, such as standard cells. ASIC fabrication costs incomparable amount of time and money to obtain the first device. On the other hand, reconfiguration of an FPGA takes less than a second. But the flexible nature of an FPGA appears negatively as a significant cost in power consumption, delay, and area. As per the comparison of implementing digital designs on FPGAs versus standard cell ASIC [27], the speed performance for FPGAs is 2 to 4 times slower, the physical area for using FPGAs is 20 to 30 times bigger, and the consumption of power of FPGAs is 10 times higher. These drawbacks basically stand out from the FPGA's programmable routing fabric which trades power, speed, and area in return for immediate fabrication. The two essential technologies which distinguish FPGAs are architecture and CAD tools which users must adapt to build FPGA designs [24].

FPGAs, as shown in Figure 2.1, consist of an array of programmable logic blocks of noticeably different types, as follows [28]:

1. Programmable logic blocks, whose task is to implement logic functions.
2. Programmable routing blocks, which work on connecting these logic functions.
3. I/O blocks, which are wired to logic blocks by routing interconnects and make off-chip connections.

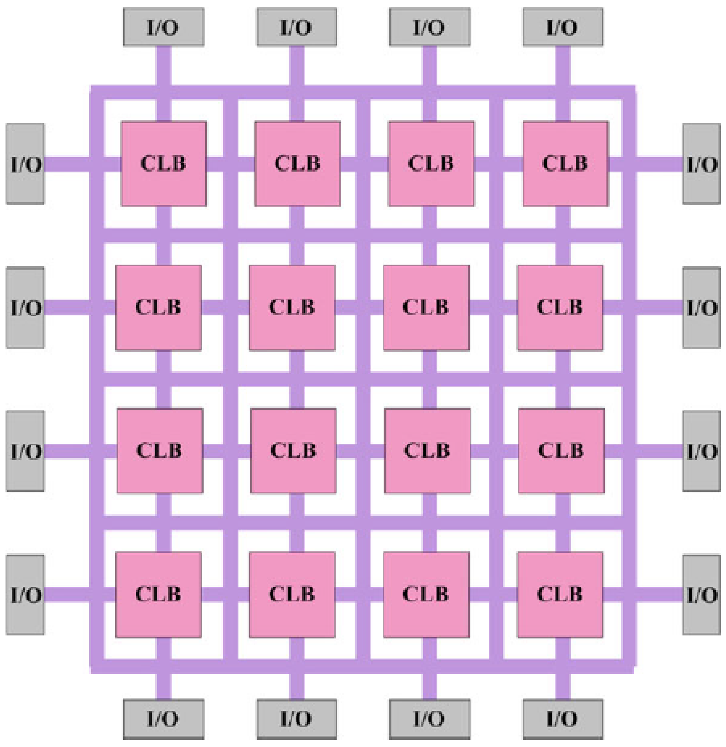


Figure 2.1: Basic FPGA structure [24]

2.2. FPGA Programming Technologies:

FPGA re-programmability depends on reconfigurable switches, which are controlled by an underlying programming technology. There are various technologies for FPGA programming, such as EPROM, EEPROM, flash [64], static memory [65], and anti-fuses [66]. The differences between these technologies have an outstanding influence on the architecture of the programmable logic. In modern FPGAs, only flash [64], static memory [65] and anti-fuse [66] technologies are commonly utilized. In this section, all modern technologies of FPGA programming will be reviewed to give a more comprehensive understanding of all technologies used in FPGA manufacturing.

2.2.1. **Static Memory**

Static memory cells are the building blocks for SRAM programming technology which is commonly utilized in Xilinx, Intel (Altera), and Lattice devices. In these devices, static memory cells are spread throughout the device to support configurability. An example for static memory cell is shown in Figure 2.2. SRAM cells are used for two main purposes. One of them is to control the values of the routing multiplexers' select lines, while the other one is to store the data in lookup-tables, which are used to implement logic functions. Figures 2.3 and 2.4 illustrate these two different approaches.

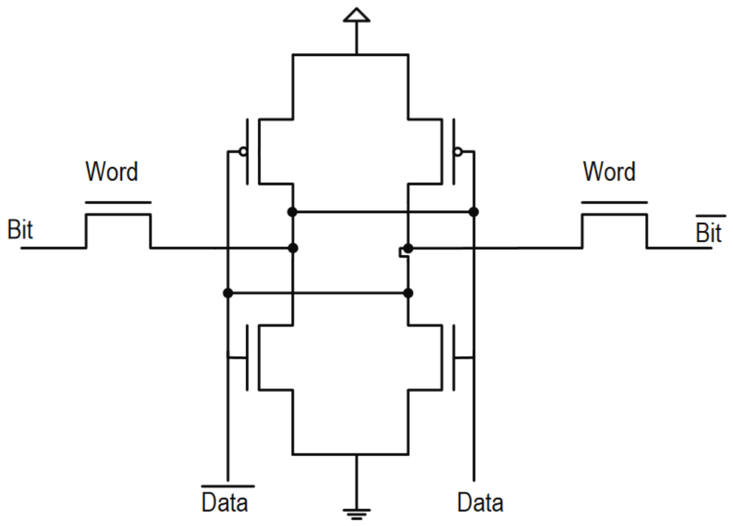


Figure 2.2: Static memory cell [28]

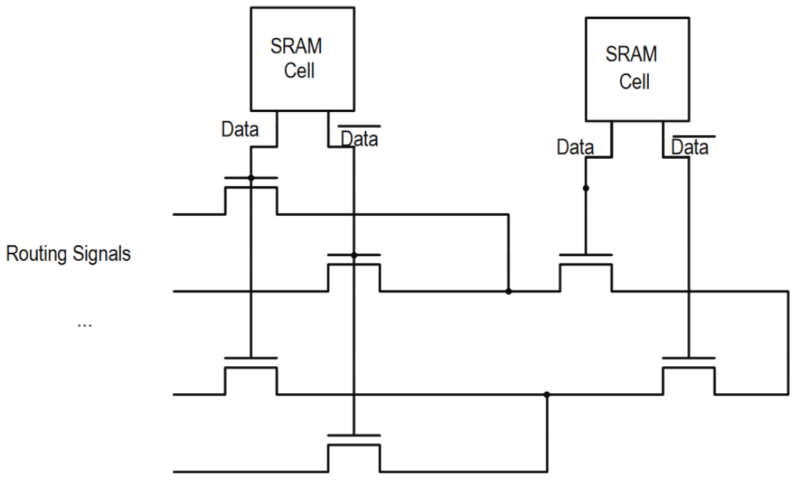


Figure 2.3: Multiplexer with static memory cell [28]

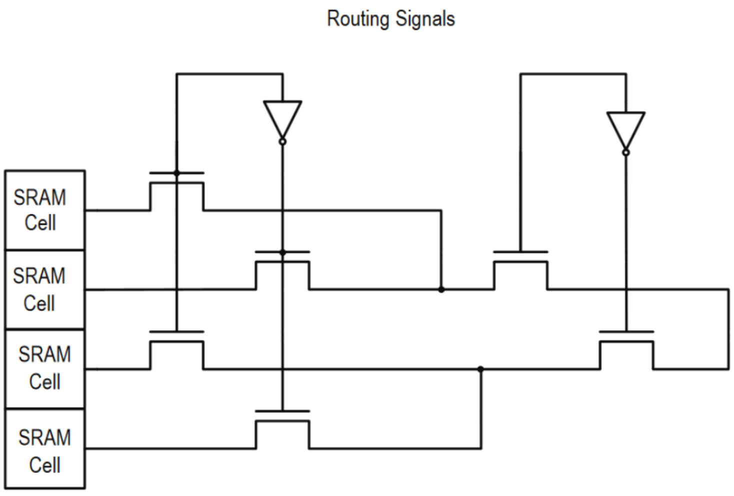


Figure 2.4: Static memory cells and lookup table [28]

SRAM technology is considered the most adequate programming technology for FPGAs because of two main reasons: compatibility with the standard CMOS fabrication process and re-programmability. Practically, an SRAM cell can be programmed an infinite number of times. A specific dedicated circuit on the FPGA does the task of initializing all SRAM bits on power up and configures the bits with a user-defined configuration. Unlike other technologies of FPGA programming, the utilization of SRAM cells needs no special IC processing beyond standard CMOS. So, SRAM-based FPGAs can use the latest CMOS technology available, and therefore, make use of the increased integration, the enhanced speeds, and the minimized dynamic power dissipation of new processes with smaller minimum geometries. However, SRAM-based programming technologies have the following disadvantages:

(1) Size. An SRAM cell consists of either 5 or 6 transistors and the programmable element used to interconnect signals needs at least a single transistor.
(2) Volatility. The volatility of the SRAM cell requires the use of external devices for permanent storage of configuration data when the device's power is down. These external flash or EEPROM devices are an added cost to SRAM-based FPGA [67].
(3) Security. The possibility of the configuration information being viewed or stolen for use in a competing system exists. This is due to having configuration information loaded into the device at power up stage. Currently, some FPGA families secure the configuration information through the use of encryption systems [68].
(4) Electrical properties of pass transistors. SRAM-based FPGAs surely depend on the use of pass transistors to implement multiplexers. However, they are not considered perfect switches as they have high on-resistances and present a significant capacitive load.

2.2.2. Flash Programming Technology

One substitute that addresses some of the limitations of SRAM based technology is the use of floating gate programming technologies that inject charges onto a floating gate above the transistor. This methodology is used in flash or EEPROM memory cells. These cells are non-volatile; in other words, they do not lose electrical signals (information) when the device is turned off. Traditionally, EEPROM memory cells were mainly used to implement wired-AND functions in PLD devices. They were not used directly to switch FPGA signals [69].

Such methodologies are no longer used because of their static power consumption, they are only used for very low-capacity devices. With modern IC manufacturing techniques, it is possible to implement switches using floating gate cells. Particularly, flash memory cells are used due to their area competence. The extensive use of flash memory cells for non-volatile memory chips guarantees that flash fabrication processes will benefit from steady reductions in process geometries. Figure 2.5 illustrates the flash-based approach used in Actel's ProASIC devices [59].

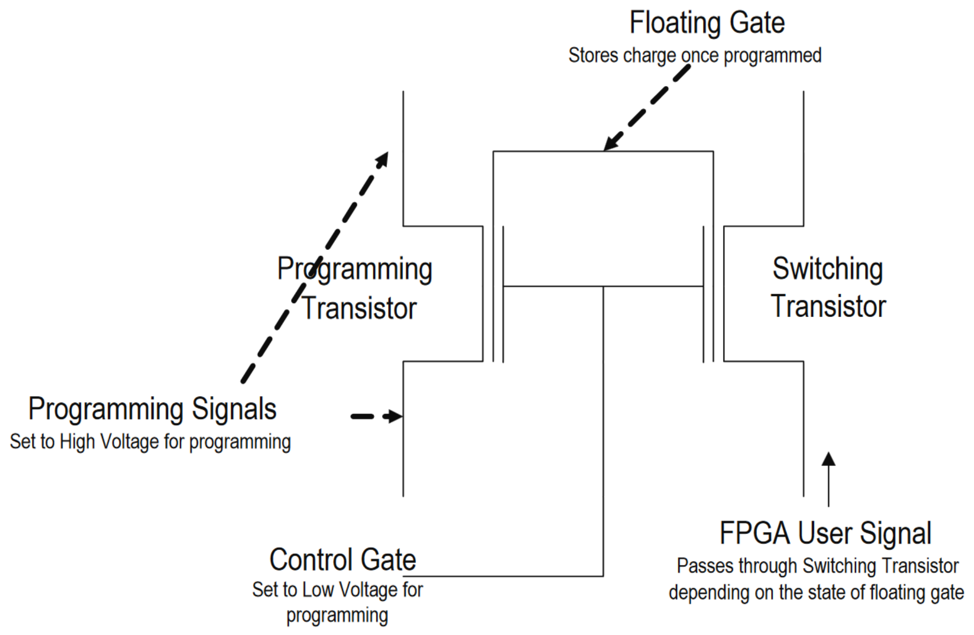

Floating Gate
Stores charge once programmed
Programming Transistor
Switching Transistor
Programming Signals
Set to High Voltage for programming
Control Gate
Set to Low Voltage for programming
FPGA User Signal
Passes through Switching Transistor depending on the state of floating gate

Figure 2.5: Floating gate transistor [28]

2.2.3. **Anti-fuse programming technology**

Anti-fuse FPGA programming technology is used as an alternative to SRAM and floating gate-based technologies. This technology depends on structures, which reveal very high-resistance under ordinary surroundings, but can be re-programmed to create a low resistance connection. This link is permanent if compared to floating gate or SRAM programming technologies. The programmable component, an anti-fuse, is directly used for propagating FPGA signals. The major advantage of anti-fuse programming technology is the drop in programmability area overhead. As there is no silicon area required to establish connections, only metal-to-metal anti-fuses. But, this area reduction is compensated by the need for large programming transistors, which are needed for the anti-fuse programming to provide the large currents required to program the anti-fuses [28]. This area can be paid back with clever programming architecture, which contributes considerably to the overall area. An added advantage to the anti-fuse technology is that they have lower parasitic capacitances and on resistances than other programming technologies. As a result, it is possible to include more switches per device than that of other technologies. Also, the whole system cost is reduced as there is no need for additional memory for storing programming information as the device works instantly once programmed. Programming and transmitting the bitstream to the FPGA need only to be done once. As a result, this can be done in a secure environment which improves the security of the design on the FPGA [70].

This programming technology still has some disadvantages. Specifically, anti-fuse-based FPGAs require a nonstandard CMOS procedure; they are typically late in the manufacturing processes that they can adopt compared to SRAM-based FPGAs. Moreover, scaling challenges emerge when considering new IC fabrication processes as the fundamental mechanism of programming using this technology requires significant changes to the properties of the fuse materials.

2.3. Configurable Logic Blocks

The elementary component of an FPGA, which provides the basic logic and storage functionality for a target application design, is the configurable logic block (CLB). In order to provide the fundamental logic and storage capability, the basic unit can be either a transistor or an entire processor. However, this example is very extreme. For the transistor example, which is in a very simple form, and requires a large amount of programmable interconnect. That leads to an FPGA that might suffer from area-inefficiency, low functionality, and high power dissipation. On the other hand, for the processor example, the basic logic block is very sophisticated and cannot be used to implement small functions as it will lead to resource waste. As a compromise of these two extremes, there exists a range of basic logic blocks. Some of them include logic blocks that are made of NAND gates, an interconnection of Multiplexers (MUXes), Look Up Table (LUT), and Programmable Array Logic (PAL) style with wide input gates [71].

LUT-based CLBs are used by commercial vendors, such as Intel (Altera) and Xilinx. These vendors use LUT-based CLBs to offer fundamental logic and storage functionality. LUT-based CLBs offer a good trade-off between too simple and too complicated logic blocks. A CLB can consist of one Basic Logic Element (BLE), or a cluster (i.e. group) of BLEs which are locally interconnected, as shown in Figure 2.7. The basic component of a simple BLE is a LUT, and a Flip-Flop (FF). A LUT with n inputs (LUT-n) contains 2^n configuration bits and it can implement any n-input boolean function. Figure 2.6 shows a simple BLE comprising of a 4 input LUT (LUT-4) and a D Flip-Flop. The LUT-4 uses 16 SRAM bits to implement any 4-inputs boolean function. The output of LUT-4 is connected to an optional Flip-Flop. A multiplexer selects the BLE output to be either the output of a Flip-Flop or the LUT-4. Additionally, a CLB can contain a cluster of BLEs connected through a local routing network. Figure 2.7 shows a cluster of four BLEs; each BLE consists of a LUT-4 and a FF. The BLE output is accessible to other BLEs of the same cluster through a local routing network. The number of cluster’s output pins equals the total number of BLEs in a cluster. However, the number cluster’s input pins can be less than or equal to the summation of input pins required by all the BLEs in the cluster. Modern FPGAs contain typically 4 to 10 BLEs in a single cluster [69].

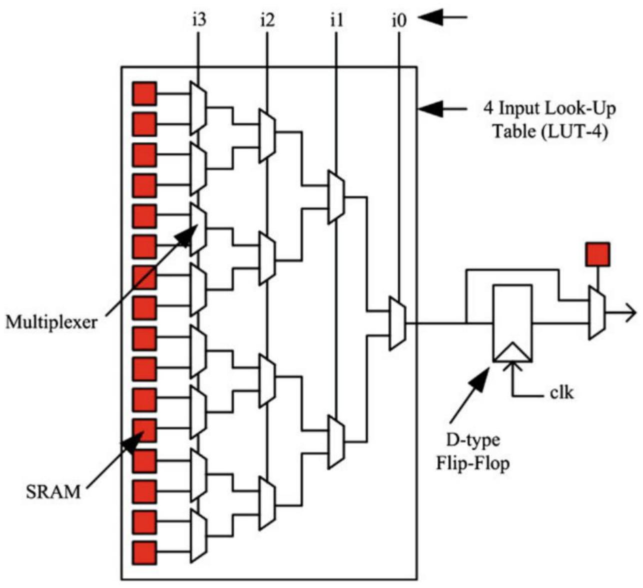


Figure 2.6: Basic logic element (BLE) [24]

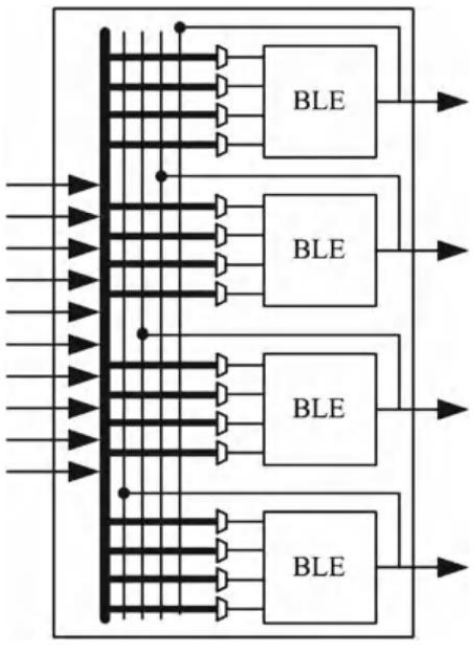


Figure 2.7: A CLB having four BLEs [24]

2.4. FPGA Routing Architectures

Programmable logic blocks provide computing functionality. These blocks are connected through re-programmable routing network, which provides routing for any pre-defined circuitry through enabling/disabling connections among I/O and logic blocks. Wires and programmable switches are the main component of FPGA interconnects. The used programming technology is responsible for the configuration of these programmable switches. Since it has been known that any digital circuit can be implemented on FPGA architecture, the flexibility of FPGA routing interconnects is a must. So, they can adopt a wide-ranging diversity of circuits, which require variable routing limitations. FPGA routing connects can be designed in an optimum way if they support specific common features of routing requirements of most circuits (taking into consideration that these requirements might differ from a circuit to another). For instance, for designs that require locality, considerably-short wires are to be used. Yet simultaneously, there might be some detached connections, which will need thin, but long wires. Consequently, both flexibility and efficiency need to be considered during the design of routing interconnects for FPGA. The relative arrangement of both architecture logic blocks and routing resources must be well-thought-out, as it dramatically affects the overall architecture efficiency. This arrangement is labeled here as global routing architecture, while the tiny details regarding the switching topology of different switch blocks are labeled as detailed routing architecture. According to the routing resources global arrangement, FPGA architectures can be classified to either island-style or hierarchical.

2.4.1. Island-Style Routing Architecture

Figure 2.8 shows traditional island-style FPGA architecture, which is also known as mesh-based FPGA architecture. From both academic and industrial point of view, island-style architecture is the most widely-used architecture. The reason behind this naming convention (island-style) is that in this architecture, configurable logic blocks look exactly like islands surrounded by a sea of routing interconnects. CLBs are organized on a 2D grid and are connected internally by a programmable routing network. The peripheral (I/O) blocks are also connected to the programmable routing network.

The routing network includes pre-manufactured wiring segments and programmable switches that are organized in vertical and horizontal routing channels. 80-90% percent of FPGA total area is occupied by the routing network, while only 10-20% of the total area is occupied by the logic blocks. The flexibility of an FPGA totally depends on programmable routing network. A mesh-based FPGA routing network consists of vertical and horizontal routing channels, which are connected through switch boxes (SB). Connection boxes (CB) are used to connect logic blocks to the routing network. The flexibility of a connection box (Fc) is calculated as the number of routing tracks of the neighboring channel connected to the pin of a block. Fc(in) is the connectivity of logic blocks input pins with the neighboring routing channel, whereas Fc(out) is the connectivity of logic block output pins with the neighboring channel. For example, if Fc(out) equals 1, it indicates that all neighboring routing channel tracks are connected to logic blocks output pins.

Architecture channel width is calculated as the number of tracks in routing channel. The very same channel width is used for all vertical and horizontal architecture's routing channels. Commonly, pass transistors are used to connect a block's output pins to routing tracks. Each pass transistor creates a tri-state output that can be turned on/off individually. Nevertheless, the technique of single-driver wiring can similarly be used to connect output pins of a block to the neighboring routing tracks. Tristate logic cannot be used in single-driver (unidirectional) wiring as the block output needs to be connected to the neighboring routing network through multiplexors in the switch box. The commercial trend in FPGA made modern FPGA architectures move towards using single-driver, directional routing tracks. It has been proven that 9% improvement in delay, 25% improvement in area, and 32% improvement in area-delay can be accomplished if the single-driver directional wiring is used instead of bidirectional wiring [24]. All these gains are attained without any major changes in the CAD flow of FPGA. Variable-length wires are created to reduce delay in mesh-based FPGAs. Figure 2.9 shows an example of dissimilar length wires. Longer wire segments go across multiple blocks requiring fewer switches, thus decreasing routing delay and area. On the other hand, routing flexibility is reduced, which decreases the probability to route a hardware circuit efficaciously. Up-to-date commercial FPGAs frequently use a permutation of short and long wires to balance routing network area, delay, and flexibility [72].

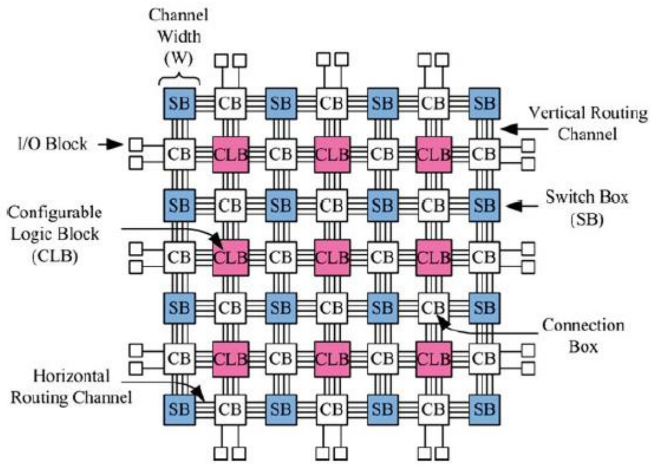


Figure 2.8: Overview of mesh-based FPGA architecture [24]

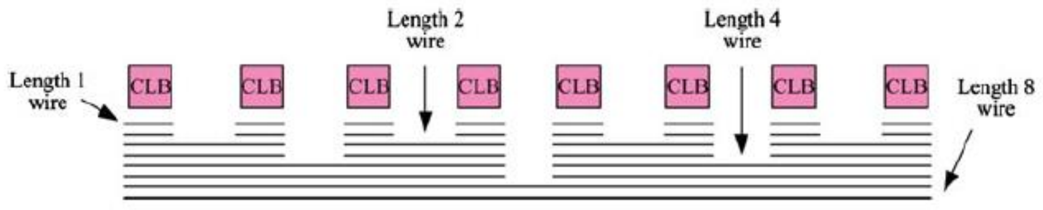


Figure 2.9: Distribution of channel signal [24]

2.4.2. Hierarchical Routing Architecture

Most logic designs demonstrate locality of connections; therefore indicating a hierarchy in connections placement and routing between different logic blocks. Hierarchical routing architectures take advantage of the locality principle by dividing FPGA logic blocks into individual clusters. These clusters are recursively connected to create a hierarchical structure. In a hierarchical architecture, connections between logic blocks within the same cluster are made by wire segments at the hierarchy lowest level. Though, the connection between blocks existing in different groups involves the traversal of one or more hierarchy levels. The signal bandwidth varies as it moves further from the bottom level and generally it reaches its widest at the top level of hierarchy in a hierarchical architecture. A large number of commercially-based FPGAs families use the hierarchical routing architecture, such as Altera Flex10K, Apex and ApexII architectures [73].

2.5. Software Flow

FPGA architectures have been strongly explored for the past 20 years. A key aspect of FPGA architecture research is the improvement of CAD tools for mapping applications to FPGAs. It is well recognized that the superiority of an FPGA-based implementation is largely defined by the efficiency of the associated suite of CAD tools. Benefits of a well-designed, feature-sufficient FPGA architecture might be compromised if the CAD tools cannot take advantage of the features that the FPGA supports. Thus, research in CAD algorithms is essential to the architectural advancement to fill the performance gaps between other computational devices, such as ASICs. The software flow takes an application design described in HDL language and converts it to a stream of bits that is actually programmed on the FPGA. The procedure of altering a circuit description into a format that can be loaded into an FPGA can be divided into five distinct steps, which are: synthesis, technology mapping, clustering, placement, and routing. FPGA CAD tools' final output is a bitstream that configures the state of the memory bits in an FPGA. The state of these bits determines the logical function that the FPGA implements. Figure 2.10 shows a comprehensive software flow for programming an application-specific circuit on an FPGA. A description of several steps of software flow is given in the following part of this section. The details of these steps are usually similar to the kind of routing architecture used and they can be applied to both architectures described earlier.

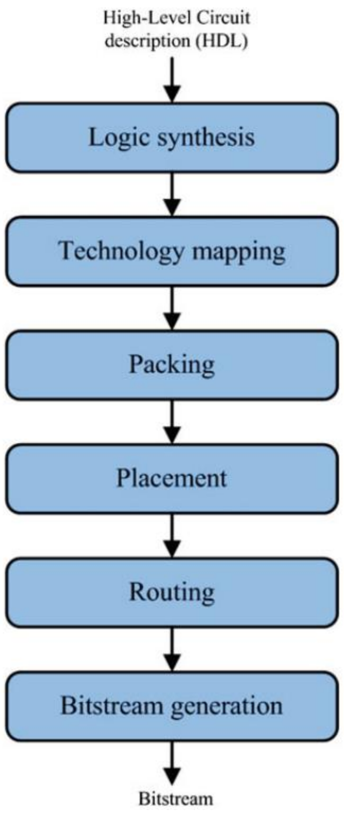


Figure 2.10: FPGA software basic flow [24]

2.5.1. Logic Synthesis

The FPGA flow begins with the logic synthesis of the netlist mapped on it. Logic synthesis transforms an HDL code (Verilog or VHDL) into a group of boolean Flip-Flops and gates. The synthesis tools transform the RTL interpretation of a design into a hierarchical boolean network. Numerous technology-independent methodologies are being applied to optimize the generated boolean network. The conventional cost of optimizations which are technology-independent is the total exact count of the factored representation of the logic function. Such count is directly proportional to the area of the circuit [74].

2.5.2. Technology Mapping

Synthesis tools output is a netlist. The netlist contains a circuit description of boolean logic gates, wiring connections, and flip-flops between these elements. The circuit can similarly be characterized by a Directed Acyclic Graph (DAG). Each node in the graph represents a gate, a primary input/output, or a flip-flop. Each edge in the graph symbolizes a connection between two circuit elements. Figure 2.11 demonstrates an example of a circuit DAG representation. Given a library of cells, the technology mapping problem can be stated as finding a network of cells that implement the boolean network. In the problem of technology mapping for FPGAs, the library of cells consists of n-input flip-flops and LUTs. Thus, technology mapping for FPGA includes converting the boolean network into n-bounded cells. After that, each cell is implemented as an independent n-LUT. Figure 2.12 shows an example of transforming a Boolean network into n-bounded cells. Algorithms of technology mapping can optimize a design for a set of goals including power, area, or depth. The FlowMap [64] algorithm is the most widely used tool for FPGA technology mapping in academic research. FlowMap is able to find a depth-optimal solution in polynomial time and promises depth optimality as a return of logic duplication. Hence, it is considered a great discovery in technology mapping for FPGAs. After the first presentation of FlowMap, a lot of technology mapping tools have been designed that optimize for run-time and area while still maintaining the depth-optimality of the circuit. The result of the technology mapping step generates a network of n-bounded LUTs and flip-flops.

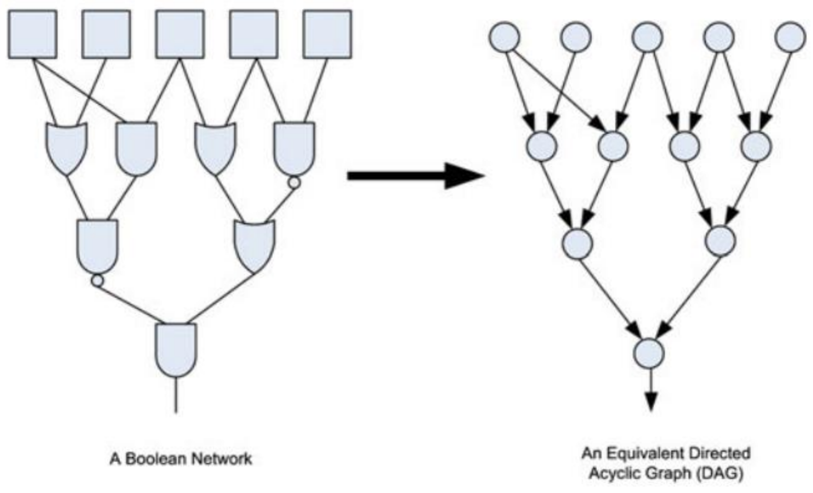


Figure 2.11: DAG representation of a circuit [24]

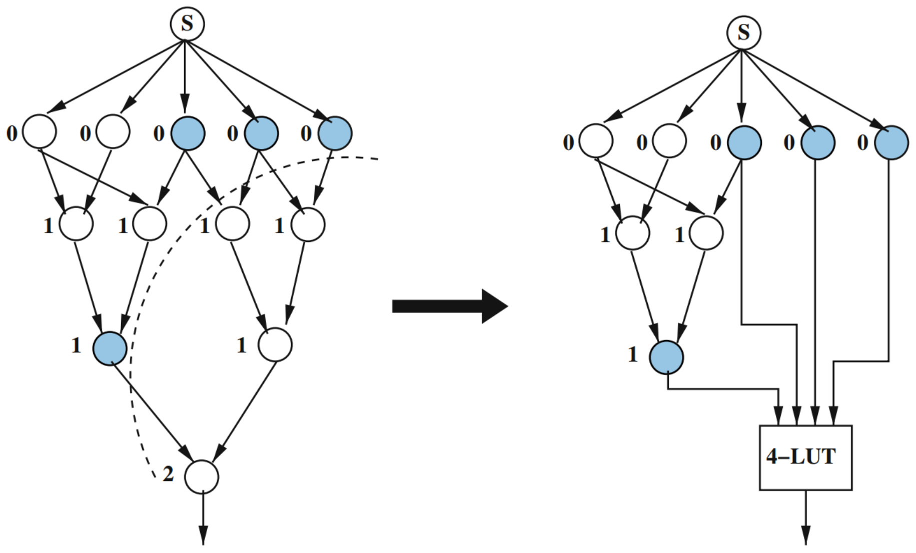


Figure 2.12: Example of technology mapping [24]

2.5.3. Clustering/Packing

The logic elements in Mesh-based FPGAs are naturally arranged in two levels of hierarchy. The first level contains LBs which are flip-flops and n-input LUT pairs. The second level hierarchy combines each k LBs together to create logic blocks clusters. The clustering stage of the FPGA CAD flow is the process of creating groups of k LBs. These clusters can then be mapped instantly to a logic element on an FPGA. Figure 2.13 shows an example of the clustering process. Clustering algorithms can be roughly classified into three general methodologies, which are depth-optimal, top-down, and bottom-up. Depth-optimal methodology tries to decrease delay at the expense of logic replication [75]. Top-down methodology divides the LBs into clusters by consecutively subdividing the network or by iteratively moving LBs between parts [76]. The bottom-up methodology is commonly favored for FPGA CAD tools due to their fast run times and sensible timing delays [77]. They consider only the information of local connectivity and can simply meet constraints of clusters pin. The top-down approaches offer the best solutions. But, they still have the disadvantage of unaffordable computational complexity.

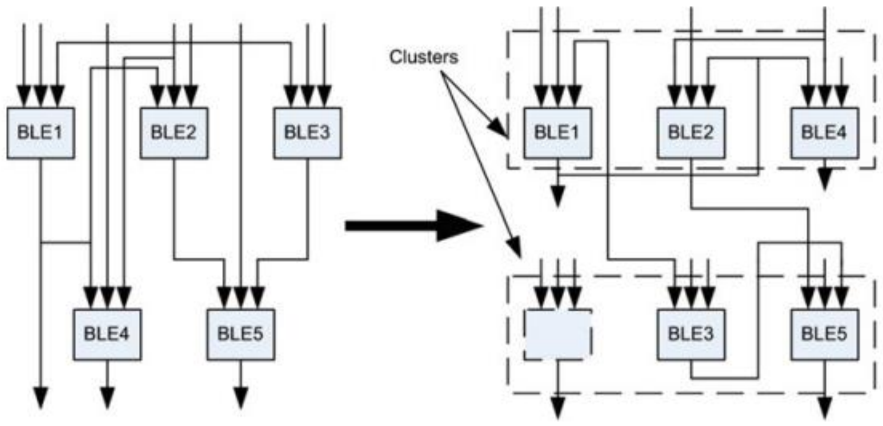


Figure 2.13: packing example [24]

2.5.4. **Placement**

Determination of which logic block in an FPGA should implement the corresponding logic block required by the circuit is the responsibility of placement algorithms. The optimization objectives are to locate connected logic blocks close to each other to decrease the required wiring, and sometimes to locate blocks to balance the wiring density across the FPGA or to take full advantage of circuit speed. The 3 major approaches of placers used nowadays are min-cut [78], analytic [79], which are often followed by local iterative enhancement, and simulated hardening based placers [80]. To inspect architectures objectively, users must validate that CAD tools are trying to use every FPGA's feature. This means that the optimization approach and objectives of the placer might be altered from architecture to another. The most commonly-used in FPGA CAD tools are partitioning and simulated hardening approaches.

2.5.5. **Routing**

The routing problem of an FPGA lies in assigning nets to the routing resources to guarantee that no routing resource is being shared by more than one net. Path finder is the current and most up-to-date FPGA routing algorithm [81]. Path finder operates on a directed graph abstraction G(V,E) of the routing resources in an FPGA. The set of vertices V in the graph represents the I/O terminals of logic blocks and the routing wires in the interconnect structure. An edge between two vertices represents a possible connection between them. Figure 2.14 represents part of the routing graph in a Mesh-based interconnect. Given this graph, finding a directed tree that is embedded in G and connects the source and sink terminal together is the definition of the routing problem. Because there is an inadequate number of routing resources in an FPGA, the aim of finding non-intersecting, unique trees for all the nets in a netlist is a challenging problem. Path finder uses an iterative, negotiation-based methodology to fruitfully route all the nets in a netlist. Nets are easily routed without taking care of resource sharing only during the first routing iteration. Individual nets are routed using Dijkstra's shortest path algorithm [82]. Resources may be overcrowded because many nets have used them at the end of the first iteration. During subsequent iterations, the cost of using a resource is greater than before, depending on the history of congestion on the resource and the number of nets that share that resource. If a resource is highly congested, nets which can use lower congestion alternatives are forced to use this capability. In contrast, if the alternatives are more overcrowded than the resource, then a net may still use that resource.

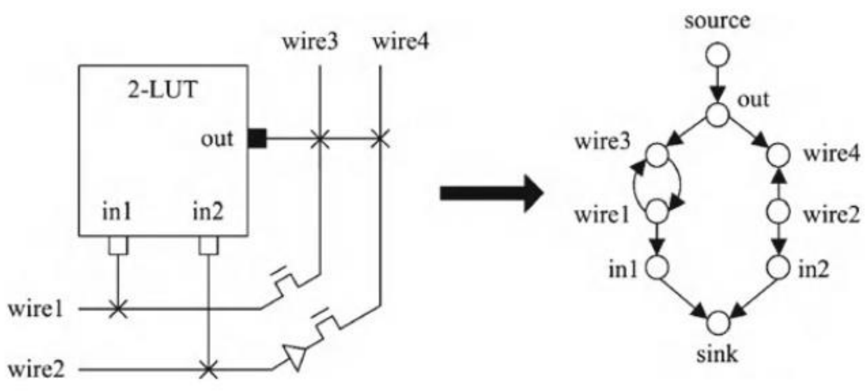


Figure 2.14: Modeling FPGA architecture as a directed graph [24]

2.5.6. Timing Analysis

Timing analysis is used for two main motives; first, to specify the circuits' speeds, which have been entirely placed and routed, and second, to estimate the slack of each source-sink connection during routing in order to decide which connections must be made through fast paths to avoid slowing down the circuit [83].

At the beginning, the considered circuit is presented as a directed graph. Nodes in the graph symbolize input and output pins of circuit elements such as LUTs, I/Os, and registers. Connections between these nodes are shown with edges in the graph. Edges are added between the inputs of combinational logic Blocks (LUTs) and their outputs. These edges are marked with a delay consistent with the physical delay between the nodes. Register input pins are not joined to register output pins. A traversal is done on the graph starting at sources to identify the circuit delay. Then the arrival time ($T_{arrival}$) at all nodes in the circuit is computed with the following equation:

$$T_{arrival}(n) = \max_{m \in fanin(n)}\{T_{arrival}(m) + delay(m,n)\} \qquad (1)$$

Where node n is the node currently being analyzed, and delay(m, n) is the delay value of the edge connecting node m to node n. The circuit delay is then calculated as the maximum arrival time, D_{max}, of all the circuit nodes. For guiding a placement or routing algorithm, it is beneficial to know how much extra delay may be inserted into a connection before the path that the connection is on becomes critical. The amount of extra delay that may be inserted into a connection before it becomes critical is called the slack of that connection. To calculate the slack of a connection, one must calculate the required arrival time, $T_{required}$, at all the nodes in the circuit. The $T_{required}$ is adjusted at all sinks (register inputs and output pads) to be D_{max}. Required arrival time is then propagated backward starting from the sinks with the following equation:

$$T_{required}(n) = \min_{m \in fanout(n)}\{T_{required}(m) - delay(m,n)\} \qquad (2)$$

Finally, the slack of a connection (n, m) driving node, m, is defined as:

$$Slack(n,m) = T_{required}(m) - T_{arrival}(n) - delay(n, m) \qquad (3)$$

2.5.7. Bitstream Generation

Bitstream information is generated for the netlist immediately after a netlist is placed and routed on an FPGA. A bitstream loader is used to program this bitstream on the FPGA. The bitstream of a netlist contains information of which SRAM bit of an FPGA is programmed to a logic value of 0 or 1. The bitstream generator reads the technology mapping, packing, and placement information to program the SRAM bits of Look-Up Tables. Finally, the routing information of a netlist is used to correctly program the SRAM bits of both connection and switch boxes.

2.6. Dynamic Partial Reconfiguration

DPR is a feature of SRAM-FPGAs that offers the benefit of flexibility to reconfigure a part of FPGA at runtime with reusing the same hardware resources [60]. Xilinx DPR design flow imposes the splitting of the design into a dynamic part and a static part [7] as shown in Figure 2.15. The dynamic part consists of the reconfigurable modules (RMs) of the system, whereas the static part consists of the static modules that are not changed during the reconfiguration (i.e. they are available in all the operating modes of the design). The dynamic part contains multiple Reconfigurable Regions (RRs). Each RR is used for a set of RMs, which can be swapped during runtime without disruption. A partial bitstream is generated for each RM to be mapped into a specific RR during reconfiguration. Partial bitstreams are loaded from a non-volatile memory to the FPGA configuration memory using dedicated configuration interfaces. DPR are categorized based on the configuration modes as internal or external reconfiguration methodologies, based on how the reconfiguration is handled either internally within the FPGA or via an external device such as a PC or another FPGA. Xilinx 7-series FPGAs have two internal configuration interfaces to the FPGA configuration memory [61]: (i) The Internal Configuration Access Port (ICAP) that is physically located on the FPGA fabric. (ii) Processor Configuration Access Port (PCAP) only available for the Xilinx 7-series Zynq FPGA equipped with a hard macro ARM processor. Also, three external configuration interfaces are used through the serial configuration ports: JTAG, Serial mode, and Select-Map.

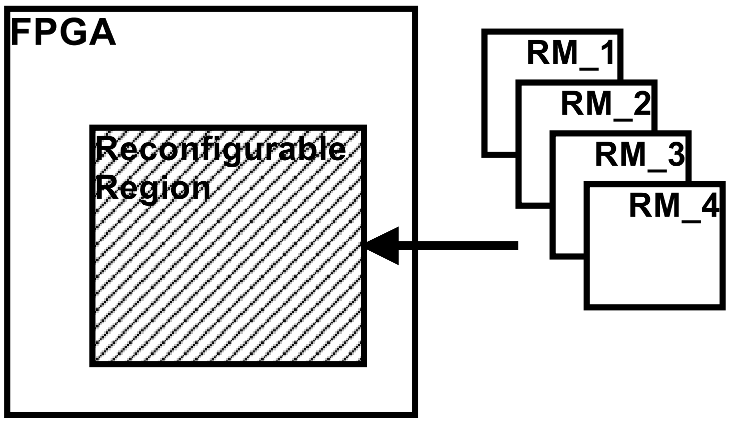


Figure 2.15: Dynamic Partial Reconfiguration in SRAM-FPGAS.

2.6.1. Configuration Modes

DPR can be performed by loading RMs partial bitstreams to the FPGA configuration memory. Accessing the configuration memory is achieved through numerous FPGA configuration modes or configuration ports [61]. Configuration modes are classified according to the type of configuration interface used to access the configuration memory. Table 2.1 shows the different configuration modes for Zynq FPGA.

2.6.1.1. External Modes

External configuration modes use external FPGA interfaces to load the partial bit files to the configuration memory of the FPGA. JTAG is the only external configuration port for Zynq FPGA. The partial bitstreams are transferred from an external memory storage source, for example, the PC through the JTAG serial interface to the configuration memory. The data rate of the JTAG configuration interface is limited to 8.25 MB/S and not suitable for real-time application such as the SDR system

[62] and requires an external PR controller, such as CPU or another FPGA to control the process of reconfiguration.

