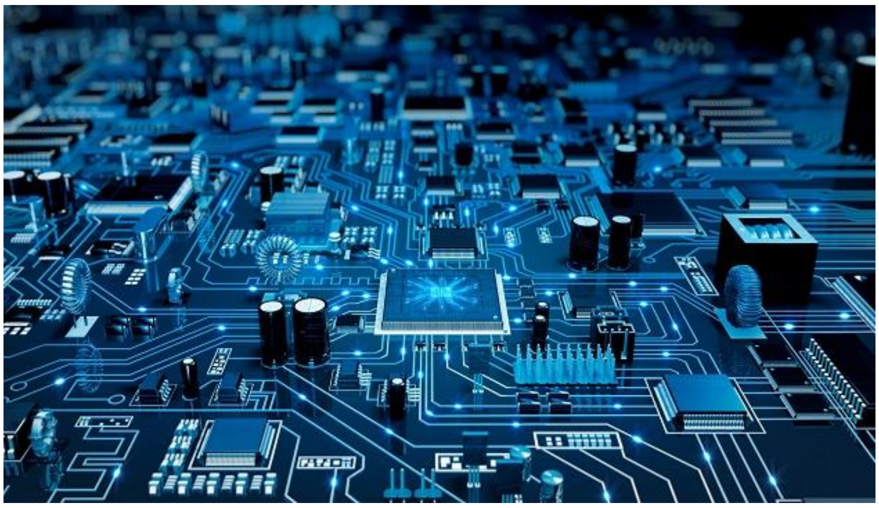
Youssef BENDOU

MNIS master
2020

Mercury Mission Systems International, Lancy, Geneva

FPGA Dynamic Function eXchange

from 02/03/2020 to 28/08/2020



Confidentiality : no

Under the supervision of:

- **Christian RUPPERT, christian.ruppert@ch.mrcy.com**

Present at the defense : yes

- **Lorena ANGHEL, lorena.anghel@grenoble-inp.fr**

Acknowledgement

I would like to express my special thanks and gratitude to some people who, by their experience and guidance, made my internship a great learning experience in my journey to become an engineer:

- Mr Christian Ruppert, Manager at MMSI, my tutor, for his very valuable help and guidance through this project.
- Mr Pierrick Hascoet, Software engineer at MMSI, for his help with the software part of this project.
- Mrs Winnie Wong, FPGA project manager at MMSI, for her valuable advice on how to approach my work.
- Special thanks also to the rest of the FPGA team and the MMSI employees who could always make time to answer my questions and show me the way when needed.

Table of Contents

Figures

Tables

Glossary

AXI	Advanced eXtensible Interface
CPU	Central Processing Unit
ECC	Error Correcting Code
FPGA	Field Programmable Gate Array
DFX	Dynamic Function eXchange
ID	Identifier
MMSI	Mercury Mission System Intl
PCIe	Peripheral Component Interconnect Express
PLL	Phase Locked Loop
VHDL	Very High Speed Integrated Circuit Hardware Description Language
PRC	Partial Reconfiguration Controller
SEM	Single Error Mitigation
APB	Advanced Peripheral Bus
ICAP	Internal Configuration Access Port
DRC	Design Rule Checks
OOC	Out of Context
SSI	Stacked Silicon Interconnect
SLR	Super Logic Region
MSB	Most Significant Bit
LSB	Least significant Bit
TCL	Tool Command Language
GUI	Graphic User Interface
RTL	Register Transfer Level
C++	Programming language
ASIC	Application Specific Integrated Circuit
IP	Intellectual Property
VSM	Virtual Socket Manager
HDL	Hardware Description Language
RP	Reconfigurable Partition
RM	Reconfigurable Module
IEEE	Institute of Electrical and Electronics Engineers

Abstract

1.1. English

FPGAs are electrical circuits that can be reprogrammed in-field to suit the user’s needs and implement whatever digital functionality wanted. This programmability feature is however constrained by the fact that the design implemented in it needs to be shutdown each time before. DFX is an upgrade of this feature that enables the user to do it dynamically and on specific partitions of the FPGA. This opens the door to many applications and enables reductions of size and cost to implement a design in an FPGA.

1.2. French

Les FPGA sont des circuits électriques qui peuvent être reprogrammés pour répondre aux besoins de l’utilisateur et mettre en œuvre toutes les fonctionnalités numériques souhaitées. Cette fonction de programmabilité est cependant limitée par le fait que le design qui y est implémentée doit être arrêté à chaque fois auparavant. DFX est une mise à niveau de cette fonctionnalité qui permet à l'utilisateur de le faire de manière dynamique et sur des partitions spécifiques du FPGA. Cela permets de nombreuses applications et facilite la réduction de la taille et le coût de mise en œuvre d'une conception dans un FPGA.

1.3. Italian

Gli FPGA sono circuiti elettrici che possono essere riprogrammati sul campo per soddisfare le esigenze dell'utente e implementare qualsiasi funzionalità digitale desiderata. Questa caratteristica di programmabilità è tuttavia limitata dal fatto che il progetto in esso implementato deve essere chiuso ogni volta prima. DFX è un aggiornamento di questa funzione che consente all'utente di farlo dinamicamente e su partizioni specifiche dell'FPGA. Questo apre la porta a molte applicazioni e consente riduzioni di dimensioni e costi per implementare un progetto in un FPGA.

Introduction

FPGA is an acronym for Field Programmable Gate Array and it is a device that contains a significant number of transistors and other electrical components in a way that it can be programmed to implement whatever digital functionality is needed by the user. This reprogramming feature makes these FPGA devices very appealing but the constraint is that every time the circuit needs to be reprogrammed, the whole design implemented in it has to be shut down and reset to implement the new functionality. The goal of this internship project is to examine the possibility of doing this in a partial and dynamic way, which means reprogramming only some partitions of the FPGA on the fly without disturbing the other partitions in their jobs.

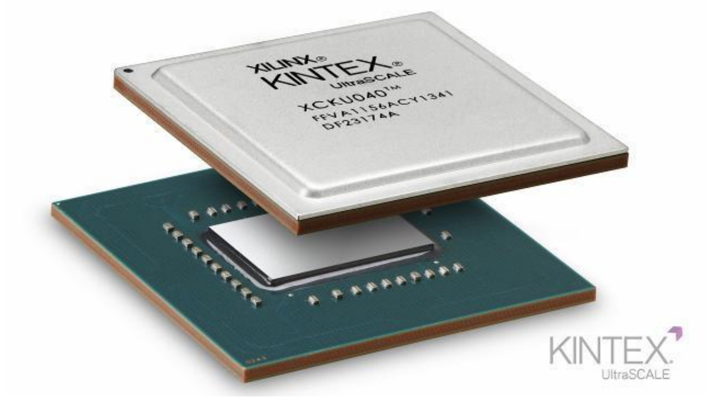


Figure 1. Xilinx Kintex UltraScale FPGA

Mercury Mission Systems International is a company involved in the market of defense and aerospace electronics. I was placed as an intern in the FPGA team, which is a team whose mission is to deal with the design aspect of projects. During this time, I had the chance to work on a complete design by myself from writing RTL code to bitstream generation for the programming of the FPGA and verifying the feasibility of this dynamic partial reconfiguration feature. I also did some software programming using the C++ programming language. Spending 6 months within this team has given me the chance to work under the supervision of experienced designers and managers, it was a great opportunity to learn and to get a grasp of a design engineer's work environment.

This report contains a first section about the company MMSI and its industrial context, then three sections about my work within the company to furthermore describe the assignments I had and the results I achieved.

The goal of this internship project was to make a DFX example design with a single event error mitigation feature from scratch, generate corresponding files that will serve to program the FPGA with the design wanted. Make a small C++ software that will program a CPU to do some read and write commands that will control the different blocks in the FPGA to perform a DFX operation. The aim of all of this is to prove DFX technology's feasibility to a client of Mercury Systems. I also had to write some documentation for the client to explain my work.

2. Mercury Mission Systems International[MMSI]

Mercury Mission Systems International (MMSI) is based in Geneva, Switzerland, and is specialized in the design, manufacturing and maintenance follow-up of computers, more specifically complex safety avionics and defense computers. It is part of Mercury Systems, an American leading commercial provider of secure sensors and mission processing subsystems.

2.1. Facts at a Glance

- **Mercury Systems was founded in 1981.**
- **Mercury's solutions power a wide variety of critical defense and intelligence programs**
- **Mercury Systems is based in Andover, Massachusetts**
- **Mercury Mission Systems International is based in Geneva, Switzerland**
- **It counts approximately 1900 employees worldwide**
- **Fiscal year 2020 revenue: $796.6M**

2.2. Markets

Defense and commercial electronics:

- **Radars**
- **Electronic warfare and signal intelligence**
- **Command, Control, Communications, Computers, Intelligence, Surveillance, Reconnaissance(C4ISR)**
- **Sonar**
- **Missiles and munitions**
- **Mission computing & avionics**

3. FPGA Dynamic Function eXchange

FPGAs provide the ability to program and reprogram a circuit in-field to suit the user's needs. Dynamic function exchange takes this feature one step further, by enabling the user to do it partially if needed, and on the fly.

3.1. FPGA reprogramming

Field Programmable Gate Arrays contain a huge number of electrical components and routing resources that enables it to have this reprogrammable feature. To reprogram it, the most important thing is a file, called the bit file or bitstream, which contains a series of 0 and 1 binary elements that, once sent to the internal configuration memory of the FPGA, will set the functionality of the different blocks inside.

To generate this bit file, there is a whole flow to follow:

- **RTL writing**

This is done using hardware description languages (HDL), which are coding languages that will serve to describe the functionality of the circuit to be programmed into the FPGA. This internship project was coded using VHDL, a hardware description language standardized by the IEEE.

- **RTL elaboration**

Tools exist to do RTL elaboration, which is basically reading and understanding the HDL coding and translating it into a circuit with blocks but with no optimization, just a complete translation of the code into a schematic for the circuit. For this internship project, Vivado Design Suite from Xilinx was used all along for all of the flow.

- **Synthesis**

The synthesizer will then use advanced and complex algorithms to optimize the circuit elaborated and look for shortcuts that will simplify the circuit but keep the same functionality. A netlist will be generated which is a list of all the resources and connections needed to implement this design.

- **Implementation**

The implementation phase is where the tool will try to virtually place the different blocks of the design in the FPGA and assign which resources of the FPGA and which routing paths will be used to implement the design. There are also some optimization algorithms used in this step. After this, all of the placements and routings needed to make the design work are well known and chosen.

- **Bitstream generation**

The bitstream generation phase is where the tool will translate the implementation results into the famous file, the bit file, that contains information on which resources will be used and which connection routes will be chosen to implement the design inside the FPGA. This bit file is then loaded in the FPGA to start its job.

3.2. Dynamic Function eXchange (DFX)

DFX is the ability to reprogram partitions of an FPGA dynamically. After a full bit file configures the FPGA and gets it up and running, partial bit files need to be loaded to change the functionality of specific blocks without compromising other blocks that are outside the scope of this dynamic programmability.

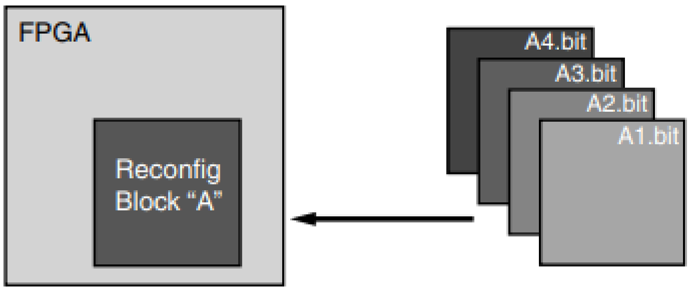


Figure 2. Dynamic function exchange principle [DFXUG]

The gray area represents static logic implemented in the FPGA, and the black area labeled Reconfig Block "A" represents reconfigurable logic that can be replaced with different partial bit files depending on the need. The black area is called a reconfigurable partition (RP), the partial bit files are called reconfigurable modules (RM).

It is a very practical feature since it enables the reduction of the size of an FPGA required to implement a function, leading to consequent reductions in cost and power consumption of a circuit. It enables some flexibility in the choice of algorithms and functions needed and it is also an efficient way to deliver updates to deployed systems.

3.3. Applications

- **Networked multiport interface**

The ports at the client's side of this interface can support many protocols for interfacing but it is impossible to predict which protocol is needed so the designer is forced to have all the possible port interfaces and multiplex the inputs and outputs to be sure that all the possible protocols are treated.

Using DFX technology, port interfaces can be made as reconfigurable modules and interchanged every time depending on the type of protocol used. This is considerably better in terms of size since the designer doesn't need to implement all the different possible port interfaces in the FPGA, and the multiplexing elements would no longer be needed.

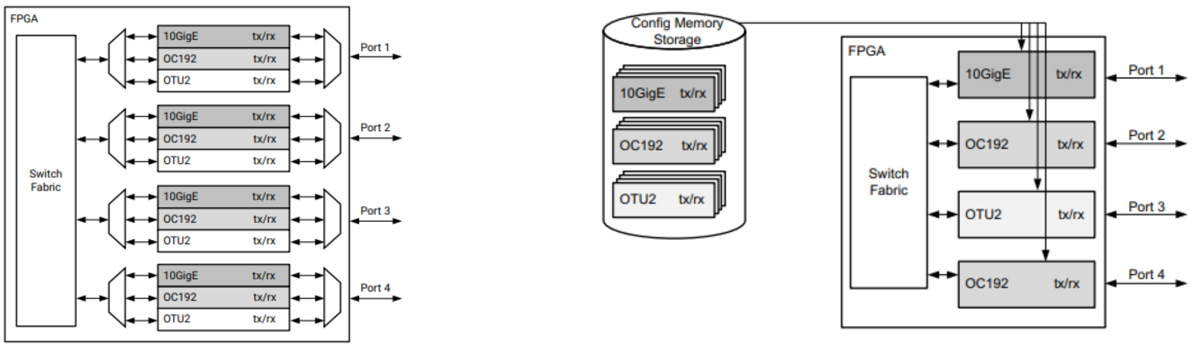


Figure 3. Networked multiport interface without and with partial reconfiguration [DFXUG]

- **Dynamically reconfigurable packet processor**

It is possible for a packet processor to change its processing functionalities.

Packets that the packet processor receives have headers, these headers could contain partial bit files that will be used to dynamically reconfigure a co-processor, thus changing the processing functionalities.

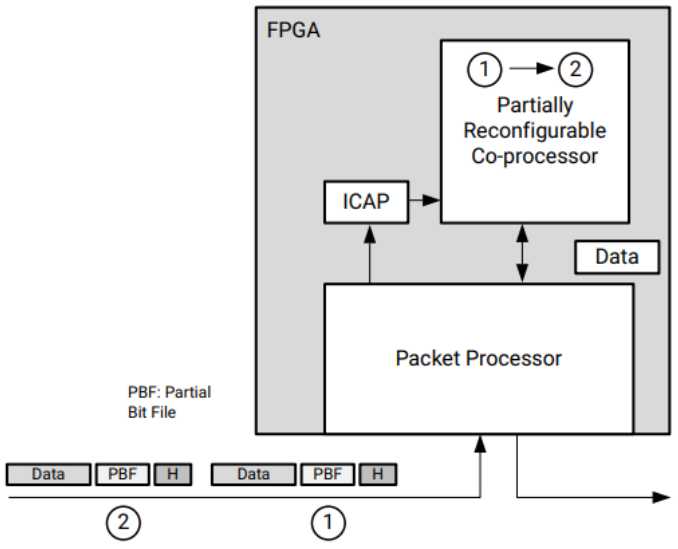


Figure 4. Packet processor [DFXUG]

4. DFX example design

The example design that is targeted will include two controllers that are Xilinx IPs, a controller that manages the dynamic reconfiguration and a controller that is almost constantly scanning the internal configuration of the FPGA to look for errors. Both these controllers are the heart of this design and both of them need access to the internal configuration of the FPGA to do their respective jobs. An internal configuration access port (ICAP) is present in the design for this purpose.

A PCIe connection ensures communication between the controllers and an external CPU. This CPU will be programmed to send read and write requests on some internal registers of the two controllers, through this PCIe connection, to command them into doing the job needed by the user.

