POLITECNICO DI TORINO
Department of Electronic Engineering

Master degree in Electronic Engineering - Electronic systems

Design of a high-speed data buffer based on DDR memories for real-time processing of hyperspectral data and its implementation in a radiation-tolerant FPGA

Supervisors:

Guido Masera - Politecnico di Torino

Davide Fiorini - Leonardo S.p.A.

Candidate:

Laura Chisciotti - s274728

Accademic year
2021-2022

Abstract

In the space environment, during the image acquisition, the hyperspectral payloads need to process a data flow which could swing between hundreds of Mbit/sec and some Gbit/sec.
In this scenario, one of the key aspects is to find the correct dimension of the memory buffers, based on access speed and dimension, in order to optimize the payload and guarantee the time allocation of the whole sequence of tasks which have to be executed between the rough data sending by the detector and the moment in which the processed data are sent to the platform.
The hyperspectral payload called CHIME (Copernicus Hyperspectral Imaging Mission for the Environment), implemented by the Leonardo S.p.A. for the European Space Agency (ESA), needs to access a high-speed buffer concurrently in order to memorize the data coming from the detector, execute a spectral editing/binning, fix bad pixels, apply some coefficients to obtain a linear calibration of their radiometric value and read again the data grouping them in homogeneous packages before shipping them towards the mass memory of the satellite. All the tasks have to be executed in real-time with approximately 8 Gbit/sec input data flow.
The intention of this work of thesis is to design the high-speed data buffer based on DDR3 memory banks, which has to be projected, thus, it could be subsequently implemented in a radiation-tolerant FPGA, in particular an FPGA RT4G150 of the Microchip.
This high-speed data buffer is a controller implemented in VHDL, which has to make available to the other blocks two different interfaces, through which it is possible the writing and the reading: a Direct Access (DA) interface and a First In First Out (FIFO) one.
Moreover, on the other side, the controller interfaces with a hardware macrocell FDDR of the FPGA Microchip RT4G150, which has the aim to manage, in an efficient way, the writing and the reading of the data burst in the DDR3 memory banks.
Therefore, since the FDDR block could be accessed only through an AXI interface, another of the high-speed data buffer tasks is to translate the signals deriving from the Direct Access interface or the FIFO one into signals of the AXI interface.
Another task of this controller is to manage the concurrent accesses to the DDR3 memory banks, in fact, inside the controller there is also an arbiter block, which, based on the Round Robin algorithm, coordinates the concurrent readings and concurrent writings, when two users want to access one through the Direct Access interface and one through the FIFO interface both to read or write in the same source.
All the project has to follow the ECSS-Q-ST-60-02C standard which is one of the European Cooperation for Space Standardization.

Keywords: AXI interface, direct access, FIFO, arbiter, FDDR, FPGA RT4G150

Contents

List of Figures **6**

List of Tables **9**

1 Introduction **10**

2 Development standard in space environment **11**
2.1 Standard ECSS-Q-ST-60-02C 11
2.1.1 Defination phase 12
2.1.2 Architectural design phase 13
2.1.3 Detailed design/layout phase 14
2.1.4 Prototype Implementation phase 15
2.2 FPGA development outputs 15
2.2.1 Requirement Specification document (ARS) 15
2.2.2 Feasibility and Risk analysis (FRA) 16
2.2.3 DataSheet (DS) 16
2.2.4 Design Verification Document (DVD) 17

3 FPGA space devices **18**
3.1 RT PolarFire FPGAs 19
3.2 RTG4 FPGAs 20
3.3 RTAX-S/SL 20
3.4 RT ProASIC3 21
3.5 RTSX-SU 22

4 RTG4 overview **23**
4.1 Key components 24
4.1.1 Board power up 24
4.1.2 Current measurement 24
4.1.3 Memory Interface 25
4.1.4 SerDes Interface 26
4.1.5 Programming Interface 26
4.1.6 System Reset Interface 26
4.1.7 Clock Oscillator 27
4.2 SDRAM Memory 27
4.2.1 Device operation - SDRAM as a state machine 28
4.2.2 SDRAM core scheduling 30
4.2.3 Page Hit, Page Miss, Page Empty 31
4.3 FDDR (DDR controller) 32

5 AXI interface **34**
5.1 AXI Protocol . . . 34
5.1.1 AXI Write Transaction . . . 35
5.1.2 AXI Read Transaction . . . 37
5.1.3 Implementation of an AXI Master Interface on the User Logic . . . 38
5.1.4 Implementation of an AXI Slave Interface on the User Logic . . . 38

6 High speed data buffer design **39**
6.1 User requirements (URD) . . . 42
6.1.1 Introduction . . . 42
6.1.2 Functions . . . 42
6.1.3 Interfaces . . . 44
6.1.4 Error management . . . 45
6.1.5 Performances . . . 46
6.2 FPGA requirements (ARS) . . . 46
6.2.1 Functional requirements . . . 47
6.2.2 Logic architecture . . . 47
6.2.3 Direct access write block . . . 49
6.2.4 Direct access read block . . . 52
6.2.5 FIFO access write block . . . 54
6.2.6 FIFO access read block . . . 56
6.2.7 Arbiter . . . 58

7 HDL implementation **59**
7.1 Direct Access block . . . 59
7.1.1 Write Direct Access block . . . 60
7.1.2 Read Direct Access block . . . 63
7.1.3 Write and Read Direct Access blocks assembling . . . 66
7.2 FIFO block . . . 78
7.2.1 Write FIFO block . . . 78
7.2.2 Read FIFO block . . . 82
7.2.3 Write and Read FIFO blocks assembling with evaluator . . . 85
7.3 Arbiter block . . . 87
7.3.1 Write Arbiter block . . . 88
7.3.2 Read Arbiter block . . . 90
7.3.3 Write and Read Arbiter blocks assembling . . . 92

8 Simulation HDL code **95**
8.1 Direct Access block simulation . . . 95
8.1.1 Code Coverage . . . 102
8.2 FIFO block simulation . . . 104
8.2.1 Code Coverage . . . 109
8.3 Final structure with arbiter block simulation . . . 111
8.3.1 Code Coverage . . . 119

9 Hardware implementation and deployment **122**
9.1 Hardware tests . . . 128
9.1.1 Test 1: writing and reading in memory without errors through Direct Access interface . . . 128
9.2 Test 2: DA_WR_ERR2 error during writing through Direct Access interface 131

9.3 Test 3: DA_RD_ERR2 error during reading through Direct Access interface 132
9.4 Test 4: writing and reading in memory without errors through FIFO interface 133
9.5 Test 5: FIFO_WR_ERR2 error during writing through FIFO interface . 135
9.6 Test 6: FIFO_RD_ERR2 error during reading through FIFO interface . . 135
9.7 Test 7: FIFO_RD_ERR1 error during reading through FIFO interface . . 136

10 Conclusion and future work **137**

11 References **138**

List of Figures

2.1 Development flow and reviews 12

3.1 RT PolarFire FPGA structure 19
3.2 RTG4 FPGA structure 20

4.1 RTG4 Development Board 23
4.2 Board power up structure 24
4.3 Core power measurement structure 24
4.4 Memory Interface structure 25
4.5 Programming Interface structure 26
4.6 System Reset Interface structure 27
4.7 50 MHz and 100 MHz Clock Oscillators 27
4.8 SDRAM state transition diagram 28
4.9 Memory read and write operations 31
4.10 System-Level FDDR Block Diagram 32

5.1 AXI Write Flow 35
5.2 AXI Read Flow 35
5.3 Write Transaction Timing Diagram with a Burst Length of 4 36
5.4 Read Transaction Timing Diagram with a Burst Length of 4 37

6.1 Sketch of block architecture 43
6.2 Sub-blocks architecture 48

7.1 Finite State Machine of Write Direct Access block 60
7.2 Direct Access write block 62
7.3 Finite State Machine of Read Direct Access block 63
7.4 Direct Access read block 65
7.5 Write and Read Direct Access AXI block 66
7.6 Structure with AXI master and AXI slave 67
7.7 Zoom of 50 MHz, PLL and system reset blocks 67
7.8 Clock Conditioning Circuitry PLL 1:1 68
7.9 Basic option Clock Conditioning Circuitry PLL 68
7.10 Advanced Option Clock Conditioning Circuitry PLL 69
7.11 PLL options Clock Conditioning Circuitry PLL 69
7.12 Zoom of Reset synchronizers 70
7.13 Reset Synchronizer scheme 70
7.14 Reset Synchronizer timing diagram 71
7.15 Zoom Direct Access AXI master block 71
7.16 Zoom AXI switch 72

7.17 Zoom FDDR AXI slave . . . 72
7.18 General section of FDDR configuration . . . 73
7.19 Memory initialization section of FDDR configuration . . . 73
7.20 Memory timing section of FDDR configuration . . . 74
7.21 Complete structure RTG4 (Bench), with emulator and DDR3 memory blocks 74
7.22 Zoom of the AXI block . . . 75
7.23 Zoom of the emulator block . . . 76
7.24 Zoom of the DDR3 memory bank . . . 77
7.25 Zoom of the reset generator . . . 77
7.26 Finite State Machine of Write FIFO block . . . 78
7.27 FIFO write block . . . 81
7.28 Finite State Machine of Read FIFO block . . . 83
7.29 FIFO read block . . . 84
7.30 Write and Read FIFO AXI block with evaluator . . . 86
7.31 Zoom of the FIFO AXI master block . . . 86
7.32 Zoom of the emulator and AXI block . . . 87
7.33 Write Arbiter structure . . . 88
7.34 Finite State Machine of Write Arbiter . . . 89
7.35 Read Arbiter structure . . . 90
7.36 Finite State Machine of Read Arbiter . . . 92
7.37 Direct Access and FIFO masters with arbiters structure . . . 93
7.38 Zoom of the AXI master block . . . 93
7.39 Zoom of the emulator and AXI block . . . 94

8.1 Zoom of LOCK signal in Direct Access block simulation . . . 95
8.2 Zoom of INIT_DONE and reset_sync_1 signal in Direct Access block simulation . . . 96
8.3 Writing Direct Access block simulation . . . 97
8.4 Reading Direct Access block simulation . . . 97
8.5 Writing Direct Access block simulation with two errors . . . 98
8.6 Read Direct Access block simulation with two errors . . . 99
8.7 Writing of the concurrent writing and reading in Direct Access block simulation . . . 99
8.8 Reading of the concurrent writing and reading in Direct Access block simulation . . . 100
8.9 Simulation to check the right writing during concurrent writing and reading in Direct Access block . . . 101
8.10 Simulation to check the third error in the writing of the write Direct Access block . . . 101
8.11 Simulation to check the third error in the reading of the read Direct Access block . . . 102
8.12 Initialization sequence of the FIFO block . . . 104
8.13 Simulation of a correct writing in FIFO block . . . 105
8.14 Simulation of a correct reading in FIFO block . . . 106
8.15 Simulation of writing with FIFO_WE high for a number of clock cycles different from 8 . . . 106
8.16 Simulation of reading with FIFO_RQ higher than 1 clock cycle . . . 107
8.17 Simulation of writing with FIFO_WE high when FIFO_WRDY is low . . . 107
8.18 Simulation of reading with FIFO_RQ high when FIFO_RRDY in low . . . 108
8.19 Simulation of writing when FULL FIFO . . . 109

8.20 Simulation of reading when EMPTY FIFO 109
8.21 Simulation of simultaneous writing from FIFO and Direct Access interfaces 113
8.22 Simulation of Arbiter behaviour for simultaneous writing from FIFO and Direct Access interfaces 113
8.23 Simulation of concurrent writing from FIFO and Direct Access interfaces . 114
8.24 Simulation of Arbiter behaviour for concurrent writing from FIFO and Direct Access interfaces 115
8.25 Simulation of concurrent writing from FIFO and Direct Access interfaces . 115
8.26 Simulation of Arbiter behaviour for concurrent writing from FIFO and Direct Access interfaces 116
8.27 Simulation of concurrent reading from FIFO and Direct Access interfaces . 116
8.28 Simulation of Arbiter behaviour for concurrent reading from FIFO and Direct Access interfaces 117
8.29 Simulation of concurrent reading from FIFO and Direct Access interfaces . 118
8.30 Simulation of Arbiter behaviour for concurrent reading from FIFO and Direct Access interfaces 118

9.1 Edge detector structure 122
9.2 Structure for hardware test 123
9.3 Zoom of the Direct Access structure added for the hardware test 124
9.4 Zoom of the FIFO structure added for the hardware test 125
9.5 PLL structure for hardware test 125
9.6 Laboratory setup for the tests 126
9.7 RTG4 Development Kit for the test 127
9.8 FMC XM105 Xilinx board for the test 127
9.9 First Direct Access writing without errors 129
9.10 First Direct Access reading without errors 129
9.11 Second Direct Access writing without errors 130
9.12 Second Direct Access reading without errors 130
9.13 Third Direct Access writing without errors 131
9.14 Third Direct Access reading without errors 131
9.15 Writing with DA_WR_ERR2 error 132
9.16 Reading with DA_WR_ERR2 error 132
9.17 First FIFO writing without errors 133
9.18 First FIFO reading without errors 134
9.19 Second FIFO writing without errors 134
9.20 Second FIFO reading without errors 134
9.21 Writing with FIFO_WR_ERR2 error 135
9.22 Reading with FIFO_WR_ERR2 error 135
9.23 Reading with FIFO_WR_ERR1 error 136

List of Tables

6.1 AXI interface signals from slave to master 40
6.2 AXI interface signals from master to slave 41

Chapter 1

Introduction

The space is a more complex environment, in which what could be seen as an easy problem on the Earth, there becomes an hard problem, since the quantity of radiation is much higher and the Single Event Effects (SEEs) are present. The SEEs are caused by a single, energetic particle, and can take on many forms, like Single Event Upsets (SEUs), which are soft errors, and non-destructive, that normally appear as transient pulses in logic or support circuitry, or as bitflips in memory cells or registers; and Single Event Latchup (SEL), which is an hard error potentially destructive, that appears as a high operating current, above device specifications and it must be cleared by a power reset.[12]
Moreover, in the space, the quantity of power that could be used is limited and there is to take into account that when the device is sent to the space, it is subjected to some shocks, which could damage the device leading to its not correct function.
In fact, the FPGAs used in space are implemented in order to be radiation tolerant and to be resistant to the SEE problems.
In particular, the FPGA used in this thesis project is the RT4G150 Microchip FPGA, which is a rad-tolerant (RT) FPGA, thus, it is able to mitigate the radiation effects. Moreover, this FPGA is equipped with a Single Error Correct Double Error Detect (SECDED), so, if an electron hits the FPGA and changes a bit, the RTG4 is able to identify this error and correct it.
This FPGA is used in an European Space Agency (ESA) project, which has been assigned to the Leonardo S.p.A. . This project is called CHIME, which means Copernicus Hyperspectral Imaging Mission for the Environment, and it is an hyperspectral payload.
In particular, the thesis project is focused on a VHDL block, which is an high-speed data buffer, that is part of the Video Acquisition Unit (VAU) of CHIME.
In order to respect the space rules, the implementation of this block has to follow the ECSS-Q-ST-60-02C Standard, which is one of the European Cooperation for Space Standardization, which is composed of specific steps and at the end of each of these steps there are specific documentations, which have to pass a review to each step.
Thus, following these rules, the thesis project has been implemented in order to obtain a high-speed data buffer to which it is possible to access through either a direct access interface or a FIFO interface, and that converts these signals in AXI interface signals, in order to communicate with the next block, which is a controller of the DDR3 banks of memory (FDDR), in which the data are written or from which the data are read. Moreover, this block has to arbitrate concurrent accesses both in writing and in reading using the Round Robin algorithm.

Chapter 2

Development standard in space environment

Nowadays, when a space device is developed, there are standard rules that have to be followed and respected since in the space environment there are different conditions to take into account with respect to the Earth environment, such as a high quantity of radiation and extreme temperatures. In particular, for the Field Programmable Gate Arrays (FPGAs) the standard which has to be followed is the ECSS-Q-ST-60-02C.

2.1 Standard ECSS-Q-ST-60-02C

The Standard ECSS-Q-ST-60-02C is one of the series of ECSS (European Cooperation for Space Standardization) Standards intended to be applied together for the management, engineering and product assurance in space projects and applications.
This Standard defines the requirements for the user development of digital, analog and mixed analog-digital custom designed integrated circuits, such as ASICs and FPGAs.
The user development starts by setting initial requirements and ends with the validation and release of the prototype device.
The aim of this Standard is to ensure that the custom designed components, which will be used in space, meet their requirements in terms of functionality, quality, reliability, schedule and cost.
Each stage of the development activity has to be consolidated before starting the subsequent one and it is ensured through the support of appropriate planning and risk management.[1]
The general development flow is based on 5 phases:

- Defination phase,
- Architectural design,
- Detailed design/ Layout,
- Prototype implementation,
- Flight Model (FM) production.

At the end of each phase there is a formal review:

- System Requirement Review (SRR) after the defination phase,

- Preliminary Design Review (PDR) after the architectural design,
- Critical Design Review (CDR) after the detailed design/ layout,
- Qualification Review/ Acceptance Review (QR/AR) after the prototype implementation.

This flow is better visualized in Figure 2.1.

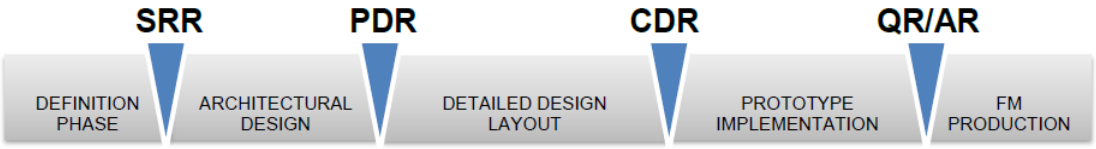

Figure 2.1: Development flow and reviews

2.1.1 Defination phase

This phase has the aim to ensure that all relevant system configuration and characteristics, and all issues imposing requirements on the device are considered, that the definition status of the collected requirements is settled without any ambiguity and that all necessary resources for the design activities are available.
The Definition phase is composed of three documents: ADP, ARS and FRA.
The purpose of the ADP document, called Developed plan, is to implement the proposed development strategy by identifying all phases of the ASIC and FPGA development with the major activities therein, the project external interface and constraints, the design flow, resources, the allocation of responsibilities, outputs to be produced and a schedule with milestones, decision points, type and number of design reviews.
The main target of the Definition phase is to establish a Requirements Specification document (ARS), which contains the requirement that has to be used during design entry and verification. At this point of the development, this document should contain all the requirements except for some related to the flight model implementation.
Moreover, during this phase, feasibility and risk of the design are summarized in a document called Feasibility and Risk Analysis (FRA). This document covers all the aspects of the project, such as FPGA technology maturity and suitability, device resources and performances, design uncertainties and testability, in order to declare feasibilities, giving an estimation of risks and when and how to mitigate them. To determine the risk, two values are taken into account: severity (S) and likelihood (L), which corresponds to the risk probability. Through these two parameters, the Risk Index (RI) is obtained:

$$RI = S \cdot L \tag{2.1}$$

The ranges of the risk index are:

- Red: RI $\geq$ 15, when the risk is high and cannot be acceptable;
- Yellow: 15 $>$ RI $>$ 4, in this case, an evaluation of the intermediate risk has to be done to determine if the risk could be acceptable;
- Green: RI $\leq$ 4, when the risk is acceptable.

Moreover, a risk domain is determined, in order to understand which area the risk affects. The domains that could be considered are: p=performance, s=schedule and c=cost.
The definition phase ends with a review (**SRR**) of these documents, that are issued and checked for consistency and completeness against higher level applicable documents.
Thus, the outputs of the Definition phase, presented at the System Requirement Review (SRR), are the following:

- Development plan (ADP)
- Feasibility and Risk Analysis (FRA)
- Requirement Specification (ARS)
- System Requirement Review minutes of meeting (MoM_SRR)

2.1.2 Architectural design phase

The Architecture Design phase is composed of HDL development (firmware HDL), HDL simulation (bench HDL), datasheet (DS) and design verification document (DVD).
In this phase, the architecture of the chip and the baseline choices made during the Definition phase are frozen, verified and documented, updating the Requirement Specification (ARS) document.
Moreover, selections for the implementations of the design, such as FPGA family and technology, are made or confirmed, updating the Feasibility and Risk Analysis (FRA) document.
The architectural design document, which will be part of the datasheet document, reports the design down to block identifications and detailed interfaces between them, relevant functions and interactions, and outlines a suitable clocking and reset scheme.
At the end of this phase, a first release of HDL code is issued and verified according to the verification plan, which is part of the design verification document (DVD), containing strategies about how each function and requirement are verified in simulation. It contains also detailed test procedures and a requirements coverage matrix, in which on rows there are the requirements and on columns, there is the test procedure.
The HDL simulation suite is composed of several test-benches, which provide a 100% coverage against requirements and a target of 100% coverage at HDL level. In the case in which some part of the code cannot be covered, it has to be identified, justified and documented. Test-benches for formal verification are developed for top-level design, thus these could be run again at any stage of development, such as functional, post-synthesis, post-layout and even on hardware.
Therefore, the verification report is composed of the simulation output files, stored together with relevant test benches sources files. This phase is concluded only when all the verifications are passed.
The DVD is connected to the ARS since every simulation reported in the DVD has to satisfy a requirement contained in the ARS.
The architectural design phase ends with a Preliminary Design Review (**PDR**) of the issued documents and an update of problems.
Thus, the outputs of the Architectural Design phase, presented at the PDR, should be the following:

- Update of Requirement Specification (ARS)
- Update of Feasibility and Risk Analysis (FRA)

- Data Sheet (DS)
- Design Verification Document (DVD)
- Verification Report
- HDL source code
- HDL test- bench
- Preliminary Design Review minutes of meeting (MoM_PDR)

2.1.3 Detailed design/layout phase

In this phase, the HDL function design in frozen and detailed design implementation begins.
For what concerns the FPGA qualification model (QM) and the FPGA flight model(FM) implementations, the synthesis on QM and FM target devices is performed and the post-synthesis netlist is verified running again the test-benches simulation.
At this point, the obtained resulting timing parameters are verified and documented
Thus, the design entry report is composed of a collection of report files generated by dedicated software tools. An extract of these files will be part of the datasheet (DS) document, such as the report of resource usage, number of flip-flops and so on.
In this phase, the design verification document (DVD) is updated, since it is composed of the simulation output files, according to the same philosophy of the previous phase, but generated by simulating the technological netlist both for the FPGA qualification model and for the FPGA flight model.
Then, the Validation Plan (VP) and the procedure are prepared, in order to be able to validate the FPGA design in the qualification model, with the relevant requirements coverage matrix. Thus, in the VP document, the check of the FPGA requirements (ARS) when the device is placed on the board is reported.
The Detailed Design phase ends with a Critical Design Review (**CDR**) of the issued documents and an update of problems.
Thus, the outputs of the Detailed Design phase, presented at the CDR, should be the following:

- Update of Requirement Specification (ARS)
- Update of Feasibility and Risk Analysis (FRA)
- Update of Data Sheet (DS)
- Update of Design Verification Document (DVD)
- Validation Plan and procedure (VP)
- Verification Report
- HDL source code
- HDL test bench
- FPGA QM programming file for the qualification model
- FPGA FM programming file for the flight model

- FPGA FM netlist
- Critical Design Review minutes of meeting (MoM_CDR)

2.1.4 Prototype Implementation phase

In this phase, the FPGA prototypes are programmed.
Once programmed, the prototypes are tested on the qualification model and a formal Design Validation is performed.
Therefore, the datasheet (DS) document is updated with parameters resulting from prototype testing.
The Prototype Implementation phase ends with a Qualification Review/Acceptance Review (QR/AR) of the issued documents and FPGA flight model programming is authorized.
The validation is performed on the qualification model based on the FPGA Validation Plan and the procedure. Thus, a report is issued.
The outputs of the Prototype Implementation phase, presented at the QR/AR, should be the following:

- Update of Data Sheet (DS)
- FPGA prototype data package
- Validation Report of FPGA prototypes on the qualification model

2.2 FPGA development outputs

This section is dedicated to a summary of the content of the main output documents requested by ECSS-Q-ST-60-02C, in order to explain better what is the aim of each document.

2.2.1 Requirement Specification document (ARS)

The requirements are documented in form of Requirement Specification covering the following aspect:

- System requirements:
  - Overall system partitioning, system configurations and operating modes and, when applicable, for each block:
    - Functional requirements
    - Interfaces of the block to the system or other blocks, communication protocols to external devices, including memory and registers mapping
    - Applicable algorithms
    - Error handling
    - Test modes: system and device tests, on ground and in flight
  - Simulation coverage required
- Timing requirements:
  - Timing demonstration of interfaces and critical signals

- Requirements dependencies:
  - Requirements should be traced upwards to higher-level document
- Electrical and other implementation requirements:
  - Device pinout
  - Electrical constraints
  - Operating frequency range
  - Power-up and initialization state
  - Reset and power cycling requirements
  - Required fault coverage

2.2.2 Feasibility and Risk analysis (FRA)

The purpose of the FRA document is to provide an assessment of the design feasibility, as specified in requirement documents, and to assess and control the risks involved in the development. The aspects that this document has to cover are:

- Evaluation of selected device technology
  - Maturity of technology and related processes and tools
  - Experience and familiarity with engineering resources with technology and tools
  - Impacts on development flow and plan
- Design evaluation
  - Requirements maturity and stability
  - Design breakdown in main blocks, and for each block
    - Estimate number of pins (interface blocks)
    - Estimate number of logic gates (flip-flops, memories)
    - Estimate necessary frequencies and data throughput
    - Perform early assessment of testability
  - Check overall needed resources against what provided by selected devices
  - Identify critical blocks/functions (for resources or performance requirements) and provide a strategy for risk control and mitigation

2.2.3 DataSheet (DS)

The data sheet document is composed of different sections with data collected from different phases of the design. At Preliminary Design Review (PDR) the architectural section is provided, with the detailed architectural report and a parametric timing analysis laid down. While, at Critical Design Review (CDR) the design detailed description is added, with reports from detailed design activities and preliminary timing parameters. Finally, at Qualification Review/Acceptance Review (QR/AR), the timing analysis is completed and the pinout is updated with actual FPGA flight model data and electrical parameters. The DS contents are the following:

- The first part contains the system architecture description with a summary of the functionality, architecture definition, a block diagram and short list of features for each block.

- Subsequently, a detailed report about each block implementation, including all interface signals between them, follows.
- The detailed design implementation section contains all the data related to the final flight model design:
  - Pinout constraint (placement, direction, strength, slew rate)
  - Synthesis and layout constraint
  - Synthesis and layout report
  - Timing report composed by timing parameters and worst-case timing analysis when applicable.
- All characteristics and limitations introduced during the design are described, such as detailed interface descriptions, register definitions, memory maps and so on. The full functionality and all operating modes are specified in detail.
- All the electrical data are specified, together with their relevant applicable conditions.

2.2.4 Design Verification Document (DVD)

The Design Verification document demonstrates how the logic design in verified. Verification means that each requirement implementation is checked against the requirement itself.
The DVD document is composed of the following contents:

- Verification strategy and plan section (VP) explains the overall strategy for the design verification.
- The definition of the setup in term of software for logic simulation and architecture of the test benches.
- The test procedures in detail, which explain how each requirement is verified in the relevant test-bench simulation of the behavioral model. Target simulation coverage for requirements is 100%, with exceptions identified and justified.
- The expected results for each test procedure.
- The traceability matrix between requirements and tests.

The results of test procedures simulation are in the form of simulation reports files and when necessary text report files assert all requirements successful verification and graphic waveforms.

Chapter 3

FPGA space devices

FPGA, together with ASIC, is the key technology in the development of space missions and perform the hearts of the data processing system of satellites.
The Field Programmable Gate Array is very complex and dense integrated circuit used to contain control and data processing functions. The complexity can be defined by the number of gates and the package number of pins, in fact, today, space FPGAs could have several million gates and packages with more than 1500 pins.
Moreover, the integrated circuits are of capital importance in order to achieve the necessary miniaturisation and performance levels that today and future space system demand. However, there is to take into account that the field programmable integrated circuits implementing specific functions are always one of the most critical microelectronics elements inside the space system, in fact, even thought the FPGA follows a strict and quality manufacturing process, there are some causes that can procedure their failure:

- **Design mistake** : some nominal or corner cases never simulated;
- **Manufacturing problem** : silicon wafer defects, operator error, poor or inefficient error screening and so on;
- **System environment** : out of specification use;
- **Aging effects** : electro migration, channel hot carriers and so on.

Furthermore, the FPGAs used for space applications need to be more resistant because of the space environment effects, which are the vibration and mechanical shock, the extreme temperatures, the contamination, the single-event-upset, which causes a latch-up effect, and radiation effects.
The **radiation effects** are the main concern for FPGA use in space because they could bring temporary or permanent integrated circuit malfunctions with the risk of mission failure or loss and these issues could not be repaired in space.
This is the reason why the FPGA technologies for space application have to follow special design process and are implemented with countermeasures to strength the protection against space radiation effects. These complex integrated circuits are required to pass very strict and severe tests and simulations in order to be qualified for space applications.
Some of the most important FPGAs used in the space environment are produced by the Microsemi Corporation, which is a wholly owned subsidiary of Microchip Technology and offers a comprehensive portfolio of semiconductor and system solutions for communications, defence & security, aerospace and industrial markets.
The radiation-tolerant (RT) FPGAs produced by Microsemi are:

- **RT PolarFire FPGAs**
- **RTG4 FPGAs**
- **RTAX-S/SL FPGAs**
- **RT ProASIC3 FPGAs**
- **RTSX-SU FPGAs**

3.1 RT PolarFire FPGAs

The RT PolarFire FPGA structure is reported in Figure 3.1. It is composed of 481000 logic elements, 33 Mbits of embedded SRAM, 1480 DSP blocks and 24 lanes of 10 Gbps transceivers.

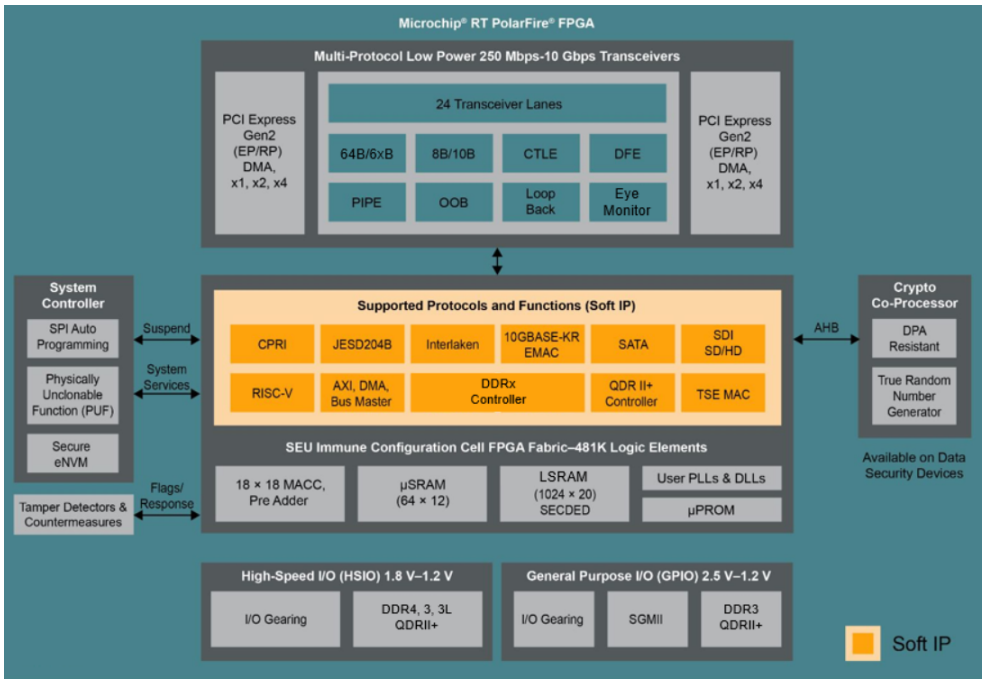

Figure 3.1: RT PolarFire FPGA structure

About the power, RT PolarFire uses low-power SONOS configuration switches embedded in a power-efficient architecture. This FPGA provides a total power savings of 40% to 50% relative to comparable SRAM FPGAs. This power saving is reflected in a major cost-of-ownership saving, as results in a simpler and less expensive power supply design and a reduced heat output results in simpler and less expensive thermal management.
The SONOS configuration switches used in RT PolarFire have been shown to be robust to more than 100kRad of total dose exposure, indicating their suitability for the vast majority of earth-orbiting satellites and for many deep-space missions.
Moreover, the SONOS configuration switches have been submitted to many rounds of heavy-ion single event tests and these have demonstrated an absence of configuration upset problems, unlike the SRAM FPGAs which do experience configuration upsets in space and require additional components in order to mitigate this phenomenon.
Thus, the robust nature of the RT Polar Fire configuration switches deletes significant

costs, power consumption and system overhead associated with configuration scrubbing and repair which is needed with SRAM FPGAs.[5]

3.2 RTG4 FPGAs

The RTG4 FPGA integrates the Microchip's fourth-generation Flash-based FPGA fabric high-performance serialization/deserialization transceivers on a single chip, while it maintains resistance to radiation-induced configuration upsets in the harshest radiation environments, making them excellent options for use in Low Earth Orbit (LEO), Medium Earth Orbit (MEO), Geostationary Equatorial Orbit (GEO), Highly Elliptical Orbit (HEO) and deep-space flight applications. This is why this device is used in the space flight, but also in other environments such as high-altitude aviation, medical electronics and nuclear power plant control.
An ideal scheme of an RTG4 FPGA is reported in Figure 3.2. It is manufactured on a low-power 65 nm process with substantial reliability heritage and it is immune to radiation, induced changes in configuration due to the robustness of the flash cells used to connect and configure logic resources and routing tracks. In this type of FPGA there is not the need to execute a background scrubbing or a reconfiguration of the FPGA to mitigate changes in configuration due to radiation effects, since the data errors due to radiation are mitigated by hardwired single event upset (SEU) resistant flip-flops in the logic cells and math blocks. While, the Single Error Correct Double Error Detect (SECDED) protection is optional for the embedded SRAM and the DDR memory controllers and this means that if a one-bit error is detected, it will be corrected and the errors of more than one bit are detected only and not corrected.[4]

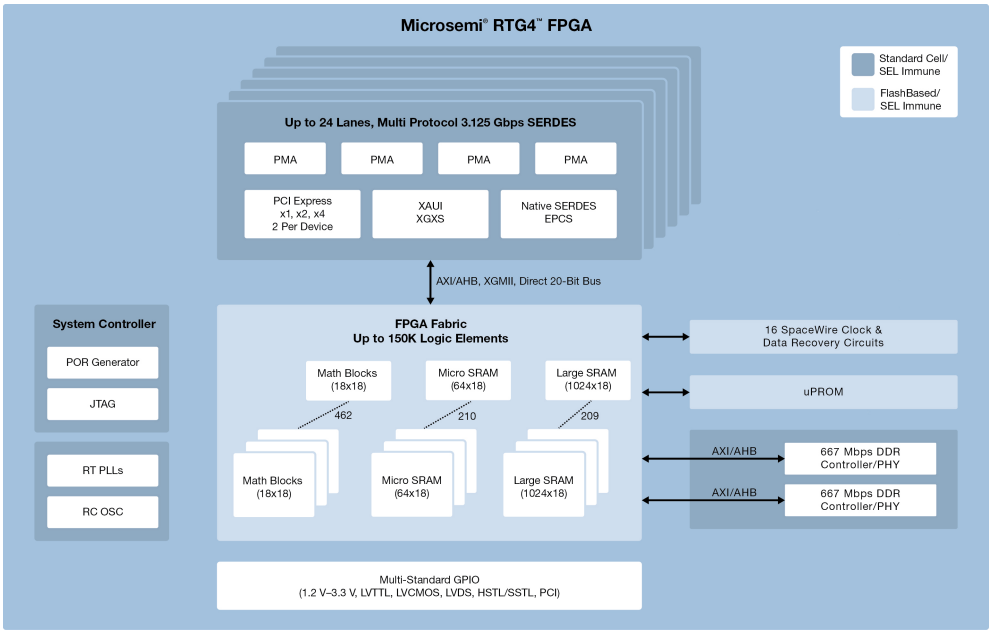

Figure 3.2: RTG4 FPGA structure

3.3 RTAX-S/SL

Starting from the RTAX-S radiation tolerant FPGA, this offers industry-leading advantages for designers of space-flight systems. The features which make the RTAX-S the

FPGA of choice for space designers are the high performance and low-power consumption, the true single-chip form factor and the live-at-power-up operation.
Moreover, for space applications which have a need for a lower standby current, another type of RTAX has been implemented, called RTAX-SL, that is characterized by a low-power grade option that has half of the standby current at worst-case conditions.
Both these FPGAs offer high performance at densities of up to 4 million equivalent system gates and 840 user I/Os for space-based applications.
Furthermore, the RTAX-S and RTAX-SL offer for space applications features SEU-hardened flip-flops implemented without any user interventions and the benefits of user-implemented triple module redundancy (TMR) without associated overhead.
Thus, RTAX-S and RTAX-SL offer higher density, higher performance, and more features than previous generations of Microsemi radiation-tolerant FPGAs. SEU-hardened flip-flops use built-in TMR, so these do not require user intervention and do not consume additional programmable logic gates for hardware implementation or host CPU machine cycles for software implementation. SEU-hardened flip-flops are not sensitive to place-and-route locations, thus preserving their SEU immunity.
A big advantage of these FPGA is that the RTAX-S/SL family has hot-swap and cold-sparing capabilities, which enable a device to be turned off for minimal power consumption during long space missions and activated only when functionality is required for mission completion.[8]

3.4 RT ProASIC3

The RT ProASIC3 FPGA is the first to offer a Radiation-Tolerant (RT), reprogrammable, nonvolatile logic integration vehicle to the designers of space-flight hardware. This type of FPGA is intended for low-power space applications requiring up to 350MHz operation up to 3 million system gates.
Unlike all other Microsemi's radiation-tolerant, space-flight FPGAs, which use antifuse programming technology, the devices in the RT ProASIC3 family use flash cells to store configuration information. Thus, positive or negative charge stored on floating-gate transistors is used to hold pass transistors in either the "on" or "off" states, thereby opening or closing connections between routing tracks and logic resources. This use of flash-based interconnects present some unique opportunities and advantages to designers of space-flight electronic hardware. These opportunities are the following:

- **The flash cells are reprogrammable**, thus, the designer could change the design of the FPGA without removing the FPGA from the board, making prototyping easier. Moreover, it also allows last-minute design change and code update to provide maximum design flexibility.

- **The flash cells are nonvolatile** and this means that flash-based FPGAs are standalone devices which do not require the provision of external code-storage devices, unlike SRAM-based FPGAs. This minimizes the board space used, and has an associated saving in mass.

- **RT ProASIC3 FPGAs are operating almost at the instant of power-up**, which is another advantage of the nonvolatility of the flash programming cells. There is no boot sequence required, as in SRAM-based FPGAs which need to download their configuration code from an external storage device.

- **The flash cells do not exhibit single-event upsets (SEUs) in the presence of heavy ion radiation**, therefore no triple-chip redundancy to mitigate configuration upsets is required, unlike SRAM FPGAs.[6]

3.5 RTSX-SU

The RTSX-SU radiation tolerant FPGA is specifically designed for enhanced radiation performance. Even in this case, this FPGA is characterized by SEU-hardened D-type flip-flops that offer the benefits of Triple Module Redundancy (TMR) without the associated overhead.
The RTSX-SU is a unique product for space applications since it is manufactured using 0.25 μm technology at the United Microelectronics Corporation (UMC) facility in Taiwan. Moreover, this FPGA offers levels of radiation survivability far in excess of typical CMOS devices. The Microsemi's RTSX-SU architecture has been designed to improve upset and total-dose performance in radiation environments with several enhancements, such as SEU-hardened flip-flops, wider clock lines and stronger clock drivers. Therefore, the RTSX-SU architecture is designed in a sea-of-modules structure, in which the entire floor of the FPGA is covered with a grid of logic modules with virtually no chip area lost to interconnect elements or routing.[9]

Chapter 4

RTG4 overview

The FPGA used in this thesis project is an RTG4, more, in particular, an RT4G150 of the Microchip. This type of FPGA is fully supported by a design software provided by Microsemi, called Libero System-on-chip (SoC), which guides the user through the FPGA design flow and provides seamless design tool integration, as well as project, data file and log file management.
While, about the development kit for the RTG4, the Microsemi supplies designers with an evaluation and development platform for applications such as data transmission, serial connectivity, bus interface and high-speed designs using the RTG4 devices. The development board features an RT4G150 device offering 151824 logic elements in a ceramic package with 1657 pins.
Moreover, it includes two 1GB Double Data Rate 3 (DDR3) memories and 1 GB of SPI flash memories. The board has also several standard and advanced peripherals, such as PCIe x4 edge connector, two FMC connectors for using several off-the-shelf daughter cards, USB, Philips interintegrated circuit (I^2C), gigabit Ethernet port, serial peripheral interface (SPI) and UART.[10]
The RTG4 Development Board is reported in Figure 4.1.

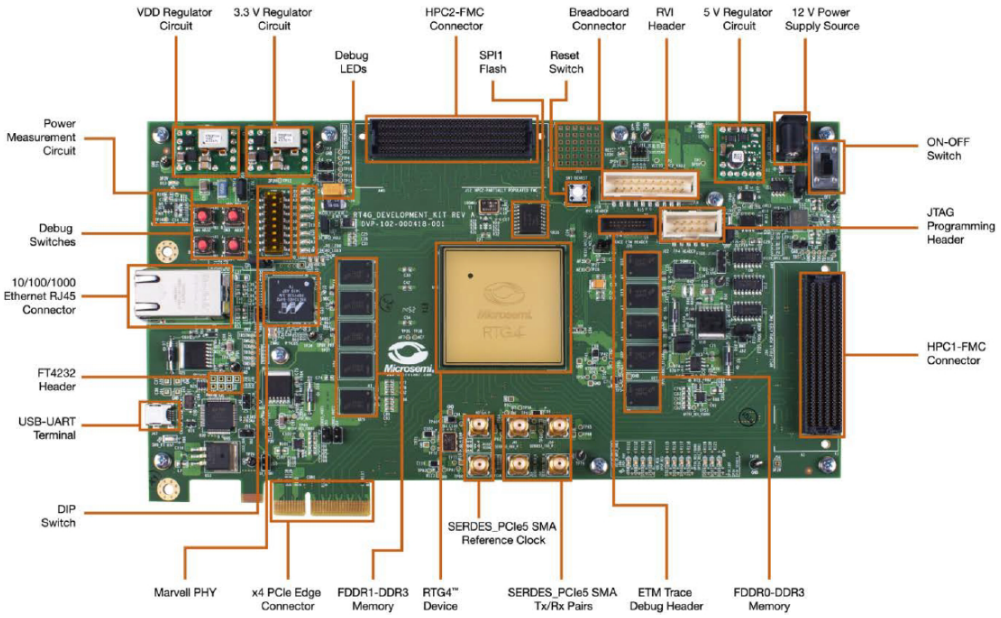

Figure 4.1: RTG4 Development Board

4.1 Key components

The RT4G150 device inside the RTG4 Development Kit is surrounded of key component interfaces, which are better explained in this section.

4.1.1 Board power up

In order to turn on the RTG4 Development Board a 12 V external DC jack, called 12P0V_Ext, is used, as shown in Figure 4.2.

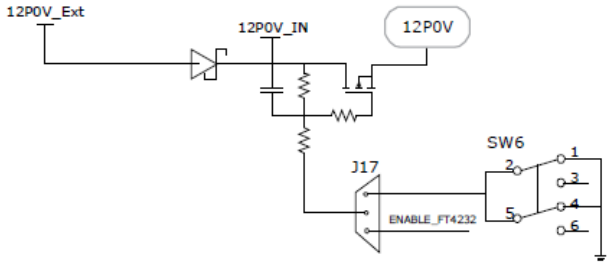

Figure 4.2: Board power up structure

Thus, to power up the board, two are the things which have to be executed, i.e. connect the 12 V power supply brick to the J17 in order to supply power to the board and then switch on the SW6 power supply switch.

4.1.2 Current measurement

In case of applications that require current measurement, the circuit reported in Figure 4.3 is used, which is composed of an high-precision operational amplifier circuitry (U5), with a gain of 5, through which the output voltage is measured from the TP16 test point.

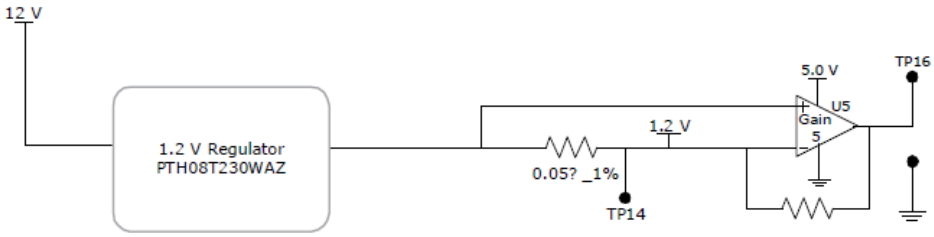

Figure 4.3: Core power measurement structure

Moreover, the steps to how to measure the core power are the following:

- measure the output voltage at the test point T16;
- compute the current through this formula:

$$I = \frac{V_{out}}{5 \cdot 0.05} = V_{out} \cdot 4 \tag{4.1}$$

where 5 is the gain of the operational amplifier (U5) and 0.05 is the current sense resistor value in ohms;

- compute the core power consumed:

$$P = V \cdot I = V(TP14) \cdot V(TP16) \cdot 4 \tag{4.2}$$

  where **V(TP14)** is the voltage measured at test point 14 and **V(TP16)**· **4** is the current **I** obtained at the previous step.

4.1.3 Memory Interface

This FPG4 is provided with banks of memory both on the east side and on the west side of the RTG4 chip. These banks of memory are managed by 2 controllers, as could be observed in Figure 4.4, called FDDR_E, which coordinates the banks of memory on the east side, and FDDR_W, which takes control of the banks of memory on the west side. To each FDDR, four chips of 256 MB DDR3 memory as flexible volatile memory for storing user data are assigned. In addition to the DDR3 memories, there is also, for each FDDR, a SECDED chip with the mansion of Error Correction Code (ECC), which enables the single-error correction and double-error detection (SECDED).
Moreover, the features of the memory interface are the following:

- Type: MT41K256M8: 32 Meg × 8 × 8 banks
- Density: 256 MB
- Clock rate: 333 MHz
- Data rate: DDR3, 666 MHz
- Total capacity: 1 GB from four chips

These components will be described deeper in the next sections since these are some of the more important items used in the thesis project.

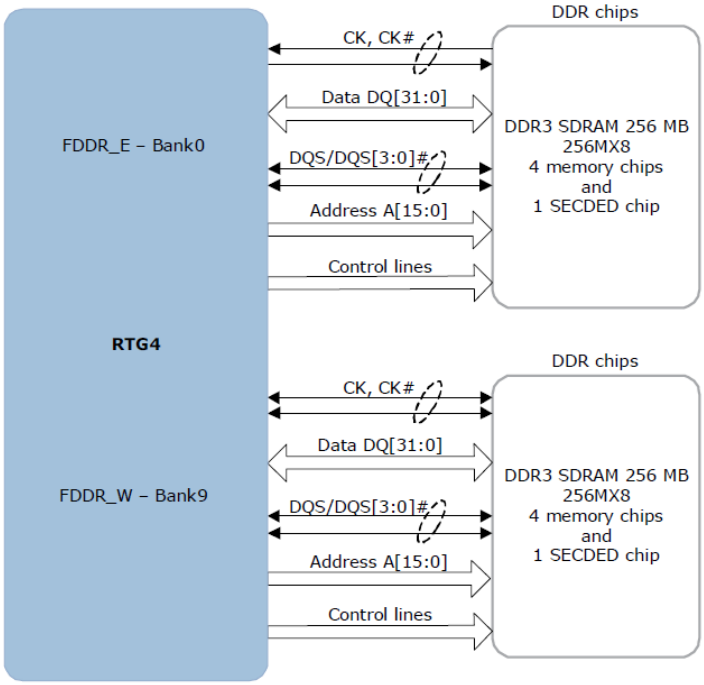

Figure 4.4: Memory Interface structure

4.1.4 SerDes Interface

The RT4G150 FPGA device on the RTG4 Development Kit has 24 SerDes (Serial-Deserial) lanes. Therefore, the SerDes blocks can be accessed using different connectors: PCIe edge connector, high-speed SMA connectors and on-board FMC connectors.
Thus, starting from the SERDES PCIe interface inside the FPGA, this interface is directly routed to the PCIe connector. Here, the reference clock is directly routed from the PCIe connector and optionally from the 100 MHz differential clock source.
Then, the SERDES1, SERDES2, SERDES3 and SERDES4 interfaces are routed to the FMC connector, which routes the reference clock of these four SERDES.

4.1.5 Programming Interface

RTG4 FPGAs support multiple programming interfaces and these can address a wide range of platform requirements. An RTG4 device can be programmed through the JTAG and SPI interfaces. Moreover, the dedicated programming SPI port can operate in SPI slave mode.
The programming interface of the RTG4 Development Board is reported in the figure below.

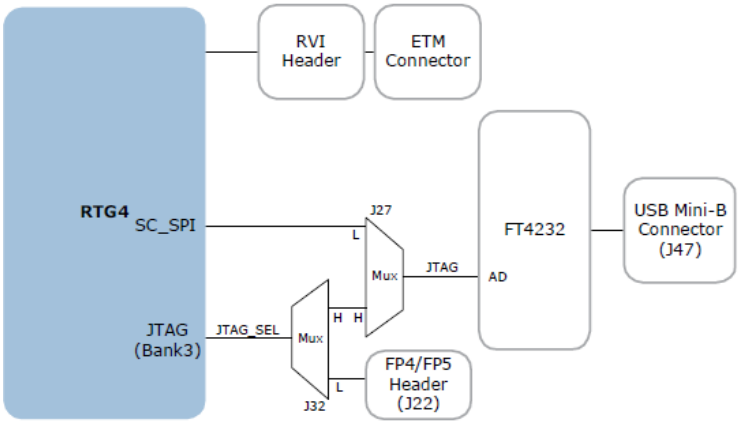

Figure 4.5: Programming Interface structure

4.1.6 System Reset Interface

The reset signal is signed in Figure 4.6 as G4M_RSTB active-low signal, which could be generated by the SW7 push-button switch, U35 chip (DS1818), or U22 chip (TPS3808G09). Therefore, the DEVRST_n is an input-only reset pad that allows assertion of a full reset to the chip at any time.
Analysing the U35 and U36 blocks, the DS1818 (U35) has the aim to monitors the status of the power supply (V_{cc}), in fact, when an out-of-tolerance condition is detected, an internal power fail signal is generated, which forces reset to the active state. When V_{cc} returns to an in-tolerance condition, the reset signal is kept in the active state for approximately 150 ns to allow the power supply and processor to stabilize.
While the TPS3808G09DBVR (U36) device monitors the voltage at the VDD_REG terminal. If the voltage at this terminal sense-drops below the threshold voltage of 0.9 V, the G4M_RSTB signal is asserted.

