UNIVERSITÉ DE MONTRÉAL

DESIGN AND ARCHITECTURE OF A HARDWARE PLATFORM TO SUPPORT THE DEVELOPMENT OF AN AVIONIC NETWORK PROTOTYPE

DAVIDE TRENTIN
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
AVRIL 2012

© Davide Trentin, 2012.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

DESIGN AND ARCHITECTURE OF A HARDWARE PLATFORM TO SUPPORT THE DEVELOPMENT OF AN AVIONIC NETWORK PROTOTYPE

présenté par: TRENTIN, Davide
en vue de l'obtention du diplôme de: Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury d'examen constitué de:

M. DAVID, Jean-Pierre, Ph.D., président.
M. SAVARIA, Yvon, Ph.D., membre et directeur de recherche.
M. ZHU, Guchuan, Doct., membre et codirecteur de recherche.
M. LIU, Xue, Ph.D., membre.

À mes parents, Carmen et Claudio,
et à mon frère Stefano
qui m'ont toujours supporté
même si ça voulait dire me voir partir.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my research directors Prof. Yvon Savaria at École Polytechnique de Montréal, and Prof. Cristiana Bolchini at Politecnico di Milano, and to my research codirector Prof. Guchuan Zhu at École Polytechnique de Montréal for their suggestions, guidance, and encouragement, that helped me completing this thesis.
I also want to thank Bombardier Aerospace and Thales Canada Inc., as well as CRIAQ, NSERC/CRSNG, and MITACS, for funding the AVIO 402 Project, allowing me to explore the world of embedded avionic systems and networks, and to get in contact with professionals and experts of this domain.
Leaving Italy to study in Montreal has not been easy, but it was definitely an experience worth having, not only for what I learned in school, but also for what I learned about myself and for the great people I met. Thanks to my university colleagues José-Philippe, Talal, and Safwen, who shared with me problems and achievements during the entire project, and to Normand, who always had precious suggestions ready when I needed them. Thanks to the people who are living the same amazing international experience, Sergio, Mauro, Giuseppe, Francesca, Giovanni, Andrea, and the rest of the Italian gang for the nice moments spent together. Thanks to those who lived great adventures with me and to the greatest roommates I have ever had: Danai, Massimo, Ilaria, Sarah, and Jon.
Thank you Marie for always supporting me, for being at my side in the best and in the worst moments of the last year, and for being so special. And yes, thank you for revising my french! Most importantly, a great thank you to my beloved family, Stefano, Carmen, and Claudio, who always supported, loved, and encouraged me, even if this meant watching me go far away for two years and a half.

RÉSUMÉ EN FRANÇAIS

La récente évolution des architectures des systèmes avioniques a permis la création de réseaux avioniques modulaire embarqués (IMA) et l'augmentation du nombre de systèmes embarqués numériques dans chaque avion. Cette transition vers une nouvelle génération d'avions plus électriques permet une réduction du poids et de la consommation énergétique des aéronefs et aussi des couts de production et d'entretien. Pour atteindre une réduction du poids encore plus poussée et une amélioration de la bande passante des réseaux utilisés, des technologies innovatrices ont récemment été adoptées : ARINC 825 et AFDX qui permettent en fait une réduction du câblage nécessaire pour réaliser le réseau embarqué.

Dans le cadre du projet AVIO 402, qui inclut plusieurs sujets de recherche qui concernent aussi les capteurs et leur interface avec le système IMA, une nouvelle architecture a été proposée pour la réalisation du réseau utilisé pour le système de contrôle de vol. Cette architecture est basée sur des bus ARINC 825 locaux, connectés entre eux en utilisant un réseau AFDX qui offre une meilleure bande passante ; les ponts entre les deux protocoles et les modules qui connectent les nœuds au réseau ont une structure générique pour supporter des protocoles différents et aussi plusieurs types des capteurs et actionneurs. Pour une évaluation des performances et une analyse des défis de son implémentation, la réalisation d'un prototype du réseau proposé est requise par le projet.

Dans ce mémoire, le développement d'une plateforme matérielle pour soutenir la réalisation de ce prototype est traité et trois modules fondamentaux du prototype ont été conçus sous forme de "IP core" pour être subséquemment intégrés dans l'architecture du réseau qui sera implémenté en utilisant des FPGA. Les trois systèmes sont le contrôleur du bus CAN, utilisé comme base pour l'implémentation du protocole ARINC 825, le "End System" AFDX et le commutateur nécessaires pour la réalisation d'un réseau AFDX. Dans la première partie de ce mémoire, les objectifs visés sont présentés et une analyse des spécifications des protocoles considérés est fournie, cela permet d'identifier les fonctionnalités qui doivent être incluses dans chaque système et de déterminer si des solutions pour leur implémentation ont déjà été publiées et peuvent être réutilisées. Ensuite, le développement de chaque système est présenté et les choix de conception sont expliqués afin de montrer comment les fonctionnalités requises par les spécifications des deux protocoles peuvent être implémentées pour mieux répondre aux nécessités du projet AVIO 402.

Au début du projet AVIO 402, CAN, CANaerospace et ARINC 825 étaient considérés comme des solutions acceptables pour la réalisation des bus locaux, donc un contrôleur CAN facilement reconfigurable a été développé pour supporter ces trois technologies différentes, en

profitant des leurs grandes similitudes. Dans un deuxième temps, il a été préféré d'inclure le seul ARINC 825 dans le prototype et le système conçu a été adapté à cette solution. La plupart des architectures pour des contrôleurs CAN publiées dans les dernières années proposent une structure basée sur deux modules de traitement séparés pour les trames entrant et sortant du nœud. Pour rendre le IP plus facilement adaptable au bus ARINC 825, où les trames de surcharge ne sont pas permises, un gestionnaire central a été ajouté à ce type de système. Puisqu'il est responsable de toutes les fonctionnalités qui différencient un protocole de l'autre, ce gestionnaire est la seule partie du contrôleur qui nécessite une modification. Le système conçu garantit une occupation de seulement 3% des ressources du FPGA Spartan-6 utilisées pour la réalisation du prototype.

Pour le développement du "End System" (ES), une approche logiciel a été préférée à la réalisation d'un IP matériel à cause des interconnexions avec les applications du NCAP, le pont entre le bus local et le réseau principal, qui seraient exécutées par un processeur embarqué. Pour profiter des similitudes avec Ethernet et pour garantir une meilleure portabilité du code développé, le protocole AFDX est implémenté à partir de fonctionnalités de réseautage du noyau Linux exécuté par le processeur Microblaze. Cette solution permet de réutiliser l'interface API généralement utilisée pour Ethernet, basée sur les sockets, et les protocoles UDP et IP fournis par Linux, et aussi de rendre le "End System"indépendant du materiel. Pour émuler un environnement ARINC 653, requis par le projet AVIO 402, le système embarqué a été abandonné et un ordinateur a été utilisé pour continuer la modification des fonctions de réseautage du noyau Linux ; cette migration a permis d'apprécier la bonne portabilité du design conçu, puisque le code développé est indépendant du processeur utilisé.

Le dernier système réalisé est le commutateur du réseau AFDX. Pour minimiser la latence maximale des trames, ce module a été complètement implémenté en VHDL pour implémenter un traitement matériel parallèle des trames reçues. Une architecture à routage en parallèle, souvent utilisées dans des switch Ethernet, a été adoptée pour effectuer aussi le filtrage requis par la norme AFDX en parallèle. Un algorithme "token bucket" est exécuté par le module de gestion, qui détermine aussi la destination des paquets reçus, pour éliminer tous ceux qui ne respectent pas la bande passante allouée pour leur lien virtuel. L'ordonnancement des trames vers les destinations correspondantes tient compte des deux niveaux de priorité prévus par la norme. Un double tampon a été utilisé pour le stockage de trames en entrée pour séparer le traffic critique du traffic non critique, et conséquemment réduire encore plus la latence des premières ; la présence d'un tampon supplémentaire permet aussi de créer un système redondant où le deuxième tampon peut gérer les trames critiques quand l'autre est en panne. Le commutateur réalisé peut gérer un traffic à la vitesse maximale que le

réseau peut supporter (100 Mbit/s) sur chaque port et éviter l'accumulation des paquets dans les mémoires internes. Le module de routage est non bloquant, il peut donc transmettre plusieurs paquets simultanément quand ils n'ont pas la même destination et donc éviter toute congestion dans cette situation.

Le travail effectué dans ce mémoire sera utile non seulement pour la production du prototype en profitant des modules développés, mais aussi pour les leçons apprises et les solutions identifiées pour leur implémentation. L'architecture du commutateur AFDX en particulier est une contribution originale; en effet, la littérature concernée est très limitée et donc le système de filtrage des paquets et de ségrégation de trafics critiques et non critiques de ce type de système n'a jamais été étudié.

ABSTRACT

The objective of the present project is to design three modules for a hardware platform that will support the implementation of an avionic network prototype based on the FPGA technology. The considered network has been conceived to reduce cabling weight and to improve the available bandwidth, and it exploits the recently introduced ARINC 825 and AFDX protocols. In order to support the implementation of both these protocols, a CAN bus controller, an AFDX End System, and an AFDX Switch have been designed. After an extensive review of the existing literature about the two related avionic protocols, a study of the existing solutions for CAN and Ethernet protocols, on which they are based, has been done as well to identify what knowledge and technology could be reused.

Because they are very similar, a flexible CAN controller has been implemented in hardware instead of an ARINC 825 one in order to support both these technologies and in order to reduce the IP core size. A combined HW/SW approach has been preferred for the AFDX End System architecture to leverage an existing UDP/IP protocol stack and the Ethernet layer included in the Linux kernel has been modified to create a portable and configurable implementation of AFDX. Since various problems have been encountered to reproduce an ARINC 653 compliant environment on the embedded system, the suggested design has been ported in a PC. Finally, an original solution for the implementation of the AFDX switch fabric has been finally presented; a space-division switching architecture has been chosen and tailored to meet the AFDX specification. Hardware parallelism is exploited to reduce the latency introduced on each frame by filtering them concurrently. Input buffers have been duplicated to separate high from low priority traffics, further reducing latency of critical frames and creating a redundancy that reduce the possibility of packet loss. Packet scheduling and double queuing guarantee that all critical frames are forwarded before low priority ones.

Keywords: Avionic Full-Duplex Switched Ethernet, AFDX, ARINC 664, ARINC 825, CAN, Avionic Data Networks, Ethernet Switch, FPGA.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ EN FRANÇAIS v

ABSTRACT viii

TABLE OF CONTENTS ix

LIST OF TABLES xii

LIST OF FIGURES xiii

LISTE DES ANNEXES xv

LIST OF ABBREVIATIONS xvi

INTRODUCTION 1

CHAPITRE 1 CONTEXT AND OBJECTIVES 3
- 1.1 The AVIO 402 project 4
- 1.2 Project objectives 8
- 1.3 Hardware platform 9
  - 1.3.1 Development environment 9
  - 1.3.2 IP modules 10

CHAPITRE 2 AVIONIC DATA COMMUNICATION NETWORKS 12
- 2.1 Overview of avionic protocols 12
  - 2.1.1 The CAN protocol 13
  - 2.1.2 The AFDX protocol 17
- 2.2 Literature review 20
  - 2.2.1 CAN bus controller 21
  - 2.2.2 AFDX End System 22
  - 2.2.3 AFDX Switch 25

CHAPITRE 3 CAN BUS CONTROLLER 29
3.1 Specification and requirements 29
3.2 Design 30
3.2.1 Hardware vs. Software considerations 30
3.2.2 Architecture 31
3.3 Implementation 33
3.3.1 Functional verification 37
3.3.2 Migration towards ARINC 825 40

CHAPITRE 4 AFDX END SYSTEM 42
4.1 Specifications 42
4.2 Proposed solution 43
4.2.1 The Linux Ethernet protocol stack 44
4.2.2 End System design 45
4.3 End System development 47
4.3.1 Hardware embedded system 48
4.3.2 Software implementation 49
4.4 Practical Problems and Lesson Learned 52

CHAPITRE 5 AFDX SWITCH 55
5.1 Specification and requirements 55
5.1.1 ADFX switch specification 55
5.1.2 Switch Fabric 56
5.1.3 AVIO 402 requirements 59
5.2 Core Design 59
5.2.1 Hardware advantages 60
5.2.2 Switch Architecture 60
5.3 Synthesis results 71
5.3.1 System size 72
5.3.2 Timing 73
5.3.3 Considerations on the implementation 75
5.4 Test and validation 75
5.4.1 Testbenches 76
5.4.2 System behaviour 78
5.4.3 Performance measurement 79

CHAPITRE 6 CONCLUSION AND FUTURE WORK 84

BIBLIOGRAPHY 87

APPENDICES 90

FSM implementation 90

B.1 Queue implementation 98

B.2 Manager implementation 107

B.3 Scheduler implementation 112

B.4 Filter implementation 115

C.1 Switch Fabric test cases 121

C.1.1 Test test case 1 121

C.1.2 Test test case 2 122

C.1.3 Test test case 3 123

C.1.4 Test test case 4 123

C.1.5 Test test case 5 124

C.1.6 Test test case 6 125

C.1.7 Test test case 7 126

C.1.8 Test test case 8 128

C.1.9 Test case 9 128

C.1.10 Test case 10 129

C.1.11 Test case 11 129

LIST OF TABLES

Table 3.1	Size of the CAN controller modules	36
Table 3.2	Hexadecimal value of each state of the reception branch of the FSM	38
Table 3.3	List of the scenarios used to validate the *Manager* behaviour	39
Table 3.4	Size of the ARINC 825 controller	41
Table 5.1	System size for 10 and 20 ports	72
Table 5.2	Size of single modules	72
Table 5.3	After synthesis operating frequence	74
Table 5.4	List of the most significant tests	77
Table 5.5	Technological latency for frames of minimum and maximum length	79
Table 5.6	Output FIFO overflow	81
Table C.1	Test 3: VLs and their destinations	124
Table C.2	Test 4: VLs and their destinations	125
Table C.3	Test 7: Output FIFO overflow	127
Table C.4	Test 10: Frame sizes	130

LIST OF FIGURES

Figure 1.1 Avio 402 network overview 5
Figure 1.2 Network architecture 6
Figure 1.3 NCAP structure 7
Figure 1.4 Prototype overview 8
Figure 2.1 CAN extended data frame structure 14
Figure 2.2 AFDX network example 18
Figure 2.3 BAG and Jitter in ES transmission 19
Figure 2.4 AFDX frame structure 20
Figure 2.5 HW implementation of the ES 24
Figure 2.6 Combined input/output buffering structure 27
Figure 3.1 CAN controller internal architecture 31
Figure 3.2 FSM execution flow 34
Figure 3.3 Simulation example: FSM reception reception flow 37
Figure 4.1 SW architecture overview 44
Figure 4.2 ES hardware architecture 48
Figure 4.3 SW architecture overview 49
Figure 4.4 Modified prototype structure 53
Figure 5.1 Representation of the modules of the Switch taken from the specification 57
Figure 5.2 Example of frame-based leaky bucket algorithm application 59
Figure 5.3 Architecture of the Switch 61
Figure 5.4 Reception modules 63
Figure 5.5 Transmission modules 64
Figure 5.6 Head of Line Blocking 67
Figure 5.7 Frame treatment by the core functional modules 74
Figure 5.8 Basic routing functionalities 77
Figure 5.9 Minimal technological latency measure 79
Figure 5.10 Latency of the last byte for a 1ms single-port burst 80
Figure 5.11 Routing of concurrently received frames and priority management . . 82
Figure 5.12 Example of error detection: bad CRC 83
Figure C.1 Test Case 1 131
Figure C.2 Test Case 2 132
Figure C.3 Test Case 3 133
Figure C.4 Test Case 4 134

Figure C.5 Test Case 5 - first example 135
Figure C.6 Test Case 5 - second example 136
Figure C.7 Test Case 6 137
Figure C.8 Test Case 7 138
Figure C.9 Test Case 8 139
Figure C.10 Test Case 9 - overview 140
Figure C.11 Test Case 9 - CRC control 141
Figure C.12 Test Case 10 142
Figure C.13 Test Case 11 143

LISTE DES ANNEXES

Annexe A IMPLEMENTATION OF THE MANAGER OF THE CAN CORE . 90
Annexe B IMPLEMENTATION OF THE AFDX SWITCH 98
Annexe C SWITCH FABRIC TEST VERIFICATION 121

LIST OF ABBREVIATIONS

ADN	Aircraft Data Networks
AFDX	Avionic Full-Duplex Switched Ethernet
AMP	Arbitration based on Message Priority
BAG	Bandwidth Allocation Gap
COTS	Component Off The Shelf
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
ES	End system
FCC	Flight Control Computer
FCS	Frame Control Sequence
IMA	Integrated Modular Avionics
LRM	Line Replaceable Module
LRU	Line-replaceable Unit
MAC	Media Access Controller
MEA	More Electric Aircaft
MTU	Maximum Transmission Unit
NCAP	Network Capable Application Processor
NIC	Network Interface Controller
QoS	Quality of Service
SOF	Start of Frame
TIM	Transducer Interface Module
VL	Virtual Link

INTRODUCTION

The aerospace industry has historically been reluctant to introduce significant innovations to replace old, well known, reliable systems. In recent years though, the market has been pushing toward the realization of more efficient and easily maintainable aircrafts to reduce production and maintenance costs. These needs allowed the introduction not only of new electronic technologies in the avionic environment, but also a general reorganization of the aircraft support structure. Integrated Modular Avionics (IMA) is slowly replacing the traditional federated architecture and the fly-by-wire philosophy is being applied more and more instead of the old mechanical and hydraulic control used in airborne systems, resulting in a general trend towards a More Electrical generation of Aircrafts (MEA).

All these innovations involve a significant increase in the complexity of electronic controls, and number of actuators, and sensors; therefore, the volume of digital data exchanged between avionic systems is growing and becoming harder to handle. Because of this transitions old and mature technologies exploited so far for the communication of electronic data, such as ARINC 429, are now showing their limits and problems, especially for what concerns their low bandwidth and the extremely large bundle of wiring they require. This is why new interesting solutions are finally finding place in avionic networks. For instance, AFDX and ARINC 825 are two of the most promising technologies available for the aerospace industry to solve the aforementioned problems.

The work presented in this thesis concerns the development of three IP cores to realize a hardware platform that will support the implementation of an avionic network prototype. Such a prototype will support verification as well as testing of reliability and performance of a new network architecture proposed as a part of the AVIO 402 CRIAQ project, which aims at helping the industrial partners migrate towards a greener, less costly, and more energy-efficient generation of aircrafts. ARINC 825, one of the considered protocols, is used for the communication between sensors and actuators grouped in local clusters. By contrast, AFDX provides a connection between these local networks and the central Flight Control Computers (FCCs) and other IMA systems. This architecture offers not only high bandwidth and reliability, but also a significant reduction of the required cabling, and consequently of the overall weight of the system.

With this thesis work, a development environment has been set for the prototype implementation and three fundamental modules of the network have been studied and conceived: a CAN bus controller to support ARINC 825 connectivity, an AFDX End System, and an AFDX switch fabric. Each module has been analysed to determine the best approach for its

implementation and to propose a design for its architecture considering the needs concerning performance and system size required by the protocol specification and by the AVIO 402 project. An implementation of their fundamental functionalities has been done as well to determine if the selected solutions can provide the necessary performance and to provide the starting point for the realization of the final prototype. The contribution of this work is not limited to the development of this hardware platform, but also to the identification of critical aspects of the implementation of these recently introduced technologies to propose possible design solutions that would help improving their Technology Readiness Level.

Outline of the Thesis

More precise information about the objectives of this work can be found in chapter 1, together with a description of its context and of the AVIO 402 project. In chapter 2, the ARINC 825 and AFDX protocols are introduced to highlight their peculiarities and their innovative features, and the state of the art concerning their implementation is presented together with a review of the existing literature. The following three chapters focus on the three developed systems, presenting the conceived design, the implemented features, and some obtained results. In chapter 3, the design of the CAN bus controller is described and the synthesis results are provided as well. The AFDX End System is treated in chapter 4, where the proposed solution is explained and its advantages and problems are discussed. Chapter 5 presents the design and implementation of the AFDX switch, describing how the conceived design can reduce frame latency and reduce the impact of the head-of-line blocking. Final considerations on the obtained results, on the lesson learned, and on the perspective opened by this work are included in chapter 6, where future work is discussed as well.

CHAPITRE 1

CONTEXT AND OBJECTIVES

The main trend of the aerospace industry in recent years has been directed towards the development of greener, less costly, and easily maintainable aircrafts. These goals are being achieved by making airplanes "more electrical", replacing mechanical controls with electrical fly-by-wire controls, and adopting Integrated Modular Avionic (IMA) architectures. All these aspects are closely interconnected with each other.

The idea of More Electrical Aircraft (MEA) became interesting thanks to advances in solid-state power-related electronics, that can now provide the necessary electrical power to replace heavy an expensive hydraulic parts, as explained by Rosero *et al.* [1]. More and more mechanical and hydraulic systems are now being eliminated from airplanes and electrical motors and actuators are taking their place. These are usually lighter and more energy-efficient than their predecessors, and they can also be easily digitally controlled. This last aspect made also possible the transition from mechanical controls to electrical ones, that brought to the development of fly-by-wire systems. MEA and fly-by-wire largely contributed to the reduction of aircraft weight and consequently of the overall fuel consumption, also reducing components wear out and maintenance costs of the machine.

A classic aircraft is based on a federated architecture, where each of its systems is developed as a Line Replaceable Unit (LRU) that can be easily changed or replaced, and that provides a single peculiar functionality in order to guarantee segregation of faults occurring in one of them and to avoid their propagation. Digital control of engines and actuators made possible the development of IMA architectures, where embedded systems are exploited to handle more than one task, thus reducing the number of sub-systems on board, and consequently of the overall weight and power consumption. This concept has been studied in the 90s and it has been finally standardized at the end of the 2000s, with the production of the Airbus A380 and the Boeing 787. The new sub-systems of the network are often called Line Replaceable Modules (LRMs) and they must ensure logical and physical segregation of the various tasks in order to keep each one of them independent from anything else and to prevent fault propagation. LRMs often exploit commercial off-the-shelf (COTS) components in order to reduce production costs and, when they are processor based, they usually rely on the ARINC 653 compliant operating systems to ensure time and spatial independence of the various applications running on it. The IMA architecture offers other important advantages such as reduction of inventories in Airline Maintenance Centres and reduction of cost for

version upgrades and functional enhancements.