2.6.1.2. Internal Modes

Internal configuration modes use internal FPGA interfaces to load the partial bit files to the FPGA configuration memory. Two internal configuration modes are used in Xilinx Zynq FPGA. 1) ICAP configuration mode is based on the ICAP hard macro 32-bit configuration port primitive located on the PL side to access the configuration memory with a theoretical data rate of 400 MB/S. 2) PCAP configuration mode is based on the PCAP 32-bit configuration port in the PS side controlled by the ARM processor to access the configuration memory with a data rate of 400 MB/S:

Table 2.1: Configuration Modes of Zynq FPGA

Configuration Mode	Type	Max Clock	Data Width	Max Bandwidth
ICAP	Internal	100 MHZ	32-bit	400 MB/S
PCAP	Internal	100 MHZ	32-bit	400 MB/S
JTAG	External	66 MHZ	1-bit	8.25 MB/S

2.6.2. Advantages and Disadvantages of DPR

The main advantages of the reconfigurable systems are:

1. Resources utilization: in traditional design implementation, most of the hardware resources are not used at till the time when it is activated to operate for a certain period of time. Using reconfigurable hardware and DPR will increase the resource utilization by only implementing the active part of the design in the required time and time multiplexing the resources between the design hardware modules in consistence with activity schedule.
2. Scalability: using reconfigurable hardware allows upgrading system to accommodate freshly defined tasks to handle the growth in technology and features. It also enables the deploying of bug fixing in hardware, which decreases the cost of re-deploying new hardware and increase the time-to-market for products.
3. Reusability: reusing the resources for different design implementations is enabled, where a system can be customized for adaptability.
4. Power reduction: considered the most important detail, where power dissipated in the system although most of the parts are not working. In the Integrated Circuits (IC) design, the leakage power is the power consumed by the device, while it is even not active. FPGA reconfiguration helps in delaying the implementation of a specific part until the time of operation, which decreases the consumed power over time and though the battery lifetime.
5. Area: instead of implementing a full system in a horizontal way, which consumes area, a system can be optimized by vertical implementation idea which uses programming in space and time, where a stack of blocks are stored and loaded at the time of operation. This will save the area used by the same blocks in the horizontal design.

Quite the reverse, there are some disadvantages for the DPR and they are being improved by research, such as:

1. Latency: latency increased by the time overhead added by the reconfiguration time [63]. It could be improved by using high-speed PR controller to accelerate the reconfiguration time.
2. Memory: as blocks will be stored, more memory is needed for storing the different implementations until the time of operation. As the storage sizes are increasing, this item is improved. For example, 5 files of few kilobytes contain the new reconfiguration can be stored on gigabytes of the attached storage device. Reconfiguration files can be stored on servers and accessed through the network, as the network accessing process is improving by time.

2.6.3. **Terms of DPR**

Reconfigurable Region (RR) is “the region of the FPGA logic core that will be reconfigured, each RP can be reconfigured with one or more Reconfigurable Module (RM), among which swapping occurs”. Reconfigurable Module (RM) is “the module that contains the application to be run. It is designed using HDL or using netlist”.

2.7. Summary

In this chapter, various aspects of FPGA and FPGA DPR were covered. The chapter presented an introduction of FPGA basics o cover FPGA programming technologies, routing architecture, and software flow. Then, the FPGA reconfiguration technology was presented, such as reconfigurable logic and routing techniques, benefits of using DPR. In the next chapter, the verification of DPR using ABV is covered.

Chapter 3 : Dynamic Partial Reconfiguration Verification Using Assertion Based Verification

DPR on FPGAs permits a portion of the logic to be reconfigured at runtime while the rest of the logic keeps operating. This feature allows the designers to build complex systems such as SDR in a reasonable area. However, utilizing DPR needs extra care to be taken for new issues, such as guaranteeing proper connections for the ports of the Reconfigurable Modules (RMs) which share the same Reconfigurable Region (RR) on the FPGA, waiting for running computations on a module before reconfiguring it, isolation of the reconfigurable modules during the process of reconfiguration, and initialization of the reconfigurable module after the process of reconfiguration is done. This chapter proposes a technique to verify these newly introduced issues using Assertion Based Verification (ABV). The proposal is to first automatically model these issues using System Verilog Assertions (SVAs), then instrument the design with the generated assertions. Following that, the instrumented design is verified using simulation or formal methods to check the existence of these issues. The proposed technique proves effectiveness in finding issues on real designs that utilize DPR technique.

3.1. Introduction

DPR on FPGAs allows reconfiguration of some of the logic at runtime while the rest of the logic keeps operating. It allows the implementation of complex circuits as SDR and Internet of Things (loT) applications within a reasonable area on the FPGA. Consequently, the power consumption of the circuit is reduced. Currently, Xilinx and Intel (Altera) are the main FPGA device vendors on the market. They provide a series of FPGA families that support the DPR design flow. In this chapter, the Xilinx DPR design flow is considered [7].

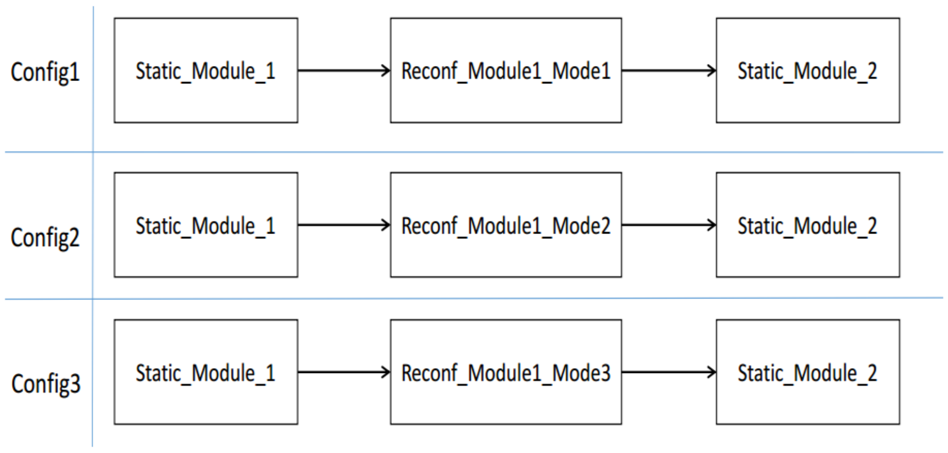


Figure 3.1: An example of DPR design with 3 modes of configuration and 1 reconfigurable module per configuration

In DPR, the design consists of a number of Reconfigurable Modules (RMs). At runtime, each module has modes that are swapped according to the system operating modes. A Reconfigurable Region (RR) is a location on the FPGA in which the reconfigurable module is implemented on. An example of DPR system is shown in Figure 3.1, it has three configuration modes: *Config1*, *Config2, and Config3*. Each configuration has three modules, two of them are static (i.e. they are not changed in any of the operating modes of the design): *Static_Module_1* and *Static_Module_2*, and one of them is reconfigurable module: *Reconf_Module1*, the reconfigurable module has three modes: *Mode1*, *Mode2*, and *Mode3*.

The DRS extend the design flexibility through the mapping of multiple reconfigurable modules to the same physical reconfigurable region, which reduces the design cost and the resource usage. In the example of Figure 3.1, the design will have one RR on the FPGA for the reconfigurable module *Reconf_Module1*. The RR can be configured by an RM mode according to the configuration mode of the design. In the configuration mode Config1, the RR will be loaded by the RM mode *Reconf_Module1_Mode1* and so on. Utilizing DPR technique for FPGA designs adds a new aspect in the design and verification of FPGA designs. For Xilinx FPGAs, the consistency of RMs is one of the basic requirements of a partially reconfigurable design [7]. As one module is swapped for another, the connections between the static design and the RM must be identical. Such requirement adds an extra work on the designer to create a wrapper module to encapsulate all the modes of the RM, and to have a fixed interface between the static design and the RM. This interface must work for all the modes of the RM, the process of connecting the interface to different modes of the RM is an error-prone task and should be verified on the RTL before moving to the implementation of the design on the FPGA.

Designers also add extra logic in their DPR designs for 1) delaying any reconfiguration request till the computations done by the RM is completed, 2) isolating the RM during the process of reconfiguration, and 3) initializing the RM after the process of reconfiguration is done. The added logic for these tasks should be verified on the RTL to make sure they are working as expected, and any bugs are caught as early as possible in the design cycle. The detection of real reconfiguration issues is very challenging, especially in the early design stages. If such errors are not tackled and verified early in the design cycle, they may cause functional errors during on-chip verification which are hard to debug. In this chapter, a new methodology is proposed to verify the added logic for the reconfiguration process of DPR using ABV, the contributions of this work are:

1. Automatically extract the connections of the ports of the modes of the RM, and write SVA properties to verify these connections on the RM wrapper module.
2. Model the functionality of the added logic for the partial reconfiguration process (i.e. delaying the reconfiguration request, isolating the RM during the reconfiguration, and initializing the module after reconfiguration) using SVA properties to verify their functionality.
3. Embed the generated assertions into the RTL, and feed them to simulation or formal verification to verify the functionality of the design.

4. A case study for using the proposed verification methodology on a DPR design and identifying bugs in the design.

3.2. Background

3.2.1. Functional Verification

To understand why verification is important and what methods are used for testing circuits, it is important to understand the hardware development cycle. The first step in the hardware development cycle is the specification stage, where architects specify the behavior of a circuit. This may include creating system-level models to simulate this behavior. The next step is to specify the RTL implementation using an HDL, such as Verilog, which describes the flow of data in a circuit, and how that data is manipulated to achieve the desired behavior.

The RTL implementation is then synthesized into a gate-level implementation, which specifies how the circuit must be constructed out of individual logic gates. This gate-level implementation is then mapped out to determine where the transistors and wiring will be physically located on a chip. This physical layout is then manufactured at a fabrication plant where the circuits are printed onto silicon. This silicon is placed into a package which can potentially interface with other systems. For using FPGA as the target device for designs, there is no physical layout needed for the design and similarly for the fabrication, instead the design is synthesized into a gate-level representation in terms of the FPGA basic cells (LUTs and FFs) of the target device. After that, a bit-stream is generated to be programmed on the FPGA to implement the circuit, the generation of the bit-streams is done by the vendor software such as Xilinx ISE [13].

Since there are so much work and cost that goes into each step of the development cycle of hardware, hardware designers exert an extremely large effort into making sure that each step is done correctly. Making a mistake in one of the steps means that all of the following steps will be wrong, costing even more time and money. Classification of functional verification is shown in Figure 3.2. This chapter focuses on the testing of the RTL design. There are many strategies used in the testing of the RTL design. Traditionally, designers use black-boxing techniques to testing the requirements against the design implementation. This involves the creation of a test-bench with instantiating the Design Under Verification (DUV) in the test-bench. Test patterns are saved into a file with the expected output results, and the test-bench reads the test vectors from that file to drive the DUV, the outputs are then captured and compared to the results of the reference model. For the generation of the test patterns for the DUV, designers use different techniques and strategies. One strategy involves driving the DUV with specific patterns to hit some known scenarios and create some expected behavior, that strategy is called directed testing. Sanity mechanisms (such as results comparison) with directed testing should be used to ensure the matching of the actual behavior for the design's internal states and the design's outputs. Another strategy is to drive the inputs with random stimulus to produce completely random behavior. This random simulation has to be paired with many checkers that ensure that circuit behavior is legal for the whole system. Also, this random simulation has to be guided by design constraints to avoid exploring invalid states for the design under test. Recently, test-benches have become complex verification environments that are often built with a Hardware Verification Language (HVL), which dramatically evolved to standardize and support: automatic vector generation, output response validation, code coverage analysis, constraint solver, and functional coverage.

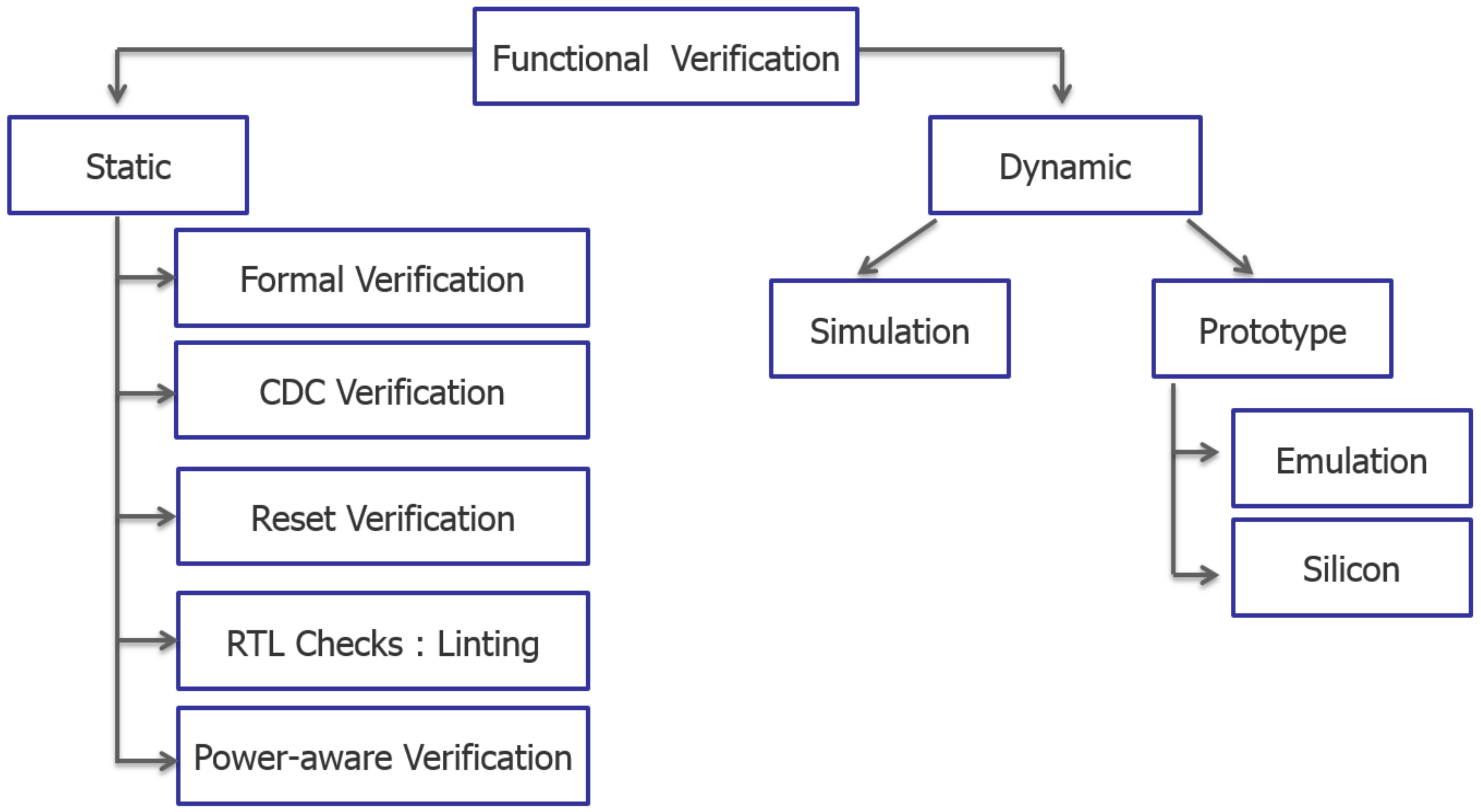


Figure 3.2: Categorization of different methods for functional verification

Figure 3.3 and Figure 3.4 shows the evolvement of the functional verification trends as announced by the Wilson Research Group [34,50]. The trends show the number of designs being verified by advanced techniques such as "Assertions", it shows that that number has been increased over the last ten years. Such increase is mainly due to the increased complexity of the hardware designs, and consequently, the amount of money that will be lost in case of any bug escapes into the fabricated chip.

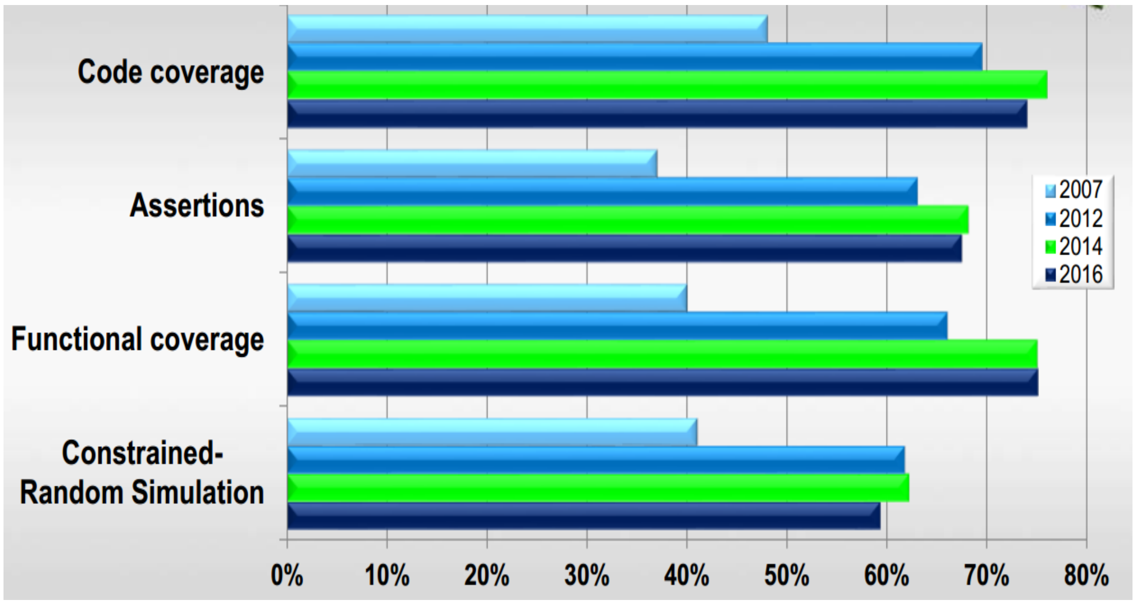


Figure 3.3: Trends for techniques of functional verification for ASIC/IC Design Projects

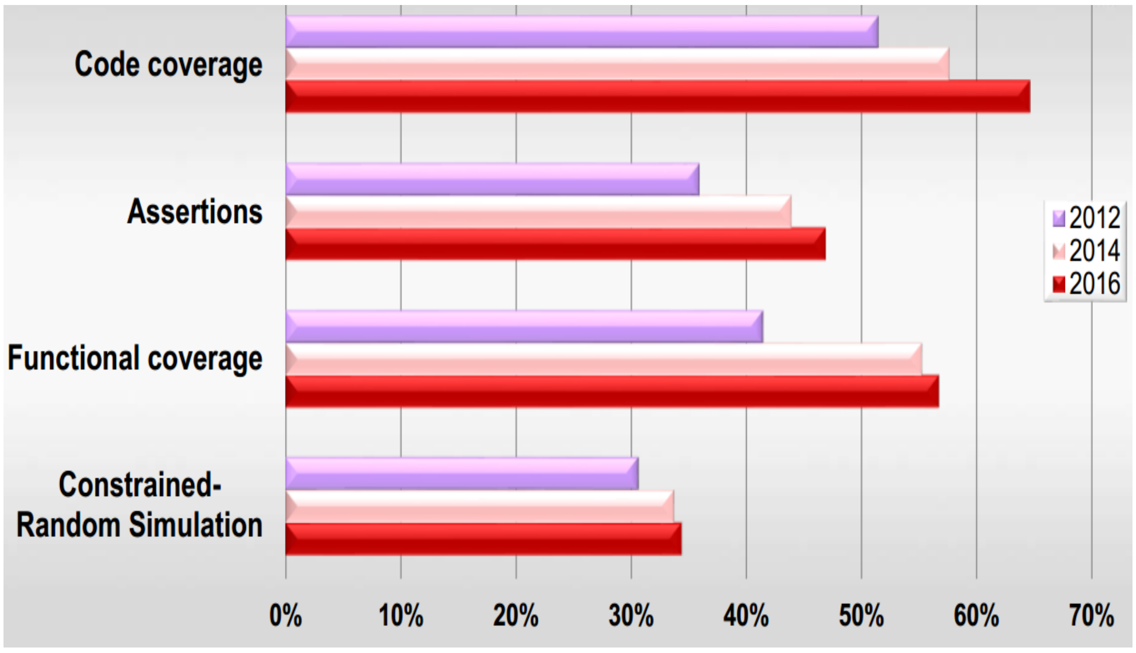


Figure 3.4: Trends for techniques of functional verification for FPGA Design Projects

In order to help designers decide when enough verification is done, they need coverage metrics to measure the progress of the verification and assess its effectiveness in simulation-based verification. With the incorporation of technologies and tools that help in bug finding, engineers can evaluate coverage results and decide on what to do next, and when a design can move to tape-out.

3.2.2. Assertion Based Verification

ABV provides techniques for designers to define assertions in one or more commonly used languages (PSL, Verilog, VHDL, SVA or OVL). These assertions (checkers, monitors) are then folded into the verification test-bench and exercised during simulation, or they can be provided as proof targets to a formal property checking engine.

When assertions are interpreted by verification tools a pass or fail result is the minimal feedback that a tool must provide. In simulation-based verification with assertions, the test-bench should contain test vectors that cover as much as possible of the design's states, i.e. the scenarios considered should be meaningful and relevant to gain confidence about the level of verification being done. If an assertion did not fail because of the absence of proper stimulus, this is not an indication that the design is error free. It indicates that the behaviors, which are specified by the set of assertions, are respected under the given test-bench. When using assertions with formal methods, they provide a proven or fired assertion. Proven means that the assertion passes under any valid test patterns, and fired means that there is a pattern that can cause the assertion to fail. For fired assertions, formal methods generate a waveform (counter examples) which causes the assertion to fail. However, input stimulus doesn't need to be provided for formal or static methods, formal methods mathematically prove the result of the assertions, that's a big advantage of the formal methods when they are compared to simulation.

In general, an assertion is a statement about a specific intended behavior of the design that must hold true under normal operating conditions. Figure 3.5 shows an example of a simple handshake behavior which is intentionally described as after the assertion of the request signal, the acknowledge signal has to be asserted 1 to 3 cycles late.

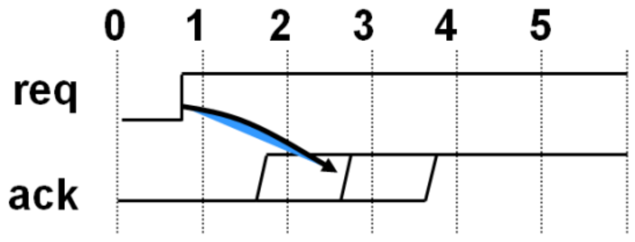
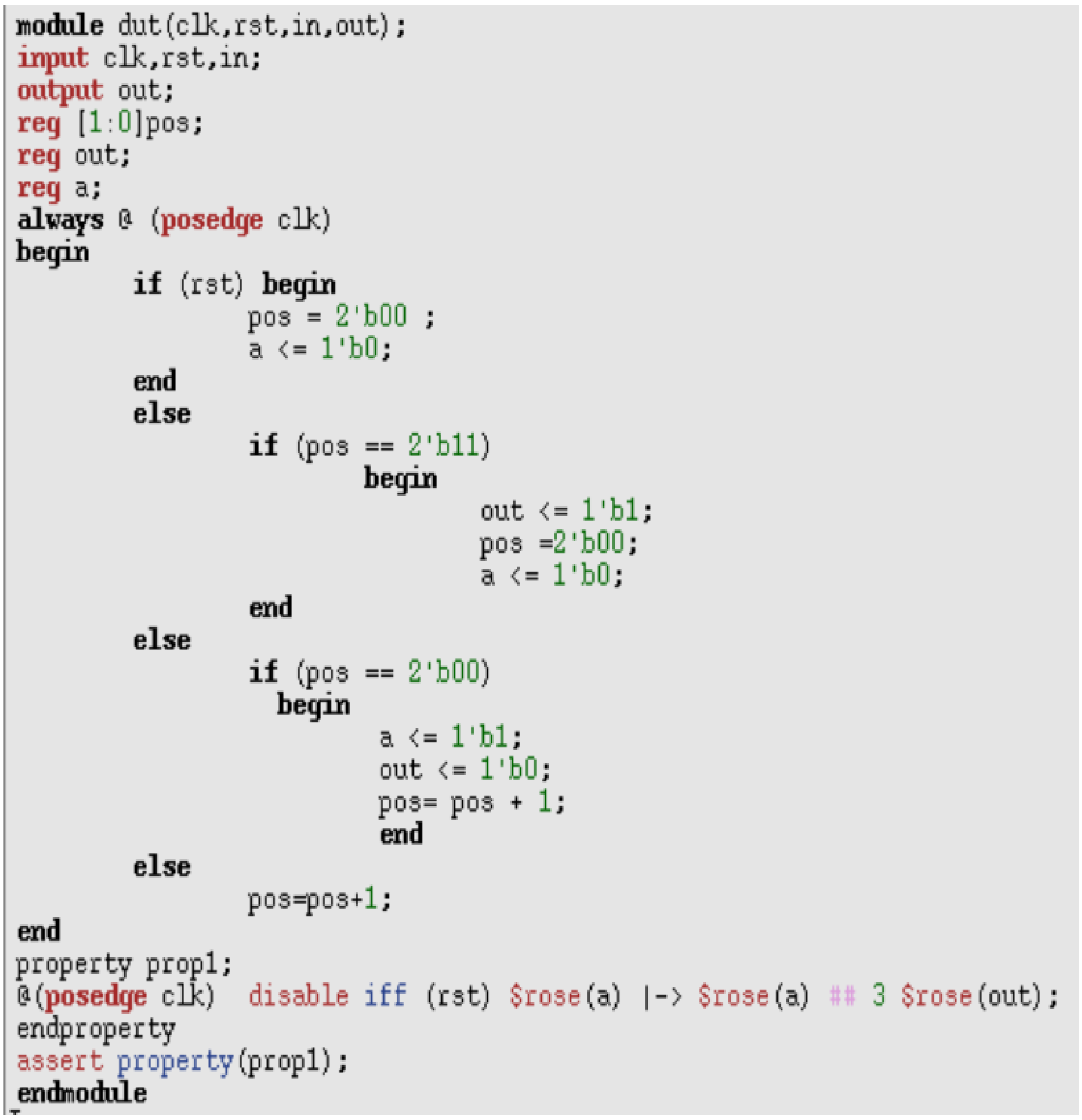


Figure 3.5: Waveform for a request-acknowledge handshake behavior

The above behavior can be described using System Verilog Assertion as

```
property single_req ;
  @( posedge clk ) disable iff ( rst )( $rose(req) ) |=> ( ( !ack && req )[*0:2] ##1 ack ) ;
Endproperty
```

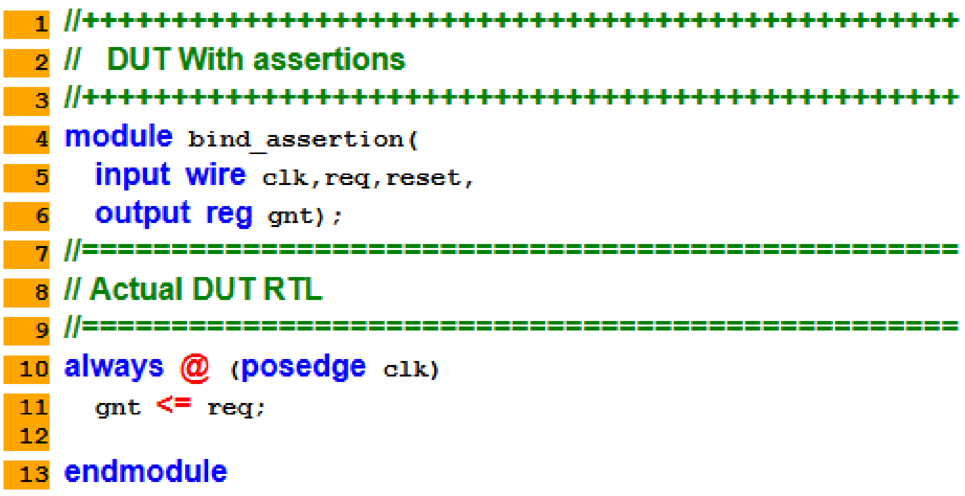
Assertions can be encapsulated within the RTL design, as illustrated in Figure 3.6, or it can be added in a separate module, and bound to the RTL design, as illustrated in Figure 3.7, 3.8, and 3.9. The RTL can be simulated with the associated stimulus .The simulator analyses the execution run and reports the status of the assertions. On the other side, the RTL with the associated properties can be passed to formal verifiers (Model Checkers), which will report proofs or counter examples for the design properties. Formal proof indicates that the property has been mathematically proven to be always true for this design, and in the event of a failure, counter examples can also be generated to show up what is the sequence of stimulus which if applied to the RTL it will violate the design properties.



```
module dut(clk,rst,in,out);
input clk,rst,in;
output out;
reg [1:0]pos;
reg out;
reg a;
always @ (posedge clk)
begin
        if (rst) begin
                pos = 2'b00 ;
                a <= 1'b0;
        end
        else
                if (pos == 2'b11)
                        begin
                                out <= 1'b1;
                                pos =2'b00;
                                a <= 1'b0;
                end
        else
                if (pos == 2'b00)
                  begin
                        a <= 1'b1;
                        out <= 1'b0;
                        pos= pos + 1;
                        end
        else
                pos=pos+1;
end
property prop1;
@(posedge clk)  disable iff (rst) $rose(a) |-> $rose(a) ## 3 $rose(out);
endproperty
assert property(prop1);
endmodule
```

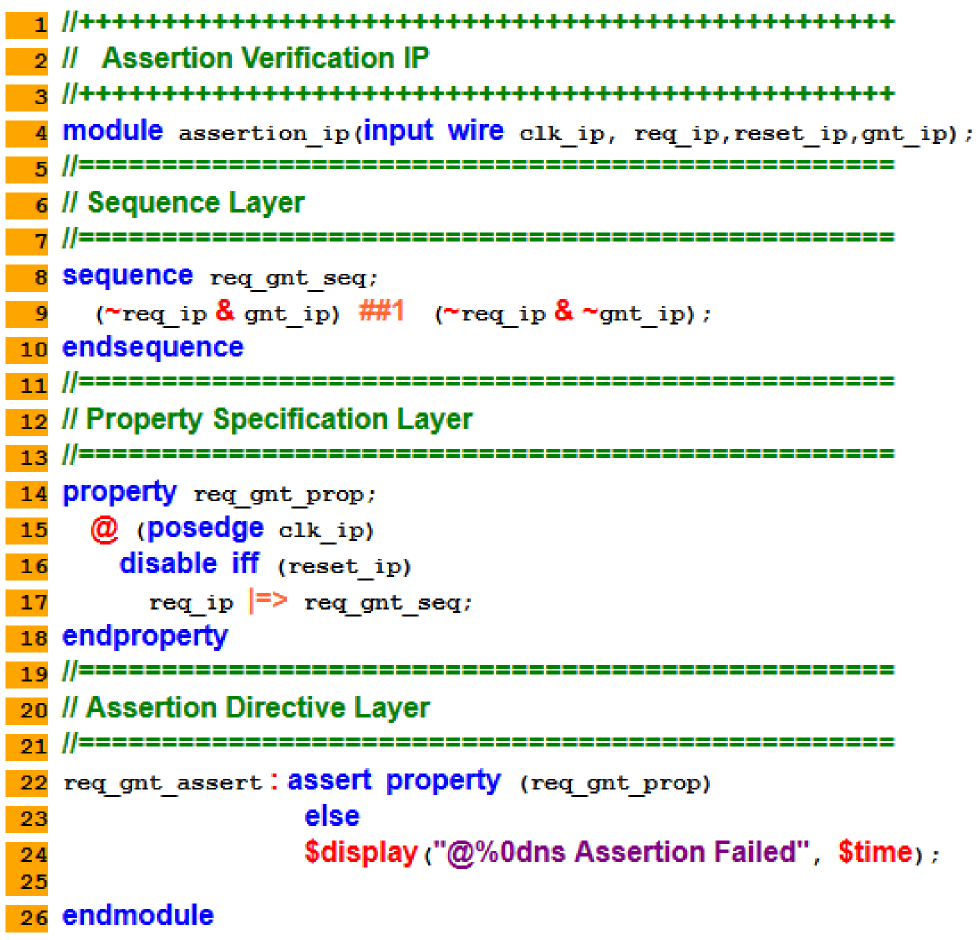
Figure 3.6: Example for assertions embedded into the RTL

Figure 3.7 shows an example for a Verilog module that will be verified using System Verilog Assertions (SVA), unlike the example in Figure 3.6, there are no assertions written into this Verilog module. The assertions used to verify the Verilog module are written into a separate module (called verification module), this verification module will be bound to the Verilog design module to apply the assertions to the design under test, the verification module is shown in Figure 3.8, the binding of the Verilog design module to the verification module is shown and illustrated in Figure 3.9.



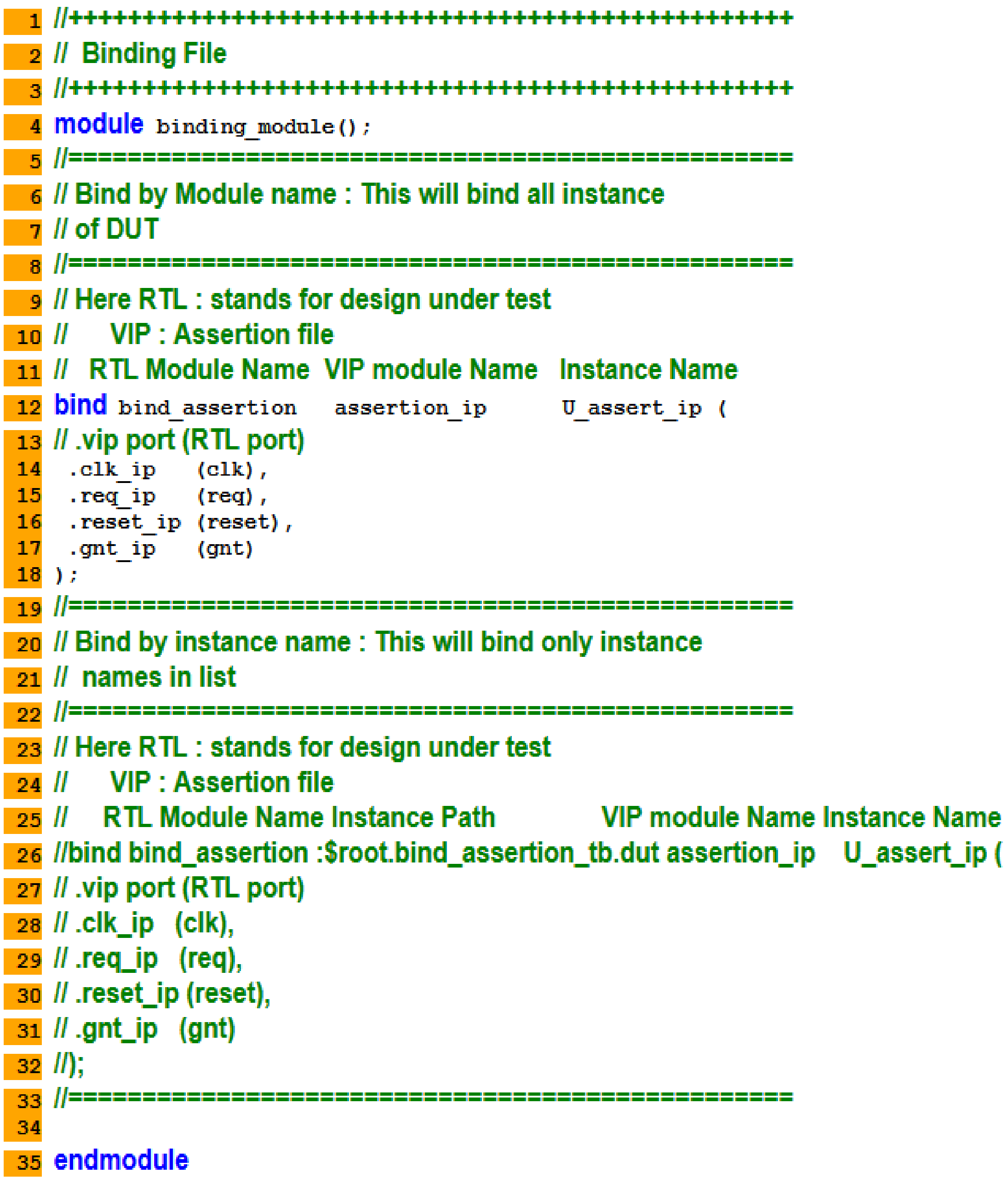
```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module bind_assertion(
  input wire clk,req,reset,
  output reg gnt);
//=================================================
// Actual DUT RTL
//=================================================
always @ (posedge clk)
  gnt <= req;

endmodule
```

Figure 3.7: Verilog design example to be bound to an assertions module



```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   Assertion Verification IP
//+++++++++++++++++++++++++++++++++++++++++++++++++
module assertion_ip(input wire clk_ip, req_ip,reset_ip,gnt_ip);
//=================================================
// Sequence Layer
//=================================================
sequence req_gnt_seq;
  (~req_ip & gnt_ip) ##1  (~req_ip & ~gnt_ip);
endsequence
//=================================================
// Property Specification Layer
//=================================================
property req_gnt_prop;
  @ (posedge clk_ip)
    disable iff (reset_ip)
      req_ip |=> req_gnt_seq;
endproperty
//=================================================
// Assertion Directive Layer
//=================================================
req_gnt_assert : assert property (req_gnt_prop)
                 else
                 $display("@%0dns Assertion Failed", $time);

endmodule
```

Figure 3.8: Assertions module to be bound to the DUT



```
//++++++++++++++++++++++++++++++++++++++++++++++++++++
// Binding File
//++++++++++++++++++++++++++++++++++++++++++++++++++++
module binding_module();
//====================================================
// Bind by Module name : This will bind all instance
// of DUT
//====================================================
// Here RTL : stands for design under test
//      VIP : Assertion file
//  RTL Module Name  VIP module Name   Instance Name
bind bind_assertion  assertion_ip      U_assert_ip (
// .vip port (RTL port)
 .clk_ip   (clk),
 .req_ip   (req),
 .reset_ip (reset),
 .gnt_ip   (gnt)
);
//====================================================
// Bind by instance name : This will bind only instance
//  names in list
//====================================================
// Here RTL : stands for design under test
//      VIP : Assertion file
//     RTL Module Name Instance Path              VIP module Name Instance Name
//bind bind_assertion :$root.bind_assertion_tb.dut assertion_ip    U_assert_ip (
// .vip port (RTL port)
// .clk_ip   (clk),
// .req_ip   (req),
// .reset_ip (reset),
// .gnt_ip   (gnt)
//);
//====================================================

endmodule
```

Figure 3.9: Binding of the Verilog design module to the verification module

For being familiar with the anatomy of hardware design properties, a property can be formally defined as: “A collection of logical and temporal relationships between and among subordinate boolean expressions, sequential expressions and other properties that in aggregate represent a set of behavior”. When studying them, it is easier to look at their compositions as four distinct layers:

1. **Boolean layer:** This layer consists of boolean expressions that are formed using variables of the design model. For example *sel1* and *sel2* are mutually exclusive can be modeled as *!(sel1 && sel2).*

2. **Temporal layer (timed sequences):** This layer permits the verification engineer to describe the boolean expressions' relationships to each other over time. It allows the engineer to define the sequence in which the boolean expression must be satisfied.
3. **Modeling layer (properties):** This layer provides a clear and concise way to describe the circuit's behavior, specify when a sequence should or should not happen. Inside this layer, the engineer can specify when a property should be disabled or enabled.
4. **Verification layer (Directives: assert, cover, assume):** This layer describes how a property is used during verification, i.e should it be used as an assertion and hence it will be checked? Or, should the property be used as an assumption or a constraint to the design? Or, should the property be used to define an event that is used to collect functional coverage information?

These layers of RTL properties specification are shown in Figure 3.10.

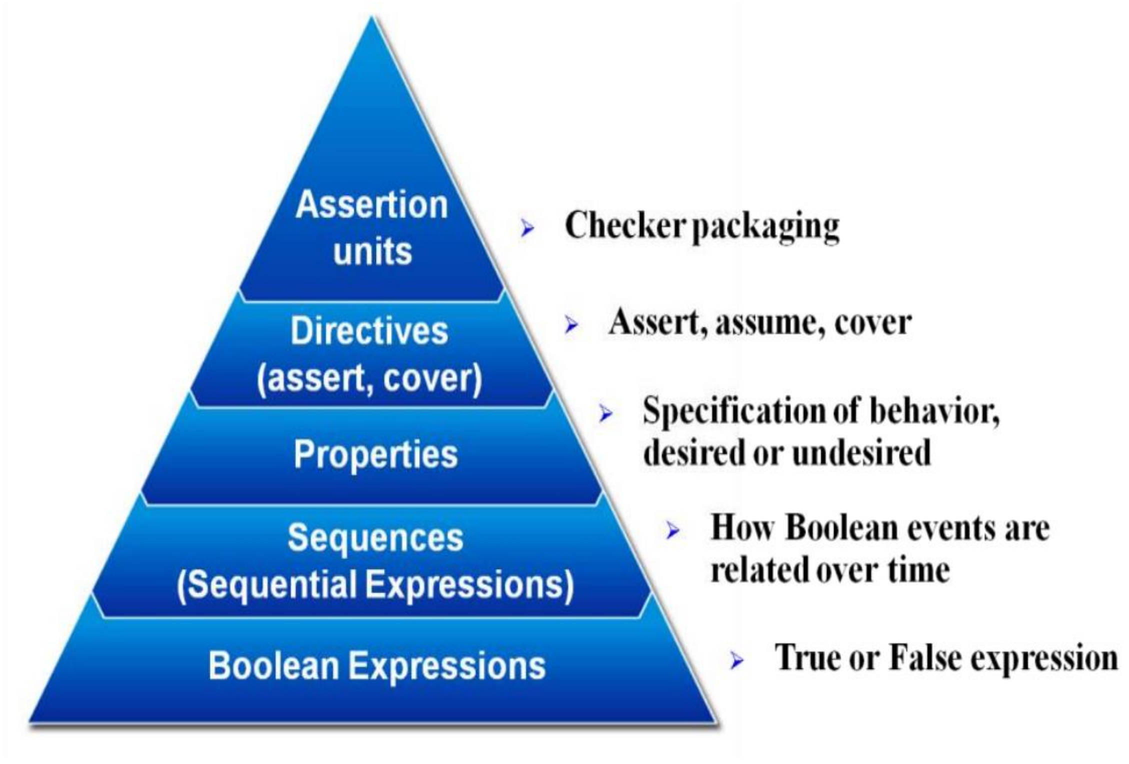

Assertion units
Checker packaging
Directives (assert, cover)
Assert, assume, cover
Properties
Specification of behavior, desired or undesired
Sequences (Sequential Expressions)
How Boolean events are related over time
Boolean Expressions
True or False expression

Figure 3.10: Compositions of hardware design assertion properties

Properties are often classified in the context of their temporal and verification layers. Furthermore, properties can be also categorized by their method of evaluation (that is, concurrent or sequential activation)

1. **Safety versus Liveness:** Safety property says that some bad sequence cannot occur. This is a property that must evaluate to true all the time. On the other side, a property that indicates some good behavior will happen in the future is called a liveness property. It defines a possibility that is unbounded in time. Examples:

   property safety_property_example ;

```
  @( posedge clk ) counter_value <= maximum_allowed_value ;
endproperty

property liveness_property_example ;
  s_eventually counter_value == 1 ;
endproperty
```

2. **Constraint versus Assertion:** A Constraint is a property that lists the acceptable values (or sequences of values) which are permitted on an input. The design cannot be ensured to function correctly if its input value (or sequence of values) violates a specified constraint. While an assertion is a property that specifies that the expected design output behavior must stay valid or true. To guarantee a correct design functionality, all the assertions should evaluate to true for any permissible sequence of input values applied to a design.
3. **Declarative versus procedural:** A declarative property describes the expected behavior of the design independent of its RTL procedural details. Hence, it is not necessary to understand the procedural code to understand the required expected behavior. On the other hand, the procedural property describes the expected behavior of the design in the current context at a particular line within the procedural code. Hence it is necessary to understand the details of the procedural code to fully understand the expected behavior. Expressing interface properties declaratively is generally more intuitive than expressing these properties procedurally, since the interface requirement is typically independent of the details of the block implementation. While capturing internal implementation of an RTL design intent procedurally, will generally reduce the amount of extra code required to express these properties.
4. **Concurrent versus sequential:** A design model typically consists of a static, hierarchal structure, in which primitives interact through the network of interconnections. These primitives may be built in simple functions or large more complex procedural or algorithmic descriptions. Within a procedural description, statements execute in sequence. However within the design as a whole, the primitives and their communication interact concurrently. Just as the design model, properties may also be represented as declarative or procedural statements. Hence, a declarative assertion is a statement that is always active and is evaluated concurrently with other layers or primitives in the design. A procedural assertion, on the other hand, is a statement within the context of a process that executes sequentially in its turn as the procedural code executes.