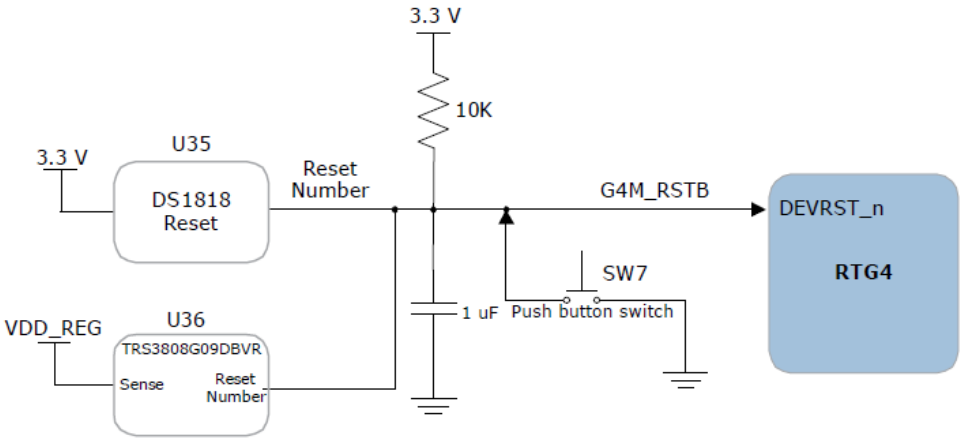

Figure 4.6: System Reset Interface structure

4.1.7 Clock Oscillator

A 50 MHz LVCMOS clock oscillator with an accuracy of ±50 ppm is available on the board, as shown in Figure 4.7. This clock oscillator is connected to the FPGA fabric to provide a system reference clock.
Moreover, this FPGA gives the opportunity to configure an on-chip RTG4 PLL to generate a wide range of high-precision clock frequencies.
In addition, a 100 MHz LVDS clock oscillator operating at 3.3 V with an accuracy of ±50 ppm is available on the board. This clock oscillator is connected to the FPGA fabric through the AB37 and AB36 pins.

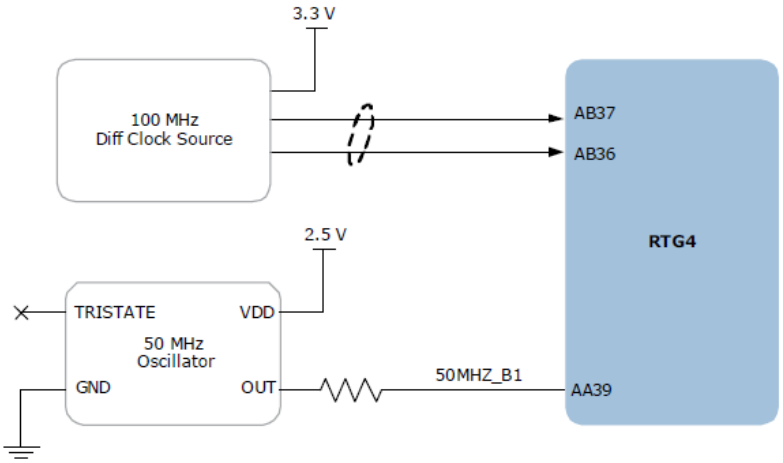

Figure 4.7: 50 MHz and 100 MHz Clock Oscillators

4.2 SDRAM Memory

One of the most important parts of the thesis project is the banks of SDRAM DDR3 memory at the east part and the west part with respect to the RTG4 chip. These are where the data are written and read during the usage of the FPGA.
In general, the Synchronous Dynamic Random Access Memory (SDRAM) is made up of multiple arrays of single-bit storage sites arranged in a two-dimensional lattice struc-

ture formed by the intersection of individual rows (Word Lines) and columns (Bit Lines). These grid-like structures, called banks, provide an expandable memory space allowing the host control process and other system components with direct access to the main system memory to temporarily write and read data to and from a centralized storage location.
When associated in groups of two (DDR), four (DDR2) or eight (DDR3), these banks form the next higher logical unit, known as a rank.[2].

4.2.1 Device operation - SDRAM as a state machine

In order to better understand the way in which the SDRAM works, it could be best described as a simple state machine reported in Figure 4.8, which is either idle, active, or precharging one or more open banks.

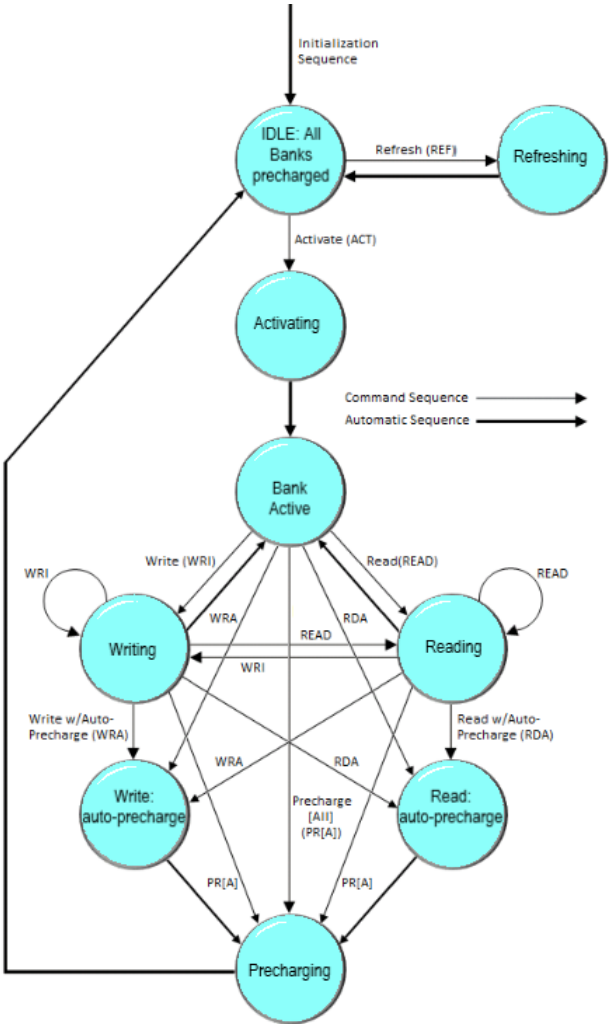

Figure 4.8: SDRAM state transition diagram

As with any machine, a transition from one state to another requires a minimum wait time before the system is ready to respond to subsequent requests to do additional work. These delays have a major impact on SDRAM read and write performances and more importantly, on the performance of the system as a whole.

Since SDRAM memory cells are just miniature capacitors, the charge they contain will dissipate away naturally over time due to many factors that can influence the leakage rate, including temperature. A marked reduction in stored charge can result in either data loss or data corruption. In order to prevent this from happening SDRAM must be periodically refreshed by topping off the charge contained in each memory cell. The frequency with which this refresh occurs depends on the silicon technology used to manufacture the core memory die and the design of the memory cell itself.
Reading or writing to a memory cell has the same effect as refreshing the selected cell by issuing a Refresh (REF) command. But, unfortunately, not all cells are read from or written to during the normal course of the operation and so each cell in the array must be accessed and written back (restored) before the expiration of the refresh interval. In most cases, refresh cycles involve restoring the charge along an entire page. Over the course of the entire interval, every page is accessed and subsequently restored, and then, at the end of the interval, the process begins again. A typical Refresh Period (t_{REF}) is hundreds to possibly a thousand or more clocks.
Moreover, all banks must be precharged and idle for a minimum of the RAS Precharge (t_{RP}) delay before the Refresh (REF) command can be applied. An address counter, internal to the device, supplies the bank address used during the course of the refresh cycle. When the refresh cycle has completed, all banks are left in the precharged (idle) state. A delay between the REF command and the next Activate (ACT) command or subsequent REF command must be greater than or equal to the Row Refresh Cycle Time (t_{RFC}). In other words, a minimum wait of t_{RFC} cycles is required following a refresh to an idle bank before it can be again activated for access.
Thus, before the SDRAM will be ready to respond to read and write commands, a bank must first be activated. The memory controller accomplishes this by sending the appropriate command (ACT), specifying the rank, bank, and page (row) to be accessed. The time to activate a bank is called the Row-Column (or Command) Delay and is denoted by the symbol t_{RCD}. This variable represents the minimum time needed to latch the command at the command interface, program the control logic, and read the data from the memory array into the Sense Amplifiers in preparation for column-level access.
Following activation, the activated bank contains within the array of Sense Amps a complete page of memory only 8 KB in length. At this time, multiple Read (READ) and Write (WRI) commands can be issued, specifying the starting column address to be accessed. Moreover, the time to read a byte of data from the open page is called the Column Address Strobe (CAS) Latency and it is denoted by the symbol CL or t_{CAS}. This variable represents the minimum time needed to latch the command at the command interface, program the control logic, gate the requested data from the Sense Amps into the Input/Output (I/O) Buffers, through a process known as pre-fetching, and place the first word of data on the Memory Bus.
Therefore, only one page per bank may be open at a time. Access to other pages in the same bank demands the open page first be closed. As long as the page remains open the memory controller can issue any combination of READ or WRI commands, sometimes switching back and forth between the two, until the open page is no longer needed or a pending request to read/write data from an alternate page in the same bank requires the current page to be closed so that another may be accessed. This is done by either issuing a Precharge (PR) command to close the specified bank only or a Precharge All (PRA) command to close all open banks in the rank.
Alternatively, the Precharge command can be effectively combined with the last read or write operation to the open bank by sending a Read with Auto-Precharge (RDA) or Write

with Auto-Precharge (WRA) command in place of the final READ or WRI command. This allows the SDRAM control logic to automatically close the open page as soon as the following conditions have been met:

- a minimum of RAS Activation Time (t_{RAS}) has elapsed since the ACT command was issued;

- a minimum of Read to Precharge Delay (t_{TRP}) has elapsed since the most recent READ command was issued.

Thus, the precharging prepares the data lines and sense circuitry to transmit the stored charge in the Sense Amps back into the open page of individual memory cells, undoing the previous destructive read, making the SDRAM core ready to sample the next page of memory to be accessed. The time to Precharge an open bank is called the Row Access Strobe (RAS) Precharge Delay and it is denoted by the symbol t_{RP}. The minimum time interval between successive ACT commands to the same bank is determined by the Row Cycle Time of the device, t_{RC}, found by simply summing t_{RAS} and t_{RP}. The minimum time interval between ACT commands to different banks is the Read-to-Read Delay (t_{RRD}).

4.2.2 SDRAM core scheduling

The process of moving data in and out of the Memory Array and over the Memory Bus is not overly complicated, in fact, both read and write access to DDR3 SDRAM is burst oriented, thus, the access starts at a selected location and continues in a pre-programmed sequence for a burst-length (BL) of 1 byte per bank. This begins with the registration of an ACT command and it is followed by one or more READ or WRI commands.
The SDRAM has a signal for each rank, called Chip Select, which could enable or disable the command decoder which works as a mask to ensure commands are acted upon by the desired rank only.
Moreover, the address bits registered coincident with the ACT command are used to select the bank and page (row) to be accessed, while the address bits registered coincident with the READ or WRI command are used to select the targeted starting column for the burst.
The length of each Read Burst (t_{Burst}) is always 4 clocks as DDR memory transmits data at twice the host clock rate, this is also called Double Data Rate.
In Figure 4.9 a top-down look at the minimum cycle needed to first open a page in memory and then read data from the activated page is reported. The steps of this procedure are:

- Step 1: the row is selected moving data to the Sense Amplifiers for executing sampling and amplification;

- Step 2: the column is selected muxing the needed bits to Output Buffer

- Step 3: the data returns over Memory Bus.

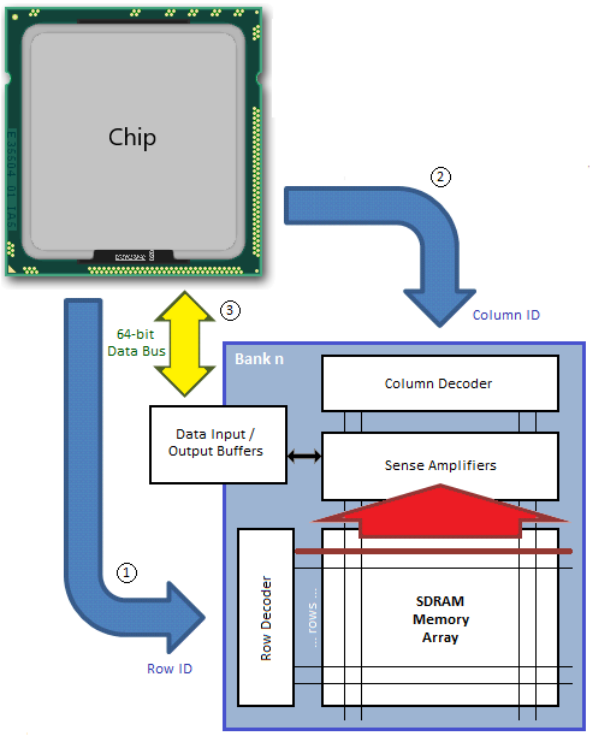

Figure 4.9: Memory read and write operations

4.2.3 Page Hit, Page Miss, Page Empty

Every read/write memory transaction can be segmented by type into one of three performance bins depending on the status from the bank/page to be accessed. These bins, in order of best to worst, are page-hit, page-empty, and page-miss.
A **page-hit** access is defined as any read or write operation to an open page. Thus, the bank, which contains the open page and it is already active, is immediately ready to service requests. Therefore, since the target page is already open, the nominal access latency for any memory transaction is approximately t_{CAS}, that is the CAS Latency of the device.
While, about the **page-empty**, a page-empty access is still preferred to a miss. In this case the bank to be accessed is Idle with no page open. Thus, commonly to read or write data to a page in a bank, previously it is necessary that the bank is activated. In other words, the nominal access latency now includes the time to open the page, thus, this is a doubling of the minimum access latency when compared to that of the page-hit case.
Finally, a **page-miss** occurs anytime a memory transaction must first close an open page in order to open an alternate page in the same bank. Only then the specified data access can take place. First closing an open page requires a Precharge, adding the RAS Precharge (t_{RP}) delay to any already lengthy operation, in fact, in this case the nominal latency of an access of this type is three times that of one page-hit operation.
Moreover, normalizing to the page-hit access latency, page-empty access is twice as long, and page-miss access is a whole three times as long. Thus, combining this with the inner functions of the SDRAM state machine, it could be observed that the page-hit and the page-miss are really just subsets of the same bank state (active). Of course, page-empty access necessarily implies an idle bank.

4.3 FDDR (DDR controller)

The FFDR controller is another of the most important parts of the thesis project together with the SDRAM DDR3 banks of memory. This is the controller of the banks of memory, managing the writing to and the reading from these banks.
More in details, the FDDR subsystem, which is part of the analyzed FPGA, is a hardened ASIC block for interfacing the DDR2, DDR3 and LPDDR1 memories. The RTG4 device has two FDDR blocks, which are used to access the DDR memories for high-speed data transfers. Inside the FDDR subsystem, there is a DDR memory controller, a DDR PHY and an arbitration logic to support multiple masters, which want to access at the same time the same source. One of the characteristics of the DDR controller is that it mitigates the Single Event Upset (SEU), thus, there is no performance impact due to SEU events. Moreover, as it could be seen in Figure 4.10, the way in which the FPGA fabric master communicates with the DDR memories interfaced to the FDDR subsystem is through the AXI or AHB interfaces, in particular, in the thesis project the interfaces which has been employed is the AXI interface. In addiction, the FDDR has also an APB slave interface to configure the the FDDR from the FPGA fabric master.[7]

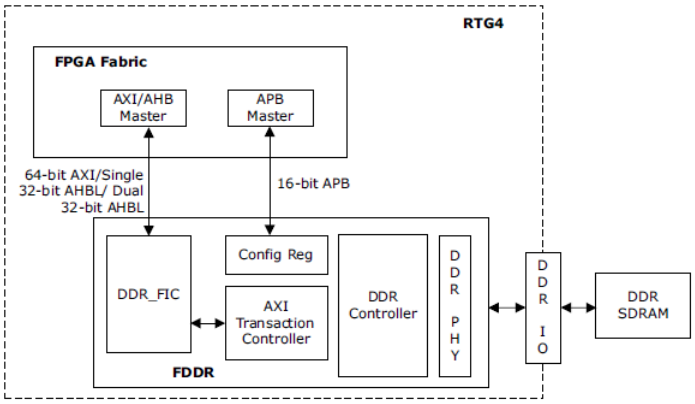

Figure 4.10: System-Level FDDR Block Diagram

As reported in the above figure, the FDDR subsystem is composed of the following blocks:

- **DDR Controller**: it receives requests from the AXI transaction controller, maps system addresses to DRAM addresses (rank, bank, row, and column), and prioritizes requests to minimize latency of reads. (especially high priority reads) and maximize page hits.
- **DDR PHY**: it provides a physical interface to DDR2, DDR3 and LPDDR1 SDRAM devices. It receives commands from the DDR controller and generates DDR memory signals required to access the external DDR memory.
- **DDR_FIC**: it facilities communication between the FPGA fabric master and the AXI transaction controller.
- **AXI Transaction Controller**: it receives 64-bit AXI transactions from DDR_FIC and translates them into DDR controller transactions.
- **Configuration registers**

The FDDR subsystem accepts data transfer requests from AXI or AHB interfaces. Any read or write transactions to the DDR memories could occur from the AXI or AHBL masters in the FPGA Fabric through the DDR_FIC interface.

Chapter 5

AXI interface

As it has been already said in the previous chapter, the FPGA fabric master communicates with the FDDR subsystem through an AXI interface. Here, how the AXI interface works is described.

5.1 AXI Protocol

The AXI protocol is targeted at high-performance and high-frequency system designs and it includes several features that make it suitable for a high-speed sub-micron interconnection.
The AXI protocol features are the following:

- Separate address
- Control and data phases
- Supports unaligned data transfers using byte strobes
- Burst-based transactions with only start address issued
- Separate read and write data channels
- Issues multiple outstanding addresses
- Out-of-order transaction completion
- Easy addition of register stages to provide timing closure

The AXI protocol is defined as a burst based protocol. Moreover, every transaction has an address and control information in the address channel that describes the nature of the data to be transferred.
This protocol specifies the independent channels used for the write transaction and the read one. The channels used for the write transaction, reported in Figure 5.1, are:

- Write address channel
- Write data channel
- Write response channel

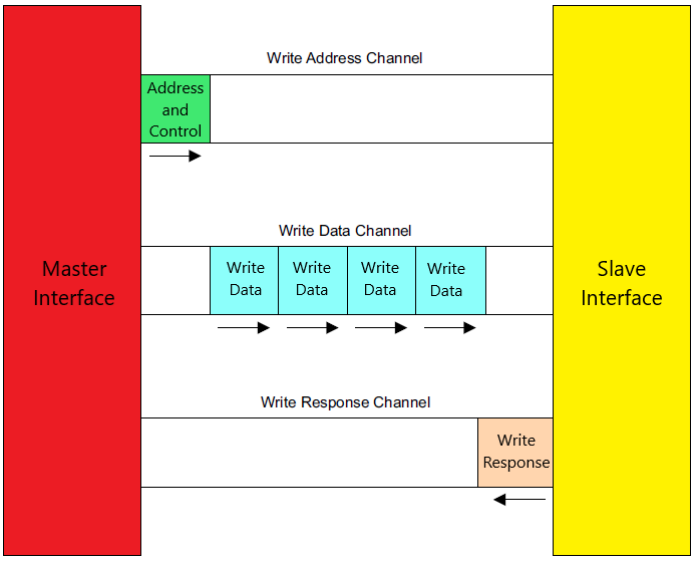

Figure 5.1: AXI Write Flow

While the channels used for the read transaction, reported in Figure 5.2, are the following:

- Read address channel
- Read data channel

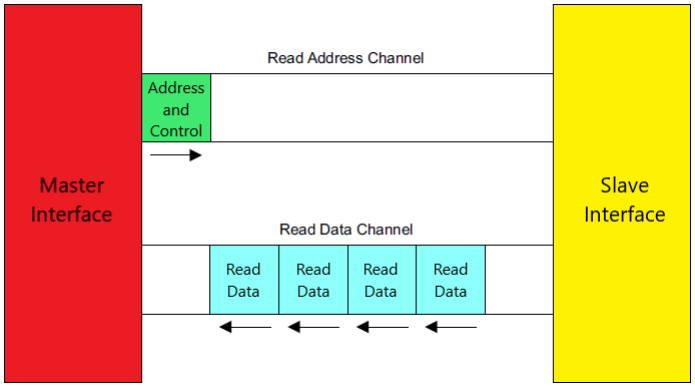

Figure 5.2: AXI Read Flow

Each of the five independent channels consists of a set of information signals and uses a two-way VALID and READY **handshake** mechanism. Therefore, the source displays the VALID signal on the channel whenever the valid data or control information is available on the channel. While, the destination displays the READY signal to show when it can accept the data. Both the read data and write data channels display the LAST signal when it transfers the final data item.[11]

5.1.1 AXI Write Transaction

During the AXI write transaction, the AXI master sends the write address using the write address channel and then it sends the write data by means of the write data channel. Finally, the slave sends the response using the write response channel.

More in detail, the write transaction mechanism in the AXI protocol could be divided in the following sub-sections, which signals behaviour is reported in Figure 5.3:

- **Write Address Channel Handshake Mechanism**:
  The AXI master asserts the AWVALID signal, at time T1 in the figure, only when it drives the valid address and control. The signal must remain asserted until the AXI slave accepts the address and control the information and asserts the associated AWREADY signal, at time T2 in the figure.

- **Write Data Channel Handshake Mechanism**:
  During a write transaction, the AXI master asserts the WVALID signal, at time T3, only when it drives the valid write data. The WVALID signal must remain asserted until the AXI slave accepts the write data and asserts the WREADY signal, at time T4.
  When the last data is sent, together with WVALID signal, also the WLAST signal is asserted, at time T9. Both these signals remain asserted till the AXI slave accepts the last write data and asserts the WREADY signal.

- **Write Response Channel Handshake Mechanism**:
  The AXI slave asserts the BVALID signal, at time T10, only when it drives the valid write response, BRESP. The BVALID signal must remain asserted until the master accepts the write response and asserts the BREADY signal. But, there is another option, indeed, the master can assert the BREADY signal before the slave asserts the BVALID signal to complete the response transfer in one cycle.

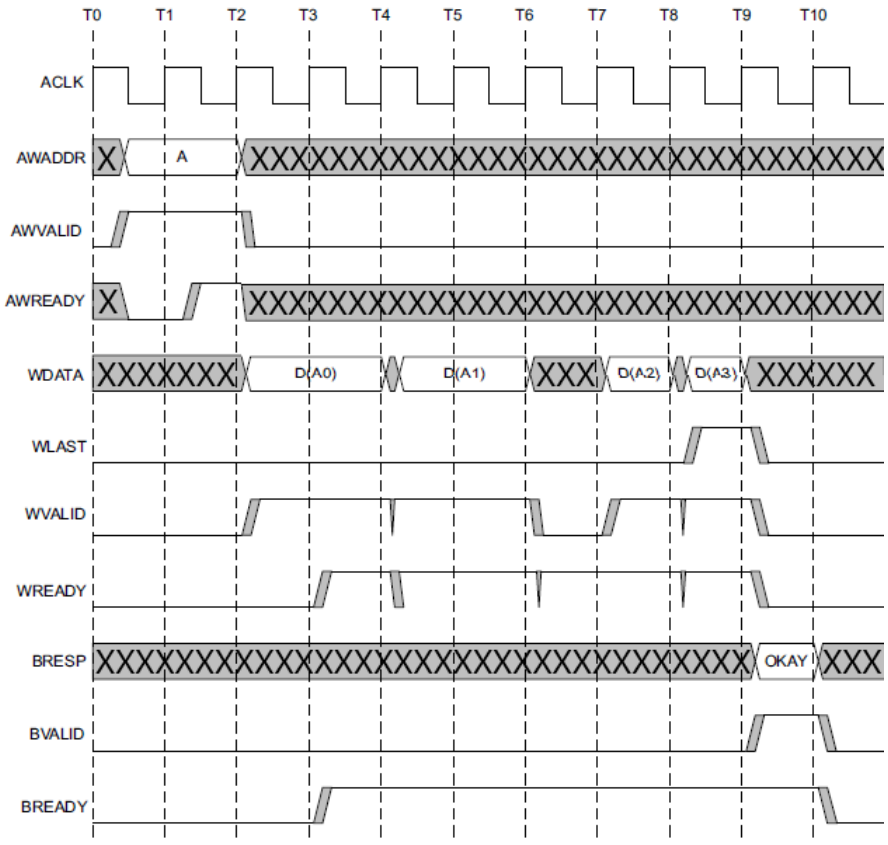

Figure 5.3: Write Transaction Timing Diagram with a Burst Length of 4

Moreover, the AXI protocol provides an ID field to enable a master to issue a number of separate transactions, each of which must be returned in order. A master can use the ARID or AWID signal of a transaction to provide additional information on the ordering requirements of the master. The slave transfers a BID to match the AWID and WID of the transaction to respond. If a master requires that all the transactions need to be completed in the same order that they are issued, then all of the transactions must have the same ID tag. In addition, the AXI protocol also provides burst type support, protection unit support, error support, and so on by using various AXI interface signals.

5.1.2 AXI Read Transaction

During the AXI read transaction, the AXI master sends the read address using the read address channel, then the slave sends read data back using the read data channel.
More in detail, the read transaction mechanism in the AXI protocol could be divided in the following sub-sections, which signals behaviour is reported in Figure 5.4:

- **Read Address Channel Handshake Mechanism**:
  The AXI master asserts the ARVALID signal, at time T1 in the figure, only when it drives the valid address and control information. It must remain asserted until the AXI slave accepts the address and control information and it asserts the associated ARREADY signal, at time T2.

- **Read Data Channel Handshake Mechanism** The AXI slave asserts the RVALID signal, at time T6, with the appropriate ID tags only when it drives the valid read data. The RVALID signal must remain asserted until the AXI master accepts the data and asserts the RREADY signal.
  Similarly to the write transaction, if the master is ready to accept data, it can assert RREADY before the slave asserts the RVALID signal. Moreover, even if an AXI slave has only one source of read data, it must assert the RLAST signal only in response to a request for the data.

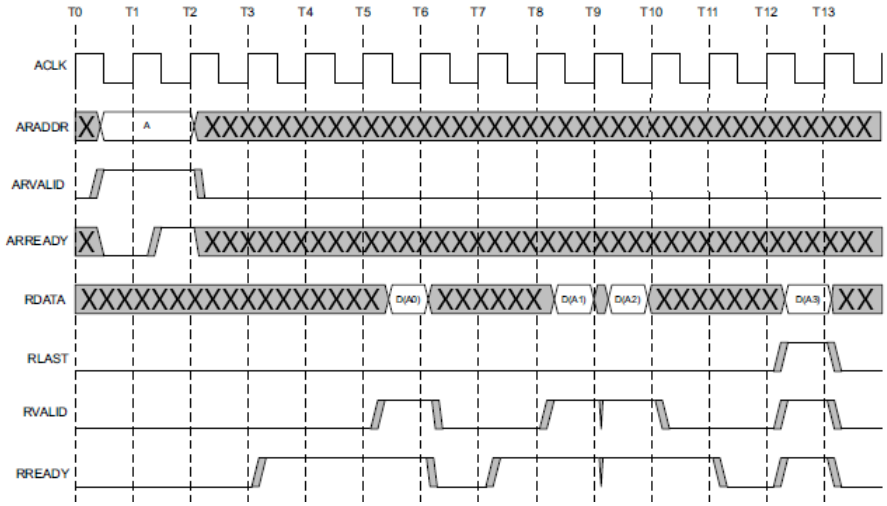

Figure 5.4: Read Transaction Timing Diagram with a Burst Length of 4

Furthermore, the master uses an ID tag during the read operation using the ARID signal and the slave must send the data back with the same ID tags using the RID signal.

Similarly, AWID and WID signals are used for the write transactions. Also the read transfer provides burst type support, protection unit support, error support, like the write transfer.

5.1.3 Implementation of an AXI Master Interface on the User Logic

To design an AXI master interface, that will be connected to the FDDR controller block, could be used a state machine, which has the behaviour described subsequently.
To initiate a write transaction, the AXI master interface uses the user interface information and sends the AXI write address and AXI write control information on the write address channel. The master needs to keep the address and control on the bus until the slave accepts and asserts the AWREADY signal. Then, the master sends each item of write data from user interface, over the write data channel. The master must keep the write data on the bus until the slave accepts the write data and asserts the WREADY signal. During Burst mode, the next data should be on the bus only after the slave receives the previous data by asserting the WREADY signal. Thus, when the master sends the last data item, the WLAST signal goes high and when the slave accepts all the data items, it drives a write response back to the master to indicate that the write transaction is complete. The master accepts the response and asserts the BREADY signal and also checks the response ID.
While, to initiate a read transaction, the AXI master interface uses the user interface information and it sends the AXI read address and AXI read control information waits for the slave to accept it. The master also drives a set of control signals that gives the length and type of the burst. The master keeps the address and control signals on the address bus, until the slave accepts and asserts the ARREADY signal. The data transfer occurs on the read data channel, the master asserts the RREADY signal to indicate that it can accept the read data. For the final data transfer of the burst, the slave asserts the RLAST signal to indicate the transfer of the last data item and the read state machine moves to the Idle state.

5.1.4 Implementation of an AXI Slave Interface on the User Logic

To create a custom AXI slave on a memory block, the below description could be followed, implementing a finite state machine, which generate the required write and read signals for the memory.
Thus, during a write transfer, the AXI slave block waits for AWVALID signal from the master and accepts the address and control information by asserting the associated AWREADY signal. The slave waits for the WVALID signal when the master drives valid write data. Therefore, the slave acknowledges receipt of the write data by asserting the WREADY signal. The slave receives data until the WLAST signal is asserted by the master. The slave indicates receipt of the data by asserting the BVALID signal with a valid write response including write response ID. The BVALID signal must remain asserted until the master accepts the write response and asserts the BREADY signal.
While, in a read transaction the AXI slave block waits for the ARVALID signal from the master and accepts the address and control information and asserts the associated ARREADY signal. The slave reads the read data from the user interface and indicates a valid read data by driving the RVALID signal high. Thus, the slave waits for the RREADY signal before driving new data and finally, the slave sends the RLAST signal when it drives the last data.

Chapter 6

High speed data buffer design

The case of study of this thesis is the VHDL design of a high-speed data buffer based on DDR memories for real-time processing of hyperspectral data implemented on the RT4G150 FPGA, which is a radiation-tolerant FPGA. This data buffer is used in a hyperspectral payload called CHIME (Copernicus Hyperspectral Imaging Mission for the Environment), implemented for the European Space Agency (ESA). This payload needs to enter in a concurrent way to a high-speed buffer in order to memorize the data coming from a detector, to do a spectral editing/binning, to fix the defective pixel, to apply some coefficients in order to obtain a linear calibration of the data radiometric value and to read again the data, grouping them into homogeneous packages before sending these towards the satellite's mass memory.
Thus, the high-speed data buffer has been implemented as a controller, since there are two different user entries, which could access to the DDR memories both in writing mode and reading mode. These two ways that the user could employ to access the memories are a direct access (DA) mode, in which only one data is written or read, and a First In First Out (FIFO) mode, in which a burst of data is written or read.
Furthermore, the controller has to manage the case in which two users want to access the memory one through the FIFO and the other one using the direct access interface at the same time, both to write or to read. To do it, inside the controller, an arbiter has been implemented, which uses the Round-Robin scheduling in order to decide whose turn it is to access the memory banks.
This block is not interfaced directly with the DDR memory banks, but between these, there is a hardware macrocell, called FDDR, that is already part of the RTG4 FPGA and it is a memory controller used to access the DDR memory banks for high-speed data transfer.
Moreover, the type of interface with which the data buffer and the FDDR communicate is the Advanced eXtensible Interface (AXI), in which a master, that in this case is the data buffer implemented in this thesis, communicates with a slave, which is represented by the FDDR.
The project of this block has been developed following the phases of the ECSS-Q-ST-60-02C standard of the European Cooperation for Space Standardization, which provides a series of contents and steps established with the aim of ensuring the quality of the final product. Furthermore, in the points of each step it is necessary to specify a verification method in order to put in evidence in which way each requirement is respected. The verification method could be one or more between these three types:

- **R**: review of design, in which the verification is a description in a report;

- **S**: simulation, in which the requirement has to be verified by a simulation;
- **T**: test, in which the requirement is verified with a test on board.

Thus, the aims of the designed block in this thesis project are:

- to make available a FIFO interface and a direct access interface to other logic block;
- to translate the access through the FIFO or direct access generic interfaces into accesses to the FDDR block, with the use of an AXI bus;
- to arbitrate by means of the round robin algorithm in case of concurrent requests, thus, when both FIFO and direct access interfaces want to write or read from the same source, which in this case is represented by the DDR memory blocks, going through the FDDR.

In the following sections, the AXI interface signals are reported. Thus, to understand in a better way the meaning of these signals, in Table 6.1 and Table 6.2 there is an explanation more in detail for each signal.[7]

Signal Name	Polarity	Description
ARREADY	High	Indicates whether the slave is ready to accept an address and associated control signals. 1=Slave ready; 0=Slave not ready
AWREADY	High	Indicates that the slave is ready to accept an address and associated control signals. 1=Slave ready; 0=Slave not ready
BID[3:0]		Indicates response ID. The identification tag of the write response.
BRESP[1:0]		Indicates write response and the status of the write transaction. 00=Normal access okay; 01=Exclusive access okay; 10=Slave error; 11: Decode error
BVALID	High	Indicates whether a valid write response is available. 1=Write response available; 0=Write response not available
RDATA[63:0]		Indicates read data.
RID[3:0]		Read ID tag. ID tag of the read data group of signals.
RLAST	High	Indicates the last transfer in a read burst.
RRESP[1:0]		Indicates read response and the status of the read transfer. 00=Normal access okay; 01=Exclusive access okay; 10=Slave error; 11=Decode error
RVALID		Indicates whether the required read data is available and the read transfer can complete. 1=Read data available; 0=Read data not available
WREADY	High	Indicates whether the slave can accept the write data. 1=Slave ready; 0=Slave not ready

Table 6.1: AXI interface signals from slave to master

Signal Name	Polarity	Description
ARADDR[31:0]		Indicates the initial address of a read burst transaction.
ARBURST[1:0]		Indicates burst type. 00=FIXED: Fixed-address burst FIFO type; 01=INCR: Incrementing-address burst normal sequential memory; 10=WRAP: Incrementing-address burst that wraps to a lower address at the wrap boundary; 11=Reserved
ARID[3:0]		Indicates identification tag for the read address group of signals.
ARLEN[3:0]		Indicates burst length. The burst length gives the exact number of transfers in a burst. 0000=1; 0001=2; ...; 1111=16
ARLOCK[1:0]		Indicates lock type. This signal provides additional information about the atomic characteristics of the read transfer. 00=Normal access; 01=Exclusive access; 10=Locked access; 11=Reserved
ARSIZE[1:0]		Indicates the maximum number of data bytes to transfer in each data transfer, within a burst. 00=1; 01=2; 10=4; 11=8
ARVALID	High	Indicates the validity of read address and control information. 1=Address and control information valid; 0=Address and control information not valid
AWADDR[31:0]		Indicates write address. The write address bus gives the address of the first transfer in a write burst transaction.
AWBURST[1:0]		Indicates burst type. 00=FIXED: Fixed-address burst FIFO-type; 01=INCR: Incrementing-address burst normal sequential memory; 10=WRAP: Incrementing-address burst that wraps to a lower address at the wrap boundary 11: Reserved
AWID[3:0]		Indicates identification tag for the write address group of signals.
AWLEN[3:0]		Indicates burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address. 0000=1; 0001=2; ...; 1111=16
AWLOCK[1:0]		Indicates lock type. This signal provides additional information about the atomic characteristics of the write transfer. 00=Normal access; 01=Exclusive access; 10=Locked access; 11=Reserved
AWSIZE[1:0]		Indicates the maximum number of data bytes to transfer in each data transfer, within a burst. 00=1; 01=2; 10=4; 11=8
AWVALID	High	Indicates whether a valid write address and control information are available. 1=Address and control information available; 0=Address and control information not available
BREADY	High	Indicates whether the master can accept the response information. 1=Master ready; 0=Master not ready
RREADY	High	Indicates whether the master can accept the read data and response information. 1=Master ready; 0=Master not ready
WDATA[63:0]		Indicates write data.
WID[3:0]		Indicates response ID, the identification tag of the write response.
WLAST	High	Indicates the last transfer in a write burst.
WSTRB[7:0]		Indicates which byte lanes to update in memory.
WVALID	High	Indicates whether a valid write data and strobes are available. 1=Write data and strobes available; 0=Write data and strobes not available

Table 6.2: AXI interface signals from master to slave

6.1 User requirements (URD)

The first phase of the ECSS-Q-ST-60-02C flow is the user requirements document (**URD**), in which the user needs are specified at high level. The output of this phase is a document called FPGA requirements (**ARS**), where the architecture and the functions of the logic are specified.
Thus, the URD is divided in 5 sections: introduction, functions, interfaces, error management and performances. Each section is composed of a list of requirements that the user want that the project respects. To each requirement is associated a code, that is used then as reference in the ARS.

6.1.1 Introduction

In the introduction section, the user defines which is the aim of this activity, the way that the designer has to follow to develop the block and the constraints that has to take into account.
More in particular the list of requirements is the following, where also the code assigned to each requirement and the verification method are reported:

- **URD-REQ-105: Subject of activity**
  Verification method: R

  The subject of this activity should be the development of a custom HDL block, which interfaces the DDR memory controller of the Microchip RTG4 FPGA and that exposes to the user a FIFO interface and a direct access interface, both with distinct read and write ports.

- **URD-REQ-110: Development methodology**
  Verification method: R

  The development of this HDL block should follow a tailoring of the methodology described in the standard ECSS-Q-ST-60-02C. Tailoring should be limited to the HDL development process (requirements, verification, validation concepts), but skipping formal reviews and document issues.

- **URD-REQ-115: Technological constraint**
  Verification method: R

  The HDL block should interface the FDDR memory controller macrocell on the Microchip RTG4 FPGA device.

- **URD-REQ-120: Development constraint**
  Verification method: R

  The FDDR memory controller should be configured to work with the DDR3 memories of the RTG4 development kit evaluation board.

6.1.2 Functions

In the function section of the URD, the information that the user gives are a sketch of how the block architecture has to be realized, the features of the FIFO and direct access emulation functions and of the arbiter function and how the DDR memory has to be partitioned.
In this case the list of requirement is the following:

- **URD-REQ-205: Block architecture**
  Verification method: R

  The designed block should have the following architecture:

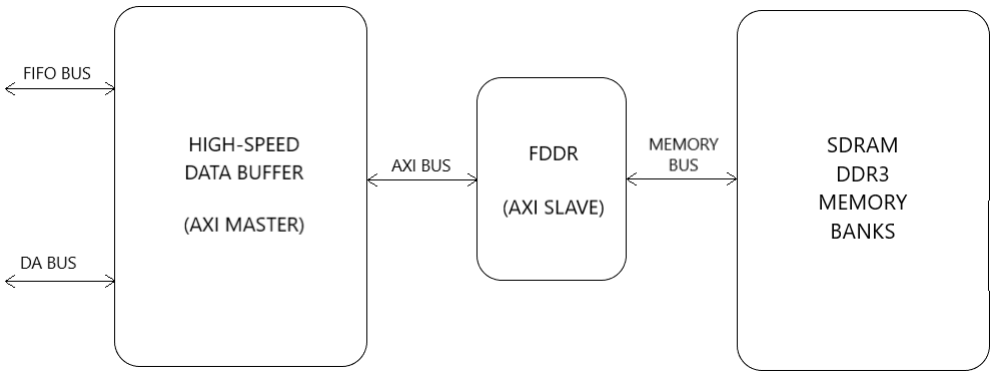

Figure 6.1: Sketch of block architecture

- **URD-REQ-210: FIFO emulation function**
  Verification method: S,T

  This block should transparently emulate a 16-bit synchronous FIFO that operates on burst of 8 words on both ports. It should expose interfaces for read/write ports and should use the DDR interface as mass memory for data to be stored.
  Rationale: operating FIFO ports with data bursts allows to simplify design and access to DDR.

- **URD-REQ-215: Direct access memory emulation function**
  Verification method: S,T

  This block should transparently emulate a 16-bit memory with simple direct access (address and data). It should expose interfaces for read/write ports and should use the DDR interface as mass memory for data to be stored.

- **URD-REQ-220: Arbiter function**
  Verification method : S,T

  This block should transparently arbitrate access to all above interfaces. All of the interfaces should be capable of accepting and queuing at least one access, that will be served by arbiter with round robin scheduling.

- **URD-REQ-225: DDR memory partitioning**
  Verification method : S,T

  The DDR address space should be divided into two distinct subsections:

  – 16 kbytes are dedicated to the FIFO emulation;

  – the remaining space is dedicated to the direct access memory emulation.

6.1.3 Interfaces

In the interfaces section of the user requirements document, the signals for the write and read interface both for FIFO and direct access are indicated and what is the behaviour of each sequence.
More in detail, the points which compose the interface section are reported below:

- **URD-REQ-305 : FIFO write interface**
  Verification method : R,S

  The FIFO write interface should expose the following signals:

  - [*O*] FIFO_WRDY : write ready flag
  - [*I*] FIFO_WE : write enable
  - [*I*] FIFO_DIN[15:0] : data input
  - [*O*] FIFO_FF : full flag

- **URD-REQ-310 : FIFO read interface**
  Verification method : R,S

  The FIFO read interface should expose the following signals:

  - [*O*] FIFO_RRDY : read ready flag
  - [*I*] FIFO_RQ : read request
  - [*O*] FIFO_DOUT[15:0] : data output
  - [*O*] FIFO_DVALID : data valid
  - [*O*] FIFO_EF : empty flag

- **URD-REQ-315 : FIFO write sequence**
  Verification method : S,T

  This block should accept a write sequence only if FIFO_WRDY is asserted and FIFO_FF is de-asserted. The sequence should be composed of 8 consecutive clock cycles with FIFO_WE asserted and data received on FIFO_DIN.

- **URD-REQ-320 : FIFO read sequence**
  Verification method : S,T

  This block should accept a read sequence only if FIFO_RRDY is asserted and FIFO_EF is de-asserted. The sequence should be initiated by a FIFO_RQ strobe and a burst of 8 data should be generated on FIFO_DOUT with FIFO_DVALID asserted.

- **URD-REQ-325 : Direct access write interface**
  Verification method : R,S

  The direct access write interface should expose the following signals:

  - [*O*] DA_WRDY : write ready flag
  - [*I*] DA_WE : write enable
  - [*I*] DA_WADDR[31:0] : write address

  - [*I*] DA_DIN[15:0] : data input

- **URD-REQ-330 : Direct access read interface**
  Verification method : R,S

  The direct access read interface should expose the following signals:

  - [*O*] DA_RRDY : read ready flag
  - [*I*] DA_RQ : read request
  - [*I*] DA_RADDR[31:0] : read address
  - [*O*] DA_DOUT[15:0] : data output
  - [*O*] DA_DVALID : data valid

- **URD-REQ-335 : Direct access write sequence**
  Verification method : S,T

  This block should accept a write sequence only if DA_WRDY is asserted. The sequence should be composed of a single clock cycle with DA_WE asserted and data received on DA_DIN.

- **URD-REQ-340 : Direct access read sequence**
  Verification method : S,T

  This block should accept a read sequence only if DA_RRDY is asserted. The sequence should be initiated by a DA_RQ strobe and a data should be provided on DA_DOUT with DA_DVALID asserted.

6.1.4 Error management

During the execution of a read from or a write to the DDR3 memory banks, some errors could occur and the controller block implemented in this thesis has to report these errors and to fix these if it is possible. The error types and their management are listed better in the following points:

- **URD-REQ-405 : Error interface not ready**
  Verification method : S,T

  The access to any port should be discarded if the port is not ready.

- **URD-REQ-410 : Error FIFO flag asserted**
  Verification method : S,T

  The access to FIFO write port should be discarded if FIFO_FF is asserted, thus, the part of memory dedicated to the FIFO is full, and the access to FIFO read port should be discarded if FIFO_EF is asserted, so the part of memory dedicated to the FIFO is empty.

- **URD-REQ-415 : Error FIFO burst length**
  Verification method : S,T

  The access to the FIFO ports with burst of length different from 8 should be discarded.

- **URD-REQ-420 : Error direct access burst**
  Verification method : S,T

  The access to the direct access port with duration greater than one clock cycle should be discarded.

- **URD-REQ-425 : Error direct access invalid address**
  Verification method : S,T

  The access to the direct access port with DDR address reserved for FIFO should be discarded.

6.1.5 Performances

In the performances section, the user specifies the features that the project has to respect. In particular, the performances which have to be satisfied are:

- **URD-REQ-505 : Logic system clock**
  Verification method : R,S

  The system clock for logic should be 80 MHz.

- **URD-REQ-510 : DDR operating clock**
  Verification method : R,S,T

  The DDR operating clock should be 160 MHz.

- **URD-REQ-515 : Concurrent access**
  Verification method : R,S

  The implemented controller should be possible to successfully perform an access to all four ports. The four accesses should be accepted and queued, to be served one after the other with round robin scheduling.

6.2 FPGA requirements (ARS)

The FPGA requirements document is created by the designer in response to the user requirements. The designer indicates if there is some change with respect to what the user has indicated in the URD and in which way the user requirements will be satisfied in the project. Even in this case, the document is divided into different sections, which are composed of a list of points (requirements), that could be referred to one of the points in the URD and in this case, the referred point is called *parent*, or these could not have any parent, thus, these are new requirements generated by the designer, whose parent is indicated as *CREATED*.
What is important is that each user requirement is covered by at least a FPGA requirement.
Thus, the sections in which the ARS is divided are: functional requirements, logic architecture, direct access write block, direct access read block, FIFO access write block, FIFO access read block and arbiter.
Even here, each requirement is assigned to an own code and the verification method is indicated to each one.

6.2.1 Functional requirements

In the functional requirements section, the user requirements related to the FIFO and direct access emulation functions and the arbiter function are satisfied, but, since, in this case, the designer believes that there is no need to change something with respect to the definition of these user requirements, these FPGA requirements report what is written in the relative user requirements.
Thus, these requirements in the ARS are reported as below:

- **ARS-REQ-100 : FIFO emulation function**
  Verification method : S,T
  Parent : URD-REQ-210

  This block should transparently emulate a 16-bit synchronous FIFO that operates on burst of 8 words on both ports. It should expose interfaces for read/write ports and should use the DDR interface as mass memory for data to be stored.
  Rationale: operating FIFO ports with data bursts allows to simplify design and access to DDR.

- **ARS-REQ-105 : Direct access memory emulation function**
  Verification method : S,T
  Parent : URD-REQ-215

  This block should transparently emulate a 16-bit memory with simple direct access (address and data). It should expose interfaces for read/write ports and should use the DDR interface as mass memory for data to be stored.

- **ARS-REQ-110 : Arbiter function**
  Verification method : S,T
  Parent : URD-REQ-220

  This block should transparently arbitrate access to all above interfaces. All of the interfaces should be capable of accepting and queuing at least one access, that will be served by arbiter with Round Robin scheduling.

6.2.2 Logic architecture

In the logic architecture, the designer describes more in detail the architecture of the block, showing the sub-blocks, and it corresponds to how the project will be implemented on Libero SoC. Moreover, in these requirements, it is also described how the system and the internal clocks are generated, the sequence of the reset signal and the synchronization of it, and, at the end, how the FDDR controller has to be configured.
These part are described more in detail in the following points:

- **ARS-REQ-200 : Block architecture**
  Verification method : R
  Parent : URD-REQ-205/210/215/220

  The block implemented in this thesis should follow the architecture reported in Figure 6.2.

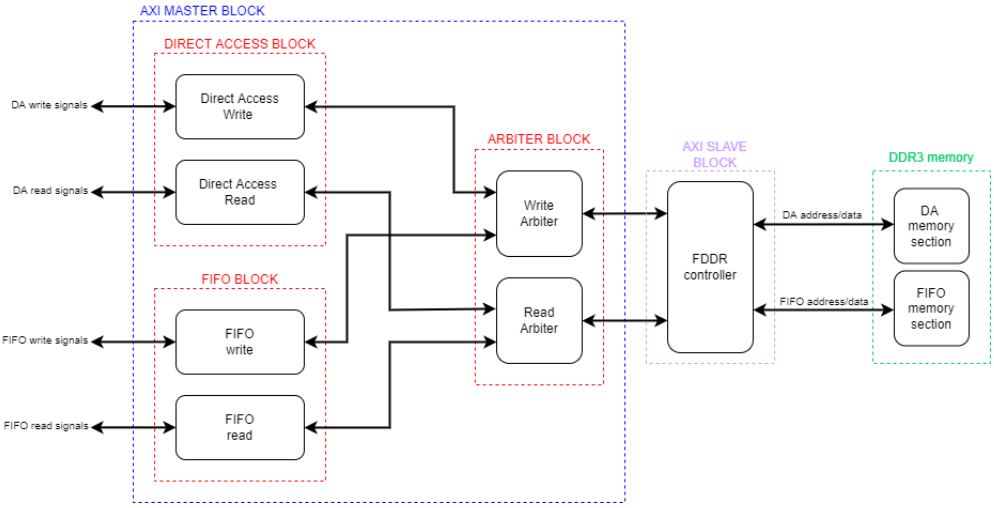

Figure 6.2: Sub-blocks architecture

Here, with respect to the user requirements, the AXI master block and the DDR3 memory are represented more in detailed, showing also how the internal blocks should be.

- **ARS-REQ-205 : Clock**
  Verification method : R,S
  Parent : URD-REQ-505

  The system 80 MHz clock should be generated using a PLL and the internal RTG4 50 MHz oscillator.