All these became popular in the last decade and they brought a significant increase in the volume of digital data that needs to be exchanged on board, exposing limitations and disadvantages of old technologies usually adopted for data communications: traditional avionic technologies offer limited bandwidth, and they also involve bulky wiring bundles when the number of connected systems increases, leading to serious problems related to wiring weight. The need for a more efficient, highly-reliable network to transmit all this digital data across the aircraft is consequently becoming a priority. Recently developed protocols such as ARINC 825 or Avionic Full-Duplex Switched Ethernet (AFDX) aim at solving these problems providing large bandwidth and a network structure that allows wiring reduction while guaranteeing high reliability.

1.1 The AVIO 402 project

This project has been proposed and is supervised by two industrial partners, Bombardier Aerospace Inc. and Thales Group, and it is supported by CRIAQ, NSERC-CRSNG, and MITACS as well. Not only École Polytechnique de Montreal, but also ETS and McGill University are involved in various aspects of the project. The project has multiple complementary goals, all intended to push the avionic industry towards a greener generation of aircrafts: the development of new MEMS, optical sensors, and of a universal smart sensors interface, and the frequency selection for on board wireless communications, are some of the other topics covered in this project along with the design of a reliable communication network.

In parallel with the studies on AFDX performance, end-to-end latency, and optimization and of the reliability of the proposed network architecture, the development of a prototype of this network is required as well: the objective is to explore practical implementation issues, and to provide a platform, or test-bed, for practical testing and verification of the achievable performance and potential limitations. To understand the test-bed structure, what needs to be included in it, and which modules will be examined in this thesis, a description of the proposed network architecture is given in the following section. A high-level overview of the general architecture of the network that is being developed is shown in Figure 1.1: the system includes some local clusters relying on a local field bus, generic IMA modules, a main backbone AFDX network, and redundant Flight Control Computers (FCC). Each remote cluster can rely on a variety of possible technologies, such as CAN, ARINC 825, but also legacy systems that use traditional protocols such as ARINC 429, and the ES (End Systems) that connect these clusters with the main network must provide a gateway between these protocols and AFDX. In particular, ARINC 825 has been chosen for the prototype realization

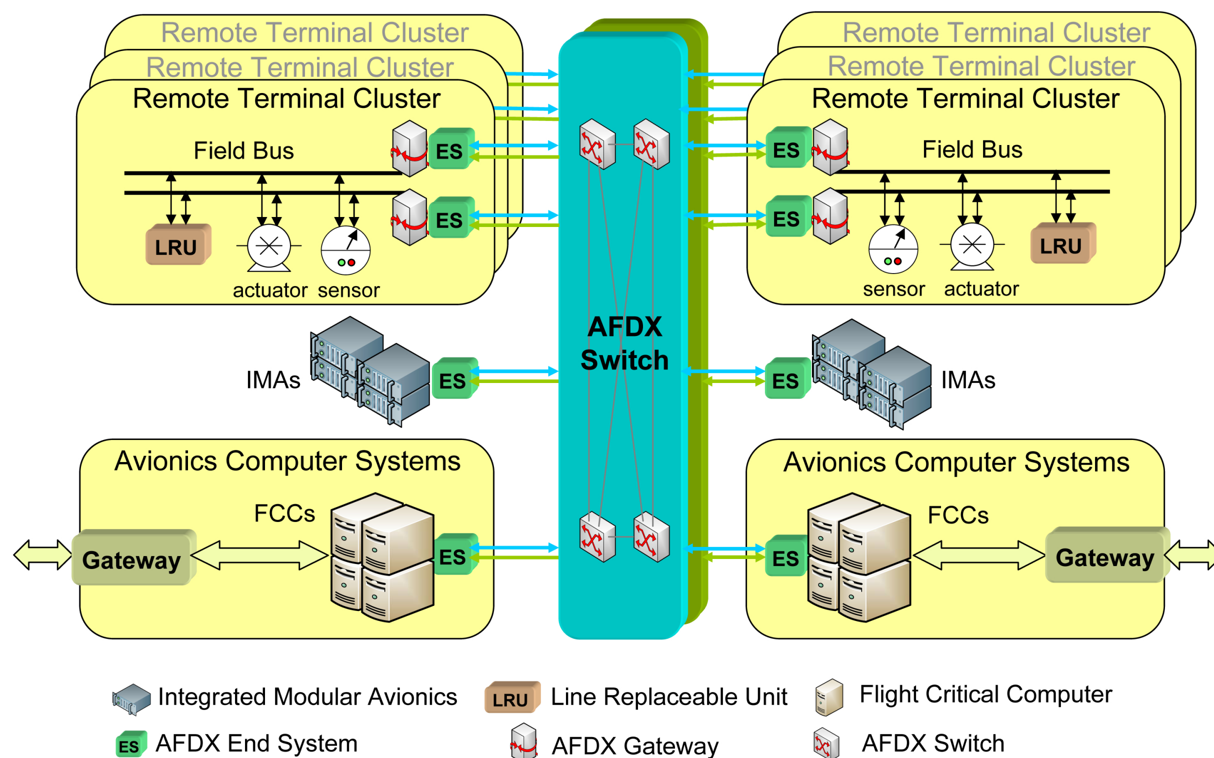

Figure 1.1 Network overview directly taken from the internal documentation of the Avio 402 project

in order to explore this fairly recent technology.

System overview

A refinement of the network architecture has been proposed by J.-P. Tremblay and it has been conceived to provide high bandwidth and wiring reduction while maintaining a high overall reliability. As shown in Figure 1.2, sensors and actuators present across the aircraft are grouped into local clusters, as specified by the AVIO 402 project, which rely on the ARINC 825 protocol. This CAN based protocol is an enhanced version of CANAerospace, which has known a wide diffusion in recent years, and it is a promising solution for small avionic networks based on the LRU concept, thanks to its high reliability and its efficient networking capability. This bus links all the sensors and actuators of the local group together and with multiple Network Capable Application Processors (NCAPs) as well. The bus is not directly connected to the peripherals, in fact Transducer Interface Modules (TIMs) are exploited to provide actuator control, or sensor management, while dealing with commands coming from the central control unit and with communications with other peripherals. On the other side,

the NCAP module realizes the bridge between the local and the main networks, providing a gateway between the two different adopted protocols and some additional services to handle data flow, to wrap packets together, and to determine data destination. The generic structure of this network, as well as the terminology and the acronyms that have been adopted are taken from the IEEE 1451 standard [2] for a smart transducer interface for sensors and actuators, that has been used as guideline but that had to be adapted to the avionic environment since it was not originally intended for it. The main network is based on the AFDX technology because it must provide a higher bandwidth than the local networks since multiple clusters will exploit it simultaneously. AFDX allows a reduction of the cabling thanks to the concept of virtual links and it is one of the most promising technologies for next generations of aircrafts.

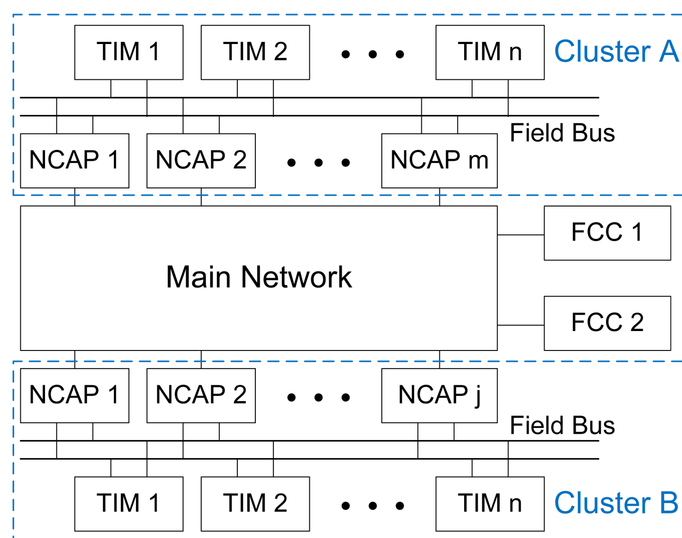

Figure 1.2 Network architecture as specified by J.-P. Tremblay in the internal documentation of the Avio 402 project

Figure 1.2 shows two clusters, one on the top and one on the bottom, connected to the main network through multiple NCAPs, their number can change from one cluster to the other. Each NCAP is connected to the field bus that put it in communication with all the TIMs that are connected to the sensors and actuators of the local cluster. Redundant paths are used inside each TIM and NCAP module as well to provide even more possible routes for transmitted data: when one of the modules stop working correctly, an alternate path can be used to guarantee data delivery, but the price to pay is that more resources are required.

The NCAP constitutes a sort of bridge between the local network and the main one, realising a gateway that must be able to communicate with the two protocols it is interfaced with, but also to provide additional services for the control of the peripherals, for traffic flow

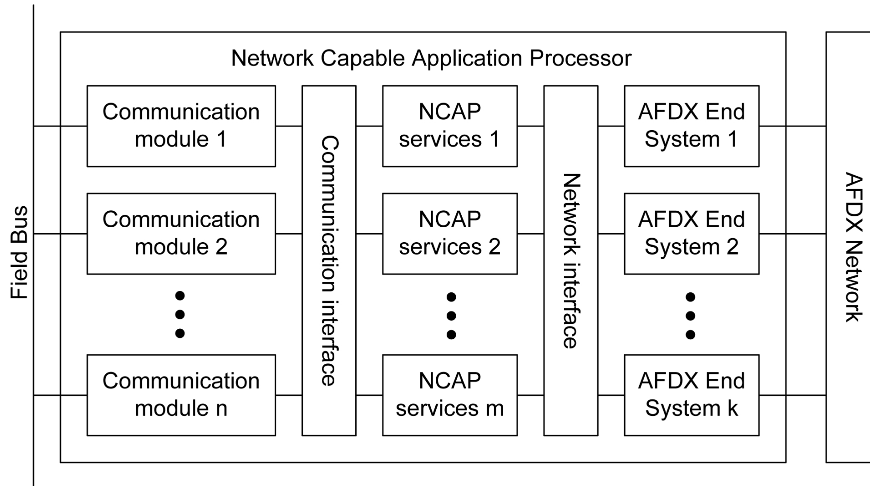

Figure 1.3 NCAP structure as designed by J.-P. Tremblay for the Avio 402 project

control, and for the identification of the nature of the information that is passing through it. As shown in Figure 1.3, the interface with each network is realized via a dedicated controller that must handle the corresponding protocol. In both directions, crossbars can redirect received packets towards any of the available service modules, and to any of the output interfaces, thereby reducing the introduced latency and avoiding the loss of frames in case of failure of one of the modules of the system. NCAP services are still not precisely determined, thus flexible interfaces must be considered to communicate with them. The TIM internal structure is not described here since it is not strictly related to this thesis, but it is important to know that a communication module identical to the one instantiated in the NCAP as interface with the field bus is present in the TIM as well to reproduce the same function. The AFDX network is based on the star topology typical of switched Ethernet, therefore it requires switches to handle frame routing and traffic control. These routing modules are mostly similar to a standard Ethernet switch but they must also provide some additional features required by the AFDX protocol.

With the development of the prototype, it will be possible to test the overall functionality of this proposed network architecture and its performance. The experimental results will also be useful to validate the models developed to evaluate end-to-end delay, AFDX jitter, and system reliability. A first implementation of the prototype will not include the redundant paths included in the general structure previously presented; the overall structure of the system that should be included in the prototype is shown in Figure 1.4 where the modules conceived in this work are highlighted.

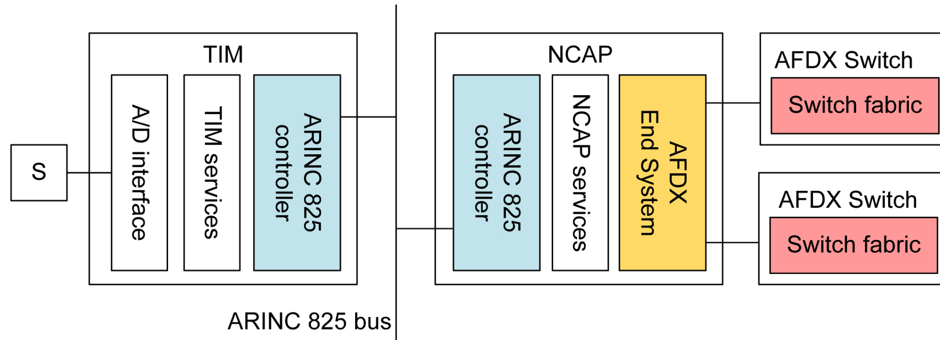

Figure 1.4 Prototype overview

1.2 Project objectives

The goal of this thesis is to contribute to the development of a hardware platform by developing three Intellectual Properties (IPs) modules to support the realization on FPGA of the network prototype required by the AVIO 402 project. These modules will constitute the starting point of the implementation of an ARINC 825 controller, an AFDX End System (ES), and an AFDX Switch. It can be observed in Figure 1.4 that these subsystems constitute the fundamental blocks of the basic implementation of the network included in the prototype. An analysis of the considered modules is required to identify which approach is the most suitable in each case to reach the best results considering the context this modules need to be used in; a fully hardware implementation in VHDL, that could be optimal for certain systems, may not be advantageous when developing other parts of the network. The parameters used to select the preferable solution (between all those that can meet the performance required by the corresponding specification) are different for the three cases: while the size is critical for the ARINC 825, and reconfiguration capabilities are important for the AFDX End System, processing time and frame latency must be minimized in the AFDX switch fabric.

The first objective is to realize the three subsystems that meet the functionalities and performance required by the corresponding specification, but this must be done considering their future integration in the prototype and the tasks included in the AVIO 402 project. Interfaces need to be compatible with those defined for adjacent modules of the prototype, and the internal structure of each IP must support potential future modifications that could become necessary to integrate solutions proposed by other colleagues to improve the network performance. In order to develop a solution that can be exploited also in a commercial product (i.e. not only a test-bed for network analysis, design reuse must be addressed as far as possible; this will also help increase the technology readiness level of the exploited

protocols.

Although ARINC 825 has been adopted in the final network architecture, this solution was not initially confirmed, and CANaerospace and CAN were considered viable choices as well. Thanks to the very close similarity between these technologies, it has been decided to initially develop a standard CAN controller, that would be easily configurable to support the two avionic protocols if necessary. The other two systems needed for the prototype, the ES and the Switch, are really different from each other even if they realize the same protocol. The ES requires the development of the protocol on multiple layers, it must in fact communicate with the physical layer, encapsulate data following the ARINC 664 specification, meet the traffic control requirements described in the same documentation, and it also has to work in close relation with the NCAP services and Gateway functionalities integrated in the NCAP. The AFDX switch fabric has a similar role in the network to the one used in Ethernet switches, it must in fact route frames towards their destination but also filter them to discard the ones that do not respect the AFDX requirements.

1.3 Hardware platform

Because of the flexibility required in a prototyping stage, the whole system will be implemented on FPGA. This solution allows easy modifications and testing even after the network development is completed, and it also provides the possibility of exploiting a mixed HW/SW approach for the realization of each subsystem. A description of the chosen development environment is given here before passing to a presentation of the three modules that will be discussed in this thesis.

1.3.1 Development environment

The FPGA development board has been chosen in function of the provided peripherals, of the FPGA size, and of its cost. Because of costs and development time, the same board should host all the various systems composing the network; the chosen device must consequently satisfy the needs of both the TIM and the NCAP modules, as well as of the AFDX switches. This solution facilitates portability of the developed modules from one system to another, for example, the CAN controller could be used on both the TIM and the NCAP with no modification at all.

The chosen board is the SP605 by Xilinx, expanded with the ISM networking FMC module by Avnet: this entry level board is an inexpensive solution that can host the whole NCAP, while the expansion FMC card provides additional communication ports. The SP605 board [3] is based on a Spartan-6 XC6SLX45T FPGA and it provides, among other features,

general I/O and UART connectors, one tri-mode Ethernet PHY, a DDR3 memory, a 1-lane PCI Express connector, and an FMC LPC (Low Pin Count) connector. This last connector is used to plug the ISMNET LPC extension card [4] that provides two additional Ethernet PHY as well as two CAN bus connectors required for the ARINC 825 communication into the board.

This generic development board provides an inexpensive and easy solution for the realization of the whole prototype. The two CAN bus ports provided by the FMC card are required for the field bus implementation, and the two Ethernet ports for the AFDX End system connection on the main network. The presence of three Ethernet PHYs overall allows the realization of a minimal switch on the same board as well.

1.3.2 IP modules

The design and development of three fundamental Intellectual Properties is discussed in this thesis, these modules are conceived according to the selected FPGA development environment and to the objectives of the AVIO 402 project.

CAN bus controller

In the description of the network proposed for the AVIO 402 project, the CAN bus was not mentioned, but, as explained above, at the time this project started, the adoption of the ARINC 825 protocol was not confirmed yet. Because of the strong analogies between CAN, CANaerospace, and ARINC 825 protocols, it was planned to initially develop a standard CAN bus controller that would support potential future modifications if necessary; the smaller the required changes for this transition the better. A detailed analysis of the two avionic standards is required to identify their differences from CAN, and to determine how they could affect the implementation of its controller. Even if inspiration can be taken from already existing designs and solutions for a CAN controller's architecture, attention must be paid to ensure a complete compliance with these avionic technologies. Unfortunately, while an extensive literature is available for the implementation of a CAN core, the same cannot be said for ARINC 825 and CANaerospace bus controllers, and even if commercial products are available, no internal description of these systems has been found in the literature. This controller constitutes the communication module included in both the TIM and in the NCAP systems for the communication with the field bus, and, as depicted in Figure 1.3, it needs to be instantiated multiple times to guarantee better reliability; as a consequence the IP size can become a critical parameter depending on the number of instantiations required, and it must be considered as a key feature when designing this system.

AFDX End System

The second module that must be designed is the front end of the NCAP module towards the AFDX network, which is included in the switches as well, and it is called an End System. The role of this system is to provide data encapsulation/decapsulation, traffic flow control, and transmission/reception of frames over the network. Data that is managed by higher level applications (that can implement NCAP services, or the Gateway towards the ARINC 825 field bus, or again the switch's health manager) is passed to the ES for transmission over the network. The strict relationship of the ES with these other required functionalities makes it impossible to develop them completely independently, therefore the design of this system must take the others into consideration and be as flexible as possible because of the provisional nature of those in this prototyping stage. Active research is in progress about optimal scheduling and other techniques aimed at minimizing frame latency and jitter in AFDX networks, and the final prototype should support and facilitate their potential implementation for testing. Key features of this IP must consequently be ease of integration and interaction with the aforementioned functionalities, and reconfigurability, to test this system with different scheduling algorithms and with different configurations of the supported Virtual Links.

AFDX Switch

The last system considered in this thesis is the routing module, that represents the heart of the AFDX switch. The AFDX network is based on the same switched topology as a standard switched Ethernet network, but the nodes that are responsible for the routing of the frames, although they share multiple identical features with their IEEE 802.3 counterpart, are different under various aspects such as frame filtering, traffic policing, latency and priority management. There are no publications in the current literature concerning the implementation of this system, thus a theoretical study of how techniques and architectures developed for ATM Ethernet switches can be applied in an AFDX environment becomes necessary to determine the most suitable approach for its development. The goal is to identify advantages and problems of existing solutions when applied to airborne networks, to successively identify the most suitable for the current application and implement it as an IP core that will be instantiated in the switches of the prototype. While many ATM switches are designed to optimize the average Quality of Service (QoS) and the average delay introduced, in AFDX switches, it is the maximum latency that must be limited, and a mandatory differentiated services (DiffServ) processing is required to handle separately high and low priority traffics.

CHAPITRE 2

AVIONIC DATA COMMUNICATION NETWORKS

In this chapter, essential information about the considered protocols is given, as well as an overview of the existing literature about the current state of the art concerning the architectures and implementations of the three systems considered in this thesis. Since a basic knowledge of the most important aspects of the studied technologies is fundamental to fully understand the architectural choices made throughout this work, a general overview of these features is depicted in Sections 2.1.1 and 2.1.2, while their detailed description can be found in the provided references. An exploration of the existing solutions for the implementation of these three systems is done afterwards in Section 2.2, together with a review of the publications concerning the considered technologies.

2.1 Overview of avionic protocols

Classical Aircraft Data Networks (ADN) primarily utilize the ARINC 429 standard. This standard, developed over thirty years ago and still widely used today on a variety of aircrafts from different companies, has proven to be highly reliable in safety critical applications and to provide the necessary performance for avionic applications. ARINC 429 networks rely on a unidirectional bus with a single transmitter and up to twenty receivers. A data word consists of 32 bits communicated over a twisted-pair cable. There are two speeds of transmission: high speed operates at 100 Kbit/s and low speed operates at 12.5 Kbit/s. ARINC 429 operates in such a way that its single transmitter communicates in a point-to-point connection with its receivers, thus requiring a significant amount of wiring overhead since every new connection requires additional cables, significantly increasing overall aircraft weight. Although it is nowadays a popular standard in civil aerospace applications, ARINC 429 requires custom hardware, which can increase aircraft cost and development time.

Apart from ARINC 429, which remains the most popular and most widely used technology for digital on board communications, other technologies can be found on most airplanes. ARINC 629 for example, firstly introduced by Boeing for its 777 aircraft, provides a higher data speed, up to 2Mbit/s, and does not require the presence of a bus master, thereby increasing reliability of the network architecture. Its major drawback though is the need for custom hardware that can make its development time and costs excessive. Another common protocol is the MIL-STD-1553, mostly used in military avionics and spacecraft for on board

data handling. It provides very high reliability but it is expensive and its bandwidth is limited.