The reconfigurable partition is a counter that will be loaded by either a count up or a count down function. The rest of the blocks are a mix of IPs and RTL I coded myself to help the main modules communicate with each other. This includes an AXI interconnect, an AXI to APB bridge, an APB bus interface, an arbiter and a DFX decoupler.

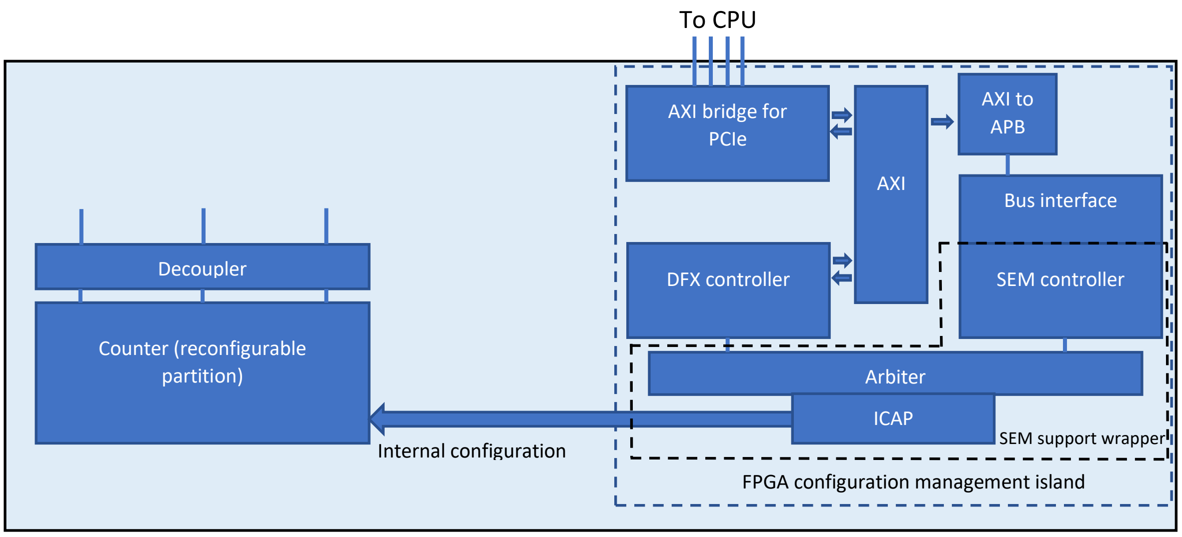


Figure 5. DFX design block diagram

4.1. Environment

The work environment is an important part of this design since there are differences that need to be taken into account depending on the type of FPGA used.

The software used for all of the flow is the Vivado Design Suite 2019.1 from Xilinx. It is a software that includes all necessary tools to make an FPGA design from RTL writing to bitstream uploading.

The FPGA used is a Kintex UltraScale FPGA, a Xilinx product that comes with the board KCU105.

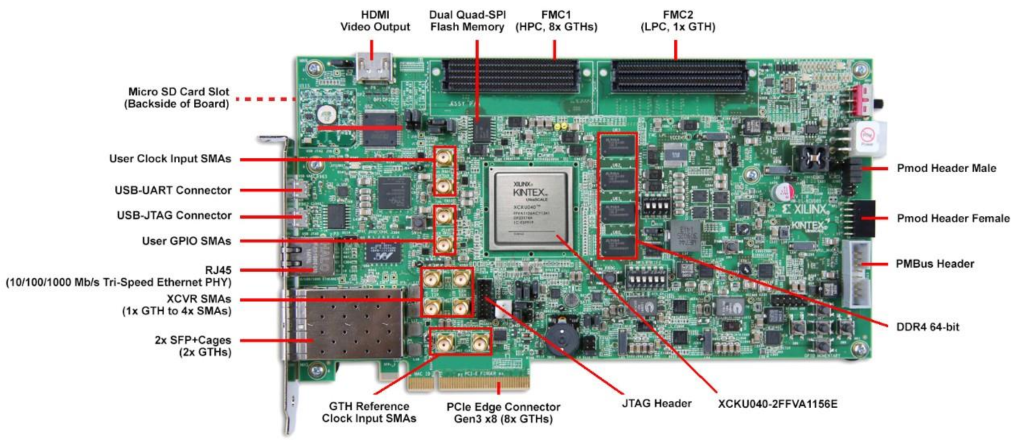


Figure 6. KCU105 board [KCU105EK]

4.2. Clock distribution

Two external clocks are used for this design, the first one is a 100 MHz PCIe differential reference clock that is exclusively generated on the KCU105 board by the PCIe edge connector for the PCIe connection in the FPGA, and the second one is a 300 MHz system clock, also generated on the board, to clock the rest of the design. The AXI_clk is a clock generated by the AXI bridge for PCIe IP exclusively for AXI interfaces that are connected to it.

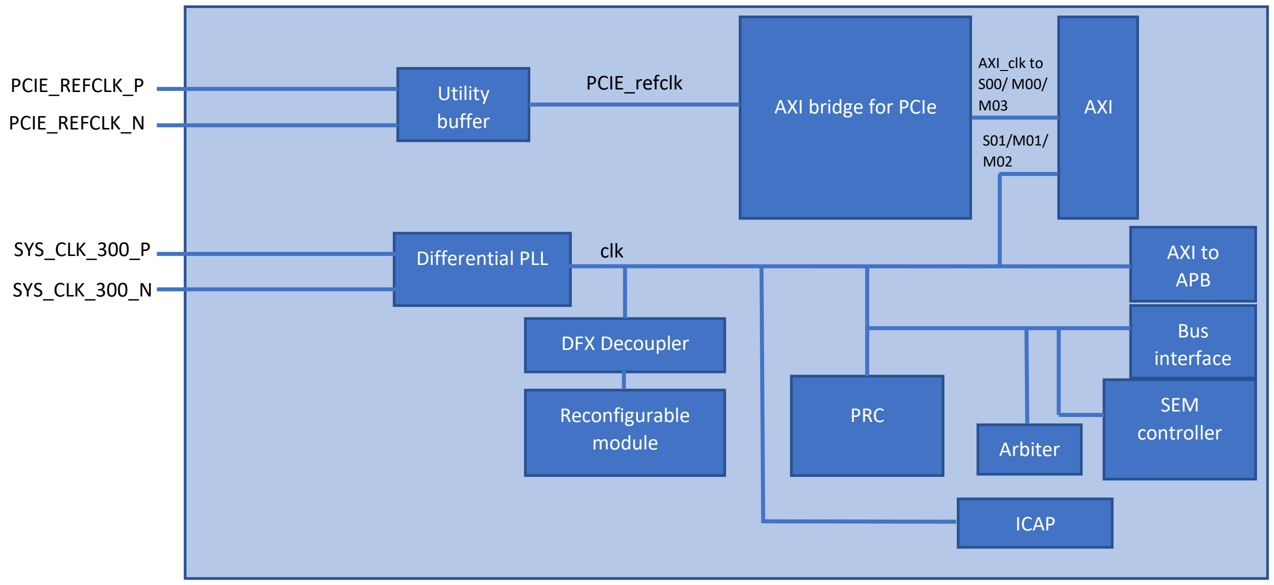


Figure 7. Clock distribution diagram

4.3. Reset distribution

An external reset generated on the board by the PCIe edge connector is used to reset the AXI bridge for PCIe IP. An axi_aresetn is generated by the same IP to reset the AXI interfaces that are connected to it. The rest of the blocks use a reset that is generated locally in the FPGA.

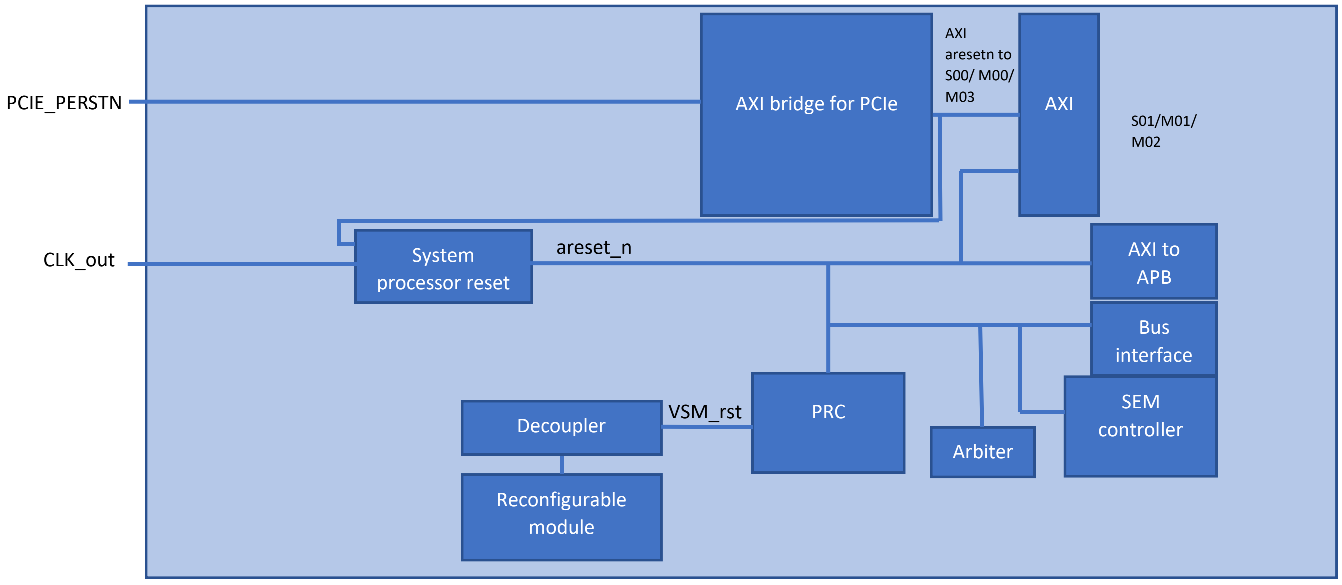


Figure 8. Reset distribution diagram

4.4. DFX controller IP

4.4.1. Role

The purpose of this Xilinx IP is to handle DFX operations in the most efficient way. It has a slave interface that is used to access its internal configuration registers, a master interface that is responsible for the fetching of the partial bitstreams from an external CPU memory through the PCIe connection, and an ICAP interface which contains the signals that will help to send the fetched partial bitstreams into the internal configuration of the FPGA to target the reconfigurable partition and change its functionality. It is mainly made of virtual socket managers (VSM), each one of these is managing one reconfigurable partition which can have many reconfigurable modules. In this design there is only one partition and two modules so only one VSM is needed.

4.4.2. Interface signals

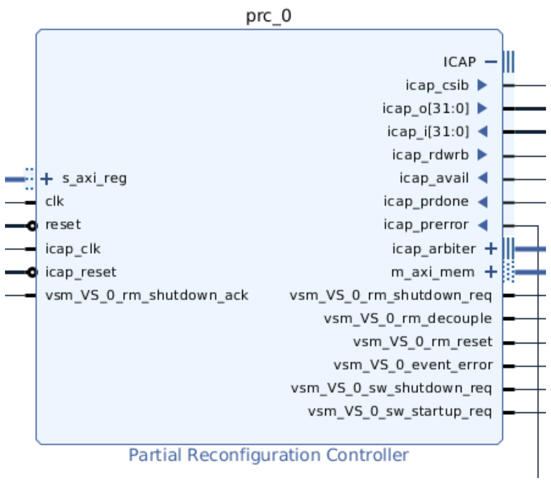


Figure 9. DFX controller signals[DFXCPG]

This table describes the interface signals of this IP. Signals that are present in the diagram but not in the table are not used for this design and can be put to a constant value if it is an input or left open if it is an output.

Table 1. DFX controller input/output ports [DFXCPG]

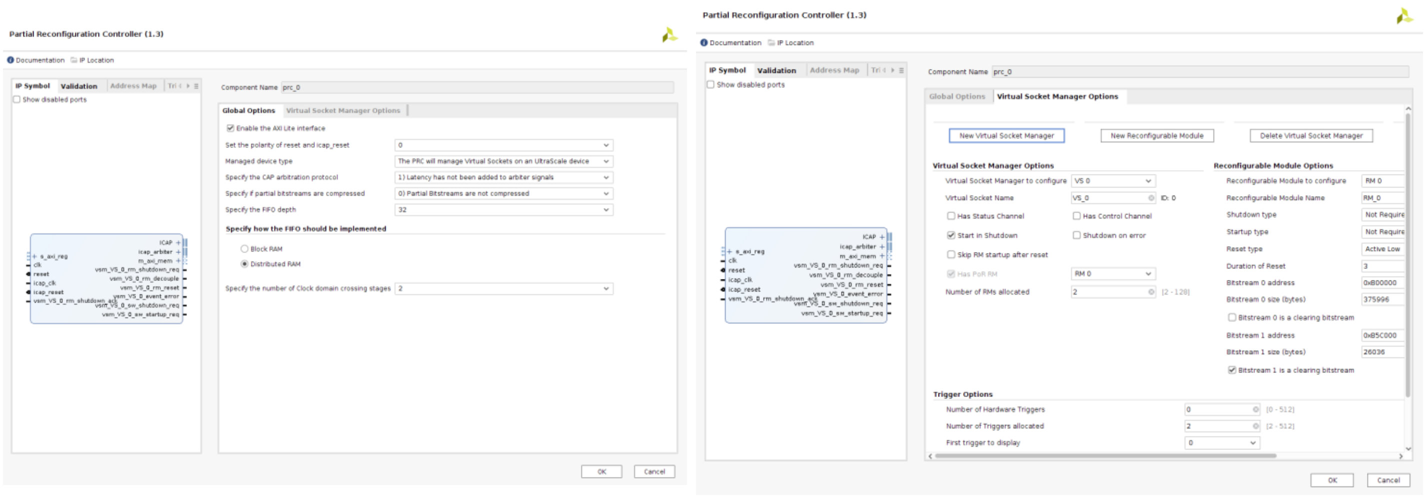
Port	Direction	Description
clk	Input	Master clock for the controller.
icap_clk	Input	Must be the same clock that is attached to the ICAP primitive
reset	Input	Reset signal for the controller
s_axi_reg	Interface	AXI slave interface of the controller. This is the interface that permits access to the internal configuration registers of this controller.
icap_reset	Input	Synchronous reset signal used to reset the ICAP interface logic. Needs to be synchronous to icap_clk.
vsm_VS_0_rm_shutdown_ack	Input	Handshake signals with the reconfigurable logic.
vsm_VS_0_rm_shutdown_req	Output	
ICAP	Interface	ICAP interface that contains all the signals that need to be connected to the ICAP primitive.
icap_arbiter	Interface	Arbiter interface that contains the signals needed to arbitrate the access of this controller to the ICAP primitive.
m_axi_mem	Interface	Master interface that is connected to the slave interface of the PCIe bridge, passing through the AXI interconnect first.
vsm_VS_0_rm_decouple	Output	Signal asserted by the controller when a decoupling operation is needed prior to executing DFX.
vsm_VS_0_rm_reset	Output	Reset signal intended to reset the reconfigurable logic.

4.4.1. Detailed description

This IP is the core of this design, since it is the one responsible for all DFX operations and their management. The better this controller is configured in a way that is adapted to the design, the better the DFX operations work. Since the controller is charged with replacing partial bit files that are already implemented in the FPGA, with new ones that it has to fetch, it must know the sizes of these partial bit files, and the addresses at which it can fetch them from the CPU external memory.

It has a master interface that is responsible for the fetching operations of the partial bit files, this master interface is connected to a slave interface in the PCIe bridge through the AXI interconnect. It also has a slave interface that enables the access to different control registers and bitstream information registers. These two interfaces are the most important ones for DFX operations.