Hardware Verification Languages (HVLs) are used to write assertions. Property Specification Language (PSL) [51] and System Verilog Assertions (SVA) [21] are the most commonly used HVLs. SVA is part of the System Verilog Language [21]. Also, HW verification engineers can select from a readymade, pre-verified assertion libraries, such as the Open Verification Library (OVL) [52]. Table 3.1 describes the advantages and the disadvantages of each of them:

Table 3.1: Advantages and Disadvantages of HVLs

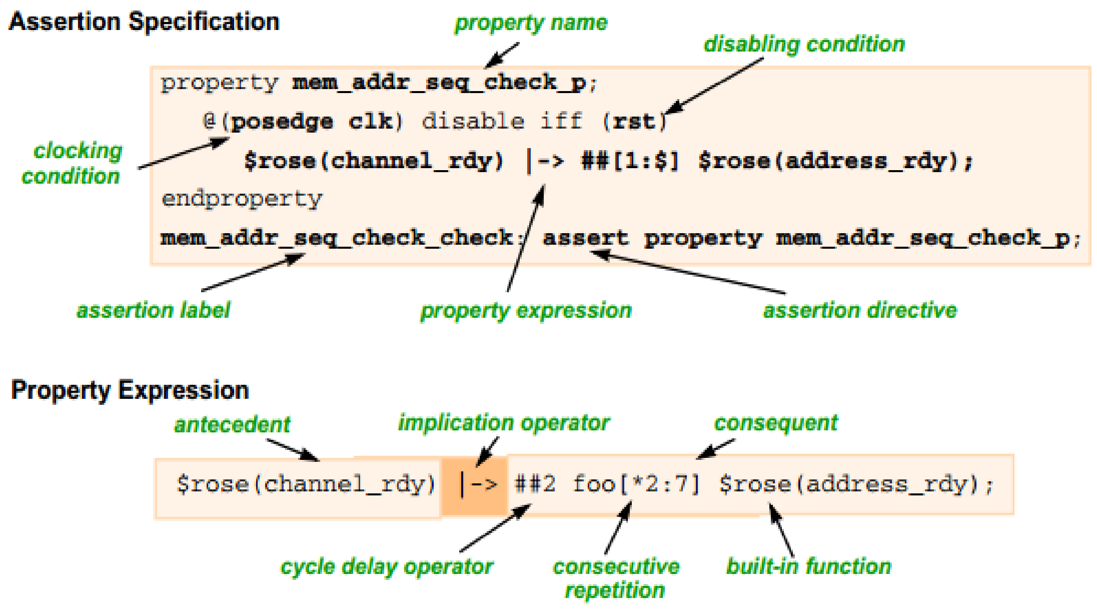
	Assertion Languages (PSL, SVA)	***Assertion Libraries (Checker-Ware, OVL)***

Advantages	• Customization • Abstraction and powerful pattern matching	• Pre-verified checker IP • Drop-in solutions for most common checking tasks • Designers like it (Low effort)
Disadvantages	• Implementation effort • Power = complexity • Learning curve	• Exact checking requirements may not match available components

To summarize, there is a vast array of scenarios where assertions and assertion checkers play an important role in verification, hardware emulation, post-fabrication debugging, permanent online monitoring, simulation, and formal verification. Synthesizing assertion checkers is beneficial and in most of these cases essential to allow the assertion paradigm to be used in these areas.

ABV is one aspect of any complete SoC or Silicon fabrication flow. The design intent and specifications are captured by the assertions in an executable form. During simulation, these assertions are acting as monitors to detect errors close to their source, and to report both errors and coverage information. Assertions also enable formal analysis, which can provide exhaustive verification of blocks and interfaces. With incorporating the usage of assertions in the verification process, verification can start earlier, design and verification teams can detect and remove bugs faster, and designers can incorporate their intent into the design code to minimize integration issues later on.

Assertion languages provide the grammar needed to explicitly codify properties. Two languages are prevalent in the industry, are accepted standards, and are supported by most of the RTL simulation or formal verification tools: SVA 21] and PSL [51]. Both languages' sets of operators and constructs are almost equivalent, they differ by some nomenclatures, syntax and minor features. Figure 3.11 shows the syntax of writing a property.



```
property mem_addr_seq_check_p;
  @(posedge clk) disable iff (rst)
    $rose(channel_rdy) |-> ##[1:$] $rose(address_rdy);
endproperty
mem_addr_seq_check_check: assert property mem_addr_seq_check_p;
```

```
$rose(channel_rdy) |-> ##2 foo[*2:7] $rose(address_rdy);
```

Figure 3.11: Property syntax

In which:

- **Property name** is an identifier for the property used in the assertion directive. Also, it can be used within the specifications of other properties to simplify the specifications of complex properties.
- **Clocking condition** specifies when the signals in the property are sampled.
- **Disabling condition** specifies when the property is disabled (used as a reset condition).
- **Assertion label** is an identifier for the assertion used in reports and to help with the debugging.
- **Assertion directive** is a statement that instantiates the property in verification logic as an assertion (*assert* keyword), assumption (*assume* keyword) or coverage monitor (*cover* keyword).
- **Property expression** is a specification for the property. Specification can be an invariant (for example, *!($isunknown(ctrl))*) or an implication. An implication is "an expression with a left-hand-side (LHS), an implication operator, and a right-hand-side (RHS)". The LHS is called the antecedent, which is a condition that, when sampled true, initiates a thread for the property. The thread starts as soon as the LHS starts evaluating. The RHS is called the consequent, which is a condition that is tested for each thread. If the consequent is true, the property holds for the thread. If the consequent condition is shown to be false for a thread, the property fails for the thread. Asserted properties are supposed to hold for all possible threads. Assumed properties are assumed to hold for all possible threads. Property expressions can include boolean expressions and the following constructs:
    - Built-in functions which are constructs that automate the specification of common expressions (for example, *$rose, $onehot, $past* and *$fell*).
    - Cycle delay operator (*##*) which separates sub-properties in different cycles (relative to the defined clock). For example, *a ##5 b* means *a is true, then 5 cycles later, b is true*.
    - Consecutive repetition operator (*[*n:m]*) — indicates repetition of signals, or cycles (when applied to the cycle operator).

System Verilog Assertions (SVA) [21] form a subset of the System Verilog extension to Verilog [53] that pertains to assertions. SVA assertion code must be embedded in System Verilog modules. The language provides structures for defining sequences of events and combining sequences into design properties. The SVA *assert* statement generates the assertion that verifies its associated property. Figure 3.12 shows an example for an SVA assert property.

```
sequence s_rst_sigs;
   ##1 (uart_out && !done);
endsequence
sequence s_rst_done;
   (sys_rst_l)[->1] ##1 (done throughout ((xmit)[->1]));
endsequence
sequence s_rst_pair;
   s_rst_sigs and s_rst_done;
endsequence

property p_post_rst;
   @(posedge sys_clk) ($fell(sys_rst_l)) |-> s_rst_pair;
endproperty

assert_post_rst: assert property (p_post_rst)
   else $display ("%m : device did not reset properly");
```

Figure 3.12: SVA assert property example

Property Specification Language (PSL) is an assertion language. PSL assertion code can be embedded in Verilog and VHDL modules, and can be placed in *vunits* bound to design units. The PSL *assert* statement generates the assertion logic that verifies its associated property. Figure 3.13 shows an example for a PSL assert property.

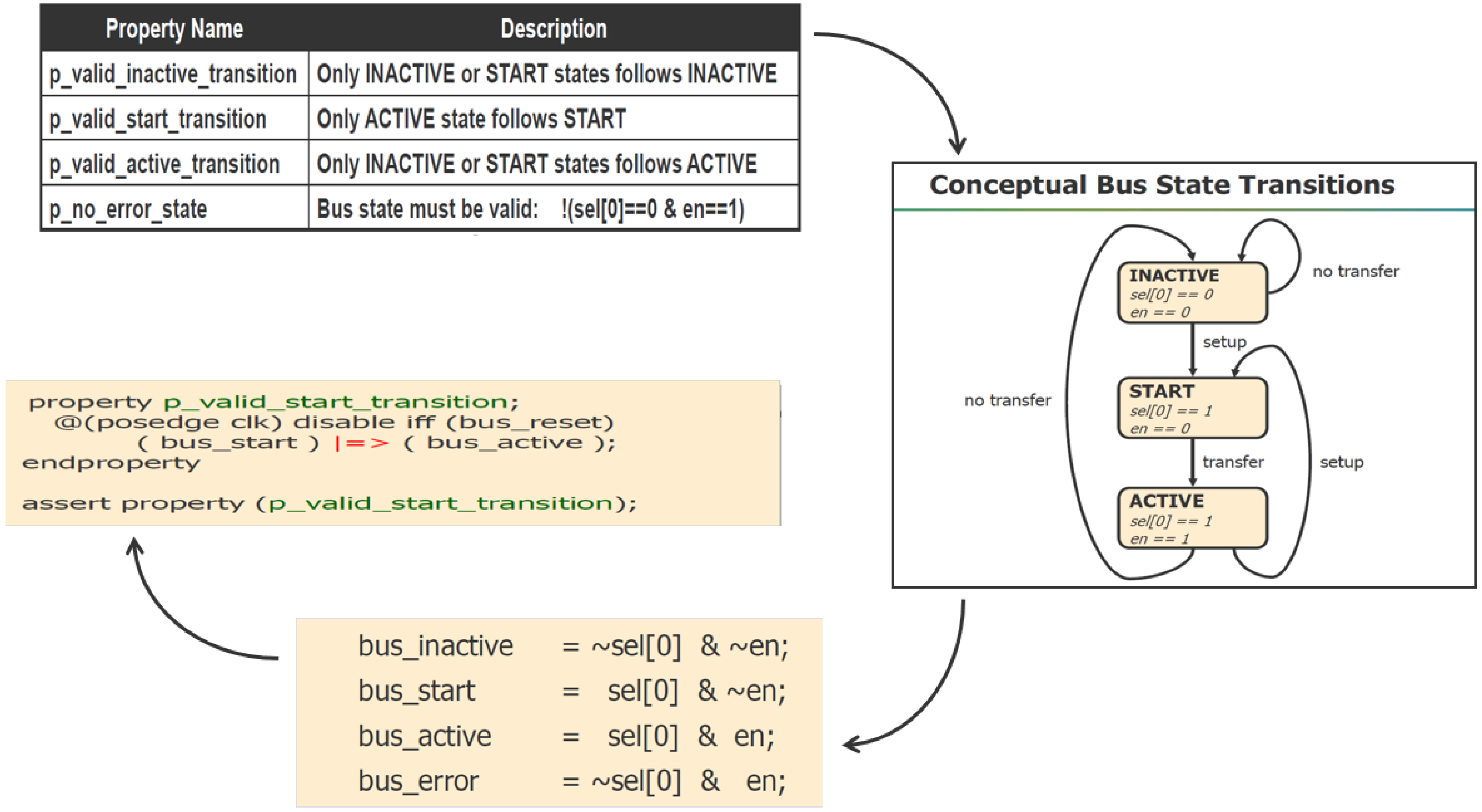
```
vunit psl_rst(top) {

   default clock is rising_edge(sys_clk);

   sequence s_rst_sigs is {uart_out:fell(done)};
   sequence s_rst_done is {(sys_rst_l)[->1];(done:(xmit)[->1])};
   sequence s_rst_pair is (s_rst_sigs & s_rst_done);

   property p_post_rst is (always fell(sys_rst_l) -> s_rst_pair);

   assert p_post_rst;
}
```

Figure 3.13: PSL assert property example

In Figure 3.14, an example is shown to how to define assertions from a given specification to verify the implemented design, the example considered in Figure 3.14 is for a bus and its states for transferring data. The bus state has 3 valid states: *START*, *INACTIVE*, and *ACTIVE*. The valid bus state transitions are as follows:

1. *INACTIVE* to *START*
2. *START* to *ACTIVE*
3. *ACTIVE to INACTIVE*
4. *ACTIVE to START*

Any other transitions are not allowed. Figure 3.14 shows the specification, as well as the Finite State Machine (FSM) of the bus transitions, and how the properties are defined in terms of the bus control signals (*en* and *sel[0]*).



Property Name	Description
p_valid_inactive_transition	Only INACTIVE or START states follows INACTIVE
p_valid_start_transition	Only ACTIVE state follows START
p_valid_active_transition	Only INACTIVE or START states follows ACTIVE
p_no_error_state	Bus state must be valid: !(sel[0]==0 & en==1)

```
 property p_valid_start_transition;
   @(posedge clk) disable iff (bus_reset)
          ( bus_start ) |=> ( bus_active );
endproperty

assert property (p_valid_start_transition);
```

```
bus_inactive    = ~sel[0]  & ~en;
bus_start       =   sel[0]  & ~en;
bus_active      =   sel[0]  &  en;
bus_error       = ~sel[0]  &   en;
```

Figure 3.14: How assertions are defined?

3.3. Related Work

Several works have proposed frameworks to help in functional verification of DRS. The Dynamic Circuit Switch (DCS) method [54] adds artifacts for simulation purposes only to mimic the behavior of reconfiguration activities such as module swapping and undefined state of the RM after reconfiguration. ReChannel [55,56] is an open source SystemC library which models DPR. In order to represent swapping of modules and other reconfiguration operations, ReChannel added new SystemC classes. The extension of ReChannel [57] proposed new classes to monitor and verify the details of reconfiguration at behavioral, Transaction Level Modeling (TLM) and RTL levels. To use ReChannel, designers should be aware of using SystemC for modeling and verification of digital designs, and extra efforts are needed to set up the simulation environment on the behavioral level, TLM level, and RTL.

In [58], a SystemC-based design methodology (OSSS+R) is used to automate the modeling, synthesis, and simulation of DRS designs. It automatically generates synthesizable code for the reconfiguration controller to manage the module swapping of RMs. But, it uses only pre-defined reconfiguration control mechanism, so it cannot handle all styles of DPR designs. ReSim [29] is a System Verilog library built on the Open Verification Methodology (OVM) which uses a simulation-only bitstream to hide the physically dependent features of DPR designs. It models traffic of bitstream and the process of reconfiguration of DPR. ReSim, as well, has a support for the cycle-accurate RTL simulation of the DRS design immediately during, before and after reconfiguration. So, it can detect functional bugs that were missed by DCS, ReChannel, and OSSS+R. Setting up the design to use the ReSim setup needs extra effort by the designer. The ReSim library is extended in [30] to support state saving and restoration of the RMs.

The existing works in literature have some disadvantages and limitations:

1. They model the DPR activities using simulation-only artifacts (i.e. un-synthesizable models), so they cannot be used with formal verification methods. The test-benches used for testing the design should thoroughly cover all the corner cases, such requirement is impractical in some cases.
2. Extra effort is needed to set up the verification environment as SystemC modeling or OVM environment setup.
3. When an error is caught, extra effort is needed to debug the error and pinpoint the root cause of the issue, it can be related to non-DPR logic.

The proposed methodology in this chapter has some advantages when compared to the existing works in literature:

1. It models specific DPR activities using ABV [22], the assertions can be used for formal verification or RTL simulation, and it also can be integrated with any previous work that performs RTL simulation.
2. It enhances the observability, reduces the debug time, and improves error detection. When an assertion fails in RTL simulation or formal verification, it pinpoints to the root cause of the issue with no extra effort.
3. The assertions can be synthesized [31] on the FPGA to perform runtime verification for DPR, this is not covered in this thesis.

3.4. Assertion Based Verification for DPR

3.4.1. Port Connections of the Reconfigurable Modules

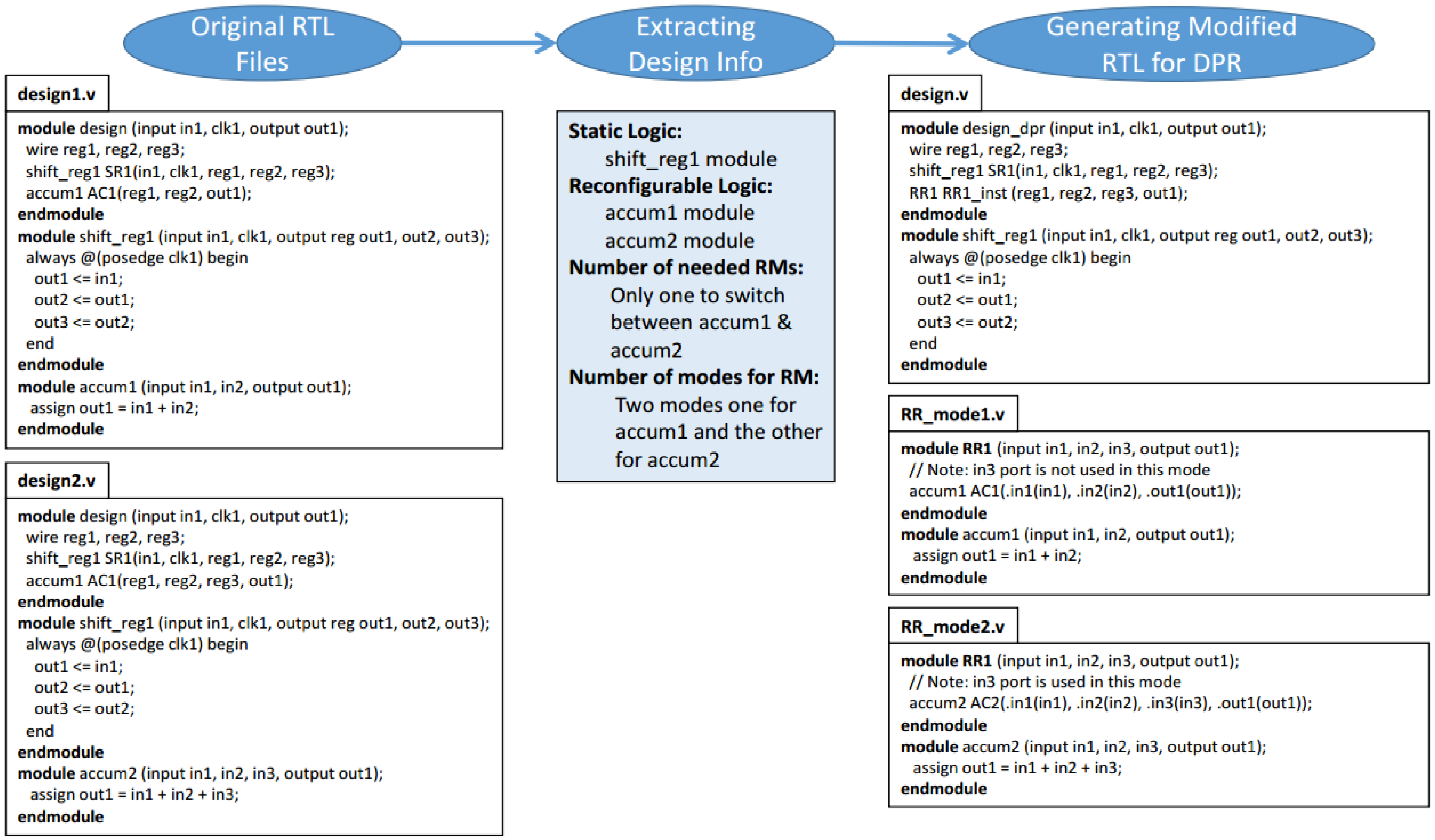
The first step for creating a dynamically reconfigurable design is to identify the static logic (i.e. logic that is always active in all the operating modes of the design), and the reconfigurable logic (i.e. the logic that can change from one operating mode to another) in the design. For Xilinx DPR flow [7], the interface of a reconfigurable module should be consistent across all its modes, such requirement adds an extra step in the design flow to create an RTL wrapper for each RM to encapsulate all its modes. Issues appear in this step when there is a mismatch in the number of ports between different modes of the RM, the RTL wrapper of that RM will have number of ports equal to the maximum number of ports in all the modes of the RM, in that case the designer should take care when connecting the ports for each mode of the RM to not affect the functionality of the circuit.

A simple example for the design modifications needed for DPR is shown in Figure 3.15, if the port *in3* is used in in the first mode of the RR *'RR_mode1.v'*, then the design functionality will be altered. Such modifications in the RTL should be verified before moving to implement the design on the FPGA. The modification for the interfaces of the RMs is an error-prone task, especially for large designs which have a large number of ports for the modes of the RMs and a mismatch in the sizes of these modules such as the case of the SDR. In this thesis, the connectivity verification approach [32,33] is utilized in this section to verify the changes in the interfaces of the reconfigurable modules.

The verification flow is shown in Figure 3.16. After the RTL files are compiled and the design is synthesized, the netlist of the design is traversed using netlist access Application Programming Interfaces (APIs) to extract the connections of the reconfigurable modules from the original design, the output of this step is a Comma Separated Values (CSV) file that lists the hierarchical paths of the RM ports and their connections. An SVA generator takes the CSV file and writes an assertion for every source and destination pair. The following SVA property is generated for every source and destination pair to verify their connection:

```
property connect_pair ( clock , source , destination ) ;
@( posedge clock ) disable iff( ~( `RM_MODE_ENABLE ) )
( source == destination ) ;
endproperty
```

Where *RM_MODE_ENABLE* is a macro which can be set by the designer such that when its logic value is 1, it indicates a specific RM mode is active. This macro is different from one RM mode to another because only one RM mode can be active at a time. For each RM mode, there will be a separate CSV file to test its connections, and consequently, a unique set of assertions. The assertions generated for each mode can be verified using RTL simulation or formal verification.



design1.v

```
module design (input in1, clk1, output out1);
 wire reg1, reg2, reg3;
 shift_reg1 SR1(in1, clk1, reg1, reg2, reg3);
 accum1 AC1(reg1, reg2, out1);
endmodule
module shift_reg1 (input in1, clk1, output reg out1, out2, out3);
 always @(posedge clk1) begin
  out1 <= in1;
  out2 <= out1;
  out3 <= out2;
 end
endmodule
module accum1 (input in1, in2, output out1);
  assign out1 = in1 + in2;
endmodule
```

design2.v

```
module design (input in1, clk1, output out1);
 wire reg1, reg2, reg3;
 shift_reg1 SR1(in1, clk1, reg1, reg2, reg3);
 accum1 AC1(reg1, reg2, reg3, out1);
endmodule
module shift_reg1 (input in1, clk1, output reg out1, out2, out3);
 always @(posedge clk1) begin
  out1 <= in1;
  out2 <= out1;
  out3 <= out2;
 end
endmodule
module accum2 (input in1, in2, in3, output out1);
  assign out1 = in1 + in2 + in3;
endmodule
```

Static Logic:
shift_reg1 module
Reconfigurable Logic:
accum1 module
accum2 module
Number of needed RMs:
Only one to switch between accum1 & accum2
Number of modes for RM:
Two modes one for accum1 and the other for accum2

design.v

```
module design_dpr (input in1, clk1, output out1);
 wire reg1, reg2, reg3;
 shift_reg1 SR1(in1, clk1, reg1, reg2, reg3);
 RR1 RR1_inst (reg1, reg2, reg3, out1);
endmodule
module shift_reg1 (input in1, clk1, output reg out1, out2, out3);
 always @(posedge clk1) begin
  out1 <= in1;
  out2 <= out1;
  out3 <= out2;
 end
endmodule
```

RR_mode1.v

```
module RR1 (input in1, in2, in3, output out1);
 // Note: in3 port is not used in this mode
 accum1 AC1(.in1(in1), .in2(in2), .out1(out1));
endmodule
module accum1 (input in1, in2, output out1);
  assign out1 = in1 + in2;
endmodule
```

RR_mode2.v

```
module RR1 (input in1, in2, in3, output out1);
 // Note: in3 port is used in this mode
 accum2 AC2(.in1(in1), .in2(in2), .in3(in3), .out1(out1));
endmodule
module accum2 (input in1, in2, in3, output out1);
  assign out1 = in1 + in2 + in3;
endmodule
```

Figure 3.15: Design modifications in RTL files for DPR

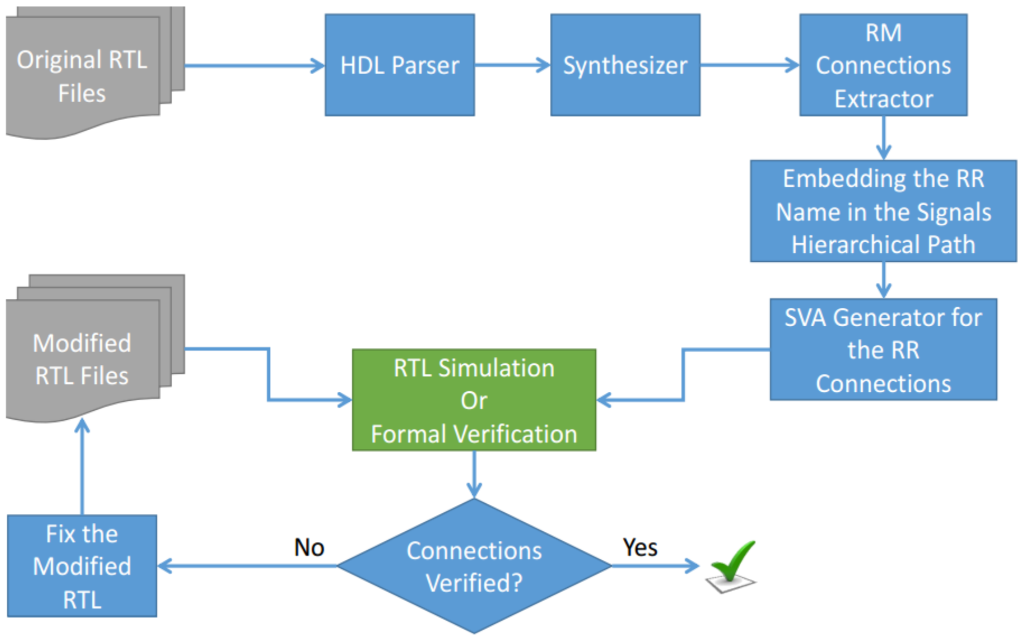

Original RTL Files
HDL Parser
Synthesizer
RM Connections Extractor
Embedding the RR Name in the Signals Hierarchical Path
SVA Generator for the RR Connections
Modified RTL Files
RTL Simulation Or Formal Verification
Connections Verified?
No
Yes
Fix the Modified RTL

Figure 3.16: Design modifications in RTL files for DPR

3.4.2. Isolation Logic

During the reconfiguration process of the RM, the values of the newly downloaded bitstream may drive incorrect values to the static logic side, so designers add isolation logic for all the ports of the RM to prevent the transmission of the data from the RM to the static logic during the reconfiguration process. The typical structure of designs that utilize DPR is shown in Figure 3.17, the Internal Configuration Access Port (ICAP) is used to interface to the FPGA configuration memory (e.g. read or write operations).

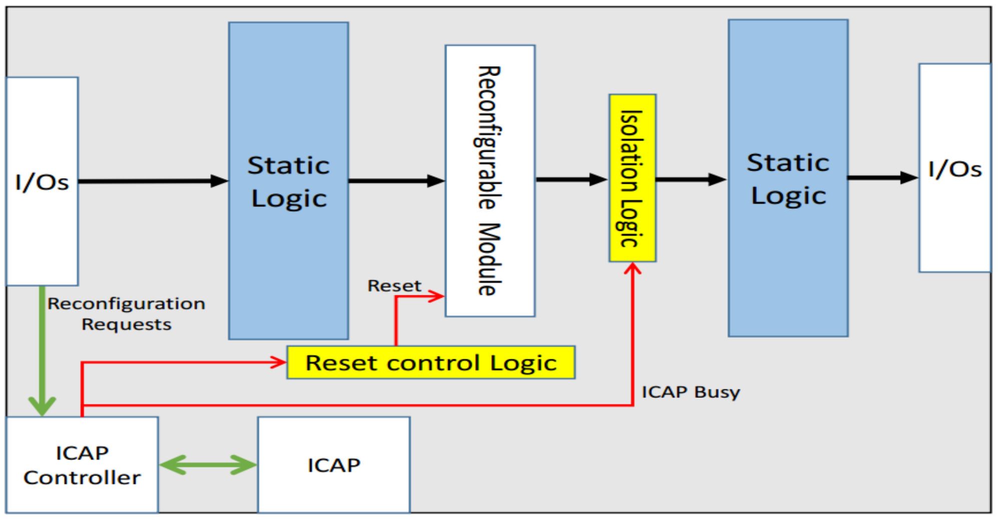


Figure 3.17: Typical structure of a design that utilizes DPR

A controller is needed for the ICAP to handle the reconfiguration requests, handle the control of the ICAP, and monitor its status. The output port of the ICAP can be used to monitor its status. The yellow blocks in Figure 3.4 are added by the designer to have a correct operation for the design during and after the reconfiguration process. For the isolation logic, it is verified using the following SVA property for every output port of the RM:

```
property verify_isol ( clock , source , destination , ICAP_BUSY ) ;
@( posedge clock )
( ( $changed( source ) && ICAP_BUSY ) |=> $stable( destination ) ) ;
endproperty
```

Where the *source* signal is an output port of the RM, the *destination* signal is the register driven by the output port on the static side, and the ICAP_BUSY is the signal which indicates that there is a reconfiguration process in progress.

3.4.3. **Reset Control Logic for the RM**

After the reconfiguration process is done, the sequential elements of the RM should be reset to guarantee proper operation of the circuit. If the RM is not reset after reconfiguration, the state of sequential elements will be undefined and may be affected by erroneous values from the previous RMs that share the same physical area on the FPGA. The reset control logic is verified using the following SVA property:

```
property verify_reset (clock , RM_reset , ICAP_BUSY ) ;
@( posedge clock )
( $fall( ICAP_BUSY ) |-> $rose( RM_reset ) ) ;
endproperty
```

Where *RM_reset* is the reset signal of the RM, and the *ICAP_BUSY* is the signal which indicates that there is a reconfiguration process in progress. The assertion implies that when the *ICAP_BUSY* is changed from a logic value of 1 to 0 (i.e. the reconfiguration through ICAP is done), then the reset signal of the RM should be asserted to reset all the sequential elements of the RM.

3.4.4. **Synchronizing the Reconfiguration Process**

When a computation is being done in the RM, the designers want to block any reconfiguration request until such computation is done. For applications such as SDR, such mechanism will be required such that when a packet is being processed for 3G standard as an example, it should be processed completely before switching to any other standard such as WiFi or 4G. The synchronization of the reconfiguration requests is verified using the following SVA property:

```
property verify_sync ( clock , RM_busy , ICAP_GO ) ;
@( posedge clock )
( $rose( ICAP_GO ) until $fall( RM_busy ) ) ;
endproperty
```

Where *RM_busy* is the signal which indicates that a computation is being done by the RM, and *ICAP_GO* is the control signal which tells the ICAP to start a new reconfiguration process. For some applications, it is not needed to check such synchronization, as it is acceptable to flush the data of the RM.

3.5. Case Study

The approach presented in this chapter for verification of DPR is applied on an SDR chain presented in [45,46]. The SDR test case has four reconfigurable modules: 1) convolutional encoder, 2) modulator, 3) Discrete Fourier Transform (DFT), and 4) Inverse Fast Fourier Transform (IFFT). Table 3.2 shows the number of modes per each block.

Table 3.2: Number of modes per each RM of the design under test

Block	Number of Modes
Convolutional Encoder	4
Modulator	3
DFT	2
IFFT	2

The block diagram and the schematic of the design are shown in Figure 3.18 and Figure 3.19 respectively.

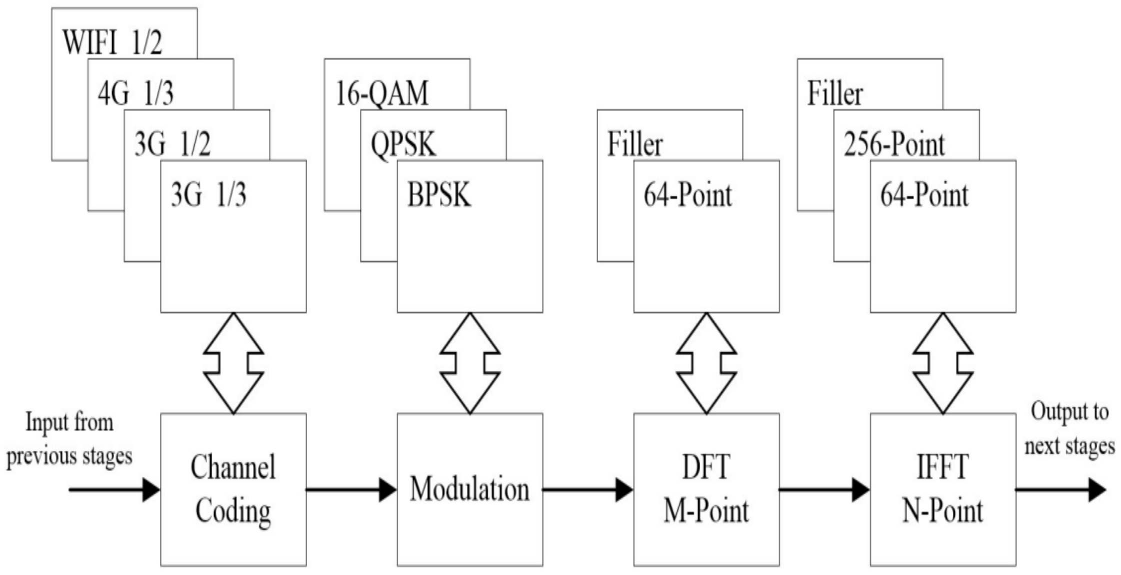


Figure 3.18: Block diagram of the SDR case study

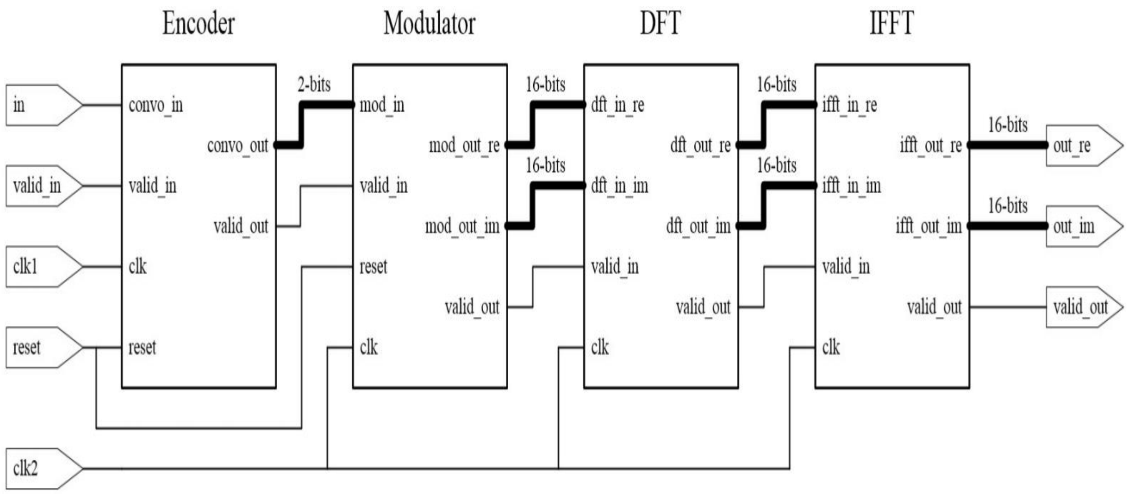
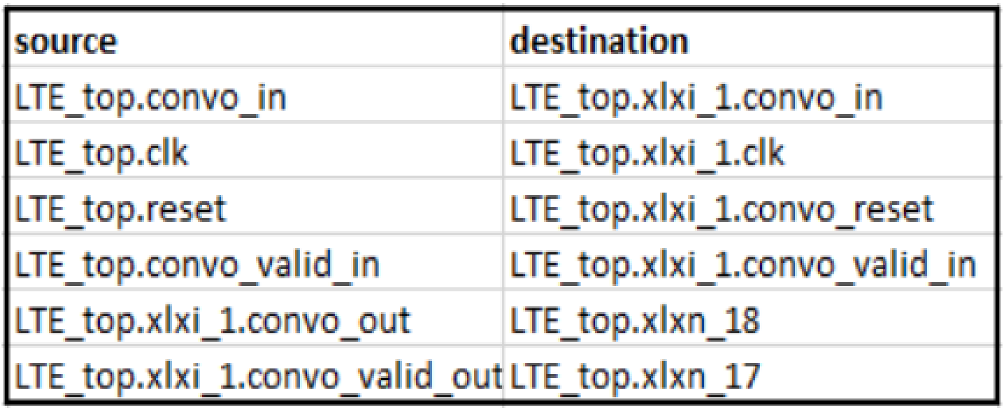


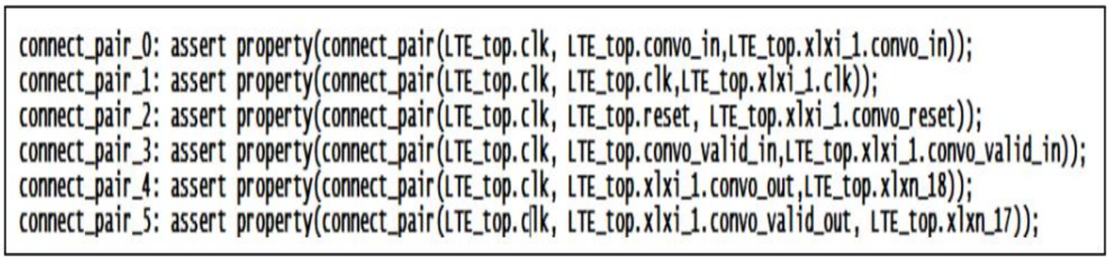
Figure 3.19: Schematic of the SDR case study

The port connections are verified using SVA properties as explained in the previous section. The port connections should be verified for every mode of each RM, the number of assertions generated for verifying port connections is proportional to the number of modes and the number of ports for each mode. The SVA properties are verified and run on the DPR design using Questa Formal tool [47]. Figure 3.20 shows an example for the CSV file extracted for the first mode of the convolutional encoder RM. Figure 3.21 shows an example for the generated assertions to verify the port connections in the CSV file of Figure 3.20, and Figure 3.22 shows the results of Questa Formal tool in which all the assertions are proven.



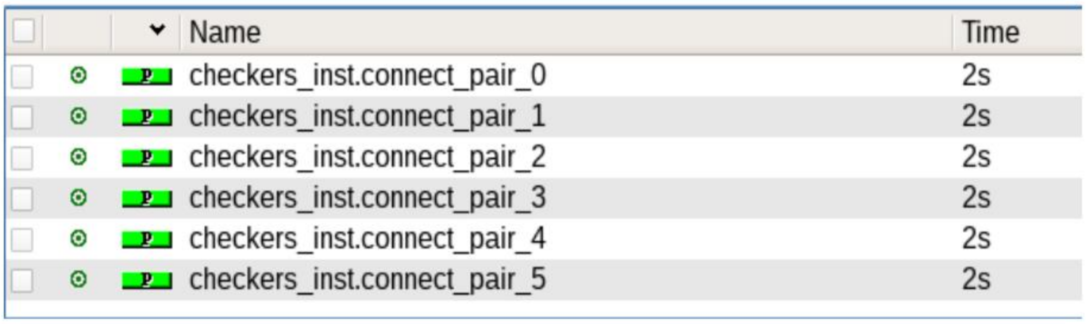
source	destination
LTE_top.convo_in	LTE_top.xlxi_1.convo_in
LTE_top.clk	LTE_top.xlxi_1.clk
LTE_top.reset	LTE_top.xlxi_1.convo_reset
LTE_top.convo_valid_in	LTE_top.xlxi_1.convo_valid_in
LTE_top.xlxi_1.convo_out	LTE_top.xlxn_18
LTE_top.xlxi_1.convo_valid_out	LTE_top.xlxn_17

Figure 3.20: CSV file extracted for connections of the first mode of the convolutional encoder block



```
connect_pair_0: assert property(connect_pair(LTE_top.clk, LTE_top.convo_in,LTE_top.xlxi_1.convo_in));
connect_pair_1: assert property(connect_pair(LTE_top.clk, LTE_top.clk,LTE_top.xlxi_1.clk));
connect_pair_2: assert property(connect_pair(LTE_top.clk, LTE_top.reset, LTE_top.xlxi_1.convo_reset));
connect_pair_3: assert property(connect_pair(LTE_top.clk, LTE_top.convo_valid_in,LTE_top.xlxi_1.convo_valid_in));
connect_pair_4: assert property(connect_pair(LTE_top.clk, LTE_top.xlxi_1.convo_out,LTE_top.xlxn_18));
connect_pair_5: assert property(connect_pair(LTE_top.clk, LTE_top.xlxi_1.convo_valid_out, LTE_top.xlxn_17));
```

Figure 3.21: Assertions generated for connections of the first mode of the convolutional encoder block



☐			Name	Time
☐	⊙	P	checkers_inst.connect_pair_0	2s
☐	⊙	P	checkers_inst.connect_pair_1	2s
☐	⊙	P	checkers_inst.connect_pair_2	2s
☐	⊙	P	checkers_inst.connect_pair_3	2s
☐	⊙	P	checkers_inst.connect_pair_4	2s
☐	⊙	P	checkers_inst.connect_pair_5	2s

Figure 3.22: Results of Questa Formal tool for the assertions generated for connections of the first mode of the convolutional encode block, all assertions are proven

Table 3.3 shows the number of ports for every RM, and Table 3.4 shows the number of assertions generated for verification of port connections, isolation logic, reset control logic, and the synchronization logic. The number of assertions for the isolation logic equals to the number of output ports for all the RMs, the number of assertions for the reset control logic equals to the number of RMs because each RM will have its own reset control logic, and only one assertion is generated to test the synchronization logic of the DPR controller.