- **ARS-REQ-210 : Reset sequence**
  Verification method : R,S
  Parent : CREATED

  The reset sequence should be the following:

  – Reset event: DEVRST pin
  – Wait for PLL lock
  – Wait for FDDR configured
  – Release reset to user logic

- **ARS-REQ-215 : Reset synchronization**
  Verification method : R,S
  Parent : CREATED

  The reset signal should be asserted asynchronously and de-asserted synchronous with the corresponding clock domain.

- **ARS-REQ-220 : FDDR configuration**
  Verification method : R,S
  Parent : CREATED

  The FDDR block is an IP of the RT4G150 FPGA and, during the design of the project, it should be configured as reported here below, in order to interface both with the AXI master and with the DDR3 memory banks:

  – About General section:

    - West FDDR used on RTG4
    - Interface to 32-bit DDR3 memory at 320MHz
    - Address bits for row, bank and column are respectively: 16-bit, 3-bit and 10-bit
    - AXI system clock 80MHz, obtained by means of a FDDR CLOCK Divisor of 4
    - IO Drive Strength is Half Drive Strength
- About Memory Initialization section:
    - 8-bit as burst length
    - Sequential burst order
    - 1T timing mode
    - 6 clock cycles as CAS latency
    - Self-Refresh is disable
    - Auto refresh after a burst of 8
    - Powerdown is enable
    - 96 clock cycles before power down
    - No additive CAS latency
    - 512 clock cycles required after a ZQ calibration long (Zqinit)
    - 64 clock cycles required after a ZQ calibration short (ZQCS)
    - 8389632 clock cycles is the average interval to wait between automatically issuing ZQ calibration short command (ZQCS Interval). It is a multiple of 1024 clock cycles.
    - Local ODT is enabled during the read transaction
    - Drive strength and Rtt_NOM are set to RZQ/6
    - Rtt_WR is disable
    - Auto self-refresh is manual
    - Self-refresh temperature is normal
- About Memory Timing section:
    - Number of cycles to assert DRAM reset signal during initialization sequence (Time to Hold Reset before INIT) is 67584 clock cycles
    - Minimum RAS time is 15 clock cycles
    - Maximum RAS time is 8192 clock cycles
    - RCD time is 6 clock cycles
    - RP time is 7 clock cycles
    - REFI time is 3104 clock cycles. It is expressed as multiple of 32 clock cycles
    - RC time is 51 clock cycles
    - XP time is 3 clock cycles
    - Minimum number of cycles of CKE HIGH/LOW during power down and self-refresh is 3 clock cycles
    - RFC time is 79 clock cycles
    - WR time is 6 clock cycles
    - FAW time is 32 clock cycles

6.2.3 Direct access write block

In the direct access write block section, the requirements about the signals, concerning the interfacing with the user and with the FDDR, the sequence that the direct access block has to follow to work properly in writing, the errors that could occur and what is the memory space dedicated to the direct access write are listed. These requirements are better described below:

- **ARS-REQ-300 : Direct access write interface**
  Verification method : S
  Parent : URD-REQ-325

  The direct access write interface should expose the following signals:

  - [*O*] DA_WRDY : write ready flag
  - [*I*] DA_WE : write enable
  - [*I*] DA_WADDR[31:0] : write address
  - [*I*] DA_DIN[15:0] : input data
  - [*O*] DA_WR_ERR1 : write error flag 1
  - [*O*] DA_WR_ERR2 : write error flag 2
  - [*O*] DA_WR_ERR3 : write error flag 3

  Here, three signals have been added with respect to its parent URD-REQ-325. These signals are: DA_WR_ERR1, DA_WR_ERR2 and DA_WR_ERR3, which are employed to indicate to the user that an error occurs. This change with respect to the URD is chosen by the designer.

- **ARS-REQ-305 : Direct access write sequence**
  Verification method : S,T
  Parent : URD-REQ-335

  If DA_WRDY is asserted, this block should accept a write sequence composed of a single clock DA_WE asserted and data received on DA_DIN.

- **ARS-REQ-310 : Error write interface not ready**
  Verification method : S
  Parent : URD-REQ-335/405

  If DA_WRDY is de-asserted, the access to this port should be discarded and DA_WR_ERR3 error flag raised.

- **ARS-REQ-315 : Error direct write access burst**
  Verification method : S
  Parent : URD-REQ-420

  The access to direct access port with duration greater than one clock cycle should be discarded and DA_WR_ERR1 error flag raised.

- **ARS-REQ-320 : DDR memory partitioning for write**
  Verification method : S,T
  Parent : URD-REQ-225

  The DDR3 address space allowed for direct access write is the following:

  - Starting address: “00000000000000000000001111111111”
  - Final address: “00000000000000011111111111111110"

  The dimension of the memory part dedicated to direct access has been chosen as 112 kbytes.

- **ARS-REQ-325 : Error direct write access invalid address**
  Verification method : S
  Parent : URD-REQ-420

  The access to direct access port with an address reserved for FIFO should be discarded and DA_WR_ERR2 error flag raised.

- **ARS-REQ-330 : Direct write access AXI interface**
  Verification method : R,S
  Parent : CREATED

  The direct write access block should interface FDDR block through the following AXI signals:

  - Write Address channel
    - [*I*] AWREADY : write address ready
    - [*O*] AWVALID : write address valid
    - [*O*] AWADDR[31:0] : write address
    - [*O*] AWID[3:0] : write address identifier → AWID=0000
    - [*O*] AWLEN[3:0] : write address length → AWLEN=0000
    - [*O*] AWSIZE[1:0] : write address size → AWSIZE=11
    - [*O*] AWBURST[1:0] : write address burst → AWBURST=01
    - [*O*] AWLOCK[1:0] : write address lock → AWLOCK=00
  - Write Data channel
    - [*I*] WREADY : write data ready
    - [*O*] WLAST : write data last
    - [*O*] WVALID : write data valid
    - [*O*] WDATA[15:0] : write data
    - [*O*] WID[3:0] : write data identifier → WID=0000
    - [*O*] WSTRB[7:0] : write data strobe → WSTRB=11111111
  - Write Response channel
    - [*I*] BVALID : valid response
    - [*I*] BRESP[1:0] : response
    - [*I*] BID[3:0] : response identifier
    - [*O*] BREADY : ready response

- **ARS-REQ-335 : Direct write access AXI sequence**
  Verification method : S
  Parent : CREATED

  The write transaction to the FDDR should be the following:

  - AWID, AWLEN, AWSIZE, AWBURST, AWLOCK, WID, WSTRB signals are fixed.
  - When the AXI master drives the valid address AWADDR, it asserts the AWVALID signal, which remains high till the AXI slave asserts the AWREADY as acknowledgment that the address has been received in a correct way.
  - After that, when the AXI master drives the valid write data WDATA, it asserts WVALID signal and at the same time even WLAST, since in the direct access the burst is made up of only 1 data. WVALID and WLAST remain asserted until AXI slave asserts the WREADY as acknowledgment that the data has been received correctly.
  - Finally, the AXI slave asserts the BVALID signal only when it drives the valid write response, BRESP, which corresponds to BRESP=00 when it is correct. The BVALID remains asserted till the AXI master accepts the write response and asserts the BREADY signal.
  - The AXI master could have the possibility to assert the BREADY before that the AXI slave asserts the BVALID, in order to terminate the response transfer in only one clock cycle. In this case, it has been chosen to assert the BREADY signal when also WVALID and WLAST signals are asserted.

6.2.4 Direct access read block

In the direct access read block section, similarly to the previous section, the requirements about the signals to interface with the user and with the FDDR, the sequence that the direct access block has to follow to work properly in reading, the errors that could occur and what is the memory space dedicated to the direct access read are reported below:

- **ARS-REQ-400 : Direct access read interface**
  Verification method : S
  Parent : URD-REQ-330

  The direct access read interface should expose the following signals:

  - [*O*] DA_RRDY : read ready flag
  - [*I*] DA_RQ : read request
  - [*I*] DA_RADDR[15:0] : read address
  - [*O*] DA_DOUT[15:0] : data output
  - [*O*] DA_DVALID : data valid
  - [*O*] DA_RD_ERR1 : read error flag 1
  - [*O*] DA_RD_ERR2 : read error flag 2
  - [*O*] DA_RD_ERR3 : read error flag 3

  Even in this case, three signals (DA_RD_ERR1, DA_RD_ERR2 and DA_RD_ERR3) have been added with respect to its parent URD-REQ-330, in order to indicate to the user when an error occurs.

- **ARS-REQ-405 : Direct access read sequence**
  Verification method : S,T
  Parent : URD-REQ-340

  If DA_RRDY is asserted, this block should initiate a read sequence with a DA_RQ strobe. The data should be sampled on DA_DOUT with DA_DVALID asserted.

- **ARS-REQ-410 : Error read interface not ready**
  Verification method : S,T
  Parent : URD-REQ-335/405

  If DA_RRDY is de-asserted, the access to this port should be discarded and DA_RD_ERR3 error flag asserted.

- **ARS-REQ-415 : Error read direct access burst**
  Verification method : S,T
  Parent : URD-REQ-420

  The access to direct access port with a duration greater than one clock cycle should be discarded and DA_RD_ERR1 error flag asserted.

- **ARS-REQ-420 : DDR memory partitioning for read**
  Verification method : S,T
  Parent : URD-REQ-225

  The DDR3 address space allowed for direct access read is the same dedicated to the writing and it is the following:

  - Starting address: “00000000000000000000001111111111”
  - Final address: “00000000000000011111111111111110"

The dimension of the memory part dedicated to direct access has been chosen as 112 kbytes.

- **ARS-REQ-425 : Error direct read access invalid address**
  Verification method : S
  Parent : URD-REQ-420

  The access to direct access read port with DDR address reserved for FIFO should be discarded and DA_RD_ERR2 error flag asserted.

- **ARS-REQ-430 : Direct read access AXI interface**
  Verification method : R,S
  Parent : CREATED

  The direct read access block should interface FDDR block through the following AXI signals:

  - Read Access channel:
    - [*I*] ARREADY : read address ready
    - [*O*] ARVALID : read address valid
    - [*O*] ARADDR[31:0] : read address
    - [*O*] ARID[3:0] : read address identifier → ARID=0000
    - [*O*] ARLEN[3:0] : read address length → ARLEN=0000
    - [*O*] ARSIZE[1:0] : read address size → ARSIZE=11
    - [*O*] ARBURST [1:0] : read address burst → ARBURST=01
    - [*O*] ARLOCK [1:0] : read address lock → ARLOCK=00
  - Read Data channel:
    - [*I*] RVALID : read data valid
    - [*I*] RLAST : read data last → last transfer in a read burst
    - [*I*] RDATA[15:0] : read data
    - [*I*] RID[3:0] : read identifier
    - [*O*] RREADY : read data ready
  - Read Response channel:
    - [*I*] RRESP[1:0] : response

- **ARS-REQ-435 : Direct read access AXI sequence**
  Verification method : S
  Parent : CREATED

  The read transaction to the FDDR should be the following:

  - ARID, ARLEN, ARSIZE, ARBURST, ARLOCK, RID signals are fixed.
  - When the AXI master drives the valid address ARADDR, it asserts the ARVALID signal, which remains high till the AXI slave asserts the ARREADY as acknowledgment that the address has been received in a correct way.
  - Subsequently, the AXI slave asserts the RVALID signal when it drives the valid read data. Together with the RVALID, even the RLAST signal is asserted since in the direct access the burst is made up of only 1 data. RVALID and RLAST remains high till the AXI master receives the data and asserts the RREADY signal as acknowledgment that the data has been received correctly.
  - The RREADY signal could be asserted previously with respect to the assertion of RVALID and RLAST, if the AXI master is ready to receive a data. In this case, it has been chosen to assert the RREADY after that the ARREADY has been asserted from the AXI slave.

6.2.5 FIFO access write block

In the FIFO access write block section, the requirements about the signals, concerning the interfacing with the user and with the FDDR, the sequence that the FIFO has to follow to work properly in writing, the errors that could occur and what is the memory space dedicated to the FIFO write are listed. These requirements are better described below:

- **ARS-REQ-440 : FIFO write interface**
  Verification method : S
  Parent: URD-REQ-305

  The FIFO write interface should expose the following signals:

  - [*O*] FIFO_WRDY : write ready flag
  - [*I*] FIFO_WE : write enable
  - [*I*] FIFO_DIN[15:0] : data input
  - [*O*] FIFO_FF : full flag
  - [*O*] FIFO_WR_ERR1 : write error flag 1
  - [*O*] FIFO_WR_ERR2 : write error flag 2
  - [*O*] FIFO_WR_ERR3 : write error flag 3

  Here, three signals have been added with respect to its parent URD-REQ-305. These signals are: DA_WR_ERR1, DA_WR_ERR2 and DA_WR_ERR3, which are employed to indicate to the user that an error occurs.

- **ARS-REQ-445 : FIFO write sequence**
  Verification method : S,T
  Parent: URD-REQ-315

  If FIFO_WRDY is asserted and FIFO_FF is de-asserted, this block should accept a write sequence composed of 8 consecutive clock cycles with FIFO_WE asserted and data received on FIFO_DIN.

- **ARS-REQ-450 : Error write interface not ready**
  Verification method : S,T
  Parent: URD-REQ-315/405

  If FIFO_WRDY is de-asserted, the access to this port should be discarded and FIFO_WR_ERR3 error flag raised.

- **ARS-REQ-455 : Error FIFO burst length**
  Verification method : S,T
  Parent: URD-REQ-415

  The access to FIFO ports with burst of length different from 8 should be discarded and FIFO_WR_ERR2 error flag raised.

- **ARS-REQ-460 : DDR memory partitioning for write**
  Verification method : S,T
  Parent: URD-REQ-225

  The DDR3 address space allowed for FIFO write is the following:

  - Starting address: “00000000000000000000000000000000”
  - Final address: “00000000000000000000001111111110"

The dimension of the memory part dedicated to the FIFO has been chosen as 16 kbytes.

- **ARS-REQ-465 : Error FULL FIFO flag asserted**
  Verification method : S
  Parent: URD-REQ-410

  The access to FIFO write port should be discarded if FIFO_FF is asserted and FIFO_WR_ERR1 error flag raised.

- **ARS-REQ-470 : FIFO AXI interface**
  Verification method : R,S
  Parent : CREATED

  FIFO block should interface FDDR block through the following AXI signals:

  - Write Address channel
    - [*I*] AWREADY : write address ready
    - [*O*] AWVALID : write address valid
    - [*O*] AWADDR[31:0] : write address
    - [*O*] AWID[3:0] : write address identifier → AWID=0000
    - [*O*] AWLEN[3:0] : write address length → AWLEN=0001
    - [*O*] AWSIZE[1:0] : write address size → AWSIZE=11
    - [*O*] AWBURST[1:0] : write address burst → AWBURST=01
    - [*O*] AWLOCK[1:0] : write address lock → AWLOCK=00
  - Write Data channel
    - [*I*] WREADY : write data ready
    - [*O*] WLAST : write data last
    - [*O*] WVALID : write data valid
    - [*O*] WDATA[15:0] : write data
    - [*O*] WID[3:0] : write data identifier → WID=0000
    - [*O*] WSTRB[7:0] : write data strobe → WSTRB=11111111
  - Write Response channel
    - [*I*] BVALID : valid response
    - [*I*] BRESP[1:0] : response
    - [*I*] BID[3:0] : response identifier
    - [*O*] BREADY : ready response

- **ARS-REQ-475 : FIFO AXI sequence**
  Verification method : S
  Parent : CREATED

  The write transaction to the FDDR should be the following:

  - AWID, AWLEN, AWSIZE, AWBURST, AWLOCK, WID, WSTRB signals are fixed.
  - When the AXI master drives the valid address AWADDR, it asserts the AWVALID signal, which remains high till the AXI slave asserts the AWREADY as acknowledgment that the address has been received in a correct way.
  - After that, when the AXI master drives the valid write data WDATA, it asserts WVALID signal. When the last data of the burst is sent, at the same time with WVALID, even WLAST is asserted. WVALID and WLAST remain asserted until AXI slave asserts the WREADY as acknowledgment that the data has been received correctly.

- Finally, the AXI slave asserts the BVALID signal only when it drives the valid write response, BRESP, which corresponds to BRESP=00 when it is correct. The BVALID remains asserted till the AXI master accepts the write response and asserts the BREADY signal.
- The AXI master could have the possibility to assert the BREADY before that the AXI slave asserts the BVALID, in order to terminate the response transfer in only one clock cycle. In this case, it has been chosen to assert the BREADY signal when also WVALID and WLAST signals are asserted.

6.2.6 FIFO access read block

In the FIFO access read block section, similarly to the previous section, the requirements about the signals to interface with the user and with the FDDR, the sequence that the FIFO block has to follow to work properly in reading, the errors that could occur and what is the memory space dedicated to the FIFO read are reported below:

- **ARS-REQ-480 : FIFO read interface**
  Verification method : S
  Parent : URD-REQ-310

  The FIFO read interface should expose the following signals:

  - [*O*] FIFO_RRDY : read ready flag
  - [*I*] FIFO_RQ : read request
  - [*O*] FIFO_DOUT[15:0] : data output
  - [*O*] FIFO_DVALID : data valid
  - [*O*] FIFO_EF : empty flag
  - [*O*] FIFO_RD_ERR1 : write error flag 1
  - [*O*] FIFO_RD_ERR2 : write error flag 2
  - [*O*] FIFO_RD_ERR3 : write error flag 3

- **ARS-REQ-485: FIFO read sequence**
  Verification method : S,T
  Parent : URD-REQ-320

  If FIFO_RRDY is asserted and FIFO_EF is de-asserted, this block should accept a read sequence composed of 1 clock cycle with FIFO_RQ strobe asserted and a burst of 8 data generated on FIFO_DOUT with FIFO_DVALID asserted.

- **ARS-REQ-490 : Error read interface not ready**
  Verification method : S,T
  Parent : URD-REQ-320/405

  If FIFO_RRDY is de-asserted, the access to this port should be discarded and FIFO_RD_ERR3 error flag raised.

- **ARS-REQ-495 : Error FIFO request duration**
  Verification method : S,T
  Parent : URD-REQ-415

  The access to FIFO ports with FIFO_RQ high more than one clock cycle should be discarded and FIFO_RD_ERR2 error flag raised.

- **ARS-REQ-500 : DDR memory partitioning**
  Verification method : S,T
  Parent : URD-REQ-225

  The DDR3 address space allowed for FIFO read is the same dedicated to the writing and it is the following:

  - Starting address: “00000000000000000000000000000000”
  - Final address: “00000000000000000000001111111110"

  The dimension of the memory part dedicated to the FIFO has been chosen as 16 kbytes.

- **ARS-REQ-505 : Error EMPTY FIFO flag asserted**
  Verification method : S,T
  Parent : URD-REQ-410

  The access to FIFO read port should be discarded if FIFO_EF is asserted and FIFO_RD_ERR1 error flag raised.

- **ARS-REQ-510 : FIFO read AXI interface**
  Verification method : R,S
  Parent : CREATED

  FIFO block should interface FDDR block through the following AXI signals:

  - Read Access channel :
    - [*I*] ARREADY : read address ready
    - [*O*] ARVALID : read address valid
    - [*O*] ARADDR[31:0] : read address
    - [*O*] ARID[3:0] : read address identifier → ARID=0000
    - [*O*] ARLEN[3:0] : read address length burst length → AWLEN=0001
    - [*O*] ARSIZE[1:0] : read address size → ARSIZE=11
    - [*O*] ARBURST[1:0] : read address burst → ARBURST=01
    - [*O*] ARLOCK[1:0] : read address lock → ARLOCK=00
  - Read Data channel :
    - [*I*] RVALID : read data valid
    - [*I*] RLAST : read data last
    - [*I*] RDATA[15:0] : read data
    - [*I*] RID[3:0] : read identifier
    - [*O*] RREADY : read data ready
  - Read Response channel :
    - [*I*] RRESP[1:0] : response

- **ARS-REQ-515 : FIFO AXI sequence**
  Verification method : S
  Parent : CREATED

  The read transaction to the FDDR should be the following:

  - ARID, ARLEN, ARSIZE, ARBURST, ARLOCK, RID signals are fixed
  - When the AXI master drives the valid address ARADDR, it asserts the ARVALID signal, which remains high till the AXI slave asserts the ARREADY as acknowledgment that the address has been received in a correct way.

- Subsequently, the AXI slave asserts the RVALID signal when it drives the valid read data. When the sent data is the last one, together with the RVALID, the RLAST signal is asserted. RVALID and RLAST remain high till the AXI master receives the data and asserts the RREADY signal as acknowledgment that the data has been received correctly.
- The RREADY signal could be asserted previously with respect to the assertion of RVALID and RLAST, if the AXI master is ready to receive a data. In this case, it has been chosen to assert the RREADY after that the ARREADY has been asserted from the AXI slave.

6.2.7 Arbiter

- **ARS-REQ-520 : Arbiter function**
  Verification method : S
  Parent : URD-REQ-220

  This block should transparently arbitrate access to all above interfaces. It is composed of 2 sub-arbiter, one that manage the writing and one that manage the reading. When both direct access and FIFO want to write at the same time, the write arbiter chooses which one could pass and puts the other waiting. The read arbiter does the same when both direct access and FIFO want to read at the same time. Both the sub-arbiters manage the choice using Round Robin scheduling.

- **ARS-REQ-525 : Concurrent access**
  Verification method : S
  Parent : URD-REQ-515

  An access to all four ports should be implemented. Then, these accesses should be managed by the arbiter, which queues these and serves one after the other with Round Robin scheduling.

Chapter 7

HDL implementation

In this chapter, it is described the flow of the HDL implementation of the blocks, which compose the high-speed data buffer controller, called AXI master block in Figure 6.2.
This block has been written in VHDL language and has been executed on LiberoSoC, which is used to create the structure to simulate the execution on particular types of FPGAs, in this case, the RT4G150 FPGA, since LiberoSoC contains a library with specific IP components that could be characterized, based on the specification of the project.
The first step of implementation has been the development of the direct access block, composed of a part that controls the writing and a part which controls the reading. Subsequently the same has been done for the FIFO block. Then, the last part of the AXI master block, that has been designed, is the arbiter, which has the aim to decide who between the FIFO and the direct access has the priority to read or write the DDR3 memory.
The clock frequency of the AXI master block and all its sub-blocks is 80 MHz and the reset has been set as asynchronous, so it could arrive at any moment, and active low, so it is asserted when it is equal to 0.

7.1 Direct Access block

One of the interfaces of the high-speed data buffer controller, available for other blocks, is the direct access interface, with which the user wants to write to or read from the DDR3 memory banks only one data.
In this scenario, the Direct Access block aim is to translate the signals sent from the user through the direct access interface into signals compatible with the AXI interface, whose operation is reported in Section 5, and vice-versa. This block is composed of two different blocks, one is the Write Direct Access block, which has the goal to manage the writing of a data sent from the user, converting in a correct way the direct access signals, reported at the **ARS-REQ-300** requirement in Subsection 6.2.3, to AXI signals, reported at the **ARS-REQ-330** requirement always in Subsection 6.2.3; and the other is the Read Direct Access block, which has to administrate the reading of a data from the memory, converting, even in this case, the direct access signals, reported at the **ARS-REQ-400** requirement in Subsection 6.2.4, to AXI signals, reported in the same section at the **ARS-REQ-430** requirement.
The behaviour of these two sub-blocks is better explained in the below subsections.

7.1.1 Write Direct Access block

The Write Direct Access block has been implemented through only a Finite State Machine (FSM), instead of a Finite State Machine and a data-path, since, inside the FSM, the behaviour of a hypothetical data path is emulated. This block has been done in this way in order to weight less the RT4G150 FPGA.
The scheme of the Finite State Machine, which implements the writing of a data in memory, translating the write direct access signals to the write AXI signals, is reported in Figure 7.1.

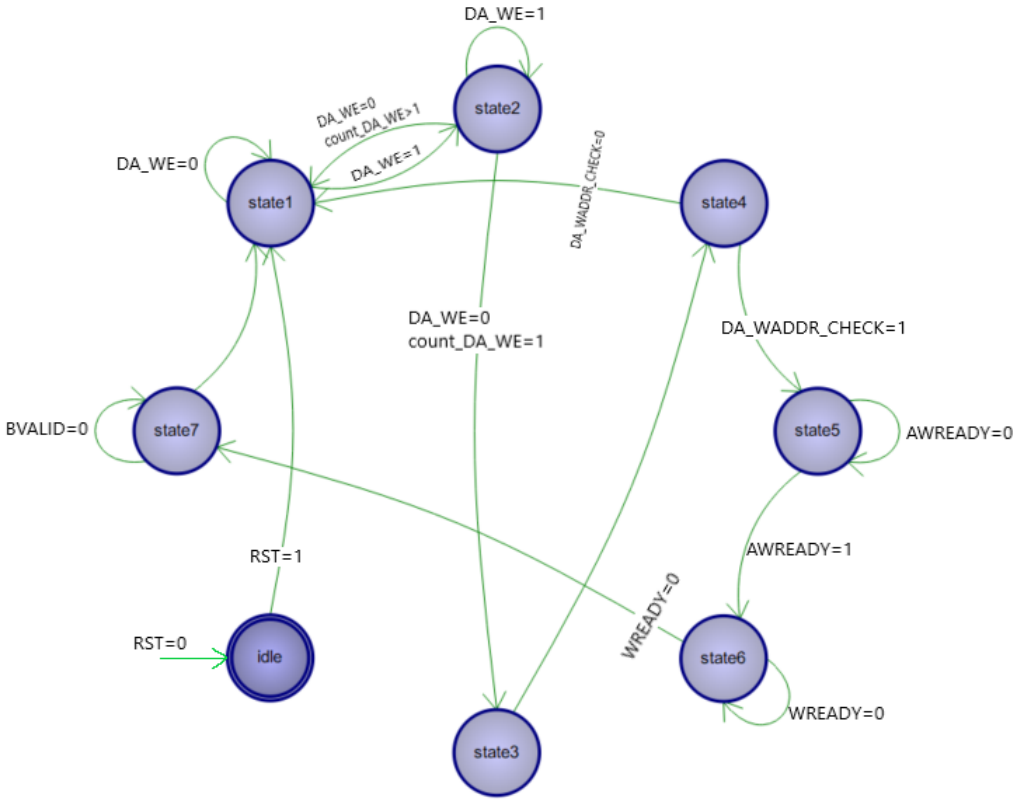

Figure 7.1: Finite State Machine of Write Direct Access block

The machine starts from the *IDLE* state, in which the signals are initialized and in which the system goes at any time in which the **reset** is asserted (RST=0, since it is active low). When the reset signal is de-asserted, so, RST=1, the FSM moves from the *IDLE* to the *STATE1*, in which the **DA_WRDY** is asserted (DA_WRDY=1), in order to indicate to the user that the machine is ready to execute a writing, since it is not busy. The machine remains in this state till the user asserts the **DA_WE** signal (DA_WE=1), which indicates that the user wants to send a data in memory. Thus, when the direct access write enable (DA_WE) is asserted, the FSM goes from *STATE1* to *STATE2*, in which DA_WRDY is de-asserted, meaning that the system is not ready to accept any other writing, since a writing is already in progress. The writing is accepted only if (DA_WE) has a duration of one clock cycle. In fact, the FSM uses the **count_DA_WE** signal to count how many clock cycles the DA_WE signal is raised. If count_DA_WE is higher than one means that the DA_WE signal has been high for more than one clock cycle and it is not acceptable, as it is possible to understand from the **ARS-REQ-315**

requirement in Subsection 6.2.3, which explains that an access to the write direct access port with a duration greater than one clock cycle has to be discarded. To signalize this error, the **DA_WR_ERR1** is raised, thus, the user could understand something wrong has happened. Therefore, if this error occurs, the FSM comes back from the *STATE2* to *STATE1* and this writing is discarded, making the system ready again to accept a writing request. While, if count_DA_WE is equal to one, it means that the signal DA_WE is high only one clock cycle and so, the specification is respected. In this case, the Finite State Machine goes from *STATE2* to *STATE3*, in which the memory address is sent from the user to the FDDR block, which is placed between the AXI master block and the DDR3 memory banks, so, the **DA_WADDR** of the direct access interface is assigned to **AWADDR** of the AXI interface. Always in this state, the write data (**DA_DIN**) sent from the user is assigned to the **WDATA** signal of the AXI interface. For this operation, the data sent from the user, which is composed of 16 bits (2 bytes), has to be stretched to 64 bits (8 bytes), since the WDATA is expressed on 64 bits.
From this state to the last state there is the check of the error, which happens when the DA_WRDY is de-asserted, so the system cannot accept another writing, but the user sends DA_WE high. From here on out, if this condition occurs, the **DA_WR_ERR3** signal is raised, to satisfy the **ARS-REQ-310** requirement, which says that if DA_WRDY is de-asserted, the access to this port has to be discarded and DA_WR_ERR3 error flag raised. In fact, in this case, the writing is not accepted.
Then, the machine moves from *STATE3* to *STATE4*, in which the **ARS-REQ-325** requirement is implemented, in fact, here the address sent from the user is checked, to understand if it belongs to the part of memory dedicated to the direct access or to the part of memory dedicated to the FIFO.
Thus, to implement this check, the signal **DA_WADDR_CHECK** is employed. In fact, if the DA_WADDR address belongs to the FIFO part, the DA_WADDR_CHECK is de-asserted and the finite state machine understands that the address is wrong and that this writing has to be discarded. To signalize this problem to the user, the DA_WR_ERR2 error flag is raised, as the **ARS-REQ-325** requirement asks. In fact, it says that an access to the direct access port with DDR address reserved for FIFO has to be discarded and DA_WR_ERR2 error flag raised. While, if the address belongs to the direct access, the DA_WADDR_CHECK signal is raised, the DA_WR_ERR2 signal remains de-asserted and the system moves from *STATE4* to *STATE5*.
In this state, the **AWVALID** signal of the AXI interface is asserted (AWVALID=1) in order to signalize to the AXI slave block, which is represented by the FDDR, that the sent address is valid. Thus, the FDDR controller answers with an acknowledgment signal, **AWREADY**, which is asserted to mean that the address has been received correctly.
At this point, the FSM could go from *STATE5* to *STATE6*, in which the signal which are asserted are: **WVALID**, **WLAST** and **BREADY**. The WVALID is asserted in order to communicate to the slave that the data sent from the master is valid and WLAST is raised since the direct access interface sends only one data, thus, the first data is also the last, and to respect the AXI interface procedure, reported in Section 5, these two signals remain asserted till the slave asserts the **WREADY** signal as acknowledgment that the data has been received.
The BREADY signal is asserted in this state, because, as it has been explained in Section 5, this signal could be asserted before the slave asserts the **BVALID** signal in order to complete the response transfer in one cycle.
Thus, when the slave asserts the WREADY, the master FSM goes from *STATE6* to the last state *STATE7*. Here, the WVALID e WLAST signal are de-asserted, while BREADY

remains asserted until the slave sends the BVALID signal. It is raised by the slave only when it drives the valid write response, **BRESP**. When the response is valid, the BRESP assumes the value of "00".
Therefore, the system is stalled in *STATE7* till BVALID=0, while it moves from *STATE7* to *STATE1* only when BVALID=1.
In Figure 7.2, the inputs and the outputs to the write block are reported.

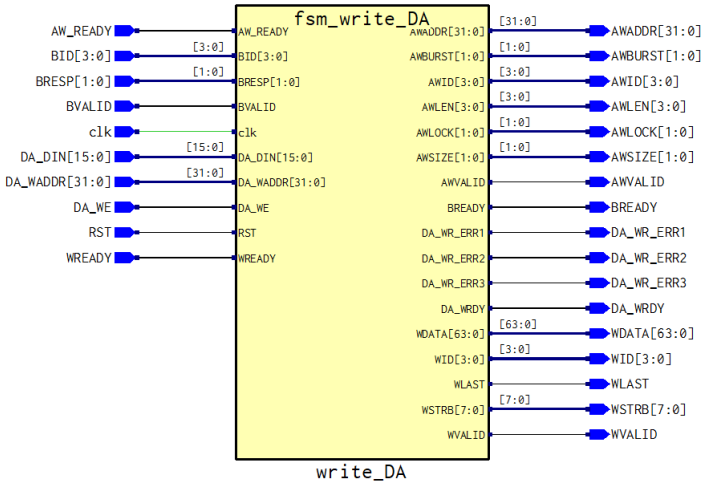

Figure 7.2: Direct Access write block

In particular, the clk and RST signals are produced respectively by a 50 MHz oscillator plus a PLL and reset generator, the DA_WE, DA_DIN and DA_WADDR are sent from the user to the AXI master, the DA_WRDY, DA_WR_ERR1, DA_WR_ERR2 and DA_WR_ERR3 from the master to the user, the AWADDR, AWBURST, AWID, AWLEN, AWLOCK, AWSIZE, AWVALID, BREADY, WDATA, WID, WLAST, WSTRB and WVALID are sent from the AXI master (high-speed data buffer controlled) to the AXI slave (FDDR), and as last AWREADY, BID, BRESP, BVALID and WREADY are sent from the AXI slave to the AXI master.
Here, as in the following blocks, there are some signals, that the master forwards to the slave, which are maintained as constant, in order to say to the FDDR how to configure it.
These signals are:

- AWID[3:0]=0000, which indicates the identification TAG for the write address group of signals;
- AWLEN[3:0]=0000, which indicates the burst length, thus, the exact number of transfers in a burst. Since in this case, the number of data transferred in a burst is one, this value is set to "0000", which corresponds to 1 for the FDDR, as it is possible to observe from the indications in Table 6.2;
- AWSIZE[1:0]=11, which indicates the maximum number of data bytes to transfer in each data transfer. In this case, the attributed value is "11", which corresponds to 8 bytes for the FDDR. This value is set to 8, since, when the input data (DA_DIN) is assigned to WDATA, it is stretched from 16 bits to 64 bits, since WDATA has this length, so, the data that the user wants to write in memory is not of 16 bits, but 64 bits, which correspond to 8 bytes;

- AWBURST[1:0]=01, which indicates the burst type. In this case, the set burst type corresponds to the **INCR** type, with which the address of each data sent in memory is determined by increasing sequentially the address sent by the user;
- AWLOCK[1:0]=00, which indicates the lock type, giving additional information about the atomic characteristics of the write transfer. In this case, "00" corresponds to a **normal access**;
- WID[3:0]=0000, which indicates the response ID, so, the identification TAG of the write response;
- WSTRB[7:0]=11111111, which indicates which byte lanes to update in memory, based on the number of bytes of which is composed the write data. For example, if the data is composed of 24 bits, which corresponds to 3 bytes, the WSTRB value would be "00000111". In the case of the direct access interface, since the data has been stretched to 64 bits, which are equal to 8 bytes, all the byte lanes have to be put to 1.

7.1.2 Read Direct Access block

Even the Read Direct Access block has been implemented only through a Finite State Machine, without a data path, in order to weight less the FPGA when the code is loaded inside.
The scheme of the FSM, which implements the reading of a data in memory, translating the read direct access signals, coming from the direct access interface, to the read AXI signals, belongs to the AXI interface, in reported in Figure 7.3.

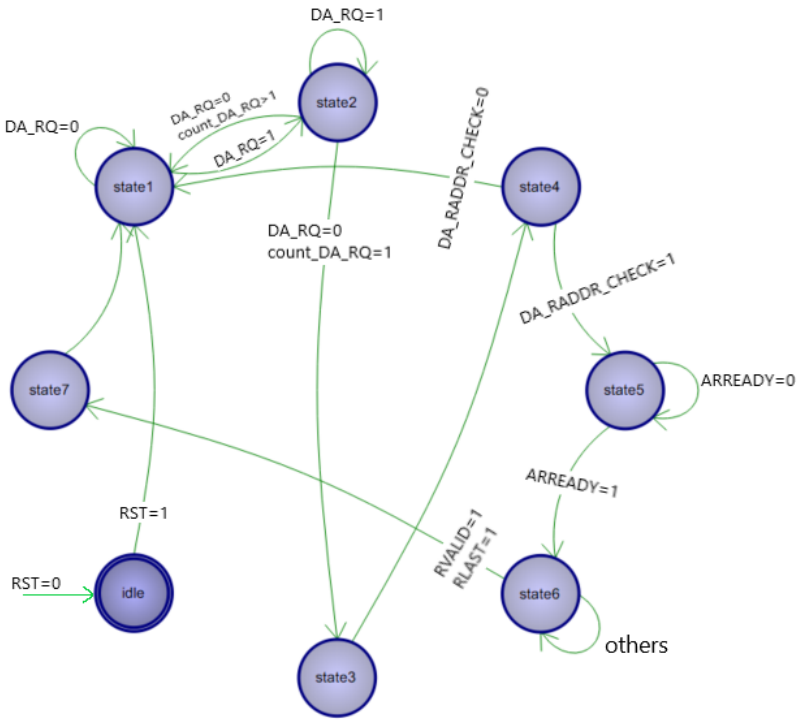

Figure 7.3: Finite State Machine of Read Direct Access block

As the previous FSM, the read direct access finite state machine starts from the *IDLE*

state, in which the signals are initialized and in which the system goes at any time in which the **reset** is asserted (RST=0). The machine goes from the *IDLE* state to the *STATE1*, when the RST=1, so when the reset signal is de-asserted. At this point, the reading cycle could start. In *STATE1*, the **DA_RRDY** signal is asserted, in order to permit the beginning of a reading if the user requires it. The system remains in this state till the user asserts the DA_RQ signal (DA_RQ=1), which corresponds to the reading request.
At this point the FSM moves to the *STATE2*, in which the DA_RRDY is de-asserted since the system, in this case, is busy and it could not start another reading. In this state, there is a signal, **count_DA_RQ** which has to count how many times the DA_RQ signal remains high, since, based on the **ARS-REQ-415** requirement, the access to the direct access port with a duration greater than one clock cycle has to be discarded. Thus, if count_DA_RQ is higher than one, it means that the DA_RQ has been high for more than one clock cycle, and it leads to the discard of this reading and the machine comes back to the *STATE1*, raising the **DA_RD_ERR1** error flag, in order to communicate to the user that an error has occurred. While, if count_DA_RQ is equal to 1, it means that the operation is correct and the system could go forward to *STATE3*, in which the address **DA_RADDR**, coming from the user, is assigned to the **ARADDR** signal, which is the address that is sent from the AXI master to the AXI slave.
Moreover, from this state to the last state, the **ARS-REQ-410** requirement is checked, in fact, from now on, if the user tries to access this port, being DA_RRDY de-asserted, has to be discarded and the **DA_RD_ERR3** error flag is asserted.
After this state, the FSM moves to *STATE4*, in which there is the address check, in order to understand if the address belongs to the direct access memory part and so it is correct or it belongs to the FIFO memory part and it is wrong.
If the address is wrong, the FSM does not assert the **DA_RADDR_CHECK** signal and the machine comes back to **STATE1** since the reading is discarded, raising the **DA_RD_ERR2** in order to satisfy the **ARS-REQ-425** requirement. While, if the address is correct, the DA_RADDR_CHECK is asserted and the finite state machine could move from *STATE4* to *STATE5*. If the machine arrives in this state, it means that nothing has gone wrong. Thus, at this point, the **ARVALID** signal could be asserted from the AXI master, in order to communicate to the AXI slave the address in which the user wants to read the data in memory is valid. The FSM is stalled in *STATE5* till the slave asserts the **ARREADY** signal, as acknowledgment that the address has been received correctly to the slave.
Thus, when ARREADY=1, the system goes from *STATE5* to *STATE6*. In this state, the ARVALID signal is de-asserted and the **RREADY** signal is asserted to 1 in order to communicate to the slave that the master is ready to receive the data, read from memory in the address sent by the user at the beginning of the reading procedure.
Therefore, only when the slave will be asserted at the same time as the **RVALID** and the **RLAST** signals, the machine will be moved to the last state, *STATE7*. While, for the conditions different from RVALID=1 and RLAST=1, the machine remains in *STATE6*.
When the FSM arrives in *STATE7*, the **RDATA** signal is assigned to the output data, DA_DOUT, of the direct access interface, and the DA_DVALID signal is asserted in order to communicate to the user that the read data is valid and it could be accepted by the user.
What is important happens in this state is that the length of the RDATA is adapted to the length of the DA_DOUT when it is assigned, since the length of data read from memory is 64 bits, while the length of the output data is 16 bits, since, before assigned

the RDATA to DA_DOUT, the first one has to be cut, in order to obtain a correct dimension.
After this state, the FSM comes back to the first state *STATE1* and it could start another reading if the user requests it.
In Figure 7.4, the inputs and the outputs to the read block are shown.

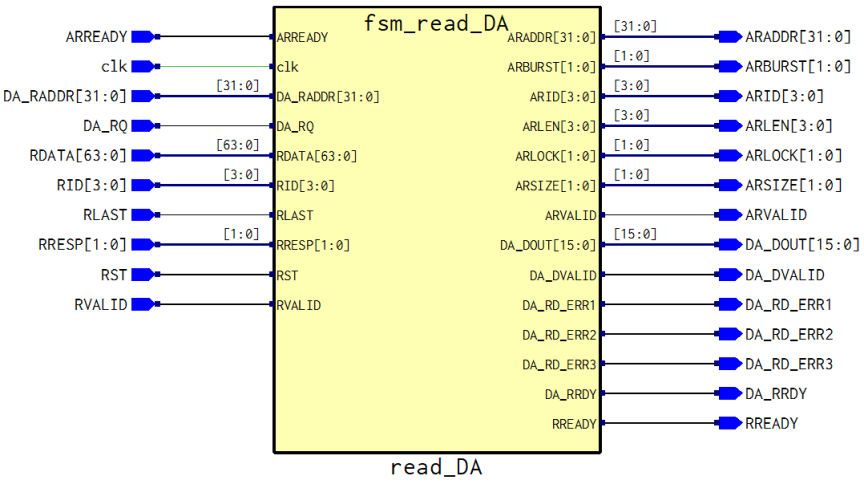

Figure 7.4: Direct Access read block

As before, the clk signal is generated by the combination of a 50 MHz oscillator and a PLL, while the RST derives from a reset generator.
The DA_RADDR and DA_RQ are sent from the user to the read direct access block of the AXI master, the DA_DOUT, DA_DVALID, DA_RD_ERR1, DA_RD_ERR2, DA_RD_ERR3 and DA_RRDY are sent from the AXI master to the user, the ARREADY, RDATA, RID, RLAST, RRESP, RVALID are sent from the AXI slave (FDDR) to the AXI master, and as last, the ARADDR, ARBURST, ARID, ARLEN, ARLOCK, ARSIZE, ARVALID and RREADY are sent from the AXI master to the AXI slave.
In this case, the signals which remain constant in order to configure in a correct way the FDDR are:

- ARID[3:0]=0000, which indicates identification TAG for the read address group of signals;
- ARLEN[3:0]=0000, which indicates the burst length, that gives the exact number of transfers in a burst. In this case, "0000" corresponds to 1 for the FDDR, in order to communicate to it that the read data which has to be read in memory is only one;
- ARSIZE[1:0]=11, which indicates the maximum number of data bytes to transfer in each data transfer. "11" corresponds to the value 8 for the FDDR, since the length of the data, which is read from the memory, is of 64 bits, that correspond to 8 bytes. Then, this length will be cut to adapt the 64 bits to the 16 bits of the output data from the direct access interface;
- ARBURST[1:0]=01, which indicates the burst type. In this case, "01" corresponds to the **INCR** option, which means that the addresses of the data are determined starting from the address sent from the user and increasing sequentially;
- ARLOCK[1:0]=00, which indicates the lock type, providing additional information

about the atomic characteristics of the read transfer. In this case, "00" means a **normal access**.

7.1.3 Write and Read Direct Access blocks assembling

After that, the read and write direct access blocks have been implemented and their correct behaviours have been checked separately by means of simulation on ModelSim, these have been reunited in a unique AXI master block, as reported in Figure 7.5.

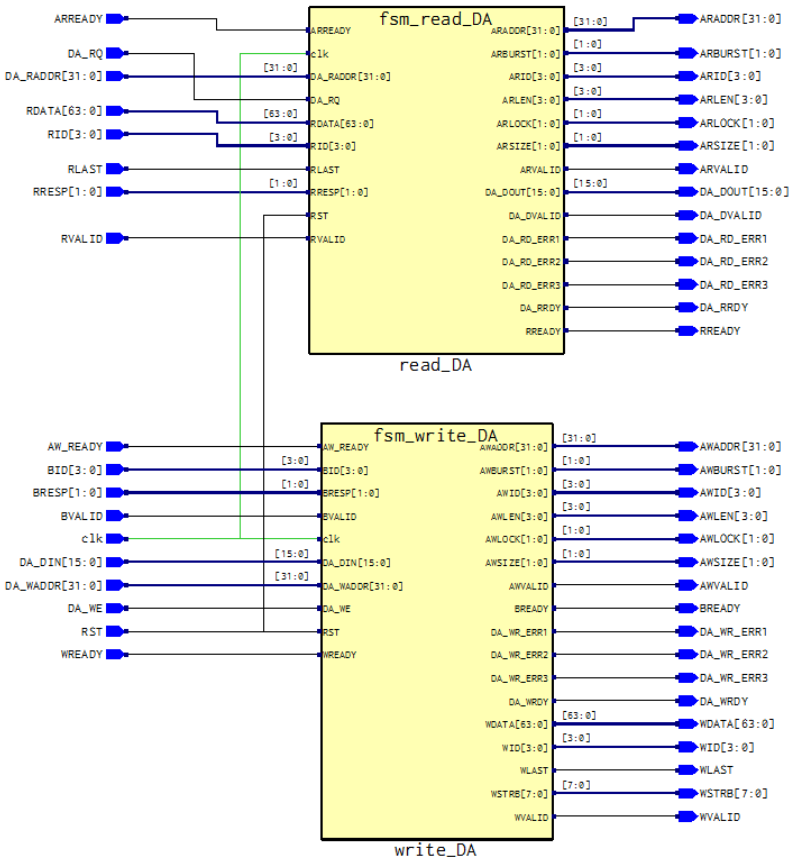

Figure 7.5: Write and Read Direct Access AXI block

At this point, with LiberoSoC, a structure that contains also the FDDR block (AXI slave), the DDR3 memory banks and the blocks to generate the clocks and the reset in a correct way has been implemented, in order to check that the AXI master, which at the moment is composed only of the write and read direct access blocks, could write to and read from the DDR3 memory banks the data correctly.
To understand better all the steps to implement this structure, in Figure 7.6 the structure with the AXI master (high-speed data buffer controller), the AXI slave (FDDR controller) and all the blocks to make this structure work properly is reported.

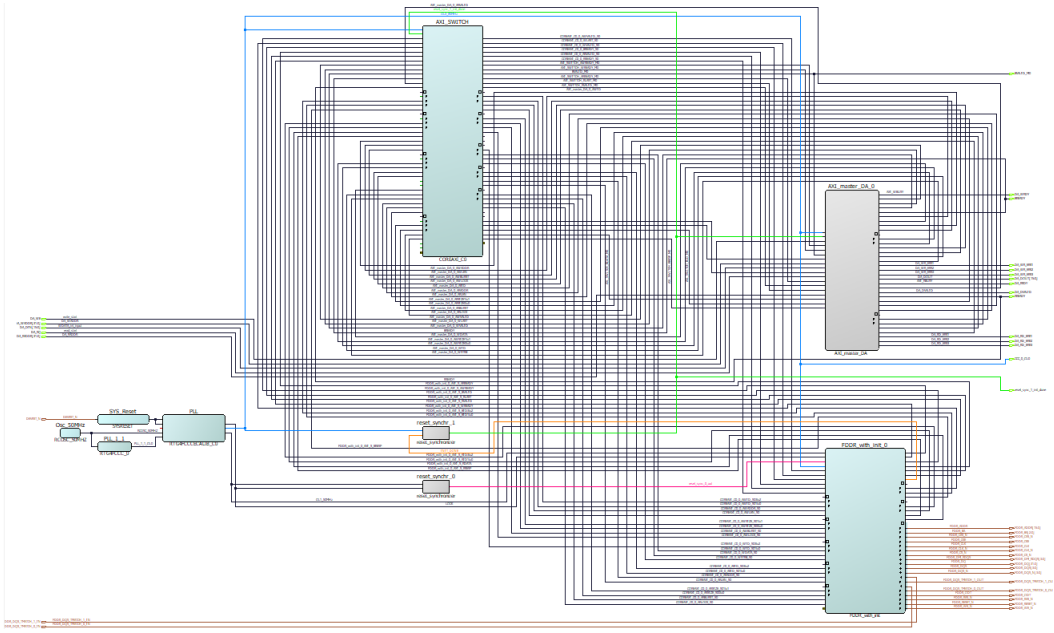

Figure 7.6: Structure with AXI master and AXI slave

Since this structure is too big and complex to be analyzed correctly from this figure, zooms for each block will be shown below. The first step to realize this structure is to generate the 50 MHz and 80 MHz clocks and the user logic reset.
In order to realize these, a 50 MHz oscillator (Osc_50MHz), a PLL 1:1 (PLL_1_1), a dedicated reset (SYS_Reset) and a PLL have to be used. All these components, which zoom is reported in Figure 7.7, are part of the Libero SoC library for this type of FPGA.

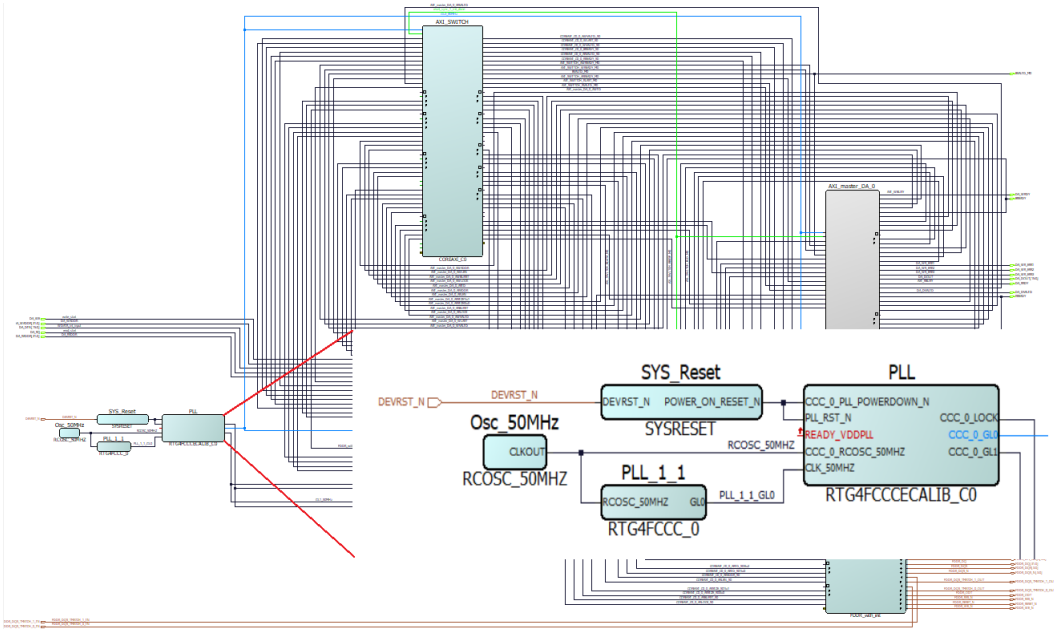

Figure 7.7: Zoom of 50 MHz, PLL and system reset blocks

Some of these blocks have to be configured to obtain the desired operation.
Starting from the **PLL 1:1**, to the input of this one is connected the 50 MHz oscillator.