The IP can be instantiated directly using the Vivado IP catalog and can be configured to function properly:


Partial Reconfiguration Controller (1.3)

Component Name: prc_0

Global Options

- [✓] Enable the AXI Lite interface
- Set the polarity of reset and icap_reset: 0
- Managed device type: The PRC will manage Virtual Sockets on an UltraScale device
- Specify the CAP arbitration protocol: 1) Latency has not been added to arbiter signals
- Specify if partial bitstreams are compressed: 0) Partial Bitstreams are not compressed
- Specify the FIFO depth: 32

Specify how the FIFO should be implemented: Block RAM [✓ Distributed RAM]

- Specify the number of Clock domain crossing stages: 2

Virtual Socket Manager Options

- Virtual Socket Manager to configure: VS 0
- Virtual Socket Name: VS_0 ID: 0
- [ ] Has Status Channel
- [ ] Has Control Channel
- [✓] Start in Shutdown
- [ ] Shutdown on error
- [ ] Skip RM startup after reset
- [✓] Has PoR RM: RM 0
- Number of RMs allocated: 2 [2 - 128]

Reconfigurable Module Options

- Reconfigurable Module to configure: RM 0
- Reconfigurable Module Name: RM_0
- Shutdown type: Not Require
- Startup type: Not Require
- Reset type: Active Low
- Duration of Reset: 3
- Bitstream 0 address: 0x800000
- Bitstream 0 size (bytes): 375996
- [ ] Bitstream 0 is a clearing bitstream
- Bitstream 1 address: 0x85C000
- Bitstream 1 size (bytes): 26036
- [✓] Bitstream 1 is a clearing bitstream

Trigger Options

- Number of Hardware Triggers: 0 [0 - 512]
- Number of Triggers allocated: 2 [2 - 512]
- First trigger to display: 0

Figure 10. DFX controller IP configuration[DFXCPG]

This configuration can change depending on the number of reconfigurable partitions and modules wanted. In this design there is only one reconfigurable partition and two reconfigurable modules that can fill this partition, so only one virtual socket manager is needed. If it wasn't the case, one virtual socket manager should be added for each reconfigurable partition using the buttons in the configuration interface. Then reconfigurable modules can be added too.

Each virtual socket manager can be configured and each reconfigurable module too. For each reconfigurable module, the controller needs to know the address and size of the corresponding partial bitstream and clearing bitstream (no clearing bitstream for UltraScale+ devices). All these parameters can nevertheless be modified after the design runs with the help of the slave AXI-Lite interface of this controller, which permits access to the internal configuration registers of the controller, where the user can write new values.

Once all necessary configurations are made, there exists a register that can be written with a certain value to launch a dynamic reconfiguration. This value depends on the number of VSMs and reconfigurable modules present in the design, but each reconfigurable module of each VSM is indexed to a precise value that once written to this register, called the SWTRIGGER register, will launch an instruction for the controller to go through the PCIe connection and fetch the corresponding partial bitstream from the external CPU memory.

4.5. SEM controller IP

4.5.1. Role

The purpose of this IP is to scan for, detect, and correct errors in the FPGA configuration. To do this, it needs two primitives which are the Frame ECC primitive, that serves for the calculation of some golden error correction code values, and the ICAP primitive, which is the internal configuration access port. This controller is commanded through a command interface, it also has a status interface to keep track of its state at the moment of reading the signals, and some other interfaces that serve for the access to the ICAP or the arbitration of this access if needed.

4.5.2. Interface signals

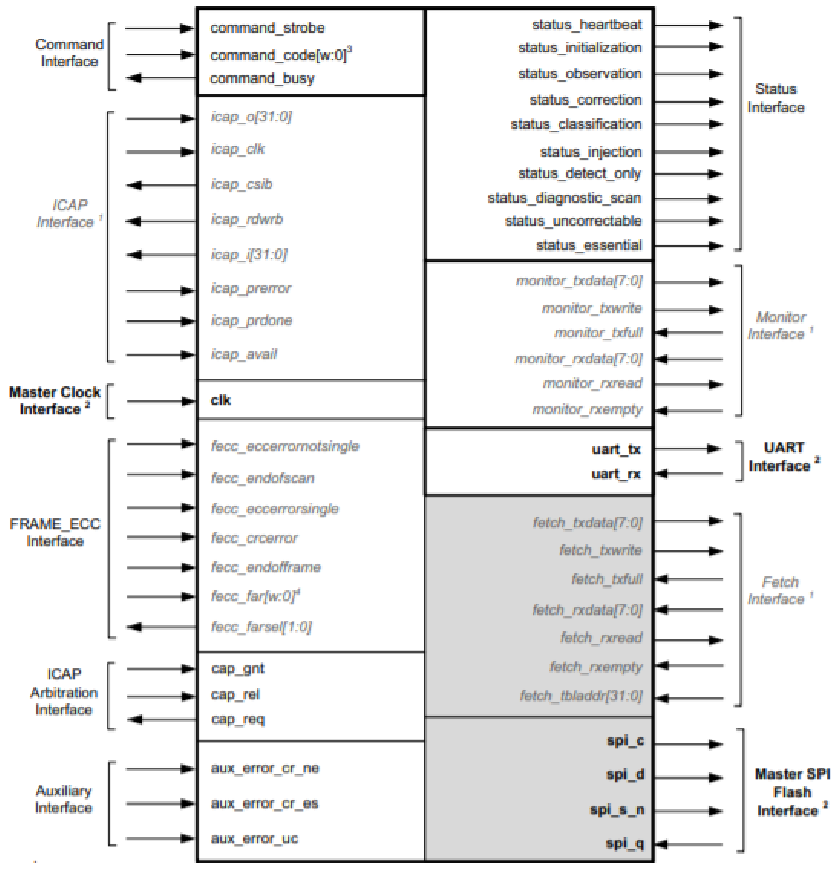


Figure 11. SEM controller interface signals diagram[SEMCPG]

This table describes the interface signals of this IP. Signals that are present in the diagram but not in the table are not used for this design and can be put to a constant value if it is an input or left open if it is an output.

Table 2. Interface signals of the SEM controller[SEMCPG]

Interface	Port	Dir	Description
Master clock interface	**clk**	Input	Main clock for the SEM controller.
Status interface	**status_heartbeat**	Output	The heartbeat signal is active and toggles every time a frame is read when status_observation, status_detect_only, or status_diagnostic_scan are active.
	status_initialization	Output	This signal is active during controller initialization.
	status_observation	Output	This signal is active during controller observation of bit upsets, it remains active after error detection while the controller queries the hardware for information.
	status_correction	Output	This signal is active during controller correction of an error or during a transition through this state if correction is disabled.
	status_classification	Output	This signal is active during controller classification of an error or during transition through this state if error classification is disabled.
	status_injection	Output	This signal is active during error injection. When the error is injected it returns inactive.
	status_detect_only	Output	This signal is active when the controller is in a detect only state. When the scan is interrupted because of an error that was found, it returns inactive
	status_diagnostic_scan	Output	This signal is active during controller diagnostic scan of the entire configuration of the FPGA. Once it finishes scanning, the signal returns inactive.
	status_uncorrectable	Output	The controller sets this signal prior to exiting the correction state to reflect the nature of a found error.
	status_essential	Output	This signal is an error classification signal. It is set by the controller prior to exiting the error classification state to reflect whether the error occurred on an essential bit.
Command interface	**command_busy**	Output	This signal indicates whether the SEM controller is ready to process a command. command_strobe should only be asserted when command_busy is low.
	command_code	Input	This signal is used to command the SEM controller. The value on this signal is captured at the same time when command_strobe is sampled active. For UltraScale devices, the width of this signal is 40 bits.
	command_strobe	Input	This signal needs to be pulsed synchronously to the clock when command_busy is low and a valid command_code signal is ready to be presented.
ICAP arbitration interface	**cap_gnt**	Input	This signal is to be asserted by an arbiter to tell the controller that it can start sending and receiving data from the ICAP.
	cap_req	Output	This signal is to be asserted by the controller to request for ICAP access.
	cap_rel	Input	This signal needs to be asserted by an arbiter to tell the controller that some other block is requesting access to the ICAP.
ICAP interface	**icap_i**	Output	Drives the data input of the ICAP.
	icap_o	Input	Is driven by the data output of the ICAP.

	icap_clk	Input	Clock for the ICAP interface.
	icap_csib	Output	Drives the CSIB input of the ICAP.
	icap_rdwrb	Output	Drives the RDWRB input of the ICAP.
	icap_prdone	Input	Is driven by the PRDONE output of the ICAP.
	icap_prerror	Input	Is driven by the PRERROR output of the ICAP.
	icap_avail	Input	Is driven by the AVAIL output of the ICAP.
Frame ECC interface	**Frame ECC interface**	Interface	Interface that connects to the Frame ECC primitive.

4.5.3. Detailed description

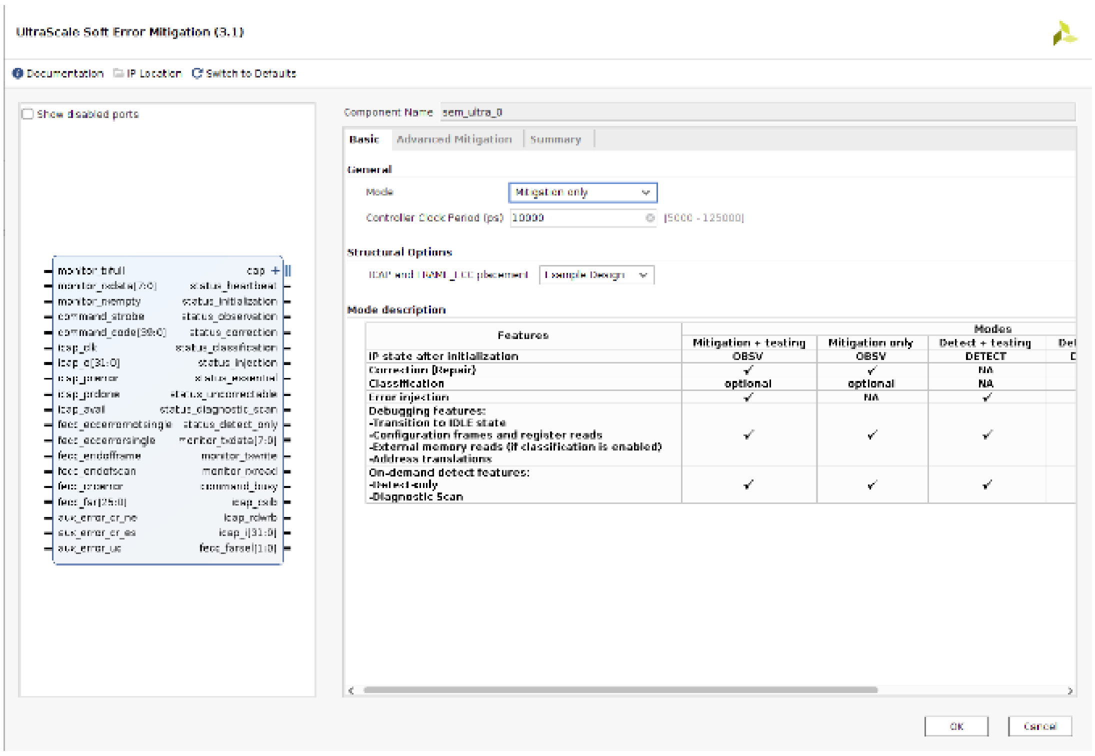
This module implements the Xilinx single error mitigation controller. The controller scans the internal configuration of the FPGA looking for errors to report and correct. To do this, it needs access to this internal configuration and this is achieved through the ICAP. There is however an arbiter in the way to the FPGA’s internal configuration, because the DFX controller needs access to it too. The SEM controller sends and receives data using 32-bit wide signals in and out. It also has an arbitration interface, a status interface, and a command interface.

The command interface has a command_code signal that is 40 bits wide, this signal is used to send instructions to the controller depending on the 4 MSBs, the 36 remaining bits are only used when in error injection mode and they serve to describe the error to be injected. However, this functionality is not used in this design as it is outside of the scope of DFX.

- **MSBs = 1110: Directed state change to the Idle State.**
- **MSBs = 1100: Directed state change to the Error Injection State.**
- **MSBs = 1010: Directed state change to the Observation State.**
- **MSBs = 1111: Directed state change to the Detect Only State.**
- **MSBs = 1101: Directed state change to the Diagnostic Scan State.**
- **MSBs = 1011: Directed state change to do a Software Reset.**

The status interface contains a number of signals that can be read to check the status of the controller at that moment and see if it is doing the job it is expected to do.

The SEM controller IP can be instantiated using the Vivado IP catalog and here is the configuration used for this design:



UltraScale Soft Error Mitigation (3.1)

Component Name: sem_ultra_0

General

Mode: Mitigation only

Controller Clock Period (ps): 10000 [5000 - 125000]

Structural Options

ICAP and FRAME_ECC placement: Example Design

Mode description

Features	Modes			
	Mitigation + testing	Mitigation only	Detect + testing	Det
IP state after initialization	OBSV	OBSV	DETECT	D
Correction (Repair)	✓	✓	NA	
Classification	optional	optional	NA	
Error injection	✓	NA	✓	
Debugging features: -Transition to IDLE state -Configuration frames and register reads -External memory reads (if classification is enabled) -Address translations	✓	✓	✓	
On-demand detect features: -Detect-only -Diagnostic Scan	✓	✓	✓	

Figure 12. Configuration of the SEM IP[SEMCPG]

The mode has been chosen here to be Mitigation Only because the purpose of this design is to demonstrate Dynamic Function eXchange on an UltraScale device and this feature is totally independent from it, but it can be modified according to the needs of the user using the Vivado interface dedicated for IP configuration.

4.6. AXI bridge for PCIe IP

4.6.1. Role

The purpose of this IP is to connect the FPGA to an external CPU using a PCI express connection. This enables the CPU to arrange read and write transactions to be able to control all of the needed operations to run dynamic reconfiguration and manage the error correction operations.

4.6.2. Interface signals

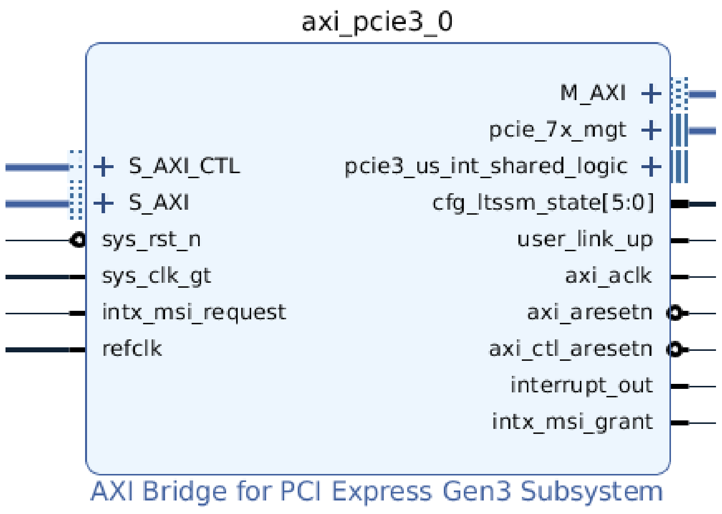


Figure 13. AXI bridge for PCIe signals[AXIPCIPG]

This table describes the interface signals of this IP. Signals that are present in the diagram but not in the table are not used for this design and can be put to a constant value if it is an input or left open if it is an output.

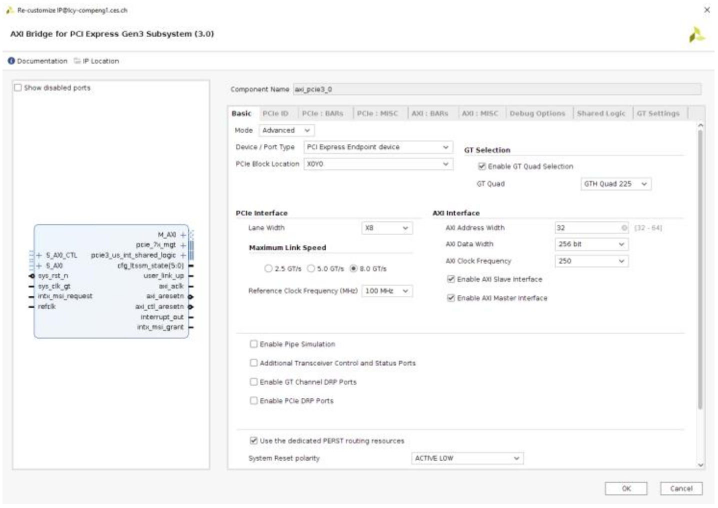
Table 3. AXI bridge for PCIe input/output ports description[AXIPCIPG]

Port	Dir	Description
M_AXI	Interface	Master AXI interface connected on the slave interface of the AXI interconnect.
S_AXI_CTL	Interface	Slave AXI interface meant for controlling and configuring the PCIe bridge.
S_AXI	Interface	Slave AXI interface connected to a master interface on the PRC.
axi_aclk	Output	Clock generated by the PCIe bridge meant to drive the clocks of the AXI interfaces.
axi_aresetn	Output	Active low reset generated by the PCIe bridge meant for the AXI interfaces.
axi_ctl_aresetn	Output	Active low reset generated by the PCIe bridge meant for the AXI interface that is connected to S_AXI_CTL.
sys_rst_n	Output	PCIe reset generated by the PCIe edge connector itself.
sys_clk_gt	Output	PCIe reference clock
refclk	Output	PCIe reference clock.