Table 3.3: Ports information about the RMs of the design under test

Module	Number of Ports (Total)	Number of Ports (Outputs only)
Convolutional Encoder	6	2
Modulator	7	3
DFT	7	3
IFFT	7	3

Table 3.4: Generated assertion properties for DPR verification

Verification Goal	**Number of Assertions**
Connections of the Ports	(6 x 4) + (7 x 3) + (7 x 2) + (7 x 3) = 80
Isolation Logic of Output Ports	11
Logic for Reset Control	1 x 4 = 4
Logic for Synchronization	1
Total	96

All the assertions of the port connections are proven by the Questa Formal tool. But, when applying the assertions for isolation logic, reset control logic and synchronization logic, Questa Formal tool reports firings for their SVA properties, and 3 bugs have been identified in the design under verification:

1. The output ports of the RMs were not isolated during the reconfiguration process. This should be fixed in the design such that the output ports of the RMs are totally isolated from the static logic during the reconfiguration process to avoid the propagation of any erroneous values from the RMs to the static logic.
2. The reset signals of the RMs were not activated right after the completion of the reconfiguration process. This should be fixed in the design such that the reset signals should be asserted after the reconfiguration to put the RM in a defined initial state before its operation.
3. The DPR controller was not handling the case in which a new reconfiguration request is received when the RM is still processing data.

3.6. Summary

In this chapter, a verification flow for DPR is presented using Assertion Based Verification (ABV). Designers can use this flow to verify their DPR designs and the dedicated logic added for DPR activities such as the isolation logic, reset control logic and the synchronization logic of the DPR controller. SVA properties are used to verify these functionalities. The SVA properties can be used in RTL simulation or formal verification. Using a case study from literature, it has been demonstrated how the proposed verification flow identified three issues in the DPR logic of the design. In the next chapter, the CDC verification for DPR is covered.

Chapter 4 : Clock Domain Crossing Verification for Dynamically Reconfigurable Systems

DPR on FPGAs permits a portion of the logic to be reconfigured at runtime while the other remaining logic keeps operating. This kind of designs are called Dynamically Reconfigurable System (DRS) designs, they can operate in multiple modes. The verification of the DRS designs is a complicated task due to the need to verify all the modes of the designs, and the lack of CAD tools support for DRS designs. In this chapter, an automatic Clock Domain Crossing (CDC) verification flow is proposed for DRS designs. A Perl utility is implemented which automates the generation of the designs files for each operating mode of the design, generates the script to run CDC analysis on the design, runs a CDC analysis tool, and collates the results in a user-friendly representation for debugging.

4.1. Introduction

DPR on FPGAs permits a portion of the logic to be reconfigured at runtime, while the other remaining logic keeps operating. It allows the implementation of complex circuits as SDR and loT applications within a reasonable area on the FPGA. Consequently, the power consumption of the circuit is reduced. Recent FPGA families support the implementation of DRS through the DPR technique.

In DPR, the design is composed of a number of Reconfigurable Modules (RM), each RM has modes that are changed during runtime according to the system operating modes. A Reconfigurable Region (RR) is a location on the FPGA in which the reconfigurable module is implemented on. An example for DPR system is shown in Figure 4.1, it has five configuration modes: *Config1*, *Config2*, *Config3*, *Config4* and *Config5*. Each configuration has four reconfigurable modules: *ModuleA*, *ModuleB*, *ModuleC* and *ModuleD*, each with four modes: *Mode1*, *Mode2*, *Mode3* and *Mode4*. DRS designs extend the design flexibility through the mapping of multiple reconfigurable modules to the same physical reconfigurable region, which reduces the design cost and the resources usage. In the example of Figure 4.1, the design will have 4 RRs on the FPGA, each RR is used for a unique RM. The RR can be configured by an RM mode according to the configuration mode of the DRS design. In the configuration mode *Config1*, the first RR will be loaded by the RM mode (*ModuleA_Mode1*), the second RR will be loaded by the RM mode (*ModuleB_Mode1*) and so on.

Most complex recent designs have more than one clock, and many of these clocks are asynchronous. For these designs, the clock domain of an asynchronous clock is formed by the logic clocked by that clock. Problems arise from signals that connect logic in different clock domains. Proper synchronization must be done for signals that traverse the boundaries of clock domain, and relevant transfer protocols must be followed. During the metastability window of the receiving register (setup and hold time), if any CDC signal is not kept stable, then the register can end up in a metastable state, which means its output can unsystematically settle to an unknown value that is not the same as the value the engineer sees in RTL simulation, an example is shown in Figure 4.2. Errors in functionality can happen due to these metastability issues.

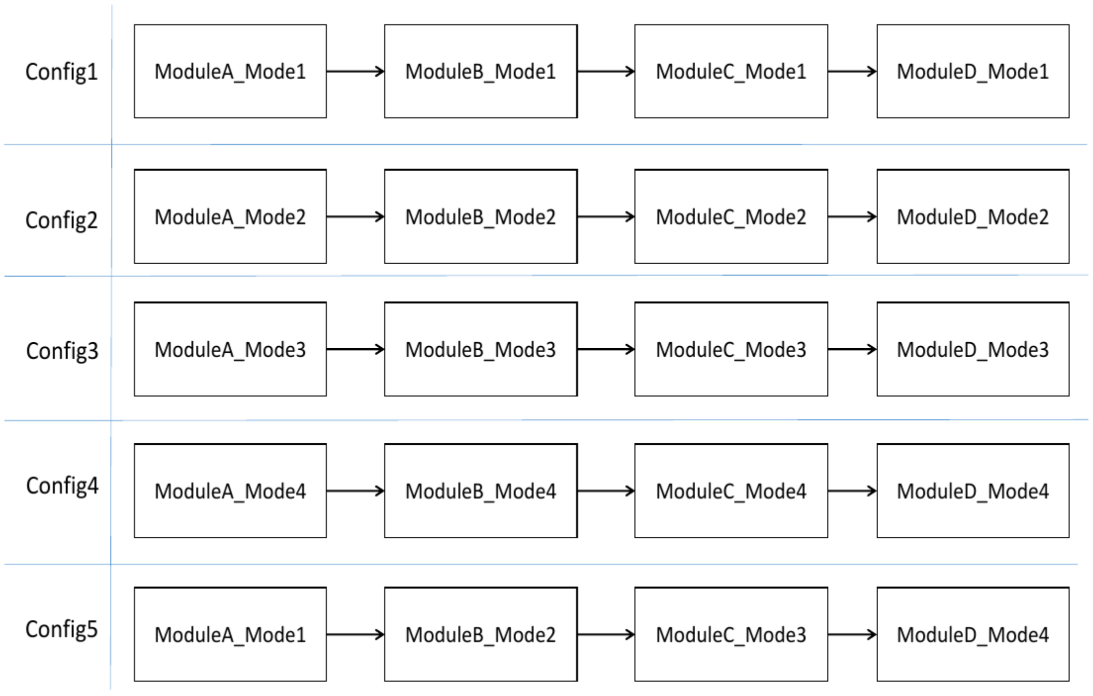


Figure 4.1: An example of DPR design with five modes of configuration and four reconfigurable modules per configuration.

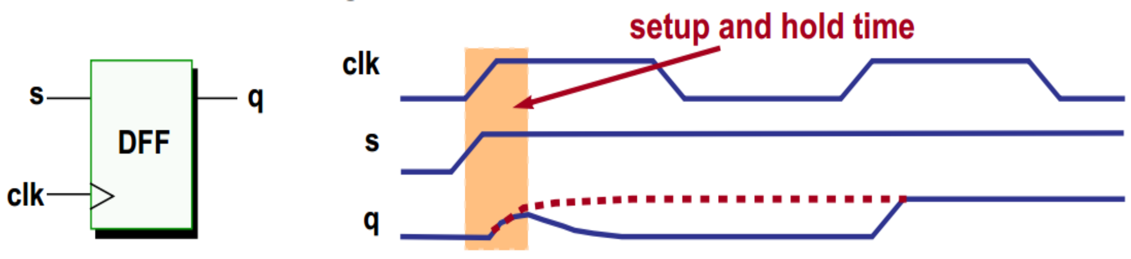


Figure 4.2: Example for a metastability issue caused by CDC signal

CDC verification [23] of DRS designs is a complicated task due to the need of verifying every operating mode of the design to make sure no metastability issues can occur in the design. Currently, there are no Computer Aided Design tools that support the CDC verification of DRS. As an example in Figure 4.1, designers should verify all the configuration modes of the design, to make sure any CDC signals between adjacent modules are properly synchronized. If CDC errors are not verified and tackled early in the design cycle, they may cause functional errors later in the synthesis and place & route phases which may waste the designer's time to repeat the design cycle after fixing the CDC errors.

In this chapter, a new automated flow is proposed for CDC verification of DRS. A Perl utility is implemented to 1) automate the generation of the RTL representation of every operating mode of the reconfigurable system, 2) generate the run scripts to run a commercial CDC tool for every mode, 3) invoke the run for CDC analysis and 4) collates the result for every mode and report it to the user.

4.2. Background

The verification of DRS designs is still an open question. The lack of CAD tools that understand the dynamic nature of these designs forces the designers and verification engineers to innovate and implement their own verification methodologies. Several works have proposed techniques for simulation-based verification of DRS designs, and verification of issues that may arise before, during, and after reconfiguration of some part of the design.

The Dynamic Circuit Switch (DCS) method [54] adds artifacts in the RTL code of the DRS design for simulation purposes only to switch between hardware tasks, it improves the simulation precision of DRS designs in various aspects. But, using this method cannot detect bugs introduced by bitstream transfer and the module swapping in DRS designs.

ReChannel [55,56] is an open source SystemC library which models DPR, it was extended in [57]. In order to represent swapping of modules and other reconfiguration operations, ReChannel added new SystemC classes. The extension of ReChannel [57] proposed new classes to monitor and verify the details of reconfiguration at behavioral, Transaction Level Modeling (TLM) and RTL levels. However, DCS and ReChannel do not accurately verify the design undergoing reconfiguration since the bitstream traffic is not simulated.

OSSS+R [58] is a methodology to automate the modeling, synthesis, and simulation of DRS designs. It generates synthesizable code for the reconfiguration controller to manage the module swapping of RMs. But, it uses only pre-defined reconfiguration control mechanism, so it cannot handle all styles of DPR designs.

ReSim [29] is a reusable library which uses a simulation-only bitstream to hide the physically dependent details of DPR designs. It models traffic of bitstream and the reconfiguration process of DPR. ReSim, as well, has a support for the cycle-accurate RTL simulation of the DRS design immediately before, during and after reconfiguration. So, it can detect functional bugs that were missed by DCS, ReChannel and OSSS+R.

The existing work in literature focuses on simulation-based functional verification of DRS designs, there are more advanced verification topics that are not still addressed for DRS designs such as CDC verification, reset verification, power-aware verification, formal verification and runtime verification. In this chapter, a framework for CDC verification is introduced for DRS designs.

4.3. What is CDC Verification?

Most complex designs have more than one clock. In addition, many of these clocks are asynchronous. For these designs, the clock domain of an asynchronous clock is formed by the logic clocked by that clock. The logic that lies completely inside a clock domain can be validated with the same methodology as that for a single-clock design. However, problems arise from signals that connect logic in different clock domains. Proper synchronization must be done for signals that traverse clock domain “boundaries”, and relevant transfer protocols must be followed. The procedure of validating these necessities is called clock domain crossing (CDC) analysis.

But, even CDC signals that are properly synchronized and obey protocol rules do not guarantee valid functionality. During the metastability window of the receiving register (i.e. setup and hold time), if a CDC signal is not kept stable, the register can end up in a metastable state, which means its output can randomly settle to an unknown value that is not the same as the value the engineer sees in RTL simulation.

In effect, data values that traverse clock domains can be advanced or delayed randomly relative to RTL simulation. Functional errors can occur if the logic of the receiver is not designed specially to be tolerable for these metastability effects. Unfortunately, standard simulation cannot precisely demonstrate effects of metastability in a design. An expansion to standard functional verification is needed to demonstrate the effects of metastability in a design.

4.3.1. Clock Domains

A clock domain is a portion of a design that has a clock asynchronous to (or which has an inconstant phase relationship to) another clock in the design. For example, suppose one clock is derived from another clock through a clock divider. These two clocks have a constant phase relationship; therefore, the two sections of the design that use these clocks are really part of the same clock domain (Figure 4.3). However, suppose two clocks have frequencies of 50 MHz and 33 MHz. These clocks’ phase relationships change over time; therefore, they clock two different clock domains (Figure 4.4).

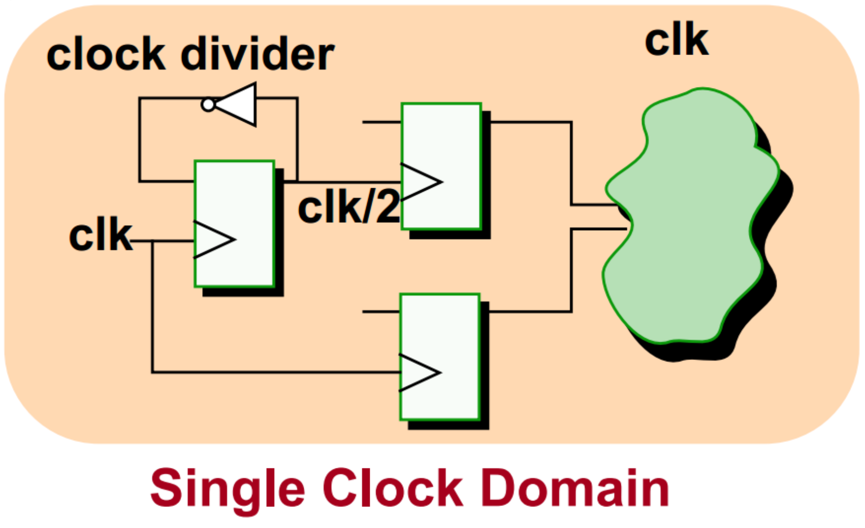


Figure 4.3: Multiple clock signals belong to the same clock domain.

When multiple clocks from different sources (i.e. asynchronous clocks) are inputs to a circuit, then these distinct clock domains are created because of these asynchronous clocks, as shown in Figure 4.5. When the circuit's inputs are asynchronous to the circuit's clock domains, then these asynchronous inputs are in distinct clock domain, as shown in Figure 4.6. Clocks are defined as the clock signals of registers and the enable signals of latches.

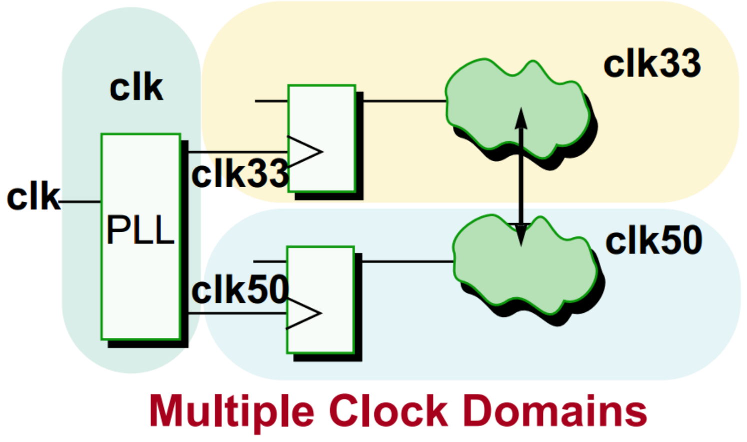


Figure 4.4: Multiple clock signals in two different clock domains

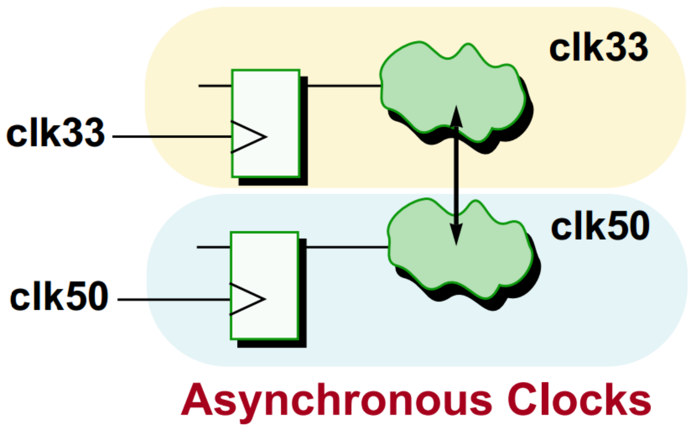


Figure 4.5: Asynchronous inputs clocks form different clock domains

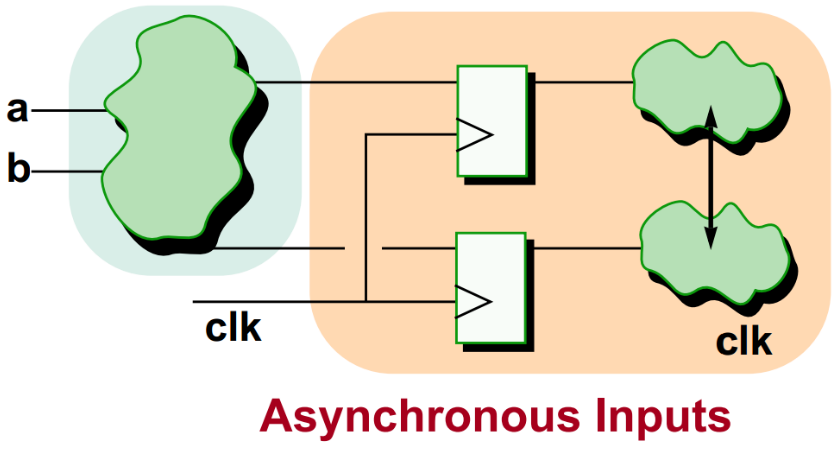


Figure 4.6: Inputs to the circuit are asynchronous to the circuit

4.3.2. **Metastability**

A clock domain crossing (CDC) signal is a signal created in a clock domain and traverses the boundary into another domain (where these two domains are asynchronous to each other), and is then sampled by a register in that asynchronous clock domain.

When the active edge of the receiver (RX) register's clock and the active edge of transmitter (TX) register's clock are too close to each other, metastability occurs if data changes within the setup or hold time. The register's output settles to an unpredictable value. Metastability can occur when having unpredictable skews between synchronous clocks, or if the clocks are asynchronous. Flip-flop and latch storage elements are sensitive to metastability. The design of flip-flops and latches must tolerate the metastability effects.

The properties of metastability are unsystematic and unpredictable in hardware as the output signal can settle randomly to 1 or 0. However, designers got predictable results in RTL simulation. As a result, the hardware behavior and implementation are not accurately modeled in RTL simulation when metastability is existing. Functional verification techniques must consider technology beyond RTL simulation to make sure a circuit design is tolerable and immune to metastability effects. Designers need to understand how hardware registers behave with metastability and how registers behave in RTL simulation under the conditions of metastability, in order to design circuits which are tolerable to the effects of metastability.

The following statement is quoted from [48] regarding metastability:

> "When sampling a changing data signal with a clock ... the order of the events determines the outcome. The smaller the time difference between the events, the longer it takes to determine which came first. When two events occur very close together, the decision process can take longer than the time allotted, and a synchronization failure occurs."

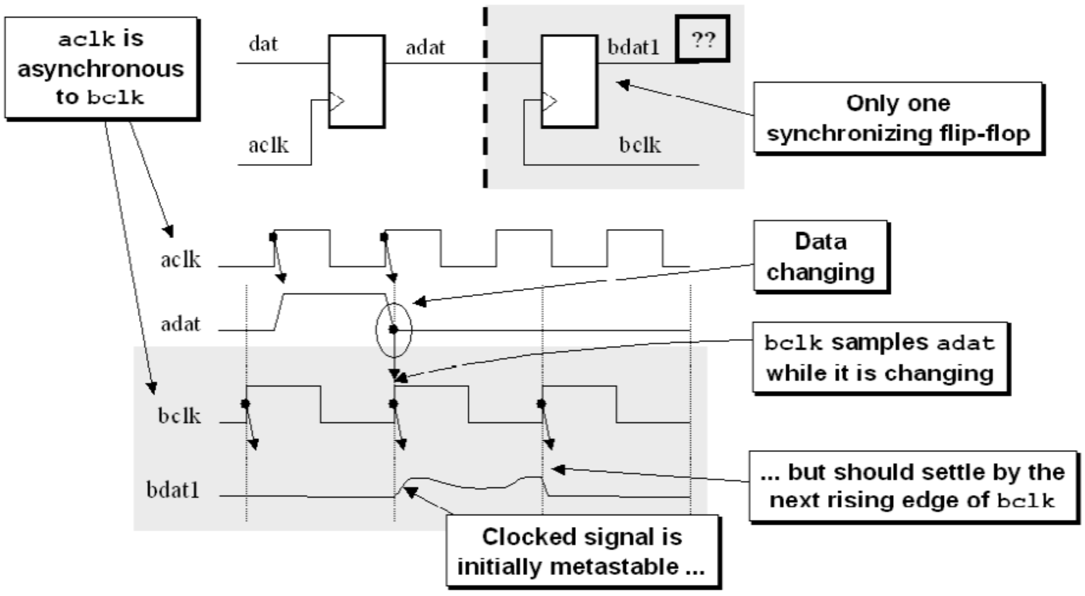

aclk is asynchronous to bclk
dat
adat
aclk
bdat1
??
bclk
Only one synchronizing flip-flop
aclk
adat
bclk
bdat1
Data changing
bclk samples adat while it is changing
... but should settle by the next rising edge of bclk
Clocked signal is initially metastable ...

Figure 4.6: Example for synchronization failure [23]

Figure 4.6 shows an example for failure in synchronization which happens when a signal is generated in one clock domain, and then sampled very close to the active edge of a clock signal from a different clock domain. Synchronization failure is triggered by an output going into a metastable state and not converging to a valid steady state when the sampling of the output must be done.

In hardware, a register value is metastable when its input signal changes the value in the transmitter's domain too close to the time the signal is sampled in the receiver's domain. In Figure 4.7, the flip-flop DFF is sampling a 1-bit CDC signal (*s*). Since signal (*s*) is originated from a different clock domain, then its value of can change at any time relative to the clock of the DFF (*clk*). If the value of the wire (*s*) is not kept stable at 0 or 1 through the metastability window of the DFF (i.e. setup and hold time of the DFF), then the output (*q*) might acquire an intermediate voltage value for an indeterminate amount of time. Following that, (*q*) settles randomly to either 0 or 1. The flip-flop is said to be metastable for that interval.

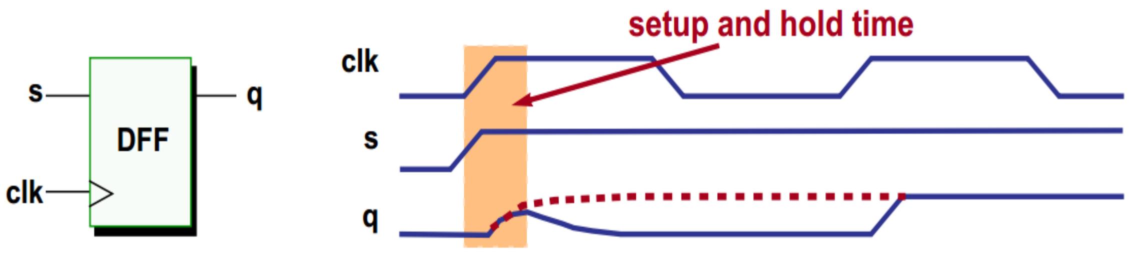


Figure 4.7: Metastable flip-flop

The following mean-time-between-failure (MTBF) equation expects the rate of occurrence of metastability:

$$MTBF = \frac{1}{f_{clk} \times f_{in} \times t_d} \quad (1)$$

Where $f_{clk}$is the clock frequency of the receiving flip-flop, f_{in} is the frequency of the asynchronous input signal, and t_d is the setup and hold window.

Metastability is considered a problem because a metastable signal which feeds additional logic in the receiving clock domain may cause invalid signal values to be propagated through the design, and consequently, the behavior of the circuit cannot be expected in such case. The metastable signal can fluctuate for some amount of time. The logic which samples the metastable signal in the receiving domain may identify the logic value of the fluctuating signal to be different values, and consequently, will cause erroneous signal values to be propagated through the design, Figure 4.8 is showing an example for such cases.

For any design, each flip-flop has a specified metastability window defined (i.e. setup and hold time window), which is the time that the input data is not allowed to be changed within, and it is mandatory to the keep the input signals stable during this window to avoid them being changed very close to the clock edge of the receiving clock edge. This protects the output of the flip-flop from going into a metastable state.

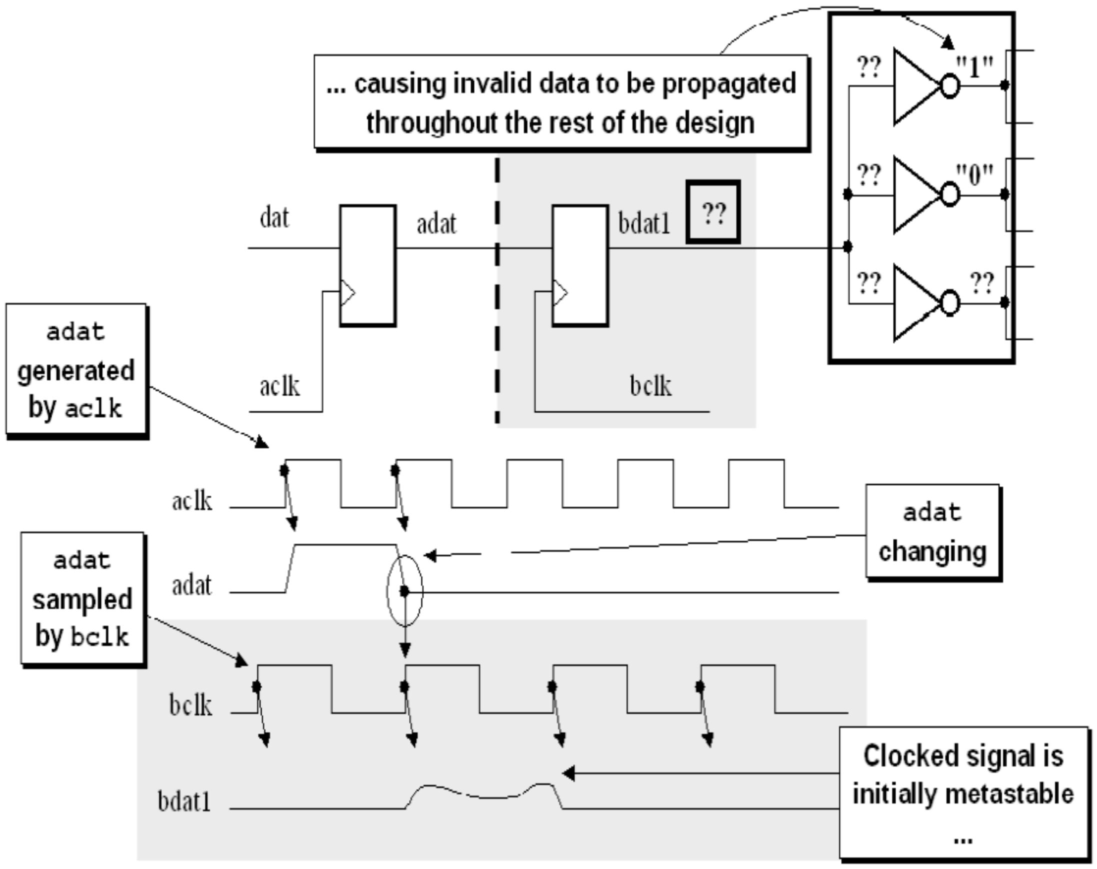

... causing invalid data to be propagated throughout the rest of the design
dat
adat
bdat1
??
aclk
bclk
??
"1"
??
"0"
??
??
adat generated by aclk
aclk
adat
adat changing
adat sampled by bclk
bclk
bdat1
Clocked signal is initially metastable ...

Figure 4.8: A metastable signal is causing erroneous signal values to be propagated through the design [23]

4.3.3. Synchronizers

Designers usually assume the signals of the circuit to be in-band, which means they have a value of either logic 0 or logic 1. Metastable signals can have values that are neither 0 nor 1; therefore, they are considered out-of-band signals. Out-of-band signals have unanticipated effects and propagate unpredictably. To handle CDC signals, designers isolate potentially metastable logic to ensure logic beyond such isolation boundary only needs to handle in-band signals. The logic inside the isolation area is called a synchronizer, an example is shown in Figure 4.9.

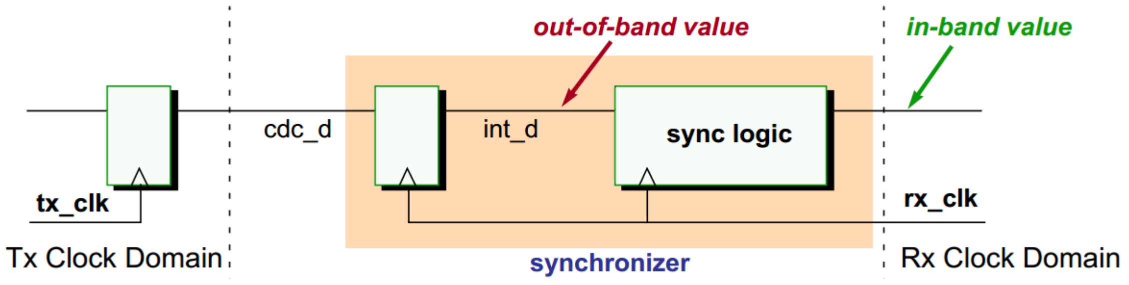


Figure 4.9: Synchronizer example

Metastability appears in the form of mutable delays in signal transitions of the outputs of registers driven by CDC signals. Transitions are accidentally advanced or delayed when compared to normal simulation. Every CDC signal is affected by that behavior. Even if a CDC signal or data bus has a synchronizer, the output of the synchronizer may suffer from mutable delays. Logic outside the isolation area in the receiving domain might not interpret receive data correctly in the presence of variable delays. Functional errors occur in hardware due to this intolerance of metastability special effects, even when RTL simulation reports **"0"** functional errors.

Designers implement different kinds of synchronizers as appropriate for particular situations and design styles. For each type of synchronizer, the implemented logic assumes a group of prerequisites about the operation of the circuit during operation and regarding the logic which is being connected to the synchronizer. During compilation, the rules for the synchronizer's connections can be checked. During simulation, transfer protocols can only be checked as the circuit operates. A synchronizer, alongside its transfer protocol and rules of connections, is called a synchronization scheme as shown in Figure 4.10.

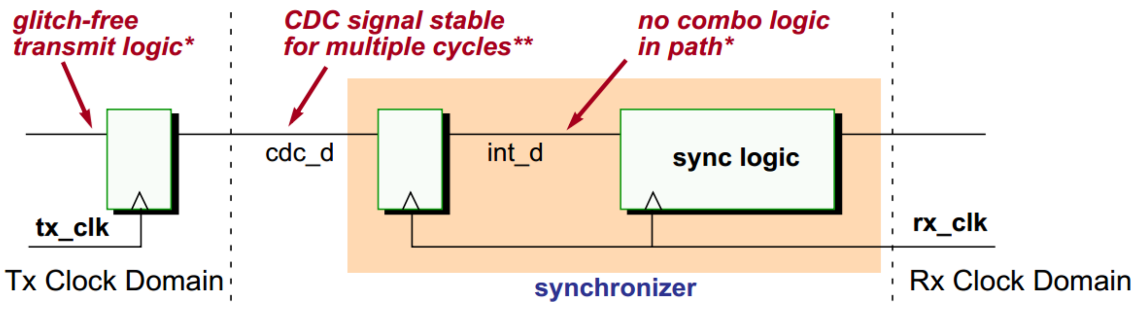


Figure 4.10: Synchronizer scheme

Most CDC implementations use one or more synchronizers from a set of popular, well-characterized synchronization schemes. These structured synchronizers must follow well-defined connection rules and should obey specific transfer protocols. Software or custom logic synchronizers should be used to synchronize any CDC signal that does not have a structured synchronizer. These ad hoc synchronizers block the receiver's registers from reading CDC signal values when they are not stable. Therefore, the receiver register's outputs cannot enter a metastable state. For example, an ad hoc synchronizer can use specific logic to control the load enable signal of the receiver register, or software might control the loading of a circuit's configuration registers.

For control signals (i.e. scalar signals) synchronizers, the two D-flip-flop (2DFF) is commonly used. An example for 2DFF synchronizer is shown in Figure 4.11, it is the most widely used synchronizer for scalar CDC control signals. In Figure 4.11, if the first register (*R1*) enters a metastable state, it almost always settles to 1 or 0 before the second register (*R2*) reads its output (*q1*). There exist various structured synchronizers, such as the 3DFF synchronizer, 2DFF synchronizer with a pulse (pulse synchronizer), and 4-latch synchronizer.

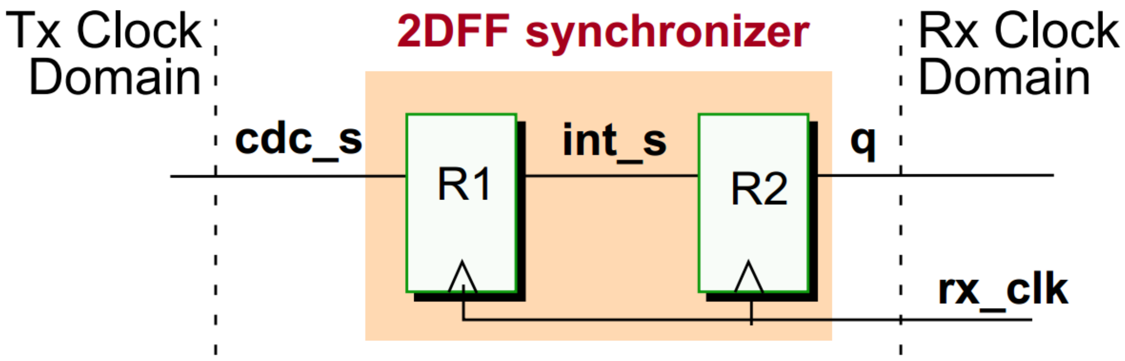


Figure 4.11: 2DFF synchronizer

The connection rules of the 2DFF synchronizers re as follows:

1) No glitches in the path of *cdc_s*
2) No combinational logic is permitted the path of *int_s*
3) The *cdc_s* signal must be held stable by the transmit clock domain logic for at least the following:

$$period_{rx\_clk} + t_{setup} + t_{hold} + t_{\max\_skew}$$

Another example for the 2DFF synchronizer is shown in Figure 4.12.

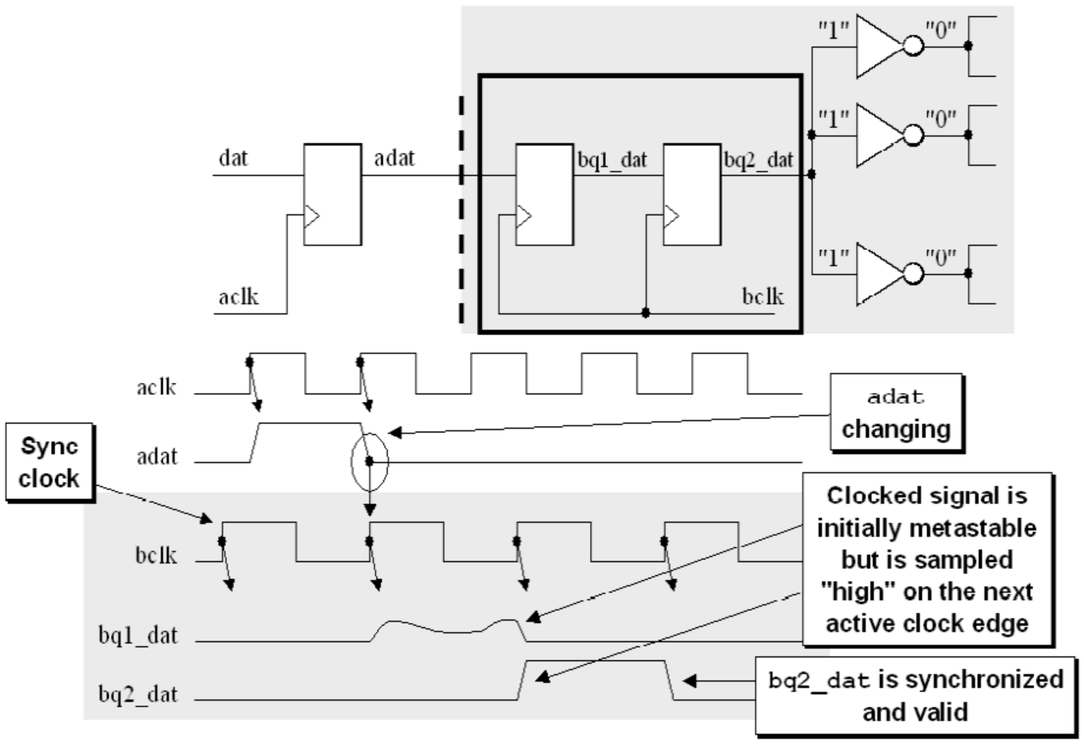

"1"
"0"
dat
adat
bq1_dat
bq2_dat
aclk
bclk
aclk
adat
Sync clock
adat changing
bclk
Clocked signal is initially metastable but is sampled "high" on the next active clock edge
bq1_dat
bq2_dat
bq2_dat is synchronized and valid

Figure 4.12: 2DFF synchronizer in operation [23]

2DFF synchronizers are adequate for synchronizing CDC control signals, but not data vectors (i.e. buses). Control signal synchronization does not ensure that correlated bits of a bus are transmitted together, since variable delays on any bit of the bus corrupt the data. Data vector synchronizers (i.e. bus synchronizers) ensure that all bits of the bus are transmitted together and prevents the corruption of data. FIFO, DMUX, and handshake synchronization schemes are used to synchronize vector CDC data using different logic configurations.

An example for the DMUX synchronizer is shown in Figure 4.12, the control select signal from the TX clock domain (which is synchronized using a 2DFF synchronizer) enables a multiplexer (MUX) when the transmitted data value is ready. Following connection rules should be respected:

1) 2DFF synchronizer must obey CDC transfer protocol for *tx_sel*.
2) *cdc_d* must be held stable by the transmit clock domain logic while *tx_sel* or *rx_sel* are asserting.

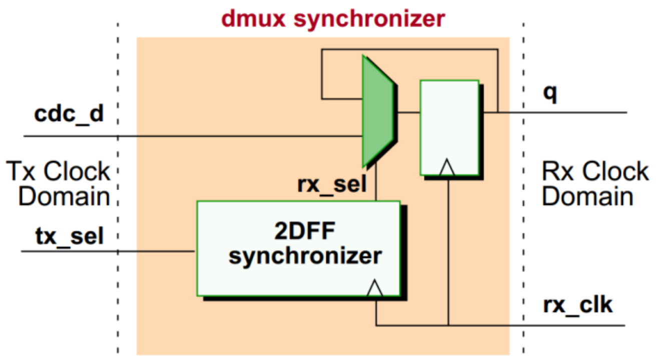


Figure 4.12: DMUX synchronizer

Asynchronous FIFO synchronizers can be used for sending and receiving multiple bits between two different clock domains. The multi-bit signals can be either data bits or control bits. An asynchronous FIFO is a dual port memory in which the data is inserted from the write clock domain and data is removed from the read clock domain. Since both transmitter and receiver operate within their own respective clock domains, using a dual-port buffer, such as a FIFO, is a safe way to pass multi-bit values between clock domains. An example for the asynchronous FIFO synchronizer is shown in figure 4.13. As long as the FIFO is not full, the data or control words can be inserted into the FIFO, and the receiver can read a control or data word from the FIFO as long as it is not empty. The operation of the asynchronous FIFO synchronizer is described in details in [49], its detailed structure us shown in Figure 4.14.

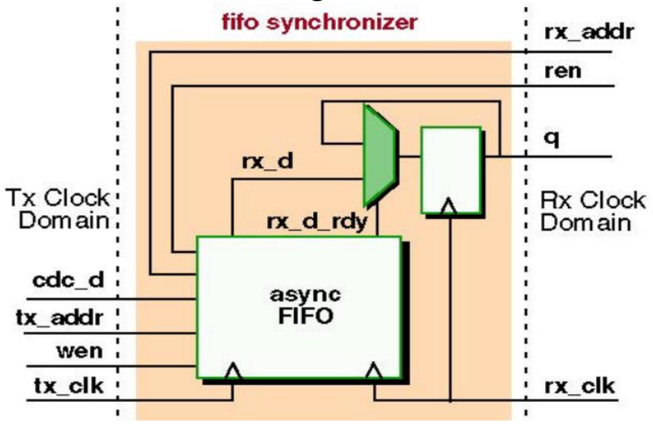


Figure 4.13: FIFO synchronizer

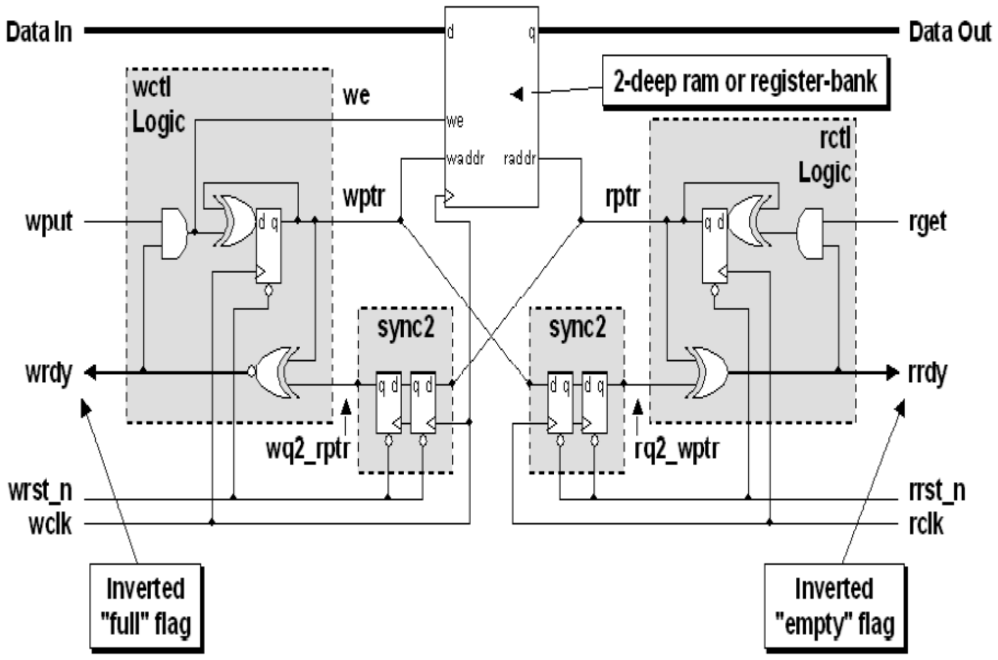
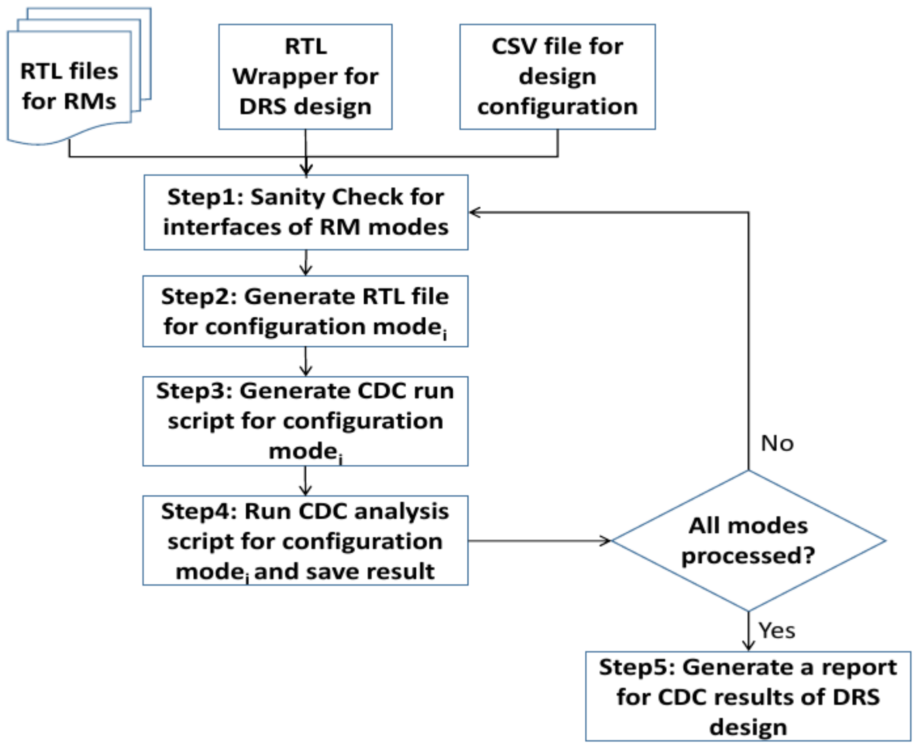


Figure 4.14: FIFO synchronizer detailed structure

4.4. CDC Verification Flow for DRS Designs

The proposed CDC Verification flow is shown in Figure 4.15.