ARINC 825 and AFDX have been conceived to solve the bandwidth, wiring weight, and costs problems of the above mentioned protocols: the important cabling reduction, together with the good reliability they guarantee, makes them two of the most promising technologies for the development of lighter, greener, and less costly aircrafts. The interest in these technologies is attested by the rich literature that is being produced, especially on AFDX: various researches are being done on multiple aspects, such as worst-case end-to-end delay analysis, ES and Switch algorithms, or guaranteed reliability [5, 6, 7]. More details on the literature available on AFDX and CAN-based technologies are given in Section 2.2.

2.1.1 The CAN protocol

The Controller Area Network (CAN) data bus is a serial communication protocol that supports distributed real-time control with a high level of security. It is designed to allow microcontrollers and devices to communicate with each other without the need for a master host computer. It was designed in the 80s by Robert Bosh GmbH [8] for automotive applications, but its success, reliability, and versatility attracted the attention of manufacturers in other industries, including process control, medical equipment, and recently Avionics. It was Airbus that first introduced it in this field during the development of the A380.

All the devices of a CAN network are connected to a single twisted-pair of wires that they must share as communication medium, but optical fibre can be used as well. The CAN bus operates at data rates up to 1Mbit/s for cable lengths less than 40m, but it is necessary to reduce the data rate for longer cables; it usually falls to 125Kb/s when the length is around 500m. Maximum speed must be decreased also when the number of LRUs connected to the bus increases. The International Standards Organisation (ISO) has formalized this protocol in the ISO 11898 (High-Speed CAN bus, up to 1Mbit/s) and ISO 11519 (Low-Speed CAN bus, up to 125Kb/s) specifications. Two version of the CAN protocol are specified: CAN 2.0 A and CAN 2.0 B, the first uses the standard or base frame format, that supports an 11-bit identifier, while the second uses an extended frame format in which the identifier is composed by 18 additional bits (for a total of 29 bits).

Controllers connected to the CAN bus must transmit and receive data while avoiding collision using the CSMA/AMP technique. With Carrier Sense Multiple Access (CSMA), a bus controller can start a new transmission only when the bus is idle, and if two nodes try to transmit at the same time an arbitration logic allows the transmission only of the message with the highest priority (Arbitration based on Message Priority or AMP).

CAN is a broadcast-type bus since each transmission is received by all the terminals

connected to it; it is each node's responsibility to determine if the received frame is relevant to that particular system or not, and to drop packets that were not addressed to it. Frames do not include source nor destination addresses, and their header is the only information the receiving terminal can use to identify relevant frames. This structure makes this bus really effective and versatile when working with LRUs.

Message arbitration

The bus can have two logic values, dominant and recessive, and whenever two terminals attempt a simultaneous transmission of a dominant bit and a recessive bit, a dominant logic value will result on the bus. In a typical implementation of a wired connection 0 is the dominant value, and consequently this is often called an "AND" implementation. The first controller that lose a contention, sending a recessive bit and reading a dominant value resulting on the bus, must immediately stop its transmission. The result is an arbitration technique based on the message header, which determines the communication priority.

Frame structure

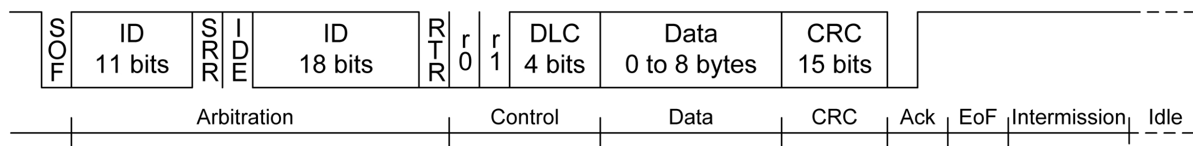

Figure 2.1 CAN extended data frame structure

CAN data frames consists of seven bit fields: *Start Of Frame (SOF)*, *Arbitration*, *Control*, *Data*, *CRC*, *ACK*, and *End Of Frame (EOF)*. At the end of a transmission there is an *Intermission* period where no other communication can start. *SoF* is always a dominant bit marking the beginning of a transmission, and it is followed by the *Arbitration* field, composed by the Identifier, the Remote Transmission Request (RTR) for a standard frame, or the Substitute Remote Request (SRR) for an extended frame, and finally the Extension bit IDE, that determines if the frame is standard or extended. r0 an r1 are reserved bits that are always dominant, and compose the *Control* field together with the Data Length Code of 4 bits, which specifies how many bytes are present in the Data field. The *Data* field contains the actual information, while the *CRC* is used to guarantee data integrity. During the recessive bit transmitted in the *Acknowledgement* field a dominant value must be received on the bus to ensure that the frame has been correctly received. In Figure 2.1, a data frame with extended identifier is shown, since it is the only one used by CANaerospace and ARINC

825: SRR and IDE equal to ‘1’ signal the presence of the additional 18 bits Identifier, and RTR is ‘0’ because this is a data frame. To ensure a strong synchronisation, the protocol avoids the presence of more than 5 consecutive bits of the same value in the transmitted frame by adding a stuffing bit after them in order to have a transition of the bus value that will let all the bus nodes readjust their sampling time.

Error detection

CAN bus is a technology that guarantees high reliability. Each terminal is in fact responsible for auto-supervision and faulty controllers are automatically disconnected from the bus, without the need for a global supervisor. Whenever an error occurs, the controller that identified the problem must generate an error frame to make all the other nodes connected to the bus aware of that. There are five different error types:

- *Bit Error*: the bus must be monitored during transmission to verify that, once the arbitration is won, the monitored bit value corresponds to the bit value that is sent;
- *Stuff Error*: it occurs whenever six consecutive equal bit values are detected in the received frame, thus violating the mandatory stuffing rule;
- *CRC Error*: it is detected if the computed CRC is not the same as the received one;
- *Form Error*: all the fixed-form fields of any received frame must have the expected structure;
- *Acknowledgement Error*: it must be flagged by the transmitter if no dominant bit is monitored during the corresponding acknowledgement slot.

A supervising module is required in any bus controller to disconnect the corresponding terminal if proven faulty multiple times, in order to keep the bus available for all the LRUs that are working correctly. A transmit error count and a receive error count are incremented whenever an error occurs following a precise algorithm, specified in [8], to give a suitable weight to each type of error, and the node is disconnected when its counter gets higher than a predetermined limit. If no error is detected, these counters are decremented and the node might eventually be reconnected to the bus. In normal working conditions, the controller is *Error Active*, meaning that, since everything is working correctly, it can participate in error detection generating error frames. If the transmit or the receive error count reaches a value of 128 the controller must be set in an *Error Passive* condition, in which it can still interact with the bus but it can only generate passive error frames, composed only by recessive bits and that consequently does not perturb the operation of other nodes. If further problems occur the controller is set as *Bus Off* so that it cannot interact with the bus any more.

Overload Frame

Another feature included in the CAN specification is the Overload Frame: whenever a node is not in condition to perform a new reception or whenever the bus is too busy, this kind of frame can be generated to reduce bus usage and let all the controllers be ready for the next communication. An overload situation can occur also if a dominant bit is monitored on the bus at the first or second bit of an *Intermission*, or in the delimiter of an error or overload frame. If the controller or the system connected to it need more time before being available for a new reception, they can start an Overload frame at the end of an *Intermission* or when the bus is *Idle*.

CANaerospace and ARINC 825: analogies and differences with standard CAN

The avionic industry opened its doors to CAN thanks to the reduction of cabling it allows, to its efficiency when working with LRUs, and its reliability. Although this technology already gave satisfying performance, some modifications were identified to fully adapt this protocol to airborne systems, bringing to the development of CANaerospace and ARINC 825. Both these avionic adaptations of CAN are based on the ISO 11898 to ensure interoperability between them and CAN as well.

CANaerospace defines additional ISO/OSI layers 3, 4, and 6 functions, i.e. in the composition of the frame header, to support node addressing, and it also introduces Time Triggered Bus Scheduling. This last feature reduces the maximum bandwidth available for each terminal, forcing it to transmit only in predetermined time slots where no other controller can use the bus. Thanks to this solution data collision is eliminated and the arrival time of each packet becomes deterministic, but inefficient bandwidth utilization is the price to pay. It must be noticed that none of these differences affects the physical layer, nor the data link layer, and consequently a CAN bus controller is compatible with CANaerospace without any modification.

The ARINC 825 specification [9] is a general standardization of CAN for airborne use. Since CAN physical layer already provides error recovery and protection mechanism necessary in avionic systems, no additional functions have been added at this level. Like CANaerospace, ARINC 825 is entirely based on the extended frame CAN 2.0B version. The 29 bits of the extended frame identifier allow the division of the identifier into sub-fields required for the creation of a standardized application layer. 11-bit identifiers may coexist on ARINC 825 buses but they are not required. The communication mechanism is derived from CANaerospace, and similar functions are added to ISO/OSI layers 3, 4, and 6 to support logical communication channels (LCCs), one-to-many/peer-to-peer communications, and station addressing.

Time Triggered Bus Scheduling is adopted in this protocol as well to improve determinism. Because of the extreme similarity with CANaerospace, ARINC 825 does not seem to require modification in the actual bus controller either, there is one difference though: since the time triggered bus scheduling prevents any terminal from starting a transmission during an intermission frame, no overload frame should occur when all the terminals are working correctly. To reduce network loading, overload frames are prohibited by ARINC 825, and situations that would cause their generation are to be considered erroneous.

2.1.2 The AFDX protocol

While presenting the network architecture proposed for the Avio 402 project, it has been stated that AFDX has been chosen to realize the central network of the system because of its two main advantages over other avionic protocols: high bandwidth and the low amount of cabling required. This technology is strongly based on the IEEE 802.3 Ethernet, thus making it possible to benefit from commercial-off-the-shelf (COTS) components, and from all the expertise and knowledge developed for it over the years, consequently reducing overall costs, making system development faster, and maintenance less costly as well. The following description of its features will make clear why this protocol is so promising and will highlight the differences and analogies with the IEEE 802.3 specification. It is important to remember that AFDX was designed to exploit Ethernet COTS components and functionalities, but following the ARINC 429 philosophy, i.e. implying point-to-point communication, known bandwidth, redundancy, and prioritized quality of service, in order to meet the reliability and determinism required in any avionic system. The official ARINC 664 (Part 7) specification [10] relies on many concept taken directly from UDP, IP, and Ethernet protocols.

Virtual Links

The first, and maybe the biggest, difference between AFDX and Ethernet is the concept of Virtual Link (VL): Virtual Links are independent virtual connections that share the same physical medium. They are point-to-point communications, but while in ARINC 429 each one of these requires a physical wire to connect its source with all the destinations, in this case they all coexist on the same star topology network, with a consequent reduction of the amount of required wiring. The virtual point-to-point communication channels are emulated on the network by allocating a limited bandwidth for each one of them: each node can transmit packets corresponding to a certain VL only in a predefined temporal window dedicated to that precise connection. This transmission window is called Bandwidth Allocation Gap (BAG) and allows various VLs to coexist on the same network without interfering with each other. In

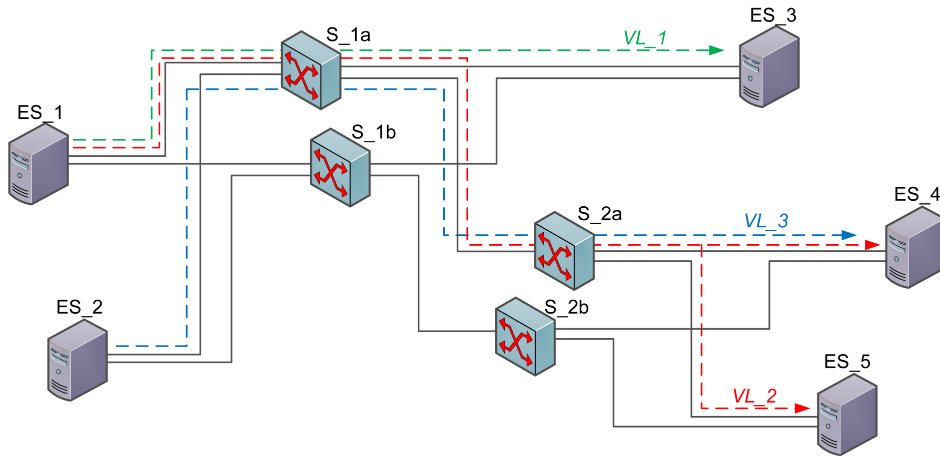

Figure 2.2 AFDX network example

Section 2.2, an example of a small AFDX network is given, showing how different VLs coexist and how redundancy is used to increase overall network reliability. Each ES communicates with two separate and redundant network in order to ensure data delivery, even when a critical problem makes one of them dysfunctional. Switch S_1b and S_2b constitute the redundant network B, that can have a different topology, but that must support the same VLs included in network A.

Flow/Traffic control

The BAG concept introduced by AFDX ensures guaranteed bandwidth and allows end-to-end delay reduction. In absence of Jitter, this parameter is defined as

> *...the BAG represents the minimum time interval between the first bits of two consecutive frames from the same VL.*

The BAG value is expressed in milliseconds and it must be 2^nms, with a minimal value of 1ms and a maximal one of 128ms. Whenever the scheduling of multiple VLs introduces jitter on their transmission, the BAG windown is always referred to the beginning of the transmission of the first frame, as shown in Figure 2.3.

In addition to the Bandwidth Allocation Gap each VL is assigned also another parameter, called L_{max}, that represents the maximum frame length, in bytes, that can be transmitted on that VL. In the official ARINC 664 documentation the maximum frame size S_{max} is often used instead of the length, and it corresponds to L_{max} plus the intermission and preamble fields, i.e. it is equal to L_{max} plus 20 bytes. This upper bound on the frame size limit the time taken for its transmission on the medium, and together with the BAG it determines the

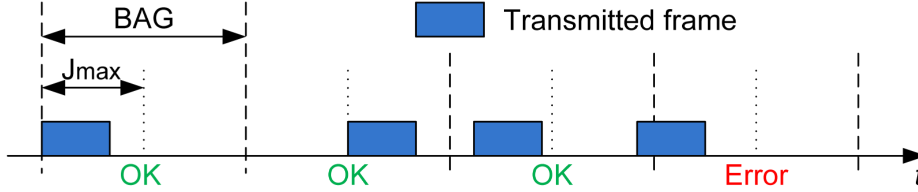

Figure 2.3 BAG and Jitter in ES transmission

resulting maximum bandwidth allocated to the VL. A lower bound S_{min} is specified also. By limiting the rate at which frames can be transmitted on a virtual link and the size of these frames, a sort of isolation mechanism is created, to prevent any VL to interfere with other Virtual Links managed by the same source node. Non-optimal bandwidth usage decreases switches load thus reducing the jitter and delay added by them to the communication.

The last parameter that characterize the flow of frames at the output of the End System is the Jitter, which is the deviation, introduced by the scheduler, from the expected transmission time. A maximum value for this parameter must be guaranteed since it is hazardous for determinism, this upper bound is determined by the following equation.

$$\begin{cases} max_jitter \leq 40\mu s + \dfrac{\sum\limits_{i \in setofVLs} (20bytes + L_{max}bytes) \times 8bits/bytes}{Nbwbits/s} \\ max_jitter \leq 500\mu s \end{cases} \tag{2.1}$$

In Figure 2.3, an example is shown where various frames are sent respecting and not respecting the maximum Jitter defined for their VL; a situation like the one represented by the fourth frame must be avoided.

Frame structure

The structure of the AFDX frames, shown in Figure 2.4, is mostly identical to the one defined in IEEE 802.3 for Ethernet frames, in order to keep these two protocols as compatible as possible. The biggest differences relies in how the MAC addresses are composed and used and in an additional field at the end of the payload that contains the frame Sequence Number.

The Sequence Number is used to guarantee data integrity: the ARINC 664 specifications determines that data order for each VL must be respected, i.e. packets must be sent and received in the correct order. To ensure protocol compatibility and make the sequence number “invisible” for a standard Ethernet network, it is added at the end of the payload field, as if it were part of it.

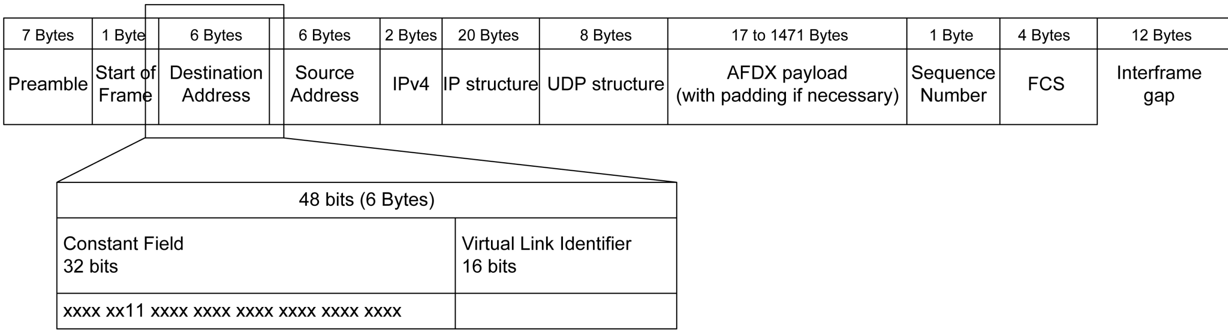

Figure 2.4 AFDX frame structure

The MAC addresses structure, both for the source and the destination, includes information about the type of data contained in the frame and the VL on which the data must be sent. The peculiarity of the destination MAC address is that it is always defined as multicast (the 8th bit of the constant field is set to 1), and the last 16 bits determine the VL the packet must be sent on, as shown in Figure 2.4. In the source address, with some constant bits, there are also an interface identifier and a used-defined ID as well.

More information about how these features can be translated in a physical implementation of an ES is given in Chapter 4, while, in Chapter 5, the peculiar functionalities that must be provided by the AFDX switch, and that have not been discussed in this overview, are presented.

2.2 Literature review

As mentioned before, various aspects of the two considered technologies have been explored in the last few years, confirming the attention that grows around them. Studies are being performed to identify the potential for CAN-based protocols in avionic networks, and Young *et al.* [11] underline how these technologies are suited for legacy line replaceable modules. The work presented by Zhang *et al.* [12] identifies CAN-based protocols as the ideal choice for unmanned helicopter systems. Even more promising is AFDX, whose innovative features opened multiple research perspectives: some examples are the analysis of the worst-case end-to-end delay [5, 13], the identification of the most performing scheduling algorithm for ES and Switches [6], or again the network modelling and its reliability analysis [7].

While a lot of attention is reserved to these topics, which are also explored in the Avio 402 project, only limited studies have been published concerning the physical implementation of these two protocols. In the following sections, a review of the proposed design and solutions for the system developed in this thesis is provided, but because of the lack of existing material,

especially on ARINC 825 controllers and AFDX switches, attention has been paid also to designs conceived for CAN and Ethernet. Since the considered systems are significantly different from each other, the review of the existing literature is being done separately for each one of them.

2.2.1 CAN bus controller

The CAN protocol has been widely used for several years now and many implementations can be found in the existing literature. Unfortunately, no adaptation of these designs has been made to port them in an avionic environment, and no ARINC 825 nor CANaerospace controller has been published so far. As a consequence an analysis of these two avionic protocols and of related commercial products has been done in parallel with a review of available designs conceived for CAN bus controllers, to understand what knowledge can be reused and what should be added to a CAN design to adapt it for an airborne use.

While describing the differences between these two avionic protocols and the standard CAN protocol in Section 2.1.1, it has been highlighted how most of the new features have been added in ISO/OSI layers that do not affect directly the design of the bus controller (layers 3, 4, and 6). Nevertheless, some of these alterations are reflected in a possible optimization of this controller since some functionalities usually necessary in a standard CAN bus are not needed any more.

The HI-3110 by HOLT [14] is a recently released integrated controller for avionic CAN bus, which can comply with both ARINC 825 and CANaerospace standards. The internal structure is based on two parallel paths for transmission and reception, while a separate Error and Status Control module generates interrupts when problems or unusual behaviours are detected. This controller can handle standard and extended frames, as determined by the CAN 2.0B specification, and, when configured to be used in an ARINC 825 environment, it does not initiate overload frames since they are prohibited by the specification to reduce bus loading. According to the data sheet, there is no other difference, other than the management of overload frames, between the ARINC 825 and the CAN configurations.

Multiple architectures and designs have been developed for CAN bus controllers over the years and, because of the age and diffusion of this standard, a good level of maturity has been reached. Probably the most popular implementation of a CAN core for FPGA is the HurriCANe IP core by Stagnaro [15] that has been widely used over the past years since it was originally open source. It provides bus synchronization and two separate modules for the transmit and receive paths, which share a single module for CRC computation. Error count and error frame generation are handled by two separate modules. Although this core used to be open source and free, its code is not readily available anymore, thus it cannot be

used as reference system in this work. The overall system size on a Xilinx Virtex-II FPGA is 1047 LUTs and 715 registers (FFs), and it can run with a maximum clock frequency of 88MHz with this device of speed grade -6.

Among other designs and implementations proposed over the years, some were inspired by HurriCANe; for example, Reges and Santos [16] suggest CAN as a suitable protocol to handle smart sensors, and propose a VHDL implementation of a CAN core based on the same architecture of HurriCANe, called MARIA. Their goal is to have an inexpensive solution to replace IPs provided by manufacturers that are generally not free. Error management and synchronization are not included in their core, that has a size comparable to the corresponding parts of the HurriCANe core. TinyCAN [17] is another interesting design thanks to the extremely small size: less than 200 slices and 500 LUTs are used by this core. This core is only an implementation of the MAC layer alone, and no header composition/decomposition and CRC control is executed. Its structure is based on a single "Bit Stream Processor" (BSP) that handles both the tx and rx routines, differently from the previously described solutions. To further reduce the system size, the error control and recovery procedures have been implemented in a different way from what is suggested in the official CAN documentation. The same architecture based on a single BSP is exploited by a commercial product developed by HiTech Global, that occupies around 1500 slices on a Xilinx FPGA for the complete controller. This design can work at 60MHz on a device with a -4 speed grade, or at 100MHz with a -6 speed grade.

