Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Study and implementation of a SpaceWire data-handling network for reconfigurable vision-based navigation systems

Relatore:
Prof. Luca Sterpone

Candidato:
Adriano Caponio

Tutor aziendale:
Ing. Antonio Tramutola

AA. 2017-2018

Study and implementation of a SpaceWire data-handling network for reconfigurable vision-based navigation systems

Adriano Caponio

Acknowledgements

During the last six months, I had the opportunity to perform my master thesis project in cooperation with Thales Alenia Space, investigating electronic systems based on FPGA for image processing and data communication. Now that this work is completed it is the time to reflect and express my appreciation to those who supported me along the way.

Firstly, I would like to express my sincere gratitude to my advisor Prof. Luca Sterpone for allowing me have this great opportunity. Secondly, I would like to thank my industrial tutor Antonio Tramutola who has been a great supporter of all the thesis activities, always believing I could overcome any difficulties. Special thanks must go out to Daniele Rolfo, that with its expertise in FPGAs, has always been my reference point: he has forced me asking the right questions and with patience has sat nearby debugging VHDL code. Finally, I am for sure indebted to Davide Paltro, always present in the Avionic laboratory and available for any questions regarding SpaceWire packets.

Last, but certainly not least, I would like to thank my family. Mum and Dad, if it weren't for your continuous support and all the opportunities you gave me, I could have never gotten this far. For sure I couldn't have made it without my brothers, who have always supported me bearing my complaints. Really thanks! Finally, I have to thank my dear friend Aldo, my electronic classmates, my friends and everyone who has allowed me accomplishing such huge goal.

Abstract

In the present thesis, I studied and prototyped a SpaceWire data-handling network for communication inside a vision-based navigation system designed by Thales Alenia Space for a space exploration mission on Mars and in particular during the final Entry, Descending and Landing (EDL) phase on the planet.

The network is composed of several Codecs for interfacing with external world, one Router used to connect several sub-systems together and one RMAP block for performing read and write operations into a target RAM memory according to ECSS standards. All the three cores have been designed by a Japanese development team which made available open-source only their VHDL codes claiming compliance with ECSS standard.

Therefore, during this thesis some missing parts have been recostructed and their working behaviour has been understood by means extensive simulations. In the end, the network has been implemented on hardware using a space qualified FPGA platform and functionally tested in a Hardware-in-the-loop system employing a Leon-3 processor. In the next future, this SpaceWire network can thus be used to exchange data and controls between feature extractor and matcher IP cores, camera and the on-board computer after additional testing on the implemented prototype.

Contents

List of Figures 7

List of Tables 9

1 Introduction 11
1.1 Vision-based navigation systems 12

2 SpaceWire Standard 15
2.1 Overview 15
2.1.1 Physical level 16
2.1.2 Signal level 17
2.1.3 Character level 19
2.1.4 Packet level 20
2.1.5 Network level 21
2.2 Remote memory access protocol (RMAP) 24
2.2.1 RMAP commands and fields 25
2.2.2 Cyclic Redundancy Code 28

3 Field Programmable Gate Arrays 31
3.1 FPGA technology 31
3.2 FPGA design flow 36
3.3 Space-grade FPGAs: Xilinx Virtex devices 37
3.4 GR-CPCI-XC4V board 39

4 SpaceWire network architecture 41
4.1 General structure 41
4.2 SpaceWire Codec IP core 42
4.2.1 Core architecture 43
4.2.2 Link state machine 45
4.3 SpaceWire Router IP core 48
4.3.1 Core architecture 48

4.3.2 CRC and Routing table generation . . . 51
4.4 SpaceWire RMAP IP core . . . 53
4.4.1 Core architecture . . . 54
4.4.2 Working behaviour dataflow . . . 56
4.4.3 FIFOs generation . . . 58
4.5 Target RAM memory . . . 59

5 Cores Simulations 61
5.1 Codec simulation . . . 61
5.2 Router simulation . . . 65
5.2.1 Routing table configuration . . . 66
5.2.2 Path addressing mode . . . 69
5.2.3 Logical addressing mode . . . 71
5.3 RMAP simulation . . . 71

6 Experimental results 75
6.1 Hardware implementation . . . 75
6.1.1 Pin assignment . . . 76
6.1.2 Clock management . . . 78
6.1.3 FPGA synthesis and implementation . . . 79
6.1.4 Timing constraints . . . 81
6.2 Hardware testing . . . 82
6.2.1 Testing setup . . . 82
6.2.2 Test procedure and results . . . 83
6.3 Summary . . . 87

7 Conclusions and future work 89

List of acronyms 91

Bibliography 93

List of Figures

1.1 System architecture of VisNav EDL project (by courtesy of Thales). 12

2.1 Spacecraft architecture based on SpaceWire network. 16
2.2 SpaceWire connector and wire assembly. 17
2.3 LVDS signal levels. 17
2.4 LVDS driver and receiver configurations. 18
2.5 Example of Data and Strobe encoding. 18
2.6 Data and control characters and control codes. 20
2.7 Structure of a SpaceWire packet. 21
2.8 An example of network. 22
2.9 Path and logical addressing in SpaceWire network. 23
2.10 Write command packet and bits of the instruction field. 26
2.11 Write command reply packet and bits of the instruction field. 27
2.12 Write command and reply sequence. 27
2.13 Read command packet and bits of the instruction field. 28
2.14 Linear Feedback Shift Register for CRC computation [15]. 29

3.1 Picture of Xilinx Virtex-4 FPGA [1]. 31
3.2 General architecture of a modern FPGA. 32
3.3 Basic structure of an FPGA Configurable Logic Block. 33
3.4 Structure of a switch matrix for a pass-transistor-based FPGAs. . . 34
3.5 FPGA programming through JTAG scan chain 35
3.6 Typical FPGA design flow. 36
3.7 Roadmap of the Xilinx space-grade Virtex families FPGAs. 37
3.8 Internal architecture of Virtex-4 CLB. 38
3.9 GR-CPCI-XC4V board block diagram. 39

4.1 High level architecture of SpaceWire network 41
4.2 SpaceWire Codec IP core block diagram 43
4.3 Timing diagram of writing into Transmit FIFO. 44
4.4 Synchronization mechanism inside Codec receiver. 45

4.5 SpaceWire Interface State machine. 46
4.6 SpaceWire State machine in Auto-start mode. 47
4.7 SpaceWire Router IP core block diagram 48
4.8 Structure of the Routing table 50
4.9 Use of Block memory generator to generate RAM/ROM. 51
4.10 Generation of *crcRomXilinx* single-port ROM memory. 52
4.11 Generation of *RamXilinx32x256* single-port RAM memory. 53
4.12 SpaceWire RMAP IP core block diagram 54
4.13 Dataflow representation of RMAP IP core working behaviour. . . . 57
4.14 Generation of *FIFO8x2KXilinx* standard FIFO. 58
4.15 Graph of Target RAM memory Finite State Machine. 59

5.1 Architecture of first Codec simulation testbench. 61
5.2 First SpaceWire Codec IP core simulation 62
5.3 Second SpaceWire Codec IP core simulation 63
5.4 Architecture of second Codec simulation testbench. 64
5.5 Architecture of Router testbench. 65
5.6 Snapshot of the RMAP Decoder present inside Router Port 0. . . . 67
5.7 SpaceWire Router core simulation and routing table configuration. . 68
5.8 SpaceWire Router simulation for path addressing mode 69
5.9 SpaceWire Router core simulation in logical addressing mode. . . . 70
5.10 Architecture of RMAP simulation testbench. 71
5.11 SpaceWire RMAP core plus target memory simulation. 72

6.1 Picture of the Gaisler GR-CPCI-XC4V board employed. 75
6.2 Picture of the SpaceWire mezzanine board *GR-SER2-SPW4*. 76
6.3 Electrical schematic of mezzanine board with SpW and GENIO signals 77
6.4 FPGA Clock different frequencies generation using a DCM. 78
6.5 Generation of DCM called *Clockbuffer*. 79
6.6 Device utilization estimation after synthesis phase. 80
6.7 Device utilization summary after Mapping and Place&Route phase. 80
6.8 Test Hardware setup (by courtesy of Thales Alenia Space). 83
6.9 Link analyzer when a SpaceWire connection is established 84
6.10 Packets transmitted during link initialization. 85
6.11 Screenshot of the test program final results. 87

List of Tables

2.1 An example of routing table in logical addressing. 24

4.1 Maximum data length handled by RMAP IP core. 55

5.1 RMAP packet fields for routing table dynamic configuration. 66

6.1 Timing settings for complete FPGA project. 81
6.2 Summary of the obtained results. 87

Chapter 1

Introduction

Vision-based navigation is a technology widely used to support space exploration missions especially in presence of an harsh environment on the planet to reach. Some of its scenarios are: fly-bys, interplanetary cruise, rendezvous and docking, entry, descent, landing and planetary surface mobility. Thales Alenia Space have a long history of study and development of space exploration mission: one of this projects, which this thesis project is part of, is the *"VISion based NAVigation system"* or briefly called VISNAV previously designed for the Moon exploration and then upgraded for the Mars scenario.

The objectives of this European Space Agency-funded program is the development and validation of building blocks of a Vision Based Navigation system for the Mars Entry, Descent and Landing (EDL) phases. As in all missions, data exchange between modules plays a crucial role. VisNav system communication is based on the SpaceWire protocol which allows to achieve high data-rate links between the different building blocks composing the systems. The SpW protocol is a standard developed by ESA, fully detailed in several European Cooperation for Space Standardization (ECSS) documents and implemented in dedicated IP cores under ESA and Industry property right.

This thesis aims at studying and implementing a SpaceWire network starting from partial open-source cores found on the web that could be inserted in this context. Each core composing the network has been reconstructed and simulated in order to assess the compliance with ECSS standards. Finally the whole design has been implemented on a space qualified FPGA platform and tested successfully to verify that the intended functionalities are deployed.

Such open-source SpaceWire network can also replace custom-tailored IP cores providing more flexibility to the system and also a significant costs reduction.

1.1 Vision-based navigation systems

Before going on with the description of the SpaceWire standard, it is worth briefly talking about the system architecture of VISNAV project. Figure 1.1 shows the designed configuration for EDL scenario.

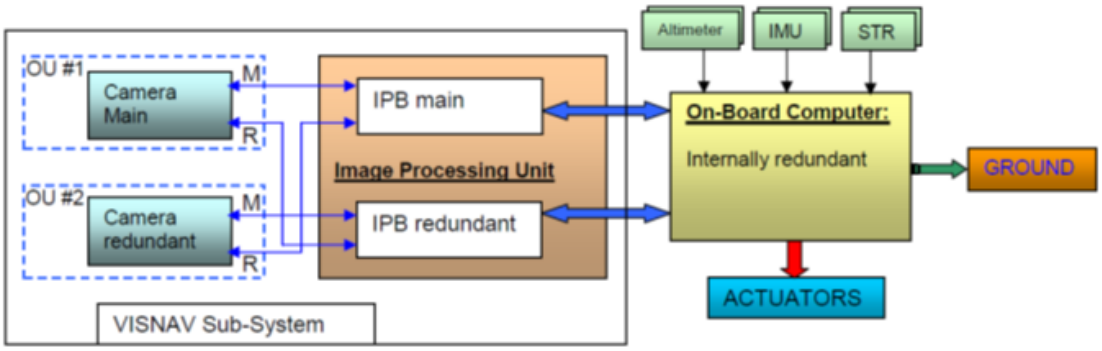


Figure 1.1: System architecture of VisNav EDL project (by courtesy of Thales).

Building blocks composing the systems are:

- a Camera, part of the Optical Unit (OU), interfacing with an Optical/Electrical stimulator for images acquisition;
- the Image Processing Board (IPB) component included in the Image Processing Unit. It hosts an FPGA which implements the FEIC (an integrated circuit for features extraction and matching designed by University of Dundee) and a processor for running the software part of the image processing algorithms;
- the On-Board Computer (OBC),based on a Leon-2 processor, where GNC (Guidance, Navigation and Control) algorithms are hosted. Moreover, it interfaces with different sensors (Altimeter, IMU...), communicates with the Ground segment and drives the different actuators.

As it is visible from figure 1.1, every component has its redundant conterpart in order to achieve an hardware-implemented single failure tolerance. VisNav system communication is based on a SpaceWire router implemented in the FPGA present on the IPB. The images taken from the camera are passed to the IPB via SpW link; inside this last component, the FEIC extracts the feature list and pass it to the On-Board Computer through the router which then performs hazard detection and GNC operations. Moreover the OBC, always through the router, is able to configure the FEIC and read its status via SpaceWire RMAP protocol.

All the SpW communication sub-system has been dealt in this thesis. In particular:

Chapter 2 provides some background informations which are relevant to this thesis work. Main features of the SpaceWire standard and the RMAP protocol are presented from a theoretical point of view using the ECSS as a reference source.

Chapter 3 deals with the SpaceWire network giving a detail description of the internal architecture and working behaviour of each single IP core which can be found inside it; moreover the way how some internal blocks have been generated, using automatic VHDL tools, will be presented.

Chapter 4 describes all the ModelSim simulations performed to assest the ECSS compliance and the FPGA development flow followed to get an hardware implementation.

Chapter 5 gives details about the board and method used to prototype the design with the test procedures employed to verify the hardware functionalities were the ones desired.

Finally chapter 6 addresses the conclusions presenting the implementated design and possible future follow-up.

Chapter 2

SpaceWire Standard

In this chapter a general overview of SpaceWire standard principles will be given as background information for this thesis work. Details will be given spanning all the ISO/OSI network model from the physical electrical level up to the high level packet transmission covering also most important RMAP features.

2.1 Overview

One of the most important needs inside a satellite spacecraft is having a bus system to allow internal subsystems and electronic components to communicate exchanging informations. Due to the harsh space environment and high computational needs this communication system should also provide several features, namely being fast, fault-tolerant and reusable among different space missions. In order to meet this requirements a spacecraft communication protocol called SpaceWire was developed by ESA (European Space Agency) in collaboration with international space agencies including NASA, JAXA and Roscosmos. It has been standardized inside ECSS-E50-12A document with the important contribution of University of Dundee in overcoming the IEEE 1355 standard which SpaceWire is based on.

SpaceWire provides a unified high speed data-handling infrastructure for connecting together sensors, processing elements, mass-memory units, downlink telemetry subsystems and EGSE equipment. The standard defines a full-duplex, point-to-point, serial data communication link capable of data rates between 2 Mbps and 400 Mbp [13]. Below in Fig.2.1 is shown an example of a SpaceWire based architecture inside a satellite data-handling section. SpaceWire has been developed to allow the compatibility between different space equipment and subsequent reuse into different scenarios increasing the flexibility within limited budget space missions. As can been seen below kernel of network system is given by packet switching

wormhole routing switches which can be stand-alone components or can be integrated into the memory or other modules and could possibly be bypassed using target logical addressing.

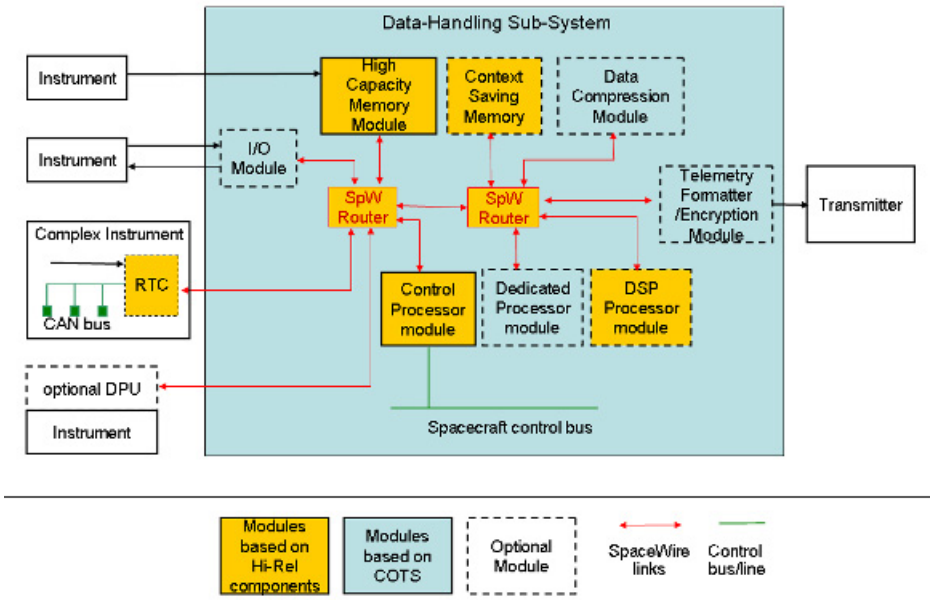


Figure 2.1: Spacecraft architecture based on SpaceWire network.

The scope of the Standard is the description of the working behaviour of physical connectors and cables, focusing on electrical properties and logical protocols that comprise the SpaceWire data link. Moreover, SpaceWire provides a means of sending packets of information from a source node to a specified destination node covering also error detection and recovery throught specific code bits set inside the packet; however standard does not specify packet contents.

2.1.1 Physical level

The physical level of the SpW protocol covers cables, connectors and printed circuit board tracks. Moreover it describes the actual interface between nodes including both the mechanical and electrical interfaces [13]. Each SpaceWire cable is made up by four twisted pair wires namely DataIn, StrobeIn, DataOut and StrobeOut that can run in both directions and working in a differential mode: therefore eight wires are present inside a cable. In addition, the ground wire runs in the middle of the connector as shown in Fig.2.2.

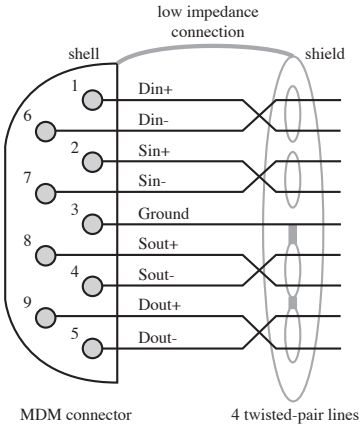


Figure 2.2: SpaceWire connector and wire assembly.

Each single twisted pair wire is coated with a polymer jacket while the whole cable is silver-shielded from the outside to reduce interference and for further protection. The ECSS standard also sets the maximum length of a SpaceWire cable to 10 meters, to keep the disturbances on the link at acceptable safety margins as well as defines the connector as a nine contact micro-miniature D-type.

2.1.2 Signal level

The signal level part of the SpaceWire standard covers signal voltage levels, noise margins and signal encoding [22]. In particular the technique adopted by the protocol is the *Low Voltage Differential Signaling (LVDS)* which allow to achieve very high-speed connections with a low voltage swing (generally 350 mV).

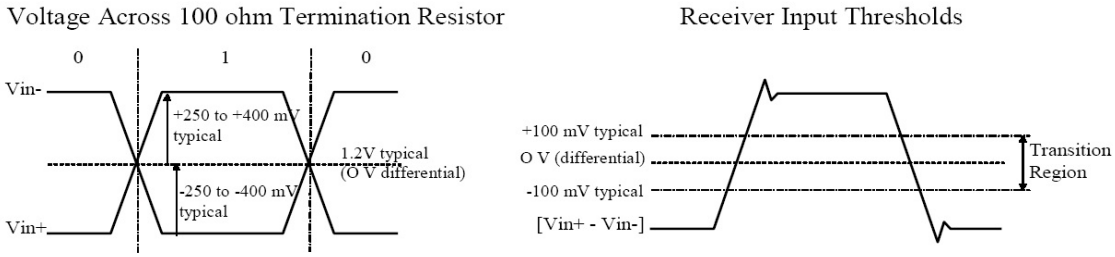


Figure 2.3: LVDS signal levels.

The adoption of this differential technique allows to reduce noise margins while using low voltages levels in information transmission ensuring furthermore relatively

low power consumption at high speed rates. Levels used in LVDS are shown in Fig.2.3.

Generally, a typical LVDS driver has on its top a constant current source of 3.5 mA which flows out of it into the transmission medium (a wire or a PCB trace) through a 100 Ω termination impedance and then back to the driver via the transmission medium. Two pairs of transistors in a differential configuration control the direction of the current flowing through the termination resistor as shown in Fig.2.4 (taken from [22]). For all this features, LVDS technique provides nearly constant drive current (+3.5 mA for logic 1 and -3.5 mA for logic 0) which decreases induced noise on power supplies and provides high immunity to interference due to its differential nature and reduced power consumption (50 mW) with respect to other techniques (120 mW for ECL/PECL).

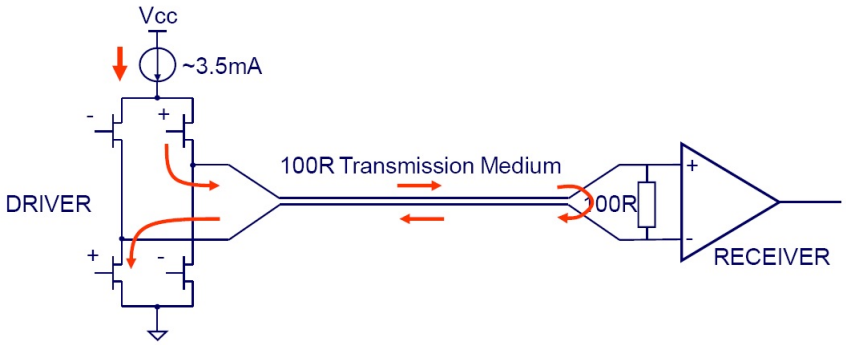


Figure 2.4: LVDS driver and receiver configurations.

SpaceWire adopts just two signals for information encoding: Data and Strobe. Both of them make use of LVDS.

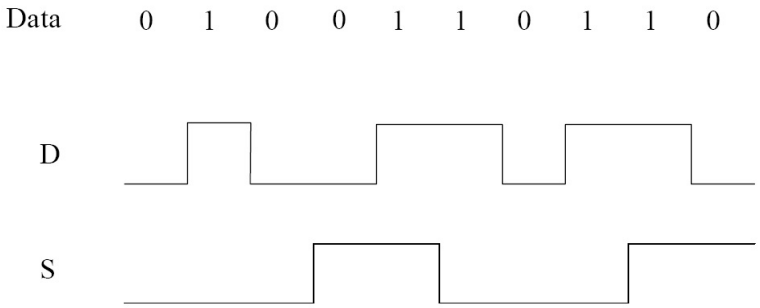


Figure 2.5: Example of Data and Strobe encoding.

While Data signal should follow normal data bit stream, the Strobe one instead

changes state whenever the Data does not change from one bit to the next. An example of Data-Strobe (DS) encoding is shown in Fig.2.5.
This is a coding scheme which encodes in itself the transmission clock along with the transmitted data: in fact, the clock can be recovered by simply using an XOR gate with the Data and Strobe lines as inputs. The reason for using this Data-Strobe encoding is to improve the skew tolerance to almost 1-bit time, compared to the 0.5-bit time for normal used encoding [22].

A SpaceWire link comprises two pairs of differential signals, one pair transmitting the Data and Strobe signals in one direction and the other pair transmitting Data and Strobe in the opposite direction.

2.1.3 Character level

SpaceWire protocol makes use of two different kinds of characters: data and control characters. In details:

- Data characters are referred to normal data generally sent in byte packets with the least-significant bit sent first. Standards defines that data characters contain also a parity bit and a data-control flag. The parity-bit covers the previous eight-bits of data and it is set to produce odd parity so that the total number of 1's in the field covered is an odd number. The data-control flag is normally set to zero indicating that the transmitted character is of data type being instead equal to one in opposite case.

- Control characters on the other hand, are not referred to data but to special informations. They are formed from a parity bit, a data-control flag and a two-bit control code; in this case as previously specified data-control flag set to one. The two-bit control code defines four possible control characters namely a flow control token (FCT), end of packet (EOP), error end of packet (EEP), and escape (ESC). While the latter may be used to form longer control codes, the flow control token (FCT) has the function of alerting the receiver that another 8-bit packet may arrive so that buffer overflow never occurs. The EOP and EEP are instead end of packet markers signaling if either an error or no error has occurred.

Data and control characters are shown in Fig.2.6. In addition to these characters, standard defines also two control codes: NULL and time-codes.

- NULL is a code sent when there is no Data character or Control character to be transferred across medium. It is formed by an ESC followed by the flow control token (FCT). NULL has the main function to keep the SpaceWire link between two nodes still active or to detect any link disconnection.

- The Time-Code is instead used to support the distribution of system time across a network. A Time-Code is formed by an ESC code followed by a single data-character.

In all cases the parity bit is in charge of detection of an error occurred during transmission. It generally covers the previous eight bits of a data character or two bits of the control character plus the current parity bit and the current data-control flag. As previously stated this bit is set to define an odd parity.