RTL files for RMs
RTL Wrapper for DRS design
CSV file for design configuration
Step1: Sanity Check for interfaces of RM modes
Step2: Generate RTL file for configuration mode$_i$
Step3: Generate CDC run script for configuration mode$_i$
Step4: Run CDC analysis script for configuration mode$_i$ and save result
All modes processed?
No
Yes
Step5: Generate a report for CDC results of DRS design

Figure 4.15: Proposed flow for CDC verification

A Perl utility is implemented to automate the flow. The inputs for the utility are 1) RTL files of RMs modes, 2) RTL wrapper for DRS design and 3) Comma Separated Values (CSV) file to define the configuration modes of the design. In a typical DPR design flow, the RTL files of the RMs modes and the wrapper of the DRS design should be provided by the designer, so there is no extra effort needed for the creation of these files to use the proposed CDC verification flow. Following is an example for Verilog RTL code which defines two modes of the RM (*ModuleA*) in Figure 4.1:

```
module ModuleA_mode1 ( input wire in1 , in2 , a_rst , clk1 ,output reg out1 ) ;
always @( posedge clk1 , posedge a _ r s t )
begin
   if ( a_rst ) out1 <= 1 ’ b0 ;
   else out1 <= in1 | in2 ;
end
endmodule

module ModuleA_mode2 ( input wire in1 , in2 , a_rst , clk1 , output reg out1 ) ;
always @( posedge clk1 , posedge a _ r s t )
begin
   i f ( a_rst ) out1 <= 1 ’ b0 ;
   e l s e out1 <= in1 & in2 ;
end
endmodule
```

The following Verilog RTL mode shows an example for the wrapper of the DPR design example in Figure 4.1:

```
module RR1( input wire in1 , in2 , a _ r s t , clk1 , output out1 ) ;
/ / Empty . A mode for ModuleA will be instantiated here
endmodule


module RR2( input wire in1 , in2 , a _ r s t , clk1 , output out1 ) ;
/ / Empty . A mode for ModuleB will be instantiated here
endmodule


module RR3( input wire in1 , in2 , a _ r s t , clk1 , output out1 ) ;
/ / Empty . A mode for ModuleC will be instantiated here
endmodule


module RR4( input wire in1 , in2 , a _ r s t , clk1 , output out1 ) ;
/ / Empty . A mode for ModuleD will be instantiated here
endmodule


module DRS1 ( input wire in1 , in2 , in3 , in4 , in5 ,
                    a_rst , clk1 , clk2 , output wire out1 ) ;
wire A_out1 , B_out1 , C_out1 , D_out1 ;
RR1 ModuleA_inst ( in1 , in2 , a _ r s t , clk1 , A_out1 ) ;
```

```
RR2 ModuleB_inst ( in3 , A_out1 , clk1 , B_out1 ) ;
RR3 ModuleC_inst ( in4 , B_out1 , clk2 , C_out1 ) ;
RR4 ModuleD_inst ( in5 , C_out1 , clk2 , D_out1 ) ;
assign out1 = D_out1 ;
endmodule
```

The above code for the RTL wrapper is a placeholder for the DRS design. In each operating mode of the design, there will be a module instantiated inside each RR module, as an example for the DRS design in Figure 4.1, in the first mode (*Config1*) of the design, the module *ModuleA_mode1* will be instantiated inside the module of *RR1,* the module *ModuleB_Mode1* will be instantiated inside the module of *RR2,* the module *ModuleC_Mode1* will be instantiated inside the module of *RR3,* and the module *ModuleD_Mode1* will be instantiated inside the module of *RR4.* The CSV file is needed to define the configuration modes of the design, so that the utility can know how many RRs in the design and what are the RMs mapped to a specific RR. The following CSV file is an example for the DPR design in Figure 4.1:

```
RR , RR1
RR , RR2
RR , RR3
RR , RR4
RM , ModuleA , {ModuleA_Mode1 , ModuleA_Mod2 , ModuleA_Mode3 ,
        ModuleA_Mode4}
RM , ModuleB , {ModuleB_Mode1 , ModuleB_Mod2 , ModuleB_Mode3 ,
        ModuleB_Mode4}
RM , ModuleC , {ModuleC_Mode1 , ModuleC_Mod2 , ModuleC_Mode3 ,
        ModuleC_Mode4}
RM , ModuleD , {ModuleD_Mode1 , ModuleD_Mod2 , ModuleD_Mode3 ,
        ModuleD_Mode4}
ConfigMode , Config1 , {{RR1 , ModuleA_Mode1} , {RR2 , ModuleB_Mode1}
        , {RR3 , ModuleC_Mode1} , {RR4 , ModuleD_Mode1}}
ConfigMode , Config2 , {{RR1 , ModuleA_Mode2} , {RR2 , ModuleB_Mode2}
        , {RR3 , ModuleC_Mode2} , {RR4 , ModuleD_Mode2}}
ConfigMode , Config3 , {{RR1 , ModuleA_Mode3} , {RR2 , ModuleB_Mode3}
        , {RR3 , ModuleC_Mode3} , {RR4 , ModuleD_Mode2}}
ConfigMode , Config4 , {{RR1 , ModuleA_Mode4} , {RR2 , ModuleB_Mode4}
        , {RR3 , ModuleC_Mode4} , {RR4 , ModuleD_Mode4}}
ConfigMode , Config5 , {{RR1 , ModuleA_Mode1} , {RR2 , ModuleB_Mode2}
        , {RR3 , ModuleC_Mode3} , {RR4 , ModuleD_Mode4}}
```

The words **RR**, **RM** and **ConfigMode** are reserved words, they are used to define an RR, RM and a configuration mode for the DRS design respectively.

The first step performed by the utility is a sanity check for the interfaces of the modes of the same RM, for DPR flow it is required to have the same number of ports for the RM modes. The sizes and names of these ports should be the same across the modes of the same RM. For Xilinx [7] tools, if this requirement is violated, the implementation of the DPR flow will fail in the place & route step, which is late in the design cycle. In the proposed Perl utility, the sanity check for interfaces is done to catch any errors as early as possible. The Perl code in Appendix B.1 is used to the check the RMs’ ports.

The second step is to pick one configuration mode of the DRS design and generate an RTL file for this mode. Following is an example for the generated Verilog RTL file for configuration mode (Config1) in the DPR example in Figure 4.1:

```
module RR1 ( input wire in1 , in2 , a _ rst , clk1 , output out1 ) ;
ModuleA_mode1 ModA_1_inst ( in1 , in2 , a _ r s t , clk1 , out1 ) ;
endmodule

module RR2 ( input wire in1 , in2 , a _ r s t , clk1 , output out1 ) ;
ModuleB_mode1 ModB_1_inst ( in1 , in2 , a _ r s t , clk1 , out1 ) ;
endmodule

module RR3 ( input wire in1 , in2 , a _ r s t , clk1 , output out1 ) ;
ModuleC_mode1 ModC_1_inst ( in1 , in2 , a _ r s t , clk1 , out1 ) ;
endmodule

module RR4 ( input wire in1 , in2 , a _ r s t , clk1 , output out1 ) ;
ModuleD_mode1 ModD_1_inst ( in1 , in2 , a _ r s t , clk1 , out1 ) ;
endmodule

module DRS1 ( input wire in1 , in2 , in3 , in4 , in5 ,
                a _ rst , clk1 , clk2 , output wire out1 ) ;
wire A_out1 , B_out1 , C_out1 , D_out1 ;
RR1 ModuleA_inst ( in1 , in2 , a _ rst , clk1 , A_out1 ) ;
RR2 ModuleB_inst ( in3 , A_out1 , clk1 , B_out1 ) ;
RR3 ModuleC_inst ( in4 , B_out1 , clk2 , C_out1 ) ;
RR4 ModuleD_inst ( in5 , C_out1 , clk2 , D_out1 ) ;
assign out1 = D_out1 ;
endmodule
```

The third step is to generate the CDC analysis run script, the generated script is written to be run by Questa CDC tool from Mentor Graphics to perform the CDC analysis on the design. The implemented Perl utility performs some heuristics based on the port names of the DRS design to constrain the design, as an example it defines the ports match *(clk)* regular expression as clocks. Similarly, it defines the ports that match *(rst)* regular expression as resets, and define scan enable and test signals as constants. Following is an example for the generated script to run CDC analysis on configuration mode (Config1) in the DPR example in Figure 4.1:

```
onerror {exit 1}

## Compile the Verilog RTL file generated from Step2
vlib work
vlog RTL_Config1.v

## Constrain the design
netlist clock clk1
netlist clock clk2
netlist reset -async -posedge a_rst

## Put the results in a separate directory
configure output directory Config1_Results

## Run CDC analysis
cdc run -d DRS1

exit 0
```

The fourth step is to run CDC analysis using Questa CDC tool, and save the results. The Perl utility then repeats the first four steps for all the configuration modes of the design. The fifth step is to generate a report for the CDC analysis of DRS design. Following is a sample of the output report for the DRS in Figure 4.1:

```
CDC Results for Mode: Config1
---------------------------------------
      A) Synchronized CDC Paths:
      <None>

      B) Un-synchronized CDC Paths:
          1) From 'ModuleB_inst.ModB_1_inst.out1' (clk1)
             To 'ModuleC_inst.ModC_1_inst.out1' (clk2)
...
```

4.5. Case Study

The value of using the proposed CDC Verification flow is demonstrated by a case study of the SDR system presented in [45,46]. This SDR system is implemented using the DPR flow, it switches between blocks of communication standards 3G, 4G and WIFI. The SDR test case has four reconfigurable modules: 1) convolutional encoder, 2) modulator, 3) Discrete Fourier Transform (DFT), and 4) Inverse Fast Fourier Transform (IFFT). Table 4.1 shows the number of modes per each block.

Table 4.1: Number of modes per each RM of the design under test

Block	Number of Modes
Convolutional Encoder	4
Modulator	3
DFT	2
IFFT	2

The block diagram and the schematic of the SDR design are shown in Figure 4.16 and Figure 4.17 respectively. The design has two clocks, the first clock *(clk)* is used for the channel encoder, while the other clock *(clk2)* is used for the rest of the blocks. It also has one asynchronous reset signal *(reset)*.

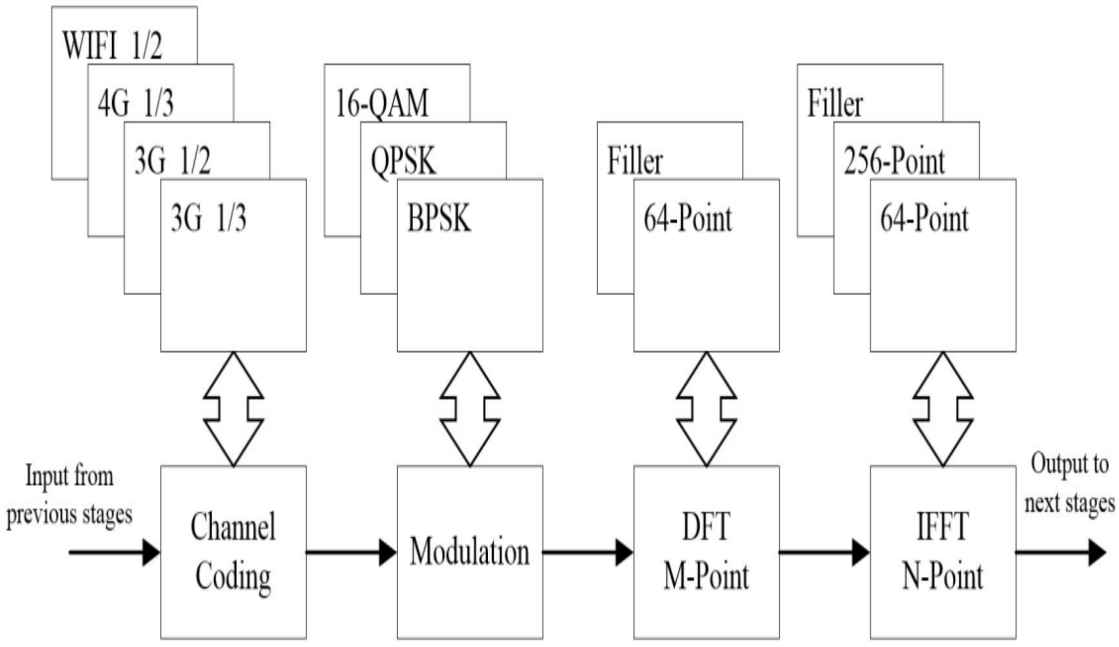


Figure 4.16: Block diagram of the SDR case study

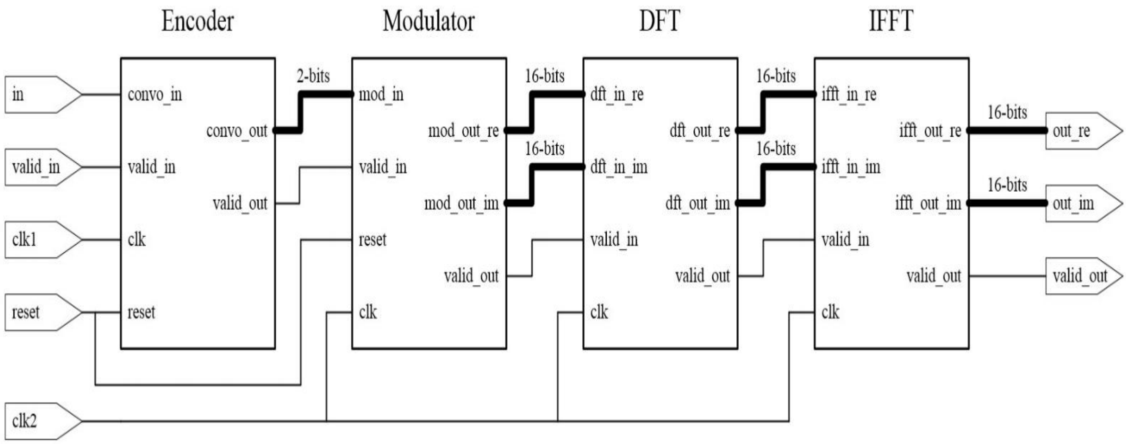


Figure 4.17: Schematic of the SDR case study

The following CSV is provided to the utility for the configuration modes of the design with the RTL files of the reconfigurable modules as explained in the previous section:

```
RR , encoder
RR , modulator
RR , dft
RR , ifft
RM , conv_enc , {enc_3G_half , enc_3G_third , enc_WIFI_half ,
enc_4G_third}
RM , modulator , {bpsk , qpsk , qam_16}
RM , dft , {dft_64_point , filler_mod}
RM , ifft , {ifft_64 , ifft_256 , filler_mod}
ConfigMode , Config1 , {{encoder , enc_3G_half} , {modulator , bpsk} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config2 , {{encoder , enc_3G_half} , {modulator , qpsk} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config3 , {{encoder , enc_3G_half} , {modulator , qam_16} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config4 , {{encoder , enc_3G_third} , {modulator , bpsk} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config5 , {{encoder , enc_3G_third} , {modulator , qpsk} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config6 , {{encoder , enc_3G_third} , {modulator , qam_16} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config7 , {{encoder , enc_WIFI_half} , {modulator , bpsk} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config8 , {{encoder , enc_WIFI_half} , {modulator , qpsk} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config9 , {{encoder , enc_WIFI_half} , {modulator , qam_16} ,
    {dft , filler_mod} , {ifft , filler_mod}}
ConfigMode , Config10 , {{encoder , enc_4G_third} , {modulator , bpsk} ,
    {dft , dft_64} , {ifft , ifft_256}}
ConfigMode , Config11 , {{encoder , enc_4G_third} , {modulator , qpsk} ,
    {dft , dft_64} , {ifft , ifft_256}}
ConfigMode , Config12 , {{encoder , enc_4G_third} , {modulator , qam_16} ,
    {dft , dft_64} , {ifft , ifft_256}}
```

The Perl utility generates RTL design for every mode and a script to run Questa CDC tool for CDC verification, the tool then generates a report for the CDC results for all the runs of the modes of the design.

Using the proposed CDC verification flow, it has identified two CDC errors in all the 12 modes of the design that may cause functional errors during the operation of the system. The first error is found for the signals that are generated in clock domain of *(clk)* inside the convolutional encoder block and sampled in clock domain of *(clk2)* inside the modulator block. The modulator block's design was missing synchronizing these CDC signals to clock domain of *(clk2)* which may cause metastability issues for the registers in the modulator block.

The second error shows up due to the usage of an asynchronous reset signal *(reset)*. The asynchronous reset signal was used without being synchronized to the clock domains of *(clk)* and *(clk2)*. This may cause metastability issues for the registers in the design, because an asynchronous reset signal will be de-asserted asynchronous to the clock signal of the register, so it may violate the reset recovery time requirement for the register. Recovery time is the "minimum required time to the next active clock edge after the reset is released". The Questa CDC results for one of the 4G modes of the design are shown in Figure 4.19, the first two violations are related to the first CDC error (i.e. signals cross from encoder to the modulator), while the other 14 violations are related to the second CDC error (i.e. missing synchronization of the asynchronous reset). The schematic of the first CDC error is shown in Figure 4.18. The design has to be fixed by using CDC data synchronizers for the crossing signals, and an asynchronous reset synchronizer for the *(reset)* signal. The proposed approach can be used again to verify the design after the design is fixed to make sure no more issues CDC issues exist in the design.

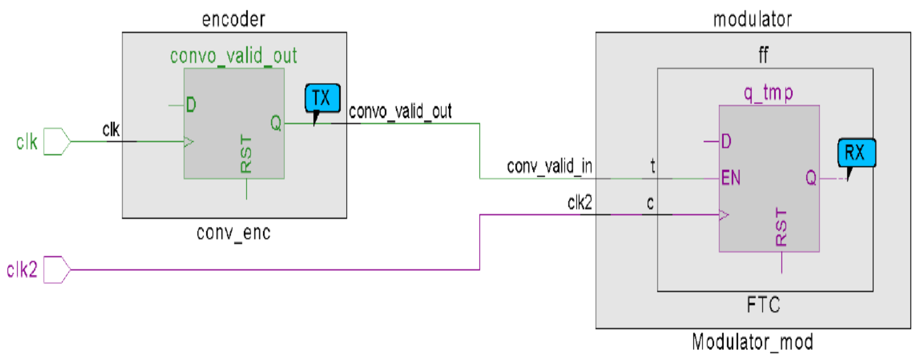
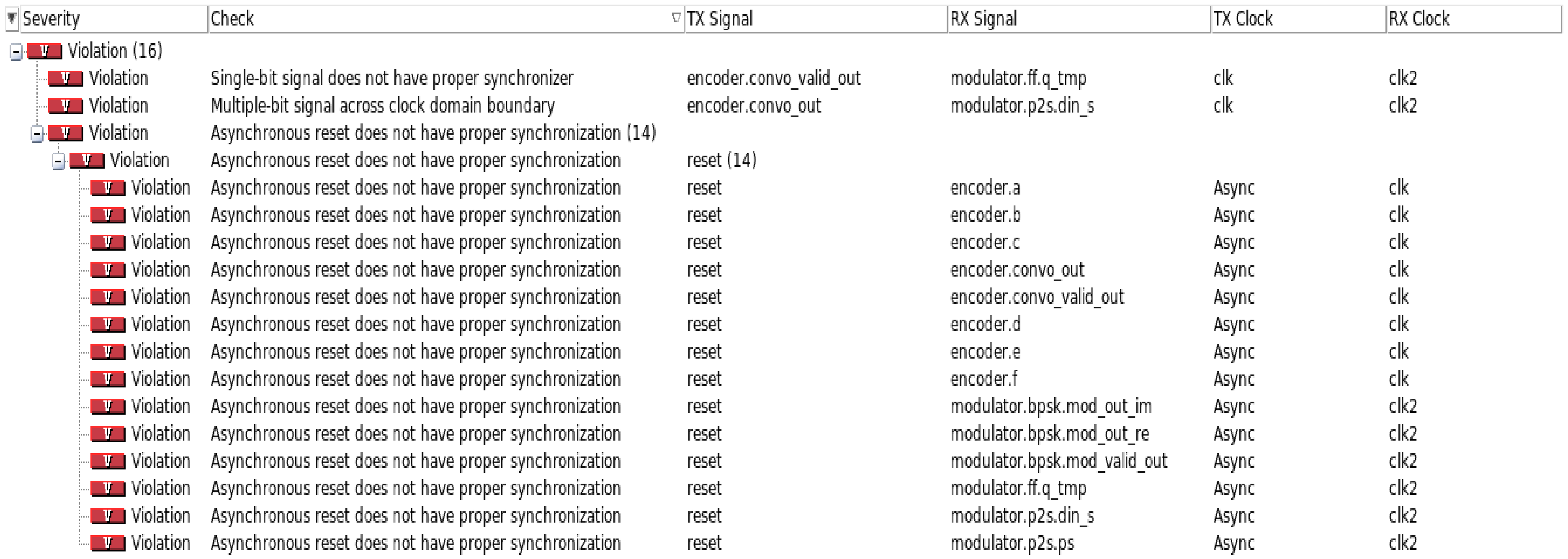


Figure 4.18: Schematic of the first CDC violation in the design

4.6. Summary

CDC verification for digital designs is essential due to the usage of multiple clock domains in the recent designs. The CDC verification for DRS designs is a challenging task due to the lack of CAD tools support for DRS designs and the multiple operating modes of the design. In this chapter a complete automated flow for CDC verification is presented for DRS designs. Designers can use this flow with no extra effort to create the new setup for CDC verification, and it can be easily integrated into the design and verification cycle of DRS designs. The CDC verification should be done before moving to implement the design on the FPGA, as any error caught during CDC verification will force the designs to restart the implementation cycle after fixing the CDC errors in the design. Using a case study from literature, it demonstrated how the proposed CDC verification flow identifies a couple of real CDC errors in the design which were overlooked during the design cycle. In the next chapter, a new methodolody for debugging on FPGAs is proposed, the methodology is utilizing DPR.



Severity	Check	TX Signal	RX Signal	TX Clock	RX Clock
Violation (16)					
Violation	Single-bit signal does not have proper synchronizer	encoder.convo_valid_out	modulator.ff.q_tmp	clk	clk2
Violation	Multiple-bit signal across clock domain boundary	encoder.convo_out	modulator.p2s.din_s	clk	clk2
Violation	Asynchronous reset does not have proper synchronization (14)				
Violation	Asynchronous reset does not have proper synchronization	reset (14)			
Violation	Asynchronous reset does not have proper synchronization	reset	encoder.a	Async	clk
Violation	Asynchronous reset does not have proper synchronization	reset	encoder.b	Async	clk
Violation	Asynchronous reset does not have proper synchronization	reset	encoder.c	Async	clk
Violation	Asynchronous reset does not have proper synchronization	reset	encoder.convo_out	Async	clk
Violation	Asynchronous reset does not have proper synchronization	reset	encoder.convo_valid_out	Async	clk
Violation	Asynchronous reset does not have proper synchronization	reset	encoder.d	Async	clk
Violation	Asynchronous reset does not have proper synchronization	reset	encoder.e	Async	clk
Violation	Asynchronous reset does not have proper synchronization	reset	encoder.f	Async	clk
Violation	Asynchronous reset does not have proper synchronization	reset	modulator.bpsk.mod_out_im	Async	clk2
Violation	Asynchronous reset does not have proper synchronization	reset	modulator.bpsk.mod_out_re	Async	clk2
Violation	Asynchronous reset does not have proper synchronization	reset	modulator.bpsk.mod_valid_out	Async	clk2
Violation	Asynchronous reset does not have proper synchronization	reset	modulator.ff.q_tmp	Async	clk2
Violation	Asynchronous reset does not have proper synchronization	reset	modulator.p2s.din_s	Async	clk2
Violation	Asynchronous reset does not have proper synchronization	reset	modulator.p2s.ps	Async	clk2

Figure 4.19: CDC results from Questa CDC tool for one of the 4G configuration modes of the design

Chapter 5 : Utilizing Dynamic Partial Reconfiguration to Reduce the Cost of FPGA Debugging

Debugging of FPGAs is a difficult task due to the limited access to the internal signals of the design. Embedded logic analyzers enhance the signal observability for FPGAs. These analyzers are implemented on the FPGA resources and they use the embedded memory blocks as trace buffers, so a limited number of signals can be observed using these analyzers due to resources constraints. Changing the traced set of signals requires re-synthesis, placement and routing of the whole design. In this chapter, a new methodology for FPGA debugging is proposed to change dynamically the set of signals to be observed at runtime, and consequently, minimize the time required for debugging. The proposed methodology utilizes the DPR technique to dynamically switch between different sets of signals. DPR creates a reconfigurable module to route each set of signals to an embedded logic analyzer. The proposed approach is demonstrated using Xilinx FPGA tools, finding that changing the set of signals to be observed requires only a few milli-seconds to re-program the reconfigurable region. The area overhead of the proposed methodology is lower than other traditional methods of using multiplexers as the DPR allows the routing module to only use buffers to connect a set of signals to the embedded logic analyzer.

5.1. Introduction

Verification is one of the most challenging tasks in the Integrated Circuits (ICs) development process. Any uncaught bugs or errors during the design and verification phases can cause re-spins for silicon IC. Studies revealed that about half of designer's effort is spent on functional verification [34]. With the increased complexity and size of the designs, traditional functional verification methodologies such as RTL simulation are no longer sufficient to uncover bugs and errors in the design because some real-world interactions only show up when implemented on hardware. The simulation also runs at lower speeds than real hardware execution [35,36] which makes the thorough analysis of large designs infeasible.

Reconfigurability of FPGAs attracts designers to do prototyping for their systems. FPGAs can run at higher speeds than that of simulation, and will catch bugs and errors that cannot be caught in simulation such as system timing issues. Debugging design and system integration issues on FPGAs is a difficult task due to the limited access to internal signals, the designer can only observe the signals connected to the FPGA output pins. Embedded logic analyzers are used to provide visibility for internal signals inside the FPGA [37,38,39]. These analyzers are implemented on the FPGA resources, they use embedded memory blocks as trace buffers. Designers use the Joint Test Action Group (JTAG) port to access the analyzer, and the recorded data can be replayed on a Personal Computer (PC). The traditional design and debug flow for FPGAs is shown in Figure 5.1.

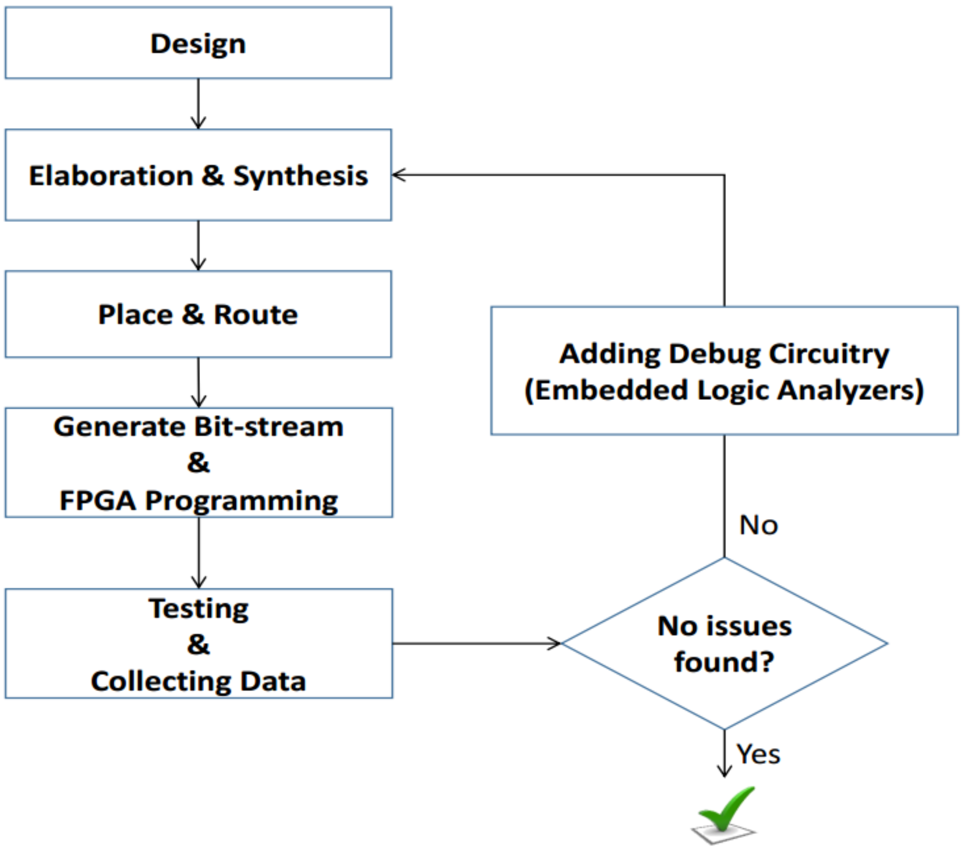

Design
Elaboration & Synthesis
Place & Route
Generate Bit-stream & FPGA Programming
Testing & Collecting Data
No issues found?
No
Yes
Adding Debug Circuitry (Embedded Logic Analyzers)

Figure 5.1: Design and debugging flow for FPGAs

The major disadvantage of using embedded logic analyzers is that the observed signals that are connected to the trace buffer of the embedded logic analyzer are selected before the user design is synthesized, placed and routed. In order to change the set of observed signals, it will require the recompilation of the FPGA design flow. Also, the debug circuitry added in the design consumes a part of the FPGA resources, so the Design Under Test (DUT) may no longer fit in the FPGA device. The amount of resources required for debugging is directly proportional to the number of selected signals to be observed.

DPR on FPGAs permits a portion of the logic to be reconfigured at runtime while the other remaining logic keeps operating. It allows the implementation of complex designs that have multiple operating modes such as SDR applications within a reasonable area on the FPGA. In DPR, the design consists of a number of Reconfigurable Modules (RMs), each module has a number of modes that are swapped at runtime according to the system operating modes. A Reconfigurable Region (RR) is a location on the FPGA in which the reconfigurable module is allocated on. An example for DPR system is shown in Figure 5.2, it has five configuration modes: *Config1*, *Config2*, *Config3*, *Config4* and *Config5*. Each configuration has four reconfigurable modules: *ModuleA, ModuleB, ModuleC* and *ModuleD*, each with four modes: *Mode1*, *Mode2, Mode3 and Mode4*. DPR extends the design flexibility through mapping of multiple reconfigurable modules to the same physical reconfigurable region, which reduces the design cost and the resource usage. In the example of Figure 5.2, the design will have 4 RRs on the FPGA, each RR is used for a unique RM.

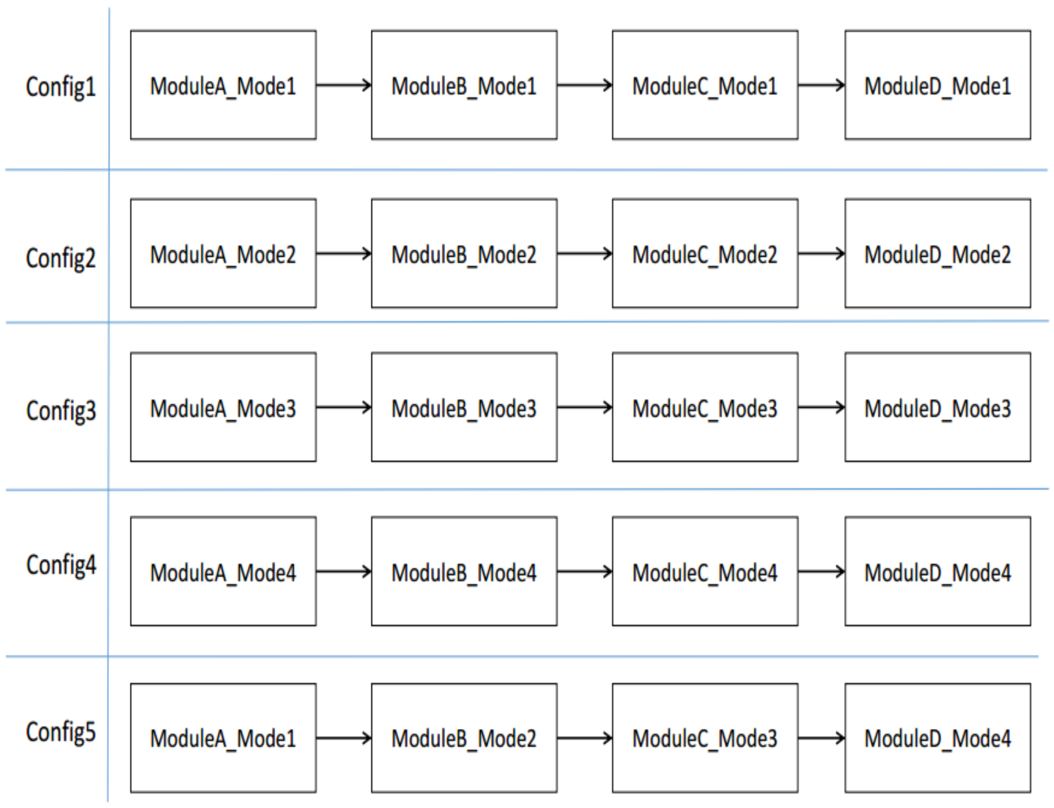


Figure 5.2: An example of DPR design with five modes of configuration and four reconfigurable modules per configuration

The approach proposed in this chapter utilizes DPR on FPGAs to alleviate the issues of using embedded logic analyzers by 1) dividing the large number of all potential signals for debugging *Nsigs* into number of small signals sets *Nsets* (equals *Nsigs/Nprobes*), 2) defining one Reconfigurable Module (RM) in the design, the number of modes for this RM is *Nmodes* (equals *Nsigs/Nprobes*), where *Nprobes* is the number of probes of the embedded logic analyzer. For every mode of the RM, a set of signals is connected to the probes of the embedded logic analyzer. The methodology can be extended to use the output pins of the FPGA for observing the selected signals instead of the embedded logic analyzer, by connecting the outputs of the RM to the output pins of the FPGA, in that case the number of modes (or signals sets) will be equal to the number of signals divided by the number of available output pins for debugging *Nsigs/Nopins*.

The changes in the connections of the signals sets to the analyzer are done at runtime. So, the proposed methodology avoids the recompilation of the whole FPGA flow by changing the observed signals during runtime. Also, it controls the size of the logic analyzer by controlling the number of its probes *Nprobes* without affecting the observability of potential debugging signals, as they are still observable by changing the mode of the RM at runtime. For large designs which need most of the FPGA resources, designers need to keep the number of the analyzer's probes as minimum as possible to limit the size of the analyzer.

5.2. Related Work

Several works have proposed techniques to enhance the debugging of FPGAs using scan-based or trace-based techniques. In [40] a scan-based technique is proposed to connect all the FFs in sequence by using the soft-logic of the FPGA. This technique has a high area overhead due to the usage of the soft-logic to implement the scan-chains in the design.

A bitstream modification technique is presented in [41] to modify the bitstreams within tens of seconds to minutes. This can reduce the time spent in debugging the design, and decrease design's time to market. But, when the selected set of signals for tracing is changed, re-routing needs to be performed which can significantly affect the design's time to market. Software-like debug features are presented in [42] such as watch-points and break-points to enhance debug capability in reconfigurable platforms. But, any change in watch-points or breakpoints needs recompilation of designs.

In [43], a new methodology is proposed to permit a large number of internal signals to be traced for an arbitrary number of clock cycles using a limited number of external pins. It operates without the need for iterative executions of the design re-synthesis, placement and routing tools. This is achieved by inserting a Multiplexer (MUX) into the design implemented on the FPGA, with the MUX inputs are all the signals that designer potentially needs to trace. Then, the select signals of the MUX are controlled by manipulating the bitstream of the design to select different signals to be traced. The disadvantage of this methodology is the area overhead of the MUX, and the need to re-program the whole FPGA for any change in the selected signals to be traced.

5.3. An Approach For FPGA Debugging Using Dynamic Partial Reconfiguration

This section presents a new approach to enhance the observability of FPGA designs for debugging. The traditional debugging flow for FPGA designs is shown in Figure 5.1, the design is synthesized, placed and routed on the target FPGA, then the generated bitstream is used to program the FPGA. During the testing, if an issue is caught, a set of signals is selected to be observed by an embedded logic analyzer, or by routing them to the available output pins. In that case, the designer needs to repeat the FPGA design flow from synthesis to FPGA programming which is time-consuming. Additionally, observing a large number of signals is not feasible in the traditional debugging flow because of the limited resources of the FPGA either for the memory blocks and look-up tables (LUTs) in case of the embedded logic analyzer, or for the output pins in case these pins are used for debugging. This forces the designer to repeat the FPGA design flow multiple times in order to observe different sets of signals to debug different faulty scenarios. For the rest of this chapter, it is assumed that an embedded logic analyzer is being used for debugging for simplicity, the proposed approach and the results presented are still applicable for using the output pins for debugging, the only difference is to replace the number of analyzer's probes by the number of the output pins available for debugging.

A new approach for FPGA debugging is presented in this chapter to overcome the limitations of the traditional FPGA debugging flow. This approach allows the designer to switch between the signals at runtime without the need to repeat the FPGA design flow. This is achieved by inserting a Reconfigurable Module (RM) in the design to switch between the signals to be observed. This RM is implemented on a Reconfigurable Region (RR) on the FPGA. All the potential signals to be observed are connected as inputs to this module. The outputs of the RM are connected to the embedded logic analyzer or the debug output pins. Figure 5.3 shows the connections of the RM. Depending on the available resources on the FPGA, the number of modes of the RM is decided. For a number of signals to be observed *Nsigs* and number of probes *Nprobes* for the embedded logic analyzer, the number of modes of the RM is *Nsigs/Nprobes*.

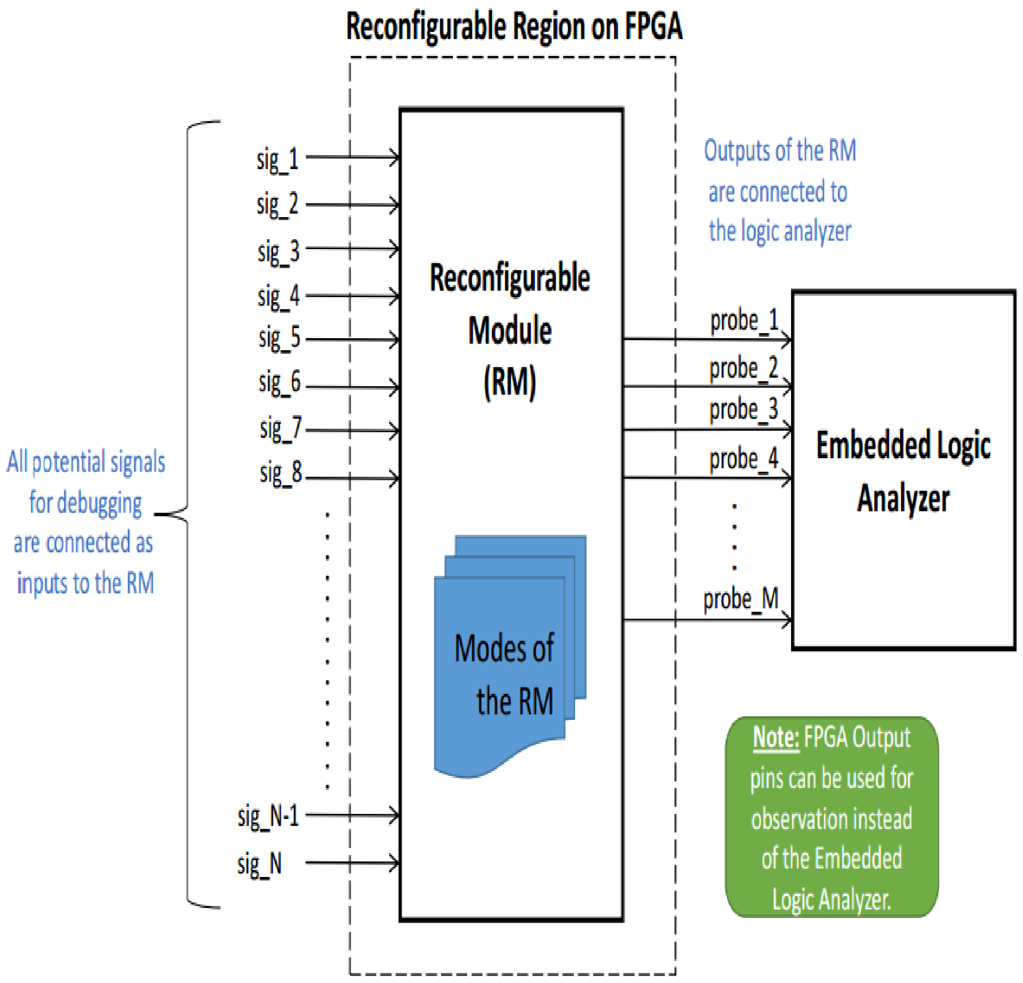

Reconfigurable Region on FPGA
All potential signals for debugging are connected as inputs to the RM
sig_1
sig_2
sig_3
sig_4
sig_5
sig_6
sig_7
sig_8
sig_N-1
sig_N
Reconfigurable Module (RM)
Modes of the RM
Outputs of the RM are connected to the logic analyzer
probe_1
probe_2
probe_3
probe_4
probe_M
Embedded Logic Analyzer
Note: FPGA Output pins can be used for observation instead of the Embedded Logic Analyzer.