Ideas can be taken from all these designs to develop the CAN controller needed for the AVIO 402 project, and the need for small size and ARINC 825 support must be the key parameters in the process. For example, while the unique BSP of TinyCAN seems suitable for reconfigurability, since it contains all the differences between CAN and ARINC 825, the modified error controller proposed in the same paper is not a viable solution, because specifications must be strictly followed when developing avionic systems.

2.2.2 AFDX End System

The AFDX End System must realize the interface between the avionic subsystem and the main AFDX network, providing frame encapsulation and traffic management. Multiple applications/subsystems can access and exploit the same ES that must consequently be able to handle data coming from different ports. Some solutions have been proposed to implement this kind of system, while research is still in progress to determine optimal scheduling and optimal management of traffics of different nature and priorities. From the development point of view, the most important choice concerns the implementation approach to be used: both software and hardware solutions are possible and a few examples exist in the literature.

It has been previously stated that one of the advantages of AFDX is how it handles communication ports, that are always used for exchanging data between applications in avionic systems, where the ARINC 653 specification must be met. This document, whose importance is described for example by Prisaznuk [18], determines how operating systems should work to ensure temporal and physical segregations of the various applications, and ports are the only way this otherwise independent environments have to communicate. Applications must be expressly developed to be supported by ARINC 653 compliant OSes, and Kinnan *et al.* [19] state that it is generally easier and more efficient to directly design an application considering this context than porting an already existing code for ARINC 653. This is due to multiple issues that must be faced when adapting the code for this specification.

Most of the reported works about possible implementations of an AFDX End System suggest a software approach; for example, Khazali *et al.* [20] give an analysis of their experience when implementing a commercial ES and support the argument that a software solution has many advantages over its hardware counterpart: it is faster to develop and has lower costs, it can be portable on different platforms, it is more flexible and easily modifiable, and recent embedded processors usually have the necessary computing capability to meet the required performance. In their case, they worked with the A-Stack, a certifiable protocol stack developed by Embvue, using the XPedite1000 board as development board; this board exploits a PowerPC 440GX processor running at 666MHz. The modified A-Stack processing time is up to 11.8μs, and the jitter introduced by the VL scheduling is up to 377μs for maximal sized frames in a 32-VLs system.

Another possible software implementation based on a soft processor on a Xilinx FPGA is presented in [21]. The device used is a Xilinx Spartan 3AN FPGA, while the embedded processor is a MC8051 open core microcontroller. The design employs the processor as system controller and protocol processing unit, and the interface between this central core and the dual redundant physical port is realized using two reduced media access controllers (MACs). Two dual-ports RAMs (DPRAMs) are exploited to simplify the data exchange between the processor and the MACs. The processor must handle the avionic application system as well as the AFDX protocol stack, whose UDP and IP layer protocols are inherited from the standard TCP/IP protocol, while its Virtual Link layer is unique. This virtual link layer must implement all the features defined in the official documentation [10], and handle transmission and reception with two parallel paths. Unfortunately, details on how this has been implemented, and on the actual results that have been obtained are not given.

Chen *et al.* [22] propose a full software solution of a feasible framework to implement the AFDX protocol on the VxWorks Operating System. A single task framework has been chosen over a multi task one, which would handle protocol processing and VL scheduling

separately, because of the intrinsic determinism in the execution time. A description of the data structures used for the transmitting and receiving processes is given and some interesting considerations are done: the profiled nature of the AFDX network simplify the TCP/IP protocol since some modules like ICMP and ARP are not required.

In his MS thesis Erdinç [23] discusses the development of the AFDX protocol stack with a standard PC and Ethernet card to develop a platform for test and validation of this protocol. He created some specific DLLs to modify the execution flow followed when communicating with the network. With this kind of implementation, he attained good performance in terms of maximum number of VLs that the system can handle, and he obtained an average jitter of about 250μs.

Maybe the most interesting work on the development of an AFDX End system has been done by Pusik *et al.* [24] during a workshop for NetFPGA developers, since a comparison between a full HW and a SW implementation has been done. NetFPGA is a Linux based platform, thus a hard real-time environment is not possible, but it allows fundamental implementation and evaluation nevertheless. The software implementation they propose is based on the modification of the low level drivers that control the Ethernet MACs, while the HW solution exploits the processor only to provide the payload to be sent and to determine when to start a transmission. The HW implementation requires 14,960 slice Flip-Flops and 18,957 4-inputs LUTs on the xc2vp50 FPGA mounted on the NetFPGA board, which is around 40% of the available resources. The results obtained with these two implementations are fairly similar, and only a slight improvement in the transmission jitter results from a full hardware implementation thanks to its hardware timers. Unfortunately, the non real-time nature of the operating system that handles the data flow in both cases causes excessively large delays in some occasions, especially when the BAG is 1ms. In that case the maximum jitter can reach 9.5% of the transmission window.

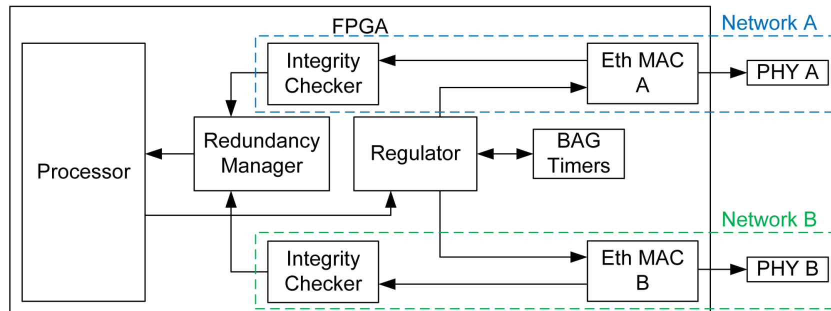

Figure 2.5 HW implementation of the ES

While the previous papers attest that it is possible to achieve the performance required

by the specification, they also underline that attention must be paid to reach these results in an embedded environment. Since a Xilinx board has been chosen for this project, useful information can be found also in the white paper XAPP1130 [25] where various solutions for the problem are proposed, and both hardware-centric and processor-centric designs are suggested. The suggested architecture for both design approaches resembles the one proposed in [24]. The only difference resides in the choice of which functional modules should be realized in hardware and which ones should be executed by the processor instead. Figure 2.5 shows a possible architecture for this core, inspired by the hardware-centric solution of the Xilinx white paper: the *Regulator* handles the BAG for each VL during transmission, *Integrity checker* and *Redundancy manager* realize the features required for the receive path, and the processor is in charge of the application and TCP/IP stack execution. The functional modules here suggested as hardware blocks can be integrated in the stack managed by the processor, transforming this design into the processor-centric solution proposed by the same white paper and in the previously discussed papers.

2.2.3 AFDX Switch

Although it is a critical and fundamental part of the network, the AFDX switch does not have a developed literature concerning its implementation or optimization, such as in the case of the End System. While there are various papers studying end-to-end delays, optimal scheduling algorithms, or simulating techniques for AFDX networks, no article has been found that addresses the development or the architecture of the switches, nor about which techniques should be adopted to minimize technological latency and jitter in this system. In Section 2.1.2, peculiar features of the AFDX protocol have been explained, and it has been highlighted how they make this protocol different form Ethernet even if it remains strongly based on it. The switch reflects these differences: the profiled network, the presence of critical frames with high priority in the network, fault detection and confinement, and latency control are some of the features that should be added to a typical ATM switch design.

Because of the lack of information about this topic, a review of the existing solutions developed for the implementation of Asynchronous Transfer Mode (ATM) Ethernet Switches has been done to identify which architectures could be most suitably adapted to the AFDX protocol. The most common architecture are presented here in order to give a general view of the state of the art.

Ethernet switches

Various architectures have been proposed over the years for the implementation of ATM Ethernet switches, each one with its own advantages and problems, and thanks to the maturity of this technology, all these designs have already been studied and optimized. Chao and Liu [26] propose one possible classification of Ethernet switches based on their switching techniques: time-division switching (TDS) and space-division switching (SDS), the former is further divided into shared memory type and shared medium type, while the latter is divided into single-path switches and multiple-path switches, which can in turn be further divided into several other types.

In time division switching, packets from different inputs are multiplexed and forwarded through a single data path connecting all inputs and outputs. The maximum number of ports is limited by the internal communication bandwidth that should be as high as the aggregated bandwidth of all the input ports. In a shared-medium switch, incoming frames are time-division multiplexed into a common high-speed medium, such as a bus or a ring, of bandwidth equal to N times the input line rate. Shared-memory switches save them in a central memory instead, before scheduling them towards the corresponding output. Shared-memory structure is better in memory utilization than the shared-medium structure, but requires higher memory speed.

Space division switching is based on a structure where multiple data paths are available between input and output ports, thus offering the possibility of concurrent forwarding of frames when no blocking is present. The total switching capacity is the product of the bandwidth of each paths and the number of paths that can be used simultaneously for the forwarding operation. This type of switch is classified in function of the number of available paths between any input-output pair, the two main structures are single-path and multiple-path. While the first one has simpler routing control, the latter has better connection flexibility, non-blocking features and fault tolerance. Some of the existing structures for single-path switches are crossbar-based switches, fully interconnected switches, and banyan-based switches. Augmented banyan switches, multiplane switches, Clos switches, and recirculation switches are, on the other hand, some of the most popular multiple-path architectures.

Incoming traffic is often unbalanced and burst-based, therefore contention of output ports and internal links is likely to happen, and buffering techniques become necessary to store frames that lose the contention. The most common buffering strategies are shared-memory queuing, output queuing, input queuing, and combined input and output queuing. In shared-memory queuing a single memory is used to store frames coming from all input ports, dividing them into virtual queues, and memory utilization is maximized, while switch size is limited because all the input and output ports must read/write to one single memory, whose access

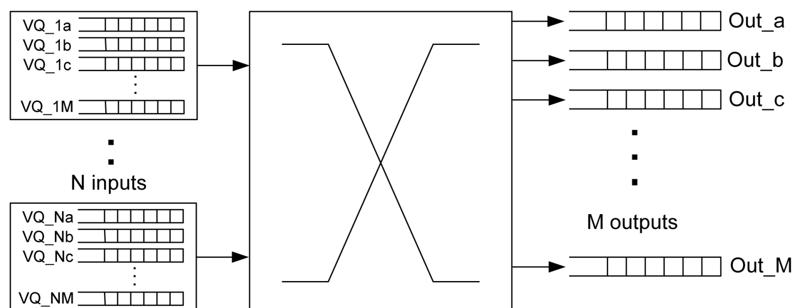

Figure 2.6 Combined input/output buffering structure

time limits the number of ports N. In output queuing structures, incoming packets are immediately forwarded to the corresponding output ports' buffers. This approach provides better QoS, but lacks in memory utilization and maximum number of ports, since it must be possible to forward packets from all the N input ports to the same output buffer in one time slot. Input queuing attracted much more attention in recent years because of the size limitation of the previous architecture: each input port has its own FIFO, and a scheduling algorithm is adopted to choose which packet must be forwarded to the desired destination. One of the problems of this structure is the Head-Of-Line (HOL) blocking that limits the throughput of this type of switch since the head of each buffer blocks all the following frames saved in that same FIFO. The most often adopted solution to eliminate this problem is the Virtual Output Queuing (VOQ): each input buffer is divided into N logical queues, one for each output port. This solution unfortunately increases the complexity of scheduling algorithms, and its optimization is a very hot research topic and various scheduling algorithms have been proposed, for example in [27, 28, 29]. The results published by Karol *et al.* [30] prove that input-buffered switches, although advantageous from a size and resources point of view, limit the maximum throughput and add a bigger delay to communications when compared with output-buffered architectures. For large numbers of input ports, the mean waiting time of input-buffered switches explodes for offered loads that are half of those that cause the same phenomenon in the second type of routers. A combination of these last two solutions is often adopted to reach a tradeoff between throughput and switch size; this solution is represented in Figure 2.6.

Migration towards AFDX

As mentioned before, a review of the literature concerning Ethernet switches has been done because of a lack of a similar literature about the AFDX counterpart. In Chapter 5, some considerations on the previously presented architectures is done to highlight which charac-

teristics developed for these kind of switches are the most suitable for an avionic application. The innovative features of AFDX were not considered when these designs were conceived, and while throughput and support for a wire speed higher than 1Gbit/s drove the design choices made for Ethernet switches, latency and jitter minimization must be prioritized when developing the AFDX swithc fabric. Also, although DiffServ capabilities are considered in Ethernet routers, it is mandatory to have a mechanism to deal with the two priority levels of AFDX, where low priority frames must not interfere with critical ones.

CHAPITRE 3

CAN BUS CONTROLLER

The development of the CAN bus controller is discussed in this chapter. As already mentioned, the design of this system started when it was still uncertain if the prototype would be based on CAN, CANaerospace, or ARINC 825, therefore it was decided to design a CAN controller that would simplify potential future readjustments aimed at tailoring it for one of the other two protocols. Thanks to the great similarities presented in Section 2.1.1 between the CAN protocol and its two avionic adaptations, it was possible to conceive a design that would support all of these technologies, exploiting the fact that the main differences between them only concerns the 3, 4, and 6 ISO/OSI layers. Since the controller must implement only the physical and the data link layers (layer 1 and 2 of the ISO/OSI model) it is mostly unaffected by these modifications. What really affects the design of this system is the elimination of overload frames from ARINC 825 networks, while the choice of using only CAN 2.0B and removing remote frames only makes standard CAN controllers oversized compared with their avionic counterparts.

3.1 Specification and requirements

Following the requirements provided in the official specification, it has been determined that the functionalities that this controller must provide are the following:

- Auto-synchronization with the bus communications
- CSMA/ASM techniques for transmission priority arbitration
- Stuffing and destuffing
- Received frame analysis and CRC validation
- Error detection and error status management
- Frame composition for transmission

Traditional CAN transceivers, such as the SN65HVD233 mounted on the ISM networking FMC module, usually only requires two serial 1-bit signals for outgoing and incoming data, therefore the developed IP core must provide this type of interface. Since no synchronization

mechanism is generally present in the transceiver, it is necessary to include it in the design of the core.

Bus synchronization must always be active and provide precise information about when the bus value should be read, so that destuffing and frame composition can be executed in real time while reception is in progress to immediately execute CRC validation and error detection. Outgoing data must be retrieved in the transmission buffer together with the corresponding header, CRC must then be computed and added at the end of the frame; the completed frame is stored waiting for the bus to be idle before a transmission can be initiated. While the transmission is in the arbitration period the bus value must be monitored to verify that no frame with higher priority is being transmitted at the same time. While executing transmission and reception procedures the controller must also continuously provide supervision for error detection, to set the node state as “error passive” or even “bus off” when a faulty behaviour persists.

Since this IP needs to be instantiated multiple times in each TIM and each NCAP module of the network for redundancy purposes, as shown in Figure 1.3, the resulting size becomes a critical parameter and it must be kept as small as possible to reduce the area and resources taken by this core in the final system.

3.2 Design

This module has been developed with other colleagues in order to accelerate its implementation and testing. Its design and architecture keep this aspect into consideration to facilitate group work and task share-out. In the following sections, the IP architecture is described after some considerations on how to best implement the CAN bus controller are given.

3.2.1 Hardware vs. Software considerations

As already mentioned in Section 3.1, size is an important parameter to be kept into consideration when developing this module since it will be instantiated multiple times and its area will consequently limit the maximum number of redundant paths that could be integrated in the FPGA. Exploiting a processor to run a software CAN protocol stack is an excessively oversized approach for the implementation of this module since the protocol presents a pretty simple frame composition/decomposition procedure. Furthermore, a hardware module would be required anyway for precise synchronization with the bus. It has been consequently decided that a full hardware solution realized in VHDL would be the most suited approach for the development of this core.

3.2.2 Architecture

The internal architecture of the controller is based on two independent paths for frame composition and bus interface management, in order to allow the generation of a new frame for transmission while reception is on progress. Transmission and reception routines, on the other hand, cannot be separated because of the carrier-sense mechanism adopted by the CAN protocol. Each module of the system is responsible for a single feature that the controller must provide in order to facilitate independent testing and implementation, thus simplifying its design. The two separate paths are inspired by the architecture of the HurriCANe core [15], but a central management unit has been added to facilitate reconfigurability for ARINC 825.

The reception path requires bus synchronization, destuffing, deserializing, and CRC validation before the payload can be passed to the output buffer. On the other hand, frame composition, CRC computation, and serializing are necessary for data transmission. A central Manager is responsible for the management of the general execution flow while an independent supervising module provides error detection and control. In the overall architecture presented in Figure 3.1, all these modules are visible; the receive path is on the top part of the picture, coloured in green, while the transmit one on the bottom, composed by the orange blocks. The blue arrows represent the state of the Finite State Machine (FSM) of the *Manager*, that controls the behaviour of the other modules, and the red arrows show the error flags: the *Bitcheck* detects bit acknowledgement errors, the *Destuffer* form errors and stuffing ones, and finally the *CRC_rx* must evidently recognize CRC errors.

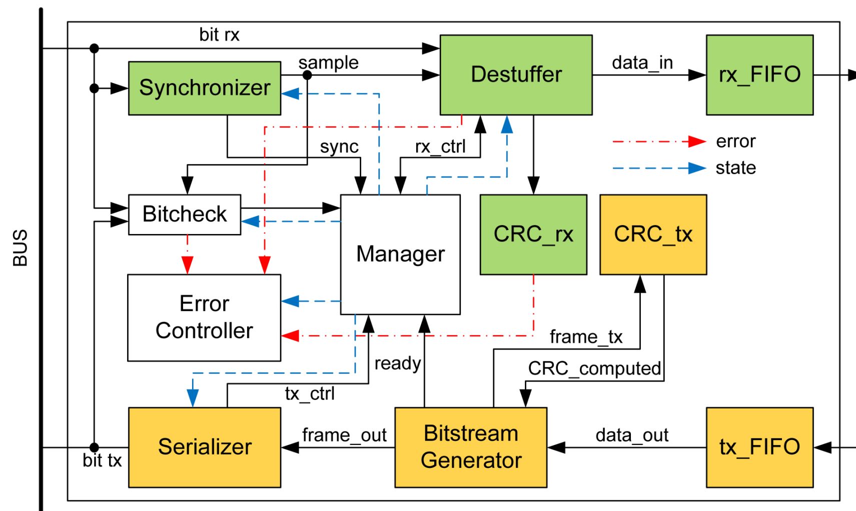

Figure 3.1 CAN controller internal architecture

The *Synchroniser* module implements the synchronisation algorithm presented in the

official specification [8], where each recessive-to-dominant transition of the bus is used as a reference to determine the right sampling time since no clock is transmitted with the data. The `sample` signal tells the *Destuffer* module when to sample the input value and the *Synchronizer* must anticipate or postpone it whenever a bus transition occurs too early or too late compared to the expected moment. A different signal called `sync` is used to control the FSM execution flow: while the sampling moment is placed around the middle of each bit time, the synchronization signal identifies the start of this period. The *Destuffer* must then sample the bus value when the synchronization signal is asserted, and save each bit in an array that will contain the entire received frame. As it name states, this module must also execute a destuffing algorithm to discard all the stuffing bits added to the frame by the source of the communication. The `data_in` array contains the received frame which must be passed to the NCAP/TIM, that still needs to analyse the header to determine if the received data are addressed to it or not.

Frames to be sent are passed to the CAN controller using the *tx_FIFO* and are retrieved by the *Bistream Generator*, which composes the frame and completes it by adding the Frame Control Sequence at its end, before forwarding it to the *Serializer*. Once the CRC has been computed by the corresponding block and the outgoing frame is ready to be sent, the *Bitstream Generator* advises the *Manager* that a new transmission can be performed setting the `ready` signal. The *Manager* initializes a new transmission when the bus is idle by performing the corresponding routine.

The Error Controller supervises the entire core observing all the error signals generated by the various blocks to determine in which operative state the controller should operate: error active when everything works fine, error passive or even bus off when too many problems occur. The transition between these operating states is performed following the algorithm specified in [8]. Error detection is assigned to every single functional module because it is easier for each of them to recognize any deviation from the expected behaviour, significantly reducing the complexity of the supervising unit. The Bitcheck block, in addition to the trivial task of comparing the transmitted value with the one read on the bus, exploits the resulting information to detect acknowledgement and bit errors and to perform arbitration.

It can be noticed that, differently from the HuriCANe core and the other controllers found in the literature, two CRC modules have been included in the design instead of a single one, the first is dedicated to the transmit path and the other to the receive path. Although this design choice increase the overall core's size, it has been considered a small price to pay for segregating frame construction from packets reception. In [16], the Reges and Santos show that both in their core and in the HurriCANe controller the CRC module size is at least 10 times smaller than those of the transmit or receive modules, thereby its impact on the

overall size is limited. The internal structure has been evidently inspired by the HurriCANe architecture, but a central management unit has been added to make the design more easily adjustable to support ARINC 825. The *Manager* is the only responsible for the generation of Overload frames, therefore it is sufficient to change the corresponding branch of the FSM to disable them and generate Error frames instead. The details on the implementation of the Manager are given in the next section, where the system implementation is discussed. Since TinyCAN managed to attain an extremely reduced size with an architecture based on a central BitStream Processor that handles both transmission and reception, a similar optimization of the resources used by the core in the FPGA can be sought.

3.3 Implementation

In this section, the system implementation is described and, to understand how the design exposed in the previous section can effectively realize a CAN bus controller, the FSM execution flow is presented as well; also, some synthesis results obtained before and after the modifications to support ARINC 825 are given.

In Figure 3.2, the flow chart of Finite State Machine of the *Manager* module is represented. For clarity purposes, it has been separated into two parts, one corresponding to the standard flow, and the other to the error and overload flow. From any of the states included in Figure 3.2(a), it is possible to move to the `err_active` state (or `err_passive` when the node is in an "error passive" condition) when an error is detected and the error flag is set by the *Error Controller*.