4.6.3. Detailed description

The PCI express bridge enables a PC to communicate with the FPGA to enable operations with the goal of controlling the different blocks in it. Its important interfaces are the S_AXI_CTL interface, which serves to configure this block by accessing its internal configuration registers, the S_AXI interface which is a slave interface to be driven by the master interface of the DFX controller for the purpose of fetching partial bit files, and the M_AXI interface which is a master interface driving a slave interface of the AXI interconnect.

It can be directly instantiated using the Vivado IP catalog but it needs to be configured. Here is the configuration that was used for this design:



AXI Bridge for PCI Express Gen3 Subsystem (3.0)

☐ Show disabled ports

Component Name: axi_pcie3_0

Mode: Advanced
Device / Port Type: PCI Express Endpoint device
PCIe Block Location: X0Y0

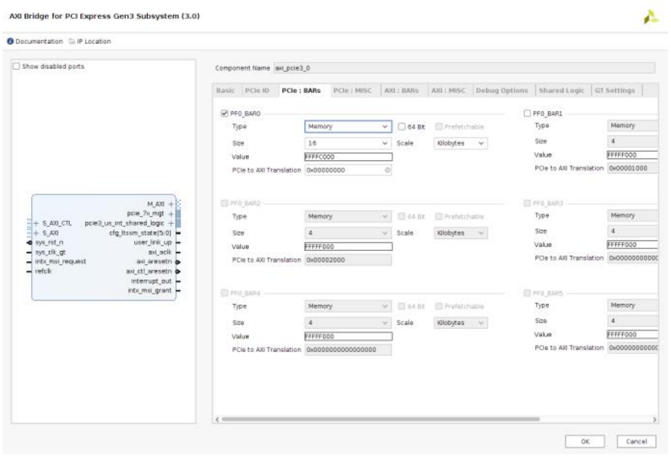
GT Selection
☑ Enable GT Quad Selection
GT Quad: GTH Quad 225

PCIe Interface
Lane Width: X8
Maximum Link Speed
○ 2.5 GT/s ○ 5.0 GT/s ◉ 8.0 GT/s
Reference Clock Frequency (MHz): 100 MHz

AXI Interface
AXI Address Width: 32 [32 - 64]
AXI Data Width: 256 bit
AXI Clock Frequency: 250
☑ Enable AXI Slave Interface
☑ Enable AXI Master Interface

☐ Enable Pipe Simulation
☐ Additional Transceiver Control and Status Ports
☐ Enable GT Channel DRP Ports
☐ Enable PCIe DRP Ports

☑ Use the dedicated PERST routing resources
System Reset polarity: ACTIVE LOW



AXI Bridge for PCI Express Gen3 Subsystem (3.0)

☐ Show disabled ports

Component Name: axi_pcie3_0

☑ PF0_BAR0
Type: Memory ☐ 64 Bit ☐ Prefetchable
Size: 16 Scale: Kilobytes
Value: FFFFC000
PCIe to AXI Translation: 0x00000000

☐ PF0_BAR1
Type: Memory
Size: 4
Value: FFFFF000
PCIe to AXI Translation: 0x00001000

☐ PF0_BAR2
Type: Memory ☐ 64 Bit ☐ Prefetchable
Size: 4 Scale: Kilobytes
Value: FFFFF000
PCIe to AXI Translation: 0x00002000

☐ PF0_BAR3
Type: Memory
Size: 4
Value: FFFFF000
PCIe to AXI Translation: 0x00000000000

☐ PF0_BAR4
Type: Memory ☐ 64 Bit ☐ Prefetchable
Size: 4 Scale: Kilobytes
Value: FFFFF000
PCIe to AXI Translation: 0x0000000000000000

☐ PF0_BAR5
Type: Memory
Size: 4
Value: FFFFF000
PCIe to AXI Translation: 0x00000000000

Figure 14. AXI bridge for PCIe configuration[AXIPCIPG]

PCIe BARs are important to be correctly configured to ensure a proper address translation operation for successful read/write operations.

4.7. AXI interconnect IP

4.7.1. Role

The purpose of this IP is to distribute read and write operations depending on the address signal, since the transactions coming from the CPU can address different blocks, an interconnect bus is needed to determine which block inside the FPGA was meant to receive the transaction, and this is the role of this IP.

4.7.2. Interface signals

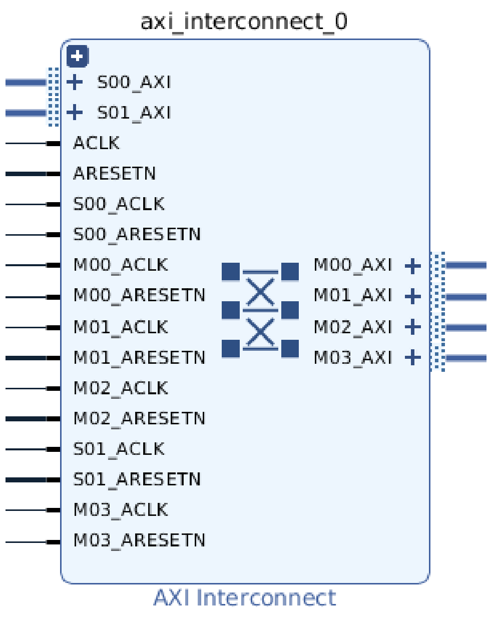


Figure 15. AXI interconnect signals[AXIPG]

Table 4 – AXI interconnect input/output ports description[AXIPG]

Port	Dir	Description
ACLK	Input	Main clock of the AXI interconnect.
ARESETN	Input	Main reset of the AXI interconnect.
S00_ACLK	Input	Clock for the S00 slave AXI interface.
S00_ARESETN	Input	Reset for the S00 slave AXI interface.
S01_ACLK	Input	Clock for the S01 slave AXI interface.
S01_ARESETN	Input	Reset for the S01 slave AXI interface.
M00_ACLK	Input	Clock for the M00 master AXI interface.
M00_ARESETN	Input	Reset for the M00 master AXI interface.
M01_ACLK	Input	Clock for the M01 master AXI interface.
M01_ARESETN	Input	Reset for the M01 master AXI interface.
M02_ACLK	Input	Clock for the M02 master AXI interface.
M02_ARESETN	Input	Reset for the M02 master AXI interface.

M03_ACLK	Input	Clock for the M03 master AXI interface.
M03_ARESETN	Input	Reset for the M03 master AXI interface.
S00_AXI	Interface	Slave interface connected to the master interface of the PCIe bridge.
S01_AXI	Interface	Slave interface connected to the master interface of the PRC
M00_AXI	Interface	Master interface connected to the S_AXI_CTL interface of the PCIe bridge.
M01_AXI	Interface	Master interface connected to a slave interface of the PRC.
M02_AXI	Interface	Master interface connected to the slave interface of the AXI to APB bridge.
M03_AXI	Interface	Master interface connected to the slave PCI control interface on the PCIe bridge

4.7.3. Detailed description

An AXI interconnect enables communication between masters and slaves following the AXI protocol. In this design, it enables read and write operations from the PCIe bridge to three different slaves, and the other way around from the DFX controller to the PCIe connection.

In this design there are 2 AXI slave interfaces and 4 AXI master interfaces connected as in the figure:

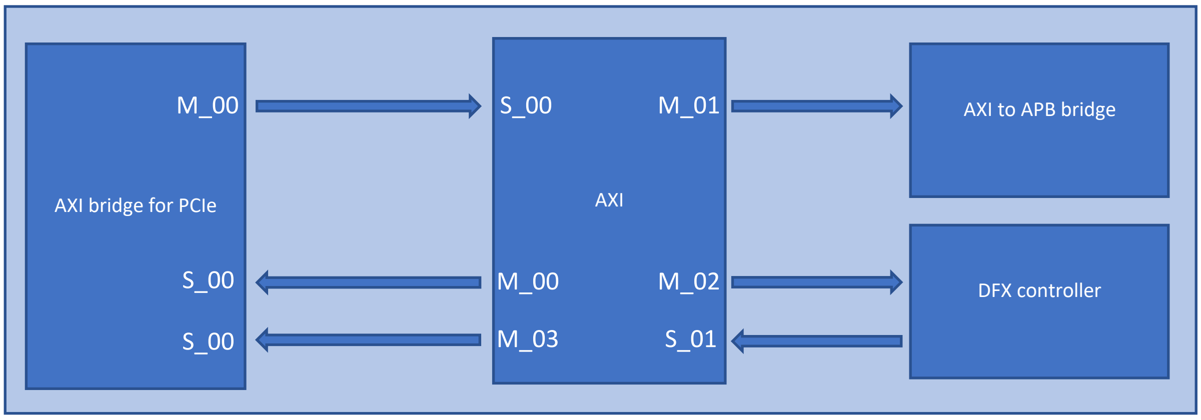


Figure 16. Slave and master interfaces connections to different FPGA blocks

An address mapping is set in the Vivado interface to ensure each block has its proper address window; this address window needs to be sufficiently large to include all of the registers of the different blocks. It is an IP that can be directly instantiated from the Vivado IP catalog.

4.8. Bus interface

4.8.1. Role

The purpose of this module is to arrange read and write operations on the interfaces of the SEM controller using APB protocol. This is a block that was made using VHDL RTL writing and an APB protocol was chosen because it is much simpler than the AXI protocol, and an AXI to APB bridge is used for this reason.

4.8.2. Interface signals

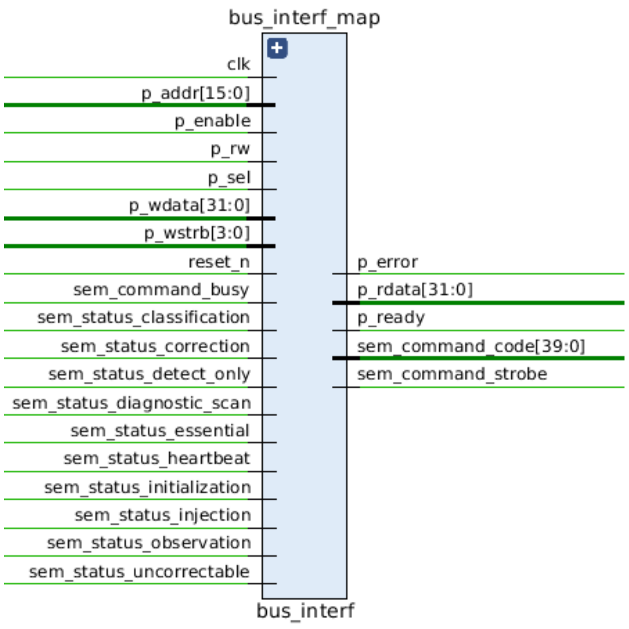


Figure 17. Bus interface signals

Table 5 –Bus interface input/output ports description

Port	Dir	Description
clk	Input	Clock of the register interface
reset_n	Input	synchronous reset
p_addr	Input	16-bit address signal
p_wdata	Input	32 bits write data signal
p_wstrb	Input	Write strobe signal. In this design this signal should always be set to “1111” else the register interface returns an error.
p_sel	Input	Selection signal driven to 1 by the APB bus when a transfer is required
p_rw	Input	Driven by the APB bus to a 1 for write transactions and to a 0 for read transactions
p_enable	Input	Driven by the APB bus to a 1 when the second cycle has started for an APB transfer
sem_status_heartbeat	Input	(See signal description of the SEM controller section)
sem_status_initialization	Input	

sem_status_observation	Input	
sem_status_correction	Input	
sem_status_classification	Input	
sem_status_injection	Input	
sem_status_detect_only	Input	
sem_status_diagnostic_scan	Input	
sem_status_uncorrectable	Input	
sem_status_essential	Input	
sem_command_busy	Input	
sem_command_code	Output	
sem_command_strobe	Output	
p_rdata	Output	32 bit read data signal
p_ready	Output	Ready signal that indicates when the slave is finishing up a transaction. Can be used to extend a transfer if needed by keeping this signal low.
p_error	Output	Error signal that indicates that the slave has encountered an error during a transfer and thus couldn't complete it.

4.8.3. Detailed description

The SEM IP has an interface that provides information about its current state and the job it is performing at that instant, called the status interface, along with another interface that makes it possible for the user to send commands to the IP, called the command interface. Both these interfaces are connected to the design presented here in a way that makes it possible for the user to retrieve or send data in an APB protocol compatible way.

Table 6. List of the registers

Register	Address	Width	Type	Description
Status interface	0x0000	32 bits	Read only	11 status signals in read only mode. To gather information about the status of the SEM controller and its current state.
Command code	0x0004	32 bits	Write only	Register to send the command signal to the SEM controller. The command signal is 40 bits wide.(44 in Ultrascale +) This register contains the 32 most significant bits of the command_code signal
Complementary command code	0x0008	32 bits	Write only	This register contains the 8 LSB of the command_code signal
Status correction counter	0x000C	32 bits	Read only	This register contains a count up of the number of times the signal status_correction has toggled since the last reading.
Status heartbeat counter	0x0010	32 bits	Read only	This register contains a count up of the number of times the signal status_heartbeat has risen since the last reading.

- **Status register (SR)**

The status register is used to read status signals of the SEM controller. These signals are an indicator of the state of the controller and are important to the managing CPU.

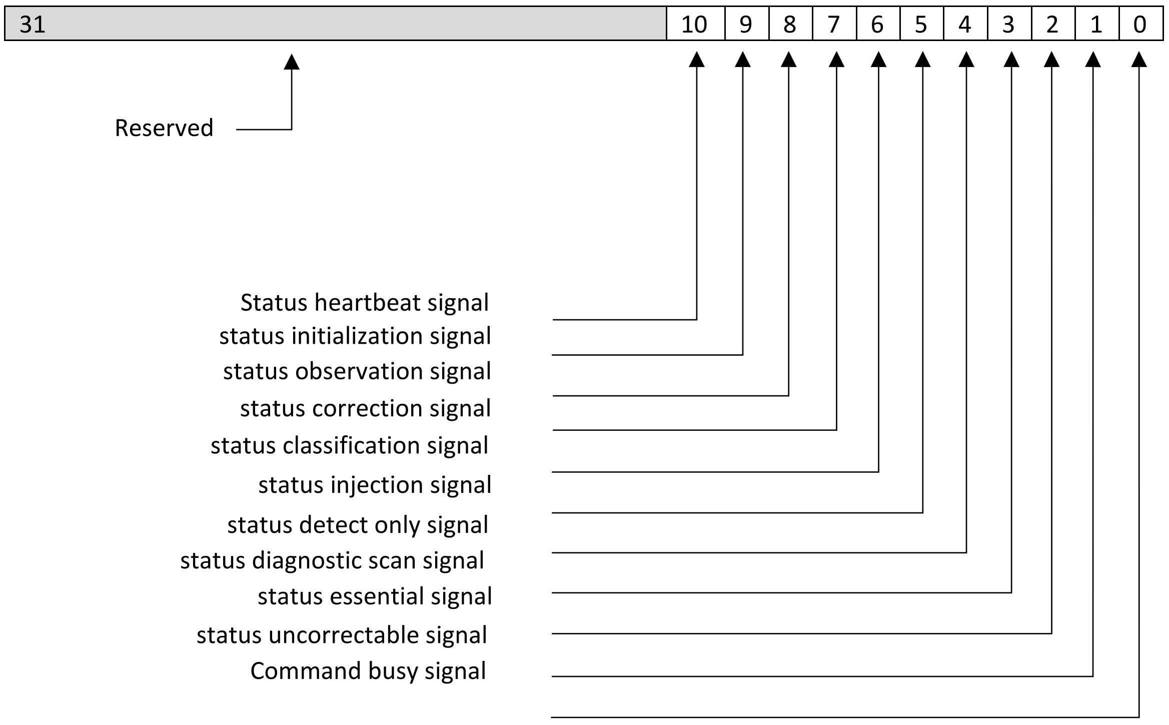

31
10
9
8
7
6
5
4
3
2
1
0
Reserved
Status heartbeat signal
status initialization signal
status observation signal
status correction signal
status classification signal
status injection signal
status detect only signal
status diagnostic scan signal
status essential signal
status uncorrectable signal
Command busy signal

Figure 18. Status register bit by bit

Table 7. List of the bits for the status register

Register Bits	Name	Access type	Reset value	Description
31-11	Reserved	N.A	0	Reserved bits.
10	status heartbeat	Read	0	Signal active when the controller is in the observation, detection or diagnostic scan states.
9	status initialization	Read	0	Read 1 when the controller is in the initialization state. Which occurs one time after the design begins to work.
8	status observation	Read	0	Read 1 as long as the controller is observing bit upsets to check for errors.
7	status correction	Read	0	Read 1 when the controller is correcting an error.
6	status classification	Read	0	Read 1 when the controller is classifying errors.
5	status injection	Read	0	Read 1 during controller injection of an error.
4	status detect only	Read	0	Read 1 as long as the controller is in the detect only state.
3	status diagnostic scan	Read	0	Read 1 when the controller is executing a diagnostic scan.
2	status essential	Read	0	Before exiting the classification state, the controller sets this signal to 1 if the error detected occurred on an essential bit.
1	status uncorrectable	Read	0	Before exiting the correction state, the controller sets this signal to 1 if the error detected is uncorrectable.
0	Command busy	Read	?	Read 1 when the controller is busy and should not be presented with a command_code signal.