Inside the PLL 1:1 there is a Clock Conditioning Circuitry (CCC), that could be observed in Figure 7.8. In this block, the reference clock is set to the 50 MHz of the oscillator at the input and at the output only one (GLO) of the four outputs has been chosen, in order to implement the 1:1 relation between input and output.
This block is used in order to obtain a stable output clock of 50 MHz, in fact, the generic aim of the PLL is to lock to a specific frequency, in order to obtain a frequency which is stable and that does not suddenly change.

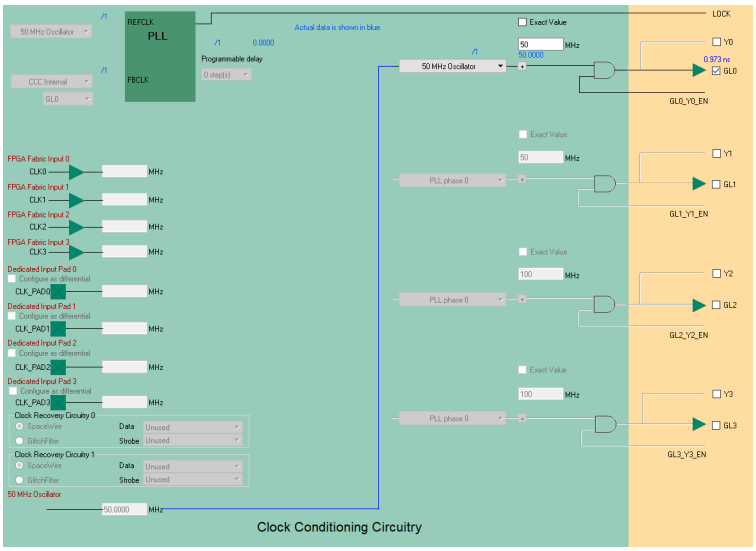

Figure 7.8: Clock Conditioning Circuitry PLL 1:1

The block called PLL is an **enhanced PLL Calibration Configuration**.
In this case, at the output there is not only one clock frequency, but two clock frequencies, as it is shown in Figure 7.9. Here, at the input there is the 50 MHz which arrives from the PLL 1:1 and at the output the desired clocks are: 50 MHz, which is the service frequency, and 80 MHz, which is the working frequency.

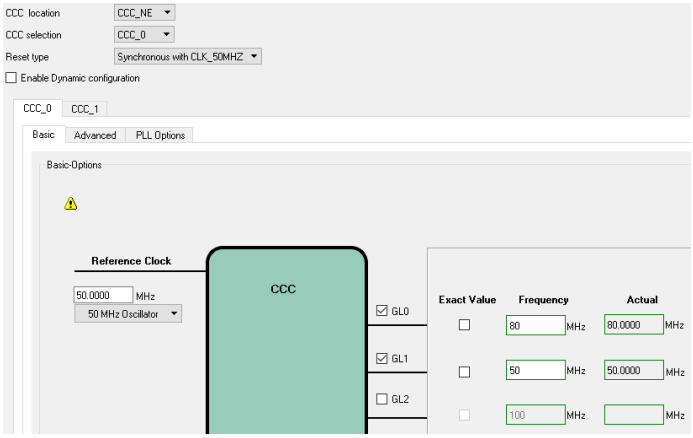

Figure 7.9: Basic option Clock Conditioning Circuitry PLL

In Figure 7.10, there is the advanced vision of this PLL. From here, it is possible to understand that it is different from the structure of the PLL 1:1, since before the 50 MHz reference clock is directly connected to the output, while here, this clock is connected to

the PLL internal to the Clock Condition Circuitry (CCC), which output is connected to the 2 outputs GL0 and GL1 of the CCC in a PLL phase 0.
Thus, at the output of this block there are 3 signals, the 50 MHz and 80 MHz clocks, which are stable, and a LOCK signal, which is the user logic reset.

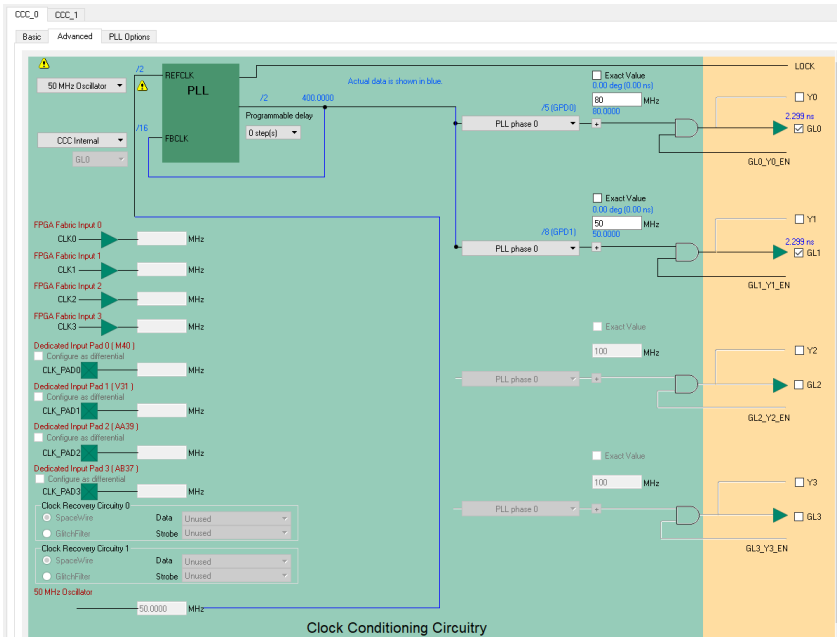

Figure 7.10: Advanced Option Clock Conditioning Circuitry PLL

Moreover, in this block there are some options that could be set, as shown in Figure 7.11. Here, one of the more important important option is the last one in the Miscellaneous section, that is the **Enable Auto-Reset of PLL on Loss of Lock** option. Therefore, if the PLL loses its lock and so the frequencies generated are no more stable, the PLL is reset automatically, in order to restart the procedure to lock again the PLL.

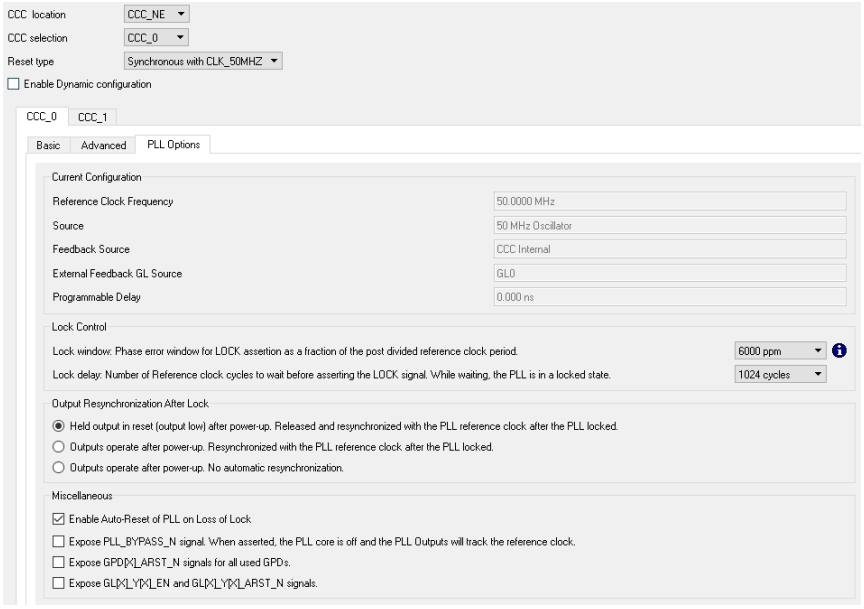

Figure 7.11: PLL options Clock Conditioning Circuitry PLL

Another zoomed block of this structure is reported in Figure 7.12. Here, there are 2 **reset synchronizers**, one for the reset related to the 50 MHz clock and one for the reset related to the 80 MHz clock.

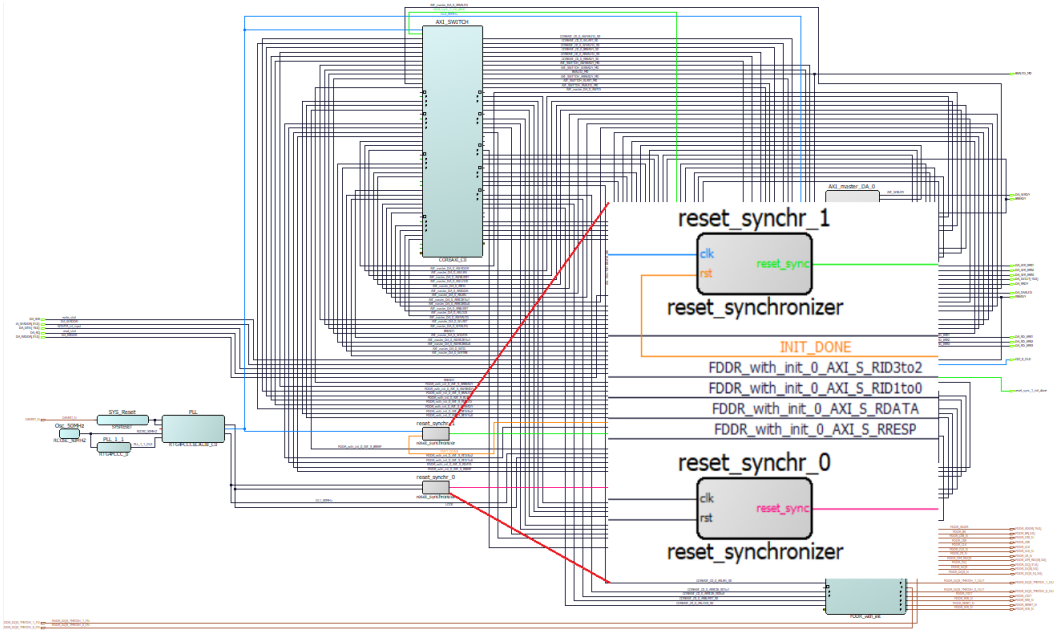

Figure 7.12: Zoom of Reset synchronizers

These blocks are used since the reset is asynchronous and the design needs an asynchronous assertion of this signal, but a synchronous de-assertion. Thus, when the reset is asserted, it does not follow the toggle of the clock, while when the reset is de-asserted, it waits for a clock edge.
Pay attention that this reset is not the top-level reset that is asynchronous both in assertion and during the de-assertion, but is the user logic reset, internal to the circuit.
Therefore, in other words, a reset synchronizer manipulates the asynchronous reset to have synchronous de-assertion.[13]
The internal scheme of the reset synchronizer is shown in Figure 7.13 in the red rectangle. It is composed of two flip flops connected in series, in which the input of the first register is tied to 1. These flip flops receive the same clock and the same asynchronous reset.

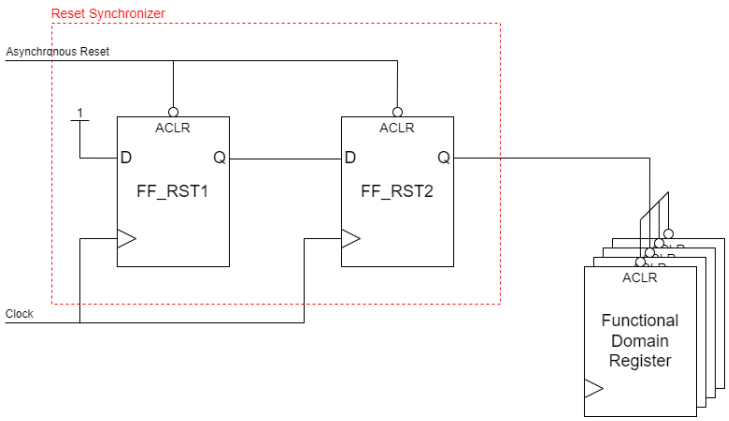

Figure 7.13: Reset Synchronizer scheme

In Figure 7.14, the timing related to the behaviour of the reset synchronizer is reported. From this image, it is possible to understand why the reset synchronizer is used, since when the asynchronous reset is de-asserted, so, it goes to 1, since the reset is active-low, there is not a certainty that the reset signal is raised before or after a certain raising edge of the clock signal. So, in order to obtain a reset which is de-asserted synchronously with the clock signal, when the reset is de-asserted, it is first propagated to reset synchronizer flip flops. Thus, the first flip flop in the chain propagates 1 to intermediate output upon arrival of a clock edge. Upon the next clock edge, this signal propagates to the output thereby reaching the fanout registers.
Therefore, the Q output signal of FFRST2 represents the negative reset which is de-asserted synchronously.

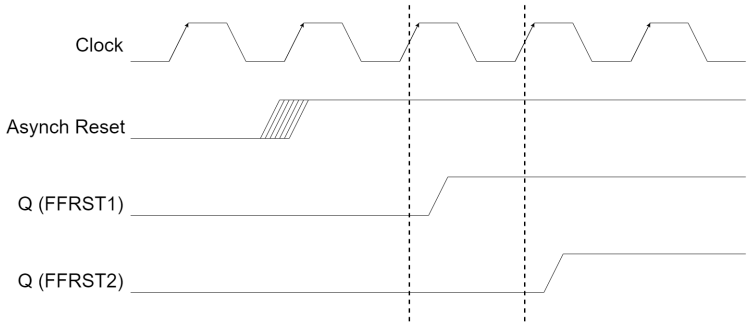

Figure 7.14: Reset Synchronizer timing diagram

Another zoom is dedicated to one of the most important blocks, the **Direct Access AXI master** block, which is reported in Figure 7.15. Inside this block, there are reunited the write direct access block and the read direct access one. The input to this block is sent or from the user, which is represented by an emulator, as it will be shown later, or from the FDDR as a response to some stimuli, and vice-versa, its output is sent to the user or the FDDR.

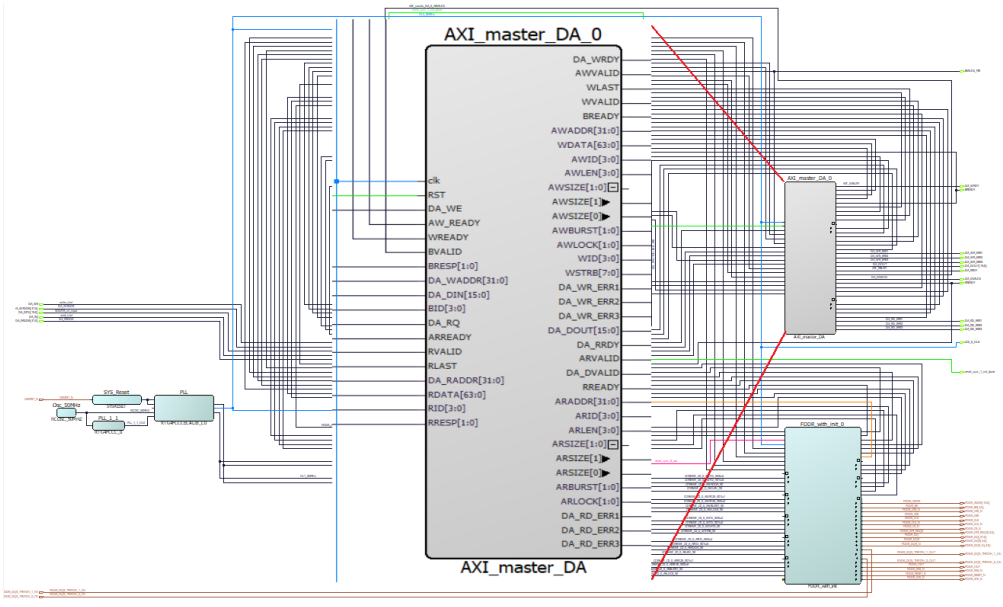

Figure 7.15: Zoom Direct Access AXI master block

At this point, another block could be analyzed and it is the **AXI switch** block. It is used in the case in which the number of AXI masters, that want to speak with the AXI slave, is higher than one. In that case, it is used as an arbiter to decide which has the priority to communicate with the AXI slave block, which is represented by the FDDR controller.
Even if, in this project, the AXI master block is only one, this block has to be put caused by the rule of the RT4G150 FPGA, but this block turns out to be transparent, since it does not have to decide which one of the AXI master has the priority, since the master is only one.

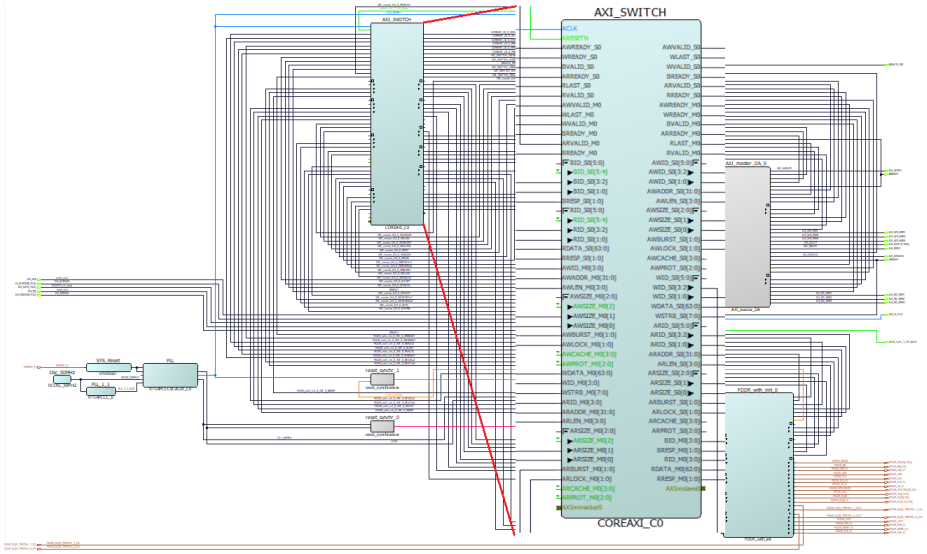

Figure 7.16: Zoom AXI switch

The last block of this structure and one of the most important blocks, is the AXI slave, that represents the FDDR controller, which is used to communicate in a correct way with the DDR3 memory banks.

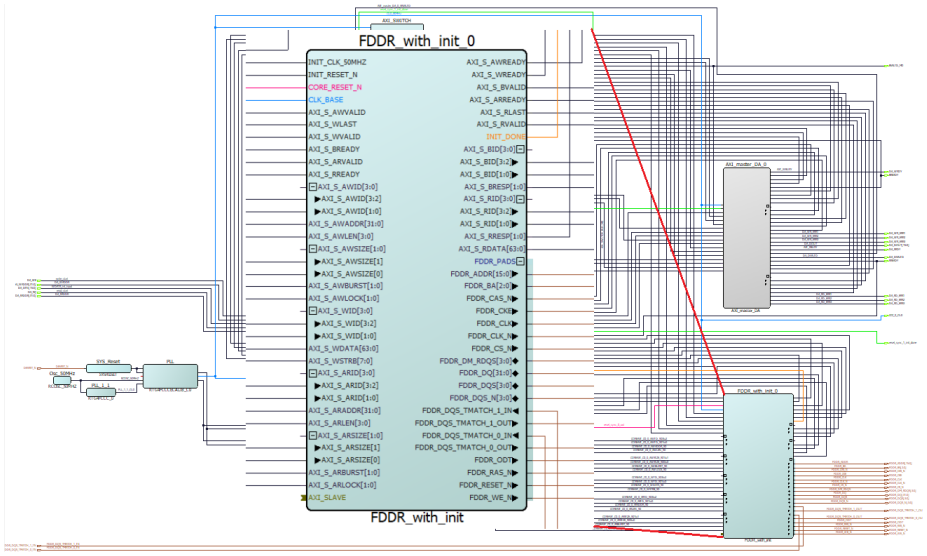

Figure 7.17: Zoom FDDR AXI slave

In Figure 7.17, this block is shown and it has been set following the **ARS-REQ-220** requirement.
Thus, in Figure 7.18, 7.19 and 7.20 there are the options that have been set based on the requirement.
In the below figure, the general options are set, in which it is decided which of the FDDRs present on the FPGA has to be used, if the East or the West one, the type of memory banks which have been used and so no. The motivation, for which there are two separate FDDRs, is that on the board there are two different sets of memory banks on the East and West sides.

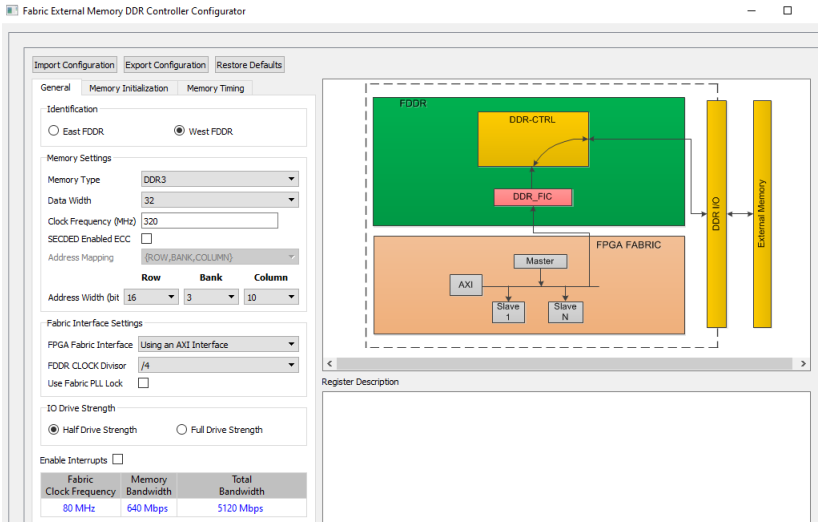

Figure 7.18: General section of FDDR configuration

While, in Figure 7.19, the memory initialization options are selected, choosing the burst length, its order and other more detailed peculiarities, related to the latency of the memory signals and some information about different types of refreshes.

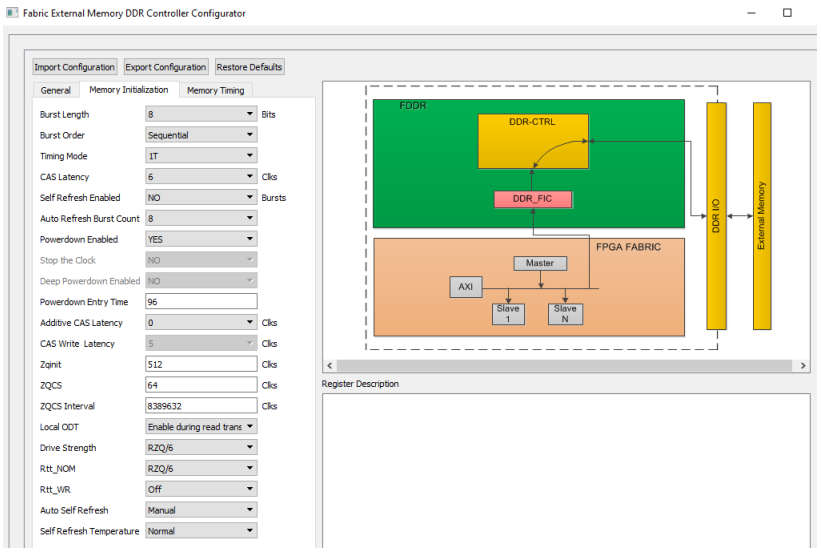

Figure 7.19: Memory initialization section of FDDR configuration

In the figure below, there are the last options related to the FDDR controlled, which

are the memory timing options.

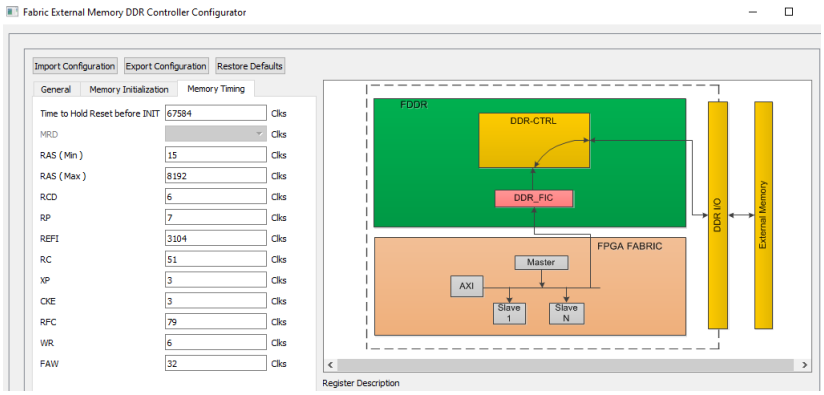

Figure 7.20: Memory timing section of FDDR configuration

Therefore, all these blocks are reunited in a unique block, called **AXI block**, which has been connected to the DDR3 memory banks and the emulator, which simulates the user behaviour. This structure, reported in Figure 7.21, represents the bench.

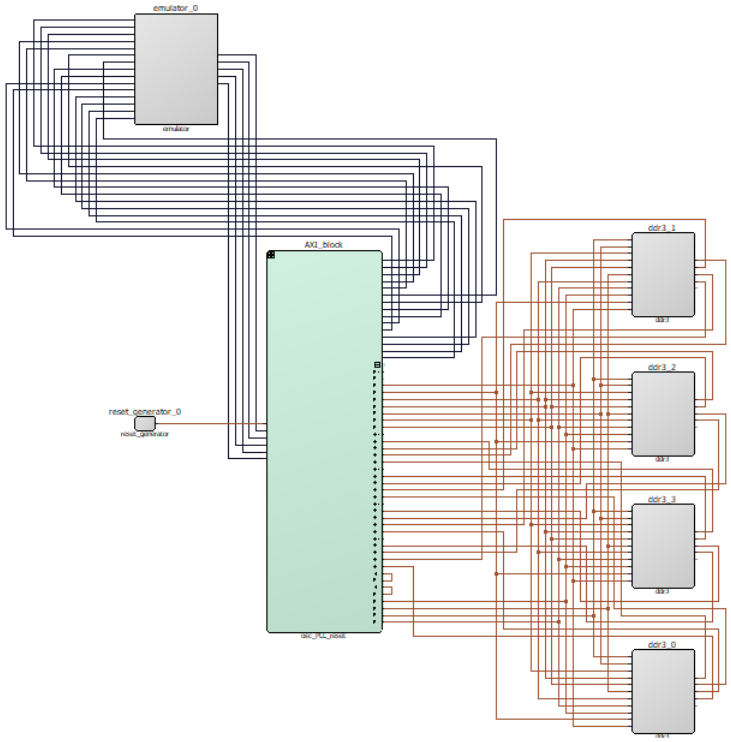

Figure 7.21: Complete structure RTG4 (Bench), with emulator and DDR3 memory blocks

Even in this case the scheme is too complex to be observed clearly, thus, for each block a zoom is necessary as in the previous cases.
In Figure 7.22, the zoom of the AXI block is reported. This block has inside the entire structure described before, in Figure 7.6. In fact, observing the inputs and the outputs of this block, these coincident with the top level inputs and outputs of the block in Figure 7.6. Between the signals, there are the FDDR_DQS_TMATCH_0_IN, FDDR_DQS_TMATCH_0_OUT, FDDR_DQS_TMATCH_1_IN and

FDDR_DQS_TMATCH_1_OUT. These signals, which are connected in pairs, are the DQS enable input used in the FPGA for timing match between DQS and system clock, computing the wire delay.

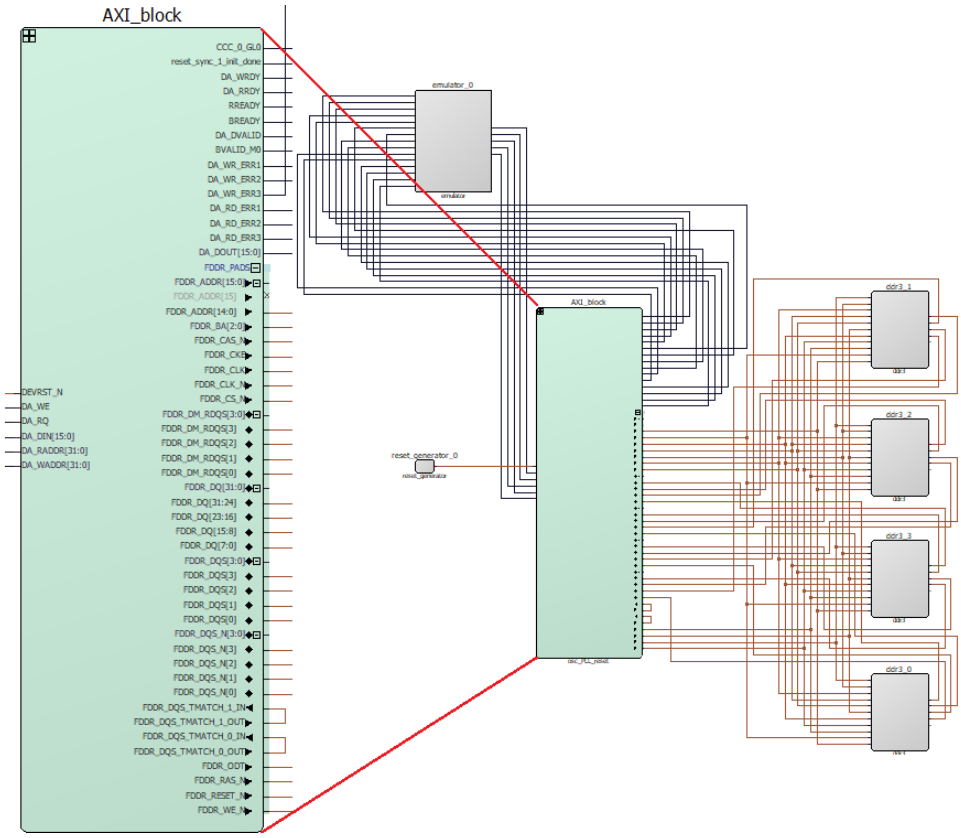

Figure 7.22: Zoom of the AXI block

While, in Figure 7.23, the zoom of the emulator block is shown. This block has the aim to emulate the user both in writing and in reading, in fact, between the outputs, it is possible to observe the requested for writing to (DA_WE) and reading from (DA_RQ) the DDR3 memory banks, the write (DA_WADDR) and read (DA_RADDR) addresses and the input data (DA_DIN) to write in memory. While, as inputs, there are the signals that communicate to the user that the AXI master is ready to receive a specific request, the signals that refer to the user if there are some errors and which types of errors, the output data, which is the data read from one of the memory banks and some acknowledge signals.

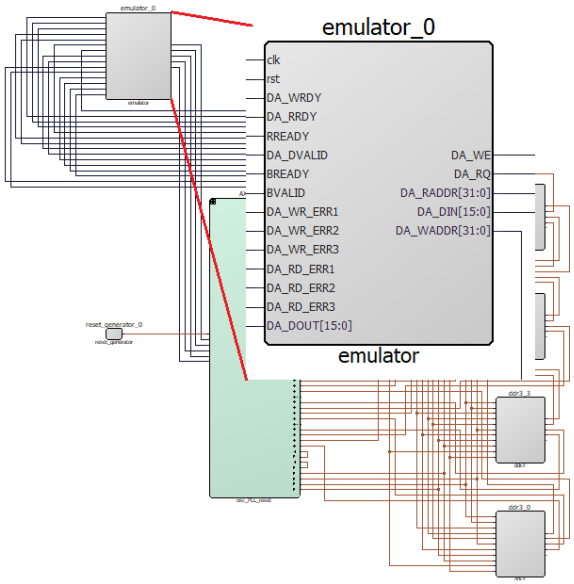

Figure 7.23: Zoom of the emulator block

In the below figure, there is a zoom of one of the memory banks. The total number of memory banks used for storing data on the RT4G150 is 8, 4 on the east side and 4 on the west side. In this project, only the 4 on the west side are used, since, when the FDDR has been configured, the west FDDR has been selected. Each bank has a dimension of 256 MBytes and the specific type of used memory is the 256 MBytes Micron DDR3 memories **MT41K256M8DA-125IT:K**. The data in these memory banks are not written or read in words, but in bytes, thus, for example, if a data of 32 bits, which are 4 bytes, is written in the memory banks, it is not inserted entirely in a unique address, but it is divided into 4 different addresses, an address for byte. It is the FDDR which manages this division in bytes, based on the address sent by the user. The FDDR distributes the bytes in order to obtain good performances and to consume less power.
The bidirectional pin of the bank in this figure, through which the data transit when these are read or written, is the **dq[7:0]** pin, which has a length of 8 since the data are divided in bytes as said before. While, the **ba[2:0]** is the bank address and defines the bank to which an *ACTIVATE*, *READ*, *WRITE*, or *PRECHARGE* command has been applied. The **ck** and **ck_n** are differential clock inputs. All control and address input signals are sampled on crossing if the positive edge of ck and the negative edge of ck_n. The **cke** signal is the clock enable, which enables and disables internal circuitry and clocks on the DRAM in order to save power. Then, **cs_n** is the chip select, which enables and disable the command detector, so, all commands are masked when this signal is high. While, the **ras_n**, **cas_n** and **we_n** are command inputs which define the command being entered.
The **rst_n** is an active low CMOS input and the **odt** signal is a on-die termination, which enables and disables termination resistance internal to the DDR3 SDRAM.
The **addr** is the address input and it provides the row address for ACTIVE commands, the column address and the auto precharge bit, which is addr[10] for reading or writing commands.
The **w_tdqs** is the input mask signal for the write data. Thus, the input data is masked when this signal is sampled high along with the input data during a write access.
While, the **dqs** or **dqs_n** are data strobe, which goes out with the read data and the edge is aligned with the read data, and it enters with write data and the edge is centred

to write data.[3]

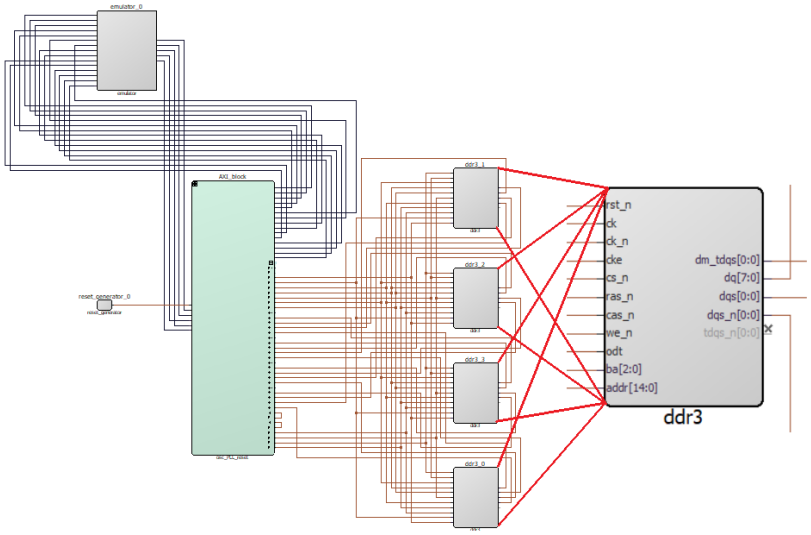

Figure 7.24: Zoom of the DDR3 memory bank

While, in Figure 7.25, the zoom of the reset generator is reported. This structure is simply implemented by means of a process in VHDL code, in which the reset, which is active low and asynchronous, is asserted for 50 ns and then, it is asserted again for 50 ns after 1200 ns. It is used to check that when the reset arrives, asynchronously, the blocks reset correctly coming back to the IDLE state.
At this point, the structure to simulate if the Direct Access AXI master is able to write to or read from the memory banks correctly is completed.

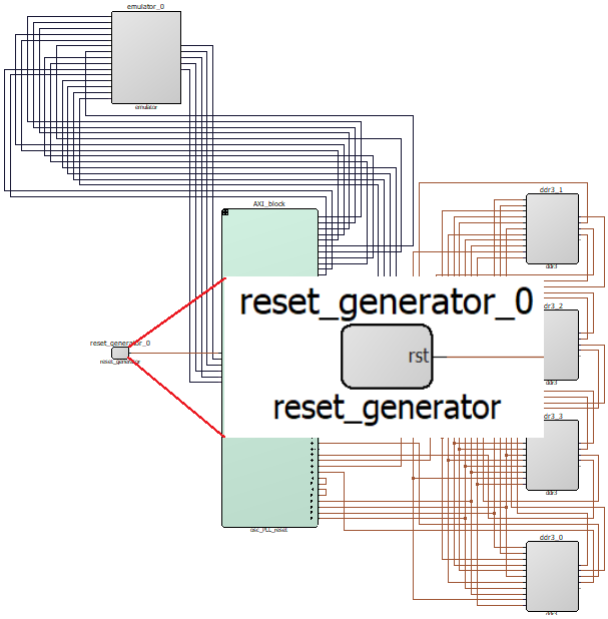

Figure 7.25: Zoom of the reset generator

In Section 8.1, the simulation of the correct behaviour of the entire Direct Access block, concerning the writing and reading parts, is reported, in order to check the correctness of the implementation.

7.2 FIFO block

The second interface of the high-speed data buffer controller, available for other blocks, is the FIFO interface, with which the user wants to write to or read from the DDR3 memory banks a burst of 8 data.
Even in this scenario, as in the direct access case, the FIFO block aim is to translate the signals sent from the user through the FIFO interface into signals compatible with the AXI interface and vice-versa.
Internally, this block is composed of two different blocks, one is the Write FIFO block, which has the aim to manage the writing of the data sent from the user to the DDR3 memory banks, converting correctly the FIFO signals, reported at the **ARS-REQ-440** requirement in Subsection 6.2.5, to the AXI signals reported at **ARS-REQ-470** requirement in the same subsection; while, the other is the Read FIFO block, which has to administrate the data reading from the memory, converting, as before, the FIFO signals, reported at the **ARS-REQ-480** requirement in Subsection 6.2.6, to AXI signals, reported in the same subsection at the **ARS-REQ-510** requirement.
The behaviour of these two sub-blocks is better explained in the below subsections.

7.2.1 Write FIFO block

The Write FIFO block, as in the previous cases, has been implemented through a Finite State Machine, without a data path, which is emulated in the FSM, in order to make the implementation on FPGA less complex.
The scheme of the write FIFO FSM is reported in Figure 7.26. This implements the writing of a burst of 8 data in memory, translating the write FIFO signals to the write AXI signals.

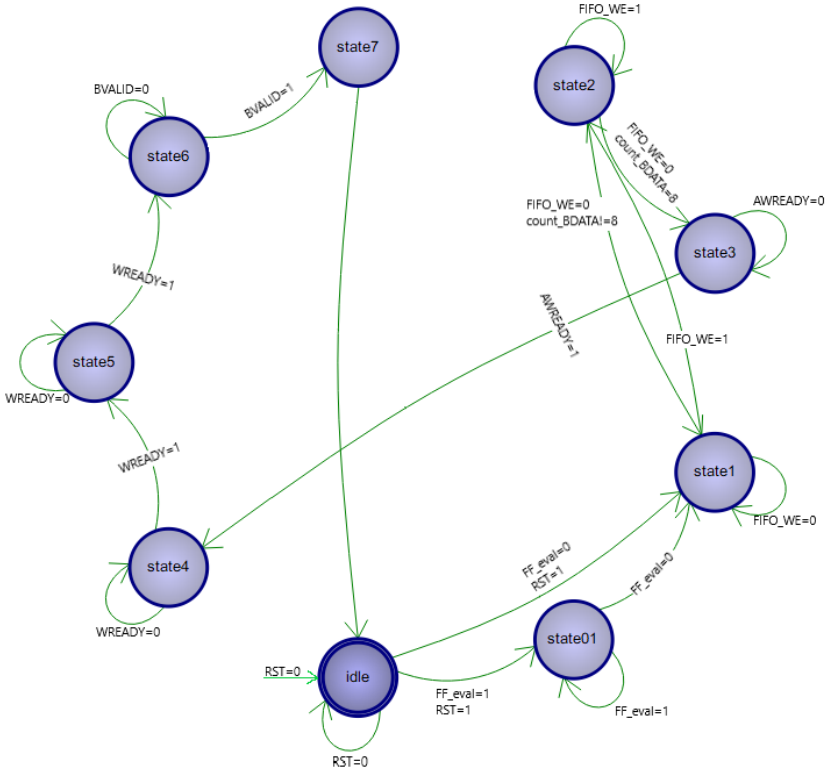

Figure 7.26: Finite State Machine of Write FIFO block

The machine starts from the *IDLE* state, in which the signals are initialized and in which the system goes every time the **reset** is asserted, thus, when it goes to 0 since the reset is active low.
In the case of FIFO, there is one more check that has to be done in writing and it is if the FIFO is full or not when a user wants to write. In reality, what is full is not the FIFO, but is the part of DDR3 memory banks dedicated to the FIFO interface, but the user from outside thinks to write in a real FIFO.
Thus, if the FIFO is full, the **FF_eval** is asserted (FF_eval=1) and the user cannot start the writing. Therefore, if the RST signal is de-asserted and the FIFO is full, the machine moves from *IDLE* to *STATE01* and in this state, the **FIFO_WRDY** and the **FIFO_FF** signals are asserted, since the FSM could accept a writing since there is no other writing in progress, but the memory part dedicated to the FIFO is full, so, the user could not write anything in memory.
Therefore, when the FSM is in *STATE01*, if the user asserts the **FIFO_WE** signals, which is its writing request, the **FIFO_WR_ERR1** is asserted, in order to communicate to the user that something wrong occurs and the writing is discarded. It is implemented to satisfy the **ARS-REQ-465** requirement, which says that the access to FIFO write port has to be discarded if FIFO_FF is asserted and FIFO_WR_ERR1 error flag is raised.
Thus, the FSM remains in this state till FF_eval=1. While, when this signal is de-asserted, the system could go from the *STATE01* to *STATE1*, where the writing could start.
In *STATE1*, it is possible to arrive also from *IDLE* state in the case in which FF_eval=0 and RST=1.
In this state, the FIFO_FF is de-asserted and the FIFO_WRDY remains high, in the case in which the original state is *STATE01*, while if the original state is *IDLE*, the FIFO_WRDY signal is asserted. At this point, if the user raises the FIFO_WE, the writing starts and the machine moves to *STATE2*, while, till FIFO_WE=0, the FSM remains in *STATE1*.
So, what has to be taken into account, in this case, is that the burst that could be written in memory has to have a length equal to 8 and that burst with a length higher or lower than 8 has to be discarded. Thus, in *STATE2* there is a signal, **count_BDATA**, which has the task to count how much data there are in the burst, which the user would like to write. The count_BDATA counts the number of data in the burst counting the number of clock cycles in which the FIFO_WE signal is asserted. At the moment in which the FIFO_WE is de-asserted, if count_BDATA is equal to 8, the FSM could go in *STATE3*, while, if count_BDATA is lower or higher than 8, the writing could not proceed and it is discarded and the machine comes back to *STATE1*. In this last case, the FIFO_WR_ERR2 is asserted in order to satisfy the **ARS-REQ-455** requirement.
Moreover, in *STATE2*, the data are inserted sequentially in an array with a depth of 8 and the length of each location equal to 16 bits. This array is used in the Finite State Machine in order to simulate a shift register, which would have to be there if a data path had also been implemented.
At this point, if all has gone well till now, the machine is in *STATE3* in which the **FIFO_WADDR**, that is not sent by the user, but it is generated internally by the write FIFO AXI master, is assigned to the **AWADDR**, which is the address of the AXI interface, which the master sends to the slave. Furthermore, in this state, the **AWVALID** signal is asserted in order to communicate to the slave that the address at that moment is valid.

The system remains in this state until the slave asserts the **AWREADY** signal, which is the acknowledgment that the address has been received correctly.
From this state till the end, there is a signal, **FIFO_WR_ERR3**, which is raised in the case in which the user tries to write while already another writing is in progress, so, if the FIFO_WE signal is asserted when FIFO_WRDY=0. It is implemented in order to satisfy the **ARS-REQ-450**, which says that if FIFO_WRDY is de-asserted, the access to this port has to be discarded and FIFO_WR_ERR3 error flag raised.
When the slave asserts AWREADY signal, the FSM moves in *STATE4*, where the first 4 data each of 16 bits, which are inside the array, are concatenated to create a unique data to send through the **WDATA**, which has a length of 64 bits. Always in this state, the **WVALID** signal is asserted in order to communicate to the slave that at that moment the data receives through the WDATA is a valid data and that could accept it. The FSM remains in this state till the slave asserts the **WREADY** signal as acknowledgment that the data has been received in a correct way.
At this point, the system goes to *STATE5*, where the remaining 4 data in the array are concatenated and sent as second data through the WDATA signal of the AXI interface. Even in this case, the WVALID signal is asserted, since in that the data is valid.
Moreover, in this state, also **WLAST** and **BREADY** are asserted. WLAST is raised since this is the last data which is sent from master to slave and BREADY is asserted in order to complete the write response in one clock cycle.
The reason why the 8 data, each one with a length of 16 bits, have been compacted to 2 data is that in this way the Finite State Machine is faster and the consumption is lower since from the AXI master to the AXI slave are sent only 2 data instead of 8. This could be done, since then the data in memory are divided into bytes.
Thus, when the FSM receives the WREADY=1, it moves from *STATE5* to *STATE6*, where the WVALID and WLAST are de-asserted since the data have been already sent and BREADY remains asserted, waiting for the **BVALID** signal. When the slave sends to the master this signal asserted, it means that the writing is terminated correctly.
At this point the FSM goes to the last state, *STATE7*, where the signal **write_en** is asserted for one clock cycle and the address of the FIFO (FIFO_WADDR) is updated for the next writing. The write_en signal is used to understand when the FIFO is full. This signal is sent to a block, called **Evaluator**, in which this signal and one, which comes from the Read FIFO block, arrive. The evaluator has the aim to count the writing and the reading in order to understand when the FIFO is full or empty, but the behaviour of this block will be better explained following.
While, about the address update, it is necessary since the FIFO is written sequentially and the address of the first data of the burst is generated by the write FIFO AXI block and not by the user itself. So, for every writing, this value has to be updated and this is incremented every time of 16 since in the memory there is a division in bytes of the data, and since the data are 8, each with a length of 16 bits (2 bytes), in total the number of bytes is 16 and so to memorize in the DDR3 memory block the burst of 8 data, 16 bytes addresses are used.
In the end, the Finite State Machine comes back to *IDLE* state where the writing procedure could restart.
In Figure 7.27, there is represented the inputs and outputs of the write block and also the signal which goes to the evaluator block.

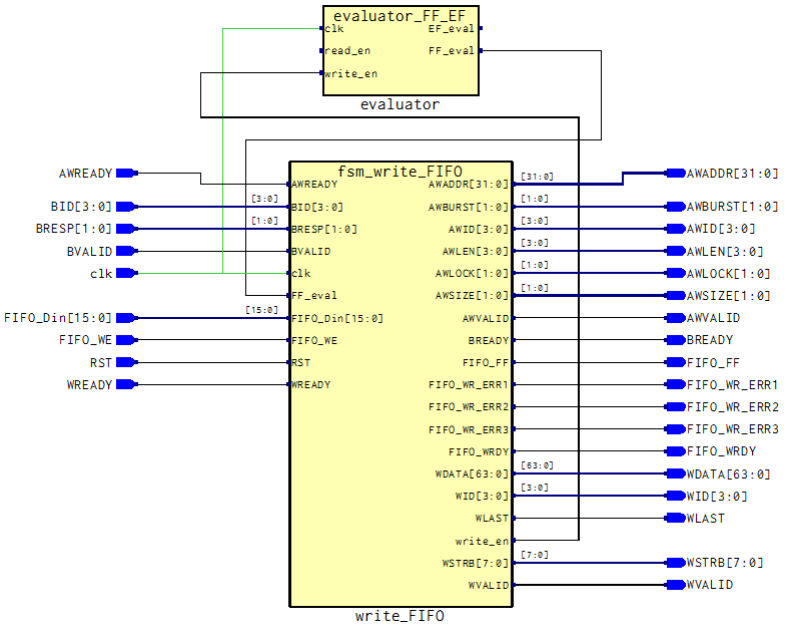

Figure 7.27: FIFO write block

In particular, as before, the clock and reset are generated respectively by a combo of a 50 MHz oscillator plus a PLL and a reset generator, while the FIFO_DIN and FIFO_WE are sent by the user to the write FIFO AXI master, the AWREADY, BID, BRESP, BVALID and WREADY signals are sent from the AXI slave to the AXI master as stimuli for the FSM inside the master, while, the AWADDR, AWBURST, AWID, AWLEN, AWLOCK, AWSIZE, AWVALID, BREADY, WDATA, WID, WLAST, WSTRB and WVALID are sent from the AXI master to the AXI slave (FDDR) as stimuli for the FSM inside the slave. The FIFO_FF, FIFO_WR_ERR1, FIFO_WR_ERR2, FIFO_WR_ERR3 and FIFO_WRDY are sent from the AXI master to the user and lastly, the write_en is sent from the master to the evaluator and vice-versa the FF_eval is sent from the evaluator to the AXI master.
Even here, there are some signals, which the master forwards to the slave, that are maintained as constant, to configure the FDDR controller.
These signals are:

- AWID[3:0]=0000, which indicates the identification TAG for the write address group of signals;
- AWLEN[3:0]=0001, which indicates the burst length, thus, the exact number of transfers in a burst. Since in this case, the number of data transferred in a burst is two from the AXI master to the AXI slave, this value is set to "0001", which corresponds to 2 for the FDDR, as it is possible to observe from the indications in Table 6.2. The reason why this value corresponds to 2 and not 8 is that the 8 data which are sent from the user to the AXI master, inside that are compacted in 2 data;
- AWSIZE[1:0]=11, which indicates the maximum number of data bytes to transfer in each data transfer. In this case, the attributed value is "11", which corresponds to 8 bytes for the FDDR. Its value is set to 8, since, the 8 data, each with a length of 16 bits, sent by the user, have been compacted in 2 data, each with a length of 64 bit, as the size of WDATA signal. So, 64 bits are 8 bytes and this is why this value is set to "11";

- AWBURST[1:0]=01, which indicates the burst type. In this case, the set burst type corresponds to the **INCR** type, with which the address of each data sent in memory is determined by increasing sequentially the address sent by the user;
- AWLOCK[1:0]=00, which indicates the lock type, giving additional information about the atomic characteristics of the write transfer. In this case, "00" corresponds to a **normal access**;
- WID[3:0]=0000, which indicates the response ID, so, the identification TAG of the write response;
- WSTRB[7:0]=11111111, which indicates which byte lanes to update in memory, based on the number of bytes of which is composed the write data. In this case, the write data has a length of 64 bits, which corresponds to 8 bytes and so, all the lanes have to be used.