When the system is turned on, or after a global reset, the *Manager* is in the `Reset` state, where all the signals generated by this module are reset to the default values. The following state is always the `Wait_Idle` one, which is necessary to determine if the bus is idle or if a communication is on progress: whenever more than 10 recessive bits are read subsequently on the bus the FSM switches to the `idle` state. Ten bit times corresponds to the length of the *End of Frame* field plus the *Interframe*, thereby after them, the bus is necessarily idle because it is not possible to have this many consecutive recessive bits during a communication (thanks to the bit-stuffing). The `t_sync` signal is used to count how many bit times have passed, it is in fact configured to have the same frequency as the bus. The `bitcheck` signal can be used to determine if a dominant bit is received, in fact, since a recessive bit is sent while the *Manager* is in this state, a dominant value on the bus would cause this signal to be equal to '1'. Once the FSM reaches the `Idle` state, three things can happen: a frame is waiting for transmission in the *Bitstream Generator* (`data_out_ready` = '1') and a transmission is consequently executed, a dominant bit corresponding to a *Start of Frame* is received and

the reception branch of the flow chart is executed, or an overload request is received by the system connected to the core, that cause the generation of the corresponding frame. There is also a fourth possibility, in fact if the error state of the node is "Bus Off", the FSM switches to the corresponding `Bus_Off` state, where it remains until the error state changes.

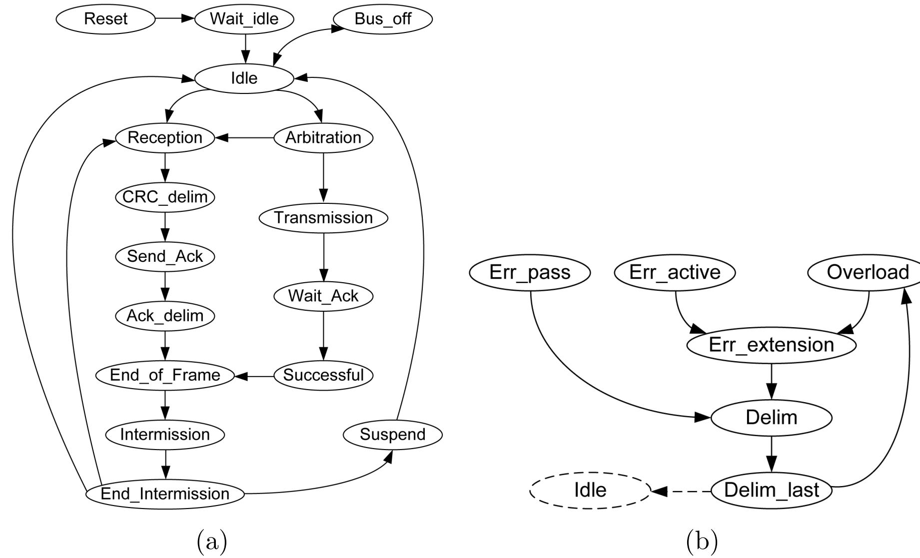

Figure 3.2 FSM execution flow: (a) standard operating state; (b) error and overload management.

If the bus is idle and there is a frame ready for transmission, the *Manager* executes the transmission branch, starting from the `Arbitration` state. In this state, the *Serializer* sends, one bit at the time, the generated frame on the bus, and the *Bitcheck* checks if the same value is received in each bit time. If a recessive bit is read when a dominant one is sent, an error occurred, while if a dominant bit is received when a recessive one is sent, it means that the arbitration has been lost. As a consequence, if the `bitcheck` signal becomes '1' while in this state, the FSM switches to the `Reception` state, otherwise it switches to `Transmission` when the *Serializer* sets the `tx_ctrl` signal to '1'. The *Destuffer*, which was operating as if a reception was on progress when the state was `Arbitration`, stops working when the `Transmission` state is reached. When the *Serializer* sets the `tx_ctrl` signal to '1' again, it means that the data has been sent and that the acknowledge should be received next, therefore the FSM switches to the `Wait_Ack` state. In this state, the value of the `bitcheck` signal is observed because, if it remains '0', no acknowledge has been received (a recessive bit is sent but a dominant value should be received in this period) and an error frame must be generated; the FSM can switch to `Successful` otherwise. The `Successful` state lasts only one bit time and it corresponds to the acknowledge delimiter, therefore the *Manager*

automatically passes to the `End_of_Frame` state when `t_sync` becomes '1' (unless and error occurs and the corresponding flag is set). A `err_pass_transmission` signal is also set to "true" if the error state of the node is "passive". The `End_of_Frame`, `Intermission`, and `End_Intermission` states automatically follows if no error nor dominant bit on the bus is detected: the first lasts 7 bit times, the second 2 bit times and the last one only 1 bit time. The *Manager* must determine how many bus periods elapsed observing the synchronization signal to execute this flow. These three states are separated and not merged in a single one (the only thing to do in these states is sending a recessive value on the bus) because the reception of a dominant bit has different consequences in each case: an error is detected if `bitcheck` becomes '1' during the *End of Frame* period, an overload frame must be generated if this happens while in the `Intermission` state, and the dominant bit is considered a new *Start of Frame* if received during the `End_Intermission` state. In Figure 3.2(a), it can be noticed that, from this last state, the FSM can switch to `Reception`, when a dominant bit is received, to `Idle` when nothing happens and the controller is in an "active" error state, or to a `Suspend` state when the `err_pass_transmission` signal is "true". This suspension state is necessary to allow nodes in an "active" error state to start a transmission, thereby giving them higher priority since they proved themselves less faulty.

If the *Manager* is in the `Idle` state and a dominant bit is received before a frame is ready to be transmitted, the reception flow must be executed. During reception everything is handled by the *Destuffer* and the *Manager* only needs to wait until the `rx_ctrl` signal is set high to switch to the `CRC_Delim` state. In this state, the result of the CRC computation is compared with the received frame control sequence (FCS), and an error is generated if they do not correspond. If no problem is detected, the FSM switches to `Send_Ack` after one bit time, and then waits for another bit time before moving to the `Ack_Delim` state. While in the `Send_Ack` state, the *Serializer* sends a dominant value on the bus, and a recessive value is transmitted instead during the `Ack_Delim` state. This last state has not been merged with the `Successful` one because the reception of a dominant bit in the two cases must be recognized as two different kinds of error. The `End_of_Frame`, `Intermission`, and `End_Intermission` states follow like in the transmission flow, but the `Suspend` state cannot be reached in this case since the `err_pass_transmission` signal has not been set to "true".

From any of the states of the execution flow shown in Figure 3.2(a), it is possible to switch to an error state in case an error is detected by one module of the system and flagged to the *Error Controller*, that consequently sends an `error` signal to the *Manager*. The FSM switches to the `Err_Active` state if the error state of the node is active, or to `Err_Passive` otherwise, to generate the corresponding error frame. Since most of the fields are the same in the two cases and when sending an overload frame as well, the corresponding states are

Table 3.1 Size of the CAN controller modules

Module	# Slice registers	# Slice LUTs
FSM	19	159
Bitcheck	0	4
CRC_rx	17	15
CRC_tx	15	7

used for the generation of all these type of frames. The `Overload` state, as mentioned above, can be reached from the `idle` state in case a "request overload" signal is received by the core, from the `intermission` frame if an unexpected dominant bit is read on the bus, or also from the `delim_last` state of the error flow chart. When the *Manager* state is `Err_Active` or `Overload`, the *Serializer* must force a dominant value on the bus, whereas a recessive value must be sent during the `Err_Passive` state. The `Err_Active` and the `Overload` states last 6 bit times, the FSM must then switch to `Err_Extension` the sixth time `t_sync` is set to '1'. In the extension state a recessive value is transmitted and the controller must wait for all the other nodes connected to the bus to finish their error/overload frame; the FSM goes to `Delim` only when a recessive value is read on the bus (and `bitcheck` is consequently '0'). The "passive" error frame is different: it does not influence the operation of the other nodes connected to the bus, that consequently continue the communication in progress, and it must wait for six consecutive recessive bits, that can occur only after the *End of Frame*. Consequently, a 6 bit-time counter is incremented for each recessive bit received during the `Err_Passive` state, but it is reset to 0 if a dominant value is read on the bus before it can reach the final. When this counter is equal to 6, the FSM switches not to the `Err_Extension` state, but to `Delim` one instead. In this state, 7 recessive bits are sent, and if a dominant value is received a new error frame is generated. The *Manager* finally arrives to the `Delim_Last` state, which lasts only one bit time; if a dominant bit is received, an overload frame must be generated, otherwise the FSM can switch to the `Idle` state.

The entire VHDL implementation of the Finite State Machine that constitutes the core of the *Manager* is provided in Appendix A, where all the transitions and the conditions that cause them can be observed.

When it has been chosen to readjust the design to support only ARINC 825, eliminating CAN from the prototype, the only modules that were completed and synthesized were the FSM, the Bitcheck, and the two CRC modules. The size of these blocks obtained with a synthesis on the same Spartan-6 used for the other systems discussed in this thesis is presented in Table 3.1. The Bitcheck module, thanks to the very basic function it performs,

occupies only 4 lookup tables. The FSM is fairly compact as well even if it is responsible for the management of the biggest part of the system behaviour. As expected, the size of each CRC module is small enough to justify the choice of having two separate modules for processing incoming and outgoing packets separately. The size difference between the two CRC modules is due to the fact that *CRC_rx* receives the frame in a serial fashion, and must also compare the result with the received frame control sequence, while *CRC_tx* has a different interface and it only has to compute it and pass it to the *Bitstream Generator*.

3.3.1 Functional verification

The functionality of the core has been validated with exhaustive simulations at logical level. Each module has been initially tested in detail to verify that all the required features were correctly implemented, and a system level simulation has been finally performed. The core precisely meets all the requirements of the official specification and the performance are largely satisfying: the maximal clock frequency attainable with this implementation is 92.7MHz, a frequency higher than necessary since this core just need to operate with a system clock equal or higher than 10MHz to provide precise synchronization with the 1Mbit/s bus. Because of the low bus wire speed, when the last bit of the frame is read during a reception, before a new communication can start, the received data has already been processed and is already saved in the output register. Error detection, frame analysis, and frame composition if we consider transmission, are executed in real-time, and system throughput is significantly higher than the bus speed, therefore data congestion is avoided.

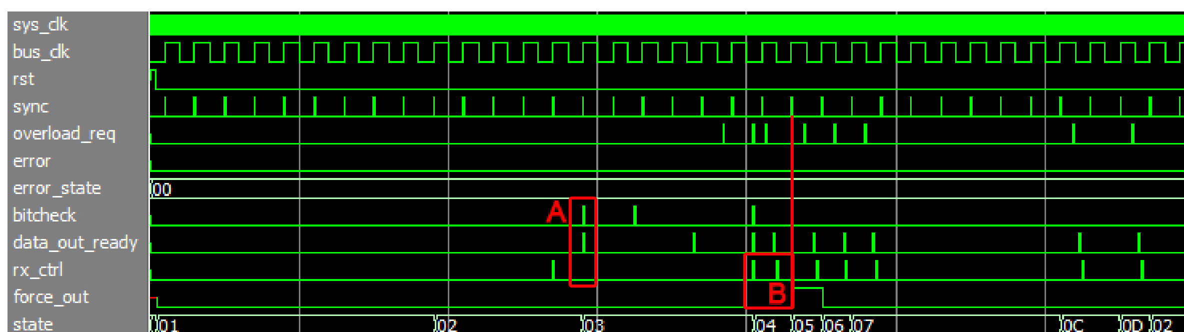

Figure 3.3 Simulation example: FSM reception reception flow

An example of the simulations run for the single module of the *Manager* is provided in Figure 3.3: the execution of the reception branch is verified providing the expected input coming from the *Synchronizer* and from the *Destuffer* (`sync` and `rx_ctrl`), together with the `bitcheck` and the `error` signals that are used to determine which branch of the execution flow must be executed. The error state of the node is “active” (“00”). For each state, the

Table 3.2 Hexadecimal value of each state of the reception branch of the FSM

State	Value
Reset	00
Wait Idle	01
Idle	02
Reception	03
CRC Delim	04
Send Ack	05
Ack Delim	06
End of Frame	07
Intermission	0C
End Intermission	0D

signal expected for the prosecution of the reception is asserted only after having set to '1' all those that should not influence the execution of a successful reception. This is done to ensure that they are ignored when they assume erroneous values. Requests for the generation of an Overload frame and of a transmission are also generated using the `overload_req` signal; as expected, the reception is not blocked by this requests, which can cause the generation of the corresponding frame only during the `Idle` state. The state signal is the output sent to the other modules, and it can be observed, comparing it with the values of Table 3.2, that it correctly follows the expected reception execution flow. When the bus is idle, the `rx_ctrl` is ignored because no reception is in progress, but in the box A in Figure 3.3, a reception and a transmission flag are received concurrently; since this means that a communication is already in progress on the bus, the reception branch is executed instead of the transmission one. During the `Reception` state, the `bitcheck` and `data_out_ready` signals are ignored and only `rx_ctrl` is considered to determine when it is necessary to switch to the `CRC_Delim` state. The box B highlights this reception control signal and the bit time that must pass before the FSM switches to the `Send_Ack` state, using the `sync` signal as a reference. The transitions to the following states is controlled by the synchronization signal as well, and the only other signals that could break this execution are the `error` one and the `bitcheck`, that are always '0' in this period to let the branch reach its conclusion. In each state, all the signals that are ignored, and that consequently do not cause any change in the execution flow, are asserted to verify that this behaviour is reproduced. Similar tests have been performed to verify the *Manager* behaviour in all the expected situations, and the list of the corresponding scenarios is provided in Table 3.3.

The code coverage for the Manager, obtained with these scenarios plus some injected faults (bit flips on the `state` register) is the following:

Table 3.3 List of the scenarios used to validate the *Manager* behaviour

# Test	Value
1	Successful reception (example provided in this chapter)
2	Successful transmission
3	Arbitration lost during transmission, and the following reception is correctly performed
4	Overload, all the situations that can bring to their generation are tested
5	Stuffing error detected during reception. Both error active and error passive situations are reproduced. The stuffing error can also be detected during the reception that follows an arbitration lost
6	CRC error detected. Like in Test 5 both error active and passive states are considered and also the detection after the arbitration lost is verified
7	Bit error detected during a transmission
8	Acknowledgement error detected during a transmission
9	Bus Off state, the system must not respond to the bus requests

```
Coverage Report Summary Data by file

File: C:/Users/Davide/Desktop/Poly/Project CAN/Raccolta VHDL/VHDL/FSM.vhd
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                      124       124     100.0
    Branches                   125       125     100.0
    Conditions                  30        30     100.0
    Fec Conditions              40        40     100.0
    States                      28        28     100.0
    Transitions                148       121      81.7


NEVER FAILED: 100.0%  ASSERTIONS: 499
```

Transitions coverage is really low because most of them are not allowed in the finite state machines, because they corresponds to erroneous behaviours, which cannot occur for standard operations such as those reproduced in the scenarios. Some unexpected transitions have forced generating bit flips on the `state` register, in order to analyze the behaviour of the FSM when the standard execution flow is interrupted; in addition to the inevitable problems these faults create in the system's behaviour, the FSM execution flow can also be broken by them, since the other modules may not provide the signals required for the following transitions, leaving the *Manager* stuck to the state generated by the bit flip. Bit flips have

been generated only on the `state` signal because this is the most hazardous case that can occur. While the error detection of the CAN protocol can detect this faulty situations, some nodes may remain for a long time blocked in an erroneous state, and an additional control mechanism can be added to the design to ensure that the FSM is reset when it does not switch state for a long time. The implementation of this mechanism and further verification of the system, using fault injection techniques, would have required more time than available.

3.3.2 Migration towards ARINC 825

When it was decided to include only ARINC 825 in the final network prototype, abandoning CAN and CANaerospace, the development of this core was not completed yet, therefore, it was possible to immediately readjust the design to realize a tailored ARINC 825 controller instead of completing the configurable CAN controller. This transition has been realized by other colleagues, but an analysis of the required modifications and of the final results is provided here to highlight how the design presented before supported the implementation of this avionic protocol.

Most of the system's modules were already designed to realize features required by the ARINC 825 as well as the CAN controller, and they have been successfully ported, simply completing and correcting the work that had already been started for the CAN core. The core architecture has not been modified, and the role of each module remained the same as in the original design, with the only exception of the *Error controller* since the ARINC specification include some differences at this level to increase the reliability of the system. A research on error detection is also intended to be performed on this system, therefore some adjustments have been necessary to support the related tests. The state machine has been changed to handle the different error control methods, and to eliminate the generation of overload frames. All the other modules have not been modified in their behaviour but only completed and improved in their implementation. The resulting size for each module after synthesis is shown in Table 3.4 as well as the overall occupied area.

The percentages shown at the end of Table 3.4 represent the occupation ratio of the resources on the Spartan-6 XC6SLX45T FPGA used for the prototype implementation. The small amount of resources required for this IP core allows of multiple instantiations of the bus controller to increase redundancy in the NCAP system. The removal of the overload routine and some improvements on the code reduced the FSM size to only 100 LUTs, and the *Bitcheck* modules is now even smaller than before because the detection of acknowledgement and bit errors is now performed by the *Error controller*, which is one of the bigger modules, together with the *Destuffer* and the *Serializer*. The overall size of the IP core remains comparable with those of the already existing CAN controllers. Input and output FIFOs have a depth of

Table 3.4 Size of the ARINC 825 controller

Module	# Slice registers	# Slice LUTs
FSM	25	100
Bitcheck	0	1
Error Control	32	154
CRC_in	16	13
CRC_out	15	7
Synchronizer	12	18
Destuffing	138	194
Bitstream Generator	68	23
Serializer	53	130
ARINC 825 Controller	529 0.97%	757 2.77%

only 1 register for a better comparison with the data found in the literature, where they are not considered in the computation of the resources occupied by the core. Even if a precise comparison is difficult because of the structural differences between the developed system and the existing ones, we can see that the transmit path (*Bistream Generator* and *Serializer*) requires less resources than the TX module of the MARIA core presented in [16], where 434 LUTs and 154 FFs are used, and that the receive one, composed by the *Synchroniser* and the *Destuffer*, is slightly smaller than their RX module, which uses 153 Flip-Flops and 315 LUTs. For these comparisons it has been considered that the *Manager* must be included partially in both computations because it has an active role in both processes. The overall controller is slightly smaller than the HurriCANe core, that counts 715 FFs but also 1047 LUTs, but it must be considered that the removal of overload frames and of the support for standard frames helped the size reduction, in fact, as stated before, a CAN core can be considered oversized for an ARINC 825 bus controller.

The adaptation of the designed core to support only ARINC 825 has been completed and it has been verified. To conclude its development, it has been planned to integrate it in the device and to connect the SP605 board with a commercial test bed for avionic data networks verification. It will be also possible to increase the core reliability by adding a redundancy mechanism, or the previously mentioned mechanism to ensure that the FSM cannot be stuck to a faulty state.

CHAPITRE 4

AFDX END SYSTEM

In this chapter, an analysis of the AFDX End System, of its features and required performances is done, in order to determine the most suitable solution for its implementation and integration in the network prototype. Differently from the other modules investigated in this thesis work, this system is strictly related to other research topics of the Avio 402 project and to other parts of the NCAP system; therefore, its realization cannot be performed independently and must consider the needs and requirements coming from these other aspects.

4.1 Specifications

The main features of the AFDX protocol, its similarities and its differences from the Ethernet protocol, described in Section 2.1.2, will be widely referenced in this chapter, where the End System is discussed. This system must implement all the features specified in the official documentation to provide data encapsulation, traffic control, and Virtual Link management. In addition to the requirements coming from Part 7 of the ARINC 664 specification, further objectives and constraints derive from the environment of this system.

For the realization of the network prototype, the ES is intended to be implemented on the same FPGA where the whole NCAP is going to be integrated, in order to be quickly and easily interfaced with the ARINC 825 controller. Even if the system is intended for a prototype, and consequently it is not required to strictly follow the safety rules imposed on avionic systems, the design choices still need to be eventually applicable in a real avionic system, thus portability must be kept into consideration to allow the reuse of the design solutions and of the developed code.

This module must also be developed in close contact with the gateway (the bridge between the AFDX and the ARINC 825 protocols) and the NCAP services that are part of the NCAP system. Both these functionalities are planned to be implemented as a software executed by an embedded processor to guarantee better flexibility and to study the use of an ARINC 653 environment within this network, as requested by the Avio 402 project. The presence of an embedded processor in the NCAP, in close contact with this core, influences the design choices concerning the ES itself.

4.2 Proposed solution

Because of the complexity of this system and of the tight interconnection with the other modules included in the NCAP, a software approach has been preferred for its development. This solution allows easier communication with the NCAP services and Gateway, that will be run by the same processor, and reuse of the already existing (software) TCP/IP protocol stack. Even if various papers presented in Section 2.2.2 suggest a software implementation for this kind of system, the system complexity alone may not be enough to justify the exploitation of an embedded processor since better results in terms of performance and area on the FPGA could be achieved with a full hardware implementation, but since the aforementioned NCAP services are planned to be implemented in software, the inevitable presence of a processor makes this solution definitely preferable. Because of the processor size and the limited number of Ethernet PHYs available on the SP605 board, it will unfortunately be impossible to include redundant End Systems in the prototype.

Software advantages

A software approach gives multiple advantages in terms of development time and complexity and it is usually a bad solution only if it cannot give the required performance or if the tasks to be performed are not enough to justify the need for a processor that would be an excessive overhead of resources. In this case, since a processor would be needed for the implementation of NCAP services and Gateway, not only this overhead is not a problem, but the interface between the ES and these applications is simplified if it is executed by the processor as well. Furthermore, the results of the publications presented in Section 2.2.2 confirm that FPGA embedded processor can provide the performance required to meet the ARINC 664 specification, and that latency and jitter obtained with a processor-based design are really close to those attained with a custom hardware implementation.