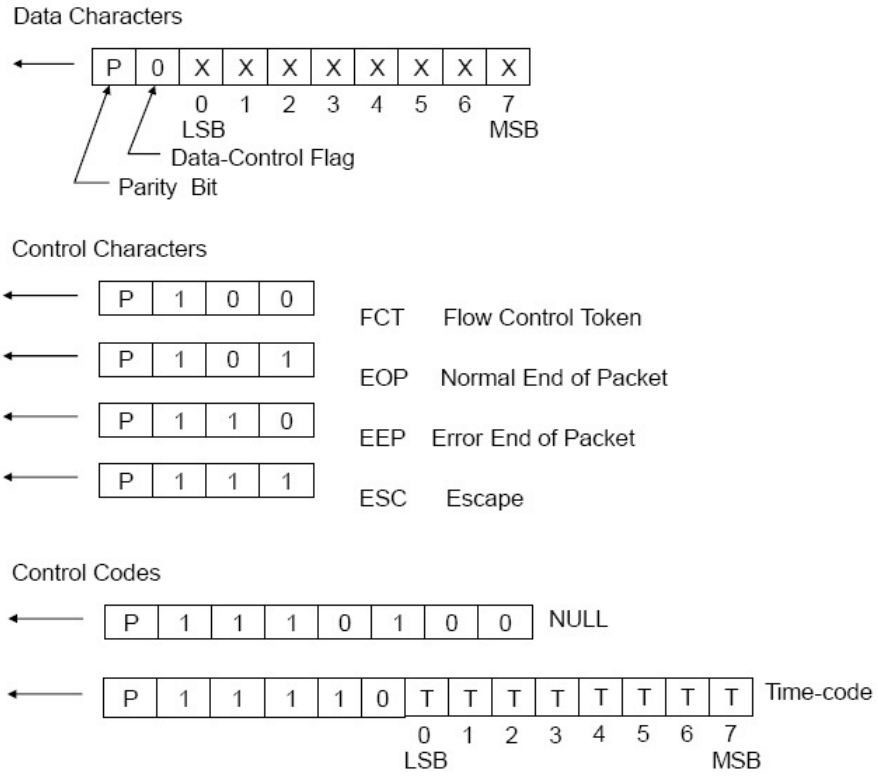


Figure 2.6: Data and control characters and control codes.

2.1.4 Packet level

The packet level of the protocol defines the simply defines the content divided in fields of a SpaceWire packet: destination address, cargo and end of packet marker, as illustrated in Figure 2.7.

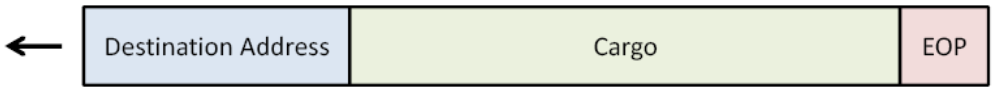


Figure 2.7: Structure of a SpaceWire packet.

The first part of the packet is the destination address of the node to which the information has to be sent. This field is crucial inside the standard since it represents either the identity of the destination node or the path that the packet has to take through a SpaceWire network to reach to the destination node. In the case of a point-to-point link directly between two nodes (so there is no routers in between) the destination address is not necessary [22].

The content of information to be transferred from source to destination is located inside the cargo field. SpaceWire standard does not specifies either the packet content or a limit length so in principle any number of data bytes can be transferred inside a single packet.

Finally a SpW packet ends with an EOP which delimits where a packet ends up and its subsequent begins. Sometimes an EEP marker may take place of normal EOP to indicate that the packet has been terminated prematurely because of an error that occurred while the packet traversed a SpaceWire network. In this case cargo is incomplete or contains corrupted information which will be later discarded as soon as receiver decodes the end of packet marker. In this case the error recovery procedure is dealt by a specific Finite State Machine (FSM) defined inside the ECSS standard.

2.1.5 Network level

As previously mentioned, SpaceWire protocol has been introduced in order to connect into a wider network different nodes like memories or sensors by means of SpaceWire links and routers so that they can exchange information and work together to perform some required function. An example of such network is shown in Fig.2.8.

Links provide the means for passing packets from one node to another with the assumption that each single node can support only a limited number of links (e.g. up to six links). Routing switches, also called wormhole routers, can instead connect together many nodes and route packets from one node to another placed in

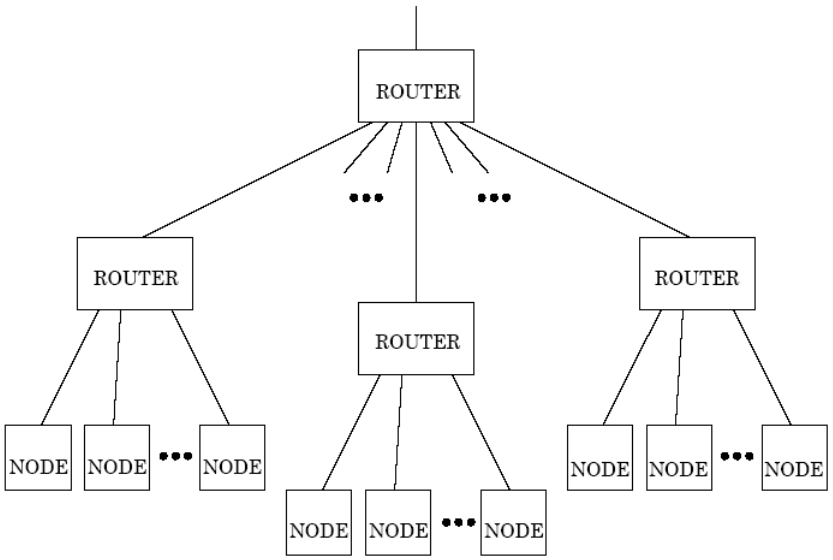


Figure 2.8: An example of network.

another area of the network.

There are two possible kinds of routers: static and dynamic. A static routing switch establishes connections between nodes and does not change them with respect to time. On the contrary, dynamic switches change the routing frequently, usually on a packet by packet basis, and are consequently also known as packet routing switches. SpaceWire routers are generally of the second kind.

Data are split into packet units to make easier their transmission across the network. Their structure has been described previously. Moreover, flow control (FCT) is employed to manage the movement of packets across a link connecting a node or a router to another node or router. In fact, a node or a router accepts an incoming data stream only if the receiving buffer for that data is available or empty otherwise the receiver stops the transmitting node from sending any more data.
The destination address at the beginning of packet is used to route the packet through a network from the source node to the destination one. There are two forms of addressing methods which can be used: path addressing or logical addressing. Both of them can be easily explained with the analogy (taken from [22]) of giving directions to car driver as shown in Fig. 2.9.

The sequence of directions to provide the driver defines the path from the source node to the destination one. In case of *path addressing*, this direction is simply

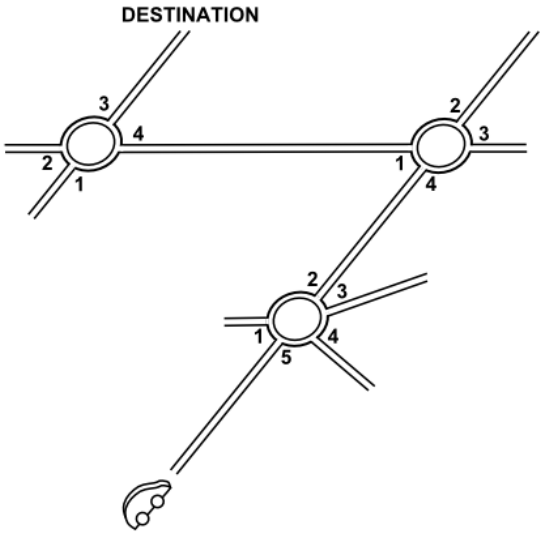


Figure 2.9: Path and logical addressing in SpaceWire network.

a data character that specifies which port of the router the packet should be forwarded through. Since each router can have a maximum of 31 external ports (plus one internal needed for configuration) the leading data character of each SpW packet is a number in this range.

In a SpaceWire network using instead *logical addressing*, each destination is given an identifier, which is a number in the range 32 to 255. This number represents a logical address. Each routing switch in the network has an internal table called routing table (like the sign at a roundabout) which specifies a matching between what port the packet should be forwarded through and each possible destination logical address. The leading data character of each packet is then set to the required destination identifier and the packet is forwarded inside the network. While in path addressing the leading data character is always discarded after forwarding, in logical addressing instead it is not, since it will be needed to look up the path to follow at the next router encountered.

An example of a routing table is shown in Tab.2.1. In this example, when a packet is received with a logical address of 20 as packet header it is forwarded to output port 1 of the router. A packet with logical address of either 3 or 15 is routed to output port 7 and a packet with logical address 31 is sent instead to port 4.

Routing Table	
Logical destination	Physical output port
20	1
3	7
31	4
15	7
...	...

Table 2.1: An example of routing table in logical addressing.

As can be understood for a medium and large network the routing table can become quickly large involving higher complexity in single routers and higher memory spaces needed. That's because when using logical addressing the complexity of packet addressing is handled by the routing switches rather than by the source node, as it is instead the case when using path addressing method.

Finally, as previously mentioned, SpaceWire routers employ generally wormhole routing. When a packet starts to arrive at an input port of a router, its destination address is looked at immediately. If the requested output port is free, then the packet is routed immediately to that port marking it as busy until an end of packet marker is identified. The packet then flows through the router as soon as it is received at the input port [13].

In case a requested output port is busy then the input port stops the incoming packet until it becomes free by not transmitting the flow control tokens (FCT). In this way the link connecting the source node to the routing switch is then blocked until the port returns free to transmit the new packet.

2.2 Remote memory access protocol (RMAP)

Together with SpaceWire, another communication protocol called RMAP (Remote memory access protocol) was proposed and standardized by ESA inside ECSS-E-ST-50-52C. RMAP can be used to configure a SpaceWire network, control SpaceWire nodes, and to transfer data to and from SpaceWire nodes [14].

In particular RMAP proves to be useful when dealing with memories or register files to perform writing operations in order to configure SpaceWire nodes internal registers (for example writing switches routing table) therefore creating or changing the desired network configuration. Similarly, RMAP also allows to perform

reading operations from embedded memories or FIFOs to collect status informations and sharing data between different network nodes.

2.2.1 RMAP commands and fields

An RMAP transaction is generally composed of two packets: a Command one (Write/Read/Modify) and an optional Reply packet from the target memory node. The possible commands defined inside the standard are:

- **Write command** allow one node inside the network, defined as initiator, to write one or more bytes of data inside one or several memory locations of another node defined as target, provided that this write operation is allowed. Write commands can be acknowledged or not by the target when they have been received correctly. If the write command is acknowledged and there is an error with the write command, the target replies with an error/status code to the initiator (or other node) that sent the command [14];

- **Read command** allow one node inside the network, defined as initiator node, to read one or more bytes of data present inside one or several memory locations of another node defined as target, provided that this read operation is allowed. Data been read is returned back to the initiator node by means of a reply packet inside cargo field as later described;

- **Read-modify-write command** is used instead to allow one node inside the network, defined as the initiator, to read the memory location of another node, the target one, modifying the value read and then writing a new value back to the same memory location. Only the value originally written inside the memory is returned back to the initiator by means of a reply packet.

Each single packet is codified using different fields, as can be seen for instance in Fig. 2.10 in case of a write command, in order to define which kind of command is willing to be used, the packet route inside the network, its content etc...
Picture also shows the content of the *Instruction field* which is in charge of defining if the packet to be sent contains a command or not (in this case packet type is 0b01), its nature (Write/read), if the reply packet is requested, its length and other kind of information.

Since the RMAP can be seen as an upper layer of the SpaceWire protocol, an RMAP packet is compliant with SpW standard. It begins with target node address in order to be routed inside the network and its structure resembles the one

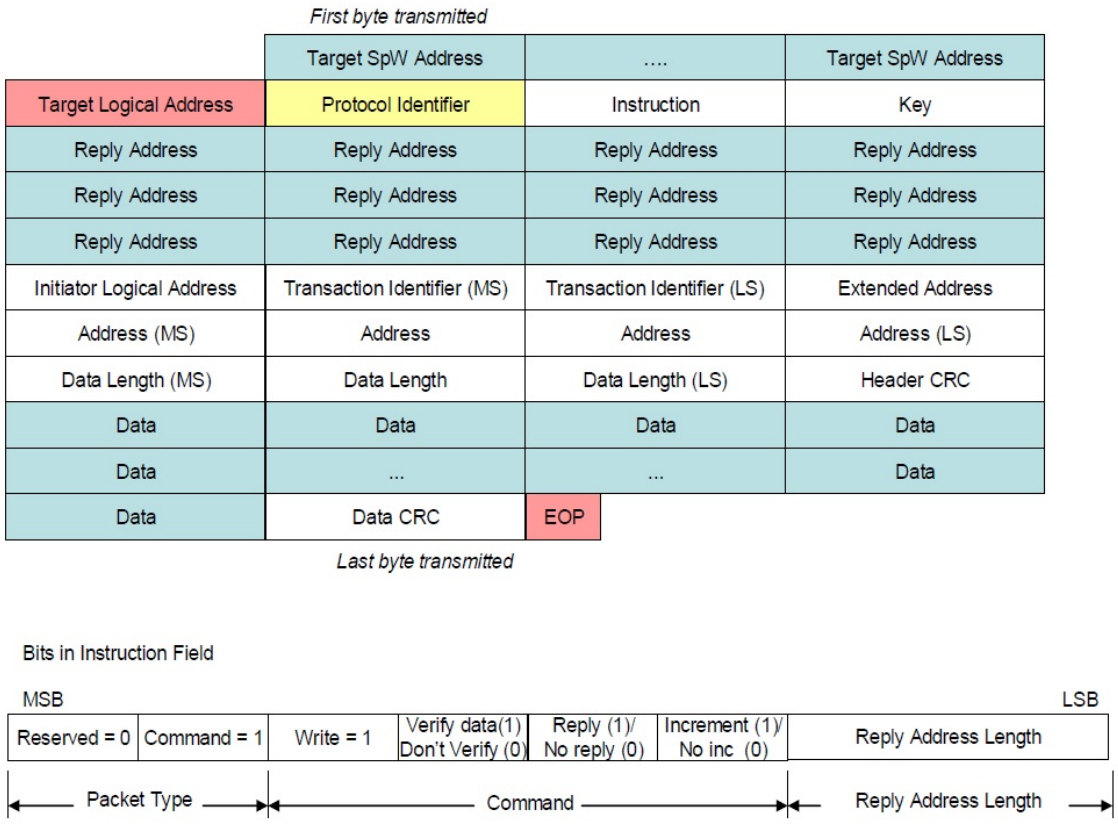


Figure 2.10: Write command packet and bits of the instruction field.

of a SpaceWire packet as defined in section 2.1.4; however since it deals with memory operations it adds new features such as CRC fields both for the header and data content in order to verify the packet integrity.

In case a reply to the write operation is requested, the ECSS standard defines also the setting of the related packet as can been seen in Fig. 2.11.
The reply in particular is used to indicate the outcome of the write operation as codified inside the *Status field*. The complete handshaking mechanism between the Initiator node and the target one is instead shown in Fig.2.12.

The write command sequence starts when the initiator is requested to perform a write operation. The latter sends a write packet through the network to the target node codifying it as previously mentioned. The target on his end scans it for errors by analyzing the CRC field and if no error occurred during transmission the permission to perform the intended operation is asked and granted by both actors

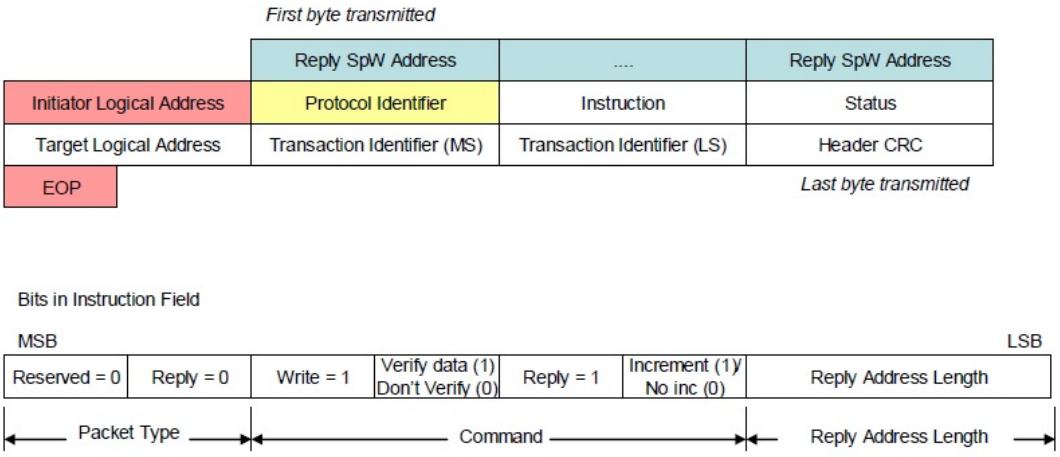


Figure 2.11: Write command reply packet and bits of the instruction field.

sending afterwards the real packet with content to write inside memory location inside data field. Once data has been written inside memory some information back to the initiator can been sent such as an acknowledgement (if requested) or more frequently a reply packet. Once the write reply is received, the initiator node indicates successful completion of the write request.

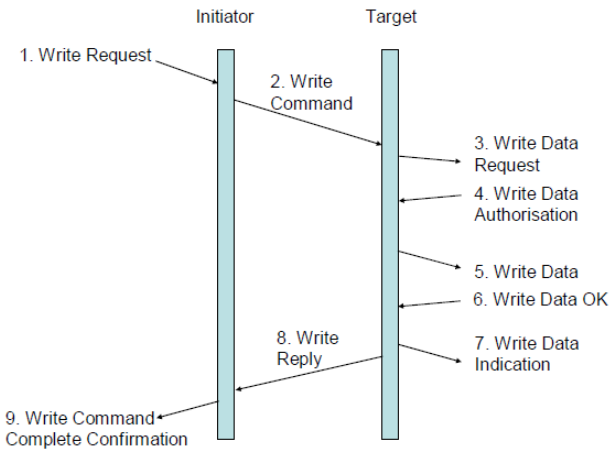


Figure 2.12: Write command and reply sequence.

The read command packet has a similar structure to the write one as can been seen in Fig.2.13. In this case the Read bit is set to 0 and the reply packet is not optional anymore more as it was for the Write command, therefore relative bit

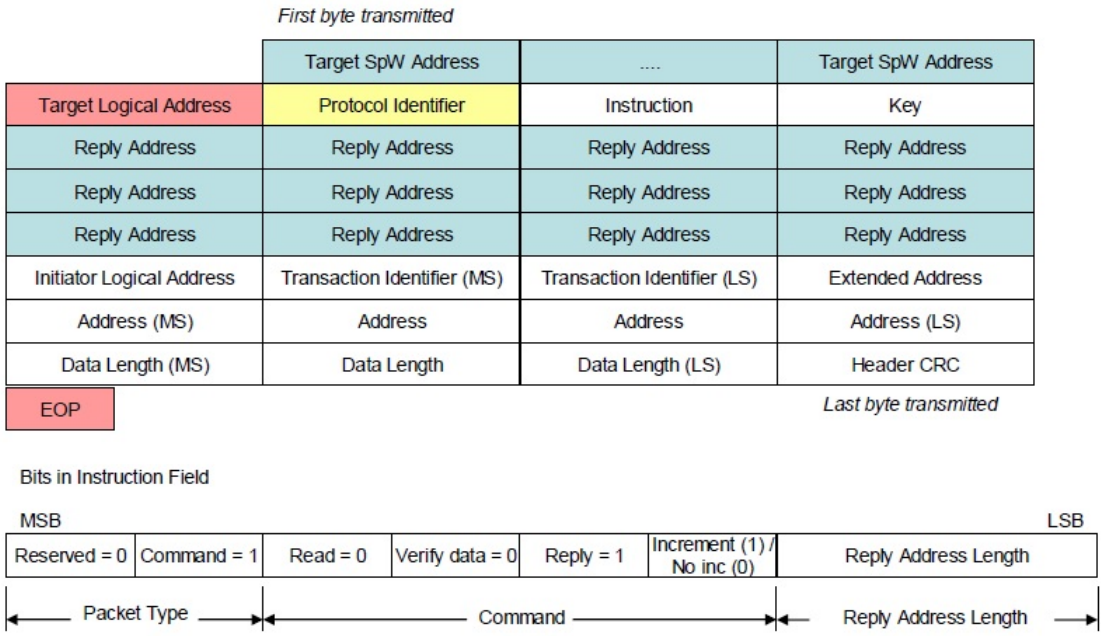


Figure 2.13: Read command packet and bits of the instruction field.

must be set to 1 in the instruction field. In this case the reply, whose structure is similar to the previous command, can contain either the data read from the target or an error code indicating the reason why reading was not performed.

The other command Read-Modify-Write have both command and reply packet structures similar to the write/read command previously reported; also its RMAP sequence is quite the same, therefore they will not be presented hereafter but can been found inside the ECSS standard.

2.2.2 Cyclic Redundancy Code

As previously mentioned, RMAP distinctive feature is to operate a check on data received by means of an error detecting code such as the CRC (cyclic redundancy code). The ECSS standard [14] uses the same methods for computing the CRC for both data and header as CRC forms a field in both parts of a packet.

CRC is one of the most common codes used in networks since is easy to be implemented on hardware and it exploits polynomial divisions. The standard adopts Galois implementation so in practice a code is computed over 8 bits with a Linear Feedback Shift Register (LFSR) implementing in the forward representation the

polynomial $g(x) = x^8 + x^2 + x^1 + 1$ as shown in Fig.2.14.

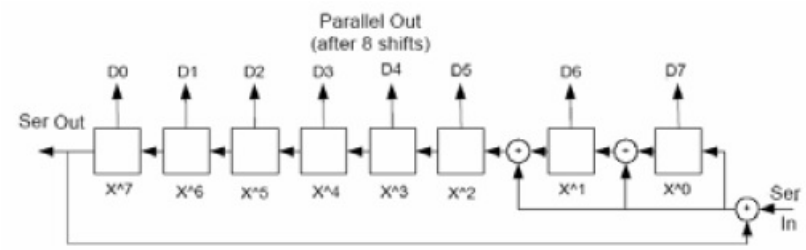


Figure 2.14: Linear Feedback Shift Register for CRC computation [15].

Since first value to be used inside the Shift register (the seed) is crucial in determing its output, the standard sets *00h* value as first one. Moreover, SpaceWire sends the LSB bit first on the link so the code computation starts from the LSB.

A polynomial *m(x)* is created out of the bits transmitted over the link (always LSB first as previously stated) considering also the control bit (always set to 0) and, if present, the parity bit according to the equation:

$$m(x) = m_{n-1}x^{n-1} + m_{n-2}x^{n-2} + ... + m_0x^0$$

At this point the remainder polynomial *r(x)* is created by following the equation:

$$r(x) = [m(x) \cdot x^8] \ \ modulo \ \ g(x)$$

where $r(x) = r_7x^7 + r_6x^6 + ... + r_0x^0$ and r_i are binary coefficients.

The two header and data CRC fields are formed from the 8-bit vector r(x); the least significant bit b_0 of the CRC is coefficient r_7 with the highest power of x, while the most significant bit b_7 is coefficient r_0 with the lowest order.

If the RMAP decoder of target node finds the exact correspondence between both data and header CRC of the received packet and the internally computed codes, it allows the memory operation (write/read/modify) to be performed. Otherwise a CRC error is generated and sent back to the initiator node.

Chapter 3

Field Programmable Gate Arrays

In this chapter it will be given an explanation of the Field Programmable Gate Array (FPGA) technology from a theoretical perspective as well as a brief overview of the space-grade FPGAs and the board used in the hardware implementation.

3.1 FPGA technology

The technology used to implement the project described in this thesis has been the FPGA one. FPGAs (which stands for Field Programmable Gate Arrays) are electronic integrated circuit that can be configured by the user to implement any type of hardware digital system through an electrically programming operation.

Figure 3.1: Picture of Xilinx Virtex-4 FPGA [1].

FPGAs are often preferred nowadays in the electronic industry as a good compromise between the flexibility (generally provided by software) and performances (which is a typical feature of hardware) both coupled with an accessible cost.

Moreover FPGAs are being constantly used as viable alternatives to ASICs due to shorter development phase and time-to-market and because together with micro-processors can form a System-on-chip (SoC) with high computing capabilities.

FPGA world market can be divided up into two main player: Intel (former Altera) and Xilinx. Their FPGAs can be found in most different contests and sectors such as AI, telecommunication, automotive, high performance computing, consumer electronics, military and aerospace and many more (Xilinx is particularly active in these last two fields). In general FPGA market has been one of the most dynamic in the past 20 years with new models being released periodically with increasing density, increasing performace and reduced power consumption.

The device used in this thesis is Xilinx Virtex-4 as shown in Fig.3.1. The use of this commercial FPGA is due to the fact that only space-grade Xilix devices are Virtex 4 and 5 which employ a radiation hardened technology and are more convenient, for all the reasons above mentioned, than space qualified ASICs.