7.2.2 Read FIFO block

Even this Read FIFO block has been implemented only through a Finite State Machine, which scheme is reported in Figure 7.28. This has the task to translate the read FIFO signals, coming from the FIFO interface to the read AXI signals, which belong to the AXI interfaces.
As before, this FSM starts from the *IDLE* state in which the machine goes if **RST** is asserted (RST=0). Then, when the reset is de-asserted, the system could go or in the *STATE01*, in the case in which the FIFO is empty and so when **EF_eval** is asserted, or in *STATE1* if FIFO is not empty, since there are data that could be read, and it happens if EF_eval is de-asserted.
So, if the machine goes to *STATE01*, this means that there are no data that could be read in memory and the **FIFO_EF** signal is asserted. Even the **FIFO_RRDY** is asserted in this state since, in theory, the FSM is not busy to execute another reading, but the problem, in this case, is that there are no data to read in memory and so the reading cannot start. Thus, in this case, the reading is discarded and the **FIFO_RD_ERR1** signal is asserted in order to signalized to the user that the memory is empty. It is implemented in order to respect the **ARS-REQ-505** requirement, that says that the access to FIFO read port has to be discarded if FIFO_EF is asserted and FIFO_RD_ERR1 error flag raised. The FSM could move from *STATE01* to *SATE1* only if EF_eval is de-asserted.
While, if the FIFO is not empty and the machine moves in *STATE1*, here only the FIFO_RRDY is asserted, so, if the user requires to start a reading, asserting the FIFO_RQ, this could start without problem. Thus, the FSM goes to *STATE2*, where there is a signal, **count_FIFO_RQ**, that counts how many clock cycles the FIFO_RQ remains raised, since, for the **ARS-REQ-485**, this has to be raised only for one clock cycle. If this condition is not respected, the FSM comes back to *STATE1* and the **FIFO_RD_ERR2** is raised to implement the **ARS-REQ-495** requirement, which says that if the access to FIFO port with FIFO_RQ high more than 1 clock cycle has to be discarded and FIFO_RD_ERR2 error flag is raised.
While, if the count_FIFO_RQ is equal to 1 and FIFO_RQ is de-asserted, it is correct, and the system could go in *STATE3*, where the **FIFO_RADDR** signal, always generated inside the FSM of the AXI master since it is not information gives by the user, is assigned to the **ARADDR**, which is the read address that the AXI master sends to the FDDR. Here, it is also asserted the **ARVALID**, since it is the state in which the address is valid and the slave has to understand when it has to accept the correct address. The

machine remains there until the **ARREADY** signal is asserted by the slave as acknowledgment that the address has been received correctly.
At this point, the system goes to *STATE4*, where **RREADY** is asserted since the AXI master is ready to accept the data. So, when the AXI slave drives the valid read data, it sends the **RVALID** stimulus (RVALID=1) and the FSM could go from *STATE4* to *STATE5*.
From the memory banks, two data will be read, since when the data have been written during a FIFO writing, these, from 8 data of the burst, each with a length of 16 bits, have been compacted in only two data, each with a length of 64 bits. In *STATE1*, the first read data has been transferred from the slave to the master through the **RDATA** signal. It is assigned to the second half of a support signal, called **RDATA_sig**, which has a length of 128 bits, twice as much as the length of RDATA. In this state, the RREADY continue to remain asserted.
Therefore, if the next and last data does not arrive immediately at the next clock cycle with respect to the moment in which the first data arrived, the RVALID and **RLAST** signals are de-asserted and the FSM moves from *STATE5* to *STATE8*, in which the RREADY signal is de-asserted.

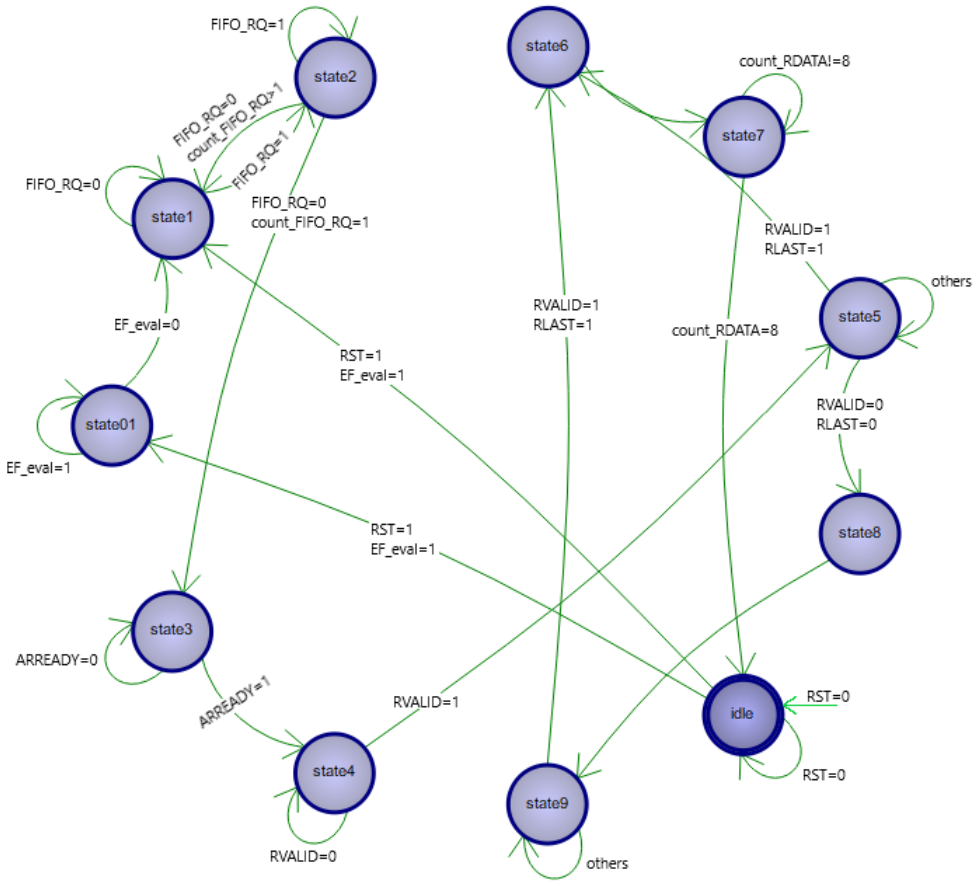

Figure 7.28: Finite State Machine of Read FIFO block

Then, the machine goes to *STATE9* in which the RREADY signal is asserted again to communicate to the slave that it is ready again to receive a data. At this point, when both RVALID and RLAST are asserted, it means that the second and last data is sent

from the slave to the master and the system goes in *STATE6* in order to take the last data.
While, if the second data arrives immediately after the first one, the FSM goes from *STATE5* to *STATE6*, since both RVALID and RLAST are asserted.
In *STATE6*, the second data, arriving through RDATA, is assigned to the first part of the RDATA_sig supported signal. These data are located in the supported signal in this way to obtain the correct order when the data will be divided into the 8 data of the burst. This is because the FDDR controller, when receiving the data from the master, divides these into bytes in order to insert these into memory, but without respecting the order with which the data bytes arrive.
Thus, assigning the two data to the supported signal in this way, the bytes are reordered in the correct way.
Moreover, in *STATE6*, the RREADY is de-asserted, the address of the FIFO is updated for the next reading since the address is not provided by the user, but it is created by the AXI master, and the **read_en** signal is asserted, in order to communicate to the evaluator that a FIFO reading occurs and so, the elements present in the FIFO have to be decremented. The behaviour of this evaluator will be explained better subsequently.
So, the FSM goes to *STATE7*, which is the last state and one of the most important since, here, from the RDATA_sig supported signal, the 8 data of the burst are extrapolated, since what the user expects to receive from the memory is a burst of 8 data, as reported in the **ARS-REQ-485** requirement. The data go out sequentially through the **FIFO_DOUT** signal. During this procedure, the **FIFO_DVALID** signal is asserted to communicate to the user that the data sent in that moment are valid.
In Figure 7.29, the inputs and the outputs of the read block are shown.

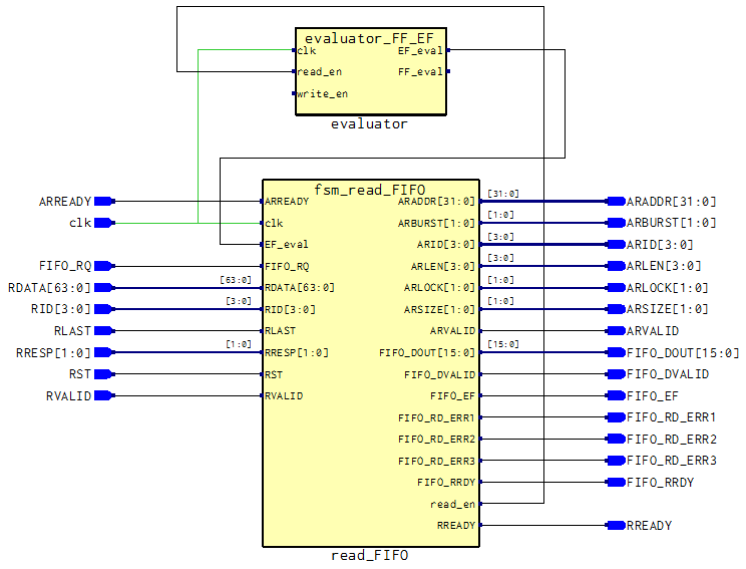

Figure 7.29: FIFO read block

Even in this case, the clock is generated by the union of a 50 MHz oscillator and PLL, and the reset is implemented by a reset generator.
The FIFO_rq, is sent from the user to the slave, while the FIFO_DOUT, FIFO_DVALID, FIFO_EF, FIFO_RD_ERR1, FIFO_RD_ERR2, FIFO_RD_ERR3 and FIFO_RRDY vice-versa, these are sent from the master to the user.
The ARADDR, ARBURST, ARID, ARLEN, ARLOCK, ARSIZE, ARVALID and RREADY

are sent from the AXI master to the AXI slave, so, from the high-speed data buffer to the FDDR, while, vice-versa the ARREADY, RDATA, RID, RLAST, RRESP and RVALID signals.
Moreover, the read_en signal goes from the read block to the evaluator and vice-versa for the EF_eval signal, which indicates if the FIFO is empty or not.
As in the previous cases, there are some signals which remain constant in order to configure in the correct way the FDDR and these are:

- ARID[3:0]=0000, which indicates identification TAG for the read address group of signals;
- ARLEN[3:0]=0001, which indicates the burst length, that gives the exact number of transfers in a burst. In this case, "0001" corresponds to 2 for the FDDR, in order to communicate to it that the read data which has to be read in memory is only two, that will be divided into 8 data, each one of 16 bits ;
- ARSIZE[1:0]=11, which indicates the maximum number of data bytes to transfer in each data transfer. "11" corresponds to the value 8 for the FDDR, since the length of the data, which is read from the memory, is of 64 bits, that correspond to 8 bytes;
- ARBURST[1:0]=01, which indicates the burst type. In this case, "01" corresponds to the **INCR** option, which means that the addresses of the data are determined starting from the address sent from the user and increasing sequentially;
- ARLOCK[1:0]=00, which indicates the lock type, providing additional information about the atomic characteristics of the read transfer. In this case, "00" means a **normal access**.

7.2.3 Write and Read FIFO blocks assembling with evaluator

After the separate implementation of the read and write blocks and their correct behaviours have been checked separately by means of simulation on ModelSim, which are reported in the following chapter, these have been reunited in an unique AXI master block, as reported in Figure 7.30, where there is also the evaluator block, called **evaluator_FF_EF**.
This evaluator is a sequential block which has the aim to control the FIFO capacity, based on how many writing and reading are done. The function of this block could be seen as a buffer with a *head* and a *tail* cursors, in which if the head and the tail point to the same element, it means that the FIFO is **empty** and so the EF_eval is asserted, to communicate to the read FIFO block, that the FIFO is empty, so, there are not elements that could be read and because of that, the reading could not start.
Every time that a writing is executed, the signal **write_en** is asserted and the head is incremented of the number of bytes which composed a burst, in this case 16 bytes, since the burst is composed of 8 data, each one with a length of 16 bits.
While, every time a reading is executed, the **read_en** is asserted and the tail is incremented of 16, since even in this case the bytes, which are read, are 16.
In the case in which the head arrives in the position before the tail, it means that the FIFO is **full** and so the FF_eval is asserted and sent to the write FIFO block, which understands that the FIFO is full and that a writing could not start since there are not empty location where to place the data.
In the other cases, the EF_eval and FF_eval are de-asserted and both writing and reading could be executed.

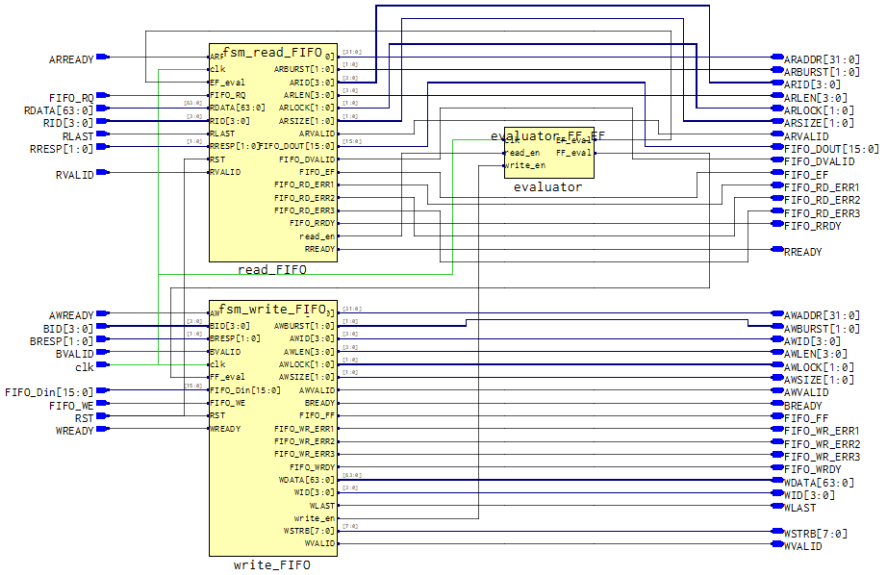

Figure 7.30: Write and Read FIFO AXI block with evaluator

At this point, as in the Direct Access case, it has been created the structure with LiberoSoC, composed of the AXI master, that in this case is composed of the write and read FIFO blocks and the evaluator, the FDDR block (AXI slave), the DDR3 memory banks and the blocks to generate the clocks and reset signals.
The only blocks which are different with respect to the structure explained in Subsection 7.1.3 are the AXI master, in which only the FIFO part of the high speed data buffer is implemented, the emulator of the user, which sends signal to a FIFO interface, and the AXI block. The other blocks remain with the same setting given for the Direct Access case.
In Figure 7.31, there is reported the zoom of the FIFO AXI master, which has some inputs and outputs which are different with respect to the Direct Access block, since in this case has been implemented a FIFO interface inside the master, that converts the FIFO interface's signals in AXI interface's signals.

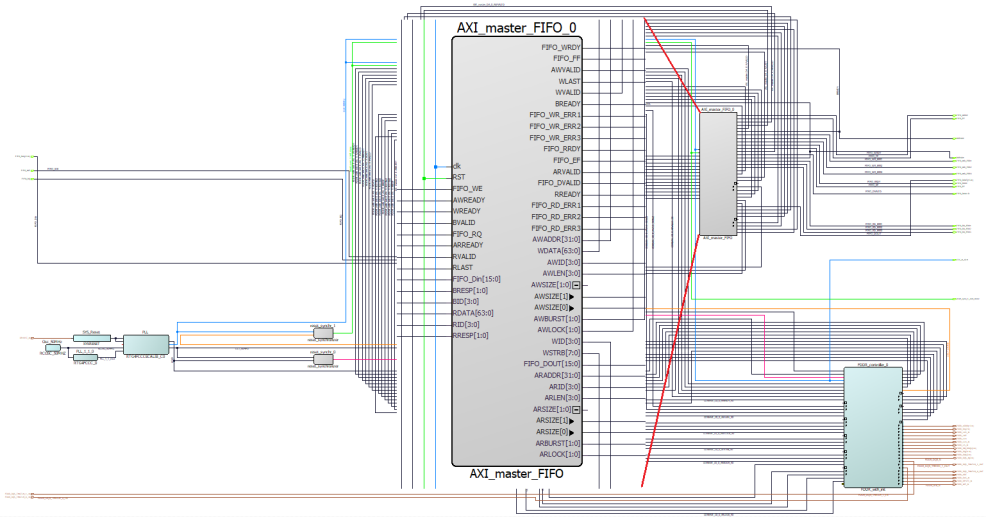

Figure 7.31: Zoom of the FIFO AXI master block

While, in Figure 7.32, there are the zooms of the emulator block and the AXI block. The AXI block has some signals different since inside there is the FIFO AXI master block, which has a different function with respect to the case of the Direct Access.
Moreover, the emulator has been changed since, in this case, it has to emulate a user that want to write to or read from a FIFO, not a user which want to use the Direct Access interface.

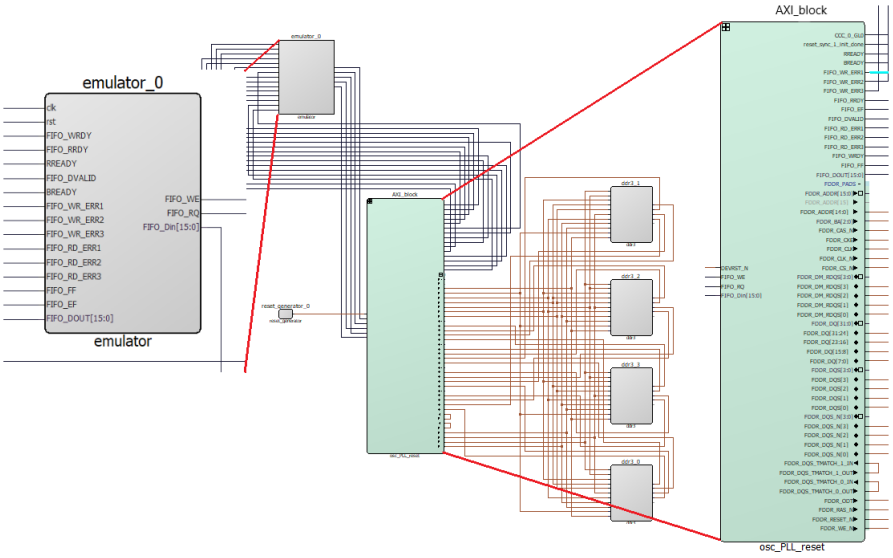

Figure 7.32: Zoom of the emulator and AXI block

In Section 8.2, the FIFO block simulation is reported, both for the writing and reading, in order to check its behaviour and the correctness of the implemented design.

7.3 Arbiter block

The arbiter block has an essential task, since by specification, the high speed data buffer has two different interfaces, a Direct Access interface and a FIFO one and both could write and read. There are not problems if an interface writes and the other reads at the same time, but the problem there is if both interfaces want to write to or read from the memory at the same time.
Thus, the arbiter has the aim to coordinate concurrent accesses, as it is indicated in the **ARS-REQ-525** requirement, when different users want to access to the same sources through different interfaces.
The way in which the arbiter coordinate concurrent accesses is using the Round Robin algorithm. When, for example, two users want to write one through the Direct Access and one through the FIFO interfaces in the same sources, with the Round Robin algorithm, the arbiter could choose who has the priority to use the sources and it could put in queue the request of the other user and execute this at the next writing. The way in which the arbiter choose to who gives the priority is based on who previously has execute a writing, so, if in the previous writing the FIFO interface has accessed to the DDR3 memory banks, in the next writing, if both interfaces want to access the memory to write data inside, the priority is given to the Direct Access interface, and vice-versa.
Inside, the arbiter is composed of a block (**write_arbiter**) dedicated to the arbitration of the writing, and another one (**read_arbiter**), which manages the arbitration of the reading.

7.3.1 Write Arbiter block

This block is an arbiter with the aim to coordinate the concurrent writing to the DDR3 memory banks. In Figure 7.33, the internal structure of the writing arbiter is reported. This is composed of a Finite State Machine and a multiplexer, which is rotated on the write FIFO signals or on the read Direct Access signals based on the selector (SEL) signal, which the FSM sends to the multiplexer.

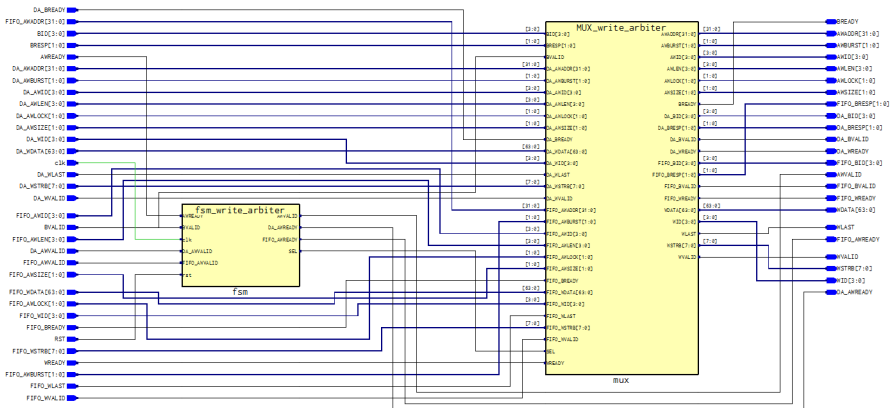

Figure 7.33: Write Arbiter structure

In Figure 7.34, the Finite State Machine of the writing arbiter is reported.
This FSM starts from the *IDLE* state, where it goes when the reset signal is asserted (**RST**=0). In this state, the initialization of the signal is done. When the reset is de-asserted, the arbiter FSM could choose in which state goes based on two complex conditions, which are related to the red wires, which connect the *IDLE* state to *STATE1* or *STATE2*.
In this choice, a signal, called **choice_bit**, has been involved. This changes from 0 to 1 if the priority is given to the Direct Access and from 1 to 0 if the priority is given to the FIFO. If choice_bit is equal to 0, it means that the last access has been done by the direct access, while if it is equal to 1, it means that the last access has been done by the FIFO.
Thus, the cases in which the system goes from *IDLE* to *STATE1* is when the priority is given to the direct access interface and the selector is asserted (SEL=1), in order to rotate correctly the multiplexer. It happens when:

- **choice_bit=0**, **DA_AWVALID=1** and **FIFO_AWVALID=1**; since both the FIFO and the direct access interfaces want to access the memory banks, there is the need to look at the choice_bit, which, being equal to 0 in this case, indicates that in the previous case the priority has been given to the FIFO interface. So, now that both the interfaces want to enter, the priority has to be attributed to the direct access interface. Then, the choice_bit is incremented to 1, to indicates to the next writing, that in the previous writing the priority has been given to the direct access interface;

- **choice_bit=X**, **DA_AWVALID=1** and **FIFO_AWVALID=0**; in this case only the direct access interface want to access in memory, since only DA_AWVALID is asserted to 1, while FIFO_AWVALID remains to 0 and it means that the FIFO interface does not want to access in memory. Thus, it is not important the value of the choice_bit, since in that case there is not concurrency.

While, the case in which the FSM goes from *IDLE* to *STATE2* is when the priority is given to the FIFO interface and the selector is de-asserted (SEL=0), in order to rotate the multiplexer on the FIFO signals. It is implemented when:

- **choice_bit=1**, **DA_AWVALID=1** and **FIFO_AWVALID=1**; since both the FIFO and the direct access interfaces want to access the memory banks, there is the need to look at the choice_bit, which, being equal to 1 in this case, indicates that in the previous case the priority has been given to the Direct Access interface. So, now that both the interfaces want to enter, the priority has to be attributed to the FIFO interface. Then, the choice_bit is decremented to 0, to indicates to the next writing, that in the previous writing the priority has been given to the FIFO interface;
- **choice_bit=X**, **DA_AWVALID=0** and **FIFO_AWVALID=1**; in this case only the FIFO interface want to access in memory, since only FIFO_AWVALID is asserted to 1, while DA_AWVALID remains to 0 and it means that the FIFO interface does not want to access in memory. Thus, it is not important the value of the choice_bit, since even in this case there is not concurrency.

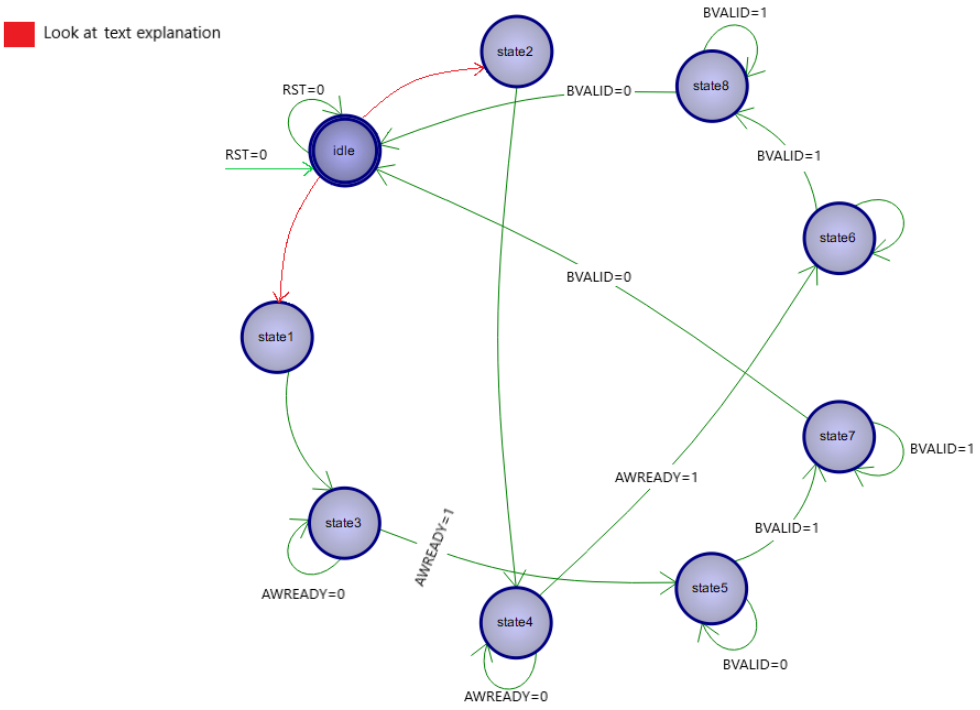

Figure 7.34: Finite State Machine of Write Arbiter

The DA_AWVALID and the FIFO_AWVALID are used to signalize to the arbiter if these interfaces want to access to the memory banks.
If the machine goes from *IDLE* to *STATE1*, it means that the priority has been given to the direct access interface. In this state, the DA_AWVALID of the Direct Access AXI master is assigned to the AWVALID of the AXI slave, and the SEL signal is set to 1, in order to rotate the multiplexer on the direct access signals.
At the next clock cycle, the FSM goes to *STATE3*, which is a delay state, where SEL is always equal to 1. The machine remains in this state till AWREADY is asserted, so, the FSM goes to *STATE5*, where the AWREADY is assigned to the DA_AWREADY signal

and SEL is maintained asserted. The FSM remains in this state until BVALID=1, thus, the machine moves to *STATE7*. This state is used in order to maintain the SEL set to 1 till the writing process ends. So, when the BVALID signal is de-asserted, it indicates that the writing process through the direct access is finished and the system could come back to the IDLE state.
While, if the FSM goes from *IDLE* to *STATE2*, it means that the priority has been given to the FIFO interface. In this state, the FIFO_AWVALID of the FIFO AXI master is assigned to AWVALID of the AXI slave signal, and the SEL is set to 0, in order to rotate correctly the multiplexer on the FIFO signals.
At the next clock cycle, the machine goes from *STATE2* to *STATE4*, which is used as delay state, in which SEL is maintained to 0. Then, when AWREADY is asserted, the machine moves from *STATE4* to *STATE6*, where the AWREADY signal coming from the AXI slave is assigned to the FIFO_AWREADY signal, which is a FIFO AXI master signal.
Thus, when the BVALID signal is asserted, the machine goes to *STATE8*, where SEL is maintained to 0 and where the FSM remain till the end of the writing in order to have the multiplexer rotated to the correct signals. So, when BVALID is de-asserted, the Finite State Machine comes back to the *IDLE* state, where another writing could start. The BVALID is the signal which is used to signalized the end of a writing.

7.3.2 Read Arbiter block

This arbiter has the aim to coordinate the concurrent reading from the DDR3 memory banks. In Figure 7.35, the internal structure of the reading arbiter is reported. It, similarly to the writing arbiter, is composed of a Finite State Machine and a multiplexer, which is rotated on the read FIFO signals or on the read Direct Access signals based in the selector signal, which the FSM sends to the multiplexer.

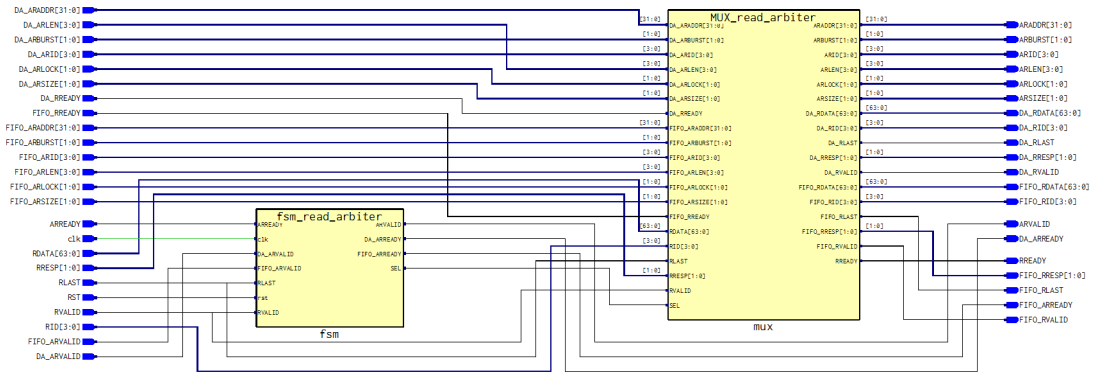

Figure 7.35: Read Arbiter structure

In Figure 7.36, the Finite State Machine of the reading arbiter is reported.
Even this FSM starts from the *IDLE* state, where it goes when the reset signal is asserted. Here, it is where the initialization of the signals is done.
As in the previous case, when the reset is de-asserted, the FSM could choose in which state goes based on two complex conditions, which are related to the red wires in the below figure, that connect the *IDLE* state to *STATE1* or *STATE2*.
Even in this case, the **choice_bit** is used, which function is the same of the choice_bit of the writing arbiter.
Thus, if the FSM goes from *IDLE* to *STATE1* it means that the the priority has been

given to the direct access interface and the selector is asserted, in order to rotate the multiplexer in the right direction. It happens when:

- **choice_bit=0**, **DA_ARVALID=1** and **FIFO_ARVALID=1**; since both the FIFO and the direct access interfaces want to read form the memory banks. Thus, there is the need to look at the choice_bit, which, being equal to 0 in this case, indicates that in the previous case the priority has been given to the FIFO interface. So, now that both the interfaces want to enter, the priority has to be attributed to the direct access interface. Then, the choice_bit is incremented to 1, to indicates to the next reading, that in the previous one the priority has been given to the direct access interface;

- **choice_bit=X**, **DA_ARVALID=1** and **FIFO_ARVALID=0**; in this case only the direct access interface want to access in memory, since only DA_ARVALID is asserted to 1, while FIFO_ARVALID remains to 0 and it means that the FIFO interface does not want to access in memory. Thus, it is not important the value of the choice_bit, since in that case there is not concurrency.

While, the case in which the system moves from *IDLE* to *STATE2* is when the priority is given to the FIFO interface and so, the selector is set to 0, to pass the FIFO signals through the multiplexer. It is implemented when:

- **choice_bit=1**, **DA_AWVALID=1** and **FIFO_AWVALID=1**; since both the FIFO and the direct access interfaces want to access the memory banks, there is the need to look at the choice_bit, which, being equal to 1 in this case, indicates that in the previous case the priority has been given to the Direct Access interface. So, now that both the interfaces want to enter, the priority has to be attributed to the FIFO interface. Then, the choice_bit is decremented to 0, to indicates to the next writing, that in the previous writing the priority has been given to the FIFO interface;

- **choice_bit=X**, **DA_AWVALID=0** and **FIFO_AWVALID=1**; in this case only the FIFO interface want to access in memory, since only FIFO_AWVALID is asserted to 1, while DA_AWVALID remains to 0 and it means that the FIFO interface does not want to access in memory. Thus, it is not important the value of the choice_bit, since even in this case there is not concurrency.

The DA_ARVALID and the FIFO_ARVALID are used to signalized to the read arbiter which of the interfaces want to read from the DDR3 memory banks.
In the case in which the FSM goes from *IDLE* to *STATE1*, it means that the priority is attributed to the direct access interface. Therefore, in this state, the DA_ARVALID signal of the read direct access AXI master is assigned to ARVALID of the AXI slave, and the SEL signal is set to 1, selecting the direct access interface signals. At the next clock cycle the FSM goes to *STATE3*, where the SEL is maintained to 1 and where the machine remains till the *ARREADY* signal is asserted. At that moment, the FSM goes in *STATE5*, in which ARREADY is assigned to DA_ARREADY and SEL=1. Thus, the machine remains in that state till RLAST and RVALID are asserted. Finally, the Finite State Machine goes in *STATE7*, which is used as delay state, in which SEL is maintained to 0, in order to ensure that the multiplexer lets out the FIFO signals. Then, the machine comes back to the *IDLE* state, to restart another reading.
While, if the FSM goes from *IDLE* to *STATE2*, it means that the priority is given to the FIFO interface. In this state, the FIFO_ARVALID, arrived from the FIFO interface, is assigned to the ARVALID, of the AXI slave and the selector is set to 0, in order to rotate

the multiplexer output to the read FIFO signal.
Then, the machine moves o the next state, *STATE4*, which is used as delay state and in which the SEL signal is maintained on 0. The FSM remains in this state till the ARREADY signal is asserted, so the machine goes to *STATE6*, where the ARREADY signal is assigned to FIFO_ARREADY and the selector is set to 0.
Therefore, when the RLAST and RVALID are to 1 value at the same time, the system moves from *STATE6* to *STATE8*, which is used to ensure that the multiplexer could make transit the FIFO signals. At the end, the Finite State Machine comes back to the *IDLE* state in order to could restart another reading.

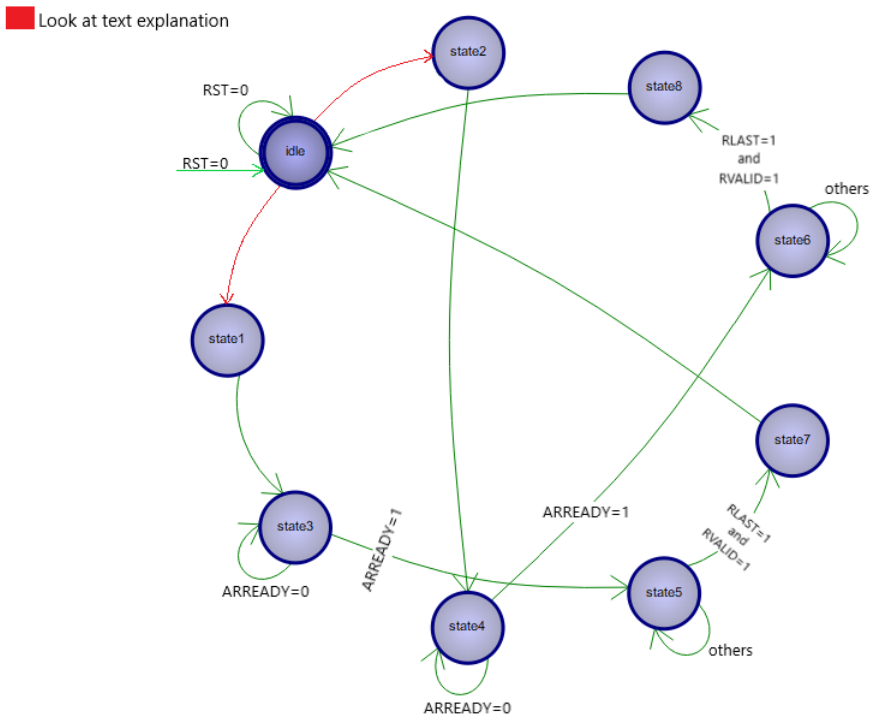

Figure 7.36: Finite State Machine of Read Arbiter

7.3.3 Write and Read Arbiter blocks assembling

The writing and reading arbiters have been put together with the Direct Access AXI master block, which manages the conversion between direct access interface and AXI interface, and the FIFO AXI master block, which translates the signals coming from the FIFO interface to signals of the AXI interface. The reading signals both of the Direct Access block and the FIFO block are connected to the reading arbiter (read_arbiter) and the writing signals belong to the Direct Access and the FIFO blocks are linked to the writing arbiter (write_arbiter), as could be observed in Figure 7.37.

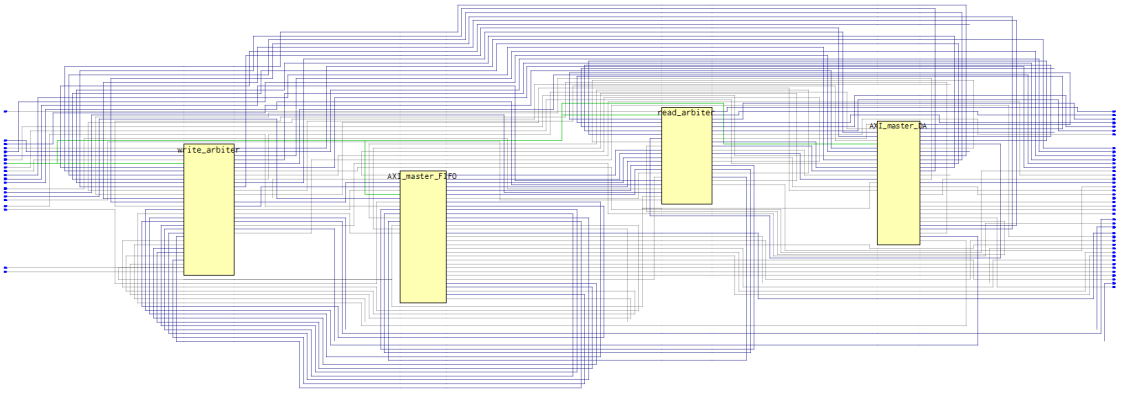

Figure 7.37: Direct Access and FIFO masters with arbiters structure

This complete the final AXI master block implementation, since at this point the high speed data buffer, which represents the AXI master, executes all the requested tasks, in fact, through that it is possible to write to or read from DDR3 memory banks by means of a direct access or a FIFO interfaces, and if there is concurrency in reading or writing there are two dedicated arbiters, which manage the accesses using the Round Robin algorithm. The structure implemented in the RT4G150 FPGA is the similar to the one realized to the Direct Access AXI master, in Subsection 7.1.3, and the FIFO AXI master, in Subsection 7.2.3, when these are implemented separately.
The ways in which the clocks and the reset are generated are the same as before, these structures are reported in Figures 7.7 and 7.25, and also the AXI switch, the AXI slave (FDDR controller), the reset synchronizers and the DDR3 memory banks maintain the set of the previous cases, which could be observed respectively in Figures 7.16, 7.17, 7.12 and 7.24.
The blocks which have been modified are the AXI master block, reported in Figure 7.38, which now contains all what is need, so, the Direct Access block, the FIFO block and the arbiter block.

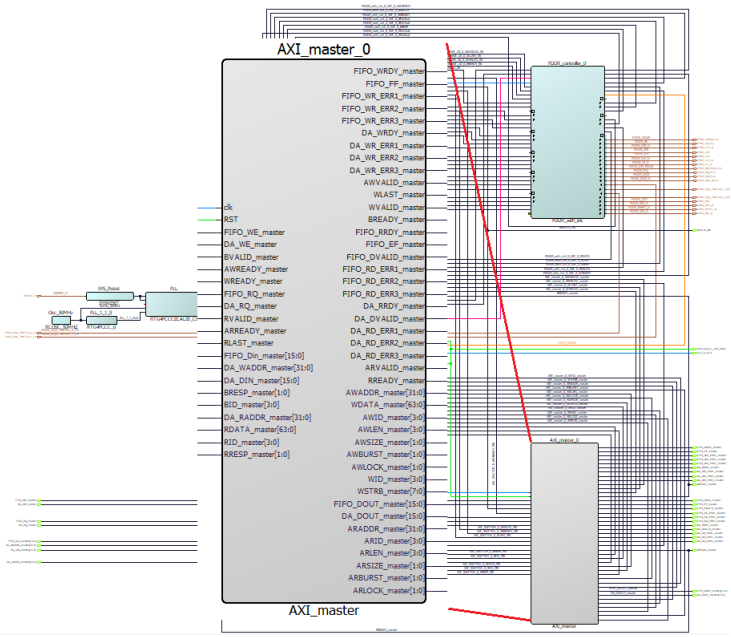

Figure 7.38: Zoom of the AXI master block

From the above figure, it is possible to observe that the AXI master block in this case contains both the signals belong to the Direct Access interface and to the FIFO interface and the signals which are related to the AXI interfaces. These signals concern both the writing and the reading operations.
While, the other two blocks, which need to be changed are the emulator and the AXI block, as it is reported in Figure 7.39.
The emulator in this case has to satisfy different tasks, since it has to represent both a user which want to write a data in memory through the Direct Access interface and a user that want to read a data from memory through the Direct Access interface, and both a user that want to write a burst of 8 data in memory through the FIFO interface and a user that want to read a burst of 8 data from the memory through the FIFO interface.
While, the AXI block is changed since inside it there is the AXI master, which now is different with respect to the previous cases, since now it is the complete version, with all the required blocks inside.

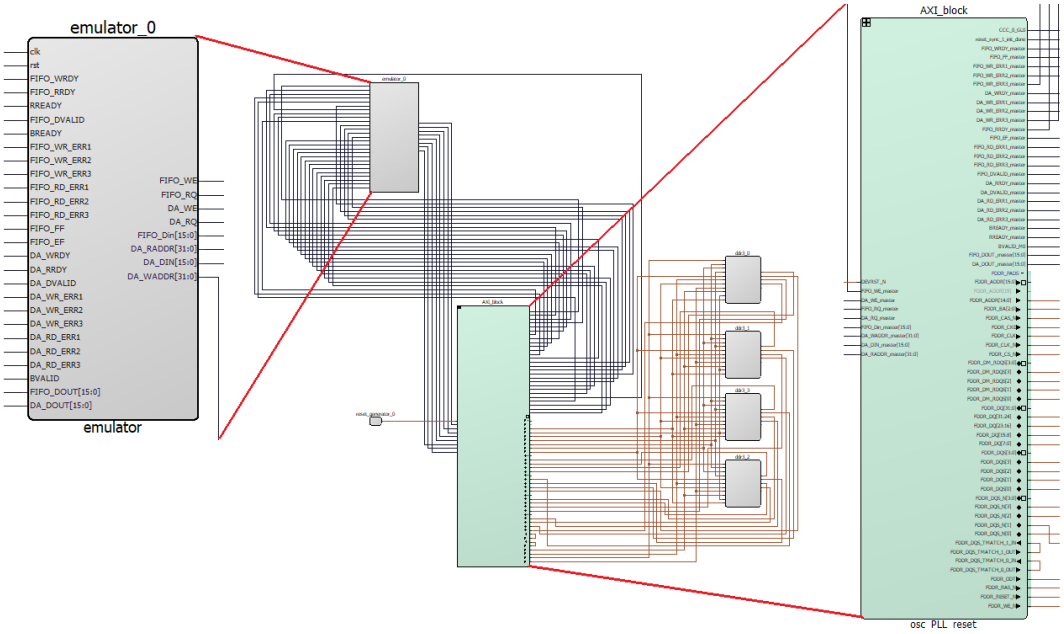

Figure 7.39: Zoom of the emulator and AXI block

It completes the entire structure used to check the right function of the high speed data buffer represented by the AXI master block.
The simulation of this structure is reported in Section 8.3.

Chapter 8

Simulation HDL code

In this chapter, the simulations of the sub-blocks and the complete structure of the high-speed data buffer implemented in the RT4G150 FPGA have been reported, in order to check the correct behaviour of each block.
These simulation are implemented by means of *ModelSim*, through which also a **Code Coverage** for each analyzed sub-block has been executed.
The Code Coverage is a report file feedback on which executable statements, branches, conditions and expressions in the source code have been executed.

8.1 Direct Access block simulation

The first implemented block is the Direct Access one, which translates the signals sent by the user to the Direct Access interface into signals belong to the AXI interface.
What is important at the beginning is to wait that the PLL finds the lock, in order to be sure to have the right frequencies: 50 MHz and 80 MHz. What the signals do before that the PLL find the lock does not have important. In Figure 8.1, there is the assertion of the **LOCK** signal, which indicates that at this moment the PLL is locked. The LOCK signal has also the task to be a sort of initialization reset.

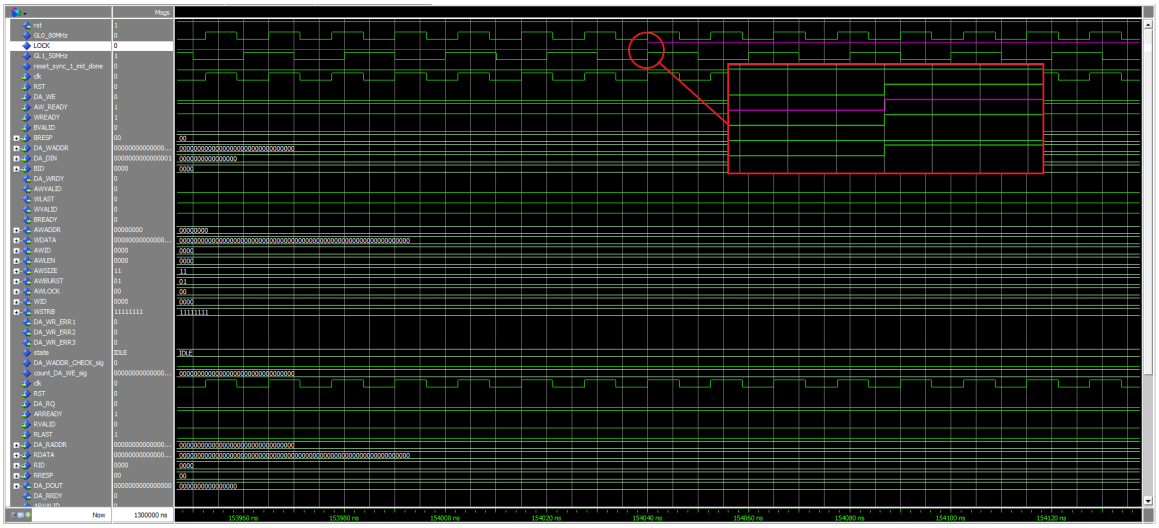

Figure 8.1: Zoom of LOCK signal in Direct Access block simulation

From now on, the signals have validity and the clocks have the right frequency (50

MHz and 80 MHz).
In Figure 8.2, there is the zoom of two important signals, the *INIT_DONE*, generated by the FDDR block, which has the aim to represent the end of the initialization of the structure, and the **reset_sync_1**, which derives by the reset synchronizer 1, to which arrives the 80 MHz clock frequency. This reset_sync_1 signal is the reset of the AXI master block, which in that case is represented by the Write and Read Direct Access block. At this point the AXI master is operative.

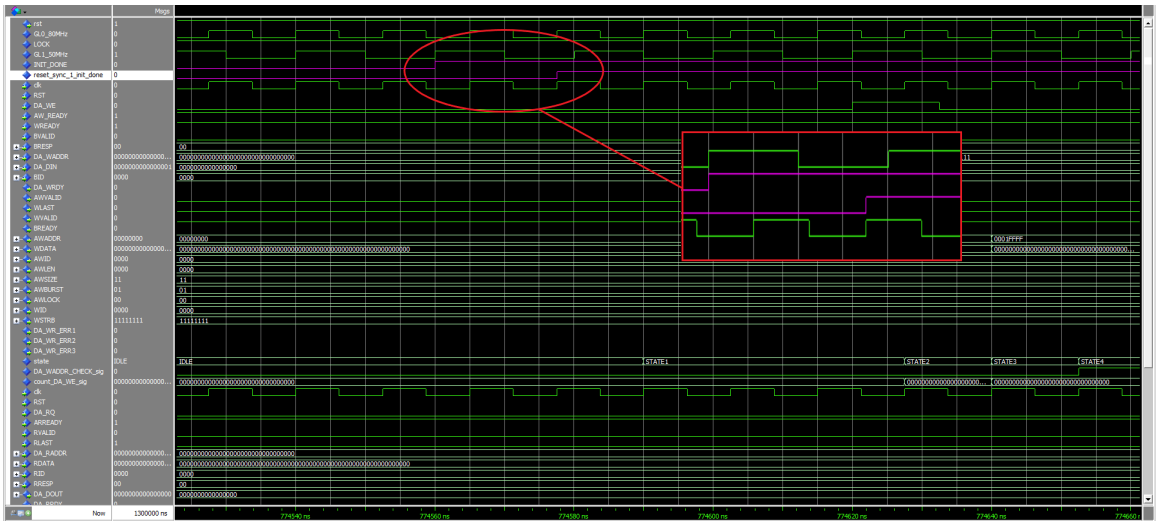

Figure 8.2: Zoom of INIT_DONE and reset_sync_1 signal in Direct Access block simulation

So, the initialization procedure is: first of all the **DEVRST_N** signal, generated by the reset generator is de-asserted after 50 ns, then, the LOCK signal is asserted, when the PLL acquires the lock and subsequently the AXI has to wait the assertion of the INIT_DONE and then of the reset_sync_1 signal before being ready to execute a writing and/or a reading.
At this point, different tests have been executed in order to check that all the requirements are satisfied.
First of all, a writing and subsequently a reading are executed without errors in order to check that effectively what the user write in memory, it has been written correctly, and so that reading in the same address, where the user has written, the read data is the same that has been written.
In Figure 8.3, a writing without error is reported, where there is represented the function of the Finite State Machine of the write Direct Access block, that could be observed in Figure 7.1. In that case, only one data is written in the DDR3 memory block, since it is a writing through a Direct Access interface and thus, the data burst is composed of only one data. In particular, in that figure, the signals belongs to the Direct Access interface are highlighted with a violet color, while the signals of the AXI interface are highlight with a cyan color.
In that part of the simulation, the user want to write the *"0000000000000001"* data, with a length of 16 bits, in the *"00000000000000011111111111111111"* address, which belongs to the part of memory dedicated to the Direct Access, since it does not generate any error, in fact from the figure could be observed that the error flags (DA_WR_ERR1, DA_WR_ERR2 and DA_WR_ERR3) are all de-asserted.