Another valid reason to choose a software approach is given by the analogies between AFDX and Ethernet, already existing software implementations of UDP/IP and Ethernet protocol stacks can in fact be reused, and Application Programming Interface (API) and socket structures developed for Ethernet applications can be exploited as well, saving development time and increasing portability. Building AFDX over the protocol stack used by the Linux kernel for Ethernet communications can save time and guarantee reliable functionality of those parts that do not need modifications, such as UDP and IP layers. Only the lower layers of the protocol stack need to be modified to integrate the specific functionalities required by AFDX, as shown in Figure 4.1

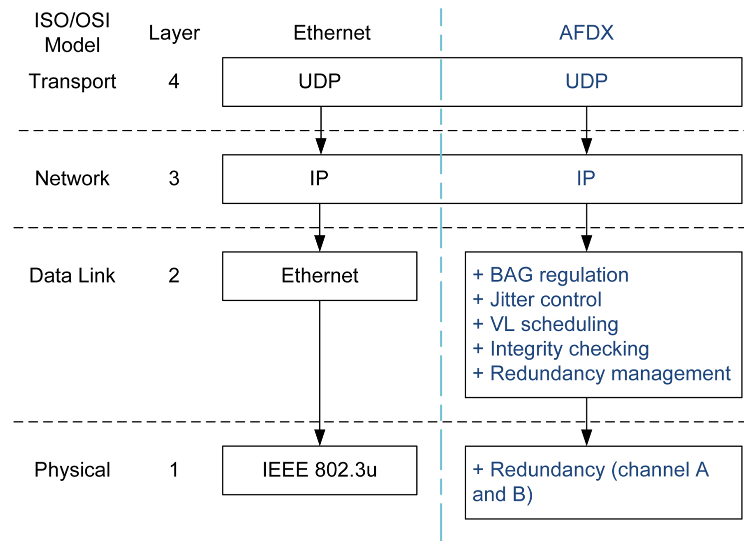

Figure 4.1 SW architecture overview

4.2.1 The Linux Ethernet protocol stack

Since the Linux 2.6.37 kernel has been chosen as operating system, a general knowledge of the structure and organization of the Ethernet protocol stack it provides is necessary to better understand where modifications can be made to successfully add AFDX features to it. The general structure of this stack is the same shown for Ethernet in figure 4.1, and the */net* folder contains all the code required for its implementation. This folder includes anything concerning networking, from IrDA, to wireless communications, from CAN, to Ethernet. The Ethernet networking functions are included in the following folders:

- */net/core*: it gathers all the functions that are use to manage the whole stack execution
- */net/ipv4*: it contains all the code needed for the transport and network layers, TCP/UDP and IP implementations
- */net/ethernet*: for anything concerning the Ethernet layer
- */net/sched*: it includes various proposed algorithms for packet scheduling

The whole networking protocol stack operates on the socket structure created and configured by the application, and passed by it to the first layer via a standard socket API. The various fields of this structure, like the IP and MAC headers, must be filled by the corresponding layer of the protocol stack following the information provided by the application when a datagram socket (the type of socket required for the utilization of the UDP protocol) is created. Once the socket is ready, it is passed to the Ethernet MAC for transmission.

After the UDP and IP encapsulation, the Ethernet layer (*/net/ethernet/eth.c*) must execute a final encapsulation of the frame by adding the source and destination MAC addresses, the frame type, and the final frame control sequence. This layer receives the packet from the IP layer, and determines the destination address by analysing the IP destination address using dynamic routing tables. The source MAC address is easily determined since it is the address of the physical device that will take care of the transmission, while the destination address is set by the application.

When a frame is received, the Network Interface Controller (NIC) generates an interrupt to advise the processor that the frame is available. The assigned driver for the NIC must then retrieve it from the memory where it has been saved and pass it to the Data Link layer. From that point the protocol stack is executed in the opposite way, filling the socket structure fields in function of the information carried by the frame. The packet is finally made available for the application that is listening to the Ethernet communication, which can analyze its content reading the corresponding socket.

4.2.2 End System design

Using this protocol stack as a starting point, the necessary modifications have been identified to add the functionalities required by the AFDX protocol. In this section, the planned design for the development of the End System is presented and a description of the features that must be added to the protocol stack is provided. During the implementation of this design, some problems have been encountered, that suggested a change in the prototype design, thus interrupting the development of this system. The system implementation is discussed in Section 4.3.

In order to reuse as much as possible the existing implementation of the Ethernet protocol stack, AFDX can be implemented on a copy of the corresponding code. A new "Ethernet protocol type" can be created to select this stack instead of the standard IPv4/Ethernet one when the socket specifies AFDX as protocol type. The socket structure could also be modified accordingly to be tailored for this stack. With this approach IEEE 803.2 and AFDX could coexist in the same kernel, and it would be the application to choose which one should be used, creating the socket correspondingly.

Transmit path

The first difference between Ethernet and AFDX concerns the datagram fragmentation: while in Ethernet the Maximum Transmission Unit (MTU) is unique for all the packets destined to the same `eth` device (it can be set using the `ifconfig` command in the Linux terminal), the

AFDX protocol defines a L*max* for each Virtual Link. The `ip_fragment()` function should be consequently modified to use a different value of MTU depending on the virtual link of the frame instead of the destination device. The value of the MTU (L*max*) for each VL should be previously saved in a table accessible by this function.

All the remaining modifications are part of the lower layers, and they can be inserted in both the *eth.c* file or in the driver used to control the physical device since all the features concerning transmission timing should be as close as possible at the end of the stack in order to reduce frames jitter. Device drivers can be found in the */drivers/net* directory, for example the *xilinx_axienet_main.c* and the *xilinx_axienet_mdio.c* files are used to control the Tri-Mode Ethernet Media Access Controller (TEMAC) used in the platform design. Even if BAG timing can be more precise if it is the driver to control it when the packet is passed to the physical device for transmission, this solution is not advantageous from the portability point of view: if the hardware platform is changed, or the MAC core is changed, all the adjustments should be ported to the drivers use by the new design. A better choice is to operate at the output of the Ethernet layer, just before calling the device driver, to remain close to the bottom of the stack while producing a more portable code; therefore, the *eth.c* function and those included in the */net/core* folder should be addressed. The functionalities that need to be included in the protocol stack are the following:

Integrity Checking First, the sequence number must be added at the end of the AFDX payload (defined as the packet delivered to the data link layer by the IP protocol, including the IP header and encapsulation). This number must also be considered as part of the payload from this moment; it is, in fact, taken in consideration when computing the payload length.

BAG and Jitter control Each VL must respect its own BAG: once the frame is complete and ready for transmission the network layer must verify that the time elapsed since the previous packet for this VL was sent is longer or equal to the corresponding BAG, and, if this condition is met, it can make it available for the scheduling and enqueuing procedures. Once the packet has been put in the output scheduler `qdisc` by the `dev_queue_xmit()` function and consequently saved in the output queue, a timestamp of the packet can be checked and saved together with the current time to retrieve information about the introduced jitter.

Virtual Link scheduling The Linux networking protocol stack has at its disposal various scheduling algorithms (gathered in the */net/sched* folder) to decide which socket should be processed and passed to the physical device first, from a basic FIFO mechanism (*sch_fifo.c*), to

generic multi-purpose algorithms (*sch_generic.c*), to more complex techniques. It is necessary to develop a dedicated scheduler which can take into consideration the presence of VLs in the End System. It is possible to implement, in this algorithm, the techniques identified by other colleagues participating in the AVIO 402 project as the most suitable for reducing jitter and overall end-to-end delay, as stated, for example, in the article by Tawk *et al.* [31].

Redundancy management Differently from the previous features, it can be advantageous to manage the two redundant channels at the driver level because, even if this choice produces less portable code, it would be more effective. The driver can in fact simply copy the socket structure to the two ring buffer of the redundant physical devices, while the network layer need to call the `dev_queue_xmit()` routine twice, specifying the corresponding device each time, thus sending the same socket to two different schedulers (each device has its own scheduler) and creating the possibility of having different jitters and behaviours on the two channels.

Receive path

When a frame is received, the corresponding interrupt makes the processor call the `netif_rx()` routine, that saves the received packet into a `poll_queue`. Before saving the received socket in the queue a redundancy and integrity check must be performed. The sequence number of the last received frame of each VL is saved in a table where they can be checked every time a new frame is received, and the socket is passed to the next (upper) layer of the protocol stack only if it is not redundant and if the integrity is respected. Redundant frames must be discarded together with those that do not comply with the sequence order. Once these controls are performed, the reception routine can continue the same way it would have been in the Ethernet protocol stack, no other modification is required.

4.3 End System development

As written in Section 2.2, some possible solutions for the AFDX End System implementation have been published in recent years; unfortunately, most of them do not give applicable solution for this situation. For example, most of the works analyzed in Section 2.2.2 exploit standard Windows machines, modifying some DLLs, or proprietary protocol stacks, while, in this case, the goal is to study an inexpensive, portable solution for embedded systems. Inspiration has been taken from [25] and [24], from which the fundamental structure of the embedded system and of the required software modules has been derived.

4.3.1 Hardware embedded system

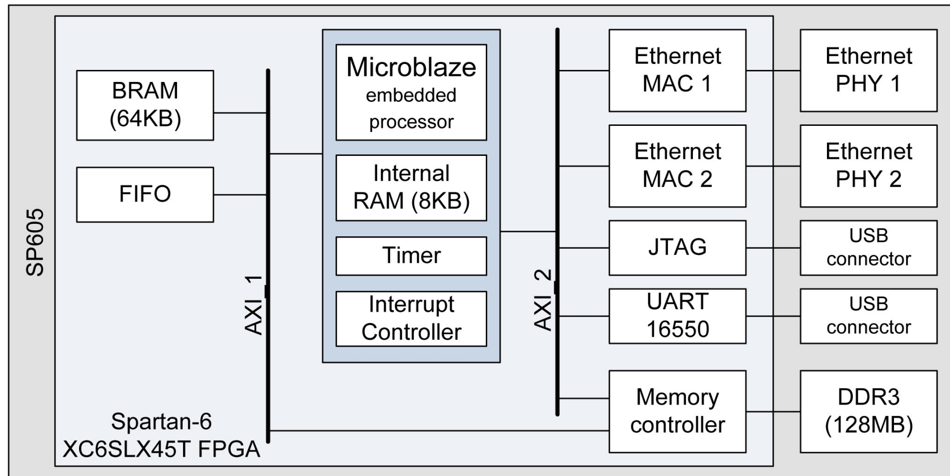

Figure 4.2 ES hardware architecture

The embedded system is evidently processor-centric, in order to support the execution of the software protocol stack described above; the only other hardware modules instantiated are those that are necessary for its configuration, remote control, and network capability. All the exploited IP cores are taken from Xilinx libraries. Since Spartan-6 FPGAs do not include PowerPC microprocessors, a Microblaze soft processor has been configured and instantiated to run the ES and the applications that will be responsible for the gateway and NCAP services. The processor includes an internal RAM memory, interrupt and timer controllers, and a Memory Management Unit (MMU) to support the Linux operative system. The USB connection included in the design is used to control the Linux OS run by the processor using a remote terminal; a second USB port used as JTAG connection is required to program the board downloading the configuration bitstream generated by the Xilinx ISE tools. A DDR SDRAM memory is instantiated to host the Linux Kernel. The Ethernet MACs constitute the connection with the PHY layer and consequently with the network; two MACs have been instantiated to provide the access to the two redundant AFDX channels. A local BRAM memory and a FIFO have been also added to the design as generic interface with the rest of the NCAP. It can be noticed that no AFDX peculiar feature is implemented in hardware. Figure 4.2 shows a general overview of the described system: two separate Advanced eXtensible Interfaces (Xilinx adopted the AXI interface beginning with Spartan-6 and Virtex-6 devices) are used in the design, the first one operates at 100MHz and it is used for memory management and access, while the second is required for communication with the peripherals and operates with a 50MHz clock frequency.

4.3.2 Software implementation

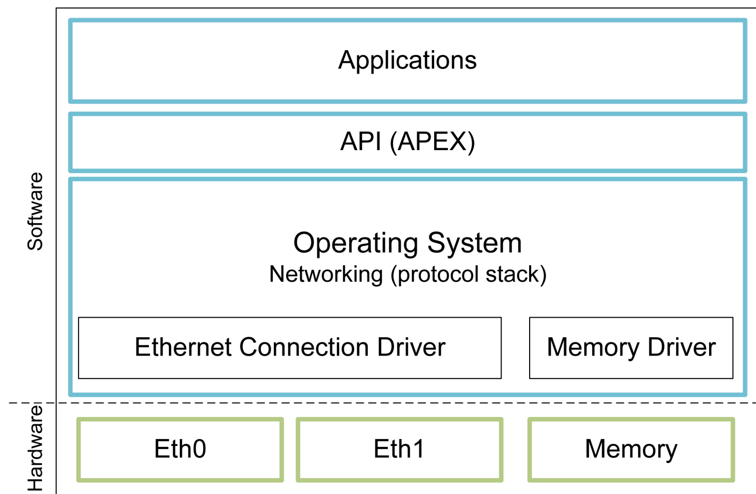

Figure 4.3 SW architecture overview

To improve portability and design reuse, a standard and widely used operative system (Linux) has been preferred to other embedded operative systems, furthermore this kernel is completely open source, allowing direct modification of the provided protocol stack. Figure 4.3 presents the overview of the software system that runs on the embedded processor, where the OS is responsible not only of the management of the networking features but it will also allow the concurrent execution of multiple applications that will implement NCAP services and gateway. The Application Programming Interface (API) is a standardized interface between these applications and the kernel and it constitutes another advantage of this design in terms of portability since it makes the applications independent from the rest of the system. The API is called APEX when the OS is compliant with the ARINC 653 standard, that concerns operating systems for safety critical applications and requires a custom interface with them to handle temporal and spatial segregation of tasks. The need for an ARINC 653 compliant OS made the development of the ES more complex, as described later. The memory shown in Figure 4.3 constitute a generic interface with the other sub-systems composing the NCAP, and frames that need to be exchanged with the ARINC 825 network can be stored in it waiting for forwarding.

A vanilla version of the Linux Kernel has been chosen as operating system, in particular the 2.6.37 version provided by Xilinx has been used. This Xilinx distribution of the Linux kernel is completely identical to the vanilla version, and it only adds drivers for all the Xilinx IPs and configuration files to support Xilinx's development boards. This kernel already includes the Ethernet protocol stack, which is mature and reliable thanks to the testing

and development that it experienced over the years. It includes all the TCP/UDP, IP and Ethernet features used everyday, and that will constitute the basis on top of which AFDX will be implemented. Ethernet and AFDX protocol stacks are compared in Figure 4.1, that highlights how most of the code provided with the Linux kernel can be reused as it is, without any modification. Although they are both supported by AFDX, UDP has been preferred to TCP for development and testing, since generally suggested: TCP introduces larger delays and it is not useful since AFDX already guarantees packet delivery.

To test if the planned modifications were actually applicable to the original Linux networking protocol stack, the Ethernet IPv4 protocol has been initially replaced with the modified AFDX version, instead of creating a separate path for AFDX sockets. This solution simplified coding and saved development time, but it is not suitable for a commercial product. Even if the standard Linux can be effectively used to explore possible fundamental implementations, the obtained results can only be a generic reference because of its non-real-time nature [24]. To obtain more significant results a Real-Time patch (RTLinux) has been applied to the kernel; this patch cannot give strict hard-real-time performance but improves the soft-real-time behaviour of the 2.6.37 Linux kernel. The frequency of the `jiffies` counter has been set to 1ms, instead of the standard 4ms. Of all the planned modifications, described in the previous section, only a basic version of the BAG control has been implemented to control the transmission time of packets on a single VL, because some problems occurred and suggested a change in the prototype design before the BAG control could be completed and before redundancy, integrity and VL management, and scheduling could be addressed. These problems are described in Section 4.4. The pseudo-code of the algorithm produced for the control of the transmission time is given in algorithm 4.3.2. This algorithm has been implemented in two different ways: in the Ethernet MAC driver, and in the *eth.c* code that realize the Ethernet layer of the protocol stack.

When the system and the Ethernet devices are initialized, the time of the "last frame sent" for each Virtual Link is set to "now" using the `jiffies` timer provided by the Linux kernel. Whenever a socket structure is received by the Ethernet layer and it is ready to be passed to the scheduler for transmission, its MAC destination address is analyzed to determine its VL, and a control over the elapsed time since the last transmission on that link is performed to determine if a time equal or longer than its BAG has passed. If the condition is respected, the packet can be sent, otherwise the function returns without doing anything, and the same control on the same socket structure is done as soon as the operating system schedules a call to the `dev_queue_xmit` function.

To test the precision on the transmission timing obtainable with this kind of implementation the modified kernel has been compiled and downloaded to the embedded processor,

Algorithm 1 BAG control

```
if initialization then
  vl_1.last_sent = now;
else
  if skb.mac_destination = vl_1.mac_address then
    if vl_1.last_sent − now < vl_1.BAG then
      exit;
    else
      send;
      vl_1.last_sent = now;
    end if
  end if
end if
```

and a simple application has been created and run on the same processor to send a series of dummy packages to a predetermined multicast destination. The kernel has been configured to only support the minimal functionalities required for this situation, and to only include the significant drivers for the IPs included in the hardware system; the size of this minimal kernel configuration is smaller than 4MB. The transmissions have been monitored using a snooping software on the computer receiving the frames. Since redundancy management is not present, the traffic is sent on a single channel. It has been observed that the arrival time of the packets followed the rate imposed by the defined BAG with an average deviation from the expected value of about $20\mu s$. All the possible BAG values have been set and tested to determine the relationship between their variation and the measured deviation, and the observed jitter shows the same characteristics in every simulation, suggesting that the BAG value does not influence it. The same behaviour and results have been observed implementing the same algorithm both in the device driver code, thus after the scheduler, and in the Ethernet layer, confirming that both solutions could be effectively used obtaining the same performance. Due to the non hard-real-time behaviour of the OS occasional delays up to $40\mu s$ have been observed. This value does not include the inevitable jitter that the device scheduler will add when dealing with multiple VLs, and consequently the present implementation may not be able to handle high numbers of virtual link. Better precision could be attained using timers, one for each VL, generating soft interrupts periodically, forcing the processor to check if there is any packet ready for transmission for the corresponding VL. this solution though would radically decrease overall system performance because an excessive amount of interrupts would be generated even in absence of any socket waiting for transmission. More time would have been necessary to implement this solution, and to perform more detailed

tests on the current implementation.

4.4 Practical Problems and Lesson Learned

The previously presented design has not been fully implemented, unfortunately, because of practical problems encountered during its development. Some of the issues are a consequence of other constraints imposed by the AVIO 402 project, while others concern the difficulty of handling a system realized on multiple levels (hardware, drivers, operating system, and applications).

The first issue is due to some incompatibilities between the Ethernet PHY chip available on the ISMNET module for the communication with two IEEE 802.3 physical ports and the corresponding drivers included in the chosen Linux kernel. The DP83640TVV PHY by National Semiconductor not only is IEEE 802.3 compliant, but it is also compatible with the IEEE 1588 standard for real time industrial connectivity. Even though this feature seemed interesting and promising in the context of the research for synchronization of local network nodes between each-other (in order to further reduce latency and packet loss), it was a major drawback because of the lack of dedicated drivers for this kind of PHYs in the stable versions of the Linux kernel. An unstable patch for the 2.6.37 kernel is available to include the required drivers in the kernel, but it was not possible to successfully compile the patched kernel. An official driver is now included in the 3.0 version of the kernel released in July 2011; unfortunately, this update, and the corresponding Xilinx release, arrived when other problems had manifested and the decision to change approach for the prototype realization was already being discussed. The previously presented implementation and testing exploits the Ethernet port available on the SP605 board instead of those of the extension mezzanine.

A second problem is a consequence of the reliability requirements of an avionic system, and even though the prototype being developed is not supposed to satisfy all of them, it is still required to be ARINC 653 compliant because this standard is part of the research objectives of the AVIO 402 project. Even if a real-time patch has been applied to the kernel, in the avionic environment hard real-time performance is not enough, and spatial and temporal segregation of the tasks performed by the processor, and supervision and control of its behaviour are required as well. ARINC 653 standardizes how task segregation and supervision must be performed by the OS, and applications must be expressly designed following the requirements of this standard and its modified API, called APEX. In the literature review provided in Section 2.2.2, it has been observed that it is generally suggested to directly design applications considering the need for an ARINC 653 compliant operating system, instead of including it only in an advanced stage of the development. Considering this aspect, it has been decided

to directly include ARINC 653 in the design of the prototype, but this involved additional issues in the implementation of the embedded system.

The problem in developing an ARINC 653-compliant embedded systems is the need for highly expensive commercial products, such as VxWorks653, LynxOS-178 RTOS, or LynxOS-SE RTOS operating systems. Furthermore, most of the ARINC 653 compliant OSes do not support the Microblaze processor, since physical, hard IP core, processors are usually preferred to softcores in avionic commercial products. A cheaper solution has been found in the SIMA tool, that emulates an ARINC 653 environment on top of a standard Linux kernel, but unfortunately it is not possible to run it on an embedded processor. Because of these considerations and the technical difficulties encountered with the Ethernet PHYs' drivers the embedded solutions has been abandoned and a PC has been chosen for the development of the AFDX End System instead. The modified structure of the prototype is illustrated in figure 4.4: the ES and all the applications that was previously planned to be implemented on the embedded processor are now executed by the PC, and a PCIe connection is used to connect it to the FPGA where the previously presented CAN controller is instantiated.

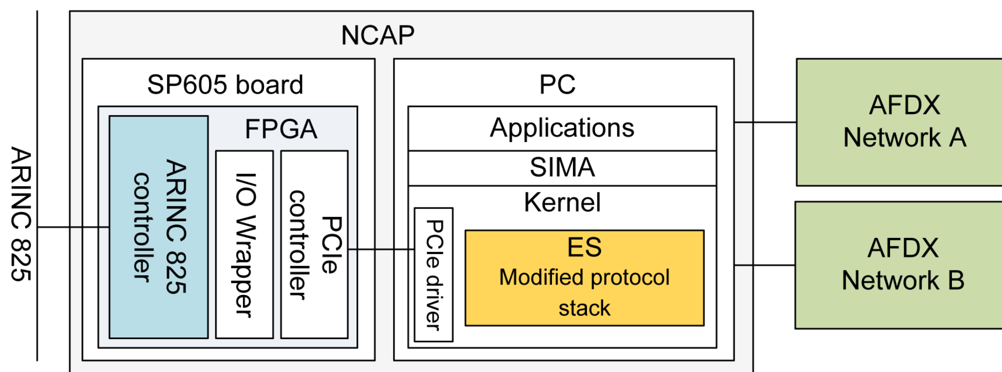

Figure 4.4 Modified prototype structure

The ES architecture proposed in this thesis proved itself portable and flexible, thanks to the possibility to reconfigure the same kernel for the Intel x86 processor available in the PC. In reality, a vanilla kernel has been used instead of the one provided by Xilinx, but the identical structure of the networking protocols allowed an easy port of the planned and completed modifications to this new kernel. The work on this system is being continued by other students that are implementing the entire End System on the PC, and interface it with the rest of the NCAP, that still resides in the FPGA, through a serial connection. This serial connection has been realized using a PCIe bus. Presently, the BAG control has been improved and the VL management integrated in the protocol stack, but redundancy and integrity mechanisms and custom scheduling are still under development.