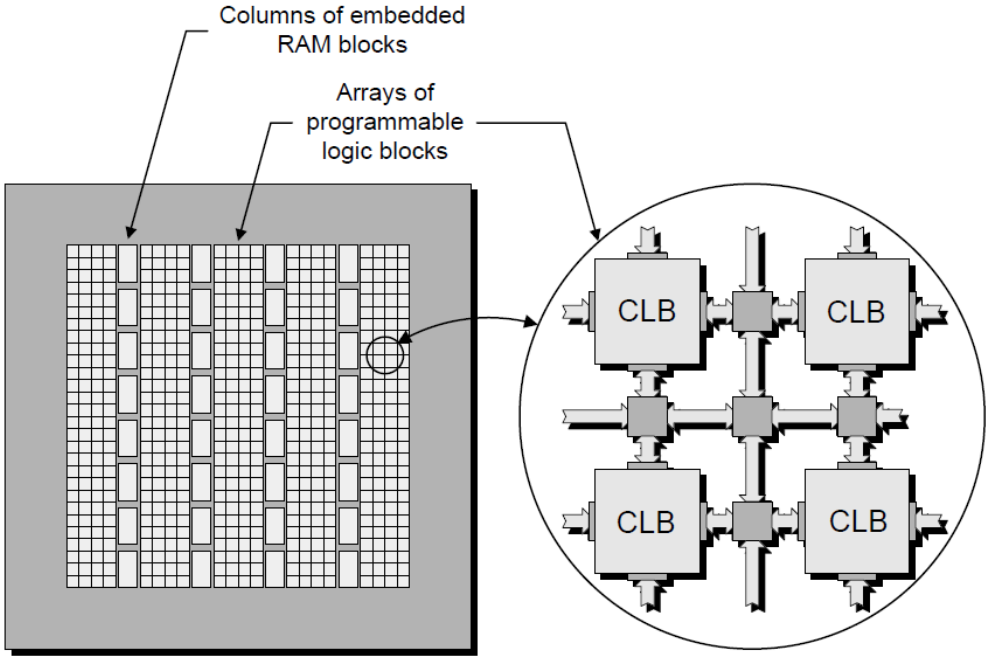


Figure 3.2: General architecture of a modern FPGA.

Figure 3.2 shows the architecture of a modern FPGA (taken from [16]). It is generally referred as an "islands structure into a sea of programmable interconnecs". In fact these logic arrays are organized in terms of:

- **Configurable logic blocks** (CLB) which are clusters of logic cells capable

of implementing any logic function and therefore creating circuits of several gates. One CLB contains 4 smaller structures called *slices* each made up of 2 logic cells;

- **Programmable interconnects** which are basically wires reconfigurable through switches to connect together the different logic blocks I/Os creating therefore larger circuits.

Moreover, since modern FPGAs are becoming more complex, inside each single chip it is possible to have several RAM blocks, multipliers to form Multiply-and-Accumulate units (MAC) and even transceivers for different communication standards (for example Ethernet).

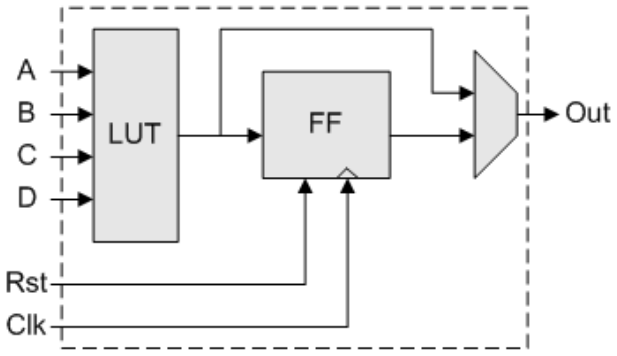


Figure 3.3: Basic structure of an FPGA Configurable Logic Block.

As we know any digital circuit implements a Boolean function which can be expressed by means of its truth table, a table summing the outputs value according to each combination of circuit inputs. The idea therefore consists into writing this table inside a memory, a SRAM so that by rewriting the memory the logic circuit to be implemented can be changed: this results into the Look-up table (LUT). LUT are present inside an FPGA logic block; moreover, modern FPGA can have from 2-input up to 6-input look-up table which are therefore capable of implementing any combinational circuits with a number of inputs from 2 up to 6.

Since combinational circuits are not enough to create complex circuit (which are sequential ones) each logic block has one flip-flop to possibly store the LUT output as shown in Fig.3.3 ([2]). This sequential element can be skipped using a multiplexer whose selection signals are stored into a latch (set by the user in the configuration bitstream as later described).

The second important element inside an FPGA is the programmable interconnects matrix. Interconnects become very relevant since occupy larger area with respect to logic blocks and may affect directly performances and power consumption (longer are the wires, bigger is the delay and power dissipated), becoming the bottleneck in FPGA technologies. Moreover, together with wires there are switches which allow to connect interconnects together.

Depending on the nature of this switches we can have different types of FPGA technologies:

- *Antifuse-based FPGAs*: here the connection between different wires is achieved through a fuse which can become an open connection if a high current is forced across it. In this way it is possible in an electrical way to program the interconnections configuration. Although this method allows to achieve denser switches it can be used only once.

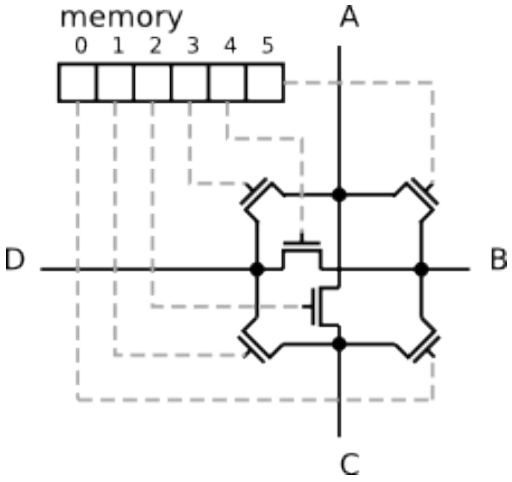


Figure 3.4: Structure of a switch matrix for a pass-transistor-based FPGAs.

- *EPROM/EEPROM/Flash-based FPGAs*: they are an evolution of the previous technology to achieve high re-programmability. They are based on a EPROM /EEPROM /FLASH cell where it is possible to find a transistor with a second gate, called floating gate. Using UV light (for EPROM) or an high voltage (for EEPROM/FLASH) it is possible to erase the cell.
- *Pass transistor-based FPGAs*: in this case two wires are linked by means of a pass-transistor controlled by the bit stored into a memory, generally an SRAM. This allows to achieve higher re-programmability since to change configuration it is sufficient to write the memory a second time but uses larger area since an SRAM cell is made up by 6 transistors.

The FPGA used in this thesis work, the Xilinx Virtex-4, is an **SRAM-based FPGA** so it belongs the last type, the one using pass-transistor for interconnections. An example of the structure of the switch matrix, located every 4 logic blocks, can be seen in Fig. 3.4 ([3]).

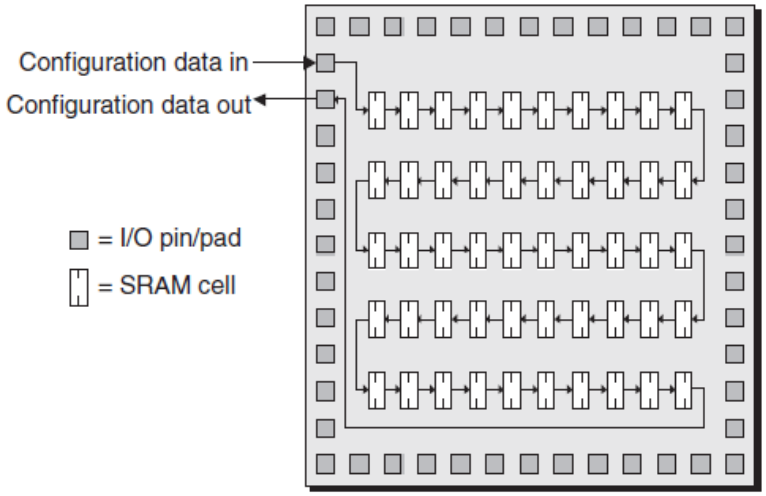


Figure 3.5: FPGA programming through JTAG scan chain

It is worth remembering that SRAM-based FPGAs are a volatile technology therefore as soon as the device is without power supply, memory loses the stored information and the device has to be reconfigured again. For this reason, generally the FPGA is associated to a Flash which is non-volatile memory and contains the configuration file (a binary file called *bitstream* which stores the bit to configure every pass-transistor) to be downloaded inside the FPGA at power-on.

The bitstream is downloaded inside the FPGA using the scan chain which is something generally used for testing purposes (known as JTAG). Basic idea is that all the millions of RAM cells described so far (for the LUTs or the memories associated with interconnections) are connected together to form a long shift register as shown in fig 3.5 (taken from [16]).

Each bit of the bitstream is inserted inside the chain one after the other from the outside of the chip by means of a serial input and a small state machine. The uploading and subsequent configuration may take several seconds. Bitstream is generated by means of CAD tools as described in next section.

3.2 FPGA design flow

The typical design flow followed in this thesis for FPGA technology as the final hardware implementation, can be seen in Fig. 3.6 (from [16]).

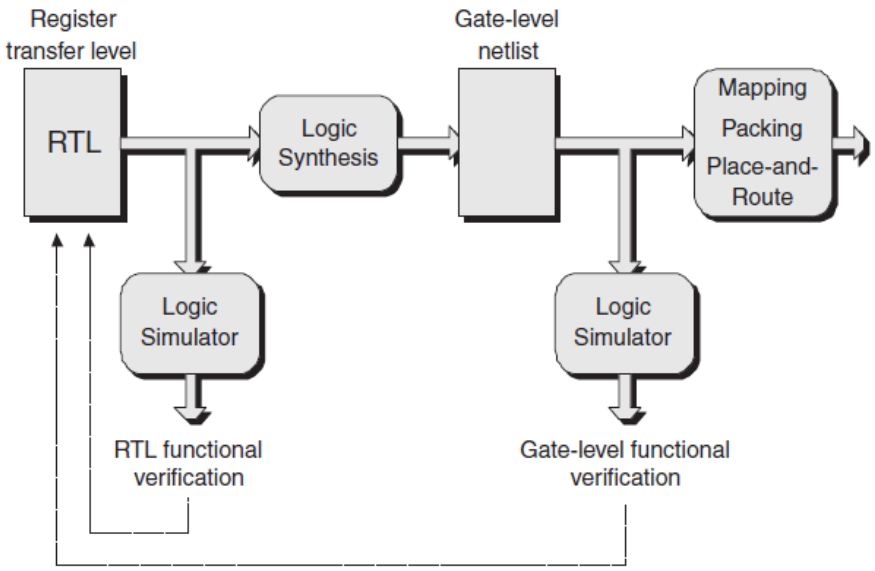


Figure 3.6: Typical FPGA design flow.

First phase consists into hardware design and its subsequent description at RTL level using hardware description languages (HDL) like VHDL or Verilog. In order to be sure that the HDL code completely meets hardware functionalities and specifications, it is generally simulated with a logic simulator (like *Mentor ModelSim*, the one used in this thesis). The passage between design and simulation and viceversa can requires several iteration cycles until the HDL code describes the desired system with all its functionalities.

After these first two phases follows the *Logic Synthesis*. During this step the CAD tool compiles the VHDL files and is able to create the synthesized circuit netlist using the different gates and inferring memory elements, providing moreover some first optimizations and a preliminary resources estimation. Synthesis phase ends also with a preliminary performance analysis: the maximum frequency at which the design can run (without any physical implementation yet performed) is provided by the tool used in this phase called *Synthesizer* which is generally part of a single CAD tool used in all the design flow of Fig. 3.6.

Subsequently the tool proceeds with the *Mapping*. In this step the program decides

which physical FPGA resources to use: it basically maps the boolean equations of the synthesized gates to LUT or chooses which FFs or memory blocks to use.

Then the tool goes on to the *Placement* so it arranges the system into the different slices inside CLBs. After that, it passes to the *Routing* step in which configures the programmable interconnections to link everything together. At this point the resource estimation provided is generally the final one. Moreover also in this last step a timing analysis is performed to provide performance estimations as in the synthesis step or to check if timing constraints, imposed by the user, are met.

3.3 Space-grade FPGAs: Xilinx Virtex devices

FPGA technology, as already said, is one of the best choice for computing intensive applications: their wide use has lead to the creation of the term **Reconfigurable computing**. However, not all FPGA devices can be adopted into the space environment since ionizing radiation particles are the cause of Single Event Upset (SEU) faults. In particulary the SEU is a fault model corresponding to the bit stored in a memory element to flip its value resulting into a device failure: this is critical since memories are at the core of FPGA devices.

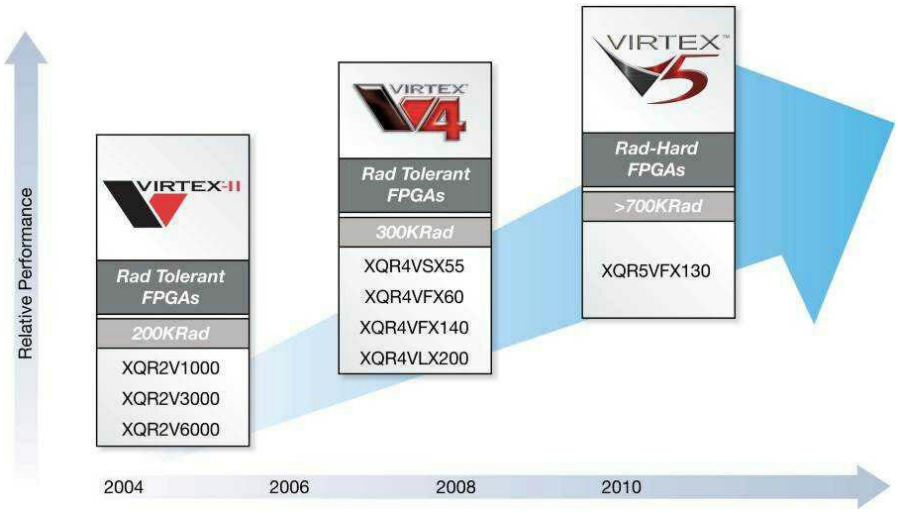


Figure 3.7: Roadmap of the Xilinx space-grade Virtex families FPGAs.

Xilinx is the main producer offering space-grade FPGAs and Virtex devices are commercialized for space oriented applications. From the roadmap (taken from [4]) shown in Figure 3.7 it is possible to see the presence of two different families:

Xilinx *Virtex-4 QV* and *Virtex-5 QV*.

The main difference between these two families is the rad-hard feature of the Virtex 5 QV family: in fact to prevent the radiations effects, the device is realized with an epitaxial layer to prevent latch-up phenomena and it is Rad-Hardened By Design (RHBD) exploiting the TMR (Triple modular redundancy). Moreover, the Virtex 4 FPGA is realized with 90 nm Copper CMOS process while the Virtex 5 FPGA employs the 65 nm Copper CMOS one: this results into a performance slight difference since Virtex 4 maximum frequency is 400 MHz while Virtex 5 can reach 450 MHz (features taken from [20] and [21]).

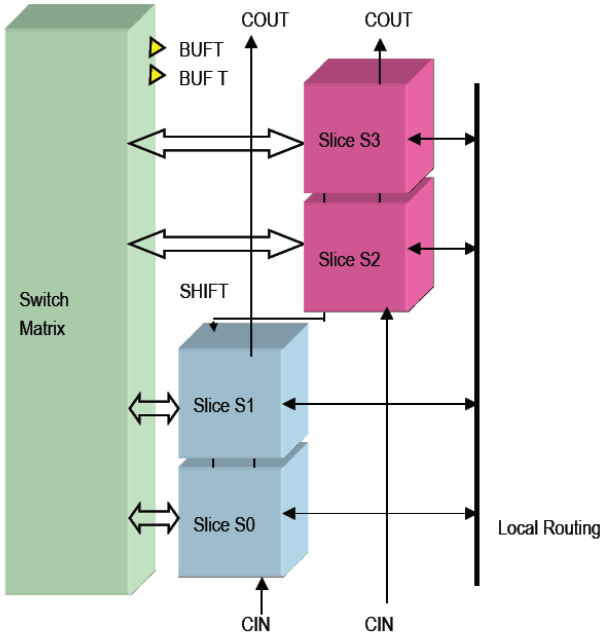


Figure 3.8: Internal architecture of Virtex-4 CLB.

Figure 3.8 shown how a CLB is structured inside a Virtex 4, which is the device used in this thesis work. Most important element is the *slice* which internally is equal to structure of Fig. 3.3 but it presents two LUT and two registers. Moreover, local routing provides feedback between slices in the same CLB while a switch matrix provides access to general routing resources of the FPGA. As visible from Fig. 3.8, a single CLB of a Virtex-4 is made up by 4 slices each containing two 4-input LUTs while in Virtex-5 we can find 4 slices each containing four 6-input LUTs. Except for this, the two families don't differ for the other resources: as visible from figure, a CLB presents running vertically a carry-in (CIN) and carry-out (COUT)

to create, for example, faster adders. Additional device resources are Block RAMs (BRAMs), Rugged DSPs, Clock Management Tiles (CMT) containing PLLs and Digital Clock Managers (DCM) and High-performance parallel IO banks ([21]).

3.4 GR-CPCI-XC4V board

In order to implement the design on hardware, the GR-CPCI-XC4V board, produced by *Aeroflex Gaisler*, has been employed since present inside Avionic laboratory. This board has been developed as a fast way to prototype either Leon 3 processor using Gaisler Research IP cores or new designs and can be also inserted in GR-Rasta systems (employed for testing) using the PCI plug-in interface.

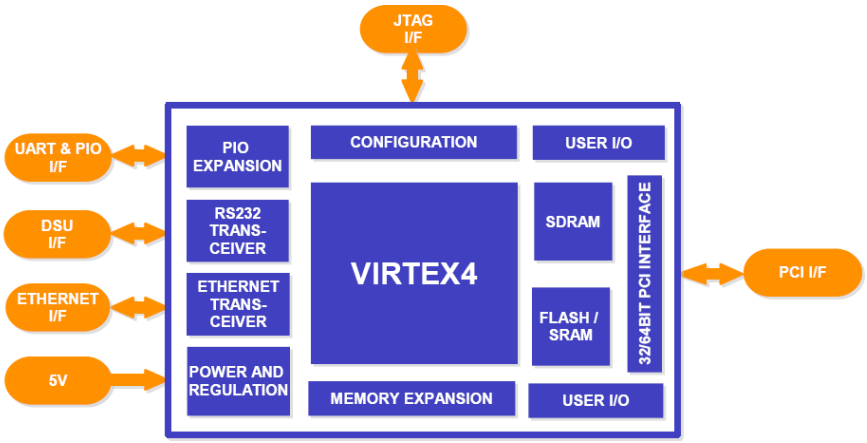


Figure 3.9: GR-CPCI-XC4V board block diagram.

The board architecture is shown in Fig. 3.9 (images and technical features have been taken from [18]). We can immediately find the Xilinx LX200 Virtex-4 FPGA used in this thesis, though other versions exist equipped with Virtex-5, together with on-board 128 Mbit Flash memory and several expansion sockets to connect SRAM/SDRAM.
Additional important features of this development board are:

- the power supply input which operates at 5V. It can come either from an external power supply source or from 5V PCI power supply via PCI connector pins. The board can also be configured to operate at 3.3V;
- the JTAG interface used for both programming and debug purposes. We can find a connector for a standard JTAG interface (TMS, TDI, TDO, TCK) that

can be used to program either directly the FPGA (in a volatile way) or to act on the Flash for making the design permanent. In this thesis, FPGA has been directly configured using *iMPACT* tool (part of the Xilinx ISE CAD);

- a standard serial UART 3 pins interface for the debug and a standard 9 pin D-type connector allowing direct interface to RS232 transceiver;
- a user expansion (called *GENIO*) which allow to attach to the main board specific mezzanine boards which are then linked to I/O connectors J8, J9, J10 and J11. These connectors provide access to the memory interface (J9) and up to 172 user I/O signals (connectors J8, J10, J11). In this thesis, the connector J10 has been used for connecting the SpaceWire mezzanine as described later;
- the main oscillator for the Virtex-4 which is a (50 MHz) precision oscillator soldered on the board. Then, in order to generate any desired frequency using this clock input, several internal DCM/DLL modules (they are present inside the Virtex-4 and have been generated as described in next sections) have been employed;
- an on-board push button switch used to reset the board and erase the FPGA.

Chapter 4

SpaceWire network architecture

In this chapter the SpaceWire network will be presented in terms of IP core block components, describing in details their structure, their working behaviour and in some cases how some crucial pieces have been generated.

4.1 General structure

The general architecture of the implemented network is shown in Fig. 4.1.

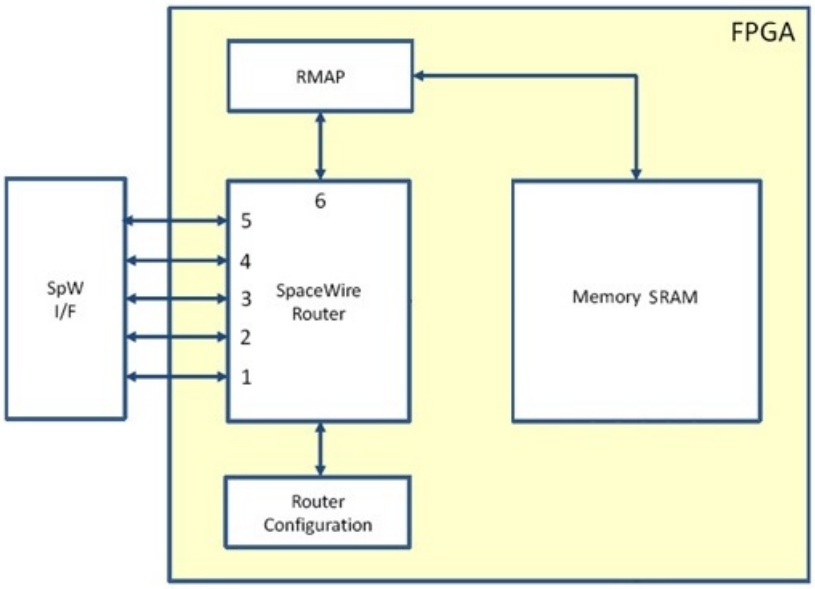


Figure 4.1: High level architecture of SpaceWire network

As can be seen the network is made up by four different blocks:

- **SpaceWire Codec**: it allows to establish a SpaceWire link compliant with ECSS standard to communicate with the outside world (in Fig 4.1 is called SpW I/F). It is embedded inside router or RMAP cores as an external interface or can be used as standalone component. In the following, first option has been preferred.

- **SpaceWire Router**: it provides five SpaceWire external links and two internal ones. In particular Port 0 is used for the configuration of the router using RMAP protocol. Router port 6 is used instead to access the target memory (or another IP's internal register file) for writing or reading informations with the aim of configuration, control and status acquisition.

- **RMAP block**: as described in ECSS-E-ST-50-52C it is used as a standard method for both Router configuration and memory operations; it has been used in the Target version.

- **Target memory**: in this thesis it is a simple single port SRAM memory using an FSM to manage successfully both write and read operation and impementing an acknowledge handshake to be compliant with RMAP interface. In more advanced project it can be replaced by register files or configuration memories of more complex IP cores.

The first three of these cores have developed by Masaharu Nomachi, Takayuki Yuasa, and Shimafuji Electric in collaboration to JAXA and have been used in this thesis to build the SpaceWire network after their reconstruction and simulation, using ***Xilinx ISE*** and ***Mentor ModelSim*** environments, to verify the declared compliance with ECSS standards.

4.2 SpaceWire Codec IP core

SpaceWire Codec is a VHDL IP core (found at [5]) aimed at implementing the SpaceWire communication protocol between the on-board components. It is declared to be compliant with the ECSS-E-ST-50-12C standard. It was intended to be implemented on either Xilinx or Altera FPGA.

The core is able to convert any data information over single or multiple bytes (considering the data over 9 bits to include also the control character as described in the previous chapter) into the SpaceWire format employing only the two Data and Strobe bits.

Originally the codec was intended to be used only in a simulation environment to emulate communication from the outside environment (on-board processor or camera for instance) towards the SpW network. However, a further analysis showed that the core had to be synthesized since it was embedded as part of the Router and RMAP cores to allow also internal conversions from SpW signals. This core supports a communication rate up to 200 Mbps. Codec technical features and following images have been taken from [23].

4.2.1 Core architecture

SpaceWire CODEC IP block diagram is shown in Figure 4.2.