Figure 5.3: Reconfigurable module to connect the set of signals to the embedded logic analyzer probes

For each mode of the RM, a set of signals is connected to the probes of the embedded logic analyzer. So, in each mode a, subset of the signals will be used while the others will not be used at all. This allows to keep the unused subset of the signals unconnected. Hence, for each mode, a set of signals is routed to the output while the others remain unconnected, so buffers only will be used to do this connection, and LUTs of the FPGA will not be used. This is a major advantage for using this approach because the area will be as minimum as possible when compared with other approaches that use MUXes to switch between the signals sets such as the proposed approach in [43].

An example for 4-inputs and 2-outputs case is shown in Figure 5.4, the RM will have two modes of operation, the first mode of operation is to connect the first two input signals to the outputs, and the second mode of operation is to connect the second two inputs signals to the outputs. Figure 5.5 shows an example for the case of 8-inputs and 2-output, the RM will have 4 modes of operation, each mode of these four modes will connect a different two inputs signals to the outputs. Figure 5.6 shows an example for the case of 8-inputs and 4-outputs, the RM will have two modes of operation, the first mode of operation is to connect the first four input signals to the outputs, and the second mode of operation is to connect the second four input signals to the outputs. The same synthesis will be applied on other cases which have a higher number of inputs to the RM, i.e. in all the operating modes of the RM, it is sufficient to use 1-input LUTs to act alike buffers to connect the inputs of the RM to the outputs.

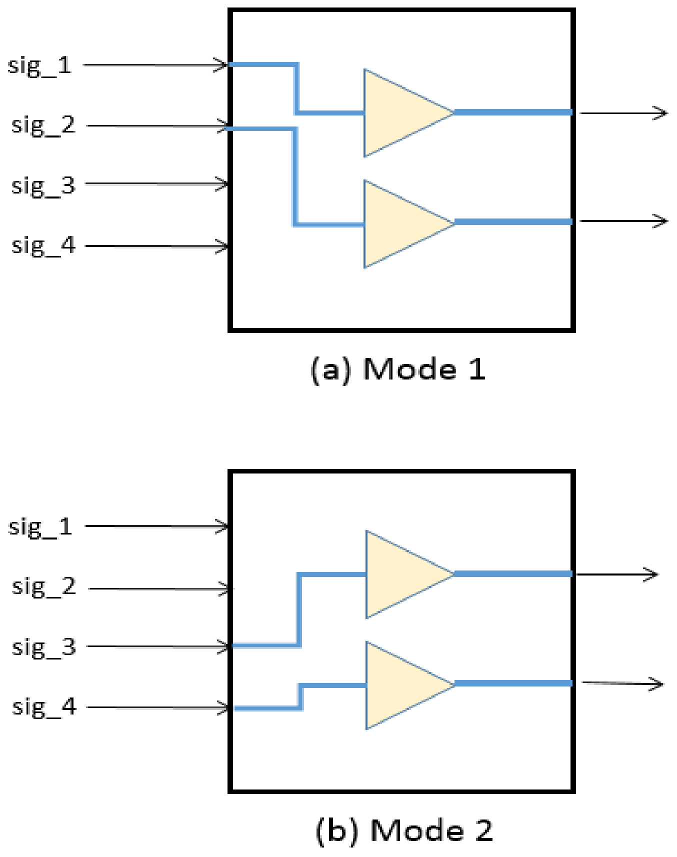


Figure 5.4: Synthesis of the modes of the RM for 4-inputs and 2-outputs case.

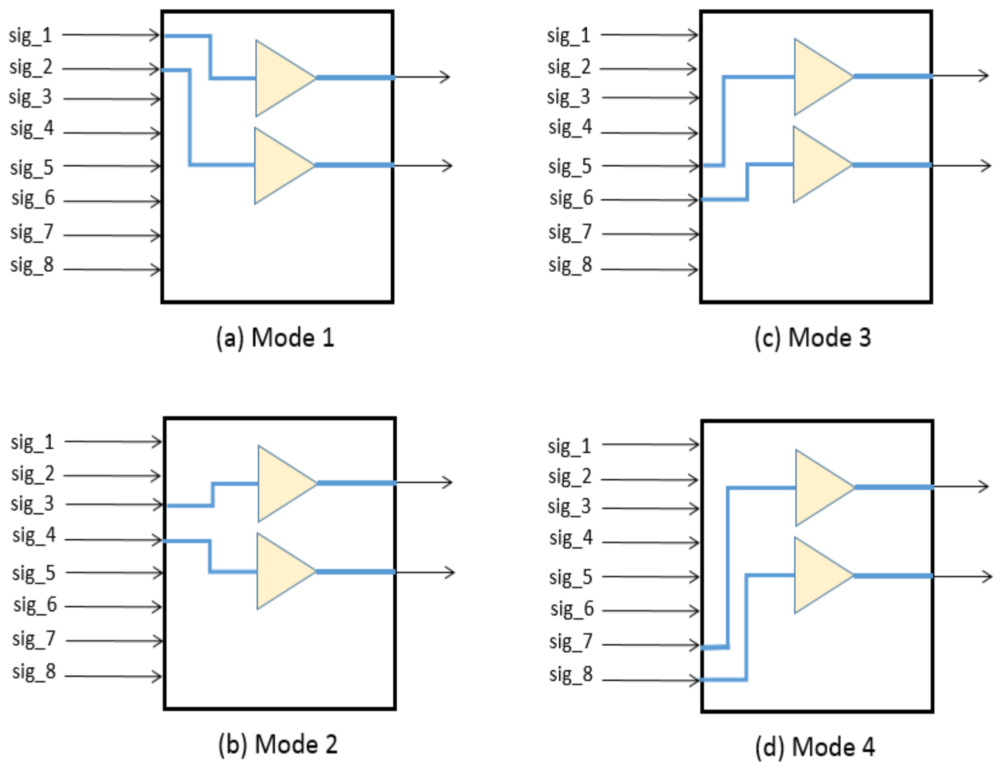


Figure 5.5: Synthesis of the modes of the RM for 8-inputs and 2-outputs case.

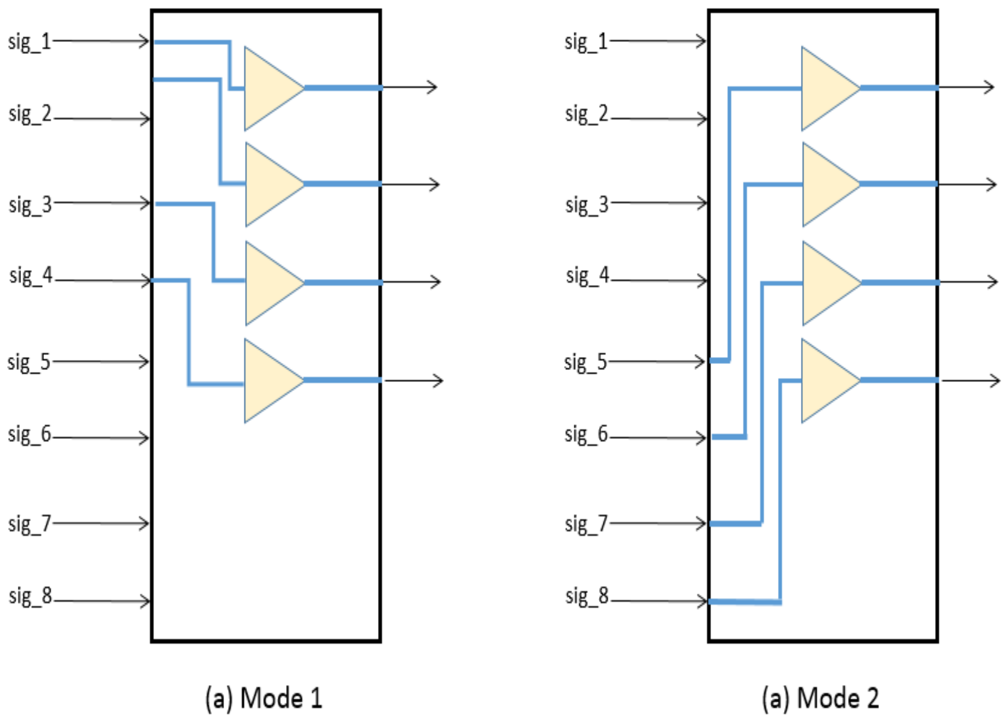


Figure 5.6: Synthesis of the modes of the RM for 8-inputs and 4-outputs case.

For each mode, a partial bitstream is generated and it will be used to re-program the RR at runtime. Partial bitstreams are generated during the DPR design flow and are saved into an external memory. The reconfigurable region size is affecting the size of the partial bitstream in a directly proportional relationship [7]. Since the area consumed by each mode of the RM is very small because it only uses buffers, the size of the partial bitstream will be small, and consequently, the reconfiguration will require a few milli-seconds to re-program the RR. The small reconfiguration time is a major advantage for the proposed approach in this work when compared with the traditional FPGA debugging flow as it avoids re-compilation, and also when compared with other approaches which do modifications in the bitstream then re-program the whole FPGA as in [43]. The proposed FPGA debugging flow is shown in Figure 5.7.

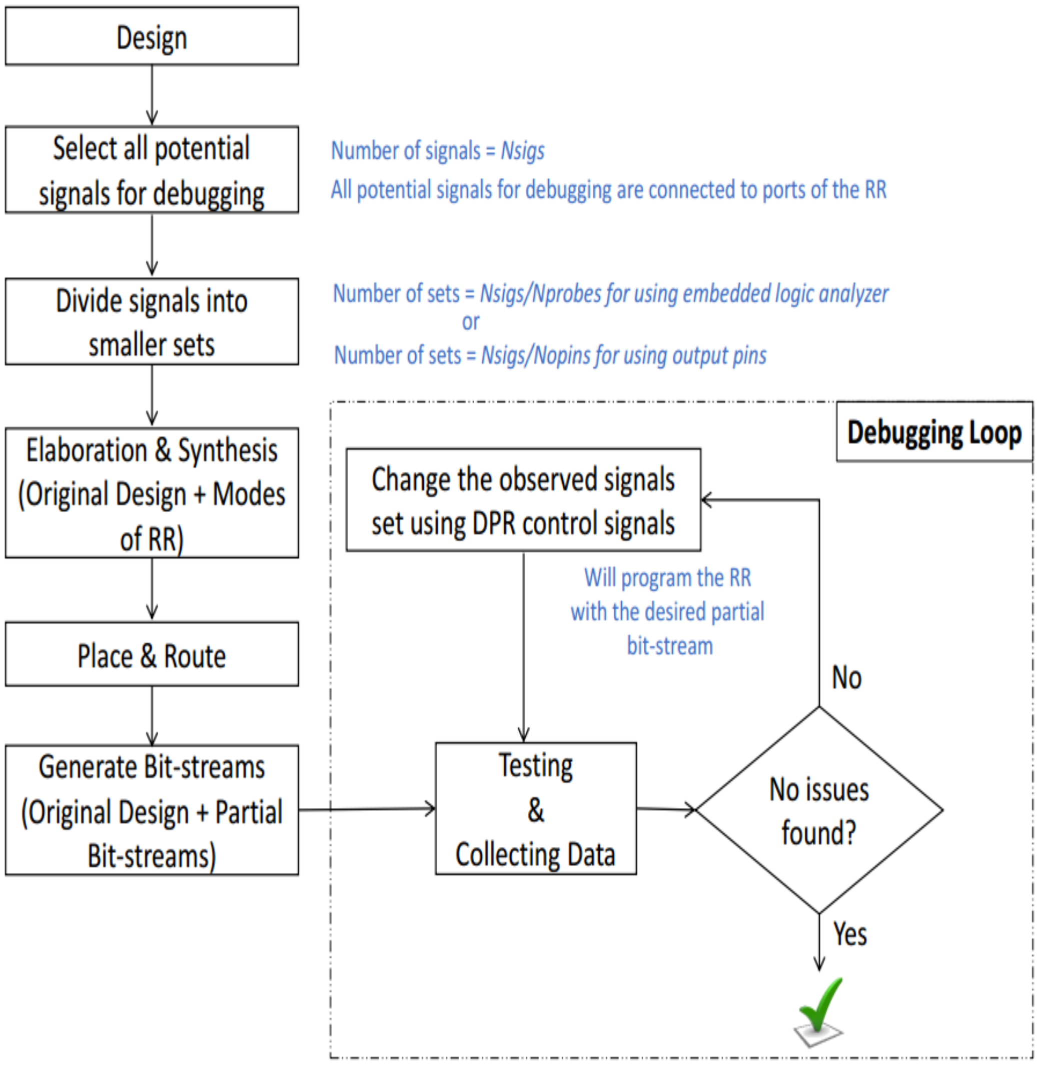

Design
Select all potential signals for debugging
Number of signals = Nsigs
All potential signals for debugging are connected to ports of the RR
Divide signals into smaller sets
Number of sets = Nsigs/Nprobes for using embedded logic analyzer
or
Number of sets = Nsigs/Nopins for using output pins
Elaboration & Synthesis (Original Design + Modes of RR)
Place & Route
Generate Bit-streams (Original Design + Partial Bit-streams)
Debugging Loop
Change the observed signals set using DPR control signals
Will program the RR with the desired partial bit-stream
Testing & Collecting Data
No issues found?
No
Yes

Figure 5.7: Proposed FPGA debugging flow.

In order to generate multiple designs to evaluate the performance of the proposed mechanism, the Perl code in Appendix B.2 is implemented to generate RTL designs and run scripts for Vivado.

1

The following RTL Verilog file is showing one of the generated designs using the script, it is for the trace setting of 128-16 (i.e. 128 signals to be traced in total, and only 16 of them are traced concurrently), the file in Appendix B.3 is used as a test case for debugging using MUX'es to compare it with the behavior of the proposed debugging flow using DPR.

When using DPR instead of the MUXes for debugging, the *ila_mux* module is replaced by the modes of the RM, below is an example of the first mode (out of 8 modes) for the 128-16 trace settings (i.e. 128 signals to be traced in total, and only 16 of them are traced concurrently).

```
module ila_mux ( in1 , in2 , in3 , in4 , in5 , in6 , in7 , in8 , in9 , in10 , in11 , in12 , in13 , in14 , in15
, in16 , in17 , in18 , in19 , in20 , in21 , in22 , in23 , in24 , in25 , in26 , in27 , in28 , in29 , in30 ,
in31 , in32 , in33 , in34 , in35 , in36 , in37 , in38 , in39 , in40 , in41 , in42 , in43 , in44 , in45 , in46
, in47 , in48 , in49 , in50 , in51 , in52 , in53 , in54 , in55 , in56 , in57 , in58 , in59 , in60 , in61 ,
in62 , in63 , in64 , in65 , in66 , in67 , in68 , in69 , in70 , in71 , in72 , in73 , in74 , in75 , in76 , in77
, in78 , in79 , in80 , in81 , in82 , in83 , in84 , in85 , in86 , in87 , in88 , in89 , in90 , in91 , in92 ,
in93 , in94 , in95 , in96 , in97 , in98 , in99 , in100 , in101 , in102 , in103 , in104 , in105 , in106 ,
in107 , in108 , in109 , in110 , in111 , in112 , in113 , in114 , in115 , in116 , in117 , in118 , in119 ,
in120 , in121 , in122 , in123 , in124 , in125 , in126 , in127 , in128 ,  out1 , out2 , out3 , out4 , out5 ,
out6 , out7 , out8 , out9 , out10 , out11 , out12 , out13 , out14 , out15 , out16 )  ;
        // Parameters
        parameter DATA_WIDTH = 1 ;
        // I/O ports
        input [ DATA_WIDTH - 1 : 0 ] in1 , in2 , in3 , in4 , in5 , in6 , in7 , in8 , in9 , in10 , in11 ,
in12 , in13 , in14 , in15 , in16 , in17 , in18 , in19 , in20 , in21 , in22 , in23 , in24 , in25 , in26 , in27
, in28 , in29 , in30 , in31 , in32 , in33 , in34 , in35 , in36 , in37 , in38 , in39 , in40 , in41 , in42 ,
in43 , in44 , in45 , in46 , in47 , in48 , in49 , in50 , in51 , in52 , in53 , in54 , in55 , in56 , in57 , in58
, in59 , in60 , in61 , in62 , in63 , in64 , in65 , in66 , in67 , in68 , in69 , in70 , in71 , in72 , in73 ,
in74 , in75 , in76 , in77 , in78 , in79 , in80 , in81 , in82 , in83 , in84 , in85 , in86 , in87 , in88 , in89
, in90 , in91 , in92 , in93 , in94 , in95 , in96 , in97 , in98 , in99 , in100 , in101 , in102 , in103 , in104
, in105 , in106 , in107 , in108 , in109 , in110 , in111 , in112 , in113 , in114 , in115 , in116 , in117 ,
in118 , in119 , in120 , in121 , in122 , in123 , in124 , in125 , in126 , in127 , in128 ;
        output wire [ DATA_WIDTH - 1 : 0 ] out1 , out2 , out3 , out4 , out5 , out6 , out7 , out8 , out9
, out10 , out11 , out12 , out13 , out14 , out15 , out16 ;
        // Logic
        assign out1 = ~in1 ;
        assign out2 = ~in9 ;
        assign out3 = ~in17 ;
        assign out4 = ~in25 ;
        assign out5 = ~in33 ;
        assign out6 = ~in41 ;
        assign out7 = ~in49 ;
        assign out8 = ~in57 ;
        assign out9 = ~in65 ;
        assign out10 = ~in73 ;
        assign out11 = ~in81 ;
        assign out12 = ~in89 ;
        assign out13 = ~in97 ;
        assign out14 = ~in105 ;
        assign out15 = ~in113 ;
        assign out16 = ~in121 ;
endmodule
```

5.4. Experimental Results

The experiment aims to study the utilization of DPR to minimize the cost of FPGA debugging in terms of area overhead of the reconfigurable module, time of reconfiguration (i.e. time needed to switch between different sets of traced signals), and the usability of the FPGA debugging flow.

5.4.1. System Implementation and Setup

The experimentation is carried out using Xilinx Zynq XC7Z020LG484-1 FPGA and tested with a ZC702 board [44]. The DPR flow has been carried out using Xilinx Vivado tool. The complete system is developed as shown in Figure 5.8. The Zynq FPGA device consists of two parts: i) The Programmable Logic (PL) and ii) The Processing System (PS) part. The PL part contains: 1) the Design Under Test (DUT) that is used as a test case to evaluate the proposed FPGA debugging flow, 2) The reconfigurable partition region which is used to host the reconfigurable module modes of the debugging interfaces, 3) The embedded logic analyzer (Xilinx Integrated Logic Analyzer (ILA)) is used to capture the observed signals and send them to an external PC. The proposed flow can be applied using the output pins of the FPGA instead of the embedded logic analyzer, so the interest of this section is to calculate the performance metrics for the reconfigurable module to compare it with the area-optimized MUX presented in [43]. The PS part contains the ARM processor and the FPGA I/O interfaces to the external ZC702 board peripherals such as UART, SD-Card ... etc. The PS unit is connected with the PL part via AXI bus interfaces. The PS unit is used to send control signals to the DUT and the ILA. The DPR process is done using the serial JTAG external configuration port to load the partial bitstreams of the debugging modes interfaces from an external PC to the FPGA configuration memory with a data rate of 66 Mb/s [7].

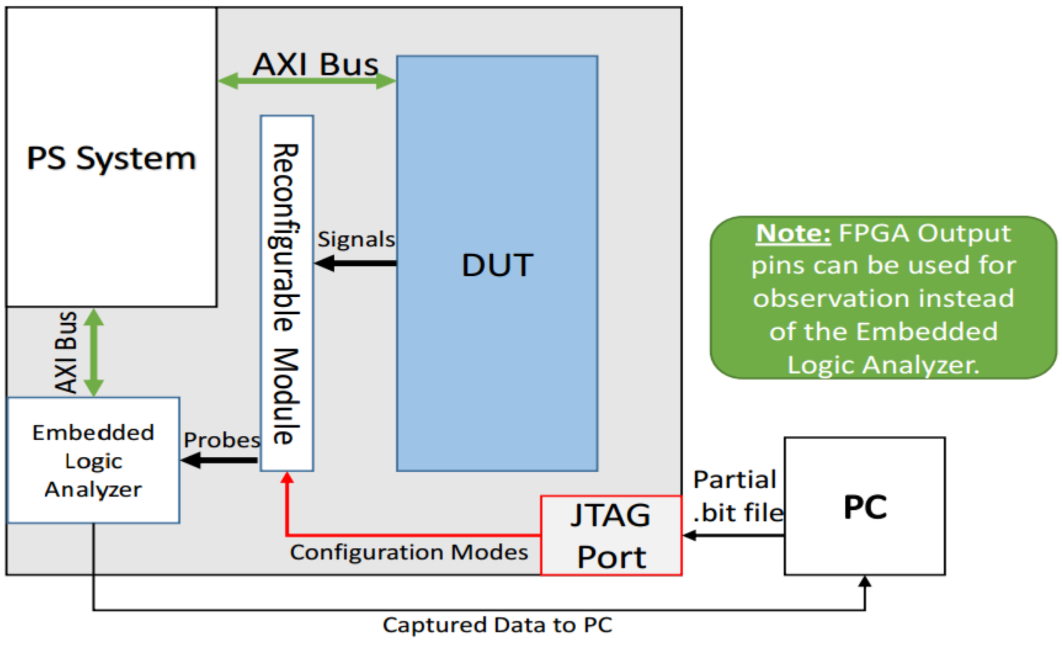


Figure 5.8: Implementation and setup of the test environment for the proposed FPGA debugging flow.

In these experiments, the same DUT setup as in [43] is used to compare the results of the two proposals against each other. The DUT was modified to connect the traced signals to the proposed RM. Xilinx's attribute, *keep*, was used to prevent the removal of these signals during optimization. In the following subsections, the notation, m-w, represents the tracing setting where m signals are candidates for tracing and w signals are traced concurrently.

5.4.2. Area Overhead

The area overheads of the proposed Reconfigurable Module (RM) for 6 different tracing settings are shown in Table 3.1. It is found that the area overhead is directly proportional to the number of signals observed concurrently (i.e. those connected to the embedded logic analyzer), it is not changing with the number of candidate signals for debugging. Xilinx Vivado's place and route tool creates a partition pin for every input port of the RM. Partition pins "are physical connections between static logic and reconfigurable logic, they are automatically created for all Reconfigurable Partition ports" [7]. The partition pins are implemented on the interconnect resources of the RR on the FPGA. In the following table, the notation, m-w, represents the tracing setting where m signals are candidates for tracing and w signals are traced concurrently.

Table 5.1: Area overhead of the RM

Trace Setting	128-2	128-4	128-8
Number of 1-input LUTs (Buffers)	2	4	8

Trace Setting	256-2	256-4	256-8
Number of 1-input LUTs (Buffers)	2	4	8

Table 5.2 reports the area overhead for the proposed structure in [43] in terms of 4-input LUTs. This overhead is calculated by multiplying the number of Adaptive Logic Modules (ALMs) by two, this is because each ALM in an Altera Stratix III device can contain two 4-input LUTs [43]. The area overhead of the proposed approach is smaller than that of [43]. This is expected because two 64:1 MUXes are needed for the 128-2 trace setting in [43], while the proposed DPR approach will only use two 1-input LUTs for the 128-2 trace setting.

Table 5.2: Area overhead of the proposed structure in [43]

Trace Setting	128-2	128-4	128-8
Average Number of 4-input LUTs	50	50	50

Trace Setting	256-2	256-4	256-8
Average Number of 4-input LUTs	100	100	100

5.4.3. Time for Changing the Traced Signal Set

The time needed to switch between different signals sets is equivalent to the reconfiguration time of the RR. The reconfiguration time of the RR is calculated as:

$$t_{reconfig} = \frac{size_{pbs}}{bit\_rate_{jtag}} \tag{1}$$

Where $t_{reconfig}$ is the time to switch between a traced set of signals to another, $size_{pbs}$ is the size of the partial bitstream file, and bit_rate_{jtag} is the bit rate of the JTAG port which is used to re-program the RR on the FPGA. For the setup considered in this work, the bit_rate_{jtag} is *66 Mb/s* [7], and the size of the partial bitstream is *~30 KB*. So, the time to switch between a traced set of signals to another is *3.63 ms*.

The time needed to cover all the signals sets is calculated as:

$$t_{total\_sw} = N_{modes} * t_{reconfig} \tag{2}$$

Where t_{total_sw} is the time needed to cover all the signals sets of the candidate signals for debugging, N_{modes} is the number of modes of the RM that are implemented on the RR, and $t_{reconfig}$ is the time needed to reconfigure the RR as calculated in (1). Table 3.3 shows the total switching time required to trace all the signals sets.

Table 5.3: Total switching time required to trace all the signal sets

Trace Setting	128-2	128-4	128-8
Number of modes	64	32	16
Time to cover all signals sets	232.32 ms	116.16 ms	58.08 ms

Trace Setting	256-2	256-4	256-8
Number of modes	128	64	32
Time to cover all signals sets	464.64 ms	232.32 ms	116.16 ms

The switching time for the proposed debugging flow is much less than that of [43]. In [43], the bitstream should be manipulated to change the select signals of the area optimized MUX, then the whole FPGA needs to be re-programmed. The authors of [43] report that it takes seconds to change the traced signal set. Similarly, the switching time of the DPR proposed flow is much faster than the switching time of the traditional debugging flow which requires minutes for the re-compilation of the FPGA design flow. Another advantage of the proposed flow, is that the switching of the signals sets can be done at runtime, unlike other methodologies which require the whole FPGA to be re-programmed.

5.4.4. Recommendations for FPGA debugging

This section is proposing recommendations for selecting a methodology for FPGA debugging. This is based on the results presented in this chapter, and results of the related works. The recommendations are based on five metrics: 1) area overhead of the debugging structure, 2) concurrent observability of FPGA internal signals, 3) ease of setup, 4) compilation time of the design, and 5) switching time to change the traced signal set. Four methodologies are considered for the recommendations: 1) DPR flow for FPGA Debugging, 2) traditional FPGA debugging flow, 3) area-optimized MUX in [43], and 4) scan-based technique in [40].

It is recommended to use the DPR flow for FPGA debugging for cases in which the designer is interested in low area overhead, and low switching time to change the traced signal set, because the DPR flow has very low area overhead and low switching time as it is shown in this chapter. If the area-overhead is not a problem and full observability and controllability is required, it is recommended to use the scan-based approach in [40], as the scan-based approach for debugging of FPGAs [40] provides full access to the FFs of the FPGA, and consequently, improves the controllability and observability during the debugging process.

If the designers are interested in very low overhead in the compilation time, the area-optimized MUX approach in [43] is recommended, as this optimized MUX approach provides low compilation time as it doesn't add lots of logic, the DPR flow also doesn't add lots of logic but it requires more time during compilation to prepare the partial bit-streams to reconfigure the RR on the FPGA. But, when using the optimized MUX approach [43], the designer should be able to manipulate the bitstream of the FPGA device, which is not an easy task and it is not fully explained in [43]. The traditional flow is recommended to be used for small designs in which a small set of signals are needed for debugging, and there is no need to change this set of signals during debugging, because in such cases no runtime changes are needed for the traced signal set, and hence it doesn't make sense to utilize one of the advanced debugging approaches. The recommendations and comparison in this section are summarized in Table 5.4.

Table 5.4: Recommendations for FPGA debugging flows

	DPR	Area-optimized MUX	Traditional Flow	Scan-based Technique
Area overhead	**Very low**	Low	**No overhead**	Very high
Concurrent observability	Partial	Partial	Partial	**Full**
Ease of setup	**Easy** (DPR flow is well documented)	Hard (Needs bitstream manipulation)	**Easy**	**Easy** (FFs are modified in the RTL)
Compilation time	Moderate (Modes of the RM are compiled)	Low	Lowest	Moderate
Switching time	**Lowest** (Few milli-seconds)	Low (Few seconds)	Very high (Minutes as it needs recompilation)	N/A

5.5. Summary

Debugging of FPGA devices is a difficult task due to the limited access to the internal signals in the design. Traditional debugging flow requires re-compilation of the FPGA design flow in order to change set of observed signals either through embedded logic analyzer or output pins of the FPGA. This chapter presented a new technique to use the DPR design flow to reduce the cost of the debugging on FPGA devices. The new technique has a small area usage as the DPR flow allows the switching between signals to use buffers only to wire a selected signal set to the embedded logic analyzer or the FPGA output pins. The FPGA reconfiguration to switch the traced signal set requires milli-seconds to program the RR on the FPGA.

Chapter 6 : Conclusion and Proposed Future Work

In this research, the problem of functional verification of DPR is discussed, as well as the usage of DPR to improve the effectiveness of debugging on FPGAs. In Chapter 2, an overview is presented for the FPGA structure and technology, as well as the DPR details and terminology. In addition, Chapter 2 discusses the advantages and disadvantages of DPR when compared to the static FPGAs design flow.

Chapter 3 presented an overview about techniques of functional verification and ABV, as well as how to define assertion properties for the design under test. It also addressed the functional verification of DPR specific logic for: 1) synchronization of reconfiguration requests when there is a computation being done by the RM, 2) initialization of the RM after the reconfiguration process is done to make sure the RM is set on an initial state, 3) isolation of the RMs during the reconfiguration process to ensure that there no buggy logic values propagate to the static logic from the RM outputs during the reconfiguration process, and 4) verification of the RM connections to make sure that these connections are not altered when translating the design to utilize the DPR technique. This DPR logic is verified using Assertion Based Verification (ABV) by modeling its functionality using System Verilog Assertion (SVA) properties, then instrument the design with these properties. Following that, these properties are using simulation or formal methods to check the correctness of the DPR logic. The presented framework is demonstrated on a case study from literature. 96 assertions were used to verify the DPR logic of the case study, and 3 functional bugs have been identified in the design which highlights the power of the proposed framework.

Chapter 4 presented an overview about the CDC problem in digital design, and how asynchronous clocks can cause flip-flops to enter a metastable state. After that, it presented the concept of clock domains in digital designs, and the common synchronizers structure that are used to avoid metastability issues. The chapter then presented a flow for performing CDC verification for designs that utilize DPR technique. The presented flow solves the issues of the lack of CAD tools that support DRS. The flow is demonstrated on a case study from literature, and 2 CDC issues have been identified in the designs, these issues should be fixed to avoid metastability issues in the design.

Chapter 5 presented the problem of FPGA debugging due to the limited resources available on the FPGA which prevent the designer to trace all the candidate signals for debugging, and also because of the limited observability and controllability of the internal signals in the design. This chapter proposed the usage of DPR in the problem of FPGA debugging to minimize the resources usage of the added circuitry as well as minimizing the time needed to switch between the traced signal sets for debugging. The proposal involves usage of one RM in the design to multiplex between the candidate signals for debugging at runtime, and since the RM only creates connections between outputs and inputs, the area usage of the RM is minimum as the design will only need 1-input LUTs to connect one input to one output. The proposal is evaluated and compared to a framework which uses a MUX to switch between the different signals for debugging. The proposed approach saved 80% of the area overhead when compared to MUX-based approach. The FPGA reconfiguration to switch the traced signal set requires milli-seconds to program the RR on the FPGA.

6.1. Proposals for Future Work

1. Exploring new functional verification areas for DPR such as power-aware verification and runtime verification

2. Investigate the synthesis of assertion properties defined for DPR logic on the FPGA to help with the debugging process and the run-time verification of the circuit

3. Implementing and developing CAD tools to help with the design and verification process for DPR

References

1. International Roadmap Commitee, “The International Technology Roadmap for Semiconductors”, 2012. [online]. Available: http://www.itrs.net/reports.html
2. Xilinx Inc., “Xilinx Corporate Overview”, 2013. [online]. Available: http://www.xilinx.com/aboutus/corporate_overview.pdf
3. Altera Corporation, “Arria V Device Handbook”, 2013). [online]. Available: http://www.altera.com/literature/lit-arria-v.jsp
4. Xilinx Inc., “Zynq-7000 All Programmable SoC Technical Reference Manual UG585”, 2016.
5. B. Lewis and G. Ramamoorthy, “Market Trends: Worldwide, ASIC and ASSP Design Starts Continue Declining Trend”, 2012. [online]. Available: http://www.gartner.com/DisplayDocument?doc_cd=229088&ref=nl
6. M.J. Wirthlin and B.L. Hutchings, “Improving functional density using run-time circuit reconfiguration”, in IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 6, no. 2, pp. 247-256, 1998.
7. Xilinx Inc., “Partial Reconfiguration User Guide UG909”, 2016.
8. P. Sedcole, B. Blodget, J. Anderson, P. Lysaght and T. Becker, “Modular partial reconfiguration in Virtex FPGAs”, in International Conference on Field-Programmable Logic and Applications (FPL), pp. 211-216, 2005.
9. S. Yusuf, W. Luk, M. Sloman, N. Dulay, E.C. Lupu and G. Brown, “Reconfigurable architecture for network flow analysis”, in IEEE Trans. Very Large Scale Integr. (VLSI) Syst., pp. 57-65, 2008.
10. R. Tessier, K. Pocek and A. DeHon, “Reconfigurable Computing Architectures”, in Proceedings of the IEEE, vol. 103, no. 3, pp. 332-354, 2015.
11. S. M. Trimberger, “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology”, Proceedings of the IEEE, vol. 103, no. 3, pp. 318-331, 2015.
12. J. Delahaye, G. Gogniat, C. Roland and P. Bomel, “Software radio and dynamic reconfiguration on a DSP/FPGA platform”, in Frequenz, vol. 58, no.5, pp. 152-159, 2003.
13. Xilinx Inc., “ISE In-Depth Tutorial (UG695)”, 2010.
14. IEEE Standard 1666-2011: SystemC Language Reference Manual, 2012
15. Xilinx Inc., “Vivado Design Suite Tutorial – High-Level Synthesis (UG871)”, 2012.
16. A. Piziali, “Functional Verification Coverage Measurement and Analysis”, Boston: Kluwer Academic, 2004.
17. Mentor Graphics Corporation, “ModelSim SE User’s Manual (Software Version 10.6)”, 2016.
18. Xilinx Inc., “Synthesis and Simulation Design Guide”, 2010.

19. Altera Corporation, “Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs”, 2010. [online]. Available: http://www.altera.com/literature/wp/wp-01137-stxv-dynamic-partial-reconfig.pdf

20. Altera Corporation, “Quartus II Handbook Version 12.1, Volume 1: Design and Synthesis, Altera Corporation”, 2012.

21. “IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language”, 2012.

22. M. Litterick, “Assertion-Based Verification using System Verilog”, 2007. [online]. Available: http://www.verilab.com/files/svug_2007_abv_litterick.pdf

23. C. Cummings, “Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog”, in Proc. Synopsys User Group Meeting (SNUG), 2008. [online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf

24. U. Farooq et al., “Tree Based Heterogeneous FPGA Architectures, Application Specific Exploration and Optimization”. Springer, pp. 7-48, 2012.

25. P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, “Modular dynamic reconfiguration in Virtex FPGAs,” in IEE, vol. 153, no. 3, pp. 157-164, 2006.

26. Wang lie and Wufeng yan, “Dynamic partial reconfiguration in FPGAs”, in Third International Symposium on Intelligent Information Technology IEEE computer society, pp. 445-448, 2009.

27. I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs”, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 2, pp. 203-215, 2007.

28. A. Schallenberg, W. Nebel, A. Herrholz, and P. A. Hartmann, “OSSS+R: A Framework for Application Level Modelling and Synthesis of Reconfigurable Systems”, in Design, Automation and Test in Europe (DATE), pp. 970-975, 2009.

29. L. Gong and O. Diessel, “ReSim: A Reusable Library for RTL Simulation of Dynamic Partial Reconfiguration”, in Field-Programmable Technology (FPT), International Conference on, pp. 1-8, 2011.

30. L. Gong and O. Diessel, “Functionally Verifying State Saving and Restoration in Dynamically Reconfigurable Systems”, in ACM/SIGDA international symposium on Field Programmable Gate Arrays, pp. 241-244, 2012.

31. I. Kastelan and Z. Krajacevic, “Synthesizable SystemVerilog Assertions as a ethodology for SoC Verification”, in First Eastern European Conference on the Engineering of Computer Based Systems, pp. 120-127, 2009.

32. H. Saafan, M. Watheq and A. Salem, “SoC Connectivity Specification Extraction using Incomplete RTL Design: An Approach for Formal Connectivity Verification”, in International Design & Test Symposium (IDT), pp. 110-114, 2016.

33. Mentor Graphics, “Questa Connectivity Check Formal Application”, www.mentor.com/products/fv/questa-connectivity-check.

34. H. Foster, "Challenges of Design and Verification in the SoC Era", 2011. [online]. Available: http://testandverification.com/files/DVConference2011/2_Harry_Foster.pdf

35. B. Hutchings and J. Keeley, "Rapid Post-Map Insertion of Embedded Logic Analyzers for Xilinx FPGAs", in Field-Programmable Custom Computing Machines (FCCM), pp. 72-79, 2014.

36. S. Asaad, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, T. Takken, and J. Tierno, "A Cycle-Accurate, Cycle-Reproducible Multi-FPGA System for Accelerating Multi-core Processor Simulation", in Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 153-162, 2012.

37. Mentor Graphics, "Certus Debug Suite," https://www.mentor.com/products/fv/certus-silicon-debug, July 2017.

38. Xilinx, "ChipScope Pro Software and Cores, User Guide UG029", 2012.

39. Altera, "Quartus II Handbook Version 12.1 Volume 3: Verification", 2012.

40. T. Wheeler, P. Graham, B. E. Nelson, and B. Hutchings, "Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification", in Proceedings of the 11th International Conference on Field Programmable Logic and Applications, pp. 483-492, 2001.

41. P. Graham, B. Nelson, and B. Hutchings, "Instrumenting Bitstreams for Debugging FPGA Circuits" in Field-Programmable Custom Computing Machines, pp. 41-50, 2001.

42. L. Lagadec and D. Picard, "Software-like debugging methodology for reconfigurable platforms", in IEEE International Symposium on Parallel and Distributed Processing, pp. 1-4, 2009.

43. Z. Poulos, Y. S. Yang, J. Anderson, A. Veneris and B. Le, "Leveraging reconfigurability to raise productivity in FPGA functional debug", in Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 292-295, 2012.

44. Xilinx Inc., "ZC702 Evaluation Board, User Guide UG850 (v1.5).

45. A. Sadek, H. Mostafa, and A. Nassar, "On the use of Dynamic Partial Reconfiguration for MultiBand/MultiStandard Software Defined Radio", in International Conference on Electronics, Circuits, and Systems (ICECS), pp. 498-499, 2015.

46. A. Sadek, H. Mostafa, A. Nassar and Y. Ismail, "Towards the implementation of multi-band multi-standard software-defined radio using dynamic partial reconfiguration," in International Journal of Communications Systems, pp. 33-42 2017.

47. Mentor Graphics, "Questa CDC and Formal Functional Verification", www.mentor.com/products/fv/questaformal.

48. William J. Dally and John W. Poulton, Digital Systems Engineering, Cambridge: University Press, 1998

49. Clifford E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design", Proc. Synopsys User Group Meeting (SNUG), 2002. [online]. Available: www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf

50. The 2016 Wilson Research Group Functional Verification Study. [online]. Available: https://blogs.mentor.com/verificationhorizons/blog/2016/08/08/prologue-the-2016-wilson-research-group-functional-verification-study/

51. 1850-2010 - IEEE Standard for Property Specification Language (PSL).

52. Accellera, Open Verification Language Reference Manual.

53. 1364-2005 - IEEE Standard for Verilog Hardware Description Language.

54. I. Robertson, J. Irvine, P. Lysaght, and D. Robinson, "Improved Functional Simulation of Dynamically Reconfigurable Logic", in Field Programmable Logic and Applications (FPL), pp. 541-574, 2002.

55. A. Raabe and A. Felke, "A SystemC Language Extension for High-Level Reconfiguration Modelling", in Specification, Verification and Design Languages (FDL), pp. 55-60, 2008.

56. A. Raabe, P. A. Hartmann, and J. K. Anlauf, "ReChannel: Describing and Simulating Reconfigurable Hardware in SystemC", in ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 13, no.1, pp. 1-18, 2008.

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

58. Kuon, I., et al., "FPGA Architecture: Survey and Challenges", in Foundations and Trends in Electronic Design Automation, vol. 2, no. 2, pp. 135-253, 2007.

59. Actel Corporation, "ProASIC3 Flash Family FPGAs", 2005.

60. D. Koch, "Partial Reconfiguration on FPGAs: Architectures, Tools and Applications", Springer Publishing Company, 2013.

61. Xilinx Inc., "7 Series FPGAs Configuration User Guide UG470", 2016.

62. A. Hassan, R. Ahmed, H. Mostafa, H. A. H. Fahmy and A. Hussien, "Performance evaluation of dynamic partial reconfiguration techniques for software defined radio implementation on FPGA", IEEE International Conference on Electronics, Circuits, and Systems (ICECS), Cairo, pp. 183-186, 2015.

63. K. Papadimitriou, A. Anyfantis and A. Dollas, "An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems", in IEEE Transactions on Instrumentation and Measurement, vol. 59, no. 6, pp. 1642-1651, 2010.

64. J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs", in IEEE Trans. CAD, vol. 13, no. 1, pp. 1-12, 1994.

65. J. Birkner, A. Chan, H. T. Chua, A. Chao, K. Gordon, B. Kleinman, P. Kolze, and R. Wong, “A very-high-speed field-programmable gate array using metal to-metal antifuse programmable elements”, in Microelectronics Journal, pp. 561-568, November 1992.

66. D. C. Guterman, I. H. Rimawi, T. L. Chiu, R. D. Halvorson, and D. J. McElroy, “An electrically alterable nonvolatile memory cell using a floating-gate structure”, in IEEE Trans. Electron Devices, vol. ED-26, no. 4, pp. 576-586, 1979.

67. Altera Corporation, “MAX II Device Handbook”, 2005.

68. Altera Corporation, “Stratix III Device Handbook, version 1.0”, 2006.

69. S. Brown, R. Francis, J. Rose, and Z. Vranesic, “Field-Programmable Gate Arrays”, Kluwer Academic Publishers, 1992.

70. Actel Corporation, “Axcelerator Family FPGAs”, 2005.

71. A. El Gamal, J. Greene, J. Reyneri, E. Rogoyski, K. A. El-Ayat, and A. Mohsen, “An architecture for electrically configurable gate arrays”, in IEEE Journal of Solid-State Circuits, vol. 24, no. 2, pp. 394-398, 1989.

72. V. Betz, J. Rose, and A. Marquardt, “Architecture and CAD for Deep Submicron FPGAs”, Kluwer Academic Publishers, 1999.

73. A. DeHon, “Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don’t really want 100% LUT utilization),” in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp 69-78, 1999.

74. R. K. Brayton, A. L. Sangiovanni-Vincentelli, and G. D. Hachtel, “Multilevel logic synthesis”, Proc. IEEE, vol. 78, pp. 264-300, 1990.

75. M. Dehkordi and S. Brown, “The effect of cluster packing and node duplication control in delay driven clustering,” in IEEE International Conference on Field Programmable Technology, pp. 227-233, 2002.

76. D. J. Huang and A. B. Kahng, “When clusters meet partitions: New density-based methods for circuit decomposition”, in Proc. Eur. Design Test Conf., pp. 60-64, 1995.

77. A. Marquardt, V. Betz, and J. Rose, “Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density,” In ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 37-46, 1999.