It will be interesting, once the development is completed, to port it back to the embedded system, to evaluate how the lower performance of the embedded processor affects the resulting jitter, and how the number of manageable VL changes.

CHAPITRE 5

AFDX SWITCH

This chapter focuses on the description of the design and implementation of an AFDX switch fabric. This module has been conceived to be used as intellectual property in the development of the switch that will be included in the network prototype. Section 5.1 defines the features and functionalities that this system must provide, and the constraints it must respect, the design and considerations about it are explained in Section 5.2, before turning to the synthesis results obtained and their analysis given in Section 5.3. In Section 5.4, a description of the most important tests performed on this subsystem to validate its functionality is given, together with the resulting performance measurement.

5.1 Specification and requirements

To better understand the implementation choices made for the realization of the routing core of the AFDX switch, it is important to fully understand its role in the network and also under the Avio 402 project point of view. In this section, an overview of the most relevant features of the AFDX Switch is given, while more detailed information can be found in Chapter 4 of the official documentation [10]; its implications, together with the requirements of the AVIO 402 project, are analysed to introduce the design choices explained in the next section.

5.1.1 ADFX switch specification

While in chapter 2, a general description of the protocol has been provided, an analysis of the additional functionalities of an AFDX Switch, compared to those of a standard Ethernet router, still need to be performed to fully understand how the architectures presented in that same chapter can be applied to an avionic environment.

An AFDX network consists of up to 24 end systems connected to a switch and switches can be cascaded to increase the capacity of the network that has, consequently, a star topology identical to a switched Ethernet network. It is important to remember that the redundancy that characterizes this avionic protocol concerns only the End Systems since the two redundant networks are completely parallel and independent, invisible one to the other, therefore switches are not influenced by this aspect. The main role of the switches in the network is to redirect the incoming frames towards the corresponding destination selecting a predetermined path, specified in a routing table. While these routing functionalities are the same as

those provided by any Ethernet switch, some additional traffic control features are specified by the AFDX protocol, to perform error detection and segregation, thus improving network reliability. The specification identifies four main functional blocks in which the system can be separated, that are shown in Figure 5.1.

The four subsystems are the following:

1. Switching Function: it is responsible for the routing of the incoming frames towards the corresponding output ports. It uses the information included in the header of each frame to determine its Virtual Link and consequently its destination. It must also provide filtering capabilities to remove erroneous frames from the network. In an Ethernet network, it is often called Switch Fabric, and it is the module developed in this thesis.

2. Configuration Tables: they include all the information used by the switching function module to choose the output ports where the frame must be sent. They also contain all the Virtual Link parameters, such as related BAG and Lmax, necessary for the detection of erroneous frames.

3. Monitoring Function: every operation executed by the switch is monitored and identified errors are recorded to keep track of the switch operating state and determine if it is able to complete its task or if it must be removed from the network.

4. End System: it is the port that puts the switch in communication with the rest of the network and with the configuration tools. It can be used to configure the system, to load data in the configuration tables, and to exchange information about the switch state with the rest of the network.

As mentioned before, only the switching functions are analysed and implemented in this work, but the module has been designed to be easily and efficiently interfaced with the rest of the system that is going to be implemented on the FPGA. The separation of the various functionalities of the switch provided in the specification allows independent development of each sub-system, but it is necessary to determine how these modules are going to interact to provide the necessary interface signals.

5.1.2 Switch Fabric

The switching core of the router is highly similar to the IEEE 802.3 counterpart, since its main task is exactly the same. The destination address of each incoming frame must be identified and analysed to determine to which output ports it must be forwarded to, in order

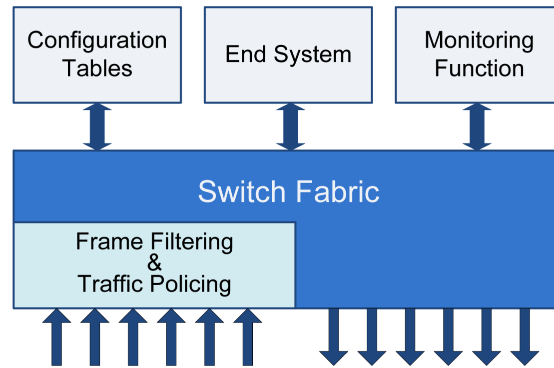

Figure 5.1 Representation of the modules of the Switch taken from the specification

for it to reach the required destinations. The frames will most probably have to be forwarded to more than one output since Virtual Links are multicast communications with generally more than one destination. As clearly shown in Figure 5.1, routing is not the only task that this module must execute: traffic policing and filtering are required as well at the reception of each transmission to identify and discard every erroneous frame. In the following sections, routing and filtering mechanisms are described.

Routing frames

This is the same exact function that any Ethernet switch must provide: while a frame is being received, its header is identified and analysed to determine to which VL it is related. The frame field that contains the relevant information is the destination MAC address, whose last 16 bits represents the virtual link. Information about the VL destination ports as well as VL parameters must then be retrieved from the configuration table where it is been previously recorded, and it is used for routing and filtering. Since VLs are usually multicast connections, the switch fabric must be able to route frames concurrently towards multiple output ports. The Part 7 of the ARINC 664 standard [10] specifies that, if the FIFO corresponding to one of the outputs where the packet is being forwarded is full, the frame must be dropped and it must not wait for the FIFO to be available; this avoids potential stall conditions that could occur if the output FIFO is dysfunctional. If there is output contention because multiple frames with the same destination are received concurrently, a scheduling algorithm must determine which packet will be forwarded first; thus buffers are necessary to store the packets that lose the contention.

Frame filtering

To ensure that only non-corrupted frames are being forwarded into the network, thus avoiding error propagation, upon arrival, corrupted frames are detected and discarded. To perform this operation, the core must test each frame's Frame Check Sequence (FCS) field according to the IEEE Std 802.3, and verify that the frame size is an integral number of octets greater than 64 and lower than 1518 bytes. There are also some VL related constraints that must be satisfied: the switch should in fact discard incoming frames which total Ethernet line size is greater than the maximum size (S_{max}) or smaller than the minimum size (S_{min}) allowed for the corresponding VL. Finally, it must also discard frames with an erroneous constant field in the MAC addresses, when the destination is not reachable, or when its VL is not allowed on the incoming port.

Traffic policing

Valid frames are then filtered for bandwidth: any frame that exceeds the bandwidth defined for its VL is discarded. The standard specifies a token-bucket algorithm for policing bandwidth and it leaves the possibility to choose between a frame-based and byte-based policing. This algorithm requires that an Account (AC_i) is created for each VL_i, and initialized to $S_i^{max} \times \left(1 + \frac{J_{i,switch}}{BAG_i}\right)$. AC_i is credited as time elapsed proportionally to $\frac{S_i^{max}}{BAG_i}$, with an upper limit equal to the initial value; whenever a valid frame is received on the virtual link i, the corresponding AC_i is debited by S_i^{max}. When a frame is received, if its AC_i is greater than S_i^{max} it is considered valid, otherwise the corresponding AC_i is not modified and the frame dropped. For the byte-based version of this algorithm, a frame is valid if its account is greater than S_i, and in that case the account is debited by S, while everything else remains the same. An example is given in Figure 5.2: the first frame determines where the BAG for its VL should start, and whenever a frame is received on that virtual link outside of the corresponding arrival window, the account is too small and the frame is consequently ignored.

Latency control

To limit the maximum end-to-end delay of the network, a maximum delay is defined for each port of the switch fabric: once a frame's latency in the core is too high, it must be considered too old to be useful and it must consequently be discarded. The specification divides the latency introduced by the switch into three parts: technological latency of the switching function, the configuration latency due to switch loading, and the time required to transmit the frame on the medium. Only an upper bound for the technological latency is specified and fixed at 100μs. No suggestion on a reasonable value for the maximum latency

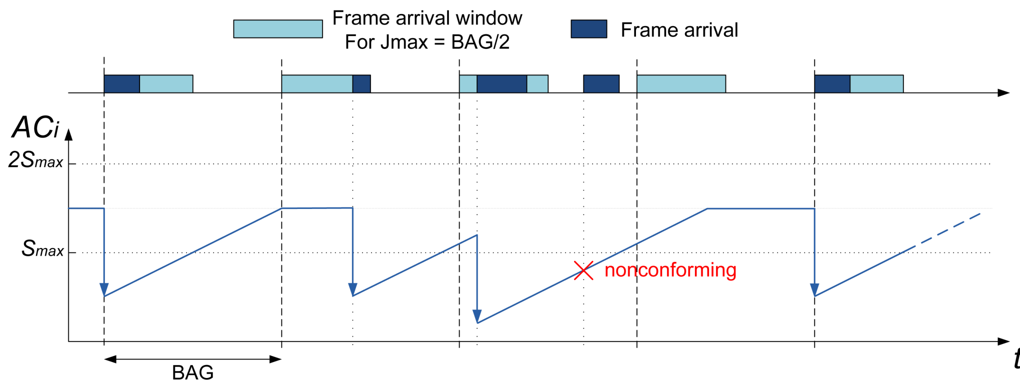

Figure 5.2 Example of frame-based leaky bucket algorithm application

is provided in [10].

5.1.3 AVIO 402 requirements

Like the other modules discussed in this thesis, the switch fabric is also intended to be used as intellectual property for the development of the network prototype on the same FPGA used in the rest of the project. Even if no precise constraint is imposed on its internal implementation, its structure must be suitable for a potential integration of the scheduling algorithms proposed by other students of the AVIO 402 project [32]. Design reuse must be considered also for this core. Therefore, the developed design must be portable to other devices and the conceived solutions must provide a general improvement of the technology readiness level.

The presence of the other functional modules specified by the ARINC 664 specification [10] in the final system must be considered, but to develop this core independently from them, configuration and routing tables are ignored in this thesis work, and the corresponding information included in internal registers of the switching fabric. To provide multiple Ethernet ports, the switching core will most probably be connected to the same PC used for the End System via PCIe, therefore the core interfaces must support this type of solution.

5.2 Core Design

In Section 2.2.3, some solutions developed for Ethernet switches have been presented and their advantages discussed. Work has been done to identify which existing architectures could be the more suitable for the AFDX network, considering the great importance of reliability and latency reduction in avionic networks over speed and throughput, which led the development of routers used in Gigabit Ethernet networks. Space-division switching

architectures seem better than time-division switching ones for an FPGA implementation, because of the lower clock frequencies it requires; time-division architectures need to work with frequencies N times higher than the wire speed (where N is the number of ports), and hundreds of MHz are generally not achievable on FPGAs. Space-division switching also allows easier implementation of parallel filtering of incoming frames.

In addition to the advantages of parallel processing of incoming data, a space-division switching architecture has been chosen for the switch fabric also because a simple implementation of a router of this type was already available. A behavioural description of an Ethernet switch fabric with this structure has been in fact created by other students of the research group. Starting from this design, the AFDX specific features previously presented have been added to adapt it to the ARINC 664 specification, and a VHDL implementation has been completed. The architecture of this switch fabric is, as anticipated, based on a parallel treatment of the incoming frames, and on a combined input/output queuing system, as shown in Figure 5.3. Thanks to the analogies between Ethernet and AFDX, the migration towards the avionic protocol is possible without any significant modification on the general structure of the core, but only with some adjustments in the internal functionalities and features of the functional modules.

5.2.1 Hardware advantages

The switch fabric processing time is translated into latency added to each communication that passes through this system. When multiple switches realize the path of a certain VL, the delay introduced by each one of them sums up and contributes to the resulting end-to-end latency of the communication. It is obvious that the maximum latency of frames inside this system is a key parameter that needs to be minimized. A full hardware implementation, with a parallel processing of incoming frames is clearly the most suitable solution to address this issue, especially in a system like the switch, where multiple packets are received in parallel at different inputs. Parallel processing must be maximized to increase the system throughput by reducing the processing time of the modules shared by all the parallel paths. This structure will also be more robust to certain kinds of faults that can occur to one single part of the core and remain invisible to the rest of the system, which can consequently continue working correctly.

5.2.2 Switch Architecture

The original architecture was conceived for an Ethernet Switch, thus it did not considered the requirements of the AFDX protocol, and it consequently needed some adjustments to add

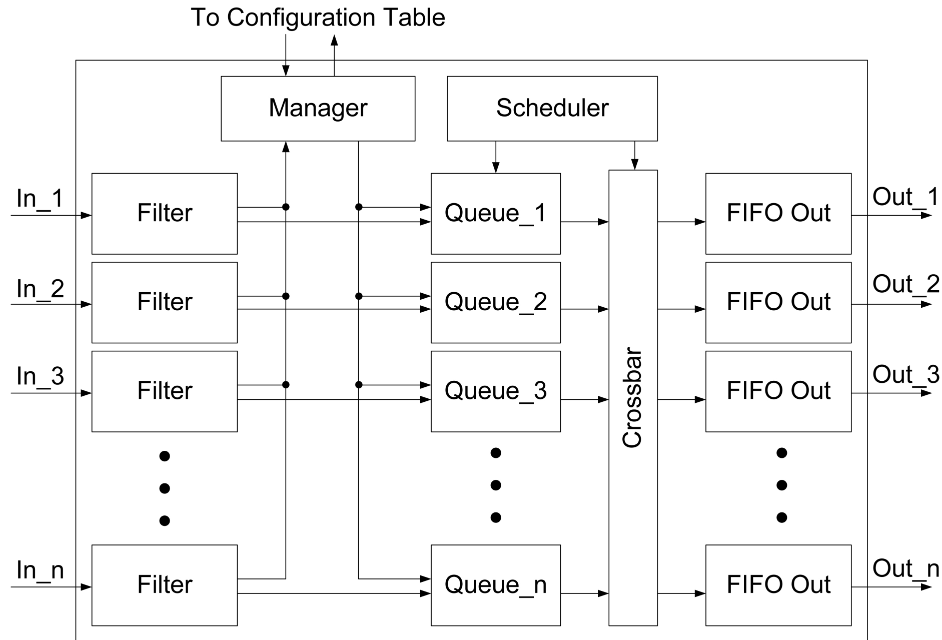

Figure 5.3 Architecture of the Switch

the filtering features described above. Figure 5.3 shows the functional modules that compose the routing core of the switch, whose general structure has not been changed. Although the overall architecture is the same as in the original Ethernet version of the design, important changes were made to the internal behaviour of some of the internal blocks, such as the input filtering module and the Manager: while the former originally only had to save the incoming packet in the input queue and to pass its header to the interpreter, which determined its destination, now, they both have filtering responsibilities. Filtering and traffic policing have been placed at the entrance of the system to eliminate erroneous frames as soon as possible from the switch, to prevent them to use important processing resources that can consequently be dedicated only to good frames. Some system signals have been added as well because of the added filtering features.

The full hardware solution allows the minimization of the latency of the routed frames in the core. The parallel processing of the incoming ones as well as the outgoing ones can keep the technological latency as low as possible especially when multiple frames are received concurrently. The only bottleneck is the presence of only one configuration table, and only one manger that can access it. While the first filtering stage can be done in a parallel fashion in the input blocks, these modules must wait for the manager to be available to complete the filtering. Because of these resources shared between all the inputs, the amount of processing executed by the manager has to be minimized to reduce the time spent processing each frame.

The number of filter blocks, of queues, and of FIFOs depends on the number of input and output ports included in the design, and it is set by a single parameter `PORT_NUMBER`. This value is obviously the same for the inputs and the outputs since each communication port is bidirectional. In the following section, the behaviour of each module is described while presenting the reception and transmission processes.

Frame reception

When a frame is received on any input port, it must be immediately stored in the input *Queue* while its destination address is analysed by the *Manager* to determine to which output ports it must be forwarded. The filtering functionalities required by the AFDX specification must be integrated in this process as well. The presence of filtering features required both a modification of the internal functionalities of the input modules and of the communications between them: the *Manager* must now ensure that each VL respects its allocated bandwidth, and send VL parameters to the *Filter* block to let it complete the frame filtering. The modifications in the communication between these modules are shown in Figure 5.4 using blue arrows. In addition to each frame destinations, also its priority is sent to the *Queue* that stores it, in order to separate high priority from low priority frames. The *Queue* must provide a mechanism to discard erroneous packets. A detailed description of each module's behaviour is given in the following paragraphs.

Filter This module is the entry point of the system, and while in an Ethernet Switch it only has a secondary role, it becomes vital when considering AFDX's reliability issues. Its main task is to read incoming frames, 8 bits at a time, from the buffer of the MAC core that received them, and to store them in the input *Queues*, while extracting their header that must be passed to the *Manager*. It also provides frame filtering in order to detect and discard erroneous frames; unfortunately, most of the required controls depend on VL parameters that are provided by the *Manager*, that consequently needs to be consulted. The controls performed by the *Filter* module are the following:

- The Frame Check Sequence received corresponds to the one computed by the filter
- Constant fields have the structure expected for an AFDX header
- The Ethernet frame size is an integral number of octets (alignment)
- The Ethernet frame size is within the range 64 to 1518 octets
- The Ethernet frame size is lower or equal to $Smax$ and greater or equal to $Smin$

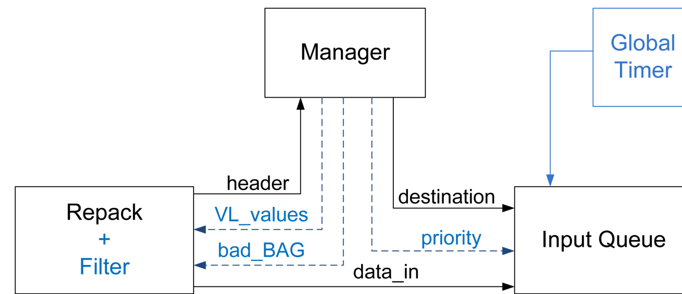

Figure 5.4 Reception modules

Evidently, information about the the upper and lower bound of the frame size (Smax and Smin) are VL dependent, and consequently the *Manager*'s intervention is unavoidable for this verification; therefore, the third point, that is otherwise redundant, is performed as well to drop the packets not conforming to those bounds before processing time of the *Manager* is wasted. Whenever an error is detected by one of the VL-independent verifications before the header is passed to the Manager, the frame is immediately discarded.

Received bytes are saved in the *Queue* without waiting for the complete reception of the frame to avoid the introduction of an additional delay to the communication. The fact that most of the time an erroneous frame is detected only at the end of its reception, once the FCS is computed and the frame size is known, makes it necessary to implement a mechanism to drop them once they are already stored in the queue. This process is entirely handled by the *Queue* itself and the *Filter* must simply send a `drop_frame` signal to it whenever it is necessary. To improve the system throughput, received bytes are merged into 16-bits words to create a larger datapath.

Manager This module has access to the Configuration Table and to the Routing Table saved in an external RAM memory, and, consequently, it has access to all the information the system needs to complete filtering as well as to determine the path each frame must follow to reach the correct destinations. As aforementioned, it is responsible for providing the *Queue* and the *Filter* information about the size, Smax, Smin, and priority of the frame they are treating, but it must also execute traffic policing, a feature peculiar of AFDX, and not expected in any Ethernet switch.

Since the priority of incoming frames is unknown before the identification of their VL, if multiple *Filters* ask for the attention of the *Manager* at the same time, a round robin scheduling approach is used. Once the header is received, the destination MAC address is extracted to determine its VL and this is used to retrieve the corresponding information from the configuration tables. If the VL is not valid, the *Manager* tells the *Filter* to drop it.

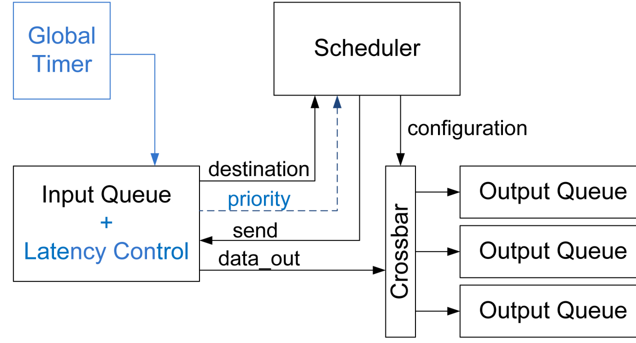

Figure 5.5 Transmission modules

Another important feature this module must provide is traffic policing. Frames that does not respect the Bandwidth Allocation Gap that corresponds to their VL must be eliminated from the network. Between the two possible algorithms proposed by the ARINC 664 specification (frame-based and byte-based leaky bucket algorithms), the frame-based one has been preferred because a study by Yao *et al.* [33] highlighted some possible weakness of the byte-based solution. To implement the algorithm presented in Section 5.1.2, an account has been created for each VL, and it has been initially set to $S_i^{max} \times N_i \times \left(1 + \frac{J_{i,switch}}{BAG_i}\right)$. It is debited by $S_i^{max} \times N_i$ each time a valid frame is received, and credited by S_i^{max} each clock cycle. S_i^{max} is the maximum frame size for the VL_i, $J_{i,switch}$ is its maximum allowed jitter and it can be expressed as fraction of its BAG_i, and, finally, N_i is the number of clock cycles that the BAG_i lasts. Since a moderate resolution is requested by the specification (better than 100μs) a clock with a lower frequency than the system one has been chosen to keep registers' and reduce power consumption. A frame is considered valid if, when it is received, its account is higher than $S_i^{max} \times N_i$, and it is dropped otherwise by using the `bad_BAG` signal.