- **Command code register (CCR)**

The command code register is used to send the command code signal to the controller. The command code signal is 40 bits wide; this register takes care of the 32 first MSBs of the command code signal and the CCCR takes care of the 8 remaining LSBs.

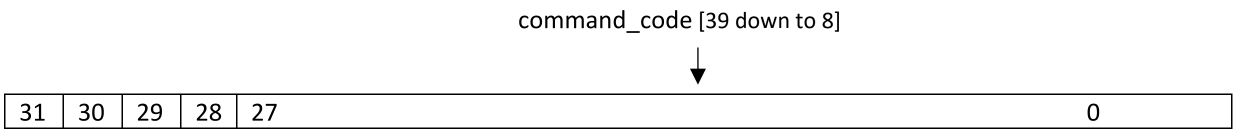


Figure 19. Command code register bit by bit

Table 8. List of the bits for the command code register

Register Bits	Name	Access type	Reset value	Description
31-28	4 MSBs of command_code	Write	0000	The 4 MSBs of the 40-bit wide command_code signal. These 4 bits are the only bits that can command the controller to transition into the following states: • **1110 for the Idle state, but only valid when in Observation and Detect only states** • **1100 for the Error Injection state, but only valid when in Idle state** • **1010 for the Observation state, but only valid when Idle state** • **1111 for the Detect only state, but only valid when in Idle state** • **1101 for Diagnostic scan state, but only valid when in Idle state** • **1011 for a Software reset, but only valid when in Idle state**
27-0	Command_code[35-8]	Write	"0"	The command_code signal is 40 bit wide. The first 4 MSBs are what decide of the controller state. The rest of the 40 bits are useful in case when the controller is in error injection state.

- **Complementary command code register (CCCR)**

The CCCR register is a complementary register used to complement the CCR on the 8 LSBs of the command_code signal. It is only relevant when the SEM controller is in error injection mode.

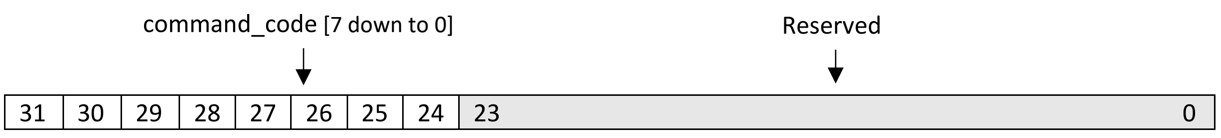


Figure 20. Complementary command code register

Table 9. List of the bits of the complementary command code register

Bits	Name	Access type	Reset value	Description
31-24	The 8 LSBs of the command_code signal	Write	0	Complementary to the other bits from the CCR when in error injection state.
23-0	Reserved	N.A	0	Reserved.

- **Correction counter register (CoCR)**

The correction counter register is a register that stores the calculated value of the number of times the signal status_correction has risen. This register is reinitialized to 0 after each reading operation if the signal status_correction doesn't experience a rising transition during the read process. Else the counter reinitializes to 1.

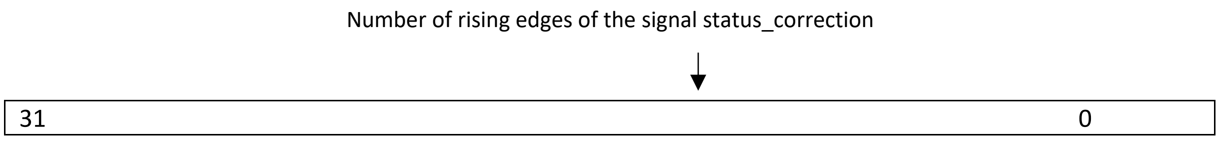


Figure 21. Correction count register

Table 10. List of the bits of the correction counter register

Bits	Name	Access type	Reset value	Description
31-0	32-bit signal indicating the number of times status_correction has risen.	Read	0	Integer number indicating the number of times status_correction has risen from 0 to 1. Every time this value is incremented by one means that the SEM controller has gone through the correction state.

- **Heartbeat counter register (HCR)**

The heartbeat counter register is a register that stores the calculated value of the number of times the signal status_heartbeat has risen. This register is reinitialized to 0 after each reading operation if the signal status_heartbeat doesn't experience a rising transition during the read process. Else the counter reinitializes to 1.

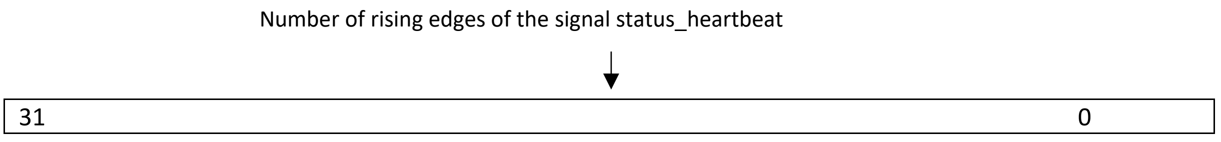


Figure 22. Heartbeat counter register

Table 11. List of the bits of the heartbeat counter register

Bits	Name	Access type	Reset value	Description
31-0	32 bit signal indicating the number of times status_heartbeat has risen.	Read	0	Integer number indicating the number of times status_heartbeat has risen from 0 to 1. This signals keeps rising and falling as long as the controller is working.

4.9. Arbiter

4.9.1. Role

The presence of the two controllers that need access to the ICAP to do their job implies the necessity of an arbiter to orchestrate control over the port. Both these controllers already dispose of grant, request and release signals whose purpose is to help the arbiter in the management of access. This arbiter is a non-trivial multiplexing and demultiplexing element.

4.9.2. Detailed description

The finite state machine of the arbiter looks like this:

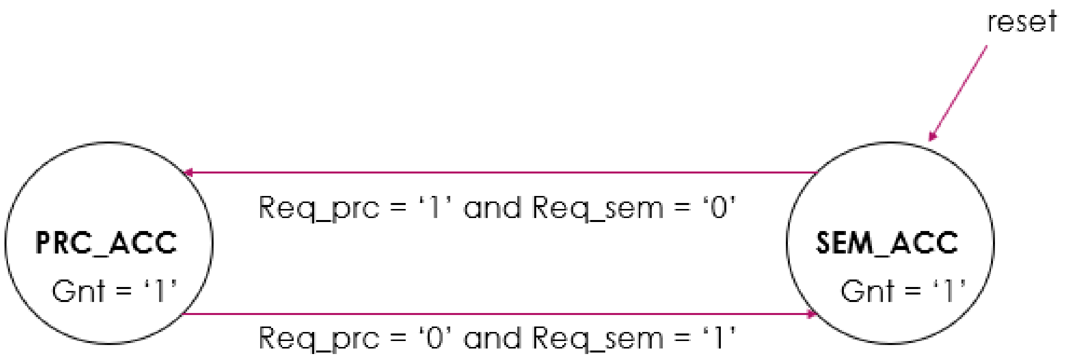

reset
PRC_ACC
Gnt = '1'
SEM_ACC
Gnt = '1'
Req_prc = '1' and Req_sem = '0'
Req_prc = '0' and Req_sem = '1'

Figure 23. Finite state machine of the arbiter

By default, pushing a reset gives the access to the SEM controller. This controller is prioritized over the DFX controller because of the simple fact that it needs constant access to the internal configuration of the FPGA to look for errors, meanwhile the DFX controller only needs access when a dynamic reconfiguration needs to be performed.

When the arbiter is in one of the two states giving access to one module, the other module can issue a request to acquire the access to the ICAP. The arbiter will then examine the state of the module that already has access, if it finishes its work, the arbiter will switch to the other state giving access to the second module.

4.10. Reconfigurable module counter

4.10.1. Role

This module is the one that will be loaded either with a count up function or a count down function depending on the trigger sent to the DFX controller. The output of this counter is connected to a series of user LEDs in the KCU105 board to have a better observation of the modules when interchanged.

4.10.2. Interface signals

Table 12 – Counter input/output ports

Port	Dir	Description
clk	Input	Clock signal.
reset_n	Input	Synchronous reset signal.
LEDs	Input	Output signal of the counter that is going to be mapped to the LEDs.
vsm_VS_0_rm_shutdown_req_0	Output	Handshake request signal coming from the PRC.
vsm_VS_0_rm_shutdown_ack_0	Output	Handshake acknowledge signal delivered to the PRC.

4.10.3. Detailed description

This module is a counter that performs either a count-up or a count-down function depending on the reconfigurable module that is loaded in the reconfigurable partition by the PRC. It's a 32-bit counter but only the 8 MSBs are mapped to user LEDs on the KCU105 board. This is because the clock frequency used to run this counter is around 100 MHz and it is impossible to detect changes on a LED that switches this fast with a human eye. Connecting only the 8 MSBs to the LEDs makes it much easier to see what is happening because the MSBs flip in a much slower way.

4.11. AXI to APB bridge IP

4.11.1. Role

The purpose of this IP is to convert the AXI protocol to an APB protocol that will then be used for interfacing with the APB bus interface of the SEM controller.

4.11.2. Interface signals

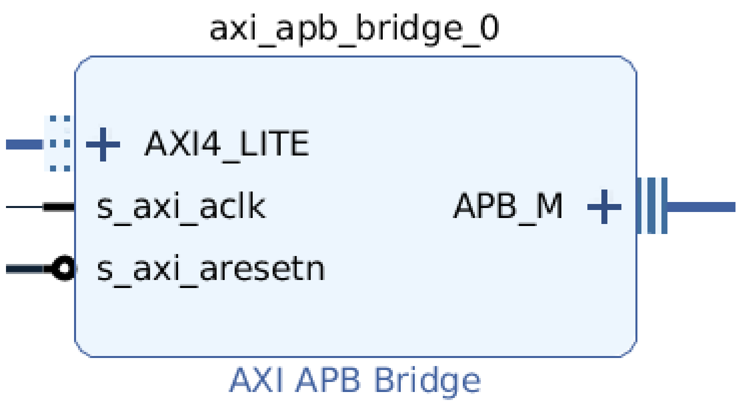


Figure 24. AXI to APB bridge signals[AXIAPBPG]

Table 13 – AXI to APB bridge input/output ports[AXIAPBPG]

Port	Dir	Description
s_axi_aclk	Input	Clock for the AXI slave interface.
s_axi_aresetn	Input	Reset for the AXI slave interface.
AXI4_LITE	Interface	AXI slave interface driven by the AXI interconnect master interface.
APB_M	Interface	APB master interface driving the bus interface.

4.11.3. Detailed description

Since the PCIe bridge here is intended for use with an AXI interconnect and since the bus interface designed for the SEM controller uses an APB protocol, an AXI to APB bridge is needed to convert communication operations between these two protocols. It is an IP that can be instantiated directly using the Vivado IP catalog.

4.12. Decoupler IP

4.12.1. Role

The purpose of this module is to isolate reconfigurable partitions from static logic in order to protect the static logic from the toggling that occurs in the interface signals between reconfigurable logic and static logic when a DFX operation is on the run.

4.12.2. Interface signals

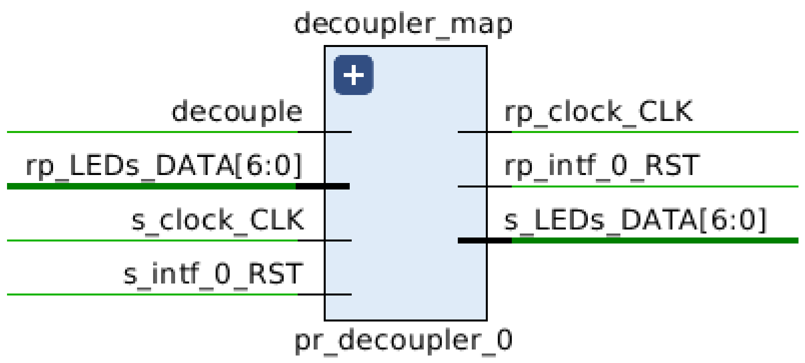


Figure 25. Decoupler IP signals[DFXDPG]

Table 14 – Arbiter input/output ports[DFXDPG]

Port	Dir	Description
s_intf_0_RST	Input	Reset signal intended for the reconfigurable logic coming from the PRC.
rp_intf_0_RST	Output	
s_clock_CLK	Input	Clock intended to clock the reconfigurable logic.
rp_clock_CLK	Output	
s_LEDs_DATA	Output	Output of the counter intended to be mapped to the 8 user LEDs on the board.
rp_LEDs_DATA	Input	
decouple	Input	Input signal driven by the PRC to execute the decoupling function.
decouple_status	Output	Output signal that indicates the decoupling status.

5. DFX flow

The deliveries expected by the client include an example DFX design but also a complete description of the flow necessary to generate the bit files and a description of how to load these bit files dynamically. The client also expects some instructions on how to migrate this design from an UltraScale FPGA to an UltraScale+ FPGA. In this context, I made a Vivado project directory called **DFX/**, where all the IPs and coded RTL logic presented earlier is present and linked together to make the full design. Another project directory, called **DFX_KCU105_count/** was also made, and this one contains folders arranged in a tree structure that I imagined myself accompanied with a script that starts with a synthesized checkpoint of the static logic from the DFX Vivado project, and VHDL files or the two reconfigurable modules to implement all of the DFX flow and generate the final bitstreams needed.

5.1. Differences from a classical flow

A classical flow and a DFX flow are somewhat the same but still have some differences one from another. The classical flow consists of elaboration, synthesis, implementation then bit file generation. So does the DFX flow but with slight differences.

The first difference is, for a DFX project we need not only one full bit file to configure the whole FPGA but one principal bit file and some partial bit files that will serve for the partial dynamic reconfiguration of the reconfigurable partitions, so what is required of this flow at the end is several bit files (in our case three).

The second difference is that since the reconfigurable partitions will be loaded with different reconfigurable modules, this flow will save the static place & route results with an empty black box for the reconfigurable partitions. Then synthesized checkpoints of reconfigurable modules will be added to the project each time and new place & route operations will be made for each configuration (static logic + reconfigurable modules one by one).

The third main difference is a directive for the implementation tool. This tool needs to be told that no optimization is permitted between the boundaries of reconfigurable partitions, simply because these partitions need to always have exactly the same interfaces and signals that connect them to their neighbors. This is done with the use of Pblocks.