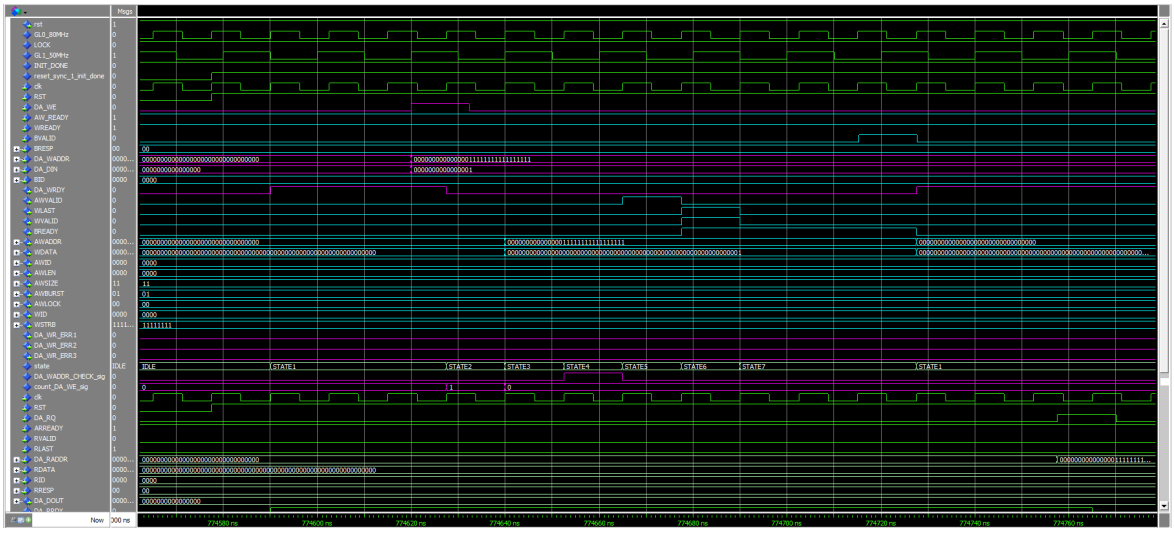

Figure 8.3: Writing Direct Access block simulation

Then, a reading is executed, as it is reported in Figure 8.4, which represents the behaviour of the Finite State Machine of the read Direct Access block, which could be observed in Figure 7.3. Even in this case, the Direct Access interface signals are represented in violet and the signals belong to the AXI interface are highlighted with the cyan color.
From the below figure, it is possible to observe that effectively the data read in the *"00000000000000001111111111111111"* address, which is the one where before the user have written the data, is equal to *"0000000000000001"*, and thus, this is the check that the previous writing has been executed correctly and that also this reading has been successful.

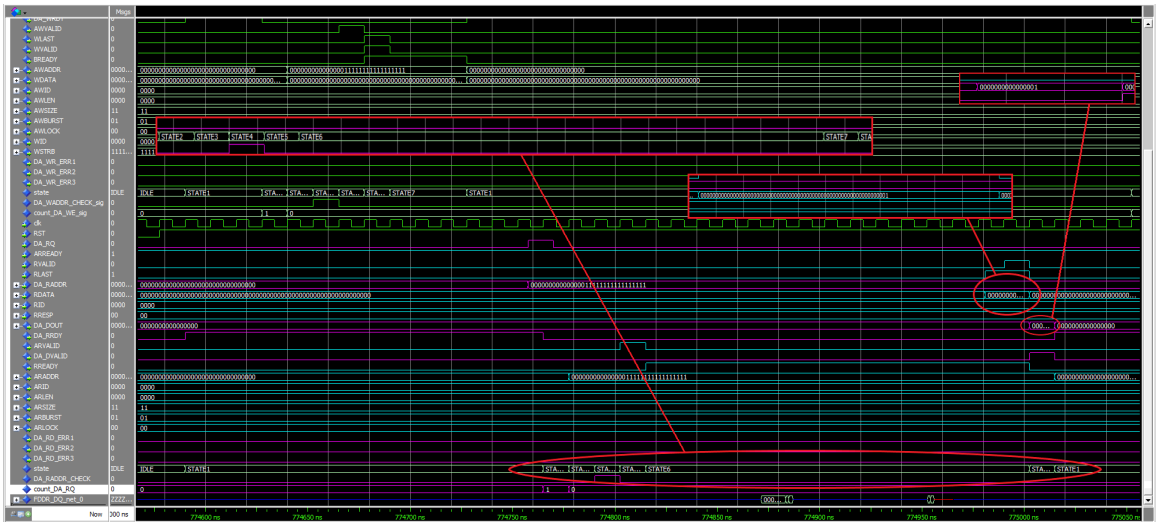

Figure 8.4: Reading Direct Access block simulation

From the above figures, it is possible to see that, in writing, the DA_Din data, which is represented on 16 bits, when it is assigned to the WDATA, it is stretched to 64 bits, in order to adopt it to the WDATA length; and that, in reading, the RDATA data, read from the memory, with a length of 64 bits, has been cut in order to adopt it to the DA_Dout dimension, which is 16 bits.

At this point, another test has been simulated, in order to check that the block is able to recognize when two types of the three errors occur, which are when the write request signal (DA_WE) remains high more than one clock cycle and when the address is not correct, since does not belong to the memory part dedicated to the Direct Access.

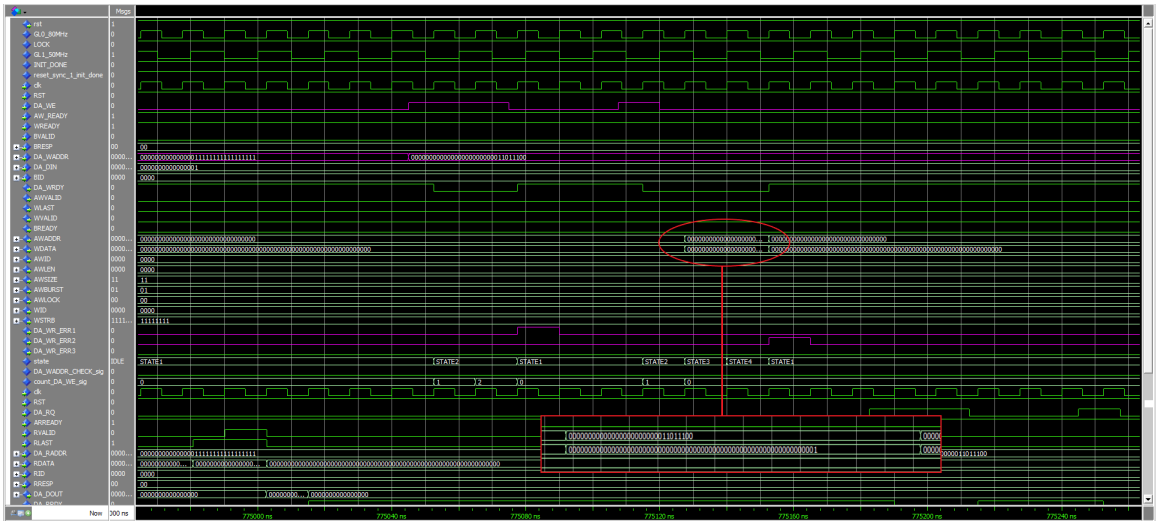

Figure 8.5: Writing Direct Access block simulation with two errors

In fact, from the above figure, it is possible to see that the signal DA_WE is asserted more than one clock cycle, and it is not acceptable, since the **ARS-REQ-305** is not respected and there is the need to signalize to the user that an error occurs and the writing is discarded, as requested by the **ARS-REQ-315** requirement. So, the DA_WR_ERR1 error flag is asserted, in order to communicate to the user which type of error is happened.
Because of that, the writing is discarded and the Finite State Machine comes back to *STATE1* to restart another writing.
Always from the below figure, it could be observed that in the next writing another error occurs, since the address, where the user would like to write, belongs to the part of memory dedicated to the FIFO, and it could not be acceptable as it is reported in **ARS-REQ-325** requirement. In fact, in this case, since the *"00000000000000000000000011011100"* is not part of the Direct Access memory, the DA_WR_ERR2 is raised, to communicate to the user that there is an error related to the address.
Even in this case, the writing request is discarded and the FSM comes back to the *STATE1*.
The same test has been executed for the reading, whose simulation is reported in Figure 8.6.
First of all, there is the case in which the DA_RQ has a duration of more than one clock cycle, and, caused of that, the DA_RD_ERR1 is raised, the reading is discarded and the FSM comes back to *STATE1*. It happens to respect the **ARS-REQ-415** requirement, which says that if the access to the direct access port has a duration greater than one clock cycle, it has to be discarded and the DA_RD_ERR1 has to be asserted.
Moreover, even the check of the invalid address has been executed, in fact, the user requests to read in the same address as before, which is *"00000000000000000000000011011100"*, and it does not belong to the Direct Access memory part, thus, based on the **ARS-REQ-425**, in this case the DA_RD_ERR2 error flag is raised, the reading is discarded and the FSM comes back to *STATE1*

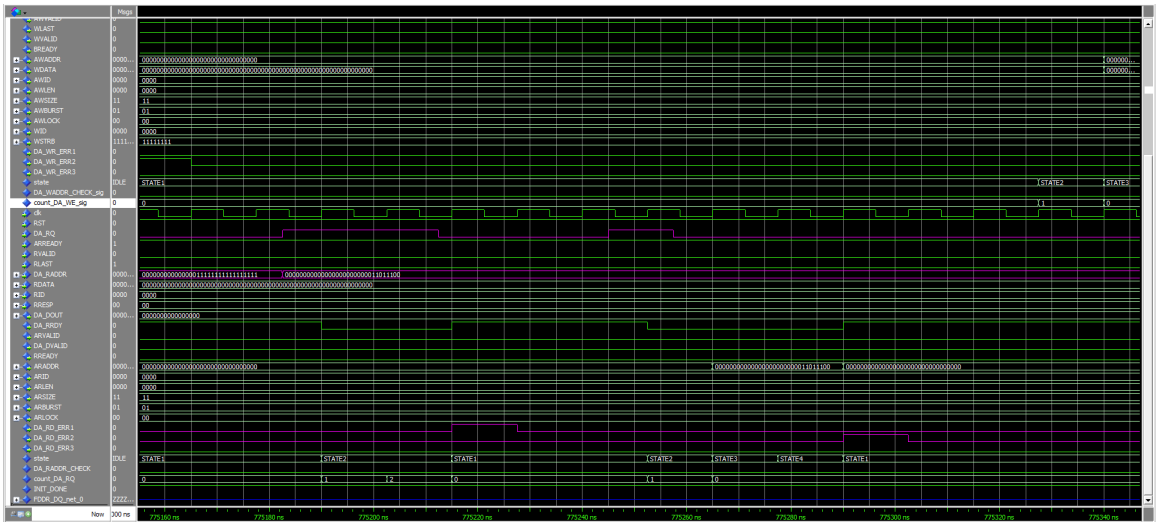

Figure 8.6: Read Direct Access block simulation with two errors

Another test is related to a concurrent access to the Direct Access by means of two users, whose one want to write to the memory and one want to read from the memory a data. Thus, the Direct Access block does not have any problem to execute concurrently a writing and a reading in the same address, since, in this case, it is the FDDR, which will choose to which gives the priority to access this address. While if the concurrent reading and writing in memory has been done to different addresses of the memory, the FDDR shouldn't decide to which one gives the priority.
Thus, to understand the behaviour of the entire structure reported in Figure 7.21, in the case two users want to access to the same address to read and write, in Figure 8.7, 8.8 and 8.9.
In particular, in Figure 8.7, it could be observed that both the DA_WE and DA_RQ are raised for one clock cycle at the same time.

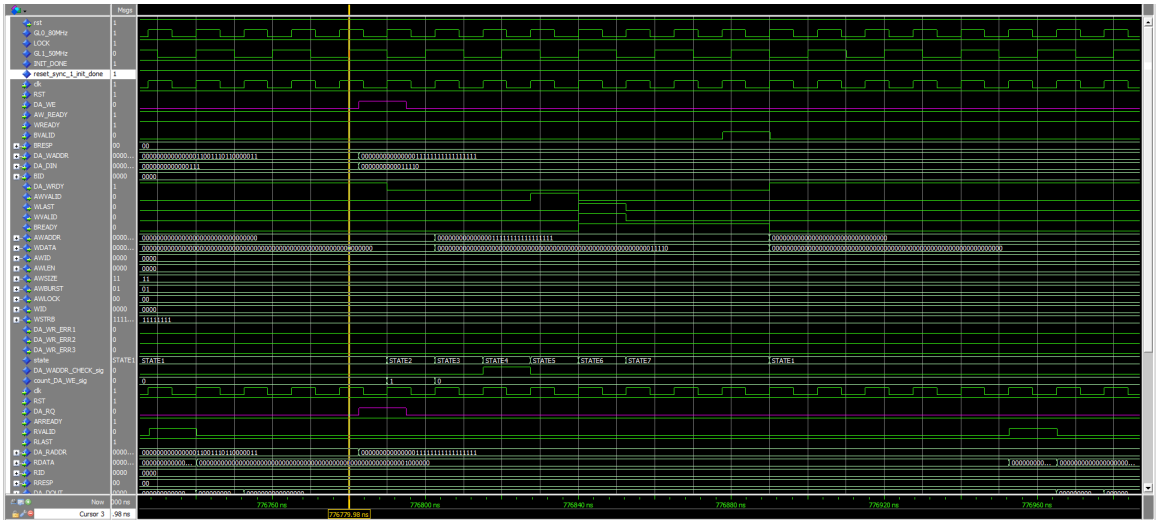

Figure 8.7: Writing of the concurrent writing and reading in Direct Access block simulation

It means that there is a user, which want to write a data in memory, and another one which want to read a data in memory. In the above figure, all the signals of the write part

could be observed and all the execution of the write Direct Access FSM is reported. In this case, the user want to write in the *"00000000000000011111111111111111"* address the *"0000000000011110"* data.

While, in Figure 8.8, it is reported the execution of the concurrent reading, in which the user requests to read the data inside the *"00000000000000011111111111111111"* address. Being the same address in which the read Direct Access part and the write Direct Access part want to access, the FDDR controller has the task to choose who is that has the priority.
In fact, since in the below figure it is possible to observe that the read data, DA_Dout, does not coincide with the data that the other user want to write (*"0000000000011110"*), since the read data is *"0000000000000001"*. It means that the FDDR have chosen to give the priority to the reading, which reads the data that was written previously in that address, and then the controller gives the access to that address to the writing.

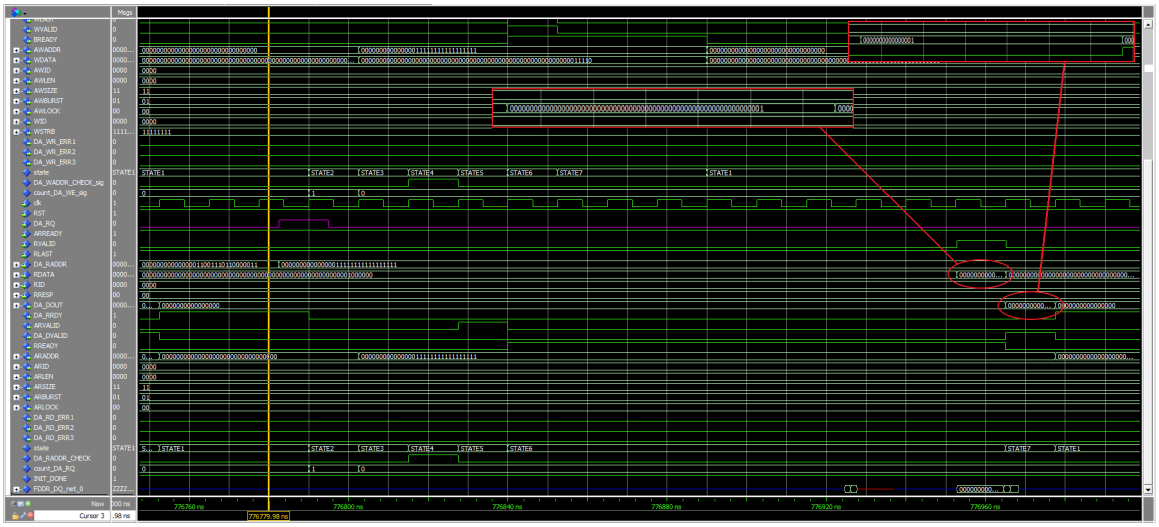

Figure 8.8: Reading of the concurrent writing and reading in Direct Access block simulation

In order to check that effectively, after that the reading has read the data in memory, then, the FDDR makes to access to the same address the user which want to write a data in memory, another reading in the same address has been performed.
This reading is reported in Figure 8.9, in which it could be observed that the user asks to read in *"00000000000000011111111111111111"*.
As it is reported, effectively now the read data is *"0000000000011110"*, which is the data that has been written previously during the concurrent access.
Thus, it is the check that the the previous writing has been executed correctly.

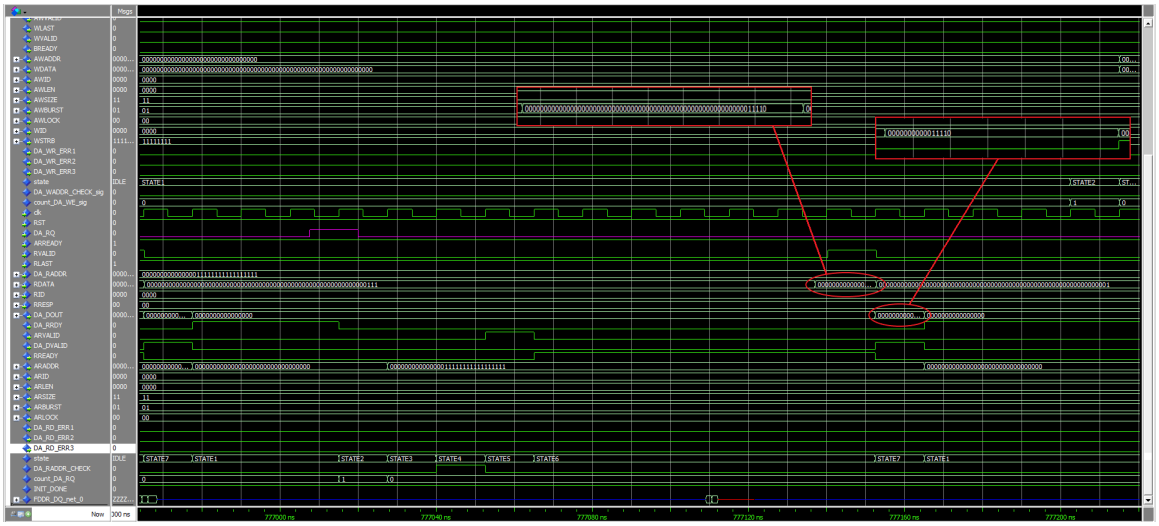

Figure 8.9: Simulation to check the right writing during concurrent writing and reading in Direct Access block

The last test about the Direct Access block concerns the check of the error in the case in which the user tries to write or read when the block it is not ready to accept a request, since it is already busy.
In Figure 8.10, there is reported the case in which the block is already executing a writing and the user does a writing request. It is what happens when the DA_WE is asserted when DA_WRDY is de-asserted, since the write Direct Access sub-block is already busy.

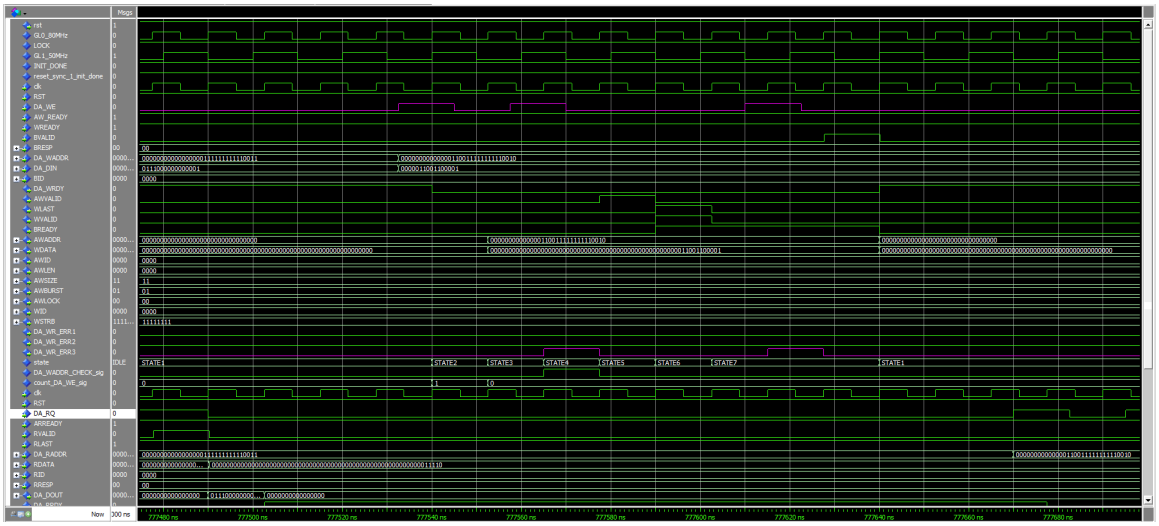

Figure 8.10: Simulation to check the third error in the writing of the write Direct Access block

This simulation has been done in order to check if the **ARS-REQ-310** requirement is satisfied, which says that if DA_WRDY is de-asserted, the access to this port has to be discarded and DA_WRERR3 error flag is raised.
In fact, as it could be observed in the below figure, in *STATE3*, the user asserts the DA_WE, even if the DA_WRDY is de-asserted, since another writing is in execution. So, with this condition, the DA_WR_ERR3 is asserted, in order to communicate to the user that the writing request could not be satisfied at this moment and that this request

has been discarded, since another writing has to be completed.
While, in Figure 8.11, there is the analogue case for the reading. In fact, here it could be observed that in *STATE3* the DA_RQ signal is asserted, since the user want to read a data in memory, even if the read Direct Access block is busy.
Thus, the DA_RD_ERR3 is asserted, to communicate to the user that another reading could not start since a reading is already in execution. This is done to satisfy the **ARS-REQ-410**. In fact, this request of reading has been discarded, since the block is busy at this moment and could not accept other reading before finishing the current one.

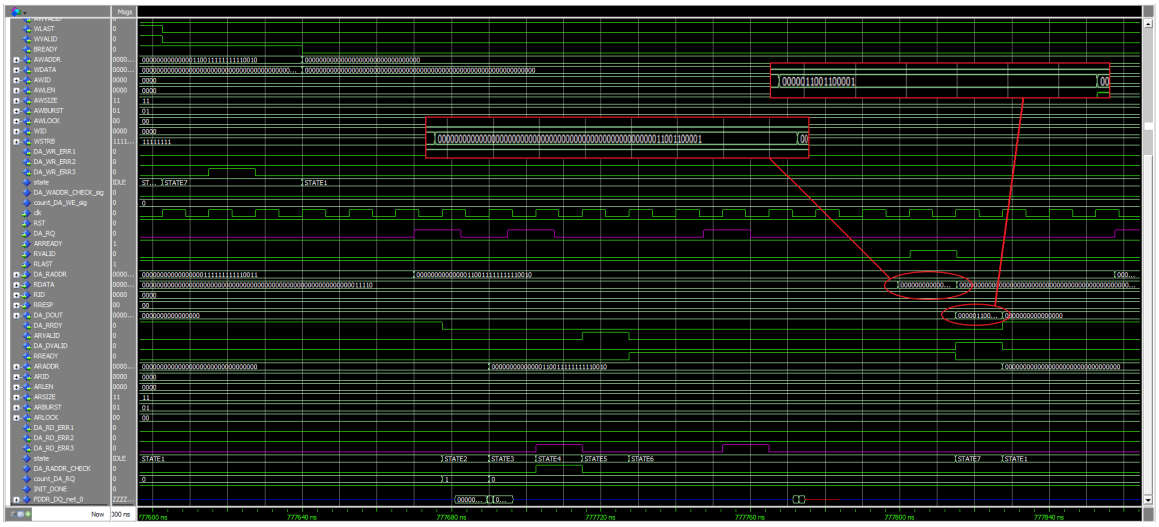

Figure 8.11: Simulation to check the third error in the reading of the read Direct Access block

8.1.1 Code Coverage

During the simulation of the Direct Access block, also the **Code Coverage** has been carried out.
The output of the Code Coverage could be observed in Report 8.1, in which only the code implemented in VHDL has been analyzed, excluding the IP blocks, taken from the library of the RT4G150 FPGA, as the FDDR, the AXI switch, the DDR3 memory banks, the 50 MHz oscillator and the PLL, and also the emulator, since it represent a simulation of the user stimuli, thus, it is not a component of the implemented block.
Therefore, the blocks to which the Code Coverage has been applied are: the Finite State Machines of the read and write Direct Access blocks, the flip flops, which composed the two reset synchronizers, and the reset generator.
From the Code Coverage analyses, some condition have been excluded, like the **when others** clauses in the FSMs, which are used to cover all the unused combination, but since in these FSMs all the states are covered, the machine could not go inside this condition during the simulation and thus, in the case in which this condition does not excluded, it goes to decrement the total coverage. This is why these conditions are excluded by the code condition analyses.
As it could be observed by the below report, both the branches and the statements in all the flip flops of the reset synchronizers are covered with the 100% and the same for the statement of the reset generator.
While, about the finite state machine blocks, in both there are branches and statements that could not be covered during the simulation. In fact, in the write Direct Access block

FSM, based on the theory of the AXI write transaction, reported in Subsection 5.1.1, there is the condition that when the machine is in *STATE5*, if the AWREADY signal is asserted, the machine goes to *STATE6*, while it remain in the same state, but since the AWREADY signal is maintained always asserted by the FDDR block, as it could be observed for example in the simulation in Figure 8.3, the case in which AWREADY is equal to zero could not be satisfy in simulation. The same for the *STATE6*, in which there is the condition to which if WREADY is asserted, the machine goes to *STATE7*, while, if it is de-asserted, the FSM remains in that state, but the FDDR takes the WREADY signal asserted all the time, and thus, the situation in which WREADY=0 could not happen.
Therefore, these are the reason why the branches and the statements of the write Direct Access block are not able to reach the 100%.
Moreover, the same happens for the read Direct Access block, in which the ARREADY is are always maintained raised by the FDDR, as could be observed in Figure 8.4, and thus, the case ARREADY=0 in *STATE5* could not never happen in the simulation. So, because of that, neither in this case, the statements and the branches are able to reach the maximum percentage.
Therefore, at the end, the Total Coverage of the code in this case is 95.14%.

Report 8.1: Direct Access block Code Coverage

```
=================================================================================
=== Instance: /testbench/osc_PLL_reset_0/AXI_master_DA_0/write_DA
=== Design Unit: work.fsm_write_da(behaviour)
=================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                        53        44         9    83.01%
    Conditions                       2         2         0   100.00%
    Statements                     108       102         6    94.44%
=================================================================================
=== Instance: /testbench/osc_PLL_reset_0/AXI_master_DA_0/read_DA
=== Design Unit: work.fsm_read_da(behaviour)
=================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                        51        48         3    94.11%
    Conditions                       2         2         0   100.00%
    Statements                     109       106         3    97.24%
=================================================================================
=== Instance: /testbench/osc_PLL_reset_0/reset_synchr_0/FFRST1
=== Design Unit: work.flipflop(architecture_flipflop)
=================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                         2         2         0   100.00%
    Statements                       2         2         0   100.00%
=================================================================================
=== Instance: /testbench/osc_PLL_reset_0/reset_synchr_0/FFRST2
=== Design Unit: work.flipflop(architecture_flipflop)
=================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                         2         2         0   100.00%
    Statements                       2         2         0   100.00%
=================================================================================
=== Instance: /testbench/osc_PLL_reset_0/reset_synchr_1/FFRST1
=== Design Unit: work.flipflop(architecture_flipflop)
=================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                         2         2         0   100.00%
    Statements                       2         2         0   100.00%
```

```
=== Instance: /testbench/osc_PLL_reset_0/reset_synchr_1/FFRST2
=== Design Unit: work.flipflop(architecture_flipflop)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	2	2	0	100.00%
Statements	2	2	0	100.00%

```
=== Instance: /testbench/reset_generator_0
=== Design Unit: work.reset_generator(behavioural)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Statements	8	8	0	100.00%

```
Total Coverage By Instance (filtered view): 95.14%
```

8.2 FIFO block simulation

Another block of the high-speed data buffer is the FIFO block, composed of the write part and the read one. This has the aim to translate the signals, sent by the user to the FIFO interface, into AXI interface signals.
Even in this case, as in the Direct Access block, what is important at the beginning is the initialization sequence, which is reported in Figure 8.12, which signals are highlighted with magenta color.
The sequence starts with the assertion of the **LOCK** signal, which ensures that the PLL has stable frequencies since it takes the lock. Thus, with a delay, the FDDR asserts the **INIT_DONE** signal, which is the reset of the reset synchronizer related to the 80 MHz. So, this block asserts, after a small delay, the **reset_sync_1_init** signal, which represents the reset of the AXI master block, that in this case is composed only of the FIFO block. At this point, the initialization sequence is terminated and the writing or reading through the FIFO interface could start.

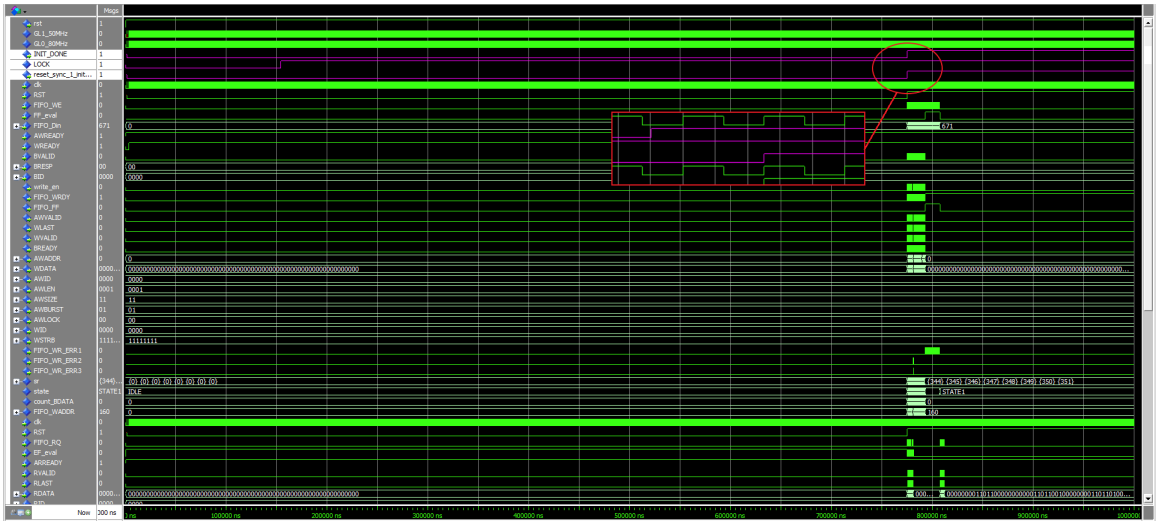

Figure 8.12: Initialization sequence of the FIFO block

To check the FIFO block behaviour, different simulations have been conducted.

The first test, reported in Figure 8.13, concerns the simulation of a writing with FIFO_WE high for 8 clock cycles, as requested by the **ARS-REQ-445** requirement. In the figure, the signals deriving from the FIFO interface are highlight with magenta, while the AXI interface signals are colored with cyan.
From the simulation sketch, it is possible to observe that the FIFO_WE signals is asserted for 8 clock cycles and that for each clock cycle in which FIFO_WE is high, a data is sent through FIFO_Din. The data in the sketch are: 1, 2, 3, 4, 5, 6, 7, 8, which in binary are respectively: "0000000000000001", "0000000000000010", "0000000000000011", "0000000000000100", "0000000000000101", "0000000000000110", "0000000000000111", "0000000000001000". These 8 data, each with a length of 16 bits, are compacted in 2 data, each with a length of 64 bits, in fact, observing the zoom of WDATA, it is possible to observe that the first data is the concatenation of the data number 1, 2, 3, 4 in binary, while the second data is the grouping of the data number 5, 6, 7, 8.
Moreover, as data is sent one by one through the FIFO interface, the data are inserted in an array (**sr** in simulation), which has a depth of 8 and a length 16 bits. This array has the aim to simulate the behaviour of a shift register.
Thus, this writing is concluded correctly without errors during the execution.

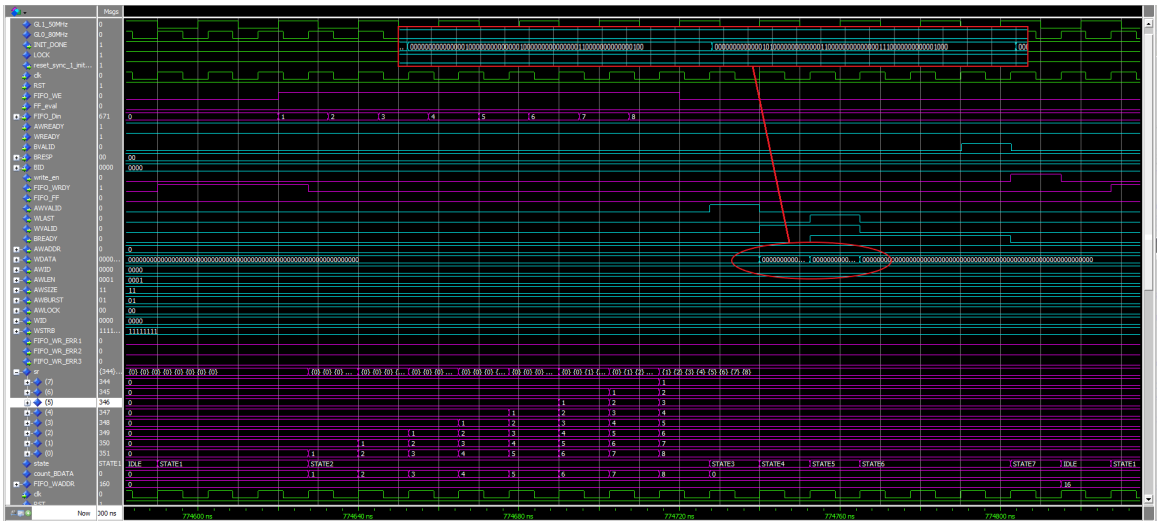

Figure 8.13: Simulation of a correct writing in FIFO block

But, in order to check that effectively the data are written correctly during the previous writing, a test with a reading has been executed, as reported in Figure 8.14.
The read part of the FIFO block receives the two data of 64 bits sent by the FDDR through RDATA. These data have been read starting from the address in which has been written before and then, these have been concatenated in a temporary signal, RDATA_sig, that has a length of 128 bits, and then, the eight original data are obtained, from the partition of the temporary signal, and sent to the user through the FIFO_Dout signal.
From the below figure, it could be observed that, before of the first writing, the Finite State Machine of the read FIFO block was in *STATE01*, since the FIFO was empty, while after the writing, the FSM goes from *STATE01* to *STATE1*, where the reading could start. Moreover, in this simulation, the request of reading, which is represented by the FIFO_RQ signal, has a duration of only one clock cycle, as requested by the **ARS-REQ-485** requirement, thus, no error occurs, in fact, the reading is concluded without any error and the read data result correct since are equal to the ones written previously in that addresses.

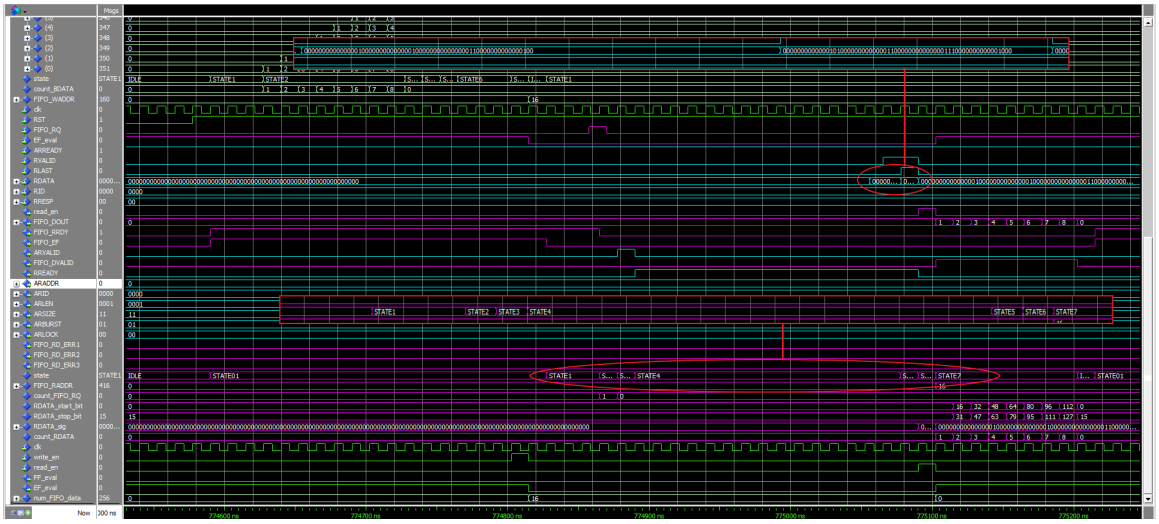

Figure 8.14: Simulation of a correct reading in FIFO block

The previous test composed of a writing followed by a reading is repeated for ten times.
Then, another test has been implemented in order to check the behaviour of the FIFO block, in particular the write part, when the FIFO_WE signal is not asserted for eight clock cycle, but more or less than 8.
It is reported in Figure 8.15, in which the first writing has the FIFO_WE asserted only for five clock cycle, instead of 8, and it is not acceptable, in fact the error FIFO_WR_ERR2 is raised, in order to signalize to the user that the number of clock cycles in which the request signal is asserted is not right.
Even in the second writing, in which the FIFO_WE is high for ten clock cycles, the FIFO_WR_EER2 error flag is asserted, since it is not the right duration of the request signal in order to accept it.

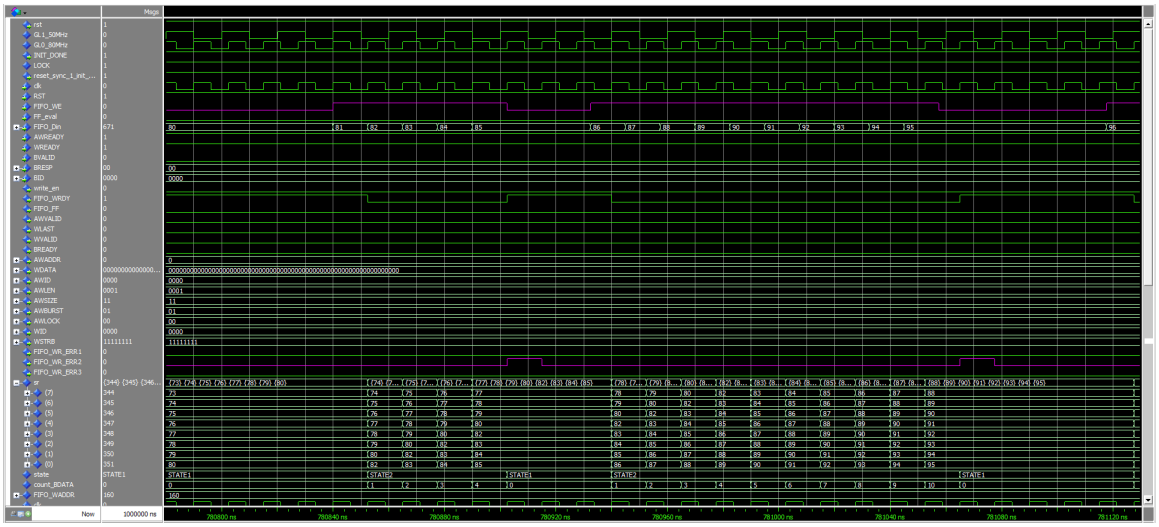

Figure 8.15: Simulation of writing with FIFO_WE high for a number of clock cycles different from 8

This behaviour satisfies the **ARS-REQ-455**, which says that the access to write FIFO port with a burst of length different from 8 has to be discarded and FIFO_WR_ERR2

has to be raised.
In both the cases, the Finite State Machine comes back into the *IDLE*, since the writing has been discarded.
The same test has been computed for the reading, in which the FIFO_RQ should be high only one clock cycle to be correct, while in this simulation it is higher than one and thus, it is not acceptable and according to the **ARS-REQ-495**, the FIFO_RD_ERR2 is asserted in order to communicate to the user, that the duration of the FIFO_RQ is not correct and so, the reading is discarded.

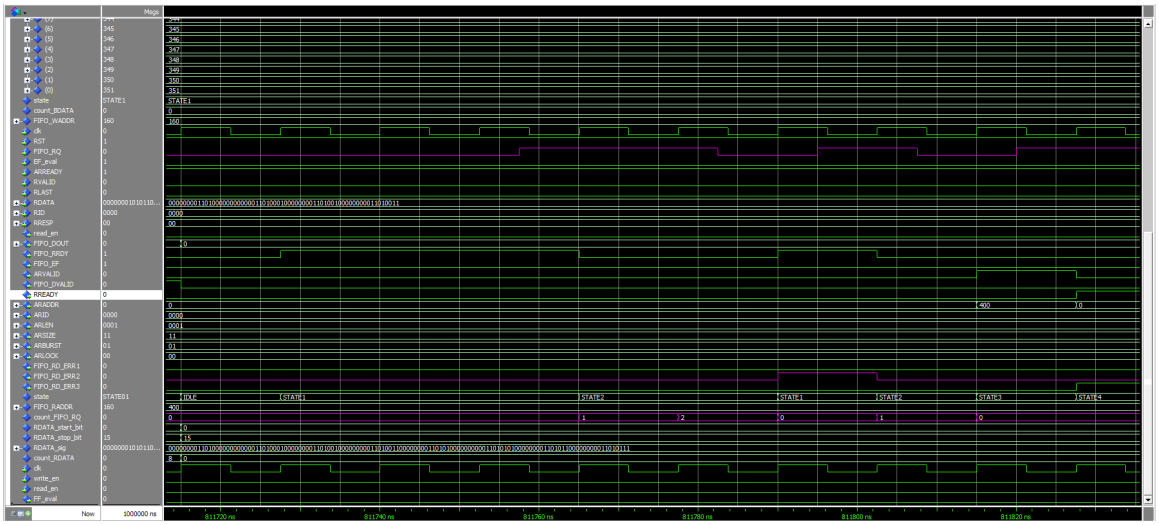

Figure 8.16: Simulation of reading with FIFO_RQ higher than 1 clock cycle

At this point , the satisfaction of the **ARS-REQ-450** and **ARS-REQ-490** requirements is tested.

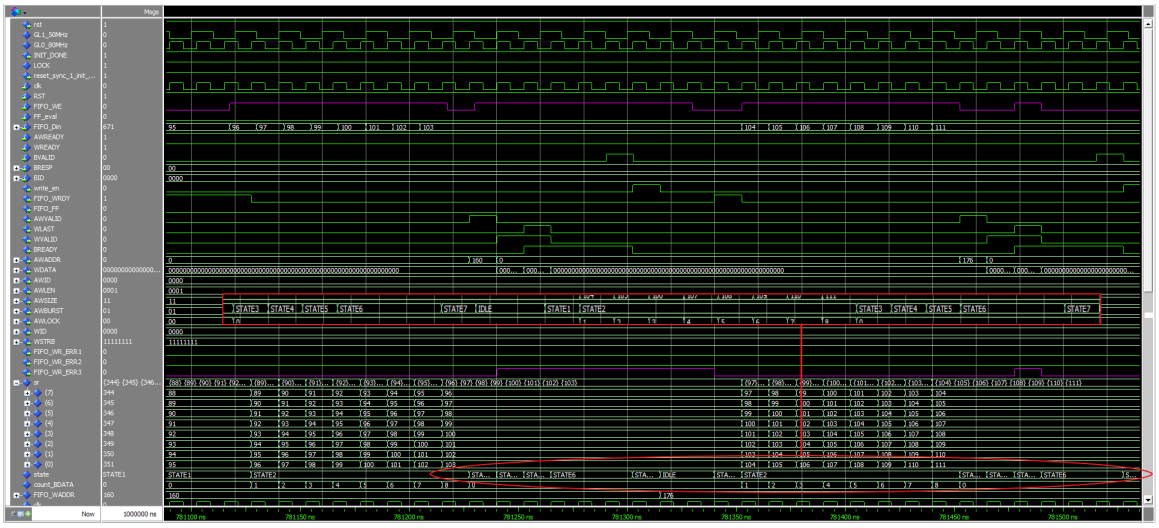

Figure 8.17: Simulation of writing with FIFO_WE high when FIFO_WRDY is low

First of all, a writing has been carried out, as reported in Figure 8.17, in which the FIFO_WE is raised at the moment in which another writing is already in execution and thus, the FIFO_WRDY signal is de-asserted, since the FSM of the write FIFO part is busy.

This condition leads to the assertion of the FIFO_WR_ERR3, which indicates to the user that at this moment the system is busy and it could not accept another writing and so this request has been discarded.
In particular, in this simulation the FIFO_WE is raised, when it cannot acceptable, first of all from *STATE3* to *IDLE*, in order to check that the writing has been discarded in every state in which the FIFO_WRDY is de-asserted, since the FSM is busy, and subsequently, during the execution of another writing, the FIFO_WE has been raised for only one clock cycle, in *STATE5*, and thus, the FIFO_WR_ERR3 is asserted and the request of writing is discarded in order to finish the current writing.
Then, the same test has been done for the reading, as it could be observed in Figure 8.18, in fact, the FIFO_RQ is asserted form *STATE3* to *IDLE* even if the machine is already busy with the execution of another reading. This is why in this case the FIFO_RD_ERR3 is raised, since it could not be acceptable to receive a reading request while another reading is in progress.

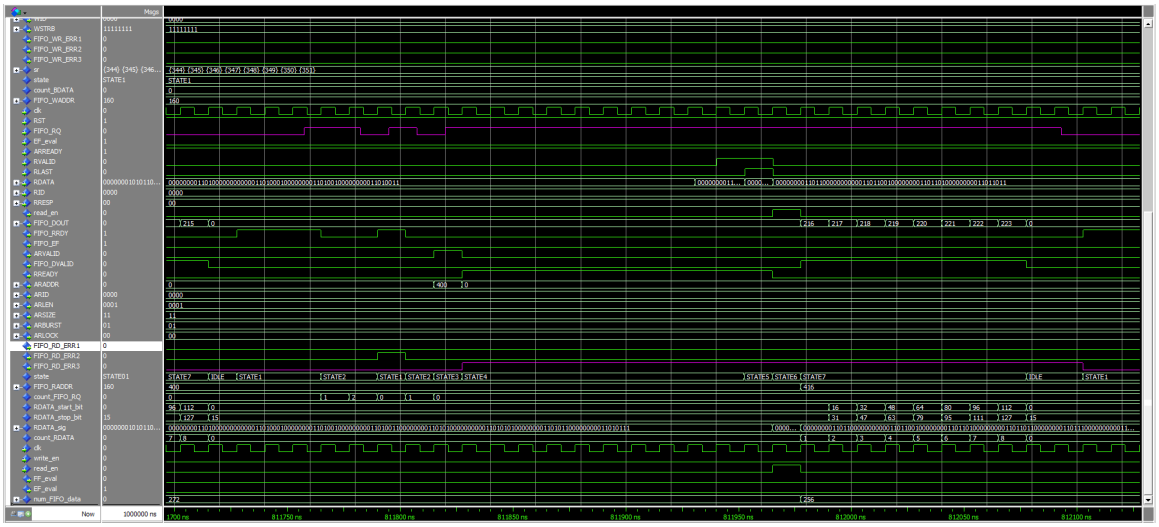

Figure 8.18: Simulation of reading with FIFO_RQ high when FIFO_RRDY in low

The last test executing both for writing and reading is the check of the **ARS-REQ-465** and **ARS-REQ-505** requirements, which want to signalize to the user respectively when the FIFO is full and when it is empty.
Starting from the writing, the related simulation is reported in Figure 8.19. From here, it could be observed that at a certain point, the FF_eval signal is asserted, it means that the FIFO is **FULL**, thus, there are no more locations where the burst of data could be inserted if the user does not read anything. Therefore, the Finite State Machine goes to the *STATE01*, in which it remains till some location becomes free. In this condition, if the user tries to write in the FIFO, this request is discarded, raising the FIFO_WR_ERR1, in order to communicate to the user that the FIFO is full and so no one burst of data could be accepted. In fact, from the below figure, it could be see that when the FF_eval is asserted and so, the machine is in *STATE01*, at the moment in which the user asserts for eight clock cycles the FIFO_WE signals, sending the burst of data composed of 352, 353, 354, 355, 356, 357, 358 and 359, the FIFO_ERR1 is asserted and this writing is discarded.
The same behaviour could be observed even in the two later writings.