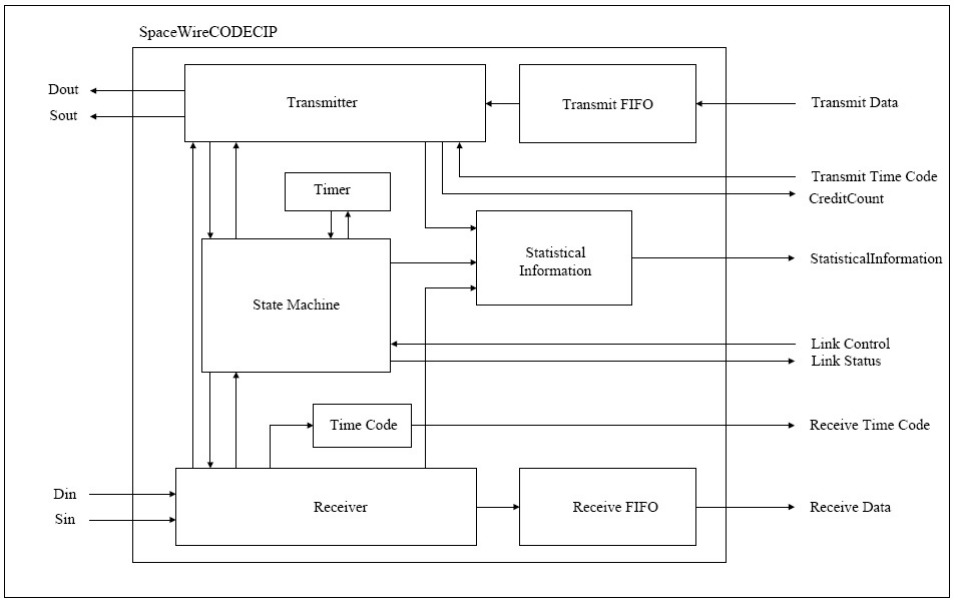


Figure 4.2: SpaceWire Codec IP core block diagram

Block diagram shows that the receiver decodes the input from SpaceWire and writes it to Receive FIFO presenting it at the output over 9 bits. Transmitter on the other hand, converts the data written in Transmit FIFO into data and strobe signals and outputs them to SpaceWire.

Both Transmitter and Receiver are connected to a SpaceWire Machine which defines timing of any action and the list of subsequent steps to allow transmission or reception. Moreover, a timer (basically a counter) is interfaced to the State machine providing the generation of 6.4 µs or 12.8 µs timer as indicated in the ECSS

standard in case of NULL or zero data reception. Finally, a Statistical Information block provides information about the current state of I/O transmission.

Transmit and Receive FIFOs

These two blocks are a 9×64 FIFOs in which the host side writes N-Char that have to be sent or received by the SpW Codec. Up to 56 N - Char can be written, FULL interface signal becomes "H" when 56 data are written in case of the TransmitFIFO while in case any data is not written inside the FIFO, the Empty signal transit to "H". The timing chart of a write operation inside FIFO is shown in fig.4.3.

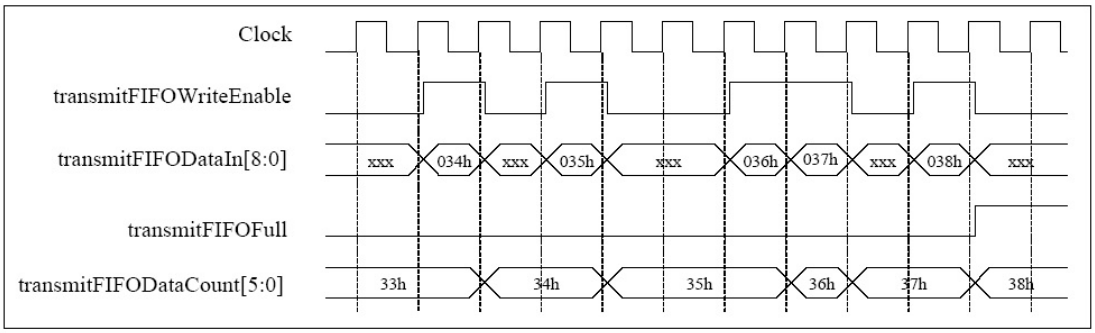


Figure 4.3: Timing diagram of writing into Transmit FIFO.

Transmitter

Only when there is transmission permission from the State Machine, data written in Transmit FIFO is converted into data and strobe signals. When instead there is no data to be transmitted, the Transmitter block outputs a NULL SpaceWire character. The transmission rate of the block is determined by the *Transmit-Clock* and the *transmit-Clock-Divide-Value* defined both as inputs values over respectively one bit and 6 bits. After the initial Link up phase the transmission rate can be changed by varying the value of *transmit-Clock-Divide-Value*, provided that the block will operate anyway at 10 Mbps during Link initialization.

Receiver

The Receiver behaviour is a little bit more complicated than the Transmitter one. One of the main differences lies into the necessary I/O synchronization. The data and the strobe signal are synchronized with the internal clock before decoding. Synchronization mechanism of data and strobe signals is shown in Fig. 4.4.

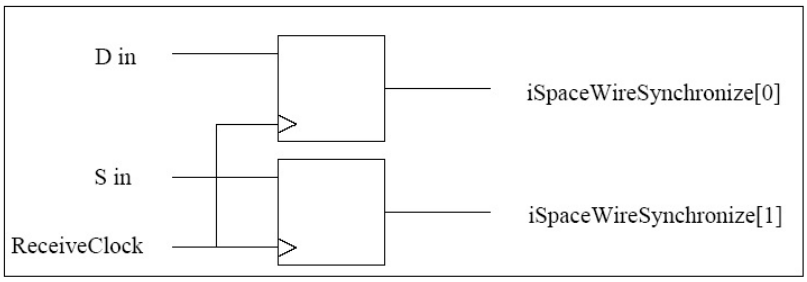


Figure 4.4: Synchronization mechanism inside Codec receiver.

When N-Char (data, EOP or EEP) is received, data is written to Receive FIFO and notified to State Machine. Even when NULL, FCT or TimeCode are received, State Machine is notified. In case instead only FCT is received, then in this case also the Transmitter is notified. All data other than N-Char (data, EOP, EEP) that will be received by SpaceWire Codec will not be written into Receive FIFO.

When an escape error or parity error is computed from the received data due to a CRC error or a not matching sequence, the data is discarded and the error is notified to State Machine resulting into stopping the reception behavior. Finally, if the data and strobe signal does not change for 850 ns, State Machine will be notified as a disconnection error.

4.2.2 Link state machine

The State Machine is a crucial actor into the Codec correct working behaviour. It basically manages link interface initialization, normal operation and error recovery processing as described in [22] (from where FSM graphs have been taken).

Link initialization is the first step into letting two actors which want to exchange data over the link to start the transmission after having reached synchronization and ready to transmit data, FCT and EOP characters. Synchronization consists of decoding the data and strobe signal to produce the bit clock, as mentioned in previous chapter, through an XOR operation. In order to begin the communication the two transmitters at the ends of the link have to be synchronized otherwise the FSM does not move from the reset state and several attempts to resynchronise will be tried until connection is established.

The State machine moves along the following states as shown in Fig.4.5:

- **ErrorReset**: in this state both *EnableTransmitter* and *EnableReceiver* becomes "L". In this case both Transmitter and Receiver stop any operation

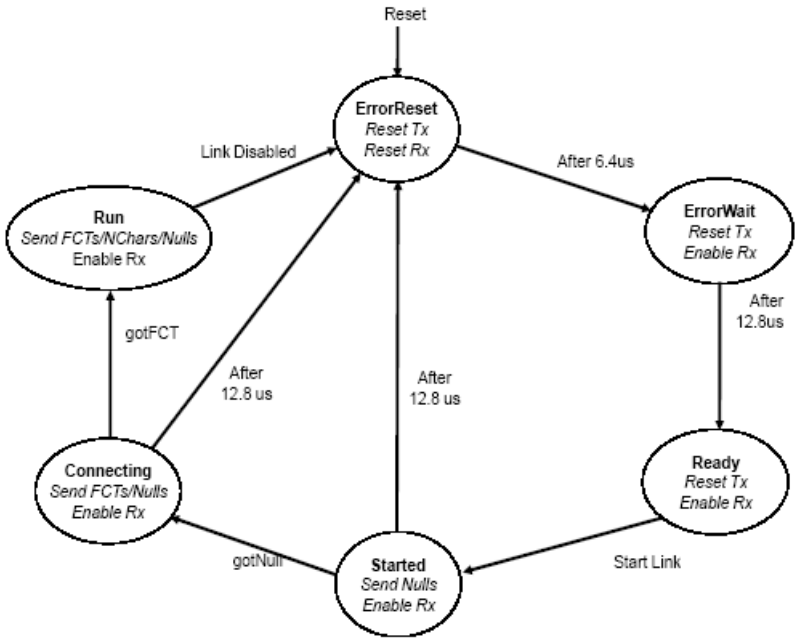


Figure 4.5: SpaceWire Interface State machine.

previously undertaken being in reset. After 6.4 µs the State Machine will automatically transit to *ErrorWait state*.

- **ErrorWait**: this state makes the signal *EnableReceive* becoming "H" and the lets the Receiver to operate. After a time of 12.8 µs the State Machine will automatically transit to *Ready* state. In case the Receiver detects any error, the machine will transit to *ErrorReset* state.

- **Ready**: in this state the machine makes the signal *LinkEnable* to go "H". This makes the machine to transit to *Started* state. Again, in case the Receiver detects any error, it transits to *ErrorReset* state.

- **Started**: in this state both the signals *EnableTransmit* and *SendNULLs* go to "H" and the Transmitter sends a NULL character. When Receiver receives NULL, it transits to *Connecting* state of the State Machine. In case the Receiver detects Error or transits to the *Started* state and 12.8 µs is elapsed without any receiving, it transits to *ErrorReset* state therefore resetting the State Machine.

- **Connecting**: in this state the signal *SendFCTs* becomes "H", and the transmitter transmits FCT. When Receiver receives FCT, it moves to the *Run*

state. As in previous state in case the Receiver detects an error or any transmission/reception activity does not take place after 12.8 µs FSM transits to *ErrorReset* state.

- **Run**: in this state the two signals *SendTimeCode*, *SendNChar* becomes "H" so the Transmitter activates the internal time code block and it will be possible to send the N-Char data. In case Receiver detects any Error, the machine will transits again to *ErrorReset* state. From now on the FMS remains into this state until one end of the link is disabled by properly asserting the Link-Disable bit at the input interface of one SpaceWire Codec.

Another procedure to automatically establish the connection consist into setting one of the codec at one end of the link to the Auto-Start mode. This results into a slightly different state machine. A SpaceWire link set in this mode will automatically transit from the *Ready* state to the *Started* state when it receives a bit (gotBit) at the receiver. This feature requires the FSM to be modified by adding a second condition in OR operation with the normal one on the transition between the two previous mentioned states. It is possible to see this in Fig. 4.6.

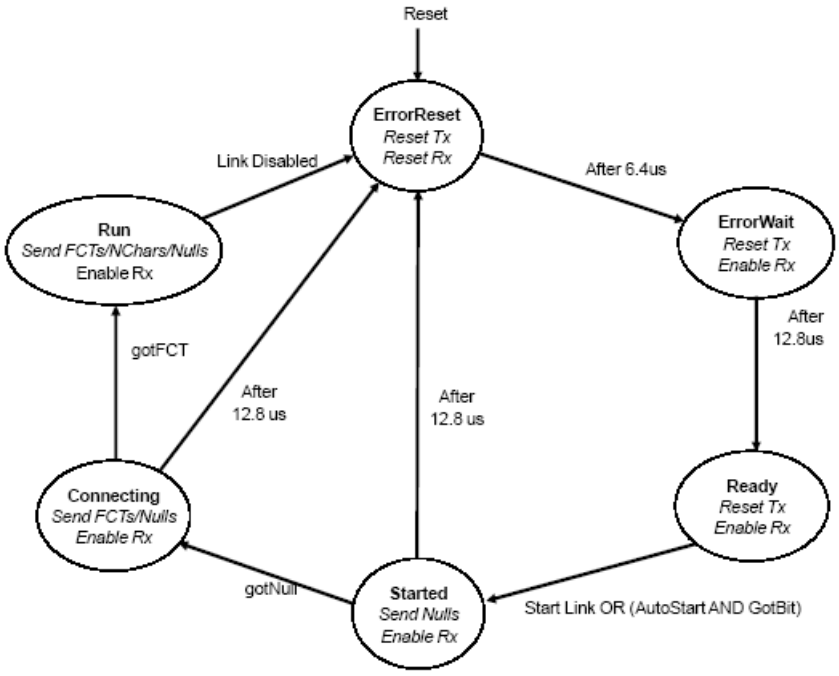


Figure 4.6: SpaceWire State machine in Auto-start mode.

This Auto-start functionality has been used during the experimental implementation of the network on FPGA.

4.3 SpaceWire Router IP core

SpaceWire Router IP core is a VHDL core (found at [6]) aimed at implementing a 6 port router able to be configured both in path and logical addressing by means of RMAP protocol. It is designed to conform to ECSS-E-ST-50-12C standard supporting a communication rate up to 200 Mbps. As before, this core is intended be synthesized on both Altera or Xilinx FPGA targeting the routing function to allow communication between different actors inside the SpaceWire network. Router technical features and following images have been taken from [25].

4.3.1 Core architecture

SpaceWire Router IP block diagram is shown in Figure 4.7.

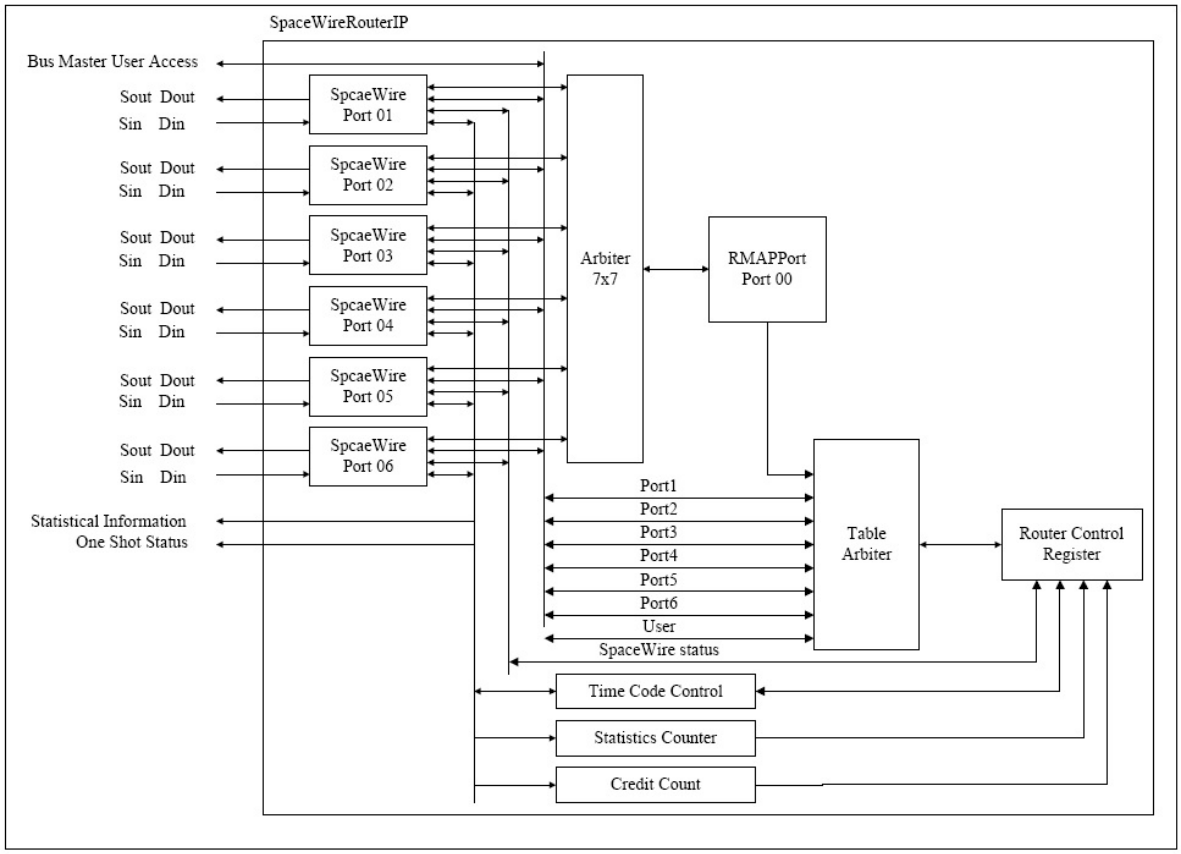


Figure 4.7: SpaceWire Router IP core block diagram

The router presents 6 external SpaceWire ports and an internal port, defined as port 0 which allow router configuration through RMAP protocol. Data received

by any SpaceWire Port is sent to the Arbiter with the request of transmitting the relative packet to the destination port. When connection permission from Arbiter is approved, data transmission starts to the destination port.

RMAP Port 0 is in charge of analyzing the received packet and accessing the internal Router Control Register, updating in writing mode the Routing Table. It is also possible to read or to write the Routing table from outside by acting on the internal bus which will be arbitrated as a normal input along with any SpW port.

SpaceWire Port

An important part of the router is the SpaceWire port. In total 6 external spacewire ports are implemented; however they can be increased acting both on *SpaceWireRouterIPPackage* parameter and on modification of Arbiter or Routing Table attributes. Inside each SpW port module, it is embedded a Codec IP core in charge of converting any incoming information from the 2-bit SpW format into a 9-bit information. I / O from each port is multiplexed by the Arbiter. In particular receiveFIFOs inside each Port send a transmission request to Arbiter, and when they are allowed to transmit, they send the packet to the port of the destination address. Of course in this process a pivot role is played by the routing table (a BRAM) since the destination address is matched with the destination port by a reading operation into this write-and-read memory.

RMAP Port

RMAPPort is a top module that incorporates *RMAPDecoder*, *TimeOutCount*, *TimeOutEEP* sub-modules. The RMAP decoder, which is also present into the RMAP Target IP core, analyzes RMAP packets and accesses the Router Control Register which consists of a set of registers and a routing table.

When the SpaceWire timeout is enabled, each port of the SpaceWire Router IP counts the time from the reception of the first data of the packet until the reception of the packet end (EOP). This is implemented by a counter embedded into the *TimeOutCount* block. If completion of packet reception is longer than the set time, a timeout error will be generated and therefore the RMAP packet will be discarded leading the state machine into the idle/reset state. The timeout value is written inside the Router timeout control register which belongs to the Router Controls Register.

TimeOutEEP block performs instead a different task. If a timeout error occurs on the source port while the source port is sending packets to the destination one,

an EEP is added to the Transmit FIFO of the destination port to complete the packet. For instance: when a timeout error occurs in Port 1 while sending a packet from Port 1 to Port 2, an EEP is added to Transmit FIFO of Port2 which will result into an error in the packet reception with the consequent packet discard.

Arbiter

It is a 7x7 Arbiter which follows a round-robin arbitration between the *SpaceWirePort*, *RMAPPort*, and an external bus called *UserAccess* which can access the Routing Table from outside. Each of them therefore is given no priority since thanks to this arbitration strategy, time is divided in slots and assigned equally to each of the actors involved in a circular order. The condition to follow is written inside the Arbiter Table to generate the control signals Requested, Granted and Occupied. Only after Arbiter authorization, packet transmission/reception between *SpaceWirePort* can be established.

Router Control Register and Routing Table

Router control register block is a sort of register file made up by a collection of different registers used for storing both setting parameters and status of the different Space Wire ports. Inside this block the Routing Table is allocated as a RAM memory area storing the association between logic address and SpaceWire port.

Offset Address	R/W	Function	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0080 ~ 0x03f8	R/W	RoutingTable32~254	Port-31	Port-30	Port-29	Port-28	Port-27	Port-26	Port-25	Port-24	Port-23	Port-22	Port-21	Port-20	Port-19	Port-18	Port-17	Port-16	Port-15	Port-14	Port-13	Port-12	Port-11	Port-10	Port-9	Port-8	Port-7	Port-6	Port-5	Port-4	Port-3	Port-2	Port-1	Port-0

Figure 4.8: Structure of the Routing table

The Routing Table is a collection of 32-bit wide registers where each bit corresponds to an output port number, and the number associated to each register corresponds to a logical address. When logical addressing is performed, data is read from the address corresponding to the logical address and received data is put in output to the port where "1" is written. For example, if we want to route to port 4 when logical address 32 (0x20) is specified, we need to write 0x00000010 value inside register number 32 which correspond, into Router memory map, to register at address 0x0080.

4.3.2 CRC and Routing table generation

SpaceWire Router IP core needs of two additional sub-blocks in order to work correctly. The first is a ROM memory which has to store coefficients needed to compute, time by time, CRC code as defined into ECSS-E-ST-50-12C standard. In case CRC field of the transmitted SpaceWire packet, is not coincident with the one computed internally the packet is discarded since an error had occurred. The block name used for this memory has been **crcRomXilinx**.

The second block needed is the one actually implementing the Routing Table. To achieve the purpose, a RAM has been used to build the $32 \sim 254$ registers where single word is 32 bits wide. This block has been called **RamXilinx32x256**.

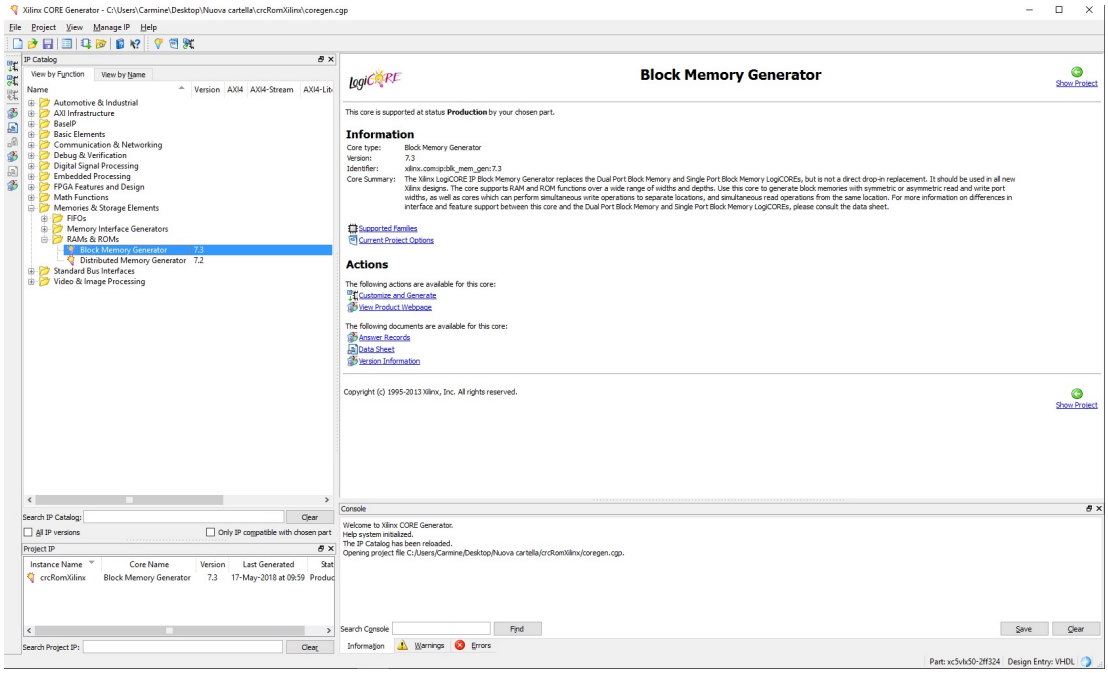


Figure 4.9: Use of Block memory generator to generate RAM/ROM.

Both sub-blocks have been generated using ***Xilinx Core generator*** tool. Since both blocks are pretty standard R or W/R memories this tool has been used to generate both netlist (*.vhdl*) and implementation (*.ngc*) files while only the last one has been used in the synthesys phase. A correct setting of memory system parameters, such as data or address width, has been performed as requested by SpaceWire Router and Codec technical documentation.

Values of CRC coefficients have been provided already memorized during the creation of the core and passed to Core Generator using *.coe* file which basically

resembles a CSV file. Also default values memorized in the routing table have been embedded in the same way, respecting the association between logical address and destination ports providing the router an initial static configuration.

After launching Core Generator, the Memory Block generator tool has been used to generate the two blocks: it has been chosen among the memory options menu as shown in figure 4.9.

As far as *crcRomXilinx* is concerned a Single Port ROM has been chosen among the different memory types available; the memory had 9 bit address field and 8 bit of data width as shown in fig. 4.10. The guided procedure followed with the *.coe* file addition and ended with block generation. As previously said only the implementation file was used (.ngc file) checking in the synthesys report that the memory block inferred by the ISE synthesizer was the desired one.