78. A. E. Dunlop, "A Procedure for Placement of Standard Cell VLSI Circuits", IEEE Transaction on Computer-Aided Design, vol. 4, pp. 92-98, 1985.

79. C. Alpert, T. Chan, A. Kahng, I. Markov, and P. Mulet, “Faster minimization of linear wirelength for global placement,” IEEE Trans. Computer-Aided Design, vol. 17, no. 1, pp. 3-13, 1998.

80. C. Sechen, A. Sangiovanni-Vincentelli, “The Timberwolf Placement and Routing Package”, in IEEE Journal of Solid-State Circuits, vol. sc-20, no.2, pp. 510-522, 1985.

81. L. McMurchie and C. Ebeling, “Pathfinder: A negotiation-based performance-driven router for FPGAs,” in Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays, pp. 111-117, 1995.

82. T.H. Cormen, C.E. Leiserson, R.L. Rivest, “Introduction to Algorithms, Section 25:2, Dijkstra’s Algorithm”, Cambridge: The MIT Press, 1990.

83. R. B. Hitchcock, G. L. Smith, and D. D. Cheng, “Timing Analysis of Computer Hardware”, IBM J. Research and Development, vol. 26, no. 1, pp. 100-105, 1982.

Appendix A: List of Publications

1. I. Ahmed, H. Mostafa, and A. Mohieldin, “Dynamic Partial Reconfiguration Verification Using Assertion Based Verification”, 13th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS 2018), Taormina, Italy, 2018.
2. I. Ahmed, A. K. Eldin, H. Mostafa, and A. Mohieldin, "Utilizing Dynamic Partial Reconfiguration to Reduce the Cost of FPGA Debugging", The 16th IEEE International NEWCAS Conference (NEWCAS 2018), Montreal, Canada, 2018.
3. I. Ahmed, H. Mostafa, and A. Mohieldin, “On the Functional Verification of Dynamic Partial Reconfiguration”, IEEE 61th International Midwest Symposium on Circuits and Systems (MWSCAS 2018), Windsor, Canada. 2018.
4. I. Ahmed, H. Mostafa, and A. Mohieldin, “Automatic Clock Domain Crossing Verification Flow For Dynamic Partial Reconfiguration”, 2018 IEEE 61th International Midwest Symposium on Circuits and Systems (MWSCAS 2018).Windsor, Canada, 2018.
5. A. K. ELdin, I. Ahmed, A. Obeid, A. Shalash, Y. Ismail, and H. Mostafa, “A Cost-Effective Dynamic Partial Reconfiguration Implementation Flow for Xilinx FPGA”, IEEE International NEW Generation of Circuits and Systems (NGCAS 2017), Genova, Italy.

Appendix B: Codes

B.1. Perl code to check interfaces of RMs ports

```
#! /usr/bin/perl
use FindBin ;
use lib $FindBin::Bin ;
use File::Basename ;
use rvp ;
use Getopt::Long ;
my $scriptName = " check_ports_for_pdr " ;
if ( &GetOptions( "filelist=s"      => \$filelist ,
                  "config=s"        => \$configFile ,
                  "help"            => \$helpOption ) == 0 ) {
  die " ERROR: Illegal command or option. Use ' $scriptName –h ' for help \n" ;
}
if ( $helpOption ) {
  print " Usage : check_ports_for_pdr.pl \n " .
        "           -filelist <File list of Verilog files>\n" .
        "           -config   <CSV file for configurations>\n" ;
  exit 0;
}
if ( ! $filelist ) {
  die "ERROR: Please specify the input filelist of the Verilog files" ;
}
my @files = ` cat $filelist ` ;
chomp( @files ) ;
## Parse the Verilog files
my $vdata = rvp->read_verilog( \@files , [] , {def1=>1} , 1 , [] , [] , '' ) ;
# Print out all the found modules
foreach $module ( $vdata->get_modules() ) {
  print " INFO : Iterating over the ports of module ' $module '\n " ;
  foreach my $port ( @{$vdata->{modules}{$module}{port_order}} ) {
    my $range = $vdata->{modules}{$module}{signals}{$port}{range} ;
    my $type  = $vdata->{modules}{$module}{signals}{$port}{type} ;
    my $size  ;
    if ($range eq '') {
      $size = 1;
    } elsif ( $range =~ m/(\d+):(\d+)/ ) {
      if ( $2 > $1 ) {
        $size = $2 - $1 + 1 ;
      } else {
        $size = $1 - $2 + 1 ;
      }
    }
    print "  $port $range $type $size\n " ;
  }
## Parse the configuration files
if ( ! $configFile ) {
  print " WARNING: No config file is provided, port checks will be skipped \n " ;
  exit 0 ;
}
my %RRs ;
open ( CONFIG_FILE , "<" , "$configFile") or die " ERROR: Config file ' $configFile ' is not
found. \n" ;
while ( <CONFIG_FILE> ) {
  if ( $_ =~ m/^RR/ ) {
    my $RR_name = $_ ;
    my $modules_of_RR = $_ ;
    $RR_name =~ s/(^RR.*?),.*/$1/ ;
    $modules_of_RR =~ s/^RR.*?,// ;
    chomp($RR_name);
    $RRs{modules}{$RR_name} = $modules_of_RR ;
  }
}
foreach ( keys ( $RRs{modules} ) ) {
  print " For RR $_, the following modules exist $RRs{modules}{$_}\n " ;
}

## Checks for every module of an RR
foreach my $RR ( keys ($RRs{modules}) ) {
  my @modules_of_RR = split (',',$RRs{modules}{$RR}) ;
  chomp ( @modules_of_RR ) ;
  if ( scalar ( @modules_of_RR ) <= 1 ) {
    print "ERROR : The Reconfigurable-region '$RR' only has 1 module, it should be part of
the static region \n " ;
  } else {
    print " INFO : The Reconfigurable-region '$RR' has " , scalar ( @modules_of_RR ) , "
modules: " , join ( " ", @modules_of_RR ), " \n " ;
  }

  print " INFO : Checking ports of modules in the Reconfigurable-region '$RR' \n " ;

  my $i = 0 ;
  my $reference_module ;
  my @ports_of_reference_module ;

  foreach my $module ( @modules_of_RR ) {
    my @ports_of_module = @{$vdata->{modules}{$module}{port_order}} ;
    if ( $i == 0 ) {
      $reference_module = $module ;
      @ports_of_reference_module = @{$vdata-
>{modules}{$reference_module}{port_order}} ;
      print "     : The module '$reference_module' will be taken as the reference. \n " ;
    } else {
      print "     : Comparing module '$module' against the reference module
'$reference_module' \n " ;
      print "     : Performing Check #1: Number of ports: \n " ;
      if ( scalar(@ports_of_module) == scalar(@ports_of_reference_module) ) {
        print "     :   Number of ports for both modules matched, both have ",
scalar(@ports_of_module), " ports. \n " ;
```

```
    } else {
      print "     :   Number of ports for both modules is different. " ,
                      " Reference module  '$reference_module ' has
",scalar(@ports_of_reference_module), "ports, ",
                      " while module ' $module ' has ", scalar(@ports_of_module), " ports. \n " ;
    }

    print "     : Performing Check #2: Name, order and size of ports: \n " ;
    for (my $i = 0 ; $i < scalar( @ports_of_reference_module); $i++ ) {
      my $port = $ports_of_module[$i] ;
      my $ref_port = $ports_of_reference_module[$i] ;
      my $port_type = $vdata->{modules}{$module}{signals}{$port}{type} ;
      my $ref_port_type = $vdata-
>{modules}{$reference_module}{signals}{$ref_port}{type} ;
      my $port_range = $vdata->{modules}{$module}{signals}{$port}{range} ;
      my $ref_port_range = $vdata-
>{modules}{$reference_module}{signals}{$ref_port}{range} ;
      my $port_size = get_port_size($port_range) ;
      my $ref_port_size = get_port_size($ref_port_range) ;
      print "     :   Port #$i --> '$module':$ports_of_module[$i] (Type: $port_type, Size:
$port_size) vs. ".
                            "'$reference_module':$ports_of_reference_module[$i] ( Type:
$ref_port_type, Size: $ref_port_size )\n" ;
      if ( ! ( $modules_of_port[$i] eq $modules_of_port[$i] ) ) {
        print "     :     Error for port naming of Port #$i \n " ;
      } else {
        print "     :     Port naming of Port #$i is OK. \n " ;
      }
      if ( ! ( $port_type eq $ref_port_type) ) {
        print "     :     Error for port type of Port #$i \n " ;
      } else {
        print "     :     Port type of Port #$i is OK. \n " ;
      }
      if ( ! ( $port_size == $ref_port_size) ) {
        print "     :     Error for port size of Port #$i \n " ;
      } else {
        print "     :     Port size of Port #$i is OK. \n " ;
      }
    }
   }
   $i++ ;
  }
}
sub get_port_size {
  my $range = $_[0] ;
  if ( $range eq '' ) {
    $size = 1;
  } elsif ( $range =~ m/(\d+):(\d+)/ ) {
    if ( $2 > $1 ) {
      $size = $2 - $1 + 1 ;
    } else {
```

```
    $size = $1 - $2 + 1 ;
   }
  }
  return $size ;
}
```

B.2. Perl code to generate RTL designs and run scripts for Vivado

```
#! /usr/bin/perl

use Getopt::Long ;

my @mux_mode_iters =  ( 8 )  ;
my @DATA_WIDTH_iters =  ( 1 , 2 , 4 , 8 , 16 , 32 , 64 , 128 , 256 )  ; my @num_sigs_iters =
( 4 , 8 , 16 , 32 , 64 , 128 , 256 )  ; @num_sigs_iters =  ( 8 , 16 , 32 , 64 , 128 , 256 )  ;

my $dpr_mode = 1 ;

foreach my $mux_mode_iter  ( @mux_mode_iters )  { foreach my $DATA_WIDTH_iter  (
@DATA_WIDTH_iters )  { foreach my $num_sigs_iter  ( @num_sigs_iters )  {

# Default Config
my $mux_mode = 0 ;
my $DATA_WIDTH = 16 ;
my $num_sigs_observed = 6 ;

# Using the iterations variables
$mux_mode = $mux_mode_iter ;
$DATA_WIDTH = $DATA_WIDTH_iter ;
$num_sigs_observed = $num_sigs_iter ;

my $project_name =
"debug_num${num_sigs_observed}_width${DATA_WIDTH}_mux$mux_mode" ; my
$project_path =
"/home/iahmed/Masters1/Debugging/Vivado_projects/prjs_mux8/${project_name}" ;

`mkdir -p $project_path` ;

my $vivado_file = "$project_path/run_vivado.tcl " ; my $dut_file =
"$project_path/dut_debug.v " ; my $dpr_mux_file1 = "$project_path/ila_mux1.v " ; my
$dpr_mux_file2 = "$project_path/ila_mux2.v " ; my $dpr_mux_file3 =
"$project_path/ila_mux3.v " ; my $dpr_mux_file4 = "$project_path/ila_mux4.v " ; my
$dpr_mux_file5 = "$project_path/ila_mux5.v " ; my $dpr_mux_file6 =
"$project_path/ila_mux6.v " ; my $dpr_mux_file7 = "$project_path/ila_mux7.v " ; my
$dpr_mux_file8 = "$project_path/ila_mux8.v " ;

if  (  $dpr_mode == 1 && $mux_mode == 2 )  {
  open  ( FHDPR1 ,  " > $dpr_mux_file1 " )  ;
  open  ( FHDPR2 ,  " > $dpr_mux_file2 " )  ; } elsif  (  $dpr_mode == 1 && $mux_mode ==
4  )  {
  open  ( FHDPR1 ,  " > $dpr_mux_file1 " )  ;
  open  ( FHDPR2 ,  " > $dpr_mux_file2 " )  ;
  open  ( FHDPR3 ,  " > $dpr_mux_file3 " )  ;
  open  ( FHDPR4 ,  " > $dpr_mux_file4 " )  ; } elsif  (  $dpr_mode == 1 && $mux_mode ==
8  )  {
  open  ( FHDPR1 ,  " > $dpr_mux_file1 " )  ;
  open  ( FHDPR2 ,  " > $dpr_mux_file2 " )  ;
  open  ( FHDPR3 ,  " > $dpr_mux_file3 " )  ;
  open  ( FHDPR4 ,  " > $dpr_mux_file4 " )  ;
  open  ( FHDPR5 ,  " > $dpr_mux_file5 " )  ;
  open  ( FHDPR6 ,  " > $dpr_mux_file6 " )  ;
  open  ( FHDPR7 ,  " > $dpr_mux_file7 " )  ;
  open  ( FHDPR8 ,  " > $dpr_mux_file8 " )  ; }

open  ( FHV ,  " > $project_path/run_vivado.tcl " )  ; open  ( FHD ,  " >
$project_path/dut_debug.v " )  ;

print FHD <<EOL ;
`timescale 1ns / 1ps

module dut ( debug_mode ,  in1 ,  in2 ,  in3 ,  in4 ,  in5 ,  in6 ,  clk1 ,  out1 ,  out2 )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input wire debug_mode ;
  input wire [ DATA_WIDTH - 1 : 0 ] in1 ,  in2 ,  in3 ,  in4 ,  in5 ,  in6 ;
  input wire clk1 ;
  output wire [ DATA_WIDTH - 1 : 0 ] out1 ,  out2 ;
  // Internal     registers  ( to be observed )
  // 256     registers
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg1   ,  reg2   ,  reg3   ,  reg5   ,  reg6   ,
reg7   ,  reg9   ,  reg10  ,  reg11  ,  reg13  ,  reg14  ,  reg15  ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg17  ,  reg18  ,  reg19  ,  reg21  ,  reg22
,  reg23  ,  reg25  ,  reg26  ,  reg27  ,  reg29  ,  reg30  ,  reg31  ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg33  ,  reg34  ,  reg35  ,  reg37  ,  reg38
,  reg39  ,  reg41  ,  reg42  ,  reg43  ,  reg45  ,  reg46  ,  reg47  ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg49  ,  reg50  ,  reg51  ,  reg53  ,  reg54
,  reg55  ,  reg57  ,  reg58  ,  reg59  ,  reg61  ,  reg62  ,  reg63  ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg65  ,  reg66  ,  reg67  ,  reg69  ,  reg70
,  reg71  ,  reg73  ,  reg74  ,  reg75  ,  reg77  ,  reg78  ,  reg79  ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg81  ,  reg82  ,  reg83  ,  reg85  ,  reg86
,  reg87  ,  reg89  ,  reg90  ,  reg91  ,  reg93  ,  reg94  ,  reg95  ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg97  ,  reg98  ,  reg99  ,  reg101 ,
reg102 ,  reg103 ,  reg105 ,  reg106 ,  reg107 ,  reg109 ,  reg110 ,  reg111 ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg113 ,  reg114 ,  reg115 ,  reg117 ,
reg118 ,  reg119 ,  reg121 ,  reg122 ,  reg123 ,  reg125 ,  reg126 ,  reg127 ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg129 ,  reg130 ,  reg131 ,  reg133 ,
reg134 ,  reg135 ,  reg137 ,  reg138 ,  reg139 ,  reg141 ,  reg142 ,  reg143 ;
   (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg145 ,  reg146 ,  reg147 ,  reg149 ,
reg150 ,  reg151 ,  reg153 ,  reg154 ,  reg155 ,  reg157 ,  reg158 ,  reg159 ;
```

```
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg161 ,  reg162 ,  reg163 ,  reg165 ,
reg166 ,  reg167 ,  reg169 ,  reg170 ,  reg171 ,  reg173 ,  reg174 ,  reg175 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg177 ,  reg178 ,  reg179 ,  reg181 ,
reg182 ,  reg183 ,  reg185 ,  reg186 ,  reg187 ,  reg189 ,  reg190 ,  reg191 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg193 ,  reg194 ,  reg195 ,  reg197 ,
reg198 ,  reg199 ,  reg201 ,  reg202 ,  reg203 ,  reg205 ,  reg206 ,  reg207 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg209 ,  reg210 ,  reg211 ,  reg213 ,
reg214 ,  reg215 ,  reg217 ,  reg218 ,  reg219 ,  reg221 ,  reg222 ,  reg223 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg225 ,  reg226 ,  reg227 ,  reg229 ,
reg230 ,  reg231 ,  reg233 ,  reg234 ,  reg235 ,  reg237 ,  reg238 ,  reg239 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg241 ,  reg242 ,  reg243 ,  reg245 ,
reg246 ,  reg247 ,  reg249 ,  reg250 ,  reg251 ,  reg253 ,  reg254 ,  reg255 ;

  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg4 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg8 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg12 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg16 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg20 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg24 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg28 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg32 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg36 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg40 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg44 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg48 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg52 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg56 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg60 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg64 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg68 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg72 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg76 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg80 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg84 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg88 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg92 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg96 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg100 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg104 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg108 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg112 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg116 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg120 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg124 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg128 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg132 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg136 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg140 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg144 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg148 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg152 ;
```

```
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg156 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg160 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg164 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg168 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg172 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg176 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg180 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg184 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg188 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg192 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg196 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg200 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg204 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg208 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg212 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg216 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg220 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg224 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg228 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg232 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg236 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg240 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg244 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg248 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg252 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg256 ;

// Logic of the registers to be observed
always @ ( posedge clk1 )  begin
  reg1   <= in1 | in2 ;
  reg2   <= in1 & in2 ;
  reg3   <= in1 ^ in2 ;
  reg5   <= in1 & in3 ;
  reg6   <= in1 ^ in3 ;
  reg7   <= in1 | in4 ;
  reg9   <= in1 ^ in4 ;
  reg10  <= in1 | in6 ;
  reg11  <= in1 & in6 ;
  reg13  <= in1 | in2 ;
  reg14  <= in1 & in6 ;
  reg15  <= in3 ^ in2 ;

  reg17  <= in5 | in2 ;
  reg18  <= in5 | in2 ;
  reg19  <= in5 | in2 ;
  reg21  <= in5 & in2 ;
  reg22  <= in5 | in2 ;
  reg23  <= in5 | in2 ;
  reg25  <= in5 & in2 ;
  reg26  <= in5 & in2 ;
  reg27  <= in5 | in2 ;
```

```
reg29  <= in5 & in2 ;
reg30  <= in5 & in2 ;
reg31  <= in5 & in2 ;

reg33  <= in6 | in2 ;
reg34  <= in6 | in2 ;
reg35  <= in6 | in2 ;
reg37  <= in6 & in2 ;
reg38  <= in6 | in2 ;
reg39  <= in6 | in2 ;
reg41  <= in6 & in2 ;
reg42  <= in6 & in2 ;
reg43  <= in6 | in2 ;
reg45  <= in6 & in2 ;
reg46  <= in6 & in2 ;
reg47  <= in6 & in2 ;

reg49  <= in1 ^ in2 ;
reg50  <= in1 ^ in2 ;
reg51  <= in1 ^ in2 ;
reg53  <= in4 ^ in2 ;
reg54  <= in4 ^ in2 ;
reg55  <= in4 & in2 ;
reg57  <= in4 & in2 ;
reg58  <= in4 & in2 ;
reg59  <= in4 | in2 ;
reg61  <= in4 | in2 ;
reg62  <= in4 | in2 ;
reg63  <= in4 | in2 ;

reg65  <= in4 | in3 ;
reg66  <= in4 | in3 ;
reg67  <= in4 | in3 ;

reg69  <= in3 | in2 ;
reg70  <= in3 | in2 ;
reg71  <= in3 | in2 ;

reg73  <= in3 & in4 ;
reg74  <= in3 & in4 ;
reg75  <= in3 & in4 ;

reg77  <= in3 | in5 ;
reg78  <= in3 | in5 ;
reg79  <= in3 | in5 ;

reg81  <= in3 | in6 ;
reg82  <= in3 | in6 ;
reg83  <= in3 | in6 ;

reg85  <= in4 | in5 ;
```

```
reg86  <= in4 | in5 ;
reg87  <= in4 | in5 ;

reg89  <= in4 & in5 ;
reg90  <= in4 & in5 ;
reg91  <= in4 & in5 ;

reg93  <= in4 ^ in5 ;
reg94  <= in4 ^ in5 ;
reg95  <= in4 ^ in5 ;

reg97  <= in4 | in5 ;
reg98  <= in4 & in5 ;
reg99  <= in4 ^ in5 ;

reg101 <= in4 & in2 ;
reg102 <= in4 & in1 ;
reg103 <= in4 ^ in3 ;

reg105 <= in5 ^ in6 ;
reg106 <= in5 & in6 ;
reg107 <= in5 & in6 ;

reg109 <= in5 | in6 ;
reg110 <= in5 & in6 ;
reg111 <= in5 ^ in6 ;

reg113 <= in5 ^ in2 ;
reg114 <= in5 ^ in2 ;
reg115 <= in5 ^ in2 ;

reg117 <= in1 & in2 ;
reg118 <= in1 & in2 ;
reg119 <= in1 ^ in2 ;

reg121 <= in3 | in2 ;
reg122 <= in1 | in4 ;
reg123 <= in1 | in5 ;

reg125 <= in1 ^ in3 ;
reg126 <= in1 ^ in4 ;
reg127 <= in1 ^ in5 ;

reg129 <= in1 | in4 ;
reg130 <= in1 | in5 ;
reg131 <= in1 | in6 ;

reg133 <= in4 | in5 ;
reg134 <= in2 | in6 ;
reg135 <= in3 | in5 ;
```

```
reg137 <= in1 & in5 ;
reg138 <= in2 ^ in6 ;
reg139 <= in4 | in4 ;

reg141 <= in1 ^ in4 ;
reg142 <= in2 & in5 ;
reg143 <= in4 | in6 ;

reg145 <= in1 | in2 ;
reg146 <= in6 & in2 ;
reg147 <= in4 ^ in2 ;

reg149 <= in1 & in2 ;
reg150 <= in2 ^ in5 ;
reg151 <= in1 | in6 ;

reg153 <= in1 ;
reg154 <= in2 ;
reg155 <= in3 ;

reg157 <= in4 ;
reg158 <= in5 ;
reg159 <= in6 ;

reg161 <= in1 ;
reg162 <= in2 ;
reg163 <= in4 ;

reg165 <= in1 ;
reg166 <= in2 ;
reg167 <= in5 ;

reg169 <= in1 ;
reg170 <= in2 ;
reg171 <= in6 ;

reg173 <= in1 ;
reg174 <= in3 ;
reg175 <= in4 ;

reg177 <= in1 ;
reg178 <= in3 | in2 ;
reg179 <= in5 | in2 ;

reg181 <= in1 ;
reg182 <= in3 ;
reg183 <= in5 ;

reg185 <= in1 & in2 ;
reg186 <= in3  ;
reg187 <= in6 ;
```

```
reg189 <= in1 ;
reg190 <= in3 ;
reg191 <= in6 ;

reg193 <= in1 | in2 ;
reg194 <= in4 & in2 ;
reg195 <= in6 ^ in2 ;

reg197 <= in1 ;
reg198 <= in4 ;
reg199 <= in6 ;

reg201 <= in1 ;
reg202 <= in5 ^ in2 ;
reg203 <= in6 & in3 ;

reg205 <= in1 ;
reg206 <= in5 ;
reg207 <= in6 ;

reg209 <= in2 ;
reg210 <= in3 ;
reg211 <= in4 ;

reg213 <= in2 ;
reg214 <= in3 ;
reg215 <= in5 ;

reg217 <= in2 ;
reg218 <= in3 ;
reg219 <= in6 ;

reg221 <= in3 ;
reg222 <= in4 ;
reg223 <= in5 ;

reg225 <= in3 ;
reg226 <= in4 ;
reg227 <= in6 ;

reg229 <= in3 | in2 ;
reg230 <= in4 | in2 ;
reg231 <= in6 | in2 ;

reg233 <= in3 & in2 ;
reg234 <= in4 & in2 ;
reg235 <= in6 & in2 ;

reg237 <= in3 ^ in2 ;
reg238 <= in4 ^ in2 ;
```

```
   reg239 <= in6 ^ in2 ;

   reg241 <= in3 & in2 ;
   reg242 <= in4 & in2 ;
   reg243 <= in5 & in2 ;

   reg245 <= in3 | in1 ;
   reg246 <= in4 | in1 ;
   reg247 <= in5 | in1 ;

   reg249 <= in3 | in2 ;
   reg250 <= in4 | in1 ;
   reg251 <= in5 | in6 ;

   reg253 <= in3 & in2 ;
   reg254 <= in4 & in6 ;
   reg255 <= in5 & in2 ;
  end

  // Logic
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid1_inst ( reg1 ,  reg2 ,
reg3 ,  clk1 ,  reg4 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid2_inst ( reg5 ,  reg6 ,
reg7 ,  clk1 ,  reg8 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid3_inst ( reg9 ,  reg10 ,
reg11 ,  clk1 ,  reg12 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid4_inst ( reg13 ,  reg14 ,
reg15 ,  clk1 ,  reg16 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid5_inst ( reg17 ,  reg18 ,
reg19 ,  clk1 ,  reg20 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid6_inst ( reg21 ,  reg22 ,
reg23 ,  clk1 ,  reg24 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid7_inst ( reg25 ,  reg26 ,
reg27 ,  clk1 ,  reg28 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid8_inst ( reg29 ,  reg30 ,
reg31 ,  clk1 ,  reg32 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid9_inst ( reg33 ,  reg34 ,
reg35 ,  clk1 ,  reg36 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid10_inst ( reg37 ,  reg38 ,
reg39 ,  clk1 ,  reg40 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid11_inst ( reg41 ,  reg42 ,
reg43 ,  clk1 ,  reg44 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid12_inst ( reg45 ,  reg46 ,
reg47 ,  clk1 ,  reg48 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid13_inst ( reg49 ,  reg50 ,
reg51 ,  clk1 ,  reg52 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid14_inst ( reg53 ,  reg54 ,
reg55 ,  clk1 ,  reg56 )  ;
   (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid15_inst ( reg57 ,  reg58 ,
reg59 ,  clk1 ,  reg60 )  ;
```

```
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid16_inst ( reg61 ,  reg62 ,
reg63 ,  clk1 ,  reg64 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid17_inst ( reg65 ,  reg66 ,
reg67 ,  clk1 ,  reg68 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid18_inst ( reg69 ,  reg70 ,
reg71 ,  clk1 ,  reg72 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid19_inst ( reg73 ,  reg74 ,
reg75 ,  clk1 ,  reg76 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid20_inst ( reg77 ,  reg78 ,
reg79 ,  clk1 ,  reg80 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid21_inst ( reg81 ,  reg82 ,
reg83 ,  clk1 ,  reg84 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid22_inst ( reg85 ,  reg86 ,
reg87 ,  clk1 ,  reg88 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid23_inst ( reg89 ,  reg90 ,
reg91 ,  clk1 ,  reg92 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid24_inst ( reg93 ,  reg94 ,
reg95 ,  clk1 ,  reg96 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid25_inst ( reg97 ,  reg98 ,
reg99 ,  clk1 ,  reg100 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid26_inst ( reg101 ,
reg102 ,  reg103 ,  clk1 ,  reg104 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid27_inst ( reg105 ,
reg106 ,  reg107 ,  clk1 ,  reg108 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid28_inst ( reg109 ,
reg110 ,  reg111 ,  clk1 ,  reg112 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid29_inst ( reg113 ,
reg114 ,  reg115 ,  clk1 ,  reg116 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid30_inst ( reg117 ,
reg118 ,  reg119 ,  clk1 ,  reg120 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid31_inst ( reg121 ,
reg122 ,  reg123 ,  clk1 ,  reg124 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid32_inst ( reg125 ,
reg126 ,  reg127 ,  clk1 ,  reg128 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid33_inst ( reg129 ,
reg130 ,  reg131 ,  clk1 ,  reg132 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid34_inst ( reg133 ,
reg134 ,  reg135 ,  clk1 ,  reg136 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid35_inst ( reg137 ,
reg138 ,  reg139 ,  clk1 ,  reg140 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid36_inst ( reg141 ,
reg142 ,  reg143 ,  clk1 ,  reg144 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid37_inst ( reg145 ,
reg146 ,  reg147 ,  clk1 ,  reg148 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid38_inst ( reg149 ,
reg150 ,  reg151 ,  clk1 ,  reg152 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid39_inst ( reg153 ,
reg154 ,  reg155 ,  clk1 ,  reg156 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid40_inst ( reg157 ,
reg158 ,  reg159 ,  clk1 ,  reg160 )  ;
```

```
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid41_inst ( reg161 ,
reg162 ,  reg163 ,  clk1 ,  reg164 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid42_inst ( reg165 ,
reg166 ,  reg167 ,  clk1 ,  reg168 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid43_inst ( reg169 ,
reg170 ,  reg171 ,  clk1 ,  reg172 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid44_inst ( reg173 ,
reg174 ,  reg175 ,  clk1 ,  reg176 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid45_inst ( reg177 ,
reg178 ,  reg179 ,  clk1 ,  reg180 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid46_inst ( reg181 ,
reg182 ,  reg183 ,  clk1 ,  reg184 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid47_inst ( reg185 ,
reg186 ,  reg187 ,  clk1 ,  reg188 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid48_inst ( reg189 ,
reg190 ,  reg191 ,  clk1 ,  reg192 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid49_inst ( reg193 ,
reg194 ,  reg195 ,  clk1 ,  reg196 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid50_inst ( reg197 ,
reg198 ,  reg199 ,  clk1 ,  reg200 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid51_inst ( reg201 ,
reg202 ,  reg203 ,  clk1 ,  reg204 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid52_inst ( reg205 ,
reg206 ,  reg207 ,  clk1 ,  reg208 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid53_inst ( reg209 ,
reg210 ,  reg211 ,  clk1 ,  reg212 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid54_inst ( reg213 ,
reg214 ,  reg215 ,  clk1 ,  reg216 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid55_inst ( reg217 ,
reg218 ,  reg219 ,  clk1 ,  reg220 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid56_inst ( reg221 ,
reg222 ,  reg223 ,  clk1 ,  reg224 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid57_inst ( reg225 ,
reg226 ,  reg227 ,  clk1 ,  reg228 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid58_inst ( reg229 ,
reg230 ,  reg231 ,  clk1 ,  reg232 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid59_inst ( reg233 ,
reg234 ,  reg235 ,  clk1 ,  reg236 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid60_inst ( reg237 ,
reg238 ,  reg239 ,  clk1 ,  reg240 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid61_inst ( reg241 ,
reg242 ,  reg243 ,  clk1 ,  reg244 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid62_inst ( reg245 ,
reg246 ,  reg247 ,  clk1 ,  reg248 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid63_inst ( reg249 ,
reg250 ,  reg251 ,  clk1 ,  reg252 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid64_inst ( reg253 ,
reg254 ,  reg255 ,  clk1 ,  reg256 )  ;

  assign out1 =  ( reg4 | reg8 | reg12 | reg16 )  &  ( reg20 | reg24 | reg28 | reg32 )  &  (
reg36 | reg40 | reg44 | reg48 )  &  ( reg52 | reg56 | reg60 | reg64 )  &  ( reg68 | reg72 |
reg76 | reg80 ) & ( reg84 | reg88 | reg92 | reg96 ) & ( reg100 | reg104 | reg108 |
reg112 ) & ( reg116 | reg120 | reg124 | reg128 ) ;
 assign out2 = ( reg132 | reg136 | reg140 | reg144 ) & ( reg148 | reg152 | reg156 |
reg160 ) & ( reg164 | reg168 | reg172 | reg176 ) & ( reg180 | reg184 | reg188 | reg192
) & ( reg196 | reg200 | reg204 | reg208 ) & ( reg212 | reg216 | reg220 | reg224 ) & (
reg228 | reg232 | reg236 | reg240 ) & ( reg244 | reg248 | reg252 | reg256 ) ;

EOL

my $num_of_mux_signals = 0 ;

if ( $mux_mode == 2 ) {
 $num_of_ila_ports = $num_sigs_observed / $mux_mode ;
 $num_of_mux_signals = $num_sigs_observed ; } elsif ( $mux_mode == 4 ) {
 $num_of_ila_ports = $num_sigs_observed / $mux_mode ;
 $num_of_mux_signals = $num_sigs_observed ; } elsif ( $mux_mode == 8 ) {
 $num_of_ila_ports = $num_sigs_observed / $mux_mode ;
 $num_of_mux_signals = $num_sigs_observed ; } else {
 # No MUX is needed
 $num_of_ila_ports = $num_sigs_observed ;
 $num_of_mux_signals =0 ;
}

print "MUX: $num_of_mux_signals\n" ;
print "MUX: $num_of_ila_ports\n" ;
print "MUX: \n" ;
my @sigs_observed ;
my @ila_mux_outs ;
for ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ ) {
  push ( @ila_mux_outs , "ila_mux_out$i" ) ; } for ( my $i=1 ; $i <= $num_sigs_observed
; $i++ ) {
  push ( @sigs_observed , "reg$i" ) ; }

my $sigs_observed_str = join ( " , " , @sigs_observed ) ; my $ila_mux_outs_str = join ( " ,
" , @ila_mux_outs ) ; if ( $num_of_mux_signals != 0 ) {
 print FHD <<EOL ;
 // MUXes for the ILA
 wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_outs_str ;
  (* keep_hierarchy = " yes " *) ila_mux # ( DATA_WIDTH ) ila_mux_inst ( debug_mode ,
$sigs_observed_str , $ila_mux_outs_str ) ; EOL } print FHD <<EOL ;
 // ILA instance
   ila_0 ila_inst_0 (
     .clk ( clk1 ) , // input wire clk EOL if ( $num_of_mux_signals != 0 ) {
 for ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ ) {
   my $probe_num = $i-1 ;
   if ( $i != $num_of_ila_ports ) {
     print FHD "     .probe$probe_num ( $ila_mux_outs[$probe_num] ) , \n" ;
   } else {
     print FHD "     .probe$probe_num ( $ila_mux_outs[$probe_num] ) \n" ;
   }
 }
```

```
} else {
  for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
    my $probe_num = $i-1 ;
    if  (  $i != $num_sigs_observed  )  {
      print FHD "      .probe$probe_num ( $sigs_observed[$probe_num] )  , \n" ;
    } else {
      print FHD "      .probe$probe_num ( $sigs_observed[$probe_num] ) \n" ;
    }
  }
}
print FHD "   )  ;\n" ;
print FHD "endmodule\n" ;
if  (  $mux_mode == 2  )  {
  my @ila_mux_out_ports ;
  for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
    push  ( @ila_mux_out_ports ,  "out$i" )  ;
  }
  my @ila_mux_in_ports ;
  for  ( my $i=1 ; $i <= $num_sigs_observed ; $i++ )  {
    push  ( @ila_mux_in_ports ,  "in$i" )  ;
  }
  my $ila_mux_out_ports_str = join  ( " , " , @ila_mux_out_ports )  ;
  my $ila_mux_in_ports_str = join  ( " , " ,  @ila_mux_in_ports )  ;
  if  (  $dpr_mode != 1  )  {
    print FHD <<EOL ;
module ila_mux ( mode ,  $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ; EOL
  } else {
    print FHD <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ; EOL
  }
  print FHD <<EOL ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
EOL
  if  (  $dpr_mode != 1  )  {
    print FHD <<EOL ;
  input wire mode ;
EOL
  }
  print FHD <<EOL ;
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
  if  (  $dpr_mode != 1  )  {
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHD "  assign out$i = mode ? in" ,  2*$i-1 ,  " : in" ,  2*$i ,  " ;\n" ;
    }
  } else {
    print FHDPR1 <<EOL ;
```

```
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
 // Parameters
 parameter DATA_WIDTH = 1 ;
 // I/O ports
 input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
 output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
 // Logic
EOL
   for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
     print FHDPR1 "  assign out$i = ~in" ,  2*$i-1 ,  " ;\n" ;
   }
   print FHDPR1 <<EOL ;
endmodule
EOL
   print FHDPR2 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
 // Parameters
 parameter DATA_WIDTH = 1 ;
 // I/O ports
 input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
 output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
 // Logic
EOL
   for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
     print FHDPR2 "  assign out$i = ~in" ,  2*$i ,  " ;\n" ;
   }
   print FHDPR2 <<EOL ;
endmodule
EOL
 }
 print FHD <<EOL ;
endmodule
EOL
} elsif  (  $mux_mode == 4  )  {
 my @ila_mux_out_ports ;
 for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
   push  ( @ila_mux_out_ports ,  "out$i" )  ;
 }
 my @ila_mux_in_ports ;
 for  ( my $i=1 ; $i <= $num_sigs_observed ; $i++ )  {
   push  ( @ila_mux_in_ports ,  "in$i" )  ;
 }
 my $ila_mux_out_ports_str = join  ( " , " , @ila_mux_out_ports )  ;
 my $ila_mux_in_ports_str = join  ( " , " ,  @ila_mux_in_ports )  ;
 if  (  $dpr_mode != 1  )  {
   print FHD <<EOL ;
module ila_mux ( mode ,  $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ; EOL
 } else {
   print FHD <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ; EOL
 }
```

```
  print FHD <<EOL ;
module ila_mux ( mode ,  $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
EOL
  if  (  $dpr_mode != 1  )  {
    print FHD <<EOL ;
  input wire [1 : 0 ] mode ;
EOL
  }
  print FHD <<EOL ;
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
  if  (  $dpr_mode != 1  )  {
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHD "   (* keep_hierarchy = \"yes\" *)  mux4_mod # ( DATA_WIDTH )
mux4_mod_inst$i ( mode , in" ,  4*$i-3 ,  " ,  in" ,  4*$i-2 ,  " ,  in" ,  4*$i-1 ,  " ,  in" ,  4*$i ,
" ,  out$i ) " ,    " ;\n" ;
    }
  } else {
    print FHDPR1 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHDPR1 "  assign out$i = ~in" ,  4*$i-3 ,  " ;\n" ;
    }
    print FHDPR1 <<EOL ;
endmodule
EOL
    print FHDPR2 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHDPR2 "  assign out$i = ~in" ,  4*$i-2 ,  " ;\n" ;
    }
    print FHDPR2 <<EOL ;
```

```
endmodule
EOL
    print FHDPR3 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHDPR3 "  assign out$i = ~in" ,  4*$i-1 ,  " ;\n" ;
    }
    print FHDPR3 <<EOL ;
endmodule
EOL
    print FHDPR4 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHDPR4 "  assign out$i = ~in" ,  4*$i ,  " ;\n" ;
    }
    print FHDPR4 <<EOL ;
endmodule
EOL
  }
  print FHD <<EOL ;
endmodule
EOL
  if  (  $dpr_mode != 1  )  {
    print FHD <<EOL ;
module mux4_mod  ( mode ,  in1 ,  in2 ,  in3 ,  in4 ,  out1 ,  out2 )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [1 : 0 ] mode ;
  input [ DATA_WIDTH - 1 : 0 ] in1 ,  in2 ,  in3 ,  in4 ;
  output reg [ DATA_WIDTH - 1 : 0 ] out1 ,  out2 ;
  // Logic
  always @ ( in1 ,  in2 ,  in3 ,  in4 ,  mode )  begin
    case  ( mode )
      2'b00 : out1 <= in1 ;
      2'b01 : out1 <= in2 ;
      2'b10 : out1 <= in3 ;
      default : out1 <= in4 ;
    endcase
  end
endmodule
EOL
  }
} elsif  (  $mux_mode == 8  )  {
  my @ila_mux_out_ports ;
  for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
    push  ( @ila_mux_out_ports ,  "out$i" )  ;
  }
  my @ila_mux_in_ports ;
  for  ( my $i=1 ; $i <= $num_sigs_observed ; $i++ )  {
    push  ( @ila_mux_in_ports ,  "in$i" )  ;
  }
  my $ila_mux_out_ports_str = join  ( " , " , @ila_mux_out_ports )  ;
  my $ila_mux_in_ports_str = join  ( " , " ,  @ila_mux_in_ports )  ;
  if  (  $dpr_mode != 1  )  {
    print FHD <<EOL ;
module ila_mux ( mode ,  $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ; EOL
  } else {
    print FHD <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ; EOL
  }
  print FHD <<EOL ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
EOL
  if  (  $dpr_mode != 1  )  {
    print FHD <<EOL ;
  input wire [2:0 ] mode ;
EOL
  }
  print FHD <<EOL ;
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
  if  (  $dpr_mode != 1  )  {
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHD "   (* keep_hierarchy = \"yes\" *)  mux8_mod # ( DATA_WIDTH )
mux8_mod_inst$i ( mode , in" ,  8*$i-7 ,  " ,  in" ,  8*$i-6 ,  " ,  in" ,  8*$i-5 ,  " ,  in" ,  8*$i-4
,  " ,  in" ,  8*$i-3 ,  " ,  in" ,  8*$i-2 ,  " ,  in" ,  8*$i-1 ,  " ,  in" ,  8*$i ,  " ,  out$i ) " ,   " ;\n" ;
    }
  } else {
    print FHDPR1 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
```

```
 input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
 output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
 // Logic
EOL
   for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
     print FHDPR1 "  assign out$i = ~in" ,  8*$i-7 ,  " ;\n" ;
   }
   print FHDPR1 <<EOL ;
endmodule
EOL
   print FHDPR2 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
 // Parameters
 parameter DATA_WIDTH = 1 ;
 // I/O ports
 input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
 output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
 // Logic
EOL
   for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
     print FHDPR2 "  assign out$i = ~in" ,  8*$i-6 ,  " ;\n" ;
   }
   print FHDPR2 <<EOL ;
endmodule
EOL
   print FHDPR3 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
 // Parameters
 parameter DATA_WIDTH = 1 ;
 // I/O ports
 input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
 output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
 // Logic
EOL
   for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
     print FHDPR3 "  assign out$i = ~in" ,  8*$i-5 ,  " ;\n" ;
   }
   print FHDPR3 <<EOL ;
endmodule
EOL
   print FHDPR4 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
 // Parameters
 parameter DATA_WIDTH = 1 ;
 // I/O ports
 input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
 output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
 // Logic
EOL
   for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
     print FHDPR4 "  assign out$i = ~in" ,  8*$i-4 ,  " ;\n" ;
```

```
    }
    print FHDPR4 <<EOL ;
endmodule
EOL
    print FHDPR5 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHDPR5 "  assign out$i = ~in" ,  8*$i-3 ,  " ;\n" ;
    }
    print FHDPR5 <<EOL ;
endmodule
EOL
    print FHDPR6 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHDPR6 "  assign out$i = ~in" ,  8*$i-2 ,  " ;\n" ;
    }
    print FHDPR6 <<EOL ;
endmodule
EOL
    print FHDPR7 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHDPR7 "  assign out$i = ~in" ,  8*$i-1 ,  " ;\n" ;
    }
    print FHDPR7 <<EOL ;
endmodule
EOL
    print FHDPR8 <<EOL ;
module ila_mux ( $ila_mux_in_ports_str ,  $ila_mux_out_ports_str )  ;
```

```
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [ DATA_WIDTH - 1 : 0 ] $ila_mux_in_ports_str ;
  output wire [ DATA_WIDTH - 1 : 0 ] $ila_mux_out_ports_str ;
  // Logic
EOL
    for  ( my $i=1 ; $i <= $num_of_ila_ports ; $i++ )  {
      print FHDPR8 "  assign out$i = ~in" ,  8*$i ,  " ;\n" ;
    }
    print FHDPR8 <<EOL ;
endmodule
EOL
  }
  print FHD <<EOL ;
endmodule
EOL

  if  (  $dpr_mode != 1  )  {
    print FHD <<EOL ;
module mux8_mod  ( mode ,  in1 ,  in2 ,  in3 ,  in4 ,  in5 ,  in6 ,  in7 ,  in8 ,  out1 ,  out2 )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input [2:0 ] mode ;
  input [ DATA_WIDTH - 1 : 0 ] in1 ,  in2 ,  in3 ,  in4 ,  in5 ,  in6 ,  in7 ,  in8 ;
  output reg [ DATA_WIDTH - 1 : 0 ] out1 ,  out2 ;
  // Logic
  always @ ( in1 ,  in2 ,  in3 ,  in4 ,  in5 ,  in6 ,  in7 ,  in8 ,  mode )  begin
    case  ( mode )
      3'b000  : out1 <= in1 ;
      3'b001  : out1 <= in2 ;
      3'b010  : out1 <= in3 ;
      3'b011  : out1 <= in4 ;
      3'b100  : out1 <= in5 ;
      3'b101  : out1 <= in6 ;
      3'b110  : out1 <= in7 ;
      default : out1 <= in8 ;
    endcase
  end
endmodule
EOL
  }
}
print FHD <<EOL ;
module mid_mod  ( in1 ,  in2 ,  in3 ,  clk1 ,  out1 )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/Os
  input wire [ DATA_WIDTH - 1 : 0 ] in1 ,  in2 ,  in3 ;
  input wire clk1 ;
  output wire [ DATA_WIDTH - 1 : 0 ] out1 ;
  // Logic
  genvar i ;
  for  ( i=0 ; i<DATA_WIDTH ; i=i+1 )  begin: LEAF_GEN
    leaf_mod leaf_inst ( in1[i] ,  in2[i] ,  in3[i] ,  out1[i] )  ;
  end
endmodule

module leaf_mod  ( in1 ,  in2 ,  clk1 ,  out1 )  ;
  // I/Os
  input wire in1 ,  in2 ,  clk1 ;
  output reg out1 ;
  // Internal signals
   (* keep = " true " *)  reg r1 ,  r2 ,  r3 ;
  // Logic
  always @ ( posedge clk1 )  begin
    r1 <= in1 ;
    r2 <= in2 ;
    r3 <= r1 | r2 ;
    out1 <= r3 ;
  end
endmodule
EOL

print FHV <<EOL ;
## project details
set DATA_WIDTH $DATA_WIDTH
set num_of_ila_ports 3
set project_name \"$project_name\"
set project_path \"$project_path\"

## clean project  ( if exists )  ,  and create a new one set to_be_removed [glob -
nocomplain $project_path/debug_*] if { \$to_be_removed != \"\"} {
  file delete -force {*}[glob -nocomplain $project_path/debug_*] } set to_be_removed
[glob -nocomplain $project_path/debug_*] if { \$to_be_removed != \"\" } {
  file delete -force {*}[glob -nocomplain $project_path/vivado*] } create_project
\${project_name} \${project_path} -part xc7z020clg484-1 set_property board_part
xilinx.com:zc702:part0:1.2 [current_project]

file mkdir \"\${project_path}/\${project_name}.srcs/sources_1/new\"

## create the dut file
file copy $dut_file \${project_path}/\${project_name}.srcs/sources_1/new/dut.v

## add files and update file lists
add_files \"\${project_path}/\${project_name}.srcs/sources_1/new/dut.v\"
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
set_property generic \"DATA_WIDTH=$DATA_WIDTH\" [current_fileset]

## create ip
```

```
create_ip -name ila -vendor xilinx.com -library ip -version 5.1 -module_name ila_0 EOL

my $ila_cmd ;
for  ( my $i=$num_of_ila_ports-1 ; $i >= 0 ; $i-- )  {
  $ila_cmd .= "CONFIG.C_PROBE${i}_WIDTH $DATA_WIDTH " ; } print $ila_cmd ,  "\n" ;

print FHV <<EOL ;
set_property -dict [list $ila_cmd CONFIG.C_NUM_OF_PROBES $num_of_ila_ports]
[get_ips ila_0 ] generate_target {instantiation_template} [get_files
\"\${project_path}/\${project_name}.srcs/sources_1/ip/ila_0/ila_0.xci\"]
update_compile_order -fileset sources_1

generate_target all [get_files
\"\${project_path}/\${project_name}.srcs/sources_1/ip/ila_0/ila_0.xci\"]

set_property generate_synth_checkpoint false [get_files
\${project_path}/\${project_name}.srcs/sources_1/ip/ila_0/ila_0.xci]
generate_target all [get_files
\${project_path}/\${project_name}.srcs/sources_1/ip/ila_0/ila_0.xci]
launch_runs synth_1 -jobs 4
wait_on_run synth_1

open_run synth_1 -name synth_1
report_utilization -file \${project_path}/utilization.rpt -hierarchical EOL

close  ( FHV )  ;
close  ( FHD )  ;
}
}
}

if  (  $dpr_mode == 1 && $mux_mode == 2 )  {
  close ( FHDPR1 )  ;
  close ( FHDPR2 )  ;
} elsif  (  $dpr_mode == 1 && $mux_mode == 4  )  {
  close ( FHDPR1 )  ;
  close ( FHDPR2 )  ;
  close ( FHDPR3 )  ;
  close ( FHDPR4 )  ;
} elsif  (  $dpr_mode == 1 && $mux_mode == 8  )  {
  close ( FHDPR1 )  ;
  close ( FHDPR2 )  ;
  close ( FHDPR3 )  ;
  close ( FHDPR4 )  ;
  close ( FHDPR5 )  ;
  close ( FHDPR6 )  ;
  close ( FHDPR7 )  ;
  close ( FHDPR8 )  ;
}
```