5.2. DFX flow[DFXT]

- **Arranging the design**

Before starting the DFX flow, the design can be adapted to the user's needs. To do this, there is a Vivado project folder called DFX that can be opened with the help of the Vivado software. This project contains all the sources and IPs necessary to implement the static logic and it is the project to be modified for adaptability with the user's needs. A synthesis is then necessary and a checkpoint saving when the synthesis is done. To write a checkpoint either graphically navigate to **File > Checkpoint > Write** or use the command *write_checkpoint.*

Once this is done, this checkpoint file should be placed in the **DFX_KCU105_count/synthesized_checkpoints/top/**project directory.

The reconfigurable modules are synthesized separately to do an out of context synthesis. The corresponding VHDL files for these modules are placed in the directory **DFX_KCU105_count/sources/** .

- **Synthesizing the design**

Since the need at the end is to have three bit files: one bit file for the static logic, and two partial bit files for the reconfigurable modules that will fill the empty slot in the reconfigurable partition, the first thing that needs to be done is a synthesis of three designs. One synthesis of all the static logic instantiating an empty module containing just the input output ports of the counter, so that the tool will acknowledge the existence of a counter module without knowing its internal functionality, this will issue a critical warning in the Vivado messages slot but it can be ignored. Then an out-of-context synthesis of each reconfigurable module separately. The out of context synthesis type is necessary here because otherwise Vivado will consider the counter modules as finalized designs and will insert some elements such as clock buffers that will make the place and route of the full design hard since the Pblock that will be created will then need to include clock buffers even if it is not necessary, this may even cause the place and route operation to not converge.

To do all of this first open Vivado in TCL or GUI mode, then navigate into the DFX_KCU105_count project directory.

Then set some variables that will help issuing commands needed.

```
set part “xcku040-ffva1156-2-e”

set board “kcu105”
```

Create an in_memory project using the command:

```
create_project -in_memory -part $part
```

Then add the reconfigurable module file and set it as the top of the design suing the commands:

```
add_files ./sources/count_up.vhd

set_property top count [current_fileset]
```

The next step is to do an out of context synthesis, write a checkpoint of the synthesized design and close the project:

```
synth_design -mode out_of_context

write_checkpoint ./synthesized_checkpoints/RM_count_up/checkpoint_count_up.dcp

close_project
```

The same process needs to be repeated for the second reconfigurable module:

```
create_project -in_memory -part $part

add_files ./sources/count_down.vhd

set_property top count [current_fileset]
```

synth_design -mode out_of_context

write_checkpoint ./synthesized_checkpoints/RM_count_down/checkpoint_count_down.dcp

close_project

After all of this is executed correctly, you can find the three synthesized checkpoints by navigating to the directory DFX_KCU105_count and going into the directory ./synthesized_checkpoints/.

- **Assembling and implementing the design**

Now that the synthesized checkpoints are done, the design can be assembled.

Create an in-memory design by issuing the following command

create_project -in_memory -part $part

Load the static design :

add_files ./synthesized_checkpoints/top/checkpoint_top.dcp

This command calls for the synthesized design checkpoint of the static logic that was just created in the previous section.

Load the top-level design constraints by issuing these commands :

add_files ./constraints/KCU_cnstrn.xdc

set_property USED_IN {implementation} [get_files ./constraints/KCU_cnstrn.xdc]

Load the first synthesized checkpoint for the count function (the count up function is chosen here but the count down is viable too) :

add_files ./synthesized_checkpoints/RM_count_up/checkpoint_count_up.dcp

set_property SCOPED_TO_CELLS {inst_count} [get_files ./synthesized_checkpoints/ RM_count_up/checkpoint_count_up.dcp]

Then link the entire design together using the command:

link_design -mode default -reconfig_partitions {inst_count} -part $part -top fpga_top

Now a full configuration is loaded, with static and reconfigurable logic. All that is left to do for this section is to save a checkpoint for this design by issuing the following command:

write_checkpoint ./synthesized_checkpoints/linked_design/top_link_up.dcp

- **Building the design floorplan**

This can either be done graphically, using a cursor and the Vivado interface, or by issuing commands.

To do it graphically:

Select the inst_count instance in the Netlist pane and right click on it, select **Floorplanning > Draw Pblock**, then draw a box on the X0Y4 clock region.

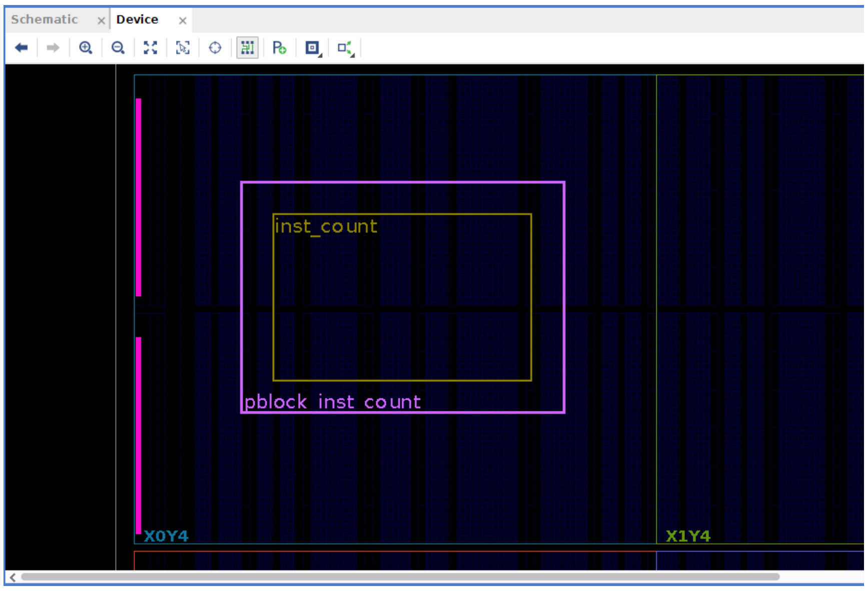


Figure 26. Pblock position in the implemented device

Run partial reconfiguration Design Rule Checks by selecting **Reports > Report DRC**. Make sure **Partial Reconfiguration** is checked and remove all others to focus this report exclusively on PR DRCs.

No DRC errors should be reported at this point. If so they must be fixed before moving forward.

Or else issue these commands instead:

create_pblock pblock_inst_count

resize_pblock pblock_inst_count -add {SLICE_X3Y257:SLICE_X19Y285 DSP48E2_X0Y104:DSP48E2_X2Y113 RAMB18_X0Y104:RAMB18_X1Y113 RAMB36_X0Y52:RAMB36_X1Y56}

add_cells_to_pblock pblock_inst_count [get_cells [list inst_count]] -clear_locs

create_drc_ruledeck ruledeck_1

add_drc_checks -ruledeck ruledeck_1 [get_drc_checks {HDPRA-62 HDPRA-60 HDPRA-58 HDPRA-57 HDPRA-56 HDPRA-55 HDPRA-54 HDPRA-53 HDPRA-52 HDPRA-51 HDPRA-21 HDPR-43 HDPR-20 HDPR-88 HDPR-41 HDPR-30 HDPR-96 HDPR-95 HDPR-94 HDPR-93 HDPR-92 HDPR-91 HDPR-90 HDPR-87 HDPR-86 HDPR-85 HDPR-84 HDPR-83 HDPR-74 HDPR-73 HDPR-72 HDPR-71 HDPR-70 HDPR-69 HDPR-68 HDPR-67 HDPR-66 HDPR-65 HDPR-64 HDPR-63 HDPR-62 HDPR-61 HDPR-60 HDPR-59 HDPR-58 HDPR-57 HDPR-54 HDPR-50 HDPR-49 HDPR-48 HDPR-47 HDPR-46 HDPR-44 HDPR-42 HDPR-38 HDPR-37 HDPR-35 HDPR-34 HDPR-33 HDPR-32 HDPR-29 HDPR-28 HDPR-25 HDPR-23 HDPR-22 HDPR-18 HDPR-17 HDPR-16 HDPR-14 HDPR-13 HDPR-12 HDPR-11 HDPR-6 HDPR-5 HDPR-4 HDPR-3 HDPR-2 HDPR-1}]

report_drc -name drc_1 -ruledecks {ruledeck_1}

delete_drc_ruledeck ruledeck_1

If the add_drc_checks commands seems too long, it can either be copied and pasted or replaced by manual commands using the Vivado interface as shown earlier.

Save these Pblocks and associated properties:

write_xdc ./constraints/top_all.xdc

- **Implementing the first configuration**

Here, the goal is to place and route the static portion of the design without omitting the presence of the reconfigurable partition.

First, issue the following commands:

```
opt_design

place_design

route_design
```

Save the full design checkpoint

```
write_checkpoint -force implemented_checkpoints/config_count_up/top_routed.dcp

report_utilization -file reports/config_count_up/top_utilization.rpt

report_timing_summary -file reports/config_count_up/top_timing_summary.rpt
```

Save a checkpoint for the reconfigurable module :

```
write_checkpoint -force -cell inst_count implemented_checkpoints/config_count_up/ count_up_routed.dcp
```

At this point of the flow, a fully implemented DFX design is generated from which full and partial bitstreams can be created. The static portion of this configuration is going to be the same for all configurations, so to isolate it, issue the following commands to remove the reconfigurable logic:

```
update_design -cell inst_count -black_box
```

Now inst_count should appear in the Netlist pane as empty.

Issue the following command to lock down all placement and routing :

```
lock_design -level routing
```

This locks the entire design consisting of the static logic and a black box instead of the reconfigurable logic.

Issue the following command to save a checkpoint of the static implemented logic:

```
write_checkpoint -force checkpoints/static_route_design.dcp
```

Now close this design before moving on to the next step:

```
close_project
```

- **Implementing the second configuration**

First, create a new in memory design :

```
create_project -in_memory -part $part
```

Then load the static design checkpoint that was just created in the last steps of the previous section :

```
add_files ./checkpoints/static_route_design.dcp
```

Load the second reconfigurable module that is in this case the count down function module :

```
add_files ./synthesized_checkpoints/RM_count_down/checkpoint_count_down.dcp
```

set_property SCOPED_TO_CELLS {inst_count} [get_files ./synthesized_checkpoints/ RM_count_down/checkpoint_count_down.dcp]

Link the entire design together using the command :

link_design -mode default -reconfig_partitions {inst_count} -part $part -top fpga_top

Optimize, place, and route the design :

opt_design

place_design

route_design

Save the resulting design and report files :

write_checkpoint -force implemented_checkpoints/config_count_down/top_routed.dcp

report_utilization -file reports/config_count_down/top_utilization.rpt

report_timing_summary -file reports/config_count_down/top_timing_summary.rpt

Save a checkpoint for the reconfigurable module :

write_checkpoint -force -cell inst_count implemented_checkpoints/config_count_down/ count_down_routed.dcp

Now the only thing left to do before generating the bitstreams is to run a pr_verify to check the place & route status of the two implemented configurations and verify if they are exactly the same, which they should be.

pr_verify implemented_checkpoints/config_count_up/top_routed.dcp implemented_checkpoints/config_count_down/top_routed.dcp

Close the project.

close_project

- **Generating bitstreams**

Now that the configurations have been verified, bitstreams can be generated.

First, open the first configuration checkpoint into memory:

open_checkpoint implemented_checkpoints/config_count_up/top_routed.dcp

then generate full and partial bitstreams for this configuration using this command :

write_bitstream -force -bin_file bitstreams/config_count_up/config_up

As a result, some bitstreams are generated (no clearing bitstreams if it's an Ultrascale+ device) :

- **config_up.bit**

  This is a power-up full configuration bitstream that will program all of the FPGA with a count up function.

- **config_up_pblock_inst_count_partial.bin**

  This is the partial bitstream for the count up module.

- **config_up_pblock_inst_count_partial_clear.bin**

  This is the clearing bitstream for the count up module.

In Ultrascale devices, the reconfigurable partition needs to be loaded with a clearing bitstream before charging the actual bitstream that is going to implement the right functionality.

The other files generated can be omitted as they are not relevant for what is coming next. The partial bit files needed here are the ones with the .bin extension because there is a step coming that will format these partial bit files to bin_for_icap files. This is a special format that is needed to perform DFX operations correctly.

To do this, issue the following commands :

```
source [get_property REPOSITORY [get_ipdefs *prc:1.3]]/xilinx/prc_v1_3/tcl/api.tcl

prc_v1_3::format_bin_for_icap -i ./bitstreams/config_count_up/config_up_pblock_inst_count_partial.bin

prc_v1_3::format_bin_for_icap -I ./bitstreams/config_count_up/config_up_pblock_inst_count_partial_clear.bin
```

*NB: The syntax of these commands can change depending on the version of Vivado used. For versions newer than 2019.1. Details are in the document **[DFXPG]** p.66.*

Two files are generated after these commands :

- **config_up_pblock_inst_count_partial.bin.bin_for_icap**
- **config_up_pblock_inst_count_partial_clear.bin.bin_for_icap**

  These are the two formatted partial bit files that will be fetched by the PRC to be sent to the icap and reprogram the FPGA dynamically. Close the project to continue.

```
close_project
```

Now the same thing needs to be done for the second configuration:

```
open_checkpoint implemented_checkpoints/config_count_down/top_routed.dcp

write_bitstream -force -bin_file bitstreams/config_count_down/config_down
```

Three bitstreams are generated again

- **config_down.bit**

  This is a power-up full configuration bitstream that will program all of the FPGA with a count down function.

- **config_down_pblock_inst_count_partial.bin**

  This is the partial bitstream for the count down module.

- **config_down_pblock_inst_count_partial_clear.bin**

  This is the clearing bitstream for the count down module.

Just like earlier:

```
source [get_property REPOSITORY [get_ipdefs *prc:1.3]]/xilinx/prc_v1_3/tcl/api.tcl
```

```
prc_v1_3::format_bin_for_icap -i ./bitstreams/config_count_down/config_down_pblock_inst_count_partial.bin

prc_v1_3::format_bin_for_icap -I ./bitstreams/config_count_down/config_down_pblock_inst_count_partial_clear.bin
```

Two files are generated after these commands :

- **config_down_pblock_inst_count_partial.bin.bin_for_icap**
- **config_down_pblock_inst_count_partial_clear.bin.bin_for_icap**

Now let’s generate a full bitstream with grey boxes and blanking bitstreams for the reconfigurable modules. The blanking bitstreams can be used to erase an existing configuration to save power and reduce the power consumption. Blanking bitstreams and clearing bitstreams are not the same thing.

```
open_checkpoint checkpoints/static_route_design.dcp

update_design -cell inst_count -buffer_ports

place_design

route_design

write_checkpoint -force implemented_checkpoints/config_grey_box/config_grey_box.dcp

write_bitstream -force -bin_file bitstreams/config_grey_box/config_grey_box

source [get_property REPOSITORY [get_ipdefs *prc:1.3]]/xilinx/prc_v1_3/tcl/api.tcl

prc_v1_3::format_bin_for_icap -i ./bitstreams/config_grey_box/config_grey_box_pblock_inst_count_partial.bin

prc_v1_3::format_bin_for_icap -i ./bitstreams/config_grey_box/config_grey_box_pblock_inst_count_partial_clear.bin

close_project
```

- **Partially reconfiguring the FPGA**

Now that the necessary bitstreams, full and partial, have been generated, Everything is ready to perform a DFX operation.

First, load the FPGA with a full configuration bit file, for example the count_up bit file, so that the design is up and running. The SEM controller automatically performs a system reset and goes into observation mode. Then load the partial bitstreams and clearing bitstreams into DMA buffers in the CPU memory using the **load_module.sh** script.

Now, three main steps need to be done:

- **Put the SEM controller in the idle state**
- **Configure the PRC registers and send a trigger**
- **Command the SEM controller to do a software reset**

Since the reconfigurable partition will be loaded with a new reconfigurable module, the SEM needs to be deactivated so that it doesn’t detect the change of modules as an error.

Then the PRC registers need to be configured correctly before launching a trigger, this mainly concerns the registers holding the sized and addresses of the necessary partial bitstreams to load the new reconfigurable module, this registers can be accessed in read and write mode as long as the virtual socket manager is in the shutdown state. Then the virtual socket manager needs to be commanded out of the shutdown state by writing a 1 to the control register. The next step is to send the trigger corresponding to the reconfigurable module that is wanted to be loaded. After a few milliseconds, the new module is loaded.