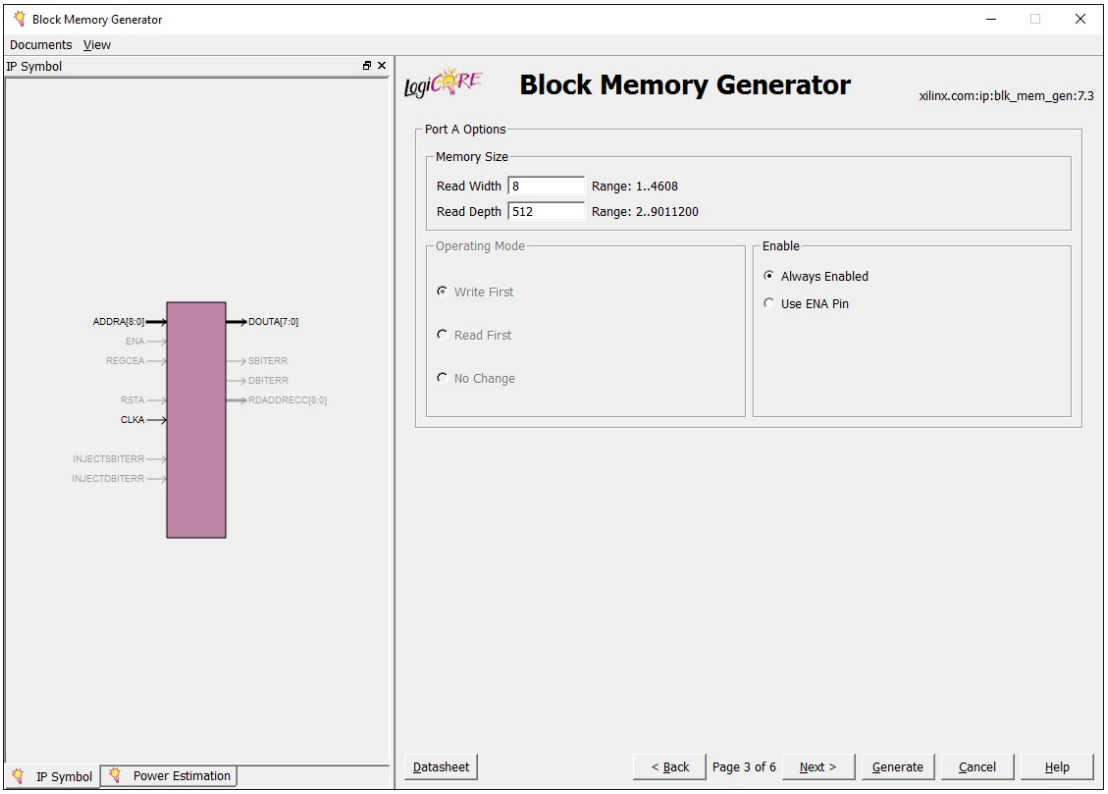


Figure 4.10: Generation of *crcRomXilinx* single-port ROM memory.

The *RamXilinx32x256* block was created using exactly the same procedure though it was a Single Port RAM kind. For this reason it presented an input and an output ports both 32 bits wide and an address port of 8 bits. The operating mode chosen

was the write-first while the write-enable input was set to a single bit which allowed to define the R/W mode. This process is shown in figure 4.11.

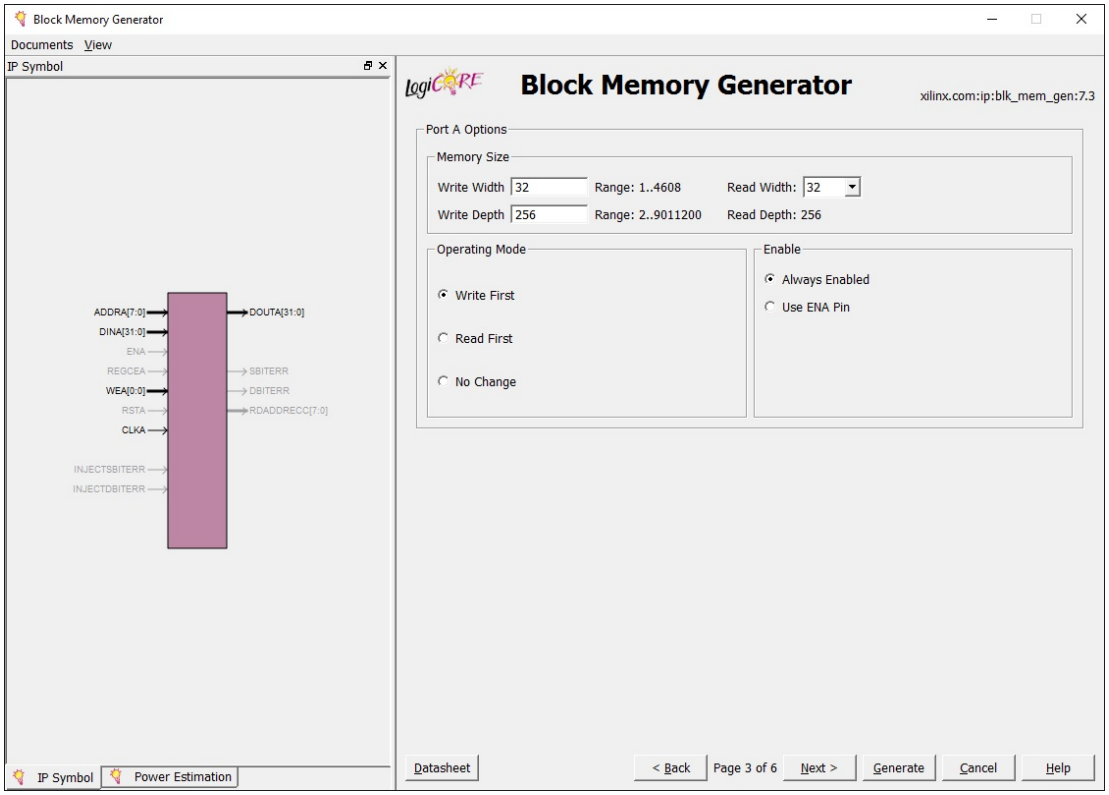


Figure 4.11: Generation of *RamXilinx32x256* single-port RAM memory.

All the other additional features for memory blocks provided by Xilinx Core Generator such as ECC options or the additional reset signal were not considered since Router and Codec IP cores contained memories wrapper not specifying these additional inputs. The creation process, as before, requested the *.coe* file and finally terminated with the generation of the core block.

4.4 SpaceWire RMAP IP core

SpaceWire RMAP Target IP core is a VHDL core (found at [7]) aimed at implementing the RMAP (Remote Memory Access Protocol) protocol to support reading and writing into the memory of a remote SpaceWire node. It can be hence used for SpaceWire network configuration, SpaceWire node control, and for information transfer between SpaceWire nodes.

It performs Write, Read, Read-Modify-Write on the memory through a WISHBONE-like bus for the RMAP packet received from the SpaceWire network. Three types of bus widths are available: 32 bits, 16 bits, and 8 bits which can be selected by setting accordingly the generic parameters inside the package file. This core is designed to conform the ECSS-E-ST-50-11C standard targeting both Altera or Xilinx platform and supporting a transfer rate of up to 200 Mps. RMAP core technical features and following images have been taken from [24].

4.4.1 Core architecture

RMAP Target IP core block diagram is shown in Figure 4.12.

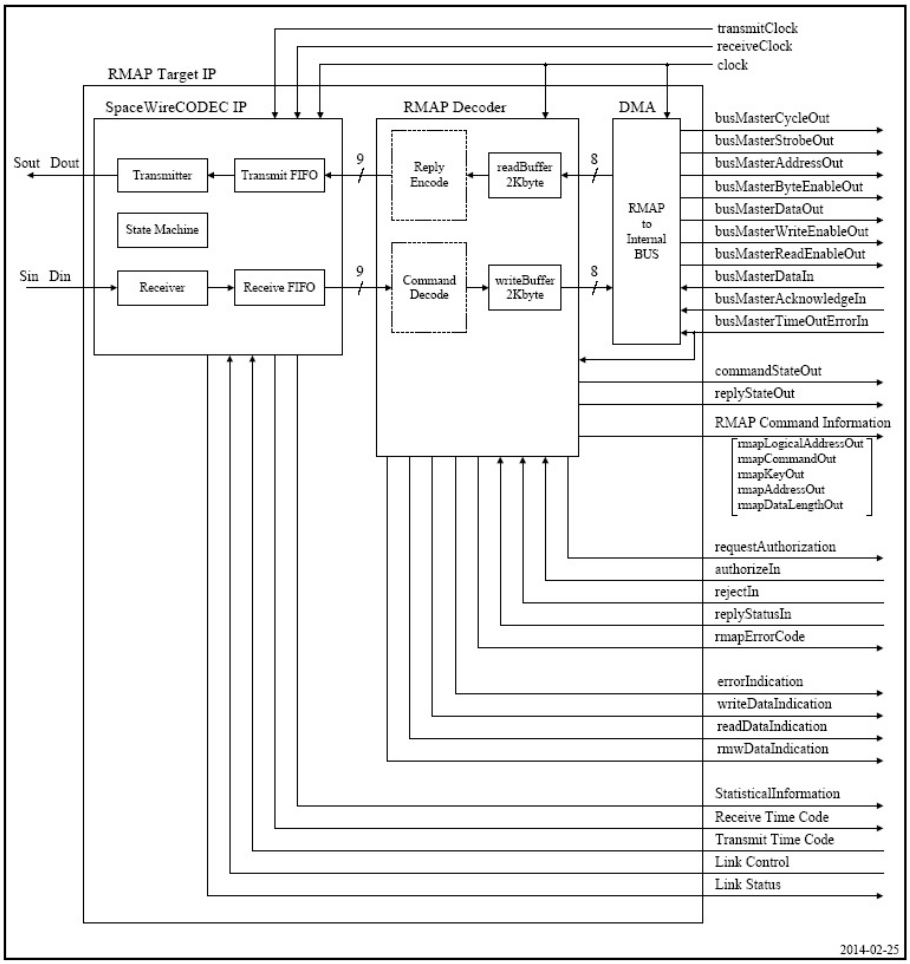


Figure 4.12: SpaceWire RMAP IP core block diagram

SpaceWire Codec is in charge of decoding any information into usual 9 bit data,

acting thus as an interface module. RMAP decoder receives data packet and following a FSM state evolution, it recognizes a valid RMAP format information. It pass over the packet to an interface module (DMA controller) which transfers information over a bus-based system.

RMAP Decoder

This is a module that analyzes RMAP Command packet and generates Reply packet. It is also embedded inside the SpW Router Port 0 since it decodes information to configure the router.

RMAP decoder reads received data from Receive FIFO and analyze Command packet. When parsing the packet, it pass the logical address, key, command, address, data length (all contained inside the packet itself) to the user side and wait for the access permission/refusal response from the external memory.
The table of the maximum data length that can be handled by RMAP core for each command type is shown in Table 4.1.

RMAP Command	**Read/Write**	**Presesence of pre-write verification**	**Reply**	**Address**	**Maximum data-lenght that can be handled**
"0010"	Read	None	Present	Fixed	16 Mbyte
"0011"	Read	None	Present	Increment	16 Mbyte
"0111"	RMW	None	Present	Increment	4 byte
"1000"	Write	None	None	Fixed	16 Mbyte
"1001"	Write	None	None	Increment	16 Mbyte
"1010"	Write	None	Present	Fixed	16 Mbyte
"1011"	Write	None	Present	Increment	16 Mbyte
"1100"	Write	Present	None	Fixed	2048 byte
"1101"	Write	Present	None	Increment	2048 byte
"1110"	Write	Present	Present	Fixed	2048 byte
"1111"	Write	Present	Present	Increment	2048 byte

Table 4.1: Maximum data length handled by RMAP IP core.

RMAP DMA Controller

This module outputs the address and data of RMAP packet, analyzed by the RMAP decoder, to the internal bus and performs read/write access to memory.

When receiving Write command from RMAP decoder, it writes the data in a *writeBuffer* (a FIFO queue) and then transmit it to the external memory via an internal bus. When receiving Read command from RMAP decoder, it reads data from external memory through this internal bus and write it to the *readBuffer* (a FIFO queue). The internal bus is used for data transfer between RMAP Target IP core and the external memory. The VHDL generic *cBusWidth* can be changed inside the *RMAPTargetIPPackage.vhdl* file, so that bus width in output of the DMA module, can be varied between 32 bits, 16 bits or 8 bits.

4.4.2 Working behaviour dataflow

The correct sequence of steps, in case a command data is processed, consists into:

① When SpaceWire CODEC IP receives SpaceWire data, data packet is stored in Receive FIFO.

② The RMAP Decoder module reads and subsequently deletes the 9 bits (Data Control Flag + data) from Receive FIFO. It then extracts the data and decodes the RMAP Command.

③ Until the header CRC is checked by RMAP decoder, the information (LogicalAddress, Command, Key, Adddress, Data Length) and transaction request signal (*requestAuthorization*) is not send to the other side.

④ Until the transaction approval signal (*authorizeIn*) is returned from the user side (the target memory), the RMAP packet is temporarily stopped.

⑤ When the transaction acknowledgment signal (*authorizeIn*) is returned from the user side, in the case of a write operation, the decoding of the data part begins. If the user does not allow access to memory at the address or data length specified in the RMAP packet, the user side returns a transaction reject signal (*rejectIn*). In this case a status code (*replyStatusIn*), which is an error code, is put in output and packet is discarded.

⑥ In the case of the write command, after decoding the RMAP data, the data is passed to the DMA module via *writeBuffer* signal and written to the memory through the internal bus. If there is a reply request, the target memory then replies with a status information. In the case of the read command instead, the DMA module reads data from the memory or register at the specified address through the bus. The read data is then passed to the RMAP Decoder module via *readBuffer* and the Reply packet is generated.

Any reply processing requires the further step:

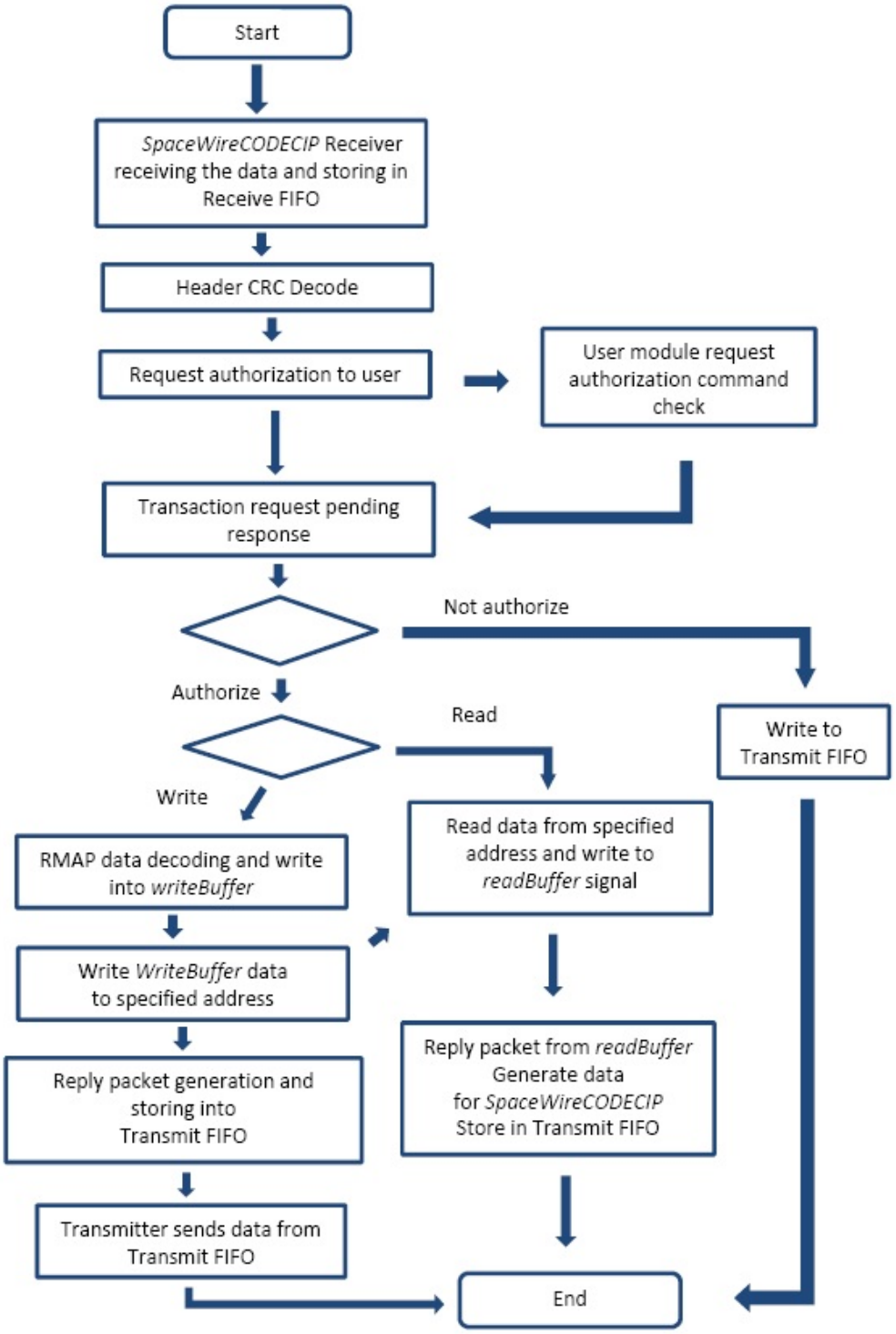


Figure 4.13: Dataflow representation of RMAP IP core working behaviour.

⑦ In the RMAP Decoder module, reply packet is generated from reply destination address field, status information field, reply data field, etc., and written to Transmit FIFO block of the SpaceWire CODEC. The data written into Transmit FIFO is then transferred by CODEC to the Transmitter block which converts it into SpW format and transmits it on the other side.

The graphical data flow representation of these steps is shown in Fig. 4.13.

4.4.3 FIFOs generation

For correct behaviour of RMAP block, the two sub-blocks *readBuffer* and *writeBuffer* play a crucial role since act as queues for I/O communication between RMAP core and target memory.

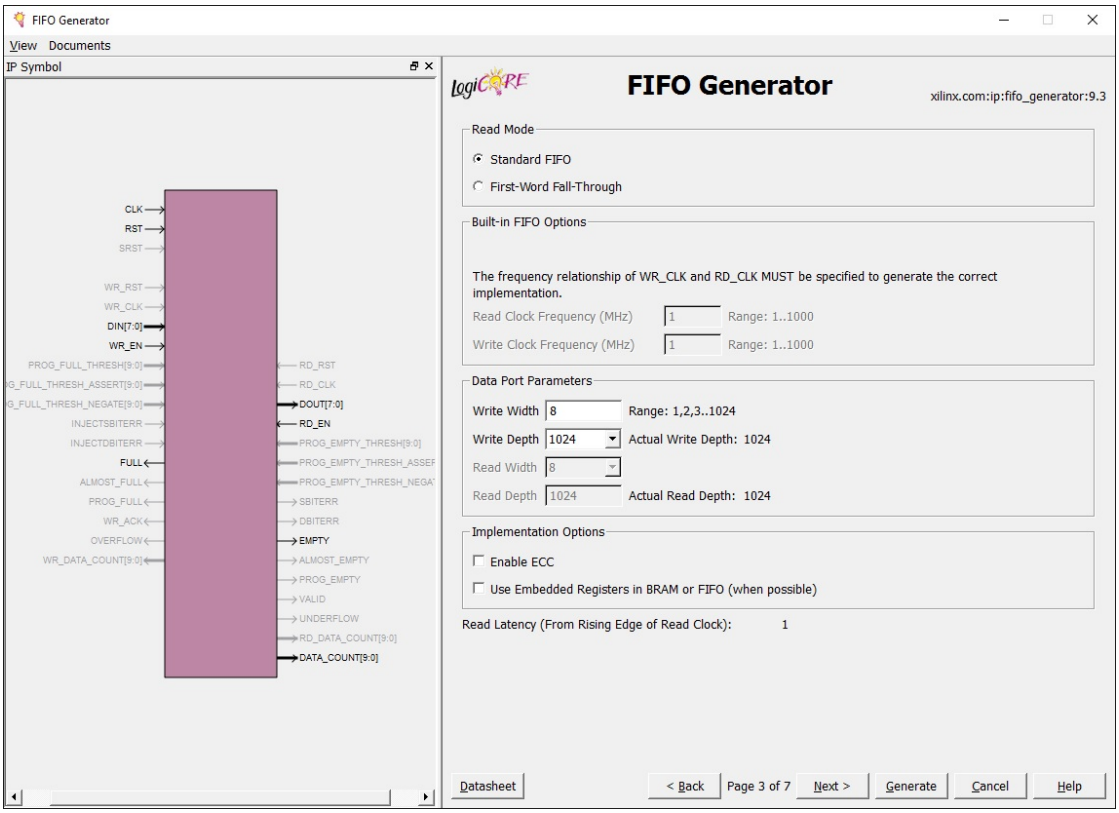


Figure 4.14: Generation of *FIFO8x2KXilinx* standard FIFO.

They are basically two 8-bit FIFOs which are 2 Kbytes wide and have been generated, as before, resorting to the ***Xilinx Core Generator*** tool. However, differently from the previous case, instead of using *Block memory generator*, the tool *FIFO Generator* has been employed as shown in Fig.4.14. The guided procedure

generated both netlist (*.vhdl*) and implementation (*.ngc*) files where also here only the last one has been used in the synthesys phase. No *.coe* file for predefined initialization of FIFO content was needed in this case.

4.5 Target RAM memory

The RMAP IP core has the main aim of performing reading and writing operation from or into the Target memory. In this thesis, a single port SRAM memory has been employed though in more advanced projects it can be replaced by register file or internal memory of more advanced IP cores such as *feature extractor and matcher* integrated circuits. In this case the SpaceWire network has the main aim of configuring this ICs, checking status and exchange informations between them and other components present inside the network (such as camera, OBC, etc...).

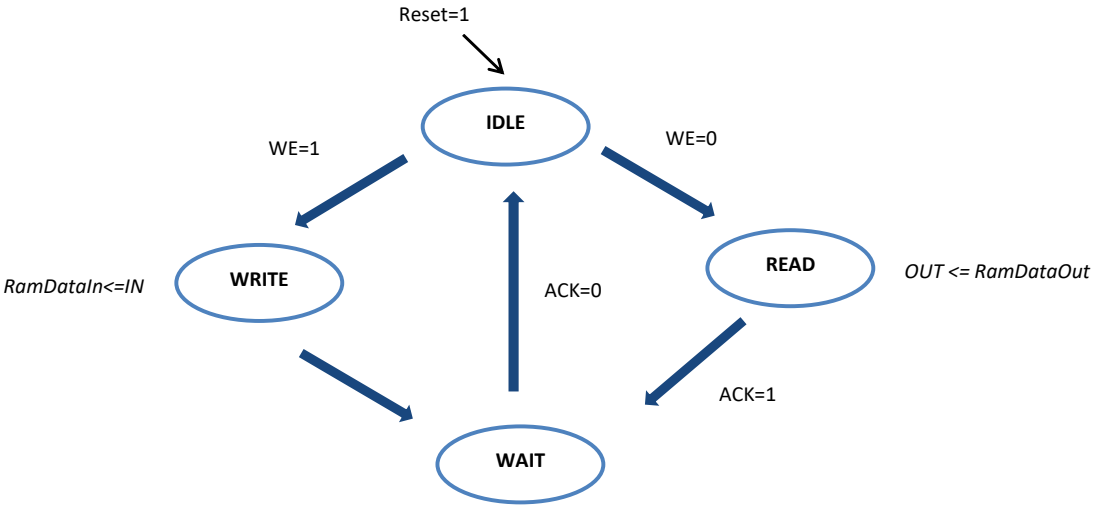


Figure 4.15: Graph of Target RAM memory Finite State Machine.

Target memory has been automatically generated by means of the *Xilinx Core Generator* tool, presenting the same features of the *RamXilinx32x256* therefore an address length of 8 bit and a data length of 32 bit. This means that the memory employed had a size of

$$M = 2^8 \cdot 32\ bit = 2^3 \cdot 2^5 \cdot 2^5\ bit = 2^{10}\ bytes = 1\ Kbyte$$

though the RMAP IP core with its interface is able to support a 32 bit address length so that memory size can be expanded up to

$$M = 2^{32} \cdot 32\ bit = 2^{30} \cdot 2^2 \cdot 2^5\ bit = 2^4\ Gbytes = 16\ Gbyte$$

Moreover, in order to stay compliant with RMAP IP core interface, a simple FSM has been built in order to manage the *acknowledge* and the *write enable* signals so that the writing operation into memory would follow the handshake protocol put in place by the RMAP core.