B.3. Verilog test case to use for debugging on FPGA using MUX'es and compare it with the behavior of the proposed debugging flow using DPR

```
`timescale 1ns / 1ps

module dut ( debug_mode , in1 , in2 , in3 , in4 , in5 , in6 , clk1 , out1 , out2 ) ;
 // Parameters
 parameter DATA_WIDTH = 1 ;
 // I/O ports
 input wire debug_mode ;
 input wire [ DATA_WIDTH - 1 : 0 ] in1 , in2 , in3 , in4 , in5 , in6 ;
 input wire clk1 ;
 output wire [ DATA_WIDTH - 1 : 0 ] out1 , out2 ;
 // Internal    registers  ( to be observed )
 // 256    registers
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg1  , reg2  , reg3  , reg5  , reg6  , reg7
, reg9  , reg10 , reg11 , reg13 , reg14 , reg15 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg17 , reg18 , reg19 , reg21 , reg22 ,
reg23 , reg25 , reg26 , reg27 , reg29 , reg30 , reg31 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg33 , reg34 , reg35 , reg37 , reg38 ,
reg39 , reg41 , reg42 , reg43 , reg45 , reg46 , reg47 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg49 , reg50 , reg51 , reg53 , reg54 ,
reg55 , reg57 , reg58 , reg59 , reg61 , reg62 , reg63 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg65 , reg66 , reg67 , reg69 , reg70 ,
reg71 , reg73 , reg74 , reg75 , reg77 , reg78 , reg79 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg81 , reg82 , reg83 , reg85 , reg86 ,
reg87 , reg89 , reg90 , reg91 , reg93 , reg94 , reg95 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg97 , reg98 , reg99 , reg101 , reg102 ,
reg103 , reg105 , reg106 , reg107 , reg109 , reg110 , reg111 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg113 , reg114 , reg115 , reg117 , reg118 ,
reg119 , reg121 , reg122 , reg123 , reg125 , reg126 , reg127 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg129 , reg130 , reg131 , reg133 , reg134 ,
reg135 , reg137 , reg138 , reg139 , reg141 , reg142 , reg143 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg145 , reg146 , reg147 , reg149 , reg150 ,
reg151 , reg153 , reg154 , reg155 , reg157 , reg158 , reg159 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg161 , reg162 , reg163 , reg165 , reg166 ,
reg167 , reg169 , reg170 , reg171 , reg173 , reg174 , reg175 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg177 , reg178 , reg179 , reg181 , reg182 ,
reg183 , reg185 , reg186 , reg187 , reg189 , reg190 , reg191 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg193 , reg194 , reg195 , reg197 , reg198 ,
reg199 , reg201 , reg202 , reg203 , reg205 , reg206 , reg207 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg209 , reg210 , reg211 , reg213 , reg214 ,
reg215 , reg217 , reg218 , reg219 , reg221 , reg222 , reg223 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg225 , reg226 , reg227 , reg229 , reg230 ,
reg231 , reg233 , reg234 , reg235 , reg237 , reg238 , reg239 ;
  (* keep = " true " *)  reg [ DATA_WIDTH - 1 : 0 ] reg241 , reg242 , reg243 , reg245 , reg246 ,
reg247 , reg249 , reg250 , reg251 , reg253 , reg254 , reg255 ;

  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg4 ;
  (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg8 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg12 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg16 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg20 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg24 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg28 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg32 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg36 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg40 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg44 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg48 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg52 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg56 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg60 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg64 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg68 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg72 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg76 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg80 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg84 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg88 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg92 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg96 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg100 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg104 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg108 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg112 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg116 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg120 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg124 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg128 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg132 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg136 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg140 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg144 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg148 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg152 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg156 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg160 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg164 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg168 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg172 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg176 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg180 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg184 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg188 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg192 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg196 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg200 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg204 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg208 ;
(* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg212 ;
```

```
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg216 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg220 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg224 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg228 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg232 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg236 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg240 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg244 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg248 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg252 ;
 (* keep = " true " *)  wire [ DATA_WIDTH - 1 : 0 ] reg256 ;

// Logic of the registers to be observed
always @ ( posedge clk1 )  begin
  reg1   <= in1 | in2 ;
  reg2   <= in1 & in2 ;
  reg3   <= in1 ^ in2 ;
  reg5   <= in1 & in3 ;
  reg6   <= in1 ^ in3 ;
  reg7   <= in1 | in4 ;
  reg9   <= in1 ^ in4 ;
  reg10  <= in1 | in6 ;
  reg11  <= in1 & in6 ;
  reg13  <= in1 | in2 ;
  reg14  <= in1 & in6 ;
  reg15  <= in3 ^ in2 ;

  reg17  <= in5 | in2 ;
  reg18  <= in5 | in2 ;
  reg19  <= in5 | in2 ;
  reg21  <= in5 & in2 ;
  reg22  <= in5 | in2 ;
  reg23  <= in5 | in2 ;
  reg25  <= in5 & in2 ;
  reg26  <= in5 & in2 ;
  reg27  <= in5 | in2 ;
  reg29  <= in5 & in2 ;
  reg30  <= in5 & in2 ;
  reg31  <= in5 & in2 ;

  reg33  <= in6 | in2 ;
  reg34  <= in6 | in2 ;
  reg35  <= in6 | in2 ;
  reg37  <= in6 & in2 ;
  reg38  <= in6 | in2 ;
  reg39  <= in6 | in2 ;
  reg41  <= in6 & in2 ;
  reg42  <= in6 & in2 ;
  reg43  <= in6 | in2 ;
  reg45  <= in6 & in2 ;
  reg46  <= in6 & in2 ;
  reg47  <= in6 & in2 ;
```

```
reg49  <= in1 ^ in2 ;
reg50  <= in1 ^ in2 ;
reg51  <= in1 ^ in2 ;
reg53  <= in4 ^ in2 ;
reg54  <= in4 ^ in2 ;
reg55  <= in4 & in2 ;
reg57  <= in4 & in2 ;
reg58  <= in4 & in2 ;
reg59  <= in4 | in2 ;
reg61  <= in4 | in2 ;
reg62  <= in4 | in2 ;
reg63  <= in4 | in2 ;

reg65  <= in4 | in3 ;
reg66  <= in4 | in3 ;
reg67  <= in4 | in3 ;

reg69  <= in3 | in2 ;
reg70  <= in3 | in2 ;
reg71  <= in3 | in2 ;

reg73  <= in3 & in4 ;
reg74  <= in3 & in4 ;
reg75  <= in3 & in4 ;

reg77  <= in3 | in5 ;
reg78  <= in3 | in5 ;
reg79  <= in3 | in5 ;

reg81  <= in3 | in6 ;
reg82  <= in3 | in6 ;
reg83  <= in3 | in6 ;

reg85  <= in4 | in5 ;
reg86  <= in4 | in5 ;
reg87  <= in4 | in5 ;

reg89  <= in4 & in5 ;
reg90  <= in4 & in5 ;
reg91  <= in4 & in5 ;

reg93  <= in4 ^ in5 ;
reg94  <= in4 ^ in5 ;
reg95  <= in4 ^ in5 ;

reg97  <= in4 | in5 ;
reg98  <= in4 & in5 ;
reg99  <= in4 ^ in5 ;

reg101 <= in4 & in2 ;
reg102 <= in4 & in1 ;
```

```
reg103 <= in4 ^ in3 ;

reg105 <= in5 ^ in6 ;
reg106 <= in5 & in6 ;
reg107 <= in5 & in6 ;

reg109 <= in5 | in6 ;
reg110 <= in5 & in6 ;
reg111 <= in5 ^ in6 ;

reg113 <= in5 ^ in2 ;
reg114 <= in5 ^ in2 ;
reg115 <= in5 ^ in2 ;

reg117 <= in1 & in2 ;
reg118 <= in1 & in2 ;
reg119 <= in1 ^ in2 ;

reg121 <= in3 | in2 ;
reg122 <= in1 | in4 ;
reg123 <= in1 | in5 ;

reg125 <= in1 ^ in3 ;
reg126 <= in1 ^ in4 ;
reg127 <= in1 ^ in5 ;

reg129 <= in1 | in4 ;
reg130 <= in1 | in5 ;
reg131 <= in1 | in6 ;

reg133 <= in4 | in5 ;
reg134 <= in2 | in6 ;
reg135 <= in3 | in5 ;

reg137 <= in1 & in5 ;
reg138 <= in2 ^ in6 ;
reg139 <= in4 | in4 ;

reg141 <= in1 ^ in4 ;
reg142 <= in2 & in5 ;
reg143 <= in4 | in6 ;

reg145 <= in1 | in2 ;
reg146 <= in6 & in2 ;
reg147 <= in4 ^ in2 ;

reg149 <= in1 & in2 ;
reg150 <= in2 ^ in5 ;
reg151 <= in1 | in6 ;

reg153 <= in1 ;
reg154 <= in2 ;
```

```
reg155 <= in3 ;

reg157 <= in4 ;
reg158 <= in5 ;
reg159 <= in6 ;

reg161 <= in1 ;
reg162 <= in2 ;
reg163 <= in4 ;

reg165 <= in1 ;
reg166 <= in2 ;
reg167 <= in5 ;

reg169 <= in1 ;
reg170 <= in2 ;
reg171 <= in6 ;

reg173 <= in1 ;
reg174 <= in3 ;
reg175 <= in4 ;

reg177 <= in1 ;
reg178 <= in3 | in2 ;
reg179 <= in5 | in2 ;

reg181 <= in1 ;
reg182 <= in3 ;
reg183 <= in5 ;

reg185 <= in1 & in2 ;
reg186 <= in3  ;
reg187 <= in6 ;

reg189 <= in1 ;
reg190 <= in3 ;
reg191 <= in6 ;

reg193 <= in1 | in2 ;
reg194 <= in4 & in2 ;
reg195 <= in6 ^ in2 ;

reg197 <= in1 ;
reg198 <= in4 ;
reg199 <= in6 ;

reg201 <= in1 ;
reg202 <= in5 ^ in2 ;
reg203 <= in6 & in3 ;

reg205 <= in1 ;
reg206 <= in5 ;
```

```
  reg207 <= in6 ;

  reg209 <= in2 ;
  reg210 <= in3 ;
  reg211 <= in4 ;

  reg213 <= in2 ;
  reg214 <= in3 ;
  reg215 <= in5 ;

  reg217 <= in2 ;
  reg218 <= in3 ;
  reg219 <= in6 ;

  reg221 <= in3 ;
  reg222 <= in4 ;
  reg223 <= in5 ;

  reg225 <= in3 ;
  reg226 <= in4 ;
  reg227 <= in6 ;

  reg229 <= in3 | in2 ;
  reg230 <= in4 | in2 ;
  reg231 <= in6 | in2 ;

  reg233 <= in3 & in2 ;
  reg234 <= in4 & in2 ;
  reg235 <= in6 & in2 ;

  reg237 <= in3 ^ in2 ;
  reg238 <= in4 ^ in2 ;
  reg239 <= in6 ^ in2 ;

  reg241 <= in3 & in2 ;
  reg242 <= in4 & in2 ;
  reg243 <= in5 & in2 ;

  reg245 <= in3 | in1 ;
  reg246 <= in4 | in1 ;
  reg247 <= in5 | in1 ;

  reg249 <= in3 | in2 ;
  reg250 <= in4 | in1 ;
  reg251 <= in5 | in6 ;

  reg253 <= in3 & in2 ;
  reg254 <= in4 & in6 ;
  reg255 <= in5 & in2 ;
end

// Logic
```

```
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid1_inst ( reg1 ,  reg2 ,  reg3 ,
clk1 ,  reg4 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid2_inst ( reg5 ,  reg6 ,  reg7 ,
clk1 ,  reg8 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid3_inst ( reg9 ,  reg10 ,  reg11
,  clk1 ,  reg12 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid4_inst ( reg13 ,  reg14 ,
reg15 ,  clk1 ,  reg16 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid5_inst ( reg17 ,  reg18 ,
reg19 ,  clk1 ,  reg20 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid6_inst ( reg21 ,  reg22 ,
reg23 ,  clk1 ,  reg24 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid7_inst ( reg25 ,  reg26 ,
reg27 ,  clk1 ,  reg28 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid8_inst ( reg29 ,  reg30 ,
reg31 ,  clk1 ,  reg32 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid9_inst ( reg33 ,  reg34 ,
reg35 ,  clk1 ,  reg36 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid10_inst ( reg37 ,  reg38 ,
reg39 ,  clk1 ,  reg40 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid11_inst ( reg41 ,  reg42 ,
reg43 ,  clk1 ,  reg44 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid12_inst ( reg45 ,  reg46 ,
reg47 ,  clk1 ,  reg48 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid13_inst ( reg49 ,  reg50 ,
reg51 ,  clk1 ,  reg52 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid14_inst ( reg53 ,  reg54 ,
reg55 ,  clk1 ,  reg56 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid15_inst ( reg57 ,  reg58 ,
reg59 ,  clk1 ,  reg60 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid16_inst ( reg61 ,  reg62 ,
reg63 ,  clk1 ,  reg64 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid17_inst ( reg65 ,  reg66 ,
reg67 ,  clk1 ,  reg68 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid18_inst ( reg69 ,  reg70 ,
reg71 ,  clk1 ,  reg72 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid19_inst ( reg73 ,  reg74 ,
reg75 ,  clk1 ,  reg76 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid20_inst ( reg77 ,  reg78 ,
reg79 ,  clk1 ,  reg80 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid21_inst ( reg81 ,  reg82 ,
reg83 ,  clk1 ,  reg84 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid22_inst ( reg85 ,  reg86 ,
reg87 ,  clk1 ,  reg88 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid23_inst ( reg89 ,  reg90 ,
reg91 ,  clk1 ,  reg92 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid24_inst ( reg93 ,  reg94 ,
reg95 ,  clk1 ,  reg96 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid25_inst ( reg97 ,  reg98 ,
reg99 ,  clk1 ,  reg100 )  ;
```

```
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid26_inst ( reg101 ,  reg102 ,
reg103 ,  clk1 ,  reg104 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid27_inst ( reg105 ,  reg106 ,
reg107 ,  clk1 ,  reg108 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid28_inst ( reg109 ,  reg110 ,
reg111 ,  clk1 ,  reg112 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid29_inst ( reg113 ,  reg114 ,
reg115 ,  clk1 ,  reg116 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid30_inst ( reg117 ,  reg118 ,
reg119 ,  clk1 ,  reg120 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid31_inst ( reg121 ,  reg122 ,
reg123 ,  clk1 ,  reg124 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid32_inst ( reg125 ,  reg126 ,
reg127 ,  clk1 ,  reg128 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid33_inst ( reg129 ,  reg130 ,
reg131 ,  clk1 ,  reg132 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid34_inst ( reg133 ,  reg134 ,
reg135 ,  clk1 ,  reg136 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid35_inst ( reg137 ,  reg138 ,
reg139 ,  clk1 ,  reg140 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid36_inst ( reg141 ,  reg142 ,
reg143 ,  clk1 ,  reg144 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid37_inst ( reg145 ,  reg146 ,
reg147 ,  clk1 ,  reg148 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid38_inst ( reg149 ,  reg150 ,
reg151 ,  clk1 ,  reg152 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid39_inst ( reg153 ,  reg154 ,
reg155 ,  clk1 ,  reg156 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid40_inst ( reg157 ,  reg158 ,
reg159 ,  clk1 ,  reg160 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid41_inst ( reg161 ,  reg162 ,
reg163 ,  clk1 ,  reg164 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid42_inst ( reg165 ,  reg166 ,
reg167 ,  clk1 ,  reg168 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid43_inst ( reg169 ,  reg170 ,
reg171 ,  clk1 ,  reg172 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid44_inst ( reg173 ,  reg174 ,
reg175 ,  clk1 ,  reg176 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid45_inst ( reg177 ,  reg178 ,
reg179 ,  clk1 ,  reg180 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid46_inst ( reg181 ,  reg182 ,
reg183 ,  clk1 ,  reg184 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid47_inst ( reg185 ,  reg186 ,
reg187 ,  clk1 ,  reg188 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid48_inst ( reg189 ,  reg190 ,
reg191 ,  clk1 ,  reg192 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid49_inst ( reg193 ,  reg194 ,
reg195 ,  clk1 ,  reg196 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid50_inst ( reg197 ,  reg198 ,
reg199 ,  clk1 ,  reg200 )  ;
```

```
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid51_inst ( reg201 ,  reg202 ,
reg203 ,  clk1 ,  reg204 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid52_inst ( reg205 ,  reg206 ,
reg207 ,  clk1 ,  reg208 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid53_inst ( reg209 ,  reg210 ,
reg211 ,  clk1 ,  reg212 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid54_inst ( reg213 ,  reg214 ,
reg215 ,  clk1 ,  reg216 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid55_inst ( reg217 ,  reg218 ,
reg219 ,  clk1 ,  reg220 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid56_inst ( reg221 ,  reg222 ,
reg223 ,  clk1 ,  reg224 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid57_inst ( reg225 ,  reg226 ,
reg227 ,  clk1 ,  reg228 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid58_inst ( reg229 ,  reg230 ,
reg231 ,  clk1 ,  reg232 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid59_inst ( reg233 ,  reg234 ,
reg235 ,  clk1 ,  reg236 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid60_inst ( reg237 ,  reg238 ,
reg239 ,  clk1 ,  reg240 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid61_inst ( reg241 ,  reg242 ,
reg243 ,  clk1 ,  reg244 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid62_inst ( reg245 ,  reg246 ,
reg247 ,  clk1 ,  reg248 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid63_inst ( reg249 ,  reg250 ,
reg251 ,  clk1 ,  reg252 )  ;
  (* keep_hierarchy = " yes " *)  mid_mod # ( DATA_WIDTH )  mid64_inst ( reg253 ,  reg254 ,
reg255 ,  clk1 ,  reg256 )  ;

 assign out1 =  ( reg4 | reg8 | reg12 | reg16 )  &  ( reg20 | reg24 | reg28 | reg32 )  &  ( reg36 |
reg40 | reg44 | reg48 )  &  ( reg52 | reg56 | reg60 | reg64 )  &  ( reg68 | reg72 | reg76 | reg80
)  &  ( reg84 | reg88 | reg92 | reg96 )  &  ( reg100 | reg104 | reg108 | reg112 )  &  ( reg116 |
reg120 | reg124 | reg128 )  ;
 assign out2 =  ( reg132 | reg136 | reg140 | reg144  )  &  ( reg148 | reg152 | reg156 | reg160 )
&  ( reg164 | reg168 | reg172 | reg176 )  &  ( reg180 | reg184 | reg188 | reg192 )  &  ( reg196
| reg200 | reg204 | reg208 )  &  ( reg212 | reg216 | reg220 | reg224 )  &  ( reg228 | reg232 |
reg236 | reg240  )  &  (  reg244 | reg248 | reg252 | reg256 )   ;

 // MUXes for the ILA
 wire [ DATA_WIDTH - 1 : 0 ] ila_mux_out1 , ila_mux_out2 , ila_mux_out3 , ila_mux_out4 ,
ila_mux_out5 , ila_mux_out6 , ila_mux_out7 , ila_mux_out8 , ila_mux_out9 , ila_mux_out10 ,
ila_mux_out11 , ila_mux_out12 , ila_mux_out13 , ila_mux_out14 , ila_mux_out15 ,
ila_mux_out16 ;
  (* keep_hierarchy = " yes " *)  ila_mux # ( DATA_WIDTH )  ila_mux_inst ( debug_mode ,  reg1
, reg2 , reg3 , reg4 , reg5 , reg6 , reg7 , reg8 , reg9 , reg10 , reg11 , reg12 , reg13 , reg14 , reg15
, reg16 , reg17 , reg18 , reg19 , reg20 , reg21 , reg22 , reg23 , reg24 , reg25 , reg26 , reg27 ,
reg28 , reg29 , reg30 , reg31 , reg32 , reg33 , reg34 , reg35 , reg36 , reg37 , reg38 , reg39 ,
reg40 , reg41 , reg42 , reg43 , reg44 , reg45 , reg46 , reg47 , reg48 , reg49 , reg50 , reg51 ,
reg52 , reg53 , reg54 , reg55 , reg56 , reg57 , reg58 , reg59 , reg60 , reg61 , reg62 , reg63 ,
reg64 , reg65 , reg66 , reg67 , reg68 , reg69 , reg70 , reg71 , reg72 , reg73 , reg74 , reg75 ,
reg76 , reg77 , reg78 , reg79 , reg80 , reg81 , reg82 , reg83 , reg84 , reg85 , reg86 , reg87 ,
reg88 , reg89 , reg90 , reg91 , reg92 , reg93 , reg94 , reg95 , reg96 , reg97 , reg98 , reg99 ,
reg100 , reg101 , reg102 , reg103 , reg104 , reg105 , reg106 , reg107 , reg108 , reg109 , reg110
, reg111 , reg112 , reg113 , reg114 , reg115 , reg116 , reg117 , reg118 , reg119 , reg120 ,
reg121 , reg122 , reg123 , reg124 , reg125 , reg126 , reg127 , reg128 ,  ila_mux_out1 ,
ila_mux_out2 , ila_mux_out3 , ila_mux_out4 , ila_mux_out5 , ila_mux_out6 , ila_mux_out7 ,
ila_mux_out8 , ila_mux_out9 , ila_mux_out10 , ila_mux_out11 , ila_mux_out12 ,
ila_mux_out13 , ila_mux_out14 , ila_mux_out15 , ila_mux_out16 )  ;
  // ILA instance
    ila_0 ila_inst_0  (
      .clk ( clk1 )  ,  // input wire clk
      .probe0 ( ila_mux_out1 )  ,
      .probe1 ( ila_mux_out2 )  ,
      .probe2 ( ila_mux_out3 )  ,
      .probe3 ( ila_mux_out4 )  ,
      .probe4 ( ila_mux_out5 )  ,
      .probe5 ( ila_mux_out6 )  ,
      .probe6 ( ila_mux_out7 )  ,
      .probe7 ( ila_mux_out8 )  ,
      .probe8 ( ila_mux_out9 )  ,
      .probe9 ( ila_mux_out10 )  ,
      .probe10 ( ila_mux_out11 )  ,
      .probe11 ( ila_mux_out12 )  ,
      .probe12 ( ila_mux_out13 )  ,
      .probe13 ( ila_mux_out14 )  ,
      .probe14 ( ila_mux_out15 )  ,
      .probe15 ( ila_mux_out16 )
   )  ;
endmodule
module ila_mux ( mode ,  in1 , in2 , in3 , in4 , in5 , in6 , in7 , in8 , in9 , in10 , in11 , in12 , in13 ,
in14 , in15 , in16 , in17 , in18 , in19 , in20 , in21 , in22 , in23 , in24 , in25 , in26 , in27 , in28 ,
in29 , in30 , in31 , in32 , in33 , in34 , in35 , in36 , in37 , in38 , in39 , in40 , in41 , in42 , in43 ,
in44 , in45 , in46 , in47 , in48 , in49 , in50 , in51 , in52 , in53 , in54 , in55 , in56 , in57 , in58 ,
in59 , in60 , in61 , in62 , in63 , in64 , in65 , in66 , in67 , in68 , in69 , in70 , in71 , in72 , in73 ,
in74 , in75 , in76 , in77 , in78 , in79 , in80 , in81 , in82 , in83 , in84 , in85 , in86 , in87 , in88 ,
in89 , in90 , in91 , in92 , in93 , in94 , in95 , in96 , in97 , in98 , in99 , in100 , in101 , in102 , in103
, in104 , in105 , in106 , in107 , in108 , in109 , in110 , in111 , in112 , in113 , in114 , in115 ,
in116 , in117 , in118 , in119 , in120 , in121 , in122 , in123 , in124 , in125 , in126 , in127 , in128
,  out1 , out2 , out3 , out4 , out5 , out6 , out7 , out8 , out9 , out10 , out11 , out12 , out13 ,
out14 , out15 , out16 )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/O ports
  input wire [2:0 ] mode ;
  input [ DATA_WIDTH - 1 : 0 ] in1 , in2 , in3 , in4 , in5 , in6 , in7 , in8 , in9 , in10 , in11 , in12 ,
in13 , in14 , in15 , in16 , in17 , in18 , in19 , in20 , in21 , in22 , in23 , in24 , in25 , in26 , in27 ,
in28 , in29 , in30 , in31 , in32 , in33 , in34 , in35 , in36 , in37 , in38 , in39 , in40 , in41 , in42 ,
in43 , in44 , in45 , in46 , in47 , in48 , in49 , in50 , in51 , in52 , in53 , in54 , in55 , in56 , in57 ,
in58 , in59 , in60 , in61 , in62 , in63 , in64 , in65 , in66 , in67 , in68 , in69 , in70 , in71 , in72 ,
in73 , in74 , in75 , in76 , in77 , in78 , in79 , in80 , in81 , in82 , in83 , in84 , in85 , in86 , in87 ,
in88 , in89 , in90 , in91 , in92 , in93 , in94 , in95 , in96 , in97 , in98 , in99 , in100 , in101 , in102 ,
in103 , in104 , in105 , in106 , in107 , in108 , in109 , in110 , in111 , in112 , in113 , in114 , in115
```

```
, in116 , in117 , in118 , in119 , in120 , in121 , in122 , in123 , in124 , in125 , in126 , in127 ,
in128 ;
 output wire [ DATA_WIDTH - 1 : 0 ] out1 , out2 , out3 , out4 , out5 , out6 , out7 , out8 , out9 ,
out10 , out11 , out12 , out13 , out14 , out15 , out16 ;
 // Logic
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst1 ( mode , in1 ,
in2 ,  in3 ,  in4 ,  in5 ,  in6 ,  in7 ,  in8 ,  out1 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst2 ( mode , in9 ,
in10 ,  in11 ,  in12 ,  in13 ,  in14 ,  in15 ,  in16 ,  out2 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst3 ( mode , in17
,  in18 ,  in19 ,  in20 ,  in21 ,  in22 ,  in23 ,  in24 ,  out3 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst4 ( mode , in25
,  in26 ,  in27 ,  in28 ,  in29 ,  in30 ,  in31 ,  in32 ,  out4 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst5 ( mode , in33
,  in34 ,  in35 ,  in36 ,  in37 ,  in38 ,  in39 ,  in40 ,  out5 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst6 ( mode , in41
,  in42 ,  in43 ,  in44 ,  in45 ,  in46 ,  in47 ,  in48 ,  out6 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst7 ( mode , in49
,  in50 ,  in51 ,  in52 ,  in53 ,  in54 ,  in55 ,  in56 ,  out7 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst8 ( mode , in57
,  in58 ,  in59 ,  in60 ,  in61 ,  in62 ,  in63 ,  in64 ,  out8 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst9 ( mode , in65
,  in66 ,  in67 ,  in68 ,  in69 ,  in70 ,  in71 ,  in72 ,  out9 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst10 ( mode ,
in73 ,  in74 ,  in75 ,  in76 ,  in77 ,  in78 ,  in79 ,  in80 ,  out10 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst11 ( mode ,
in81 ,  in82 ,  in83 ,  in84 ,  in85 ,  in86 ,  in87 ,  in88 ,  out11 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst12 ( mode ,
in89 ,  in90 ,  in91 ,  in92 ,  in93 ,  in94 ,  in95 ,  in96 ,  out12 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst13 ( mode ,
in97 ,  in98 ,  in99 ,  in100 ,  in101 ,  in102 ,  in103 ,  in104 ,  out13 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst14 ( mode ,
in105 ,  in106 ,  in107 ,  in108 ,  in109 ,  in110 ,  in111 ,  in112 ,  out14 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst15 ( mode ,
in113 ,  in114 ,  in115 ,  in116 ,  in117 ,  in118 ,  in119 ,  in120 ,  out15 )  ;
  (* keep_hierarchy = " yes " *)  mux8_mod # ( DATA_WIDTH )  mux8_mod_inst16 ( mode ,
in121 ,  in122 ,  in123 ,  in124 ,  in125 ,  in126 ,  in127 ,  in128 ,  out16 )  ; endmodule
module mux8_mod  ( mode ,  in1 ,  in2 ,  in3 ,  in4 ,  in5 ,  in6 ,  in7 ,  in8 ,  out1 ,  out2 )  ;
 // Parameters
 parameter DATA_WIDTH = 1 ;
 // I/O ports
 input [2:0 ] mode ;
 input [ DATA_WIDTH - 1 : 0 ] in1 ,  in2 ,  in3 ,  in4 ,  in5 ,  in6 ,  in7 ,  in8 ;
 output reg [ DATA_WIDTH - 1 : 0 ] out1 ,  out2 ;
 // Logic
 always @ ( in1 ,  in2 ,  in3 ,  in4 ,  in5 ,  in6 ,  in7 ,  in8 ,  mode )  begin
   case  ( mode )
     3'b000  : out1 <= in1 ;
     3'b001  : out1 <= in2 ;
     3'b010  : out1 <= in3 ;
     3'b011  : out1 <= in4 ;
```

```
    3'b100  : out1 <= in5 ;
    3'b101  : out1 <= in6 ;
    3'b110  : out1 <= in7 ;
    default : out1 <= in8 ;
   endcase
  end
endmodule
module mid_mod  ( in1 ,  in2 ,  in3 ,  clk1 ,  out1 )  ;
  // Parameters
  parameter DATA_WIDTH = 1 ;
  // I/Os
  input wire [ DATA_WIDTH - 1 : 0 ] in1 ,  in2 ,  in3 ;
  input wire clk1 ;
  output wire [ DATA_WIDTH - 1 : 0 ] out1 ;
  // Logic
  genvar i ;
  for  ( i=0 ; i<DATA_WIDTH ; i=i+1 )  begin: LEAF_GEN
   leaf_mod leaf_inst ( in1[i] ,  in2[i] ,  in3[i] ,  out1[i] )  ;
  end
endmodule

module leaf_mod  ( in1 ,  in2 ,  clk1 ,  out1 )  ;
  // I/Os
  input wire in1 ,  in2 ,  clk1 ;
  output reg out1 ;
  // Internal signals
   (* keep = " true " *)  reg r1 ,  r2 ,  r3 ;
  // Logic
  always @ ( posedge clk1 )  begin
   r1 <= in1 ;
   r2 <= in2 ;
   r3 <= r1 | r2 ;
   out1 <= r3 ;
  end
endmodule
```

الملخص

يسمح إعادة التشكيل الجزئى الديناميكى (DPR) على مصفوفات البوابات المنطقية القابلة للبرمجة (FPGA) بإعادة تشكيل بعض التصميم في وقت التشغيل بينما يستمر باقي التصميم في العمل. هذه الفئة من التصاميم تسمى التصاميم القابلة لاعادة التكوين (DRS). هذه الميزة تسمح للمصممين لبناء أنظمة معقدة مثل نظام الراديو المعرف برمجيا (SDR) في مساحة مناسبة. على الرغم من المرونة التي توفرها إعادة التشكيل الجزئى الديناميكى (DPR) ، هناك تحديات جديدة لتصميم والتحقق من التصاميم التي تستخدم أسلوب التشكيل الجزئى الديناميكى (DPR) مقارنة مع الانظمة الثابتة.

في هذه الرسالة ، يتم تقديم منهجية تحقق جديدة لاعادة التشكيل الجزئى الديناميكى (DPR). تتعامل المنهجية الجديدة مع المنطق المحدد لاعادة التشكيل الجزئى الديناميكى (DPR) و حالات مثل ضمان التوصيلات الصحيحة لمنافذ الوحدات القابلة لإعادة التهيئة (RMs) التي تشترك في نفس المنطقة القابلة لاعادة البرمجة (RR) على مصفوفات البوابات المنطقية القابلة للبرمجة (FPGA)، انتظار إجراء العمليات الحسابية على وحدة نمطية قبل إعادة تشكيلها، عزل الوحدات القابلة لإعادة التكوين أثناء عملية إعادة التكوين، وتهيئة الوحدة القابلة لإعادة التكوين بعد اتمام عملية إعادة التشكيل. يتم التحقق من منطق التشكيل الجزئى الديناميكى (DPR) باستخدام التحقق القائم على التوكيد (ABV) من خلال تحديد وظائفه باستخدام خصائص تأكيد (SVA)، ثم صك التصميم مع هذه الخصائص ، ثم يمكن التحقق من هذه الخصائص باستخدام المحاكاة أو الطرق الرسمية لإثبات صحة أو عدم صحة منطق التشكيل الجزئى الديناميكى (DPR). كما يقدم هذا البحث تقديراً آليا يقترن بالتحقق من سجلات احتياز مجال الساعات (CDC) للتصاميم القابلة لاعادة التشكيل (DRS).

توضح هذه الرسالة أيضًا قوة استخدام تقنية التشكيل الجزئى الديناميكى (DPR) لتقليل تكلفة تصميم التطبيقات التي تؤدي تغير المنطق الرقمي على مدا الوقت مثل تصحيح أخطاء على مصفوفات البوابات المنطقية القابلة للبرمجة (FPGA). يعد تصحيح الأخطاء على مصفوفات البوابات المنطقية القابلة للبرمجة (FPGA) مهمة صعبة بسبب الوصول المحدود إلى الإشارات الداخلية للتصميم. يقوم محلل المنطق المدمج بتحسين إمكانية رصد إشارات على مصفوفات البوابات المنطقية القابلة للبرمجة (FPGA). يتم تنفيذ هذه المحلل على موارد على مصفوفات البوابات المنطقية القابلة للبرمجة (FPGA) ، وتستخدم كتل الذاكرة المضمنة كمخازن تتبع ، لذلك يمكن ملاحظة عدد محدود من الإشارات باستخدام هذه المحلل بسبب قلة الموارد. يتطلب تغيير مجموعة الإشارات المتتبعة إعادة تركيب كل التصميم وتصميمه وتوجيهه. نقترح في هذه الرسالة منهجية جديدة لتصحيح الأخطاء مصفوفات البوابات المنطقية القابلة للبرمجة (FPGA) لتغيير مجموعة الإشارات التي يجب مراعاتها في وقت التشغيل بشكل ديناميكي ، وبالتالي تقليل الوقت اللازم للتصحيح. تستخدم المنهجية المقترحة تقنية إعادة التشكيل الجزئى الديناميكى (DPR) للتبديل ديناميكياً بين مجموعات مختلفة من الإشارات. ينشئ إعادة التشكيل الجزئى الديناميكى (DPR) وحدة قابلة لإعادة التكوين (RM) لتوجيه كل مجموعة من الإشارات إلى محلل منطقي مضمن. تم توضيح النهج المقترح باستخدام أدوات Xilinx FPGA ، حيث وجد أن تغيير مجموعة الإشارات المراد مراقبتها يتطلب بضع ثوان فقط لإعادة برمجة المنطقة القابلة لإعادة التكوين. إن المساحة الزائدة للمنهجية المقترحة أقل من الطرق التقليدية الأخرى لأن DPR تسمح لوحدة التوجيه أن تستخدم فقط المخازن المؤقتة لتوصيل مجموعة من الإشارات إلى محلل المنطق المدمج.

مهندس:	اسلام اسامة احمد منير مصطفى
تاريخ الميلاد:	1990\3\10
الجنسية:	مصرى
تاريخ التسجيل:	2013\10\1
تاريخ المنح:	
القسم:	هندسة الإلكترونيات والإتصالات الكهربية
الدرجة:	ماجستير العلوم
المشرفون:	
	ا.د. أحمد نادر محي الدين
	د. حسن مصطفى حسن

الممتحنون:

أ.د.................... (الممتحن الخارجي)

أ.د.................... (الممتحن الداخلي)

أ.د.................... (المشرف الرئيسي)

أ.د. .... (عضو)

عنوان الرسالة:

التحقق من إعادة التشكيل الجزئي الديناميكى و تنفيذه للتصحيح على مصفوفات البوابات المنطقية القابلة للبرمجة

الكلمات الدالة:

إعادة التشكيل الجزئي الديناميكى،التحقق، التصحيح، نظام الراديو المعرف برمجيا، مصفوفات البوابات المنطقية القابلة للبرمجة، النظم متغيرة التشَكل.

ملخص الرسالة: (7 اسطر)

....

....

....

....

....

....

....

التحقق من إعادة التشكيل الجزئي الديناميكى و تنفيذه للتصحيح على مصفوفات البوابات المنطقية القابلة للبرمجة

اعداد
اسلام اسامة احمد منير مصطفى

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستيرالعلوم
في
هندسة الإلكترونيات والإتصالات الكهربية

يعتمد من لجنة الممتحنين:

الاستاذ الدكتور:	الممتحن الخارجي
الاستاذ الدكتور:	الممتحن الداخلي
الاستاذ الدكتور:	المشرف الرئيسى
الاستاذ الدكتور:	عضو

كليـــة الهندســـة - جامعـــة القاهـــرة
الجيـزة - جمهوريـة مصـر العربيـــة
2018

التحقق من إعادة التشكيل الجزئي الديناميكى و تنفيذه للتصحيح على مصفوفات البوابات المنطقية القابلة للبرمجة

اعداد
اسلام اسامة احمد منير مصطفى

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستيرالعلوم
في
هندسة الالكترونيات والاتصالات الكهربية

تحت اشراف

أحمد نادر محي الدين	حسن مصطفى حسن
أستاذ مساعد دكتور بقسم	مدرس دكتور بقسم
الإلكترونيات والإتصالات الكهربية	الإلكترونيات والإتصالات الكهربية
كلية الهندسة ـــ جامعة القاهرة	كلية الهندسة ـــ جامعة القاهرة

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة
2018

التحقق من إعادة التشكيل الجزئي الديناميكى و تنفيذه للتصحيح على مصفوفات البوابات المنطقية القابلة للبرمجة

اعداد

اسلام اسامة احمد منير مصطفى

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستير العلوم
في
هندسة الالكترونيات والاتصالات الكهربية

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة
2018