At last, the SEM controller is commanded to perform a software reset. Since the module has changed, a software reset is needed to recalculate the ECC values and perform error mitigation correctly.

6. Migration from UltraScale to UltraScale+

The previous design was demonstrated to successfully work on a Kintex UltraScale FPGA, but it can also work on an UltraScale+ FPGA, with some changes however. These changes include:

- **Signals and interfaces for the SEM controller**
- **Bitstreams needed to do a DFX operation**
- **Constraints file**

6.1. SEM controller changes

One small change is regarding the command_code signal that commands the behavior of the SEM controller, and the fecc_far signal that connects to the Frame ECC primitive. In an UltraScale device, the command_code signal is 40 bits wide, but in an UltraScale+ device it is 44 bits wide. The 4 added bits are related to error injection. A description of how error injection works is provided in the Xilinx Product Guide for the SEM controller **[SEMPG]**. For the fecc_far signal, the width is 26 but 27 for UltraScale+ devices.

This means that migrating from UltraScale to UltraScale+ implicates some changes at the RTL level. For the bus interface, the significant bits in the complementary code register would have to be changed and extended by four. The command_code signal that links the bus interface with the SEM controller too as well as the fecc_far signal.

In SSI UltraScale+ devices, there is a status_heartbeat signal for each SLR region. There is one SEM controller and one ICAP in total but one Frame ECC for each SLR. Which means that the connectivity would have to be adapted to this specific case, at the RTL level.

6.2. DFX controller changes

The DFX controller also needs to be adapted when migrating from an UltraScale device to an UltraScale+ device. The difference is that in UltraScale devices, before charging a partial bitstream to load a module, the controller needs to load a clearing bitstream corresponding to that partial bitstream. This clearing bitstream sets the partition for a proper loading of the new reconfigurable module, it is also placed in the CPU memory and is fetched by the controller, meaning that it also has an address at which it is put in the CPU memory and a size that need to be fed to the controller in order to fetch this clearing bitstream properly. Meanwhile this isn't the case for UltraScale+ devices, for which clearing bitstreams don't exist.

6.3. Constraints file

The constraints file used for this design is specific to the KCU105 board for a Kintex UltraScale FPGA. Upgrading to an UltraScale+ would require a new constraints file because of the fact that IO names would change and the mapping of these pins with the FPGA signals would have to be modified.

Conclusion

To conclude, DFX technology's feasibility was demonstrated on a KCU105 board, using different IPs and coded RTL logic that were subject to all of the DFX flow from RTL elaboration and synthesis to bitstream generation. This last step generates a full bitstream to first run the FPGA and then partial bitstreams that will need to be placed in the CPU external memory that it totally outside the KCU105 board, these partial bitstreams will then be fetched by the DFX controller through the PCIe connection when it is commanded to do so by writing triggers to the internal configuration registers of this controller. When fetched, these bitstreams are delivered to the reconfigurable partition through the ICAP.

The result in this design is seeing the user LEDs on the board change from performing a count up shifting to a count down shifting. For the count down function, the MSBs and LSBs of the LEDs inputs and the outputs of the counter were inverted, just to switch the position of the fast-switching bits so that the loading of a new reconfigurable module is more visible to the human eye.

This internship was an excellent opportunity to grasp the importance and greatness behind the roles of a digital design engineer. It has permitted me to understand the challenges that a design engineer can face during his work on a project and enabled me to learn some new good design habits.

Annex

6.1. Annex 1. C++ software to launch a DFX design

```
#include "pci_bar.h"

#include <unistd.h>

#include <string.h>


#define PCI_BASE_ADDR          0x00000000

#define BRIDGE_INFO            (PCI_BASE_ADDR + 0x130)

#define BRIDGE_STATUS          (PCI_BASE_ADDR + 0x134)

#define BUS_LOCATION_REG   (PCI_BASE_ADDR + 0x140)

#define PHY_STATUS_REG         (PCI_BASE_ADDR + 0x144)


#define PCICTL_BASE_ADDR          0x00000000 //Offset address for the PCI control interface

#define SEM_BASE_ADDR             0x00001000 //Offset address for the SEM bus interface

#define PRC_BASE_ADDR             0x00002000 //Offset address for the PRC configuration interface

#define SPCI_BASE_ADDR                0x04000000 //Offset address for the slave interface of the PCI


// Defining the addresses of the  PRC registers

#define PRC_CTL_STAT_REG                  (PRC_BASE_ADDR + 0x000) // Address for the control and status register

#define SWTRIGGER_REG                 (PRC_BASE_ADDR + 0x004) // Address for the SW_TRIGGER register

#define TRIGGER0_REG                 (PRC_BASE_ADDR + 0x040) // Address for the TRIGGER0 register

#define TRIGGER1_REG                 (PRC_BASE_ADDR + 0x044) // Address for the TRIGGER1 register

#define RM_BS_INDEX0_REG                 (PRC_BASE_ADDR + 0x080) // Address for the bitstream index register for RM 0

#define RM_CTL0_REG                          (PRC_BASE_ADDR + 0X084) // Address for the control information register of RM 0

#define RM_BS_INDEX1_REG                 (PRC_BASE_ADDR + 0x088) // Address for the bitstream index register for RM 1
```

#define RM_CTL1_REG (PRC_BASE_ADDR + 0X08C) // Address for the control information register of RM 1

#define BS_ID0_REG (PRC_BASE_ADDR + 0x0C0) // Address for the partial bitstream identifier register for RM 0

#define ADDRESS_PBS_UP_REG (PRC_BASE_ADDR + 0x0C4) // Address for the partial bitstream address register for RM 0

#define SIZE_PBS_UP_REG (PRC_BASE_ADDR + 0x0C8) // Address for the partial bitstream size register for RM 0

#define BS_ID1_REG (PRC_BASE_ADDR + 0x0D0) // Address for the clearing bitstream identifier register for RM 0

#define ADDRESS_CBS_UP_REG (PRC_BASE_ADDR + 0x0D4) // Address for the clearing bistream address register for RM 0

#define SIZE_CBS_UP_REG (PRC_BASE_ADDR + 0x0D8) // Address for the clearing bitstream size register for RM 0

#define BS_ID2_REG (PRC_BASE_ADDR + 0x0E0) // Address for the partial bitstream identifier register for RM 1

#define ADDRESS_PBS_DOWN_REG (PRC_BASE_ADDR + 0x0E4) // Address for the partial bitstream address register for RM 1

#define SIZE_PBS_DOWN_REG (PRC_BASE_ADDR + 0x0E8) // Address for the partial bitstream size register for RM 1

#define BS_ID3_REG (PRC_BASE_ADDR + 0x0F0) // Address for the clearing bitstream identifier register for RM 1

#define ADDRESS_CBS_DOWN_REG (PRC_BASE_ADDR + 0x0F4) // Address for the clearing bistream address register for RM 1

#define SIZE_CBS_DOWN_REG (PRC_BASE_ADDR + 0x0F8) // Address for the clearing bitstream size register for RM 1

// Defining the addresses of the SEM registers

#define STATUS_REG (SEM_BASE_ADDR + 0x000) // Address of the SEM status register

```
#define CMDCODE_REG             (SEM_BASE_ADDR + 0x004) // Address of the command code register

#define CCMDCODE_REG            (SEM_BASE_ADDR + 0x008) // Address of the complementary command
code register

#define CoCR_REG                (SEM_BASE_ADDR + 0x00C) // Address of the correction counter register

#define HCR_REG                 (SEM_BASE_ADDR + 0x010) // Address of the heartbeat counter register


// Defining the addresses of the PCI registers needed to set the AXI BAR

#define AXIBARU                 (PCICTL_BASE_ADDR + 0x208) // Address of the register that holds
the upper 32 bits of the 64 bit AXI to PCI BAR

#define AXIBARL                 (PCICTL_BASE_ADDR + 0x20C) // Address of the register that holds
the lower 32 bits of the 64 bit AXI to PCI BAR


// Defining bits for write command

#define ACT_VSM_BIT             0x1 // bit to activate the virtual socket manager of the PRC

#define SHTDWN_VSM_BIT          0x0 // bit to shutdown the virtual socket manager of the PRC

#define RM0TRIG_BIT             0x0 // bit to send TRIGGER0

#define RM1TRIG_BIT             0x1 // bit to send TRIGGER1


//Defining constants

#define IDLE_CMD                0xE0000000 // Command to put the SEM controller in the idle state

#define ERROR_INJ_CMD           0xC0000000 // Command to put the SEM controller in the error injection
state

#define OBSV_CMD                0xA0000000 // Command to put the SEM controller in the observation state

#define DET_ONLY_CMD            0xF0000000 // Command to put the SEM controller in the detect only
state

#define DIAG_SCAN_CMD           0xD0000000 // Command to put the SEM controller in the diagnostic scan
state

#define SOFT_RESET_CMD          0xB0000000 // Command to do a software reset of the SEM controller


#define PBSRM0_SIZE             0x000C41DC

#define CBSRM0_SIZE             0x0000AE38
```

```
#define PBSRM1_SIZE             0x000C41DC

#define CBSRM1_SIZE             0x0000AE38


#define ZERO                    0x00000000


#define FCU_BASE_ADDR           0x00011000


#define FPGA_PCI_VENDOR_ID      0x10F6

#define FPGA_PCI_DEVICE_ID      0x8177


#define MAX_BUFFERS             32


#define SIZE_WINDOW                     0x00FFFFFF


pci_bar* bar0;

struct dma_buffer {

        char name[1024];

        uint64_t start;

        uint64_t end;

        uint64_t size;

};

struct image_fmt {

        uint16_t width;

        uint16_t height;

        uint16_t bits;

};

struct dma_buffer bufs[MAX_BUFFERS] = {0};

int32_t dump_buffer_to_file(struct dma_buffer *dma_buf) {

        int fd, f, bytes = 0;
        int r = 0;
        char filename[1024];
        char devname[1024];
        unsigned char *tmp;
        int pg_size = getpagesize();
        sprintf(filename, "%s.bin", dma_buf->name);
        sprintf(devname, "/dev/%s", dma_buf->name);
        if ((tmp = (unsigned char *) malloc(pg_size)) == NULL) {
                printf("Error: %s malloc() failed: %s\n", __func__, strerror(errno));
                return -1;
        }
        if ((f = open(filename, (O_RDWR | O_CREAT | O_TRUNC), (S_IRUSR | S_IWUSR) )) == -1) {
                printf("Error: %s open(%s)\n", __func__, strerror(errno));
                free(tmp);
                return -1;
        }
        if ((fd = open (devname, O_RDWR | O_SYNC)) == -1) {
                printf("Error: %s open failed : %s\n", __func__, strerror(errno));
                goto failed;
        }
        printf("dump: %s to %s ... ", devname, filename);
        while ( (bytes = read(fd, tmp, pg_size)) ) {
                r = write(f, tmp, bytes);
                if (r < 0) {
                        printf("Error: %s write failed : %s\n", __func__, strerror(errno));
                        break;
                }
        }
```

```
        printf("%s\n", (r < 0) ? "failed" : "done");
        close(fd);
failed:
        close(f);
        if (tmp)
                free(tmp);

        return 0;
}
uint64_t get_dma_buffer_addr(int buffer_id) {
        char attr[1024];
        uint64_t phys_addr = 0;
        int fd;

        if (buffer_id >= MAX_BUFFERS)
                return 0;
        sprintf(attr, "/sys/class/udmabuf/udmabuf%d/phys_addr", buffer_id);
        if ((fd  = open(attr, O_RDONLY)) != -1) {
                read(fd, attr, 1024);
                sscanf(attr, "0x%016lx", &phys_addr);
                close(fd);
        }
        return phys_addr;
}
uint64_t get_dma_buffer_size(int buffer_id) {
        char attr[1024];
        uint64_t size = 0;
        int fd;
        if (buffer_id >= MAX_BUFFERS)
                return 0;

        sprintf(attr, "/sys/class/udmabuf/udmabuf%d/size", buffer_id);
        if ((fd  = open(attr, O_RDONLY)) != -1) {
                read(fd, attr, 1024);
                sscanf(attr, "%d", &size);
                close(fd);
        }
        return size;
}
int setup_dma_buffers(void) {
        int i, buffers;
        uint64_t addr = 0;

        for (i = 0, buffers = 0; i < MAX_BUFFERS; i++) {
                if ( (addr = get_dma_buffer_addr(i)) ) {
                        sprintf(bufs[buffers].name, "udmabuf%d", i);
                        bufs[buffers].start = addr;
                        bufs[buffers].size = get_dma_buffer_size(i);
                        bufs[buffers].end = bufs[buffers].start + bufs[buffers].size;
                        buffers++;
                }
        }
        return buffers;
}

void dump_dma_buffer_addr(int buffers) {
        int i;
        for (i = 0; i < buffers; i++) {
                printf("dma_buf[%d]: %s start: 0x%lx ; end: 0x%lx (length: %d)\n",
                                i, bufs[i].name, bufs[i].start, bufs[i].end, bufs[i].size);
        }

        return;
}

int
main(int argc, char* argv[])
{
        int i;
        int sw_buffers;
        uint32_t data = 0;
        /* setup DMA buffers */
        sw_buffers = setup_dma_buffers();
        if (sw_buffers == 0) {
                printf("No DMA buffer available.\n");
                printf("Please load module:\n");
                exit(1);
        }

        /*Calculating the addresses to be fed to the PRC registers to ensure a proper fetching of the right
bitstreams*/
        uint32_t dmabuff_addr[sw_buffers];
        for (i = 0; i < sw_buffers; i++) {
                dmabuff_addr[i] = (bufs[i].start & SIZE_WINDOW) + SPCI_BASE_ADDR;
        }
```

```
                    /*
                     dmabuff_addr[i] is the AXI address as seen from the DMA in the PRC
                     bufs[i].start is the PCI address as seen from the CPU side
                     SIZE_WINDOW is the size of the window allocated in the AXI interconnect for access to the
PCIe bridge slave interface
                     SPCI_BASE_ADDR is the offset address for the slave interface of the PCIe bridge slave
interface as configured in the AXI

                     This operation is a calculation of the address that needs to be fed to the PRC DMA in order
to fetch the correct bitstreams needed. Address translation is about removing and adding offsets
                     so the first operation should be taking the bufs[i].start address and removing the offset that
does not coincide with the bits needed to represent the full address window.
                     This is equivalent to performing a logic & with the size of the window. The addition is
straightforward.
                     */
        printf("\nTotal allocated DMA buffers: %d\n", sw_buffers);
        if (sw_buffers) {
                dump_dma_buffer_addr(sw_buffers);
        }

        /*Setting up the PCI bar*/
        bar0 = new pci_bar(FPGA_PCI_VENDOR_ID,FPGA_PCI_DEVICE_ID,0);

        /*Dummy read to verifiy if the PCIe is enabled*/
        bar0->pci_read(PRC_CTL_STAT_REG, &data);

        /*Enabling the PCI in case it was forgotten*/
        if (data == 0xFFFFFFFF) {
                printf("Please enable PCIe slot:\n");
                printf("using this command : setpci -d %04x:%04x COMMAND=0x06\n",
                                FPGA_PCI_VENDOR_ID, FPGA_PCI_DEVICE_ID);
            exit(1);
        }
        /*Commanding the SEM to an idle state*/
        printf("\nReading SEM status\n");
        bar0->pci_read(STATUS_REG, &data);
        printf("Writing a Command to the SEM controller \n");
        bar0->pci_write(CCMDCODE_REG,(uint32_t) ZERO);          // Writing in the complementary command code register is only relevant when in error injection mode
        bar0->pci_write(CMDCODE_REG,(uint32_t) IDLE_CMD);       // Writing the command that will force the SEM controller to go into an idle state
        printf("Reading SEM status\n");
        bar0->pci_read(STATUS_REG, &data);

        /*Setting the AXI BAR in the PCI control register to ensure proper address translation operations*/
        printf("\nWriting and reading the AXIBAR\n");
        bar0->pci_write(AXIBARU,(uint32_t)0x00000008);          //Writing in this register sets the upper 32 bits of the 64-bit address signal that serves as the AXI BAR
        bar0->pci_write(AXIBARL,(uint32_t)0x1f000000);          //Writing in this register sets the lower 32 remaining bits
        bar0->pci_read(AXIBARU, &data);
        bar0->pci_read(AXIBARL, &data);

        /*Configuring the control registers of the PRC to set up DFX operations*/
        printf("\nWriting and reading the bitstream addresses and sizes\n");
        bar0->pci_write(PRC_CTL_STAT_REG,(uint32_t)SHTDWN_VSM_BIT);     //Writing a 0 in this register puts the virtual socket manager in shutdown mode
        bar0->pci_write(ADDRESS_PBS_UP_REG,dmabuff_addr[0]);            //Writing the address of the partial bitstream for the count up module
        bar0->pci_write(SIZE_PBS_UP_REG,(uint32_t)PBSRM0_SIZE);         //Writing the size of the partial bitstream for the count up module
        bar0->pci_write(ADDRESS_CBS_UP_REG,dmabuff_addr[1]);            //Writing the address of the clearing bistream for the count up module
```