A graphical representation of the implemented FSM is shown in Fig. 4.15. As can be seen, the FSM manages the transition of the *Write enable* (WE) and *Acknowledge* (ACK) among the four different states. The machine, controlled by RMAP IP core, can act on the Target RAM input and output (*RamDataIn, RamDataOut*) and transit to idle by means of an asynchronous reset.

Chapter 5

Cores Simulations

In this chapter all the different IP cores simulations and test cases will be presented together with synthesis and implementation process targeting FPGA technology.

5.1 Codec simulation

Simulation and validation of the SpaceWire Codec core was carried out with the aid of *Mentor ModelSim* environment. The aim was the verification of the core capability to transmit and receive any kind of information in accordance with SpW component technical specification.

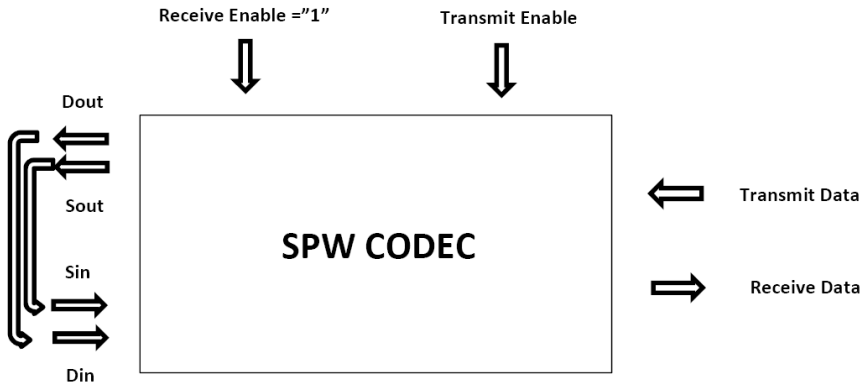
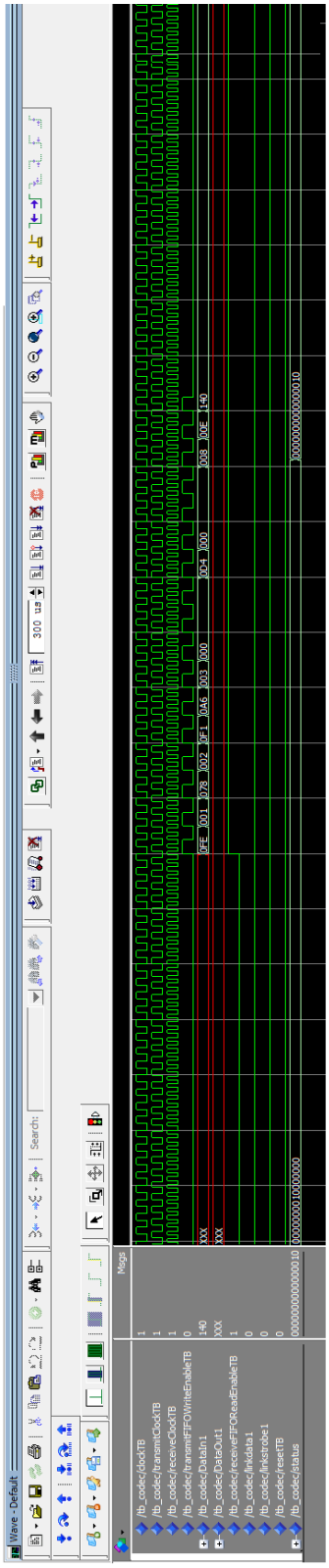
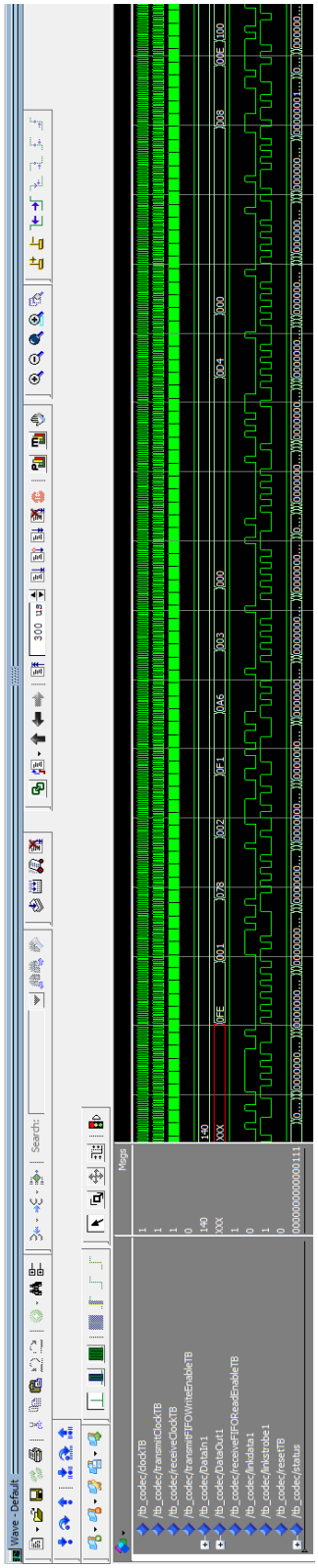


Figure 5.1: Architecture of first Codec simulation testbench.

A first simulation test bench was made up by a single Codec block which output signals were connected directly as input to the same block as shown in Fig.5.1.

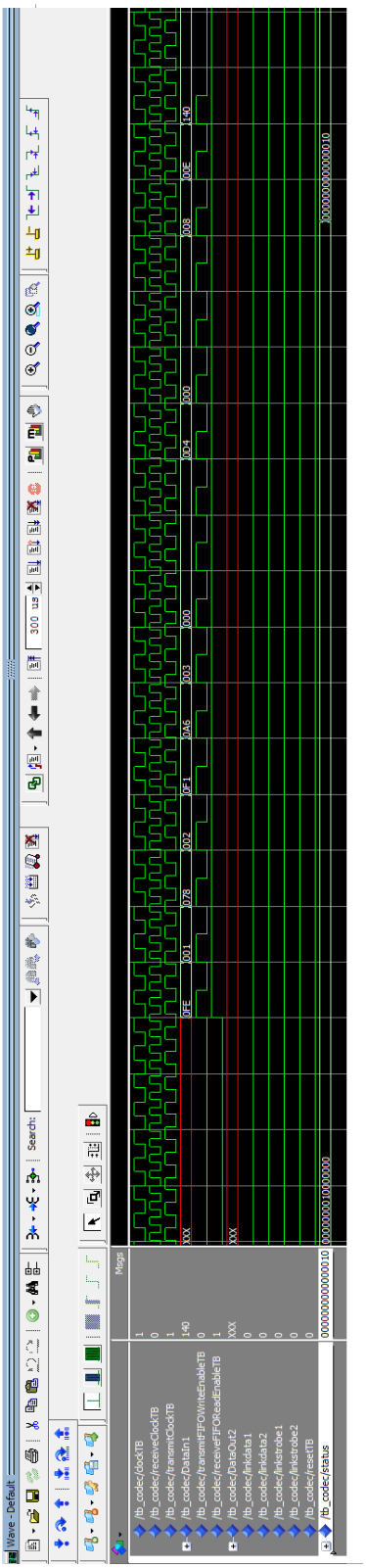


(a) SpaceWire packet fields at Codec input

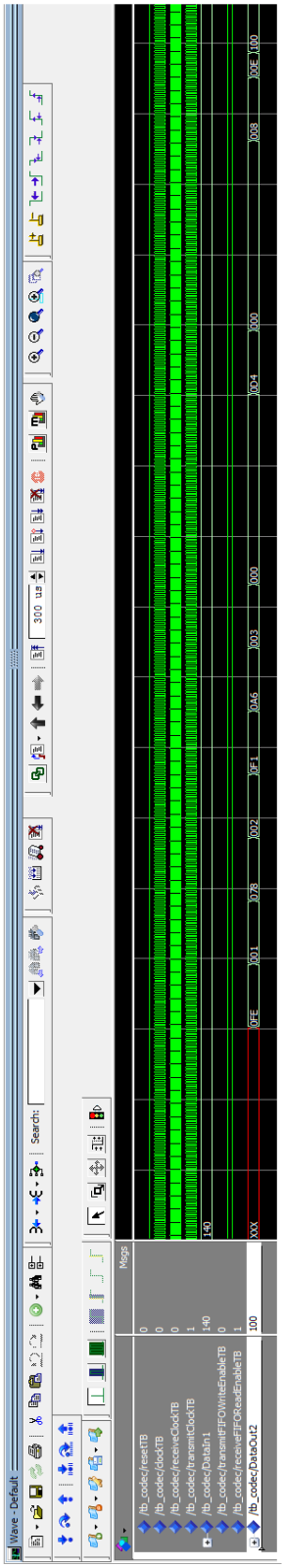


(b) SpaceWire packet fields at Codec output

Figure 5.2: First SpaceWire Codec IP core simulation



(a) SpaceWire packet fields at Codec input



(b) SpaceWire packet fields at Codec output

Figure 5.3: Second SpaceWire Codec IP core simulation

The testbench behaves like a wrapper arount the IP core to be tested as shown in Fig. 5.1. It stimulates the core with some input sets, collecting the results and analyzing them: specifically for Codec's test the successful condition is met if transmit data is equal to the received data due to the wrap around configuration previous imposed. The dataset used (number of data used in the test) was organized into a SpW packet and consisted of ten different data values (in Fig.5.3 data used is *08h*) which were reused in all the subsequent simulations.

Moreover, in all the simulations it is possible to see some red signals which are *XXX* waveforms. In ModelSim these *XXX* are present if multiple sources drive the same waveform (therefore the simulator is not capable of assigning to it an electrical level). They are produced because both the testbench and the core are driving the I/O in the simulation. In hardware instead, the electrical level is assigned by the the strongest driver (intended in terms of currents) therefore this effect is not an issue.

So it was verified that the single block was capable of transmitting and receiving correctly at the same time an equal piece of information and that the synchronization mechanisms was properly working. The simulation was successful. Screenshots of this first ModelSim simulation are shown in Fig. 5.2.

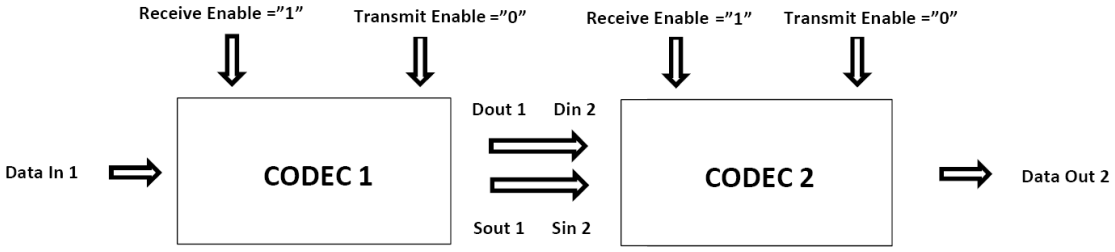


Figure 5.4: Architecture of second Codec simulation testbench.

Subsequently a second simulation testbench was set up to verify that the Codec IP core was able to work in transmission or reception mode at two different time intervals. The system employed two Codec blocks, one enabled as transmitter and the second one enabled as receiver as shown in Fig.5.4.
Also this second simulation has proved to be successful, as shown in Fig.5.3.

Finally, a third simulation, almost equal to the second one, has been performed using two codecs and allowing the contemporary transmission and reception of two different data packet in a cross-connection. Also this last simulation showed the

component performed correctly the intended operations.

The SpaceWire Codec core simulation was thus successfully carried out. Moreover, it gave a general understanding on the core working behavior since this core is used in the other two SpW IP Cores (Router and RMAP).

5.2 Router simulation

Also the SpaceWire router was tested using the HDL simulation environment (ModelSim). The simulation aimed at checking the Router behavior for both Path and Logical Addressing modes.

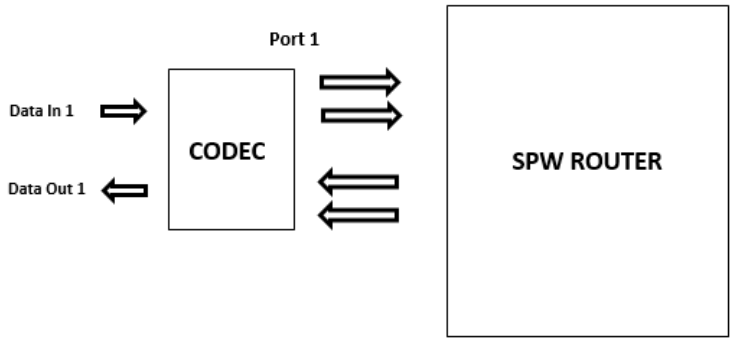


Figure 5.5: Architecture of Router testbench.

Simulation can be divided into three different parts:

- Router configuration via RMAP protocol or through external bus-access of the routing table;
- Application of SpaceWire packet to one input port to be routed to a different port using the two different addressing modes;
- Check of correct routing behavior.

In order to apply any data to the router a Codec is needed since router external interface is in SpaceWire format; therefore each of the 6 router ports is first connected to a SpW Codec as shown in Fig. 5.5 where this configuration is reported only for one port for sake of simplicity.

5.2.1 Routing table configuration

First phase is generally performed by the on-board software which has to configure the entries of the routing table in order to establish a link between two actors inside the logical network. Configuration of the router is performed by a RMAP 32 bit acknowledge-requested write operation via configuration Port 0.

The write operation has the aim of associating a logical address to a port number. In particular, as explained previously, all bits of a valid logical address table entry must be reset except for the bit corresponding to the port on the router that the logical address corresponds to.

RMAP packet bytes	**RMAP packet fields**
00	Leading byte is 00h to route the packet to port 0.
FE	Target logical address equal to Port 0 default logical address
01	Protocol ID
78	Instruction for a reply command packet
02	RMAP Key router default value
F1	Initiator logical address set arbitrary.
00 00	Transaction Identifier (MS byte first)
00	Extended address
00 00 03 F4	Address of location to write into
00 00 04	Data length (expressed in bytes)
2E	Header CRC code
00 00 00 02	Data field
E3	Data CRC code
40	End Of Packet

Table 5.1: RMAP packet fields for routing table dynamic configuration.

Let's suppose for example we want to connect the camera which is at the logical address *FDh* to port 1 of the router. Then we would have to write in the routing table at the logical address *FDh* (in decimal 253) setting to 1 only the bit corresponding to port 1. Register 253 of the routing table, as can been seen from routing table memory map in [25], corresponds to address *0x03F4* while 32 bit word to be written correspond to *0x00000002* (position 0 corresponds to Port 0).

An RMAP non-incrementing verified-write with no reply command packet to have this is behaviour (in hexadecimal bytes) is:

00 FE 01 78 02 $F1$ 00 00 00 00 00 03 $F4$ 00 00 04 $2E$ 00 00 00 02 $E3$ 40

Notice that since each input is on 9 bit, each byte has to be preceded by the control character (for normal data 0). Only for the end of packet EOP (in hexadecimal 40) this doesn't hold so we have a 1 as control character therefore the concatenation will produce a *140h* input data as visible in ModelSim screenshots in Fig. 5.7.

In Table 5.1 single RMAP packet fields are explained more in detail.

The expected replied packet, at the output of the same port, which indicate successful writing inside the Routing table is: $F1$ 01 38 00 FE 00 00 $D8$

This has been observed in the simulation as visible in Fig.5.7. Therefore the simulation has been successful since, once applied the input packet previously described, the output answer has been the one expected. It is also possible to see the content of the routing table and check its configuration, by reading this memory through the external User-access bus: the reading cycle to achieve this can be found in [25].

A screenshot of the behaviour of the RMAP decoder inside Port 0 is instead shown in Fig.5.6.

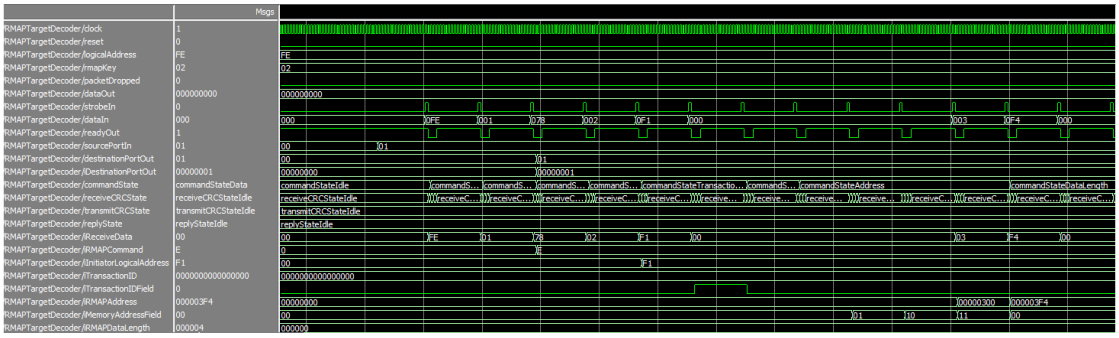
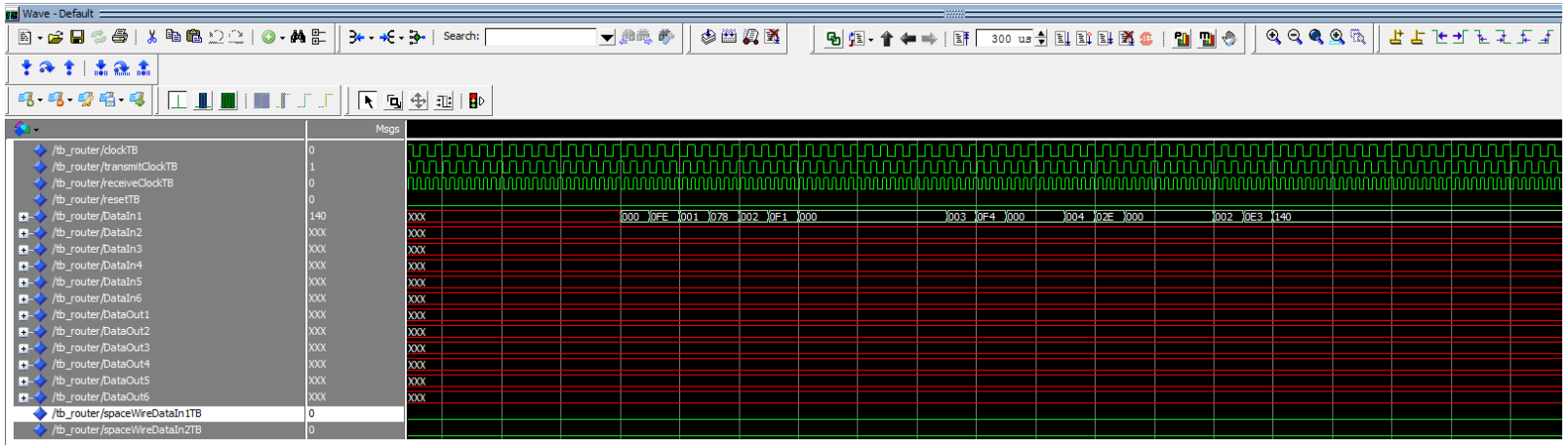
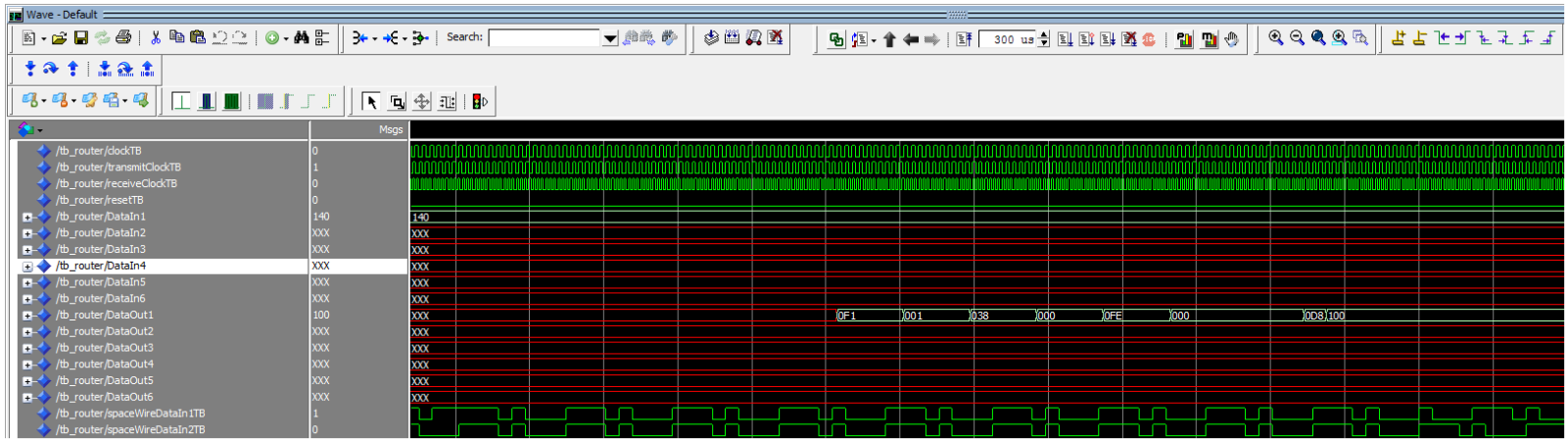


Figure 5.6: Snapshot of the RMAP Decoder present inside Router Port 0.

It is possible to see here, that each field composing the SpaceWire packet is recognized thanks to the RMAP State Machine whose dataflow has been explained in the previous chapter. In this last screenshot, it is possible to see that no *XXX* signal is present. In Fig. 5.7 this is instead clearly visible because both testbench and UUT (the router) are driving inputs/outputs.



(a) SpaceWire packet fields at Router Port 1 input



(b) SpaceWire packet fields at Router Port 1 output

Figure 5.7: SpaceWire Router core simulation and routing table configuration.

5.2.2 Path addressing mode

Subsequently, one packet has been routed to another port by simply applying as leading byte the destination port number (path addressing) as visible in Fig. 5.8.