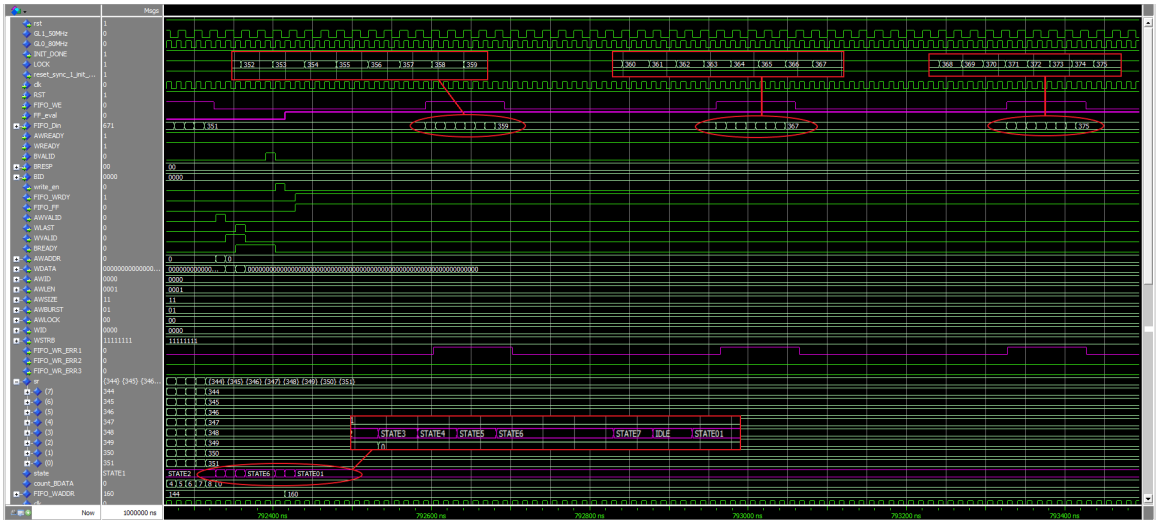

Figure 8.19: Simulation of writing when FULL FIFO

While, in Figure 8.20, it is reported the case in which the FIFO is empty, thus, the Finite State Machine is in *STATE01* where the EF_eval is raised, since, there are no burst of data that could be read.
Therefore, if in this condition, a user tries to read something in memory and so, the FIFO_RQ is asserted, the machine responses with the raising of the FIFO_RD_ERR1, in order to signalize to the user that in this moment the memory is empty and so, the reading has to be discarded, since from this one nothing could be read.
The same could be observed even for the 2 later reading requests.

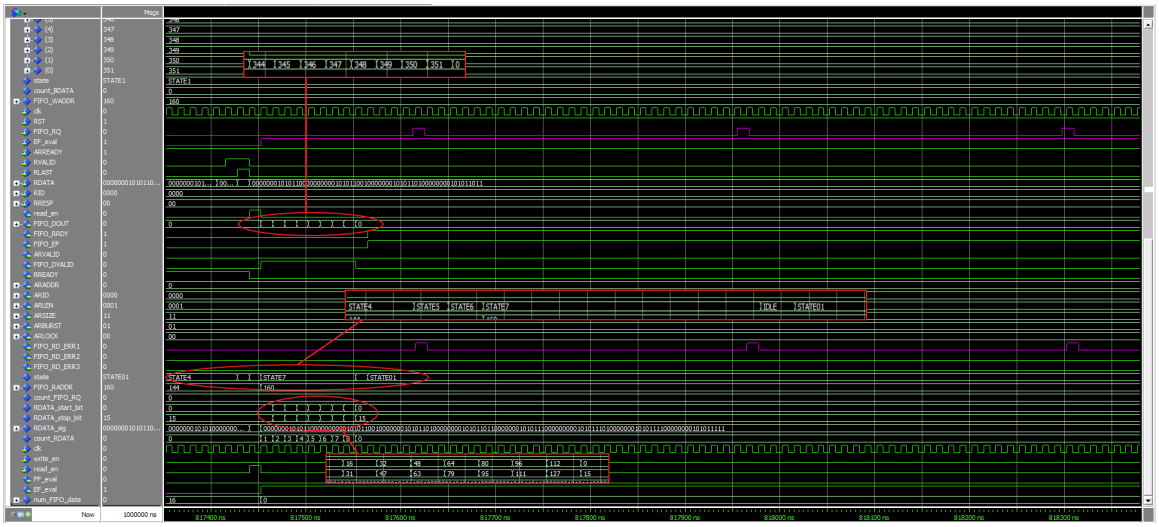

Figure 8.20: Simulation of reading when EMPTY FIFO

At this point, the simulations to test the correct behaviour of the FIFO block, both in writing and in reading are terminated.

8.2.1 Code Coverage

Even in that case, the **Code Coverage** has been executed.
The output of the Code Coverage could be observed in Report 8.2, in which only the

VHDL blocks have been analyzed, excluding the IP blocks taken from the RT4G150 FPGA, since these are blocks already existing on the board, and the emulator, which is inserted in order to send the stimuli to the structure, simulating one or more users, but it is not part of the designed project.
From the below report, it could be seen that there are some blocks that do not satisfy to the 100% the coverage, like the Finite State Machines of the write FIFO block, of the read FIFO block and the evaluator block. It happens because there are some conditions that could not be covered, for example, in the write FIFO Finite State Machine, in *STATE3*, the condition for AWREADY=0 could not be satisfied since the FDDR blocks takes this signal always high, but this condition has been implemented, based on the theory of the AXI protocol reported in Section 5.
The same happens for the WREADY=0 condition both in *STATE4* and *STATE5*, since, even in that case, the FDDR never de-asserts the WREADY signal, thus, this condition could not be satisfied and it leads to a decrement of the code coverage.
Moreover, about the read FIFO Finite State Machine, the code coverage is lower than 100% since, as in the previous case, the FDDR never de-asserts the ARREADY signal, thus, the condition for ARREADY=0 in *STATE5* could not be simulated.
Because of that, the Total Coverage in this case has been resulted equal to 90.19%.

Report 8.2: FIFO block Code Coverage

```
=== Instance: /testbench/reset_generator_0
=== Design Unit: work.reset_generator(behavioural)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Statements	8	8	0	100.00%

```
=== Instance: /testbench/RTG4_fifo_part_0/AXI_master_FIFO_0/write_FIFO
=== Design Unit: work.fsm_write_fifo(behaviour)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	66	56	10	84.84%
Statements	138	128	10	92.75%

```
=== Instance: /testbench/RTG4_fifo_part_0/AXI_master_FIFO_0/read_FIFO
=== Design Unit: work.fsm_read_fifo(behaviour)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	69	60	9	86.95%
Statements	141	132	9	93.61%

```
=== Instance: /testbench/RTG4_fifo_part_0/AXI_master_FIFO_0/evaluator
=== Design Unit: work.evaluator_ff_ef(behavioural)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	10	9	1	90.00%
Statements	11	10	1	90.90%

```
=== Instance: /testbench/RTG4_fifo_part_0/reset_synchr_0/FFRST1
=== Design Unit: work.flipflop(architecture_flipflop)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	2	2	0	100.00%
Statements	2	2	0	100.00%

```
=== Instance: /testbench/RTG4_fifo_part_0/reset_synchr_0/FFRST2
=== Design Unit: work.flipflop(architecture_flipflop)
```

```
=================================================================================
    Enabled Coverage                  Bins      Hits    Misses  Coverage
    ----------------                  ----      ----    ------  --------
    Branches                             2         2         0   100.00%
    Statements                           2         2         0   100.00%
=================================================================================
=== Instance: /testbench/RTG4_fifo_part_0/reset_synchr_1/FFRST1
=== Design Unit: work.flipflop(architecture_flipflop)
=================================================================================
    Enabled Coverage                  Bins      Hits    Misses  Coverage
    ----------------                  ----      ----    ------  --------
    Branches                             2         2         0   100.00%
    Statements                           2         2         0   100.00%
=================================================================================
=== Instance: /testbench/RTG4_fifo_part_0/reset_synchr_1/FFRST2
=== Design Unit: work.flipflop(architecture_flipflop)
=================================================================================
    Enabled Coverage                  Bins      Hits    Misses  Coverage
    ----------------                  ----      ----    ------  --------
    Branches                             2         2         0   100.00%
    Statements                           2         2         0   100.00%
Total Coverage By Instance (filtered view): 90.19%
```

8.3 Final structure with arbiter block simulation

Since, the Direct Access block and FIFO block simulations both for writing and reading parts have checked that their behaviour is correct, now the simulation of the entire structure, reported in Figure 7.37, could be executed.
Even in this case, the simulations executed in the previous sections for Direct Access and FIFO blocks have been executed, in order to check that the behaviour of the blocks is correct even with the assembling of the arbiter structure.
These simulations execute the following tests:

- writing with FIFO_WE high for 8 clock cycles and reading with FIFO_RQ high for 1 clock cycle in FIFO block;
- FIFO_WE is asserted for a number of clock cycles different from 8 in FIFO block. The FIFO_WR_ERR2 is asserted;
- FIFO_WE is asserted when FIFO_WRDY is de-asserted in FIFO block. The FIFO_WR_ERR3 is raised;
- the FIFO is full, thus, FIFO_FF is asserted in FIFO block. The FIFO_WR_ERR1 is raised;
- FIFO_RQ is higher than 1 clock cycle in FIFO block. The FIFO_RD_ERR2 is asserted;
- FIFO_RQ is asserted when FIFO_RRDY is de-asserted in FIFO block. The FIFO_RD_ERR3 is raised;
- the FIFO is empty, thus, FIFO_EF is asserted in FIFO block. The FIFO_RD_ERR1 is raised;
- writing with DA_WE high for 1 clock cycle and reading with DA_RQ high for 1 clock cycle in Direct Access block;

- DA_WE is higher than one clock cycle in Direct Access block. The DA_WR_ERR1 is raised;

- DA_WE is raised when DA_WRDY is de-asserted in Direct Access block. The DA_WR_ERR3 is asserted;

- the address where the user want to read is not belong to the Direct Access memory part, but to the FIFO memory part. The DA_WR_ERR2 is raised;

- DA_RQ is higher than one clock cycle in Direct Access block. The DA_RD_ERR1 is raised;

- DA_RQ is raised when DA_RRDY is de-asserted in Direct Access block. The DA_RD_ERR3 is asserted;

- the address where the user want to read is not belong to the Direct Access memory part, but to the FIFO memory part. The DA_RD_ERR2 is raised.

These simulations have resulted satisfactory, since, these respect the correct behaviour and the related FPGA (ARS) requirements.
In these cases, the simulation sketches are not reported, since these gives the same results of the simulations in Sections 8.1 and 8.2.
After these, other simulations have been executed in order to check that the arbiter behaviour is correct, both in writing and in reading.
Thus, starting from the writing, a first test has been conducted, in which two users want to write at the same time, one through the Direct Access interface and the other one using the FIFO interface.
In Figure 8.21, it could be observed that the FIFO_WE and DA_WE signals are raised at the same time, and thus, two users required at the to access to the same sources through different interfaces.
What is important to remember is that the writing arbiter Finite State Machine is sensible to the FIFO_AWVALID and DA_AWVALID signals in order to choose to which give the priority to access the memory banks, and it is not sensible to FIFO_WE and DA_WE signals. Thus, depending on the FSM latency from when the requested signals (FIFO_WE and DA_WE) are asserted and to when the FIFO_AWVALID and DA_AWVALID are raised, it could be the case in which the AWVALID signals are not raised in the same time and so, the arbiter has not to choose. There is the need to find the precise moment in which the concurrent writings shifting reach the case in which both FIFO_AWVALID and DA_AWVALID are high at the same time.
In the below figure, it is reported the case in which the requests arrive at the same time, but, since the Direct Access FSM is faster, the validation of the writing address signal (DA_AWVALID) is asserted before of the one of the FIFO interface, and so it takes the access.

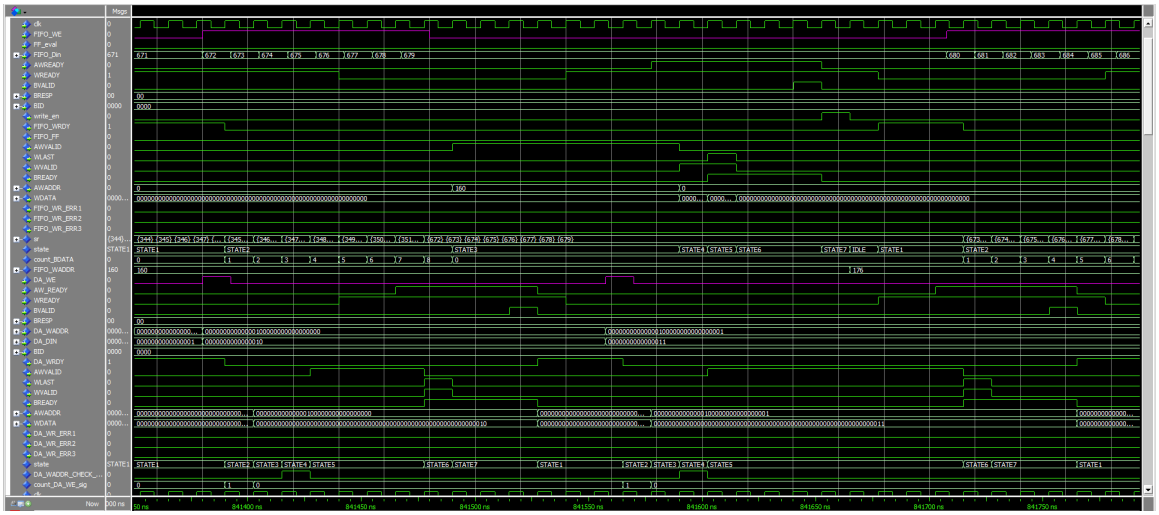

Figure 8.21: Simulation of simultaneous writing from FIFO and Direct Access interfaces

In fact, in Figure 8.22, it could be observed that the DA_AWVALID is raised before of FIFO_AWVALID, and so, the FSM of the writing arbiter decides to set the selector (SEL) of the multiplexer to 1, in order to turn the multiplexer to the Direct Access signals, obtaining these at the output. Thus, in that case, it is the direct access signals, converted in AXI ones, which are sent to the FDDR controller.
Moreover, since the Direct Access interface has been chosen, the choice_bit has been updated to 1, thus, if at the next writing there is a concurrent access, the arbiter could know which one has executed the access previously, given the priority at the other one. It is used to respect the Round Robin algorithm.

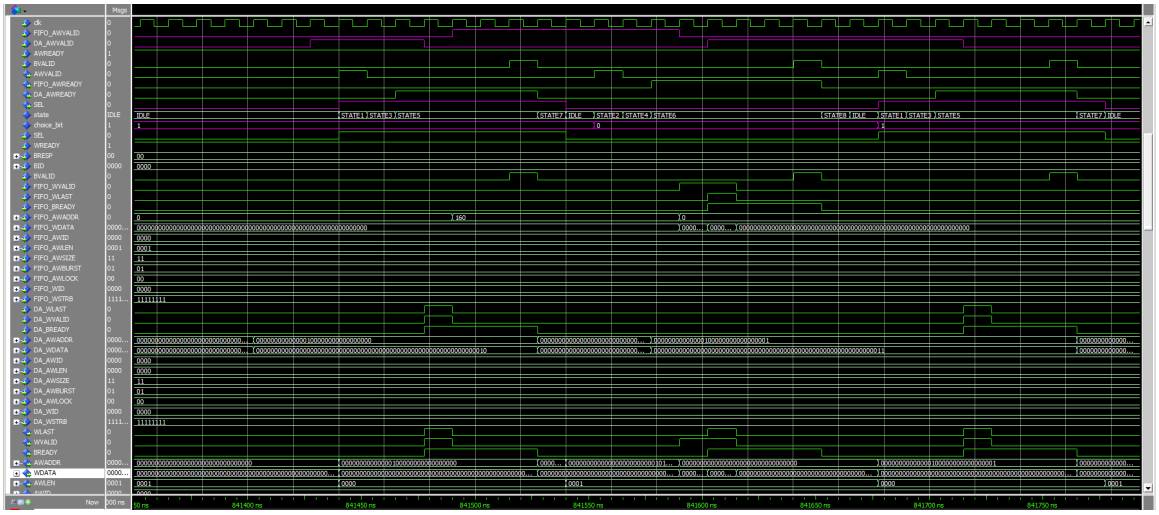

Figure 8.22: Simulation of Arbiter behaviour for simultaneous writing from FIFO and Direct Access interfaces

At the moment in which the Direct Access terminates its writing, the writing through the FIFO interface, which has been stalled, since the FDDR and the memory banks were busy by the Direct Access writing, could be continued. In fact, from figure 8.21, it is shown that the write FIFO FSM, which was stalled in *STATE3*, when the write Direct Access FSM ends its execution, could restart arriving to the end of the writing. It could

be observed even from the behaviour of the arbiter in Figure 8.22, since here the arbiter goes from *STATE1*, *STATE3*, *STATE5* and *STATE7*, which are the state of the writing arbiter dedicated to the Direct Access side, to *STATE2*, *STATE4*, *STATE6* and *STATE8*, which are the state which concern the FIFO side.
Moreover, also the choice_bit has been changed to 0 and the multiplexer selector is set to 0, in order to turn the output to the FIFO interface signals.
At this point, the simulation, in which FIFO_AWVALID and DA_AWVALID are raised at the same time, has been executed.
In Figure 8.23, there is the correct behaviour of the Direct Access and FIFO interfaces, in order to obtain the right timing, which generates the assertion at the same time of the write address validation signals (FIFO_AWVALID and DA_AWVALID).
In fact, to obtain it, the Direct Access writing request signal (DA_WE) has to be raised at the sixth clock cycle in which the FIFO_WE is asserted, in order to synchronize both the FSMs to put high the FIFO_AWVALID and the DA_AWVALID at the same time, as it could be observed in Figure 8.24.

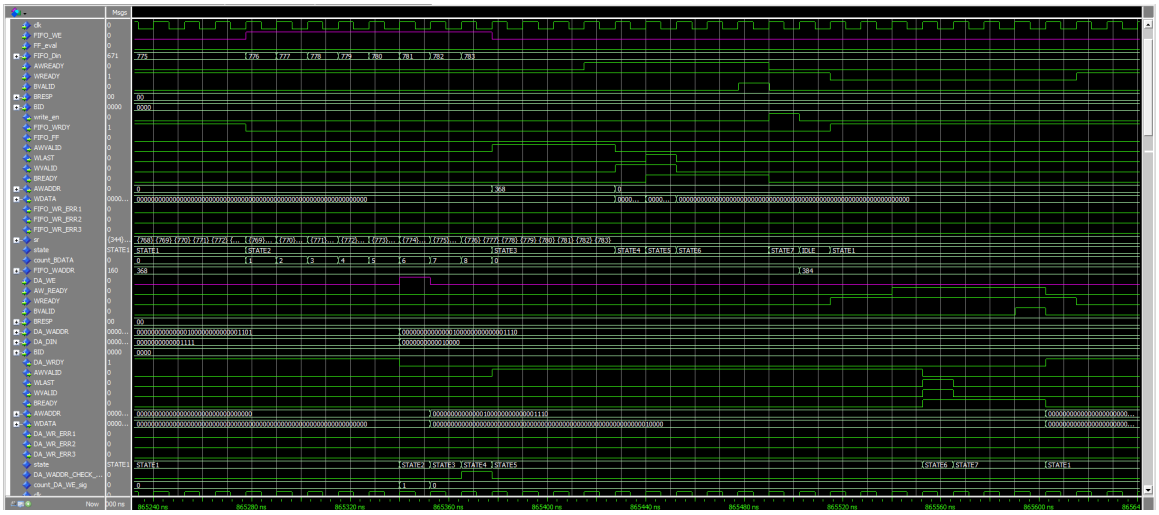

Figure 8.23: Simulation of concurrent writing from FIFO and Direct Access interfaces

The behaviour of the arbiter is reported in the below figure. At this point, the writing block of the arbiter execute its job, since the FIFO_AWVALID and DA_AWVALID are raised at the same time and thus, the arbiter has to choose to which assign the priority. Therefore, it has to look at the choice_bit value to take a decision.
In this case, the choice_bit is equal to 1, it means that previously the Direct Access block accessed to the memory banks, and so, now the priority is given to the FIFO block, in fact, the Finite State Machine of the writing arbiter moves through the branch dedicated to the FIFO access (*STATE2*, *STATE4*, *STATE6* and *STATE8*), the selector is low, in order to turn the multiplexer to the FIFO signal, and the choice_bit is set to 0, thus, in the next writing, the arbiter understands to which gives the priority, based on who has been choose previously.
When the FIFO terminates its writing, the Direct Access block, whose FSM has been stalled in *STATE5*, could terminates its writing.

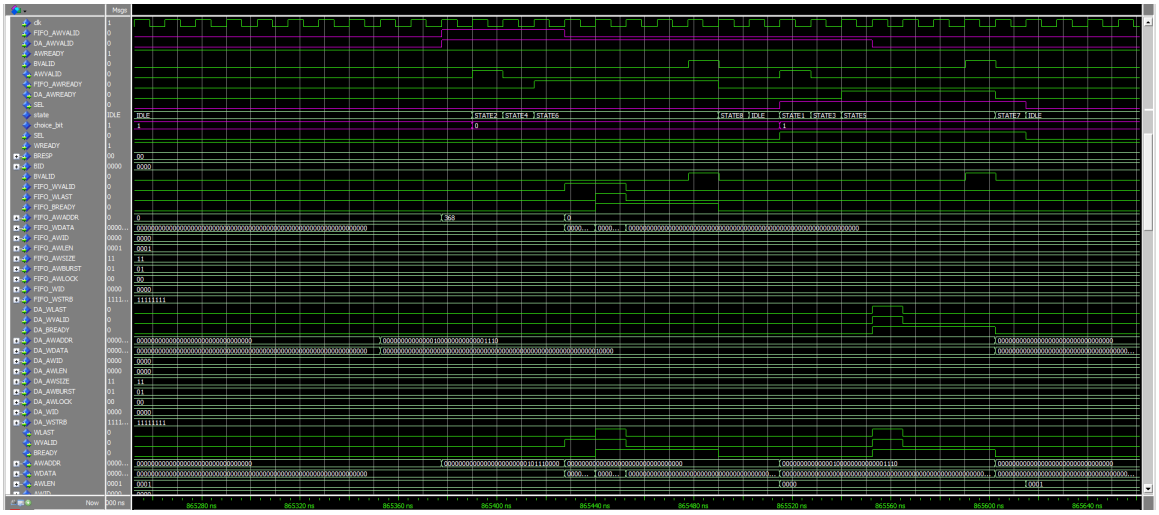

Figure 8.24: Simulation of Arbiter behaviour for concurrent writing from FIFO and Direct Access interfaces

Subsequently, the complementary behaviour of the write arbiter is checked, in which, in case of concurrent writing, if previously the FIFO was accessed, now the arbiter chooses the Direct Access block to access the memory banks.
In fact, in Figure 8.25, there is a FIFO writing execution and then, a concurrent writing one, in which both the FIFO and the Direct Access blocks want to access the memory banks through the FDDR controller.

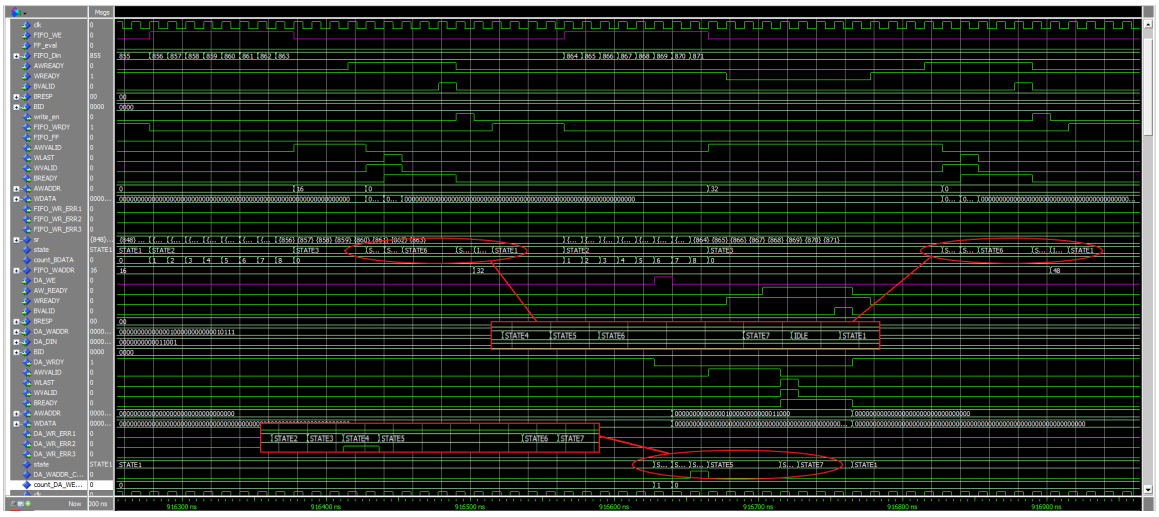

Figure 8.25: Simulation of concurrent writing from FIFO and Direct Access interfaces

In that case, since, before of the concurrent writing, a FIFO writing has been executed, the arbiter chooses to leave the access free for the Direct Access block. It could be observed in Figure 8.26, in which the FIFO_AWVALID and DA_AWVALID are asserted together, so, the write Arbiter decides that the Direct Access has the priority, since the choice_bit is equal to 0.

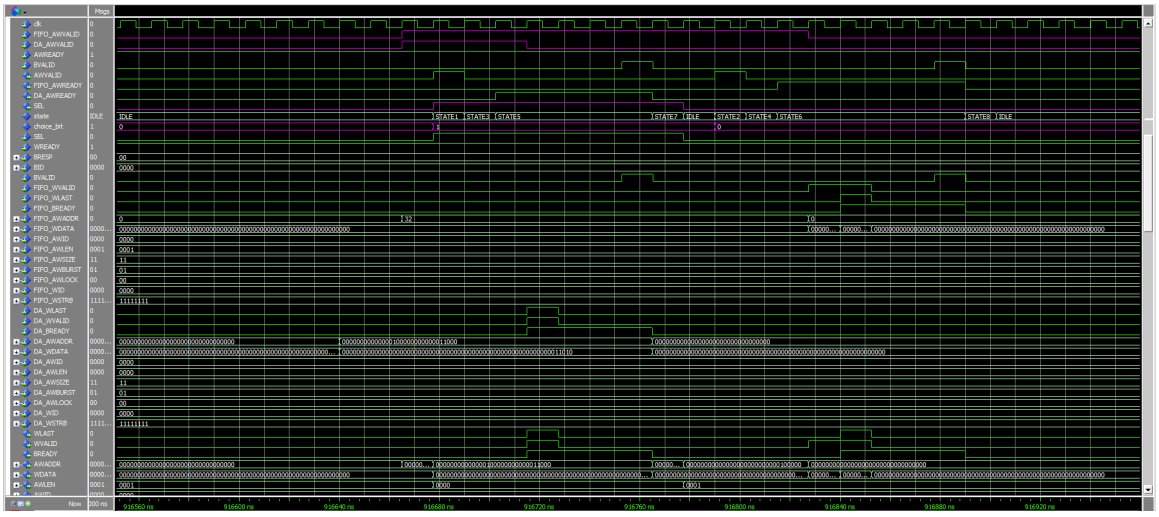

Figure 8.26: Simulation of Arbiter behaviour for concurrent writing from FIFO and Direct Access interfaces

Therefore, the branch dedicated to the Direct Access block (*STATE1*, *STATE3*, *STATE5* and *STATE7*) has been executed by the write Arbiter FSM, and so, the SEL signal is set to 1 in order to obtain, at the output of the multiplexer, the signals sent by the Direct Access block. Then, when the Direct Access block has terminate the writing, the FIFO block, which has been stalled in *STATE3*, could complete its writing.
At this point, the correct behaviour of the write Arbiter has been checked, therefore, the one of the read Arbiter has to be controlled with other simulations.
So, focusing on the arbiter behaviour in case of reading, in Figure 8.27, it could be observed that two user required to access the memory banks one through the FIFO interface, asserting the FIFO_RQ, and one through the Direct Access interface, raising the DA_RQ signal. These two signals are not asserted at the same time, but these are raised with a certain delay in order to obtain that the FIFO_ARVALID and the DA_ARVALID signals which go high at the same time. This is done because of the reading part of the arbiter is sensible to the FIFO_ARVALID and DA_ARVALID and it has to take a choice when both these signals are raised together, it is not based on the request signals.

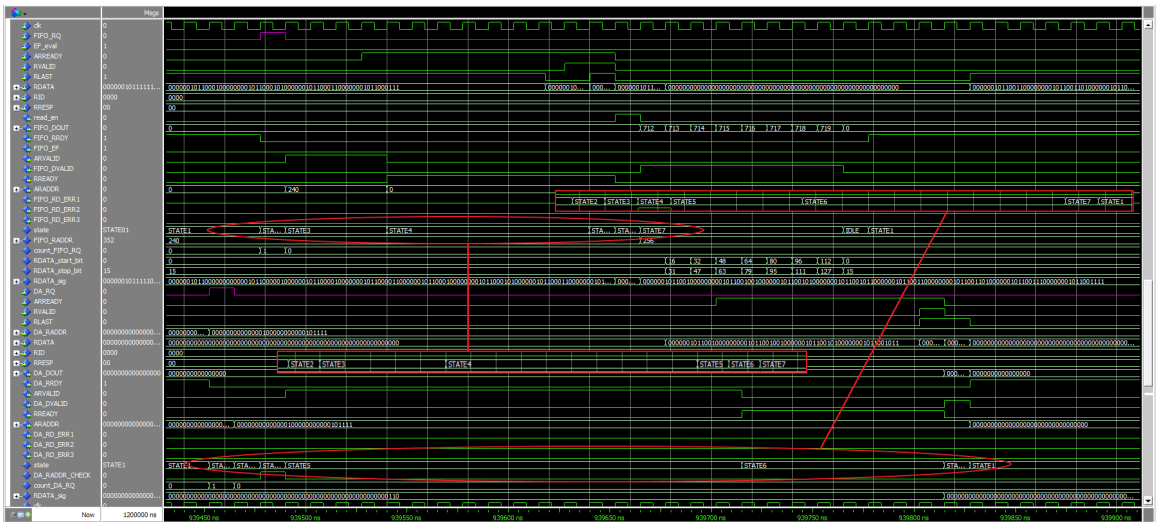

Figure 8.27: Simulation of concurrent reading from FIFO and Direct Access interfaces

In Figure 8.28, there is the signals of the arbiter part dedicated to the reading. From this one, it could be seen that both the read address validation signals (FIFO_ARVALID and DA_ARVALID) are raised at the same time, thus, the arbiter has to look at the choice_bit value in order to choose to which give the priority to enter the memory banks. In that case, the choice bit is 1 and it means that previously the priority has been assigned to the Direct Access interface. Therefore, based on the Round Robin algorithm, the arbiter decides to leave the entry free to the FIFO interface and the choice_bit is changed to 0 in order to signalized to the next reading to which has been given the priority.
At this point, the read Finite State Machine branch dedicated to the FIFO interface has been executed in order to turn the multiplexer to the FIFO signals, in fact, the selector (SEL) is maintained to 0.
Then, when the reading through the FIFO interface is terminated, the execution of the FSM of the Direct Access block, which has been stalled in *STATE5*, could go forward to the end of its reading. In fact, from the below figure, it could be observed that when the read Arbiter Finite State Machine comes back to *IDLE* state, then, it runs across the branch dedicated to the Direct Access reading execution, which is composed of *STATE1*, *STATE3*, *STATE5* and *STATE7*.

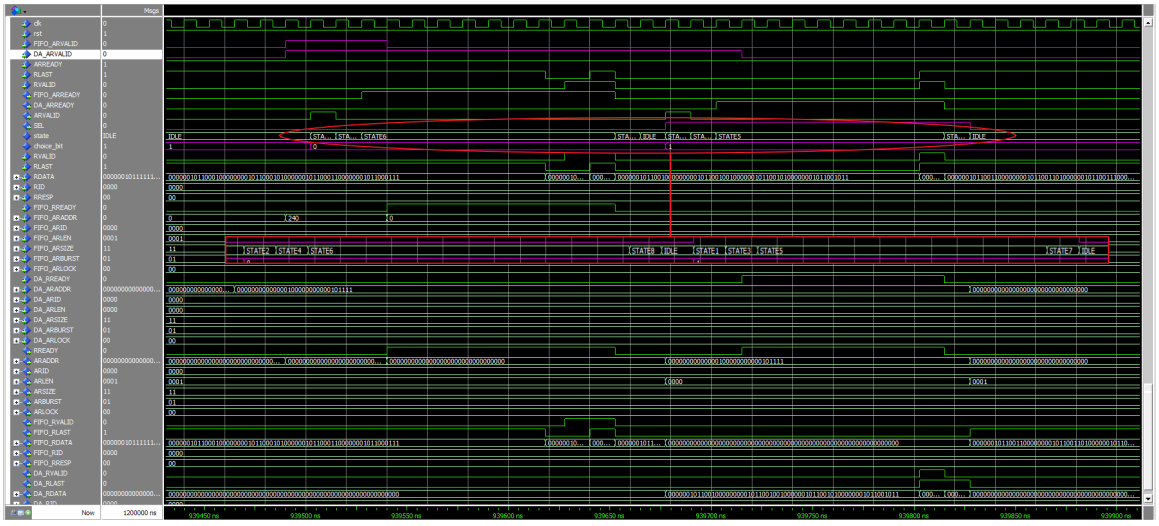

Figure 8.28: Simulation of Arbiter behaviour for concurrent reading from FIFO and Direct Access interfaces

Terminated this simulation, a complementary one is executed, in order to check if, after that in the previous reading the FIFO interface signals have been passed, with a concurrent access, the read Arbiter gives correctly the priority to Direct Access interface or not.
This simulation is reported in Figures 8.29 and 8.30. From the first one, it is shown that at the beginning there is a FIFO reading, since only the FIFO has requested to read and after the end of it, there is a concurrent access request by two users, one through the FIFO interface and the other one through the Direct Access interface.

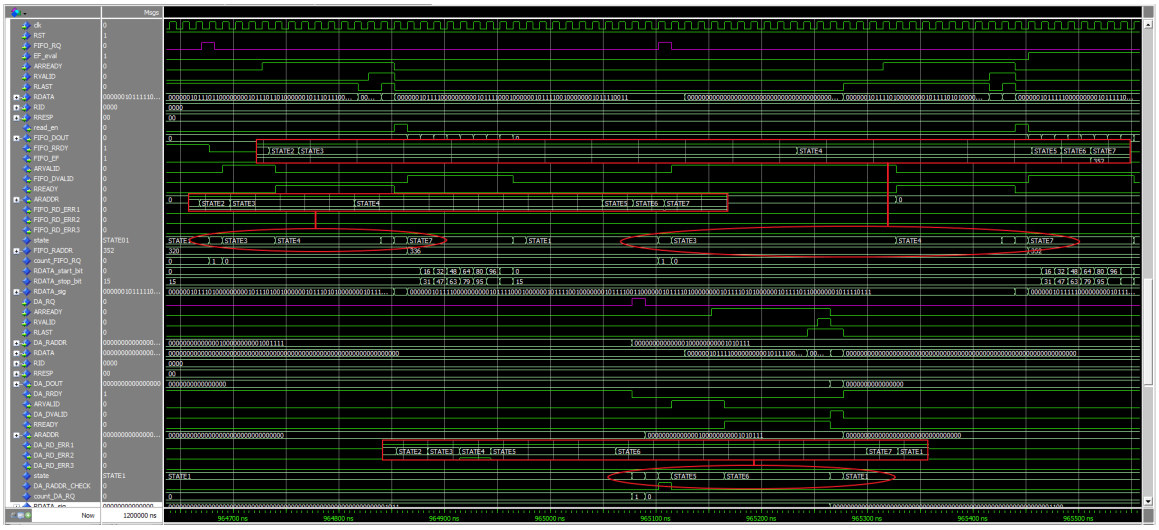

Figure 8.29: Simulation of concurrent reading from FIFO and Direct Access interfaces

At this point, in Figure 8.30, there is reported the sketch in which it could be observed the read Arbiter behaviour, in fact, in this case, it receives both the FIFO_ARVALID and the DA_ARVALID asserted at the same time, thus, it needs to look at the choice_bit value, which being 0, indicates that in the previous reading the priority has been assigned to the FIFO interface, so, at this moment, the priority has been given to the Direct Access one, the choice_bit is changed to 1 for the next reading and the SEL signal is equal to 1, in order to rotate the multiplexer to the Direct Access signals.
Even in this case, after that the Direct Access reading is terminated, the FIFO one could finish its execution, which has been stalled in *STATE3*.

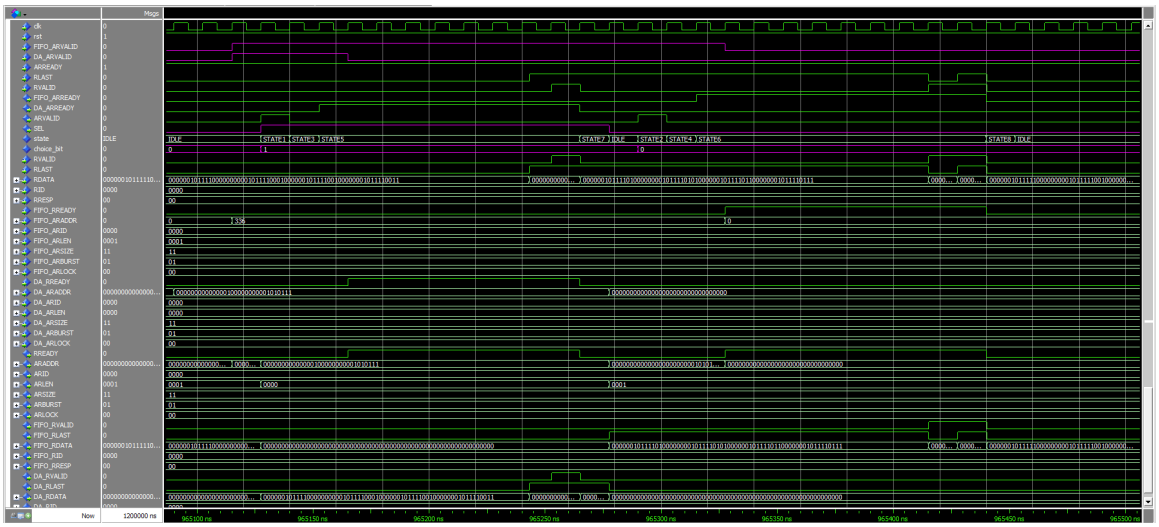

Figure 8.30: Simulation of Arbiter behaviour for concurrent reading from FIFO and Direct Access interfaces

This terminates the simulations to test the correct behaviour of the entire structure, composed of the FIFO block, the Direct Access block, the Arbiter block, each with a writing part and reading one, which are connected to the FDDR controller, the four DDR3 memory banks, the reset generator, the structure which generates the 50 MHz and the 80 MHz and the user emulator.

8.3.1 Code Coverage

Also for the final structure, which contains the Direct Access block, the FIFO block and the arbiter one, the **Code Coverage** has been executed, in order to check how much code has been simulated and to understand why some parts of code are not simulated, if there are.
Even in this case, the reason why the Finite State Machines do not have a 100% of code coverage, it is due to the fact that the FDDR block takes always raised the AWREADY and WREADY signals as regards the write parts and the ARREADY signal for the read parts.
While, based on the theory of the AXI interface, also the case in which these signals are de-asserted should happen.
Therefore, because of that, the Code coverage of the final structure results equal to 95.90%, as could be observed in the below report.

Report 8.3: Final block with arbiter Code Coverage

```
=== Instance: /testbench/AXI_block/AXI_master_0/AXI_FIFO/write_FIFO
=== Design Unit: work.fsm_write_fifo(behaviour)

=================================================================================
Enabled Coverage                  Bins      Hits    Misses  Coverage
----------------                  ----      ----    ------  --------
Branches                            66        56        10    84.84%
Statements                         138       127        11    92.02%
=================================================================================
=== Instance: /testbench/AXI_block/AXI_master_0/AXI_FIFO/read_FIFO
=== Design Unit: work.fsm_read_fifo(behaviour)

=================================================================================
Enabled Coverage                  Bins      Hits    Misses  Coverage
----------------                  ----      ----    ------  --------
Branches                            69        62         7    89.85%
Statements                         141       133         8    94.32%
=================================================================================
=== Instance: /testbench/AXI_block/AXI_master_0/AXI_FIFO/evaluator
=== Design Unit: work.evaluator_ff_ef(behavioural)

=================================================================================
Enabled Coverage                  Bins      Hits    Misses  Coverage
----------------                  ----      ----    ------  --------
Branches                            10         9         1    90.00%
Statements                          11        10         1    90.90%
=================================================================================
=== Instance: /testbench/AXI_block/AXI_master_0/AXI_DA/write_DA
=== Design Unit: work.fsm_write_da(behaviour)

=================================================================================
Enabled Coverage                  Bins      Hits    Misses  Coverage
----------------                  ----      ----    ------  --------
Branches                            50        47         3    94.00%
Statements                         108       105         3    97.22%
=================================================================================
=== Instance: /testbench/AXI_block/AXI_master_0/AXI_DA/read_DA
=== Design Unit: work.fsm_read_da(behaviour)

=================================================================================
Enabled Coverage                  Bins      Hits    Misses  Coverage
----------------                  ----      ----    ------  --------
Branches                            48        47         1    97.91%
Conditions                           2         2         0   100.00%
Statements                         100        99         1    99.00%
=================================================================================
=== Instance: /testbench/AXI_block/AXI_master_0/wr_arbiter/fsm
=== Design Unit: work.fsm_write_arbiter(behavioural)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	46	42	4	91.30%
Statements	68	64	4	94.11%

=== Instance: /testbench/AXI_block/AXI_master_0/wr_arbiter/mux
=== Design Unit: work.mux_write_arbiter(behavioural)

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	2	2	0	100.00%
Statements	41	41	0	100.00%

=== Instance: /testbench/AXI_block/AXI_master_0/rd_arbiter/fsm
=== Design Unit: work.fsm_read_arbiter(behavioural)

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	42	40	2	95.23%
Conditions	4	4	0	100.00%
Statements	66	64	2	96.96%

=== Instance: /testbench/AXI_block/AXI_master_0/rd_arbiter/mux
=== Design Unit: work.mux_read_arbiter(behavioural)

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	2	2	0	100.00%
Statements	35	35	0	100.00%

=== Instance: /testbench/AXI_block/PLL_1_1_0
=== Design Unit: work.rtg4fccc_0(rtl)

Enabled Coverage	Bins	Hits	Misses	Coverage
Statements	7	7	0	100.00%

=== Instance: /testbench/AXI_block/reset_synchr_0/FFRST1
=== Design Unit: work.flipflop(architecture_flipflop)

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	2	2	0	100.00%
Statements	2	2	0	100.00%

=== Instance: /testbench/AXI_block/reset_synchr_0/FFRST2
=== Design Unit: work.flipflop(architecture_flipflop)

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	2	2	0	100.00%
Statements	2	2	0	100.00%

=== Instance: /testbench/AXI_block/reset_synchr_1/FFRST1
=== Design Unit: work.flipflop(architecture_flipflop)

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	2	2	0	100.00%
Statements	2	2	0	100.00%

=== Instance: /testbench/AXI_block/reset_synchr_1/FFRST2

```
=== Design Unit: work.flipflop(architecture_flipflop)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Branches	2	2	0	100.00%
Statements	2	2	0	100.00%

```
=== Instance: /testbench/reset_generator_0
=== Design Unit: work.reset_generator(behavioural)
```

Enabled Coverage	Bins	Hits	Misses	Coverage
Statements	5	5	0	100.00%