bar0->pci_write(SIZE_CBS_UP_REG,(uint32_t)CBSRM0_SIZE); //Writing the size of the clearing bitstream for the count up module

bar0->pci_write(ADDRESS_PBS_DOWN_REG,dmabuff_addr[2]); //Writing the address of the partial bitstream for the count down module

bar0->pci_write(SIZE_PBS_DOWN_REG,(uint32_t)PBSRM1_SIZE); //Writing the size of the partial bitstream for the count down module

bar0->pci_write(ADDRESS_CBS_DOWN_REG,dmabuff_addr[3]); //Writing the address of the clearing bitstream for the count down module

bar0->pci_write(SIZE_CBS_DOWN_REG,(uint32_t)CBSRM1_SIZE); //Writing the size of the clearing bitstream for the count down module

bar0->pci_read(BS_ID0_REG, &data);

bar0->pci_read(BS_ID1_REG, &data);

bar0->pci_read(BS_ID2_REG, &data);

bar0->pci_read(BS_ID3_REG, &data);

printf("\nReading the PRC state\n");

bar0->pci_read(PRC_CTL_STAT_REG, &data); //Reading to check if the virtual socket manager is in shutdown state

printf("\nReading the trigger registers,trigger 0 then trigger 1\n");

bar0->pci_read(TRIGGER0_REG, &data); // This register holds the ID of the reconfigurable module that will be loaded if trigger0 is sent

bar0->pci_read(TRIGGER1_REG, &data); // This register holds the ID of the reconfigurable module that will be loaded if trigger1 is sent

printf("\nReading the reconfigurable module information registers\n");

bar0->pci_read(RM_BS_INDEX0_REG, &data); // This register holds the ID of the bitstream and the clearing bitstream for the first reconfigurable module

bar0->pci_read(RM_BS_INDEX1_REG, &data); // This register holds the ID of the bitstream and the clearing bitstream for the second reconfigurable module

/*Setting up the PRC to launch a trigger*/

printf("\nWriting to get the PRC out of the shutdown\n");

bar0->pci_write(PRC_CTL_STAT_REG,(uint32_t)ACT_VSM_BIT); // Writing a 1 in this register gets the virtual socket manager out of the shutdown state

bar0->pci_read(PRC_CTL_STAT_REG, &data); // Reading this register to check if the virtual socket manager got out of the shutdown state

printf("\nWriting the trigger\n");

bar0->pci_write(SWTRIGGER_REG,(uint32_t)RM1TRIG_BIT); // Writing a 1 in this register loads the count down module, writing a 0 loads the count up module

bar0->pci_read(SWTRIGGER_REG, &data); // Reading this register identifies the trigger that is sent, if the bit number 32 is high it means that a trigger is pending

do{

bar0->pci_read(PRC_CTL_STAT_REG, &data);

}while((data & 0xf) != 0x7); // this loop enables the tracking of the status of the controller until it finishes loading a reconfigurable module

/*Commanding the SEM to do a software reset*/ /*Reminder: it is necessary to do a software reset after a DFX operation so that the SEM controller can work properly(Check the Conceptual design doc)*/

printf("\nWriting a Command to the SEM controller \n");

bar0->pci_write(CCMDCODE_REG,(uint32_t)ZERO);

bar0->pci_write(CMDCODE_REG,(uint32_t)SOFT_RESET_CMD); // Writing this command forces the SEM controller to perform a software reset

printf("Reading SEM status\n");

bar0->pci_read(STATUS_REG, &data);

usleep(100000);

printf("\nReading SEM status\n");

bar0->pci_read(STATUS_REG, &data);

```
        /*Dumping the contents of the dma buffers to binary files*/

        for (int buf = 0; buf < sw_buffers; buf++) {

                /* dump buffer to binary file */

                dump_buffer_to_file(&bufs[buf]);

        }


        delete bar0;

        return 0;

}
```

6.2. Annex 2. TCL script to launch the DFX flow

```
################################################### DFX flow script ####################################################


##################### Setting of the board and part variables ######################################################


set part "xcku040-ffva1156-2-e" ;# Setting the right FPGA device.


set board "kcu105" ;# Setting the board.


##################### Synthesizing the count up reconfigurable module ###########################################################


create_project -in_memory -part $part ;# Creation of an in-memory project.


add_files ./sources/count_up.vhd ;# Adding the VHDL file for the count up module into the project.


set_property top count [current_fileset] ;# Setting the top file for this project, it is necessary to do a synthesis.
```

synth_design -mode out_of_context ;# Doing an out of context synthesis of the module. It is absolutely necessary that the type of this synthesis would be an out of context synthesis.

write_checkpoint ./synthesized_checkpoints/RM_count_up/checkpoint_count_up.dcp ;# Writing a checkpoint to save the synthesized module.

close_project

##################### Synthesizing the count down reconfigurable module ###

create_project -in_memory -part $part

add_files ./sources/count_down.vhd

set_property top count [current_fileset]

synth_design -mode out_of_context

write_checkpoint ./synthesized_checkpoints/RM_count_down/checkpoint_count_down.dcp

close_project

##################### Adding the files necessary for the first configuration ##

create_project -in_memory -part $part

add_files ./synthesized_checkpoints/top/checkpoint_top.dcp ;# Adding the synthesized checkpoint of the static top logic.

add_files ./constraints/KCU_cnstrn.xdc ;# Adding the constraints file.

set_property USED_IN {implementation} [get_files ./constraints/KCU_cnstrn.xdc]

add_files ./synthesized_checkpoints/RM_count_up/checkpoint_count_up.dcp ;# Adding the synthesized checkpoint of the count up module.

set_property SCOPED_TO_CELLS {inst_count} [get_files ./synthesized_checkpoints/RM_count_up/checkpoint_count_up.dcp] ;# The SCOPED_TO_CELLS property makes sure that this module is assigned to the proper instance in the top file.

link_design -mode default -reconfig_partitions {inst_count} -part $part -top fpga_top ;# This command links the whole design together.

write_checkpoint ./synthesized_checkpoints/linked_design/top_link_up.dcp

###################### Creating a pblock to define the reconfigurable region ######################################

create_pblock pblock_inst_count ;# Creation of the pblock.

resize_pblock pblock_inst_count -add {SLICE_X3Y257:SLICE_X19Y285 DSP48E2_X0Y104:DSP48E2_X2Y113 RAMB18_X0Y104:RAMB18_X1Y113 RAMB36_X0Y52:RAMB36_X1Y56} ;# Resizing the pblock.

add_cells_to_pblock pblock_inst_count [get_cells [list inst_count]] -clear_locs ;# Linking the pblock to the inst_count instance.

```
###################### Design rule checks focused on partial reconfiguration ########################################

create_drc_ruledeck ruledeck_1

add_drc_checks -ruledeck ruledeck_1 [get_drc_checks {HDPRA-62 HDPRA-60 HDPRA-58 HDPRA-57 HDPRA-56 HDPRA-55 HDPRA-54 HDPRA-53 HDPRA-52 HDPRA-51 HDPRA-21 HDPR-43 HDPR-20 HDPR-88 HDPR-41 HDPR-30 HDPR-96 HDPR-95 HDPR-94 HDPR-93 HDPR-92 HDPR-91 HDPR-90 HDPR-87 HDPR-86 HDPR-85 HDPR-84 HDPR-83 HDPR-74 HDPR-73 HDPR-72 HDPR-71 HDPR-70 HDPR-69 HDPR-68 HDPR-67 HDPR-66 HDPR-65 HDPR-64 HDPR-63 HDPR-62 HDPR-61 HDPR-60 HDPR-59 HDPR-58 HDPR-57 HDPR-54 HDPR-50 HDPR-49 HDPR-48 HDPR-47 HDPR-46 HDPR-44 HDPR-42 HDPR-38 HDPR-37 HDPR-35 HDPR-34 HDPR-33 HDPR-32 HDPR-29 HDPR-28 HDPR-25 HDPR-23 HDPR-22 HDPR-18 HDPR-17 HDPR-16 HDPR-14 HDPR-13 HDPR-12 HDPR-11 HDPR-6 HDPR-5 HDPR-4 HDPR-3 HDPR-2 HDPR-1}]

report_drc -name drc_1 -ruledecks {ruledeck_1}

delete_drc_ruledeck ruledeck_1

###################### Writing all of the constraints generated up until now ########################################

write_xdc ./constraints/top_all.xdc ;# This command writes out all of the constraints into one file.

###################### Implementation of the first configuration #########################################################
```

opt_design

place_design

route_design

##################### Creation of implemented checkpoints and reports ##

write_checkpoint -force implemented_checkpoints/config_count_up/top_routed.dcp

report_utilization -file reports/config_count_up/top_utilization.rpt

report_timing_summary -file reports/config_count_up/top_timing_summary.rpt

write_checkpoint -force -cell inst_count implemented_checkpoints/config_count_up/count_up_routed.dcp

##################### Clearing the implementation of the reconfigurable partition ###############################

update_design -cell inst_count -black_box ;# This command clears the inst_count instance and removes the implementation results associated to it.

lock_design -level routing ;# This command locks the actual routing results to make sure it is not changed in the future.

```
write_checkpoint -force checkpoints/static_route_design.dcp ;# Saving the placing and routing results of the static top design.

close_project



##################### Adding the files of the second configuration ##############################################

create_project -in_memory -part $part

add_files ./checkpoints/static_route_design.dcp ;# this command adds the checkpoint saved earlier of the static placed and routed design.

add_files ./synthesized_checkpoints/RM_count_down/checkpoint_count_down.dcp

set_property SCOPED_TO_CELLS {inst_count} [get_files ./synthesized_checkpoints/RM_count_down/checkpoint_count_down.dcp]

link_design -mode default -reconfig_partitions {inst_count} -part $part -top fpga_top



##################### Implementation of the second configuration ################################################

opt_design

place_design
```

```
route_design



###################### Writing checkpoints and reports
##############################################################

write_checkpoint -force implemented_checkpoints/config_count_down/top_routed.dcp

report_utilization -file reports/config_count_down/top_utilization.rpt

report_timing_summary -file reports/config_count_down/top_timing_summary.rpt

write_checkpoint -force -cell inst_count
implemented_checkpoints/config_count_down/count_down_routed.dcp



###################### Verification of the static routing results of the two configurations
#######################

pr_verify implemented_checkpoints/config_count_up/top_routed.dcp
implemented_checkpoints/config_count_down/top_routed.dcp ;# This command reports the conformity
of the two static place and route results of the two configurations, which should be exactly the same.

close_project
```

##################### Bitstream generation for the count up configuration ##

open_checkpoint implemented_checkpoints/config_count_up/top_routed.dcp

write_bitstream -force -bin_file bitstreams/config_count_up/config_up ;# This command writes a bin file for the top design and a partial bitstream along with its clearing bitstream for the reconfigurable partition.

#write_debug_probes -force -file bitstreams/config_count_up/config_up.ltx ;# This command would be useful if you have debug probes in your design, if not you can leave it commented.

source [get_property REPOSITORY [get_ipdefs *prc:1.3]]/xilinx/prc_v1_3/tcl/api.tcl ;# This command allows for transition into the PRC API mode where you can issue tcl commands to interact with the PRC post-implementation.

prc_v1_3::format_bin_for_icap -bs 1 -i ./bitstreams/config_count_up/config_up_pblock_inst_count_partial.bin ;# This command formats a partial bitstream to be recognizable by an ICAP primitive.

prc_v1_3::format_bin_for_icap -bs 1 -i ./bitstreams/config_count_up/config_up_pblock_inst_count_partial_clear.bin ;# This command formats a partial bitstream to be recognizable by an ICAP primitive.

close_project

##################### Bitstream generation for the count down configuration #####################################

open_checkpoint implemented_checkpoints/config_count_down/top_routed.dcp

```
write_bitstream -force -bin_file bitstreams/config_count_down/config_down

source [get_property REPOSITORY [get_ipdefs *prc:1.3]]/xilinx/prc_v1_3/tcl/api.tcl

prc_v1_3::format_bin_for_icap -bs 1 -i
./bitstreams/config_count_down/config_down_pblock_inst_count_partial.bin

prc_v1_3::format_bin_for_icap -bs 1 -i
./bitstreams/config_count_down/config_down_pblock_inst_count_partial_clear.bin

close_project




##################### Bitstream generation for the grey box configuration
########################################

open_checkpoint checkpoints/static_route_design.dcp

update_design -cell inst_count -buffer_ports

place_design

route_design

write_checkpoint -force implemented_checkpoints/config_grey_box/config_grey_box.dcp

write_bitstream -force -bin_file bitstreams/config_grey_box/config_grey_box
```

```
source [get_property REPOSITORY [get_ipdefs *prc:1.3]]/xilinx/prc_v1_3/tcl/api.tcl

prc_v1_3::format_bin_for_icap -bs 1 -i ./bitstreams/config_grey_box/config_grey_box_pblock_inst_count_partial.bin

prc_v1_3::format_bin_for_icap -bs 1 -i ./bitstreams/config_grey_box/config_grey_box_pblock_inst_count_partial_clear.bin

close_project
```

References

Reference	Title
[DFXCPG]	Dynamic Function eXchange controller v1.0 LogiCORE IP Product Guide Reference : https://www.xilinx.com/support/documentation/ip_documentation/dfx_controller/v1_0/pg374-dfx-controller.pdf
[SEMCPG]	Ultrascale Architecture Soft Error Mitigation Controller v3.1 LogiCORE IP Product Guide Reference : https://www.xilinx.com/support/documentation/ip_documentation/sem_ultra/v3_1/pg187-ultrascale-sem.pdf
[AXIPG]	AXI Interconnect v2.1 LogiCORE IP Product Guide Reference : https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf
[AXIPCIPG]	AXI bridge for PCI Express Gen3 Subsystem v3.0 Product Guide Reference : https://www.xilinx.com/support/documentation/ip_documentation/axi_pcie3/v3_0/pg194-axi-bridge-pcie-gen3.pdf
[DFXDPG]	Dynamic Function eXchange Decoupler v1.0 LogiCORE IP Product Guide Reference : https://www.xilinx.com/support/documentation/ip_documentation/dfx_decoupler/v1_0/pg375-dfx-decoupler.pdf
[AXIAPBPG]	AXI to APB bridge v3.0 LogiCORE IP Product Guide Reference : https://www.xilinx.com/support/documentation/ip_documentation/axi_apb_bridge/v3_0/pg073-axi-apb-bridge.pdf
[DFXT]	Vivado Design Suite Tutorial Dynamic Function eXchange Reference : https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug947-vivado-partial-reconfiguration-tutorial.pdf
[APBP]	AMBA 3 APB protocol v1.0 specification Reference : http://web.eecs.umich.edu/~prabal/teaching/eecs373-f12/readings/ARM_AMBA3_APB.pdf
[DFXUG]	Vivado Design Suite User Guide Dynamic Function eXchange Reference : https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf
[KCU105UG]	KCU105 board user guide Reference : https://www.xilinx.com/support/documentation/boards_and_kits/kcu105/ug917-kcu105-eval-bd.pdf
[KCU105EK]	KCU105 board evaluation kit Reference: https://www.xilinx.com/products/boards-and-kits/kcu105.html
[MMSI]	Mercury Systems Website Reference: https://www.mrcy.com/company-overview/facts-at-a-glance/