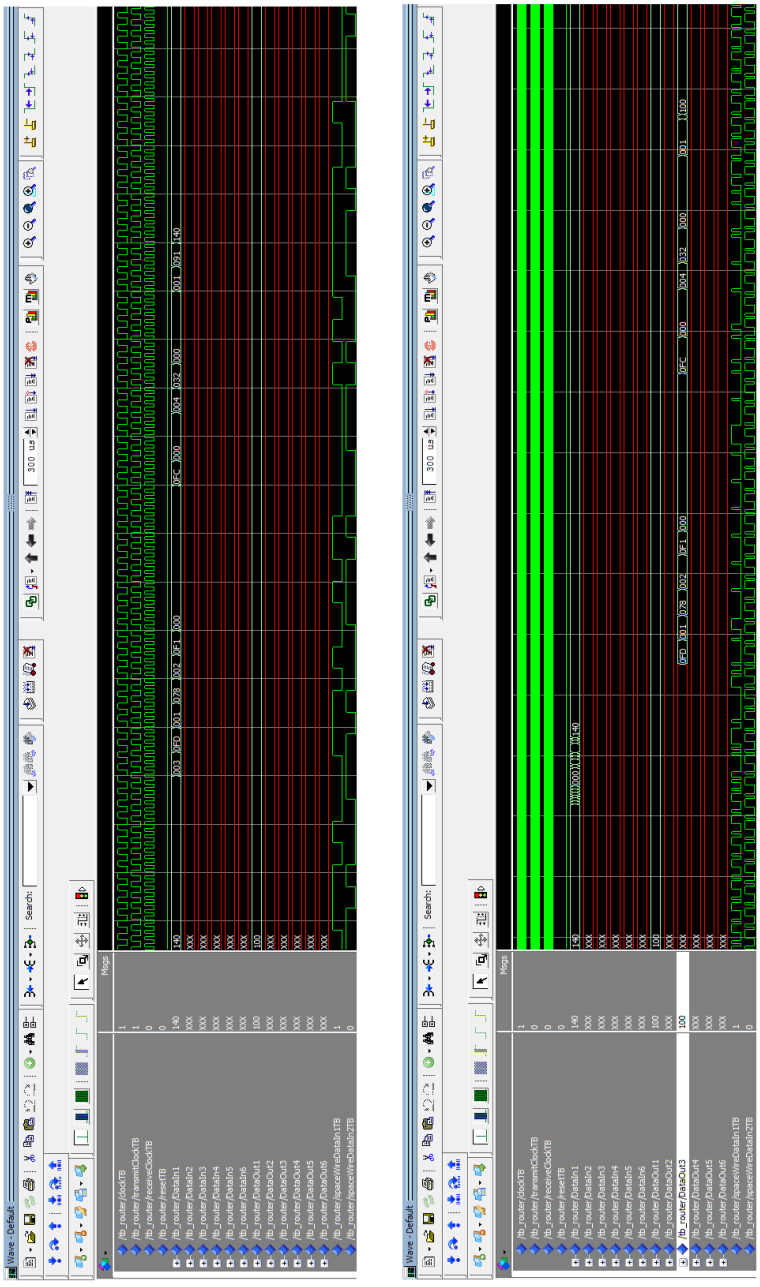
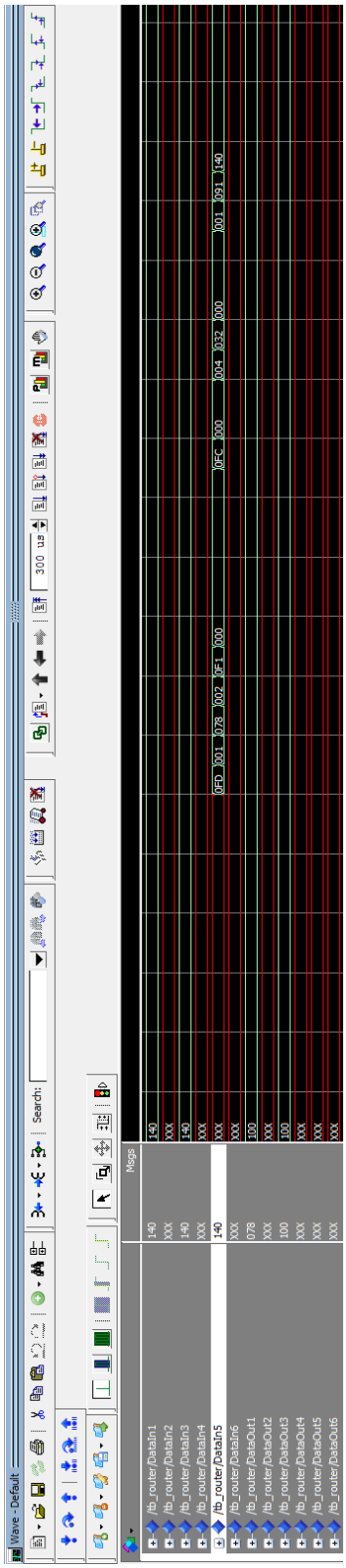
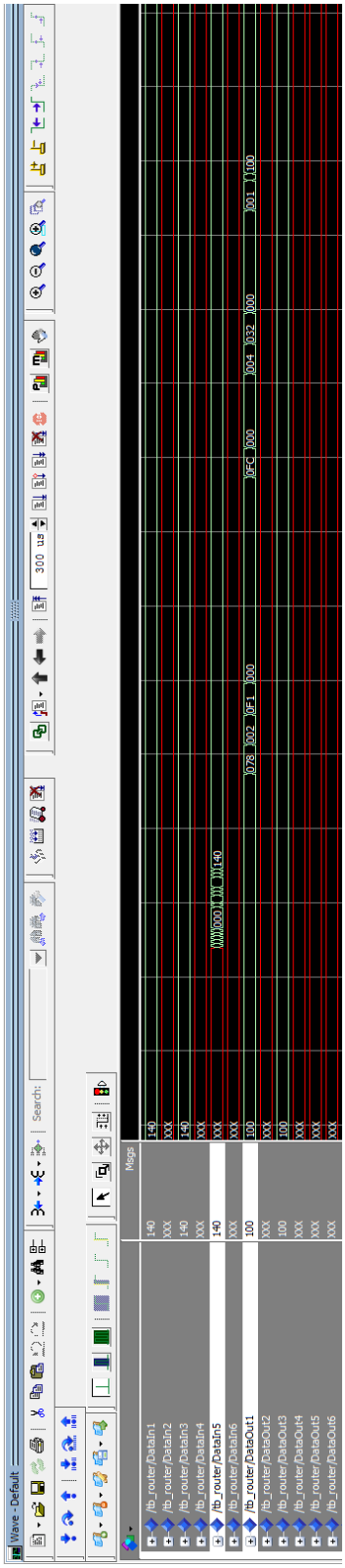


Figure 5.8: SpaceWire Router simulation for path addressing mode



(a) SpaceWire packet fields at Router Port 5 input



(b) SpaceWire packet fields at Router Port 1 output

Figure 5.9: SpaceWire Router core simulation in logical addressing mode.

5.2.3 Logical addressing mode

An example of logical addressing can be instead seen from Figure 5.9.

Once having assigned in the routing table to Port 1 logical address *FDh* a packet is sent from Port 5 to Port 1 specifying as leading byte the target logical address. The second byte is equal to *01* which is the target port number so that this can be seen as a logical addressing mode joint with path one. Actually, configuration via software is very likely using this double approach which can be seen as logical addressing strengthened by path one.

From the picture 5.9 b), we can clearly see that packet is correctly routed to the output port and the first two bytes used for routing purposes are removed. Of course the complete validation of the Router required all ports were tested using both path and logical addressing modes. Results of these tests met completely the expectations therefore the component prove to work compliant with ECSS standards and technical specifications.

5.3 RMAP simulation

Similarly, to the previous two cores, also the RMAP one was tested using ModelSim. A test bench around the core was created providing correct signal stimuli to observe the core answer to an RMAP command packet as visible in Fig. 5.10.

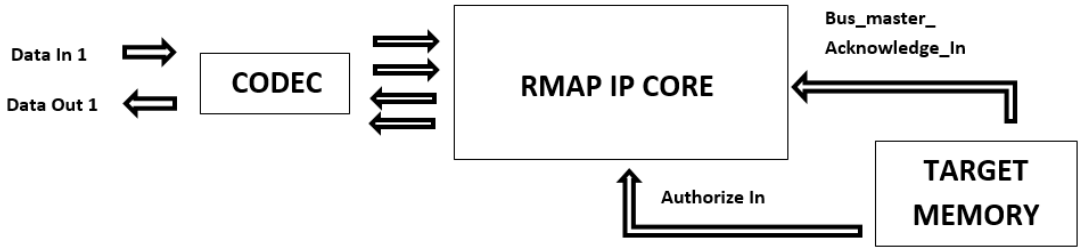
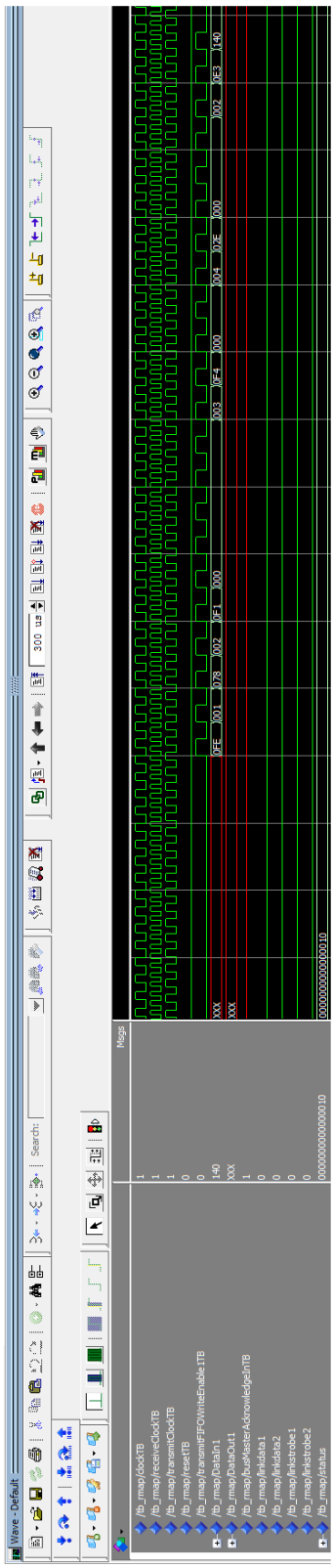
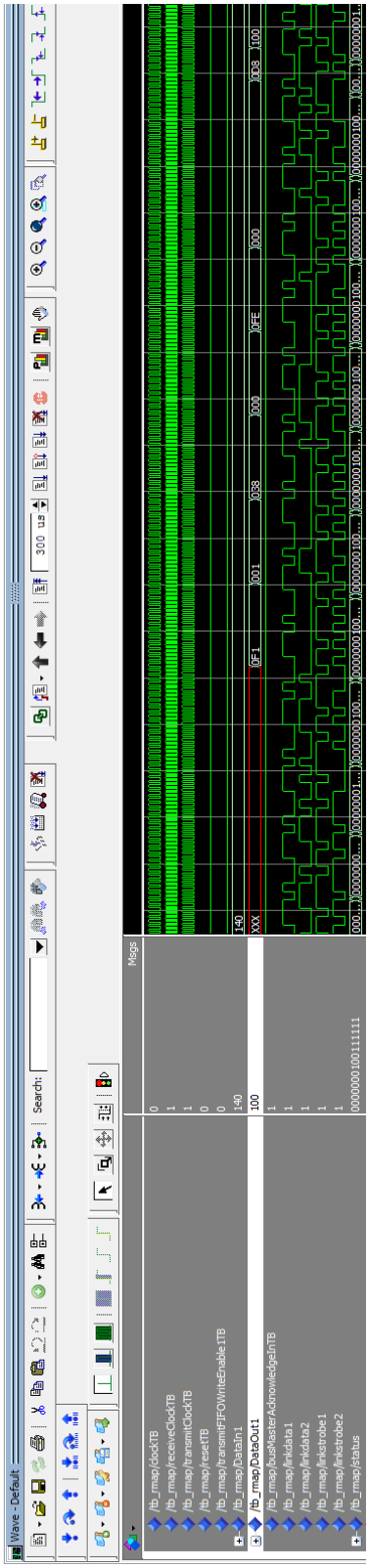


Figure 5.10: Architecture of RMAP simulation testbench.

Such packet was given, via a Codec block attached to the external SpW interface. Authorization signal were given by target memory also present inside the testbench. An important part for the inclusion of memory inside the simulation was use of the *UNISIM* library generated from Xilinx ISE project.



(a) SpaceWire packet fields at RMAP testbench input



(b) SpaceWire packet fields at at RMAP testbench output

Figure 5.11: SpaceWire RMAP core plus target memory simulation.

In picture 5.11 a), we can see that packet sent at the input is similar to the one used for router configuration described in Tab.5.1: it corresponds to a write-with-no-reply RMAP packet to assess the capability of recognizing all RMAP fields and setting the bus accordingly. Target logical address was defined as before as *FEh* while initiator logical address has been defined arbitrarily as *F1h*. The value to be written inside Target memory at location of address *03F4* is *02*.

At this point bus access starts and writing operation has been performed by means of the DMA controller. The system is then stuck in the *Wait_ bus_ Access_ End* state until the downstream block sets all the internal memory registers using bus delivered information. The memory then asserts the *Bus_ master_ Acknowledge_ in* signal to terminate bus access. In conclusion, RMAP block responds to the outside with a reply packet, as experienced in the router validation, which acknowledge the correctly writing in the downstream memory as shown in the picture 5.11 b).

Once having written inside the memory, to complete RMAP verification also reading mode operation was checked by means of a specific purpose RMAP packet. Results matched expectations proving component worked correctly in all possible situations.

Chapter 6

Experimental results

In this chapter a description of the experimental implementation on real hardware of the project described in this thesis will be presented together with the test procedures and equipment adopted which led to final results.

6.1 Hardware implementation

In order to implement the project and experimentally validate it, it has been used the development board GR-CPCI-XC4V produced by *Aeroflex Gaisler* in collaboration with *Pender Electronic Design*. Its architecture has already been explained in chapter 3; a picture of it is instead shown in Fig. 6.1 ([8]).

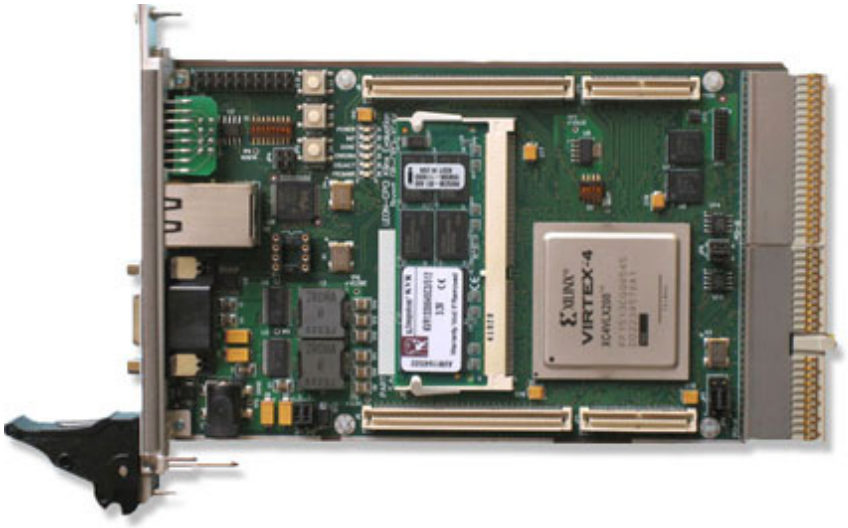


Figure 6.1: Picture of the Gaisler GR-CPCI-XC4V board employed.

6.1.1 Pin assignment

To connect the design implemented on the board with the outside environment by means of SpaceWire cables, it has been used the mezzanine board *GR-SER2-SPW4* shown in Fig.6.2 (images and technical features taken from [17]).

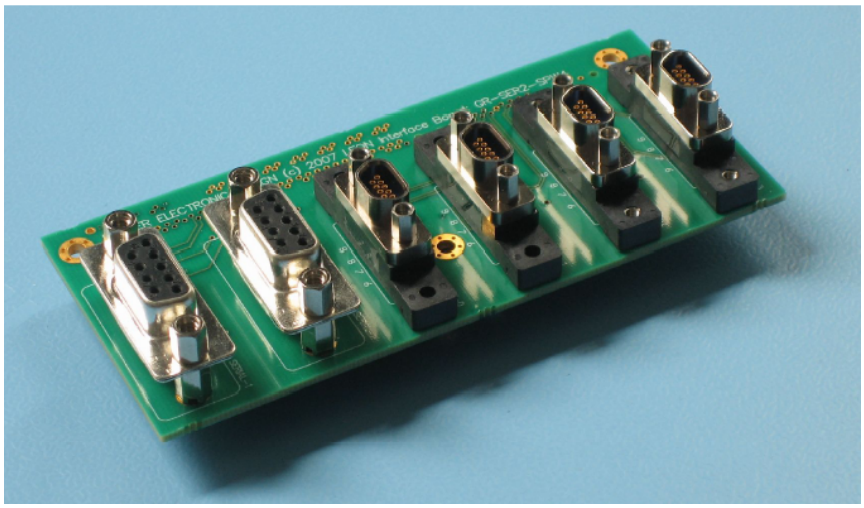


Figure 6.2: Picture of the SpaceWire mezzanine board *GR-SER2-SPW4*.

As visible from the picture the mezzanine provides two serial D-type RS232 (not employed in this project) and four SpaceWire (LVDS) electrical interfaces. The mezzanine has been mounted on the development board J10 connector so that SpaceWire signals were connected to GENIO expansions signals.

At this point the pin assignment phase was performed. It practically consisted into linking together the I/O of the project (basically the router ports) to the SpW mezzanine outputs by correctly assigning the FPGA pins to the GENIO signals. In order to do so, the mezzanine and GR-CPCI-XC4V board's schematics have been employed which can be found in [18] and [17]. In this process, since SpW protocol is a differential one, only ***positive signals*** of Data and Strobe have been considered. GENIO signals corresponding to SpW I/Os visible in Fig.6.3 have been connected to pins present on one of FPGA I/O banks by means of assignment directives inside the *User Constraint File* (*.ucf*).

In this process also the clock input has been assigned to the on-board oscillator pin (P20) while for reset it has been preferred to employ an internal signal (simply obtained using a counter resetting the system for 1000 clock cycles). An extract of the *.ucf* file performing this operation is reported below:

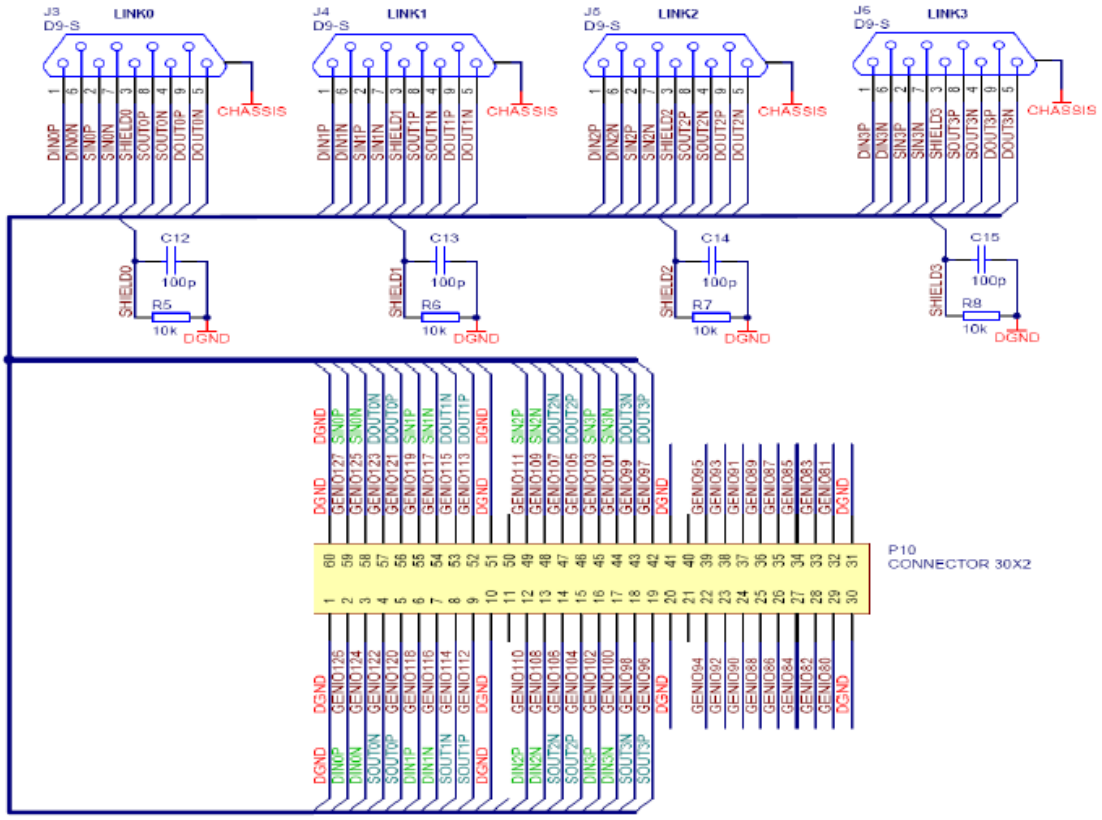


Figure 6.3: Electrical schematic of mezzanine board with SpW and GENIO signals

```
NET "CLK"        LOC = "P20"    | IOSTANDARD=LVTTL;

NET "spaceWireDataIn1"    LOC = "AA34"; # | IOSTANDARD = LVDS_25;
NET "spaceWireStrobeIn1"  LOC = "AA31"; # | IOSTANDARD = LVDS_25;
NET "spaceWireDataOut1"    LOC = "Y29"; # | IOSTANDARD = LVDS_25;
NET "spaceWireStrobeOut1" LOC = "AC35"; # | IOSTANDARD = LVDS_25;

NET "spaceWireDataIn2"    LOC ="AC38"; # | IOSTANDARD = LVDS_25;
NET "spaceWireStrobeIn2"  LOC ="AE34"; # | IOSTANDARD = LVDS_25;
NET "spaceWireDataOut2"    LOC ="AD31"; # | IOSTANDARD = LVDS_25;
NET "spaceWireStrobeOut2" LOC ="AE39"; # | IOSTANDARD = LVDS_25;

NET "spaceWireDataIn3"    LOC ="AF38"; # | IOSTANDARD = LVDS_25;
NET "spaceWireStrobeIn3"  LOC ="AG37"; # | IOSTANDARD = LVDS_25;
NET "spaceWireDataOut3"    LOC ="AB27"; # | IOSTANDARD = LVDS_25;
NET "spaceWireStrobeOut3" LOC ="AJ39"; # | IOSTANDARD = LVDS_25;

NET "spaceWireDataIn4"    LOC ="AK38"; # | IOSTANDARD = LVDS_25;
```

```
NET "spaceWireStrobeIn4" LOC ="AK36"; # | IOSTANDARD = LVDS_25;
NET "spaceWireDataOut4"   LOC ="AM36"; # | IOSTANDARD = LVDS_25;
NET "spaceWireStrobeOut4" LOC ="AJ35"; # | IOSTANDARD = LVDS_25;
```

6.1.2 Clock management

Clock has been dealt in a specific way. In fact on the board a 50 MHz clock was available while clock frequencies used in the thesis design are reported in Tab. 6.1. Therefore a *Digital Clock Manager* (DCM) inside the FPGA has been employed.

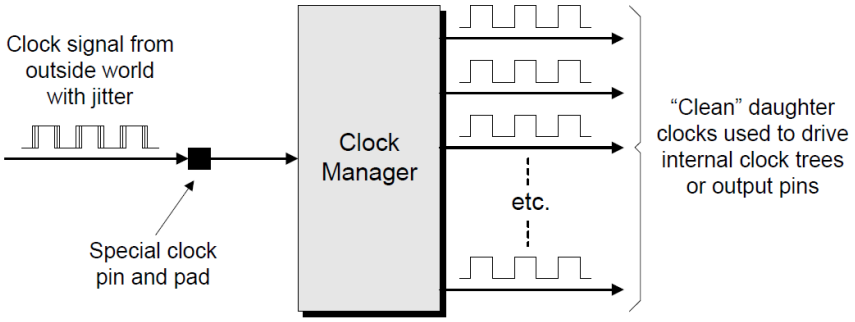


Figure 6.4: FPGA Clock different frequencies generation using a DCM.

Figure 6.4 (taken from [16]) describes basic principle of a DCM. The clock manager can be seen as a predefined block which generates some daughter clocks out of the main one coming from the external pin. These daughter clock can be hence used to drive the internal clock trees or external pins with an overall significant reduction of skew and jitter. In fact as we know, though generated by means of an high-precision crystal oscillator, clock signal is never an ideal squarewave and can be affected directly out of the external pins/pads of such problems. Digital Clock Manager instead can clean the signal from jitter by means of internal feedback mechanism while skew is removed through tree or H distribution schemes present inside the chip.

Moreover the DCM can implement phase shifters or for example frequency synthesizers which multiply and divide original signal producing a clock whose frequency is exactly the one needed inside the project. Such last feature has been employed in this thesis project.
Fig.6.5 shows how a DCM has been generated using *Xilinx Core Generator*. The guided procedure requested to insert the input clock frequency and allowed to set the freedback mechanism as internal or external (first option was used). Then

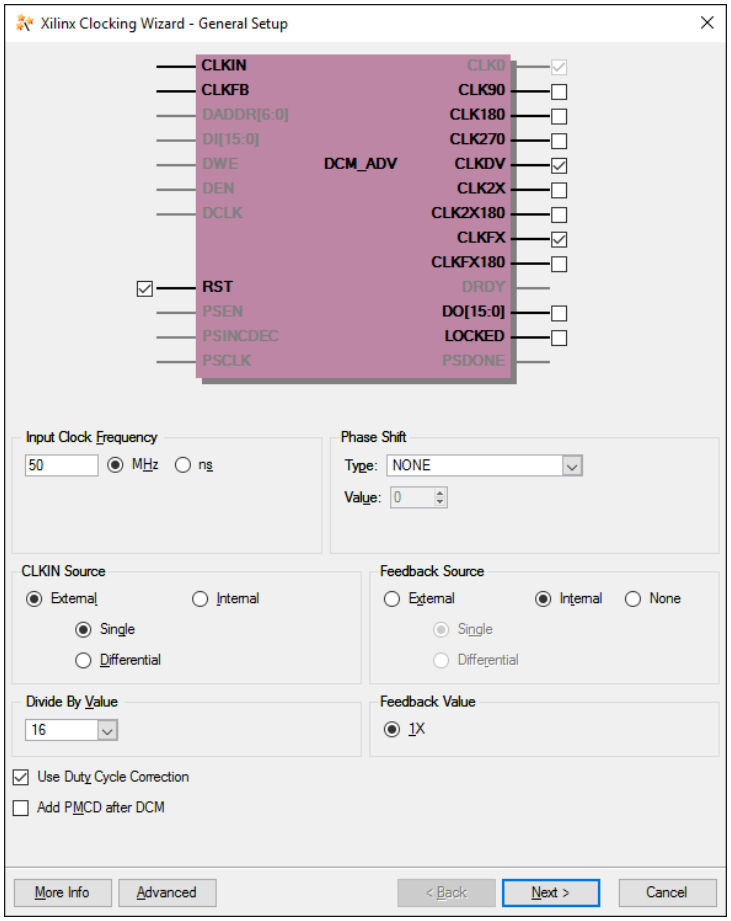


Figure 6.5: Generation of DCM called *Clockbuffer*.