```
Total Coverage By Instance (filtered view): 95.90%
```

Chapter 9

Hardware implementation and deployment

At this point, the structure, described till now, has been implemented and deployed on hardware, in particular on the RTG4 development kit, reported in Figure 4.1, which contains the RT4G150 FPGA.
In order to send a request of writing or reading through the Direct Access or FIFO interfaces, the debug switches on the board have been used, in particular the SW1 to send the Direct Access writing request, the SW2 to the Direct Access reading request, the SW3 to the FIFO writing request and the SW4 to the FIFO reading one.
Moreover, to realize this process, other additional blocks have been inserted in the structure, originally composed of the AXI block, the AXI switch, the FDDR, the reset synchronizers and the sub-structure to generate the two clock signals (50 MHz and 80 MHz). The additional blocks are:

- **ringing filter**, which is used to filter out the ringing effect of the switch, when it is pressed. A ringing filter is placed for each switch, in order to obtain a clear edge, without ringing;
- **edge detector**, placed after each filter. It is composed of a flip flop and an AND port, with one of the inputs negated, as reported in Figure 9.1, and it is used to detect the moment in which the switch is pressed, tus, when the switch changes its front, creating an only impulse, with a duration of 1 clock cycle. In this way, the structure becomes sensible to the front instead to the level of the switch signal;

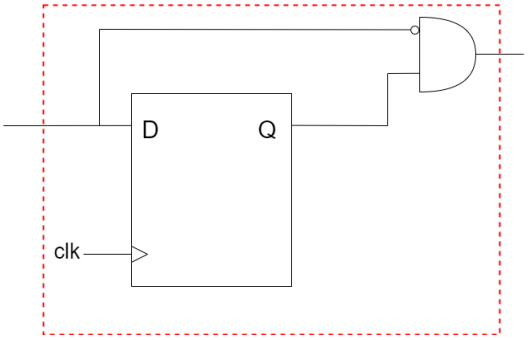

Figure 9.1: Edge detector structure

- **Direct Access signals generator**, which is a customized block sensitive to the

SW1 switch for the writing and SW2 for the reading.
In fact, when the user wants to write and so, when the SW1 switch is pushed, this block generates the writing request signal (DA_WE), the address (DA_WADDR), where the user woulds to write the data, and the input data (DA_Din) to the memory. While, when the user presses the SW2, since it wants to read a data in memory, this customized block generates the reading request (DA_RQ) and the address (DA_RADDR);

- **FIFO signals generator**, which is a customized block, as the previous one, but unlike the Direct Access signal generator, this is sensitive to the SW3 for the writing and the SW4 for the reading. Thus, if the user presses the SW3 switch, the write request (FIFO_WE) and the input data (FIFO_Din) are sent. While, if the SW4 is pushed, the read request (FIFO_RQ) is produced.

The entire structure, with these additional blocks, both for the writing part and the reading one, is reported in Figure 9.2. This is the scheme of the structure loaded on the FPGA for the hardware test, in order to check if the developed structure actually works when it is placed on the hardware.

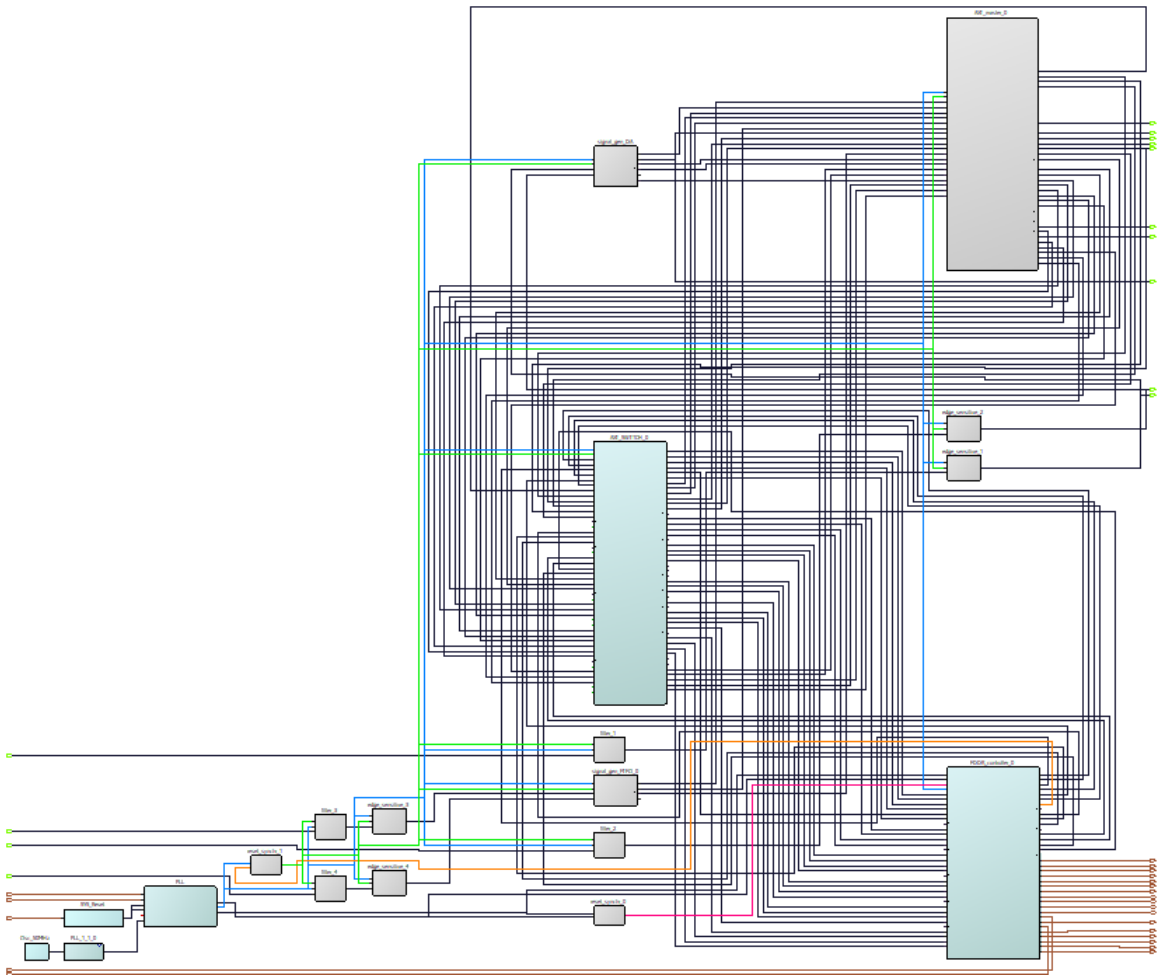

Figure 9.2: Structure for hardware test

Even in this case, the structure is too complex to visualize clearly the components, thus, some zooms of the blocks are done, in order to understand better how the new

blocks are distributed.
In Figure 9.3, the additional blocks, related to the Direct Access part, are reported. In particular, for the writing through the Direct Access interface, the blocks are: the **filter_1**, which has the aim to remove the ringing of the SW1 switch, that is connected as input; the **edge_detector_1**, which is used to identify the edge of the switch when it is pressed, reacting with an impulse of 1 clock cycle; and the **signal_gen_DA** block, whose behaviour has been described in the previous points.
The same is done even for the reading case, which is sensitive to the signal generated by the SW2 switch.

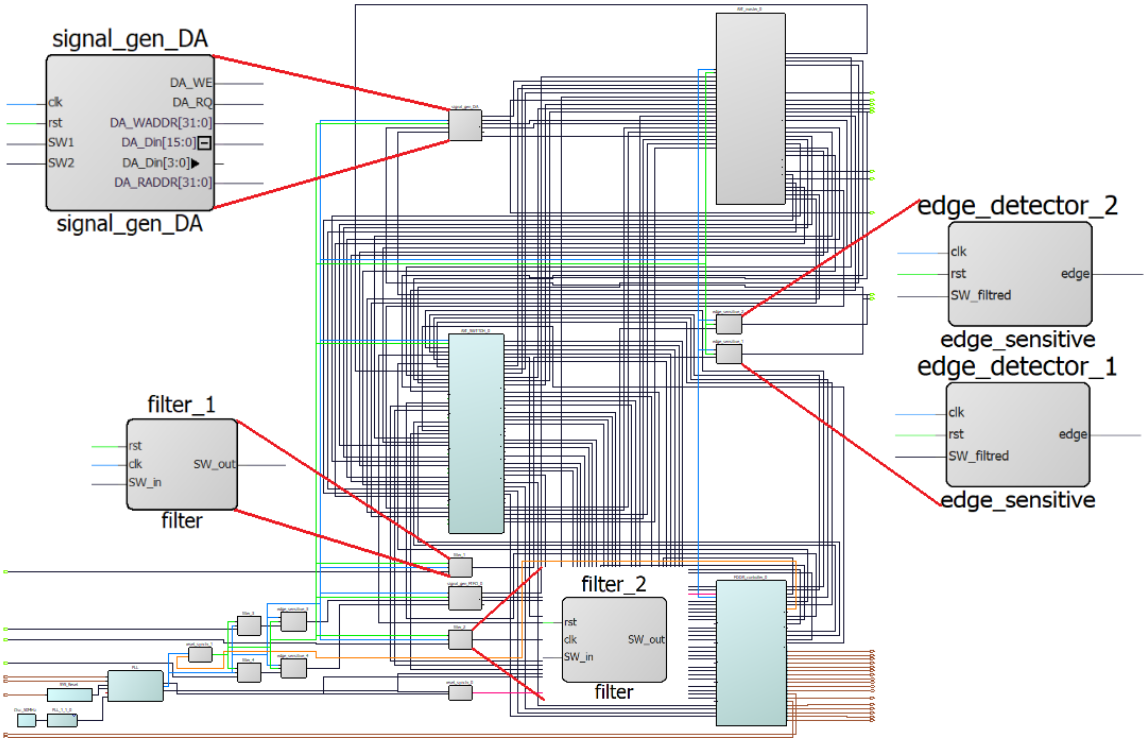

Figure 9.3: Zoom of the Direct Access structure added for the hardware test

While, in Figure 9.4, the additional blocks, concerning the FIFO part, are represented. As the Direct Access case, the SW3 and SW4 are connected each one to a filter, respectively **filter_3** and **filter_4**, whose output is connected to an edge detector (**edge_detector_3** for the writing and **edge_detector_4** for the reading), which detects the edge of the pressed switch, generating an impulse. Then, the output of these edge_detectors is linked to the **signal_gen_FIFO**, which generates the writing request and the FIFO input data if an impulse arrives from the edge_detector_3; and the reading request if an impulse arrives from the edge_detector_4.

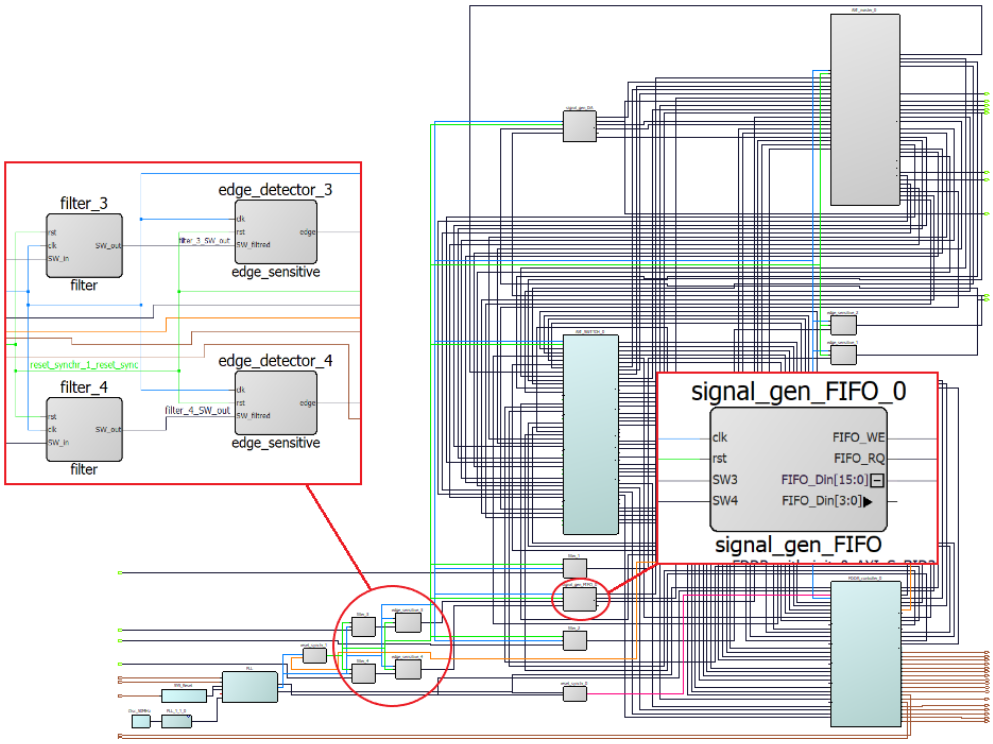

Figure 9.4: Zoom of the FIFO structure added for the hardware test

Moreover, a difference of this structure, with respect to the one used in the simulation, is that, in this case, the reference clock of the PLL is not connected to the 50 MHz oscillator, available in the IP of the RT4G150 FPGA, but it is linked to the 100 MHz LVDS clock oscillator, reported in Figure 4.7, which is a differential clock source available on the evaluation board of the RTG4.
The reason why a differential clock is used is that a differential signaling is not sensitive to the Simultaneous Switching Output (SSO) noise, which is more present in high-speed digital ICs. The problem of the SSO noise is that it produces an unwanted oscillation on the output of the IC, which could be translated into higher bit error rate (BER) at the receiver.
If each wire of pair is on close proximity of one and other, electromagnetic interference imposes the same voltage on both signals, but the difference cancels out the effect.
While, the use of a single ended clock oscillator is risky because single ended signal is subject to distortions and noise, that could not be deleted.

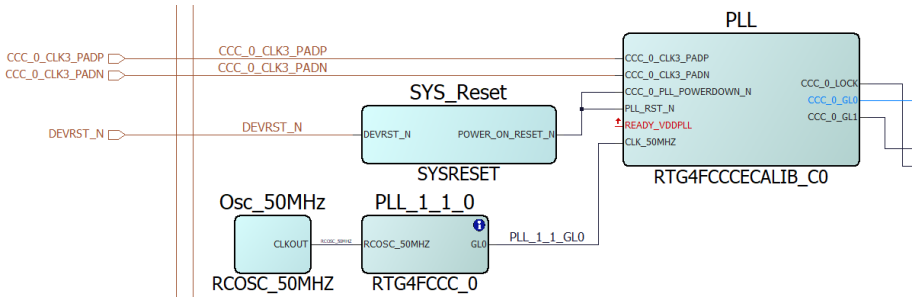

Figure 9.5: PLL structure for hardware test

Therefore, at this point, the RT4G150 FPGA has been programmed thorugh Libero

SoC, loading this structure inside that. During this procedure, the inputs and outputs of the FDDR are connected automatically to the related pins of the banks of DDR3 memory present on the development kit.
To test that the hardware has the same behaviour of the simulations executed with ModelSim, some signals have been represented on a digital oscilloscope. The setup is reported in Figure 9.6. This is composed of the RTG4 Development Kit, the XM105 Xilinx board, the Teledyne LeCroy High Definition Oscilloscope and the digital leadset.

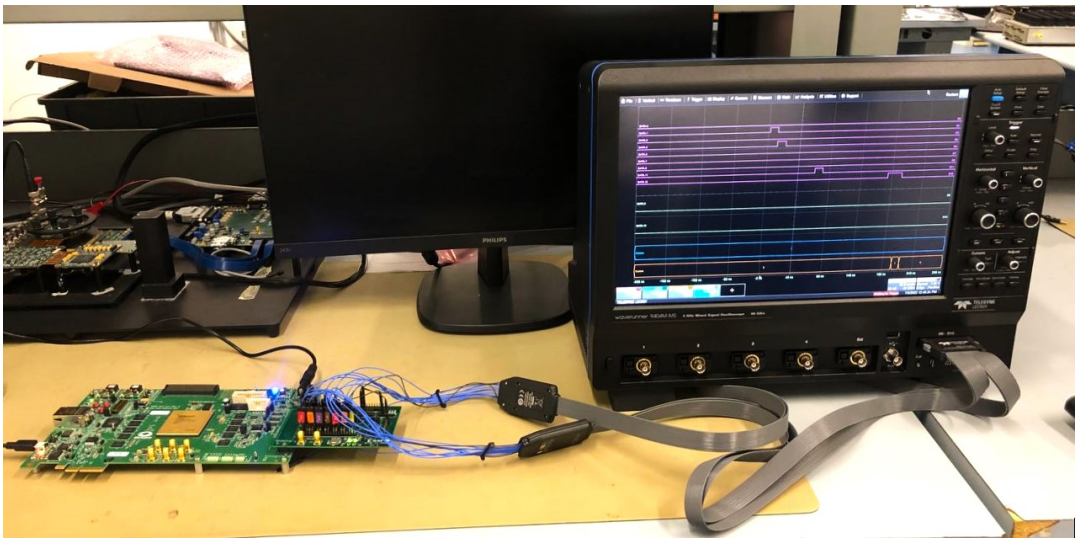

Figure 9.6: Laboratory setup for the tests

In Figure 9.7, it is reported the RTG4 Development Kit used for the tests.
The more important components of this kit used, during the tests, are enumerated and highlighted with red rectangular and these are:

1. RT4G150 FPGA, which is the FPGA of the RTG4 family of the Microsemi. This is programmed with the structure described till now;

2. the power supply switch, which is used to connect a 12 V external DC jack, when the switch is turned on;

3. reset switch (SW7), through which a full reset to the chip could be sent;

4. push-button tactile switches (SW1, SW2, SW3 and SW4) connected to the RTG4 device, which are active low. SW1 is used for the Direct Access writing, SW2 for the Direct Access reading, SW3 for the FIFO writing and SW4 for the FIFO reading;

5. DDR3 memory banks, where the data are written and/or read. In this case, only the memory banks on the west side are used, since the FDDR used in the project is the one on the west side;

6. HPC FMC connector (J34), used for connecting the daughter cards to enable future expansion of interface. In this case, this is used to connect the XM105 Xilinx board.

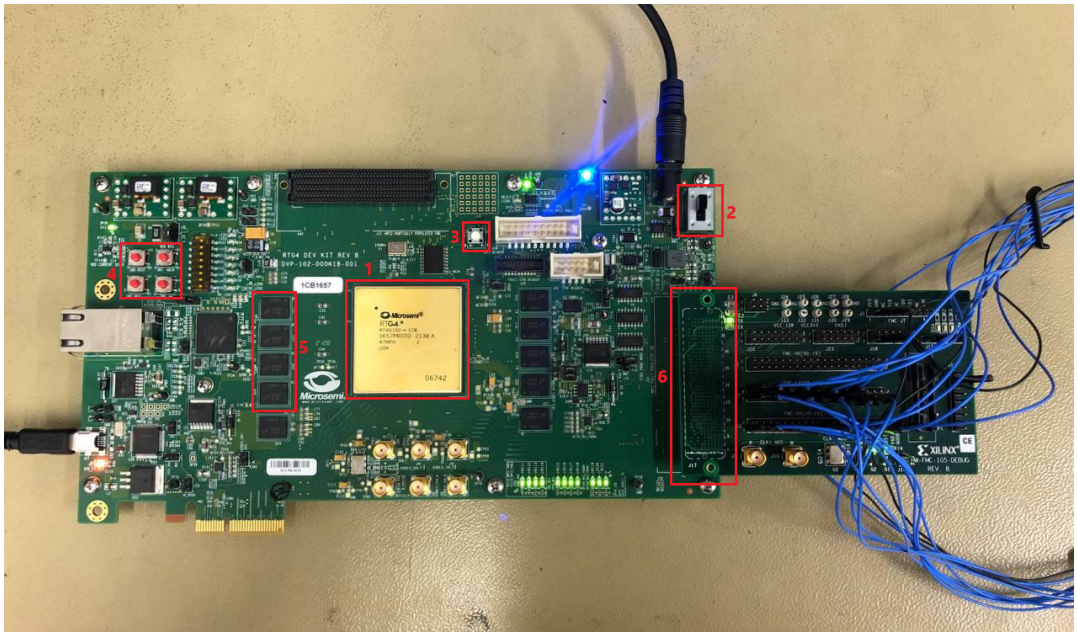

Figure 9.7: RTG4 Development Kit for the test

In Figure 9.8, the XM105 Xilinx board placed on one of the FMC of the Development Kit is reported. This provides a number of multi-position headers and connectors which break out the FPGA interface signals to and from the board interface.

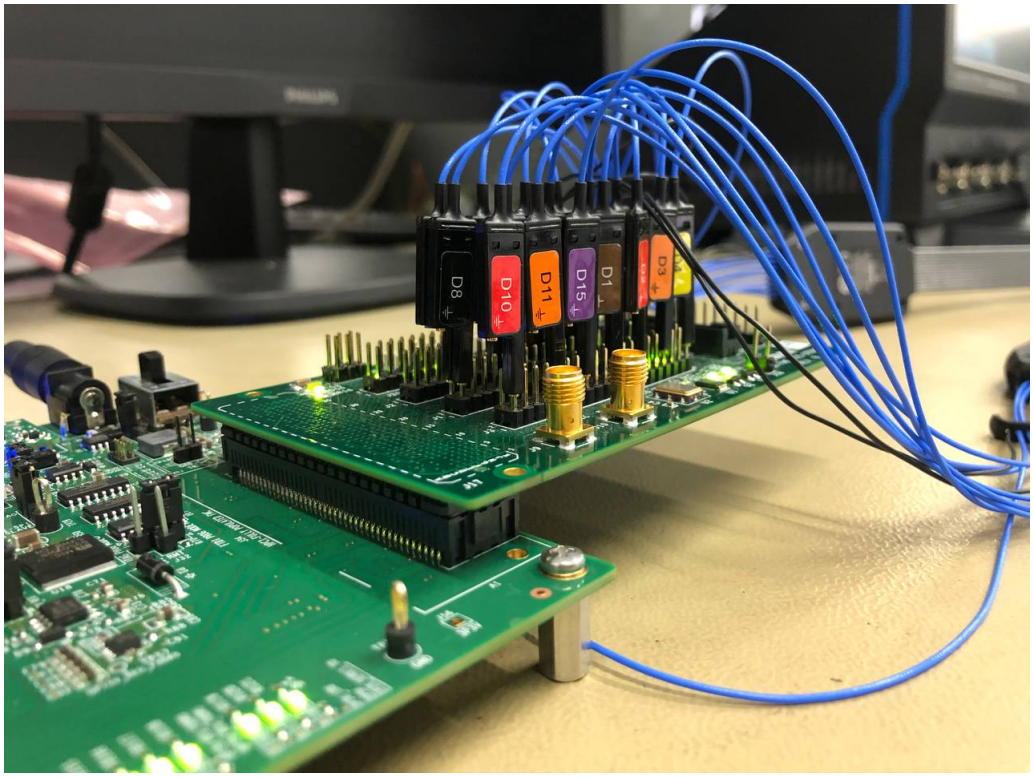

Figure 9.8: FMC XM105 Xilinx board for the test

In this case, the J1 and J3 connectors on the board are used. Each one of these is composed of 40 pins, divided in 2 rows, each of 20 pins. 16 pins of these connectors

have been linked to the oscilloscope through the digital leadset, which enables input of up-to-16 lines of digital data.
Using the digital leadset, on the oscilloscope, physical lines could be preconfigured into different logic groups, corresponding to a bus.

9.1 Hardware tests

At this point, the setup is completed and thus, the tests to check the right behaviour of the implemented structure loaded into the FPGA could be executed.
In this chapter, some tests are reported, both for the Direct Access interface and the First In First Out one, in order to check that the writing and the reading have been done correctly.

9.1.1 Test 1: writing and reading in memory without errors through Direct Access interface

The first test, which has been executed, is a writing and a reading without errors through the Direct Access interface.
In Figure 9.9, it is reported the screen of the oscilloscope, in which the behaviour of some signals related to the writing through the Direct Access interface could be observed. In particular, the signal **edge_SW1** is the signal at the output of the edge detector referred to the SW1 switch, thus, this signal indicates the impulse generated at the moment in which the front of SW1 is detected when this switch is pushed.
This impulse goes to stimulate the Direct Access signals generator, which produces the writing request, the write address and the write input data. In the figure, the writing request **DA_WE** is reported; it is raised for a clock cycle after the impulse of the edge_SW1. At this point, in order to check that the writing is executed correctly, the **AWVALID** and **BVALID** signals of the AXI interface are represented, since the AWVALID is the signal that the master sends to the slave to signalize that the address is valid and thus that the writing with the AXI protocol could start, and the BVALID is the signal that the slave sends to the master as acknowledge that the writing is finished correctly.
Therefore, since after the DA_WE assertion, the AWVALID signal and subsequently BVALID one are raised for one clock cycle, it indicates that the writing has been completed without errors, otherwise the BVALID signal would not have been raised, since the writing would have been discarded.
Moreover, from this picture, the input data that the Direct Access signal generator produces could be observed. In this test, not all the 16 bits of the data have been connected to the oscilloscope, but only the 4 LSBs have been linked, which are represented on the oscilloscope screen as **DA_Din**. In this case, the 4 physical lines of the data are grouped as a bus, in order to visualize on the screen the value in hexadecimal, instead of seeing each bit as a single line.
In this case, the data to write in memory is 1, since the Direct Access signals generate has been implemented in order to increment of one the DA_Din value every time that the user wants to write through the Direct Access interface.

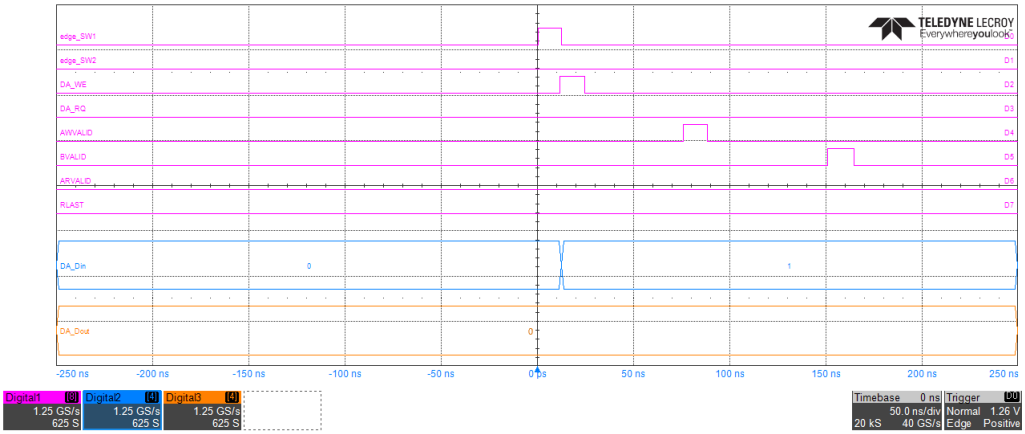

Figure 9.9: First Direct Access writing without errors

While, in Figure 9.10, the first reading without error is reported. Here, the signal **edge_SW2** is raised for one clock cycle when the user presses the SW2 switch to read a data in memory.
After that, the Direct Access signal generator produces the reading request and the memory address in which read the data. In fact, the **DA_RQ** signal is raised for 1 clock cycle after the edge_SW2 signal.
Moreover, similarly to the writing, in this case, the **ARVALID** and the **RLAST** signals are represented on the oscilloscope, in order to understand when, in the AXI interface, the reading starts (ARVALID) and finishes (RLAST).
Thus, in this case, since any error occurs, the reading is completed correctly and the RLAST is raised.
The address in which the user wants to read is the same where the data has been written previously. This reading is used even as check that effectively the previous writing has been done in a correct way.
In fact, from the below picture, it could be seen that, reading in the same address where the data has been written, the read data is 1 and it confirms the correctness both of the writing and the reading.

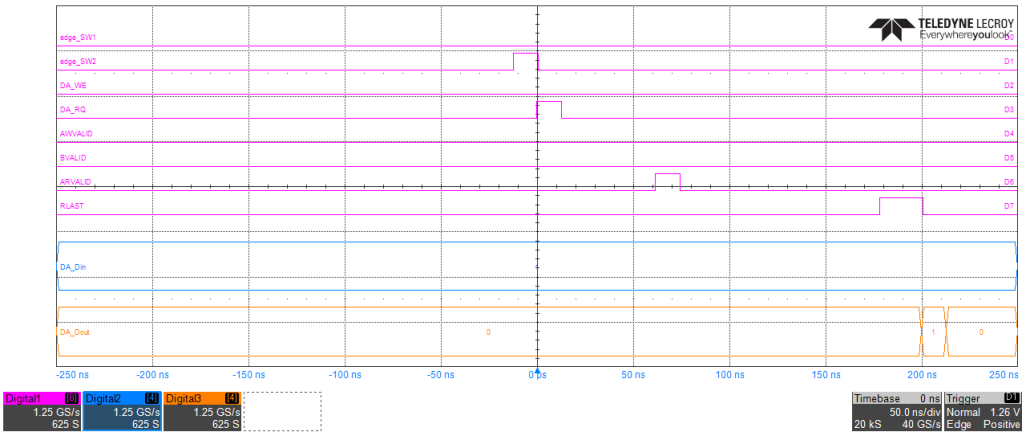

Figure 9.10: First Direct Access reading without errors

In Figures 9.11 and 9.12, respectively the second writing and reading in memory without errors through the Direct Access interface are reported.
In this case the write data is 2 and even in this case the BVALID is asserted and it means

that the writing has been completed without errors.
Then, the read data is 2, and it checks the correct execution both of the reading and the writing.
What has to be take into account in this case is that the signal RLAST is asserted, but then it is not de-asserted and it is because the slave, which is the FDDR, understands that subsequent reading through the Direct Access interface have been executed, and thus, the data that has to be read is only one, so, the first data is also the last data, therefore, there is no reason to de-assert the RLAST signal. It is an optimization in the case in which the next reading should be done though the Direct Access interface.

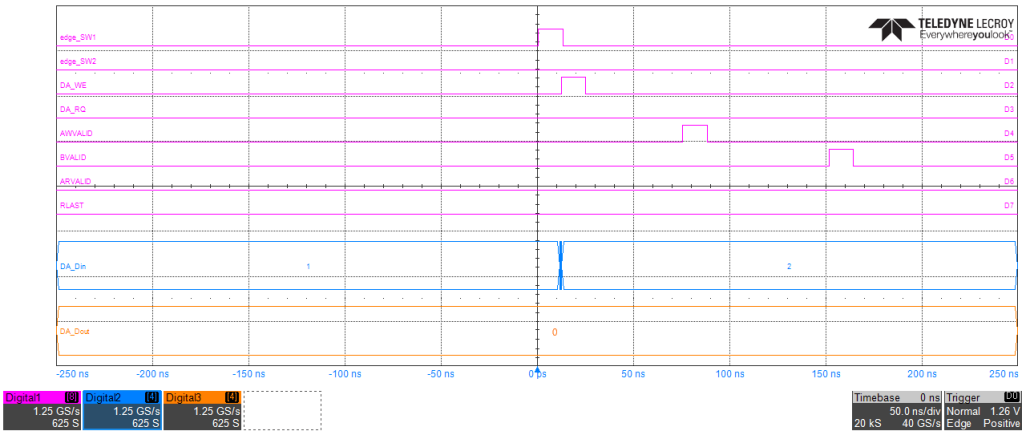

Figure 9.11: Second Direct Access writing without errors

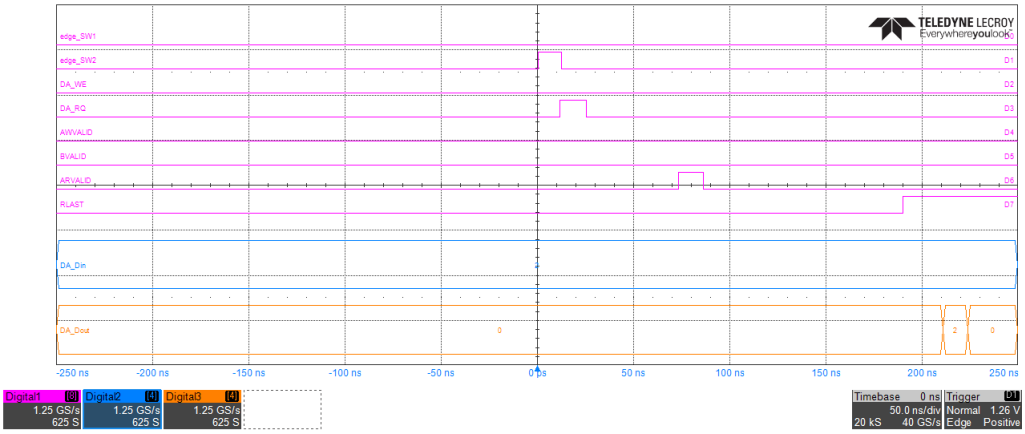

Figure 9.12: Second Direct Access reading without errors

In Figures 9.13 and 9.14, a third writing and reading are reported, in order to check that effectively the signal RLAST remains asserted if another reading through the Direct Access interface is executed. In fact, in the second figure, related to the reading, the signal RLAST is always asserted. In this way, not necessary toggles of the RLAST signal are spared, thus, even a power saving is obtained

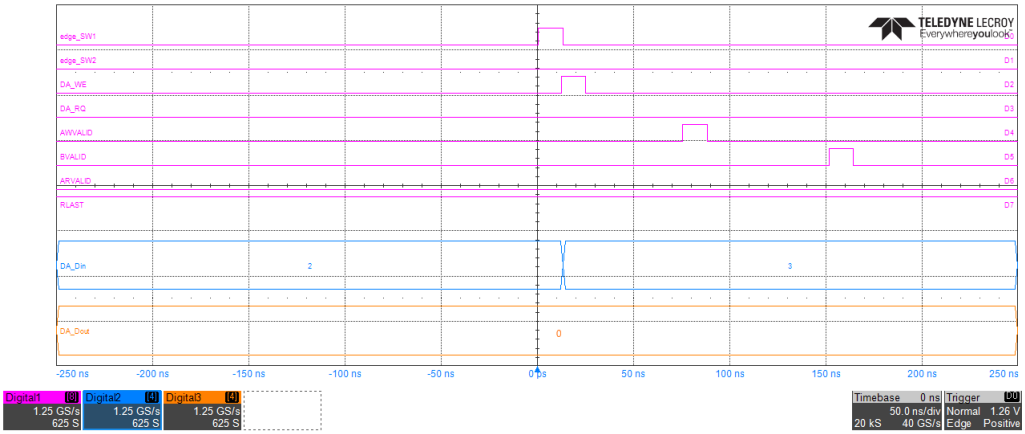

Figure 9.13: Third Direct Access writing without errors

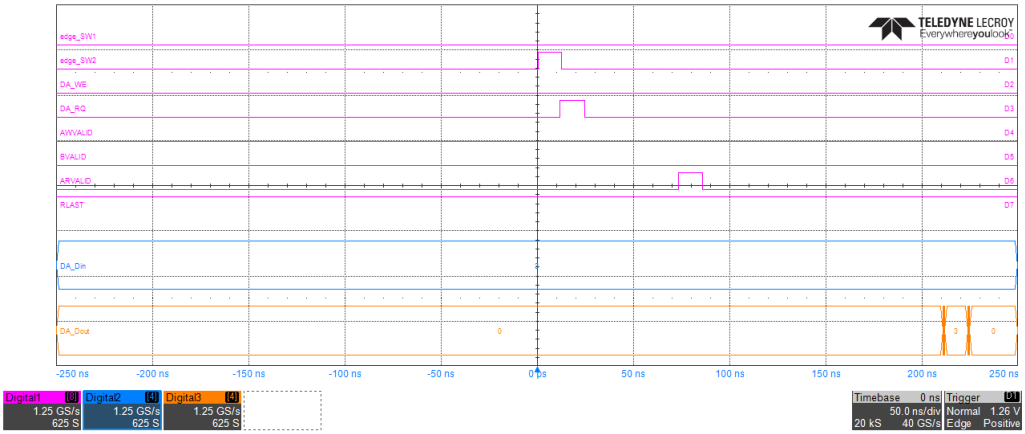

Figure 9.14: Third Direct Access reading without errors

9.2 Test 2: DA_WR_ERR2 error during writing through Direct Access interface

In this test, the right behaviour of the DA_WR_ERR2 error has been checked. In fact, in this case, the user pushes the SW1 switch, thus, the edge_SW1 is asserted for one clock cycle and then, the DA_WE is raised always for 1 clock cycle. Thus, the **DA_WRDY** signal is de-asserted at the next clock cycle, since, the machine are executing this writing and thus, it could not accept another one.
However, the issue in this test is that the user asks to write a data in an address which does not belong to the part of memory dedicated to the Direct Access, but to the part attributed to the FIFO. It is not acceptable and thus, the writing has to be discarded and the DA_WR_ERR2 has to be raised, in order to signalize to the user that an error occurs. In fact, from the below figure, it could be observed that the **DA_WR_ERR2** signal is raised and the DA_WRDY is asserted again, since this writing has been discarded, so, the machine is free to accept another writing request.

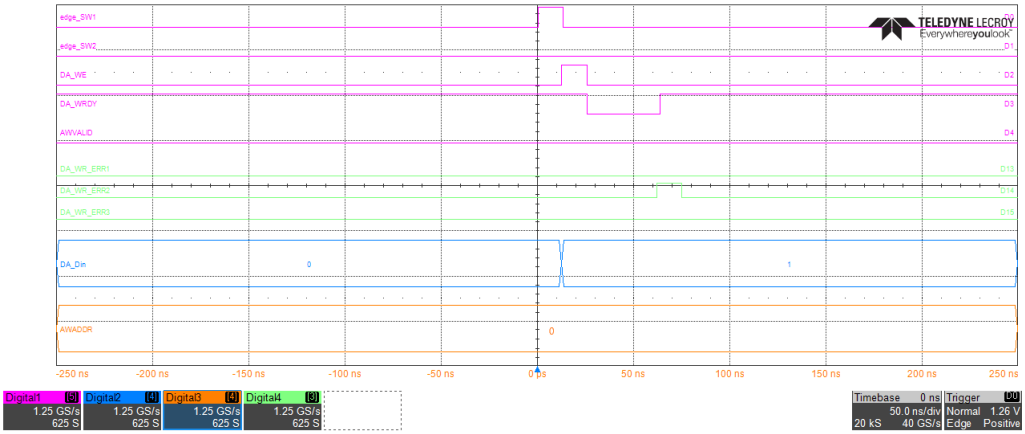

Figure 9.15: Writing with DA_WR_ERR2 error

9.3 Test 3: DA_RD_ERR2 error during reading through Direct Access interface

The test of the same error has been executed even for the reading through the direct access interface. It is reported in Figure 9.16, from which it is possible to observe the impulse of the edge_SW2 when the SW2 switch is pressed and subsequently the assertion of DA_RQ from 1 clock cycle, in order to communicate to the machine that the user wants to read a data in memory. At this point, the **DA_RRDY** signal is de-asserted in order to signalize that the machine is already busy in a reading and it could not execute another one.
The problem, even in this case, is that the user asks to read in a part of memory which is dedicated to the FIFO, thus, it is not acceptable, the reading is discarded and the **DA_RD_ERR2** is asserted in order to signalize the type of error and that the reading failed. Therefore, the DA_RRDY signal is raised again, since the machine could accept another reading.

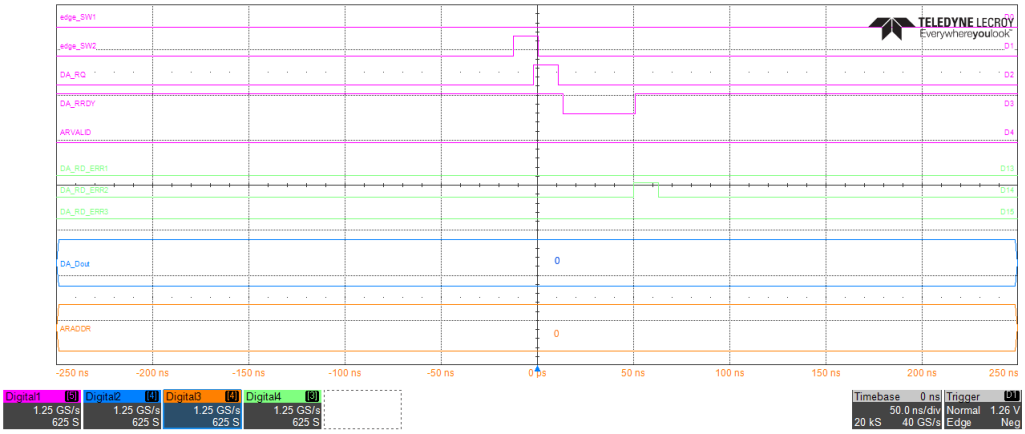

Figure 9.16: Reading with DA_WR_ERR2 error

9.4 Test 4: writing and reading in memory without errors through FIFO interface

With this fourth test, the correctness both for writing and reading in memory through the FIFO interface has been checked.
In Figure 9.17, the screen of the oscilloscope is reported. From this one, the behaviour of some signals related to the writing could be observed. In fact, the **edge_SW3** indicates that the users has pressed the SW3 in order to do a writing request through the FIFO interface, in fact, then the **FIFO_WE** is raised for 8 clock cycles, during which the FIFO input data **FIFO_Din** are sent, in this case the data are: 1, 2, 3, 4, 5, 6, 7 and 8. Even in this case, the AWVALID and BVALID signals are represented on the screen in order to check when the writing starts (AWVALID) and when and if the writing terminates.
In fact, these signals are raised and it confirms the right behaviour of the machine during the writing.

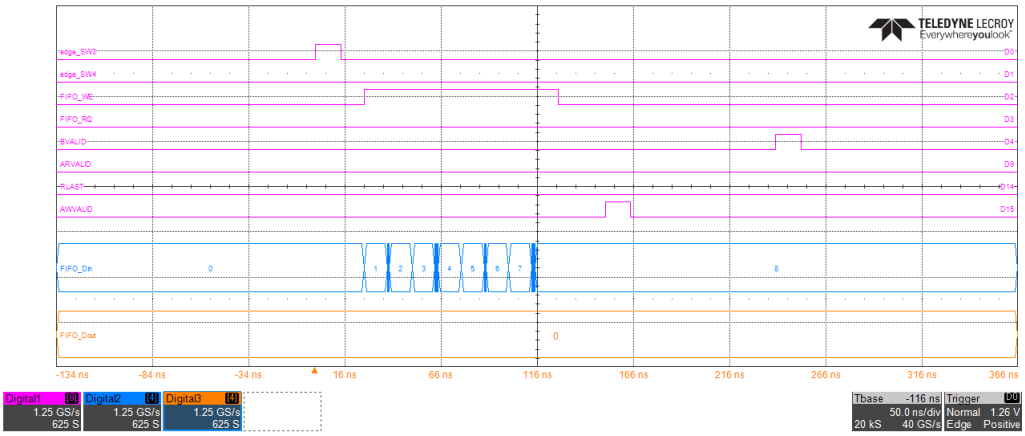

Figure 9.17: First FIFO writing without errors

Then, a reading has been executed and some signals, that concern this, are reported in Figure 9.18. In fact, since the user presses the SW4 switch, the **edge_SW4** is raised for one clock cycle and then, the reading request (**FIFO_RQ**) has been sent. As in the Direct Access case, the ARVALID and the RLAST signals are represented to understand if the reading is executed correctly. In fact, the ARVALID indicates when the address is valid, so, when the transfer through the AXI protocol could start, while the RLAST indicates when the last data coming through RDATA from the memory is read and so when the reading terminates.
Moreover, in the screen, the FIFO output data **FIFO_Dout** are also represented, after that these are extrapolated from RDATA. These data are equal to the same data that have been written in the previous writing, since the user reads in the same addresses.
This is useful also to understand if the previous writing has been executed correctly, since if it is not, in these addresses there would be no the correct data, while, since the read data are the same that have been written, it means that the writing has been done correctly.

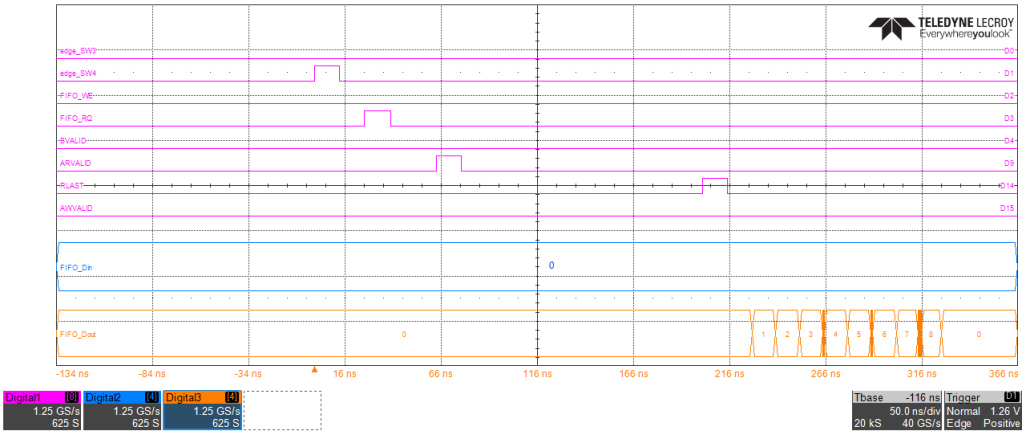

Figure 9.18: First FIFO reading without errors

In Figures 9.19 and 9.20, a second writing and reading are reported respectively, in order to check that there is no problem when another reading or writing are executed. In fact, from these images could be seen that the behaviour is the same as before. From here, it is possible to observe that effectively the bus of data is represented in hexadecimal, since the data after the 9 are not 10, 11, 12, 13, 14 and 15, but A, B, C, D, E and F.

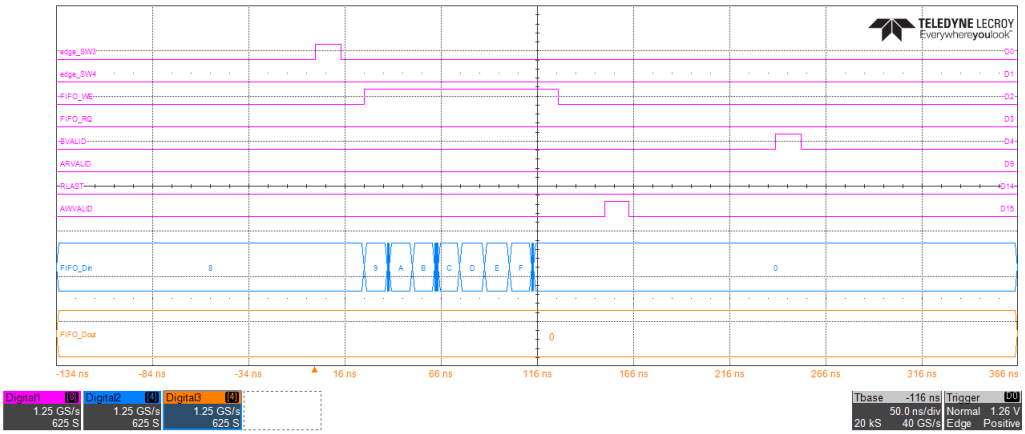

Figure 9.19: Second FIFO writing without errors

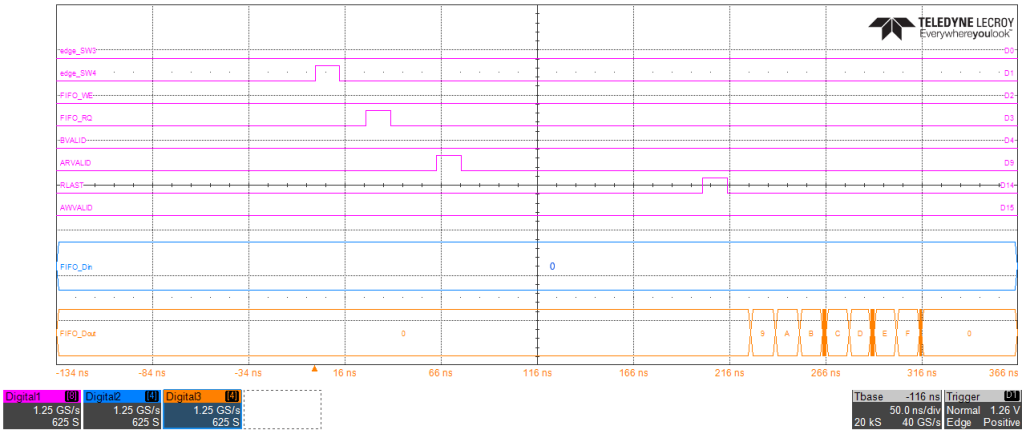

Figure 9.20: Second FIFO reading without errors

9.5 Test 5: FIFO_WR_ERR2 error during writing through FIFO interface

In this test, the correct behaviour of the **FIFO_WR_ERR2** has been checked. This error occurs when the **FIFO_WE** signal is not asserted for 8 clock cycles, but more or less than 8 and it is not acceptable based on the **ARS-REQ-455** requirement.
Thus, from the below figure could be seen that the FIFO_WE is high for 7 clock cycles instead of 8 and this leads to the assertion of the FIFO_WR_ERR2, since this condition is not acceptable.
Therefore, the writing is discarded, in fact after the raising of this error, the **FIFO_RRDY** is asserted again since the machine could accept another writing request.

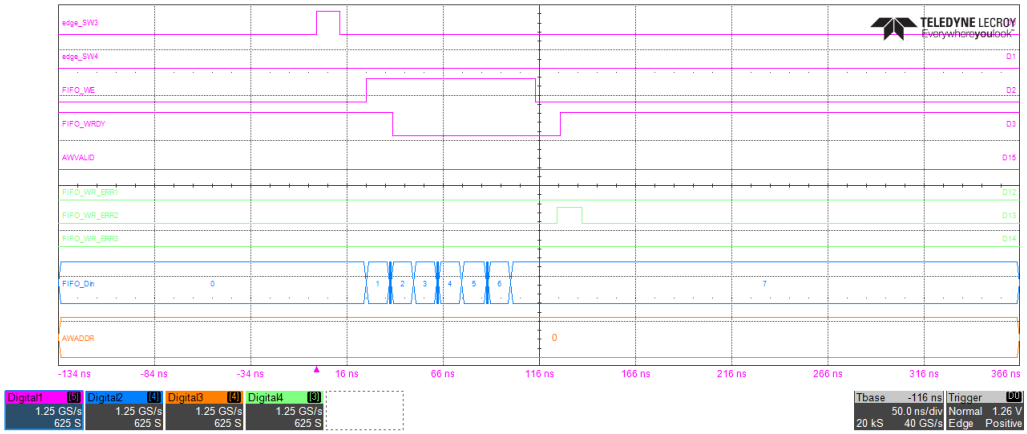

Figure 9.21: Writing with FIFO_WR_ERR2 error

9.6 Test 6: FIFO_RD_ERR2 error during reading through FIFO interface

This test is dedicated to the FIFO_WR_ERR2 error that could occur during the reading through the FIFO interface, when the request signal is raised for more than one clock cycle, since it is not acceptable, based on the **ARS-REQ-495** requirement.

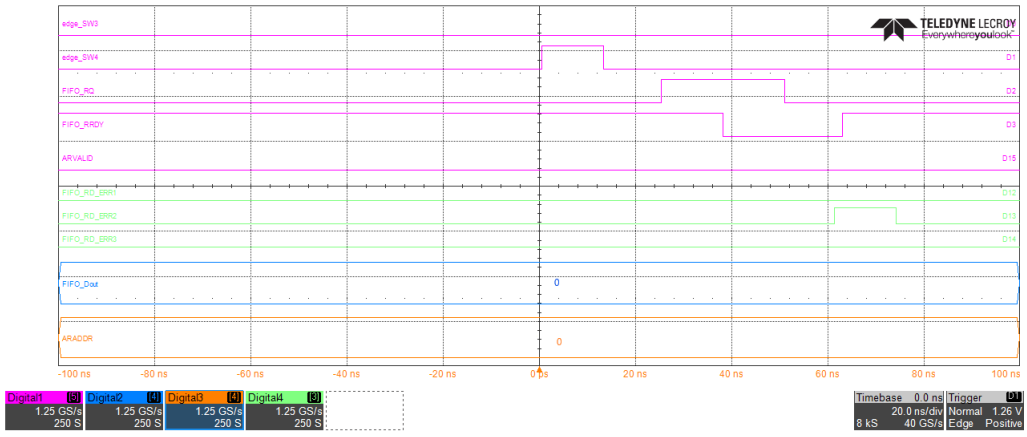

Figure 9.22: Reading with FIFO_WR_ERR2 error

In fact, in the above picture, it could be observed that the FIFO_RQ signal is asserted

for more than 1 clock cycle and consequentially the FIFO_RD_ERR2 is raised and thus, the reading is discarded and the machine becomes free to accept another reading.

9.7 Test 7: FIFO_RD_ERR1 error during reading through FIFO interface

In this case, the FIFO_RD_ERR1 error is tested. This error occurs when the FIFO is empty, but the user tries to read in memory. It is not acceptable, based on the **ARS-REQ-505** requirement and so the reading is discarded.
Moreover, the confirmation that the FIFO is empty is given by the fact that in the oscilloscope screen the **FIFO_EF** signal is always asserted and it is raised by the evaluator_FF_EF, when it understands that the FIFO is empty.

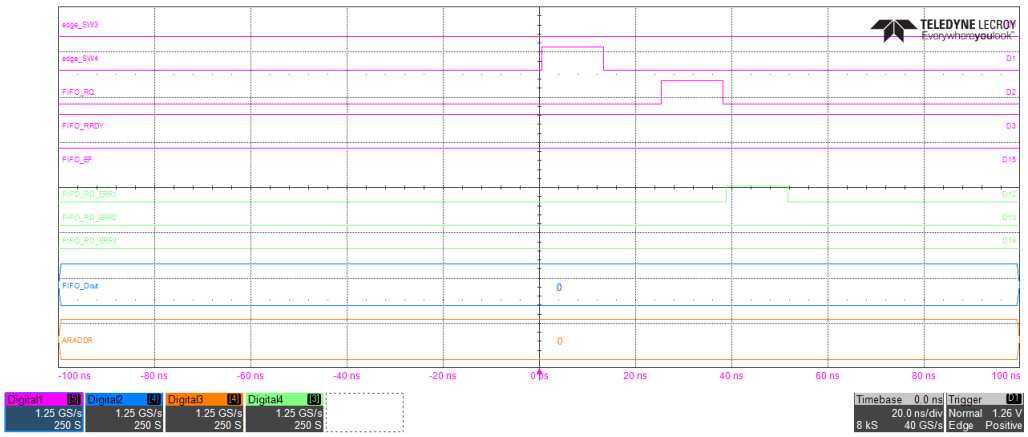

Figure 9.23: Reading with FIFO_WR_ERR1 error

Chapter 10

Conclusion and future work

The Thesis aimed to create in an RT4G150 FPGA, used in space environment, a structure able to coordinate, by means of an arbiter, concurrent writings and readings, through Direct Access and FIFO interfaces, in DDR3 memory banks.
Moreover, this block has to be able to translate the signals, which arrive from both the interfaces, in signals for the AXI interface, since this block talks with an IP of the FPGA, called FDDR, which is a DDR controller and communicates by means of an AXI interface.
The correct behaviour of the sub-blocks and of the entire structure has been tested my means of different simulations.
These are resulted satisfactory, since the implemented block writes to and reads from the DDR3 memory banks correctly and it is able to coordinate concurrent requests of writing and/or reading in a correct way, by means of the arbiter, designed taking into account the Round Robin algorithm.
Moreover, the entire structure has been also tested on hardware, using the RTG4 Development Kit with the RT4G150 FPGA and an oscilloscope, in order to check that the developed structure has a correct behaviour also on hardware and not only in simulation.

This thesis opens many paths on possible future works. A path to explore is for sure the reduction of the latency both in writing and in reading, optimizing the Finite State Machines of these blocks, in order to complete the operations with a lower number of states.
Moreover, another improvement could be done about the length of the burst data, which could be incremented, trying to find a balance between the number of data sent in a burst and the latency, which there is during a writing to or reading from the memory.
Finally, another idea for a future work could be the implementation of the arbiter based on algorithms different from the Round Robin, as the Shortest-Remaining-Time-First (SRTF) algorithm, in order to test if, with other algorithms, the performance of this structure are optimized or not.

Chapter 11

References

[1] European Space Agency. *ECSS-Q-ST-60-02C Space product assurance - ASIC and FPGA development*. ESA Requirements and Standards Division, ESTEC, P.O. Box 299, 2200 AG Noordwijk, The Netherlands, 2008.

[2] Rajinder Gill. Everything you always wanted to know about SDRAM (memory): But were afraid to ask. `https://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but\-were-afraid-to-ask`, 2010.

[3] Micron. *DDR3 SDRAM*. Micron Technology, 2006.

[4] Microsemi. RTG4 Radiation-Tolerant FPGAs. `https://www.microsemi.com/product-directory/rad-tolerant-fpgas/3576-rtg4`, 2018. Last accessed 2 April 2022.

[5] Microsemi. RT PolarFire FPGAs. `https://www.microsemi.com/product-directory/rad-tolerant-fpgas/5559-rt-polarfire-fpgas#overview`, 2019. Last accessed 22 September 2019.

[6] Microsemi. RT ProASIC3. `https://www.microsemi.com/product-directory/rad-tolerant-fpgas/5559-rt-polarfire-fpgas#overview`, 2019. Last accessed 22 September 2021.

[7] Microsemi. *UG0573 User Guide RTG4 FPGA DDR Memory Controller*. Microsemi Headquarters One Enterprise, Aliso Viejo, 2019.

[8] Microsemi. RTAX-S/SL. `https://www.microsemi.com/product-directory/rad-tolerant-fpgas/1694-rtax-s-sl`, 2020. Last accessed 9 February 2022.

[9] Microsemi. RTSX-SU. `https://www.microsemi.com/product-directory/rad-tolerant-fpgas/1697-rtsx-su`, 2020. Last accessed 9 October 2021.

[10] Microsemi. *UG0617 User Guide RTG4 FPGA Development Kit*. Microsemi Headquarters One Enterprise, Aliso Viejo, 2021.

[11] Microsemi. *Connecting User Logic to AXI Interfaces of High-Performance Communication Blocks in the SmartFusion2 Devices - Libero SoC v11.7*. Microsemi Corporation, March 2016.

[12] Martha O'Bryan. Single event effects. https://radhome.gsfc.nasa.gov/radhome/see.htm, 2021.

[13] VLSI UNIVERSE. Reset synchronizer. https://vlsiuniverse.blogspot.com/2016/09/reset-synchronizer.html, 2016.