Transmit Clock frequency was obtained by dividing the input clock by a factor of 16. *Receive Clock* was instead created using a frequency synthesizer with a multiplier factor $M = 6$ and a division factor of $D = 9$.

The guided procedure ended with the generated netlist inside a *.vhdl* file and the constraints *.ucf* file. Both had to be included inside the project to correctly drive the synthesis and implementation process as described in previous chapter.

6.1.3 FPGA synthesis and implementation

As previously stated, CAD tool used for FPGA development has been **Xilinx ISE** targeting a Virtex-4 FPGA (specifically the *XC4VLX200* model, the largest of this family). The design flow, already explained in Fig. 3.6, has consisted into several steps. After IP cores (whose block diagrams have already been shown in Fig. 4.2,

4.7 and 4.12) analysis, reconstruction and successful testing (both separately and altogether) the first phase has been *Logic Synthesis* where a first resources estimation was given as shown in Fig. 6.6.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	7200	89088	8%
Number of Slice Flip Flops	7218	178176	4%
Number of 4 input LUTs	13458	178176	7%
Number of bonded IOBs	153	960	15%
Number of FIFO16/RAMB16s	22	336	6%
Number of GCLKs	4	32	12%
Number of DCM_ADVs	1	12	8%

Figure 6.6: Device utilization estimation after synthesis phase.

Synthesis phase has ended also with a preliminary performance analysis: a maximum frequency of 132 MHz has been estimated. Subsequently the tool proceeded with the *Mapping* , then to the *Placement* step (slices arrangment) and finally it passed to the *Routing* where it dealt with connections. At this point the resource estimation is final one and can be seen fig. 6.7.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	7,178	178,176	4%	
Number of 4 input LUTs	11,534	178,176	6%	
Number of occupied Slices	8,403	89,088	9%	
Number of Slices containing only related logic	8,403	8,403	100%	
Number of Slices containing unrelated logic	0	8,403	0%	
Total Number of 4 input LUTs	13,370	178,176	7%	
Number used as logic	11,497			
Number used as a route-thru	1,836			
Number used for Dual Port RAMs	16			
Number used as Shift registers	21			
Number of bonded IOBs	153	960	15%	
Number of BUFG/BUFGCTRLs	3	32	9%	
Number used as BUFGs	3			
Number of FIFO16/RAMB16s	22	336	6%	
Number used as RAMB16s	22			
Number of DCM_ADVs	1	12	8%	
Average Fanout of Non-Clock Nets	3.50			

Figure 6.7: Device utilization summary after Mapping and Place&Route phase.

As it is possible to see from Fig. 6.7 only the 10% of the device has been used.

The complete design occupies only a small portion of the FPGA which, being the VLX200 model, comprises 200000 logic cells: therefore either additional cores can be implemented together with current design (such as feature extracting and matching cores) or a smaller FPGA can be employed.

6.1.4 Timing constraints

The very last stage of FPGA implementation consists into *Post P&R static timing analysis* where there is the design timing checks using informations that are technology dependent. The CAD tool then annotates the delay informations inside a *Standard Delay Format* file (*.sdf*).

In all the previous steps and in particular in this last stage a very important element is the user assignment of **timing constraints** so that implemented design matches the desired performances. These constraints refers mainly to clock signals and are used by the synthesizer and router tool during their operations and optimization. Of course, constraints have to be coherent with the possible achievable performances otherwise in the last step of FPGA design flow, an error will be raised for a not met timing constraint. For the project described in this thesis the following timing settings have been used:

System Clock	50 MHz
Transmit Clock	3 MHz
Receive Clock	33 MHz

Table 6.1: Timing settings for complete FPGA project.

Though the maximum achievable performance after P&R is of **87.27 MHz**, the implementation was runned at a lower frequency due to testing equipment constraints. In fact, testing rack transmission drivers were bound to transmit at 3 MHz: it was therefore necessary to lower also other clock frequencies to values described in Tab. 6.1 as suggested in Codec's datasheet [23].

Timing constraints have been passed to the CAD tool by means of an edit file that for the Xilinx development environment is called *User Constraints file* (*.ucf*). An extract of it is reported below:

```
NET "CLK" PERIOD = 20.000 ; #board clock at 50MHz

NET "clock" TNM_NET = "SYS_CLK";
NET "transmitClock"TNM_NET = "TX_CLK";
```

```
NET "receiveClock" TNM_NET = "RX_CLK";

TIMESPEC TS_SYS_CLK_to_TX_CLK = FROM "SYS_CLK" TO "TX_CLK" TIG;
TIMESPEC TS_TX_CLK_to_SYS_CLK = FROM "TX_CLK" TO "SYS_CLK" TIG;
TIMESPEC TS_TX_CLK_to_RX_CLK = FROM "TX_CLK" TO "RX_CLK" TIG;
TIMESPEC TS_RX_CLK_to_TX_CLK = FROM "RX_CLK" TO "TX_CLK" TIG;
TIMESPEC TS_SYS_CLK_to_RX_CLK = FROM "SYS_CLK" TO "RX_CLK" TIG;
TIMESPEC TS_RX_CLK_to_SYS_CLK = FROM "RX_CLK" TO "SYS_CLK" TIG;

TIMESPEC TS_clk = PERIOD "SYS_CLK" 20 ns HIGH 50%;
TIMESPEC TS_txclk = PERIOD "TX_CLK" 333 ns HIGH 50%;
TIMESPEC TS_rxclk = PERIOD "RX_CLK" 33 ns HIGH 50%;
```

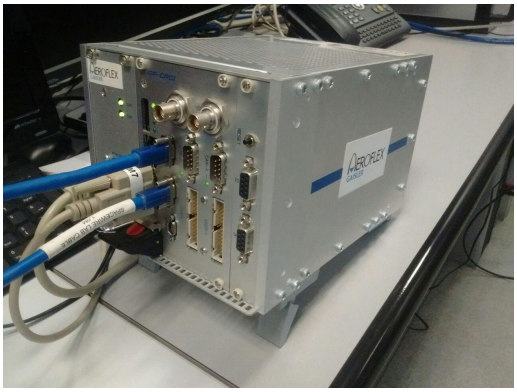
6.2 Hardware testing

6.2.1 Testing setup

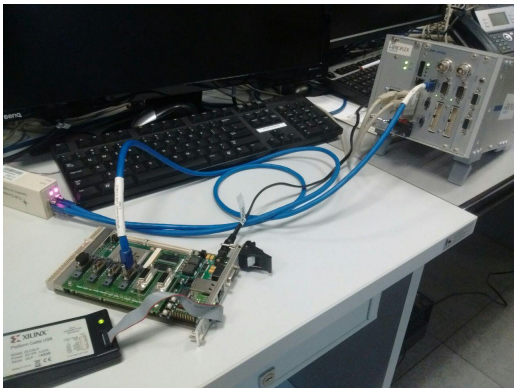
At this point a test on the implemented design was carried out. The test aim was to verify whether the design had been correctly implemented on hardware and if the development board was able to communicate using SpaceWire protocol.

In order to perform the test a Hardware-in-the-loop system was build which ,apart from the development board, was composed of:

- a **Leon 3 processor** which was used to feed the board with test vectors (basically SpW packets to perform memory operations via RMAP) and collect the results coming back, checking and displaying them on video. In order to do so it was used a GR-RASTA present inside the Thales Alenia Space *Avionics Laboratory* shown in Fig.6.8 a). GR-RASTA is a fast Leon prototyping platform produced by the Aeroflex Gaisler employing Actel FPGAs which are also pre-programmed with interface drivers for SpaceWire I/Os.
- two **SpaceWire cables** (the ones in blue in Fig.6.8) which connected the SpW ports of RASTA with the two SpW connectors of the mezzanine (attached on the J10 connector of the board). In the first part of the test, cables connected RASTA and the board passing through the link analyzer to observe how communication was put in place.
- **Link analyzer** which is an instrument produced by *STAR-Dundee* used to analyze the transmission of a SpaceWire link at both signal and packet levels. It is also capable of detecting an user-defined trigger event or error and providing an estimation of transmission rate at both ends of the link.



(a) Aeroflex Gaisler GR-RASTA.



(b) Board test using link analyzer.

Figure 6.8: Test Hardware setup (by courtesy of Thales Alenia Space).

Of course as part of the set-up a computer present inside the laboratory was used to program the GR-RASTA and to load inside processor memory the test program written in C, compiled using *Eclipse IDE* and cross-compiled using *Cygwin*. Moreover, the link analyzer was connected to the PC and the relative proprietary software was used to configure the instrument and show the packets transmitted.

6.2.2 Test procedure and results

The test was performed into two different phases:

1. At first it was checked that the boads was able to establish a connection with the GR-RASTA. SpaceWire link initializazion was needed as a basis for subsequent packet transmission. In this first part the link analyzer was inserted in between to observe the handshake protocol previously described having set the codecs implemented on the board into the Auto-start mode.

2. Then the actual testing was performed. In this part the SpaceWire link analyzer was not used anymore since it proved to slow down the transmission. Moreover, the two SpW cables connected two ports of the mezzanine with two ports of the GR-RASTA: in fact one port was used for input packets from RASTA towards the board while any reply from the design was routed out of the second port in order to check router capabilities.

   The test program applied was meant to emulate VHDL simulations already performed in ModelSim environment. For this reason, it first configured the router dynamically by assigning to the two used SpW ports two different logical addresses. It then performed two RMAP writes, one was a number and the other one some text characters, at two different memory locations. The program finally checked what was written with two RMAP read operations displaying onto the screen the result.

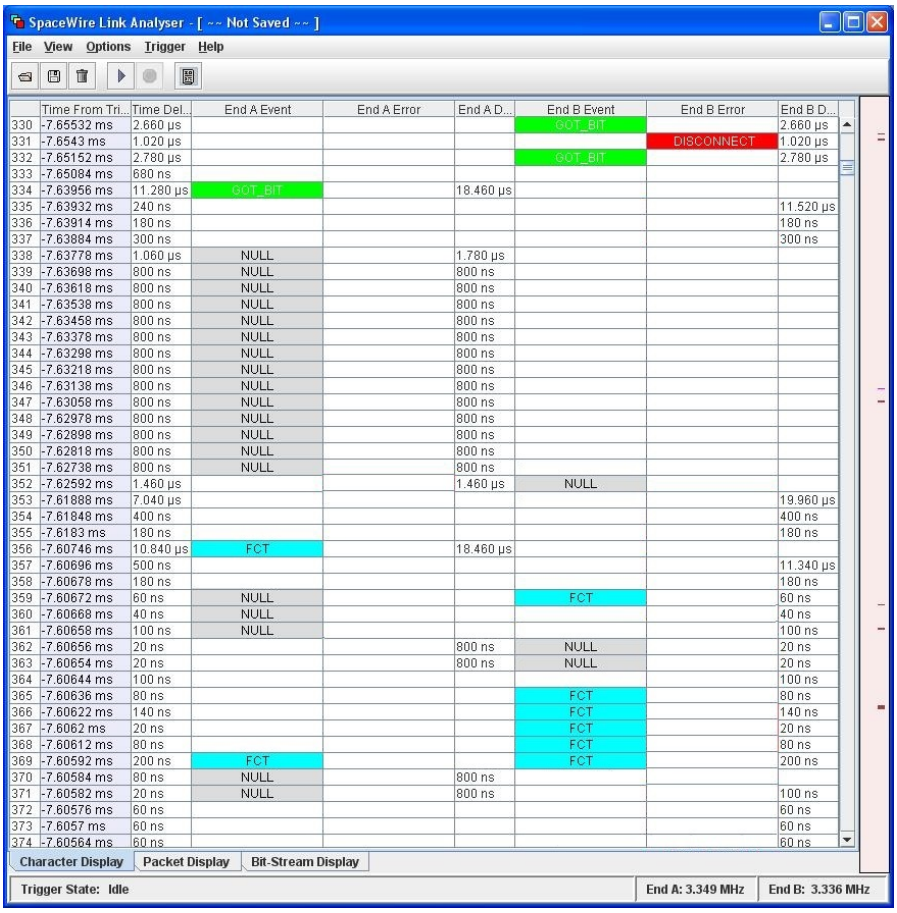
   In this way all different cores of the design were checked to properly work: codecs were constantly used as interface with external world while Router was tested by first configuring the routing table and then by writing and reading in the memory using different ports by means of the RMAP core.

Figure 6.9: Link analyzer when a SpaceWire connection is established

The first part of the test concerned the link initialization. The link analyzer was inserted between the board and the GR-RASTA to observe packets exchanged during this phase and in particular to check if the design implemented on the board responded according to the Auto-start mode.

At first, a frequency mismatch in the transmission rates of the two actors impaired the initializazion: GR-RASTA drivers in fact were were bound to transmit at a frequency of about 3 MHz which was lower than the one of the board. After lowering down the design's *Trasmit clock* at that rate, link initialization was successfully performed. At this point, link analyzer showed transmission could take place by changing lights from red (Fig. 6.8 b)) to green as shown in Fig. 6.9.

Figure 6.10 shows more in details packets exchanged during the initialization phase.



	Time From Tri...	Time Del...	End A Event	End A Error	End A D...	End B Event	End B Error	End B D...
330	-7.65532 ms	2.660 µs				GOT_BIT		2.660 µs
331	-7.6543 ms	1.020 µs					DISCONNECT	1.020 µs
332	-7.65152 ms	2.780 µs				GOT_BIT		2.780 µs
333	-7.65084 ms	680 ns						
334	-7.63956 ms	11.280 µs	GOT_BIT		18.460 µs			
335	-7.63932 ms	240 ns						11.520 µs
336	-7.63914 ms	180 ns						180 ns
337	-7.63884 ms	300 ns						300 ns
338	-7.63778 ms	1.060 µs	NULL		1.780 µs			
339	-7.63698 ms	800 ns	NULL		800 ns			
340	-7.63618 ms	800 ns	NULL		800 ns			
341	-7.63538 ms	800 ns	NULL		800 ns			
342	-7.63458 ms	800 ns	NULL		800 ns			
343	-7.63378 ms	800 ns	NULL		800 ns			
344	-7.63298 ms	800 ns	NULL		800 ns			
345	-7.63218 ms	800 ns	NULL		800 ns			
346	-7.63138 ms	800 ns	NULL		800 ns			
347	-7.63058 ms	800 ns	NULL		800 ns			
348	-7.62978 ms	800 ns	NULL		800 ns			
349	-7.62898 ms	800 ns	NULL		800 ns			
350	-7.62818 ms	800 ns	NULL		800 ns			
351	-7.62738 ms	800 ns	NULL		800 ns			
352	-7.62592 ms	1.460 µs			1.460 µs	NULL		
353	-7.61888 ms	7.040 µs						19.960 µs
354	-7.61848 ms	400 ns						400 ns
355	-7.6183 ms	180 ns						180 ns
356	-7.60746 ms	10.840 µs	FCT		18.460 µs			
357	-7.60696 ms	500 ns						11.340 µs
358	-7.60678 ms	180 ns						180 ns
359	-7.60672 ms	60 ns	NULL			FCT		60 ns
360	-7.60668 ms	40 ns	NULL					40 ns
361	-7.60658 ms	100 ns	NULL					100 ns
362	-7.60656 ms	20 ns			800 ns	NULL		20 ns
363	-7.60654 ms	20 ns			800 ns	NULL		20 ns
364	-7.60644 ms	100 ns						100 ns
365	-7.60636 ms	80 ns				FCT		80 ns
366	-7.60622 ms	140 ns				FCT		140 ns
367	-7.6062 ms	20 ns				FCT		20 ns
368	-7.60612 ms	80 ns				FCT		80 ns
369	-7.60592 ms	200 ns	FCT			FCT		200 ns
370	-7.60584 ms	80 ns	NULL		800 ns			
371	-7.60582 ms	20 ns	NULL		800 ns			100 ns
372	-7.60576 ms	60 ns						60 ns
373	-7.6057 ms	60 ns						60 ns
374	-7.60564 ms	60 ns						60 ns

Figure 6.10: Packets transmitted during link initialization.

It is possible to see that this is fully compliant with the Auto-start mode. In fact End A, which is the GR-RASTA, starts the transmission by sending NULL characters after a *gotBit* event (the one in green) waiting for a response from the other part of the link. End B, which is the design implemented on the board, replies with a NULL too since being set in Auto-start mode. Subsequently the two actors continue with the exchange of a FCT character: at this point the link is established

and transmission of SpaceWire packets can take place. If nothing is transmitted both actors will continue with this handshake mechanism of NULL/FCT characters to ensure the link stays active and fully synchronized as can been seen in Fig. 6.10.

At this point the test program was compiled and downloaded inside Leon 3 memory. It is worth mentioning that the C-program employed the SpaceWire and RMAP predefined read and write functions which were available in the Gaisler proprietary drivers library, included inside the Eclipse project. Therefore it was sufficient to define correctly all the fields to execute the wanted operations. The program:

1. Configured the two SpaceWire ports employed (SpW ports 1 and 2) and assigned to RASTA the initiator logical address of *FA* (write operation in routing table memory location *03E0*);
2. Assigned to Port 6 of router (the one where RMAP IP core is attached) the logical address *F1* (write operation in routing table memory location *03C4*);
3. Performed the writing into Target memory of the two strings "*1234*" and "*ciao*" via RMAP, passing of course through the Router, at the two arbitrary locations *00F4* and *00F8*;
4. Performed the reading via RMAP in the same Target memory locations to check if correct values were written and then displayed them on the screen.

Figure 6.11 shows a Cygwin screenshot displaying test results. As visible, test was successful since all components of this thesis design have proven to correctly work on real hardware. In particular the screenshot proves that:

- The design implemented on hardware is capable of initializing the SpaceWire link with space standard equipment (Gaisler Research RASTA rack) otherwise no communication would be possible;
- Convert correctly data from SpW format to normal representation (8 bits) and viceversa since data values are the ones expected;
- Route correctly since first two RMAP writes enters Router Port 1 towards Port 0, second two enters Port 1 and are directed to Port 6 while in the last line, answers are in output of Port 2;
- Correctly perform writing and reading operation inside a target RAM memory through RMAP protocol since data passed via software have been found at the output of hardware.

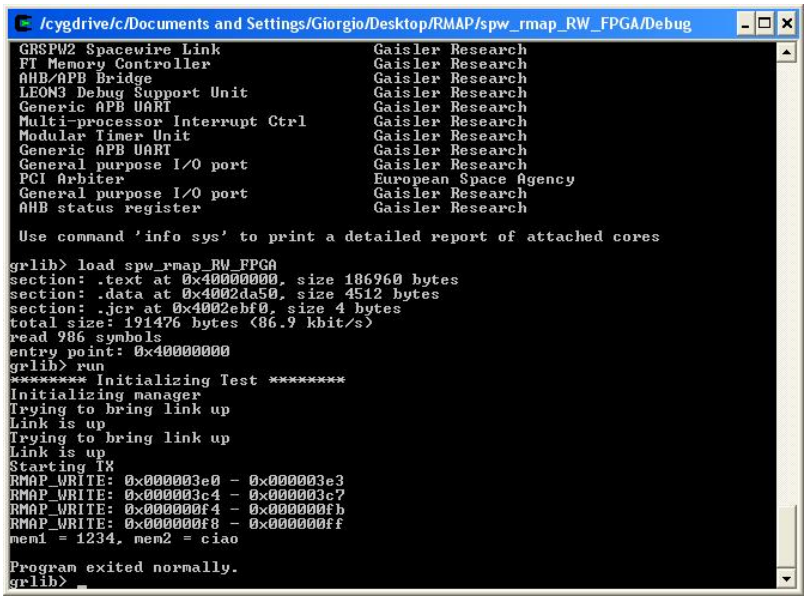


Figure 6.11: Screenshot of the test program final results.

6.3 Summary

A short summary of features of the implemented prototype and experimental results obtained is reported in Tab. 6.2.

		Obtained results
Logic resources	Occupied Slices	8403
	Device Utilization	10%
Performances	Max frequency (MHz)	87.3
Power	Dynamic (W)	0.6
	Leakage(W)	1.06
	Total (W)	1.66
Testing	SpW Interface Testing	Successful
	Network Functional Testing	Successful

Table 6.2: Summary of the obtained results.

Chapter 7

Conclusions and future work

In the present thesis, a data-link based on the SpaceWire protocol has been studied and implemented for its application on the context of vision-based navigation for Mars space exploration. All the different parts composing the communication network have been partially developed by a Japanes research team and left open-source online claiming compliance with ECSS standard. The main goal of this work was therefore their study, recostruction, simulation and organization into a single system that could be implemented in hardware. Finally, after the prototyping on a space qualified FPGA platform, the system has been tested using a Leon-3 processor to assure the correct functionalities working together with Avionic laboratory equipment.

After its study and simulations, the system proved to work as expected and functionally in compliance with the ECSS standards. Also the created board protype provided the correct functionalities even though some points of improvement have been identified: among this we can list trasmission rate speed. In fact though the design potentially could work at quite high frequencies (90 MHz) its test was carried out at low rates (3 MHz) due to the GR-RASTA's SpW drivers which resulted to be the limiting factor. Moreover, during the test phase, the SpW link analyzer proved to prevent the link initialization and slow down the transmission. Further tests should be done on this last points since without the analyzer its difficult to debug at the packet level in case of errors or the trasmission halts.

Anyway a consolidation of the implemented prototype is needed as next step before any implementation in a space project by means of a complete test verification activity that, for sure, cannot be considered in the scope of this thesis.

A consolidated product could become a valid alternative to the already existing

Intellectual Property (IP) cores providing more flexibility and budget savings especially for use within other research projects. For instance, the same SpaceWire sub-system inside the VisNav project could be included in the FEMIP project, an hardware accelerator for image processing developed by Politecnico di Torino in collaboration with Thales Alenia Space.

List of acronyms

CRC	Cyclic Redundancy Code
$ECSS$	European Cooperation for Space Standardization
ESA	European Space Agency
$FIFO$	First-In First-Out
$FPGA$	Field Programmable Gate Arrays
FSM	Finite State Machine
IP	Intellectual Property
$LVDS$	Low Voltage Differential Signalling
OBC	On-board Computer
$P\&R$	Place & Route
RAM	Random Access Memory
$RMAP$	Remote Memory Access Protocol
ROM	Read Only Memory
SpW	SpaceWire
TB	Test Bench
UCF	User Constraints File
UoD	University of Dundee
$VHDL$	VHSIC Hardware Description Language
$VHSIC$	Very High Speed Integrated Circuit

Bibliography

[1] URL: http://obsoletexilinx.com/devices/11669-XC4VLX160.

[2] URL: https://en.wikibooks.org/wiki/Programmable_Logic/FPGAs.

[3] URL: https://vjordan.info/log/fpga/trying-to-understand-the-internal-encoding-of-the-routing-switch-matrix.html.

[4] URL: https://www.embedded.com/print/4212259.

[5] URL: https://github.com/shimafujigit/SpaceWireCODECIP_100MHz.

[6] URL: https://github.com/shimafujigit/SpaceWireRouterIP_6PortVersion.

[7] URL: https://github.com/shimafujigit/SpaceWireRMAPTargetIP.

[8] URL: https://www.gaisler.com/index.php/products/boards/gr-cpci-xc4v.

[9] URL: http://spacewire.esa.int/content/Home/Purpose.php.

[10] URL: https://www.star-dundee.com/knowledge-base/packet-addressing.

[11] URL: https://www.gaisler.com/index.php/products/boards/mezzanine/gr-cpci-ser2-spw4?task=view&id=306.

[12] URL: https://www.star-dundee.com/knowledge-base/link-initialisation.

[13] European Space Agency. *ECSS-E-ST-50-12C. SpaceWire – Links, nodes, routers and networks*. 2008.

[14] European Space Agency. *ECSS-E-ST-50-52C. SpaceWire – Remote memory access protocol*. 2010.

[15] European Space Agency. *RMAP CRC implementation. Technical note*. 2006.

[16] Mentor Graphics Corp. *The Design Warrior's Guide to FPGAs. Devices, Tools, and Flows*. 2004.

[17] Pender Electronics Design. *GR-CPCI-SER2-SPW4 Board. User manual*. 2008.

[18] Pender Electronics Design. *GR-CPCI-XC4V Dev. Board. User manual*. 2013.

[19] University of Dundee. *FEIC DataSheet. VisNav-EM1*. 2015.

[20] Xilinx Inc. *Space-Grade Virtex-4QV Family Overview*. 2010.

[21] Xilinx Inc. *Space-Grade Virtex-5QV Family Overview*. 2012.

[22] Steve Parkes. *SpaceWire User's guide. STAR-Dundee*. 2012.

[23] *SpaceWire CODEC IP Core. User Manual - version 4*. 2014.

[24] *SpaceWire RMAP IP Core. User Manual - version 3*. 2014.

[25] *SpaceWire Router IP Core. User Manual - version 3*. 2013.

[26] Springer. *High-Performance Computing Using FPGAs*. 2013.

[27] *Xilinx Virtex-4 Family overview. Product specification.*