Listing 5.1 Overview of the Manager VHDL code

```
-- PROCESS -------------------------------------------------------------
-- MainManager: it's the process that handles the flow of the FSM that controls this module,
-- it also set the values of the signals used to communicate with the Filter and with the Queue
------------------------------------------------------------------------
MainManager: process(clk, reset)
variable cpt_round_robin     : integer := 0;
variable current_VL          : integer := 0;
variable index               : integer := 0;
variable last_port           : integer := 0;
begin
    if(reset = '1') then
        -- reset all signals --
        ...
        manager_state <= INIT_STATE;

    elsif ( clk'event and clk = '1') then
```

```
    case manager_state is
    when INIT_STATE =>
        -- reset all signals --
        ...
        manager_state <= WAIT_STATE;

    when WAIT_STATE =>
        ...
-- If a Filter has a header ready the routine can start
        if( header_valid /= vector_of_zeros ) then
            manager_state <= SEND_STATE;
        end if;

    when SEND_STATE =>
        -- starting from the last_port + 1 (to execute a Round Robin)
        -- the Manager determines which Filter asked its intervention
        -- index = port to be treated
        ...

        else
-- extract VL
            current_VL := conv_integer(header(index)( 79 downto 64 ) );
            VL_received <= current_VL;
-- VL is valid if in the acceptable range (VL must be mapped to a consecutive
-- list of number going from 0 to Nbr_VLs
            if((current_VL < Nbr_VLs) and (current_VL >= 0)) then

-- TRAFFIC POLICING: if the ACcount for the current VL is lower than
-- the corresponding ACmin the frame is not valid
                if(AC(current_VL) < AC_min(current_VL))then
                    bad_BAG_jitter(index) <= '1';
                    frame_received <= false;
                else
                    bad_BAG_jitter(index) <= '0';
                    frame_received <= true;
                end if;

-- The outputs towards the input port that sent the header are updated
                priority(index)     <= priority_table( current_VL );
                Smin(index) <= array_Smin( current_VL );
                Smax(index) <= array_Smax( current_VL );
                destination(index)  <= array_Ports( current_VL );
                Manager_out_valid(index)  <= '1';
            else
-- The VL is not acceptable and the frame must be dropped
                bad_BAG_jitter(index) <= '1';
            end if;
-- Round Robin: last port processed is saved
            last_port := index +1;
        end if;
        manager_state <= WAIT_STATE;
    when others =>
        manager_state <= INIT_STATE;
        end case;
end if;
```

```
end process;

--- PROCESS ------------------------------------------------------------------
-- TrafficPolicing: it handles the ACcounts for each VL, incrementing them constantly
-- and decreasing them when a frame from the corresponding Vl is received
------------------------------------------------------------------------------
TrafficPolicing: process(clk, reset)
begin
    if(reset = '1')then
    -- All the ACcounts are set to their maximum value = Smax * N * (1 + J/BAG)
        ...

    elsif(clk'event AND clk = '1')then
        case manager_state is
        when INIT_STATE =>
    -- All the ACcounts are set to their maximum value = Smax * N * (1 + J/BAG)
    -- Like if a reset occurred
            ...

-- OTHERS: in any other case they are incremented continuously, or decremented if a
-- valid frame is received on that VL
        when others =>
    -- If a frame is received, the corresponding AC is decremented by the relative AC_min
    -- AC > AC_min is not verified because that's already done in the main process
            if(frame_received)then  -- the main process detected a valid frame
                AC(VL_received) <= AC(VL_received) - AC_min(VL_received);
            end if;
            if(clk_low = '1')then
                for j in 0 to Nbr_VLs - 1 loop
                    if(AC(j) < AC_max(j))then
                        AC(j) <= AC(j) + delta_AC(j);
                    end if;
                end loop;
            end if;
            end case;
    end if;
end process;
```

The significant part of the *Manager* code is reported in Listing 5.1, this code is a shortened version of the one developed for the core implementation, which can be found in Appendix B.2. Two different processes are used to control the *Manager*'s execution flow and the accounts management for the traffic policing. A 1MHz clock is used to increment the ACs in order to use smaller registers to save the various ACs, saving memory while providing the required precision (better than 100μs.)

Frame scheduling and transmission

Once a frame is saved in the input *Queue*, it needs to be forwarded to the corresponding output ports, and the *Scheduler* is responsible for most of the related operations. Scheduling becomes crucial when multiple *Queues* contain frames waiting for transmission and output

contention occurs, because the algorithm chosen for the scheduling will determine frame latency in the system. All the high priority frames must be forwarded before considering low priority ones.

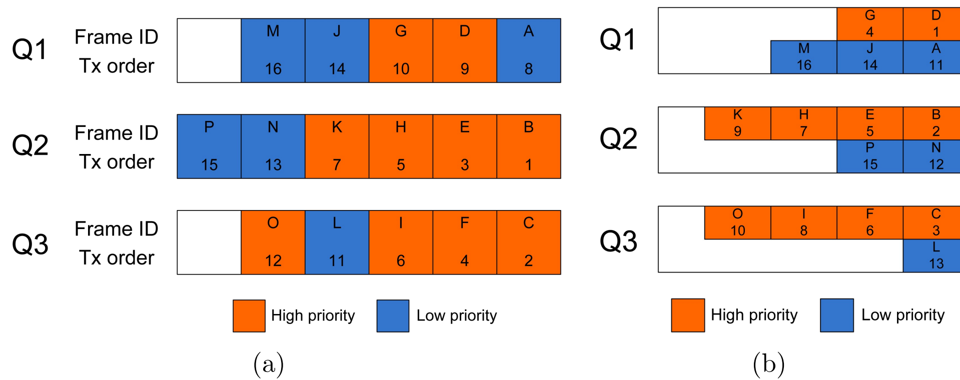

Figure 5.6 Head of Line Blocking: (a) Single buffered; (b) Double buffered. The numbers in the "packets" correspond to their transmission order.

Queue During reception, the *Queue* saves the packet in its internal RAM, saving the start and the end address that delimit the memory region occupied by it. If the drop flag is set in any moment of the reception the frame is immediately discarded. Since there are two possible priorities but they are not known at the beginning of the reception, the packet is initially saved in both of two separate FIFOs, one for high priority and the other for low priority frames, and it is dropped from the wrong one when information on the priority is received. If no information is received before the end of the reception the frame, this is removed from both queues because the system must be ready to receive a new packet.

The *Queue* must tell to the Scheduler, whenever asked, whether it contains a frame ready to be forwarded or not. If frames are present, the high priority FIFO must be emptied before low priority data can be considered. The destination and priority of the first packet to be forwarded must be provided to the *Scheduler* as a response. When the `send` signal is received a transmission must be initialized.

Maximum latency control is implemented in this module as well, since frames can remain in its FIFO indefinitely, depending on the *Scheduler* behaviour. When the last byte of a frame is received, the value of the global timer is saved as `reception_time`; when this frame is the next scheduled for transmission the difference between the global timer and the `reception_time` is computed and, if this difference becomes higher than the *Max_delay* for that port, the frame is discarded. This global timer has been added to the architecture presented before to implement this control over frame latency; it is simply a counter whose

value is provided to all the *Queue* modules of the Switch. Using the same counter, a control over the jitter of each frame can be implemented. This parameter is defined on a per-VL basis, and not on a per-port basis as the latency, and should not be higher than 10ms; it is not included in the design at the moment since it is not expressly required by the specification.

The biggest problem of input queuing, as mentioned in Section 2.2.3, is the Head of Line (HOL) blocking, that can add significant delay to frames waiting for forwarding in the *Queue*, when other *Queues* are sending packets to the same output port. This phenomenon, shown in Figure 5.6, becomes more critical when frames with different priorities are stored in the same FIFO: a low priority frame must wait that all the high priority frames are scheduled before it can be transmitted, consequently increasing the latency of all the following frames, that could be critical. In this figure, the numbers in each frame represent the scheduling order, while their ID gives an idea of the reception sequence. It can be observed that the first high priority frame in Q1 must wait 9 "rounds" when it should have been forwarded before the low priority frame in the same queue. Separating high from low priority frames, it can be noticed how the latency on the packet A has been reduced to only 1 "round". Virtual queuing becomes necessary in this case, but since only two priority levels are possible, it has been chosen to exploit two separate FIFOs instead of having virtual queues in the same physical memory; this solution not only creates a physical segregation of the two types of traffic, but it also provides a form of redundancy that can be exploited to increase the core reliability. Whenever one of the two FIFOs of one *Queue* is dysfunctional, the corresponding traffic is redirected towards the other FIFO, avoiding the loss of all the packet that should have been saved in it. In the current implementation, this mechanism is applied only if the high priority FIFO stops working, and it has been preferred to lose low priority frames instead of increasing the latency of critical ones.

Scheduler This module manages the transmission of frames waiting in the input *Queues* to the output *FIFOs* depending on their destinations. The algorithm executed by the *Scheduler* starts by checking the output *FIFOs* and the *Queues*, to determine whether there are frames ready and whether there are available outputs. The rest of the algorithm is not executed until there is at least one available output FIFO. The destination specified by the *Queue* includes information on the frame priority as well. Starting with the high priority frames, the destinations of the packet in each queue are analysed to check if the corresponding FIFOs are available and, whenever this condition is met, the control signal `send` for the corresponding *Queue* is set high, and the output *FIFOs* that will be used are set to "unavailable". The next *Queue* is then considered. After the first round, where only high priority frames are processed, a second one is performed to forward low priority frames to the output ports which

are still available. The `last` variable saves the *Queue* where the "Round" started, so that it will start from the following one next time the algorithm is performed. In addition to the `send` signal sent to each *Queue* to start a transmission, a configuration array is also set up depending on where each frame must be redirected, to properly configure the *Crossbar*.

The complete VHDL implementation of the *Scheduler* is provided in Appendix B.3, but an overview of the developed module and of the algorithm is also given here.

Listing 5.2 Overview of the Scheduler VHDL code

```
--------------------------------------------------------------------------------
-- SCHDULING PROCESS -----------------------------------------------------------
--------------------------------------------------------------------------------
Scheduling: process(reset,clk)
begin
    if (reset='1')then
    -- All the outputs and control signals are reset
       ...

    elsif (clk='1' and clk'event) then
    case state is
-- IDLE: It checks which outputs are available, and if the answer is positive, it
-- switches to REP_QUEUE to perform the scheduling
    when IDLE =>
        queue_send := (others => '0');
    -- if there is at least one available FIFO, the list of available outputs is saved
    -- and the request is sent to the queues. The scheduling routine is then started
        if(out_port_available /= vector_of_zeros) then
            v_FIFO_available := out_port_available; -- available FIFOs are saved
            request <= (others => '1'); -- request is sent to the Queues
            state <= REP_QUEUE;
        end if;
-- REP_QUEUE: It waits for the queues' response, then it saves them and finally
-- it starts the scheduling.
    when rep_queue  =>
        request<=  (others => '0');
    -- After one cycle the answer of the queue is read
     ... -- wait for 1 clk cycle
        response_reg <= response;
        state <= R_ROBIN_H;
-- R_ROBIN_H: the round robin algorithm is performed for to determine which queues can send
-- their packet because their destinations are available. Only high priority frames
-- are considered here
    when R_ROBIN_H    =>
    -- The queues destinations are analysed to determine which ones have high priority
        for j in 0 to Nbr_ports - 1 loop
    -- Round Robin: the polling starts from the queue immediatelly after the last one
            index := start_queue + j;
            ... -- index reset to 0 when higher than Nbr_ports - 1
    -- If the considered queue has a high priority frame and a non-zero destination
            if(response_reg(index)(Nbr_ports) =  '1' AND
            response_reg(index)(Nbr_ports-1 downto 0) /= null_vector)then
    -- send_possible is set back to false if the required outputs are not available
                send_possible := true;
```

```
                for i in 0 to Nbr_ports - 1 loop
                    -- If the required FIFO is not available the frame will not be sent
                    if(response_reg(index)(i) = '1' AND v_FIFO_available(i) = '0')then
                        send_possible := false;
                    end if;
                end loop;
    -- If the queue can start a tranmsission the "send" vector is updated to force this transmission,
    -- the crossbar configuration is updated, and the output ports that will be used
    -- are now considered unavailable
                if(send_possible)then
                    queue_send(index) := '1';
                    for i in 0 to Nbr_ports - 1 loop
                        if(response_reg(index)(i) = '1')then -- for each required destination
                            v_FIFO_available(i) := '0';
                            crossbar_config(i) := index;
                            be_ready(i) <= '1'; -- it tells to the corresponding FIFO that it is
                                                -- going to receive a packet
                        end if;
                    end loop;
                end if;
            end if;
        end loop;
        state <= R_ROBIN_L;
-- R_Robin_L: The remaining queues, the ones with low priority, are now considered
-- The same algorithm used for high priority queues is adopted here as well
    when R_ROBIN_L =>
        for j in 0 to Nbr_ports - 1 loop
            index := start_queue + j;
            ...
            if(response_reg(index)(Nbr_ports) =  '0' AND -- the frame must have L priority
            response_reg(index)(Nbr_ports-1 downto 0) /= null_vector)then
                ... -- determine if the "index" frame can be sent
                    -- in this case send_possible is true

                -- The outputs used by high priority queues are not available anymore
                -- low priority frames can use only the remaining outputs
                if(send_possible)then
                    queue_send(index) := '1'; -- force transmission
                    for i in 0 to Nbr_ports - 1 loop
                        if(response_reg(index)(i) = '1')then -- for each required destination
                            ... -- used destination is not available anymore, crossbar
                                -- config is updated, and output FIFO is advised that
                                -- a frame is going to be forwarded
                        end if;
                    end loop;
                end if;
            end if;
        end loop;
        state <= SEND_QUEUE_OK;
-- SEND_QUEUE_OK: the pointer to the current queue is incremented for the round robin
    when SEND_QUEUE_OK =>
    -- When at least one frame is forwarded the round robin is incremented, or set to 0 when
    -- it reaches the last input queue
        if(queue_send /= null_vector)then
            if(start_queue = Nbr_ports -1 )then
```

```
                start_queue <= 0;
            else
                start_queue <= start_queue + 1;
            end if;
        end if;
    -- The to_send output is updated with the result of the scheduling, causing the queues
    -- to start a transmission if the corrisponding bit of the array is '1'
    -- State switches back to IDLE, to start a new round in case other outputs are
    -- now available
        to_send_reg <= queue_send;
        state<=IDLE;
    when others =>
    -- This should not happen, the state is set back to IDLE
        state <= IDLE;
    end case;
    end if;
end process Scheduling;
```

Even if this algorithm is functional and respects the official specification, the scheduling algorithm could still be improved to further reduce frame latency: some results of the AVIO 402 project [32] identify a BAG-based algorithm as the best choice to reduce End-to-End delay for example. Another solution would be to forward small sized frames before large frames because they add a smaller latency on the latter than vice-versa.

Crossbar and Output FIFOs As mentioned above, the *Crossbar* is simply a programmable connection, configured by the manager, which brings the frames stored in the *Queues* to the corresponding output *FIFOs*. It is designed to allow concurrent transmission from any *Queue* to any available output, so that frames from different inputs can always be forwarded simultaneously. The *FIFOs* are the interface that the switch fabric offers to the rest of the prototype; they can consequently be connected to an Ethernet MAC core if the SP605 ports are used, or to the PCIe connection if the physical ports of the PC are used instead.

5.3 Synthesis results

This section describes how the system described in Section 5.2 was implemented and verified. Since the configuration and routing tables were emulated using internal registers in the *Manager*, it was possible to validate the functional behaviour of the system even if the other modules of the switch, presented in figure 5.1, were not ready. These modules (Configuration tables, End System, and Error controller) are necessary for the development of a complete switch, but they do not impact the behaviour of the switch fabric. To perform a verification of the system, the developed VHDL code has been simulated at logical level using ModelSim.

The hardware implementation of the core has been realized in VHDL, and it has been successively synthesized for a Xilinx XC6SLX45T using the ISE design suite version 12.4;

Table 5.1 System size for 10 and 20 ports

# ports	5 (5 in, 5 out)	10 (10 in, 10 out)
# slice registers	13166 (24%)	24691 (45%)
# slice LUTs	18684 (68%)	33607 (123%)

Table 5.2 Size of single modules

Module	# Slice registers	# Slice LUTs	# Instantiations
Repack	368 (<1%)	461 (1%)	N
Queue	1898 (3%)	2305 (8%)	N
FIFO	459 (<1%)	523 (1%)	N
Manager	107 (<1%)	1121 (4%)	1
Scheduler	416 (<1%)	2939 (10%)	1
Crossbar	0	245 (<1%)	1

the following results are consequently related to this platform and software. The system has been synthesized for the same device used for the End System, even if larger FPGAs could be more appropriate for this particular system, because in this way it will be possible to easily interface it with the PC using the same PCIe connection developed for the end System. All the code was developed from scratch, and no pre-compiled IP cores, from Xilinx or other vendors, have been used; therefore, this core should be portable to any other FPGA without needing modifications, but synthesis has been tried only for devices from Xilinx.

5.3.1 System size

The system size obviously depends on the number of input and output ports that are included in the switch since it determines the number of parallel paths that will be included in the core. Even if the *Manager* and the *Scheduler* occupy a considerable area, this is only a small fraction of the available resources, and the real limit is imposed by the resources taken by the *Queues* because they need to be instantiated multiple times. In Table 5.1, it can be noticed that doubling the number of ports, the occupied resources almost double as well, proving that the impact of the *Manager*, *Scheduler*, and *Crossbar* is small on the overall size, especially for high number of ports. The area of a single *Queue* significantly increased once the separation of high and low priority traffic has been integrated, because it is equivalent to doubling the number of *Queues*, and they are now the most critical module in terms of

occupied resources, as shown in Table 5.2.

If the network does not need to support the 100Mbit/s speed (supported by the specification, but the network could be configured to work just at 10Mbit/s making a 100Mbit/s capable switch overpowered), the datapath could be modified to reduce the system size while reducing its throughput as well.

The occupied resources by the entire core depends not only on the number of ports, but also on the size of the RAM blocks used for implementing the FIFOs included in the *Queues* and at the outputs. Their depth influences the size of the registers used to handle the read and write addresses, and the complexity of the logic connected to them. The results previously shown are referred to a core that exploits 4k RAMs wherever a FIFO must be implemented, in order to allow the storage of at least two frames of maximum size in each buffer.

5.3.2 Timing

Timing results are given in Table 5.3. They show that the number of ports affects the critical path, thus decreasing the maximum possible operating frequency; this is due to the additional work demanded to the *Scheduler* when more *Queues* are present. A frequency of 30MHz is easily achievable even for high number of I/O ports; considering the 16-bit datapath implemented in the system, a 480Mbit/s data rate per path in the switch fabric is achieved for this clock frequency. This result is more than sufficient to handle the 100Mbit/s transmission rate of the AFDX port without generating congestion. If a size reduction is desired, a 4-bit datapath cold be a valid tradeoff between occupied resources and throughput, it would in fact suffice to handle a 100Mbit/s network with its 120Mbit/s data rate. If the switch is designed to be used only in 10Mbit/s networks the core could be made even smaller adopting datapath down to 1-bit wide. It must be considered though that keeping a good margin between the network operating frequency and the switch throughput helps dealing with high-load situations.

Since the datapath can already give the required data transfer rate, an improvement of the system operating frequency could be important only if the processing performance proves itself a limitation for high input data rates. In this case, the scheduling algorithm implementation should be addressed since the longest path of the synthesized core is part of the *Scheduler* module. The maximum operating frequency is determined also by the adopted device, a Spartan-6 in this case, and better performances could be obtained using a different FPGA family, or an FPGA with a better speed grade.

The latency of the frames inside of the switch is influenced by the size of the packet and the number of frames that must be redirected towards the same output port at the same

Table 5.3 After synthesis operating frequence

# ports	5	10
Max frequency	52.814MHz	34.301MHz
min period	18.934ns	29.153ns

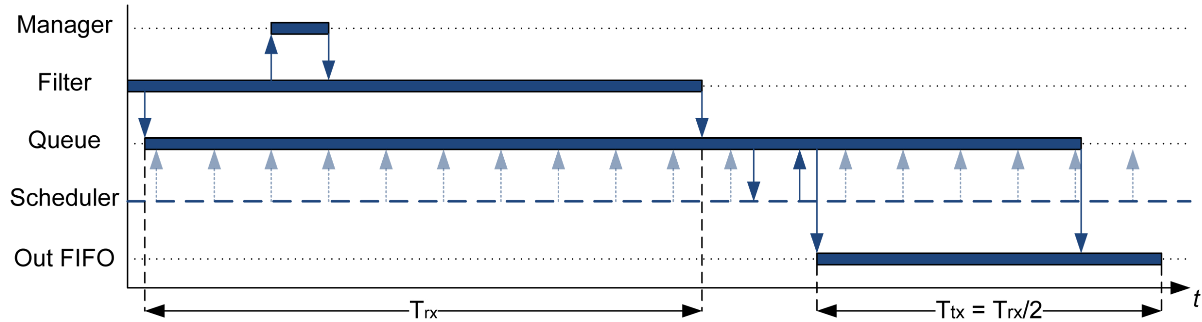

Figure 5.7 Frame treatment by the core functional modules

time; in fact, the longest time lost in the system is due to the transmission from the *Queue* to the *FIFO*. A frame with the maximum allowed size of 1542 octets needs around 20us to be written into the output FIFOs (16 bits datapath at 30MHz clock frequency); this time limits the maximum number of frames that can be simultaneously received and forwarded to the same output without exceeding the maximum latency constraint. The header processing (the other potential bottleneck because it is a shared resource between the parallel paths) is not critical since the computations that must be performed by the *Manager* have been minimized. The overall time lapse between the moment the header is passed to the *Manager* and the moment it provides the results is of only 4 clock cycles.

Figure 5.7 illustrates how the frame is processed and propagated in the core; the boxes represent the time each module is working on the considered frame. It can be noticed that the *Manager* is active only for a limited percentage of the time taken by the frame to be completely received by the *Filter* module, that is equal to $T_{rx} = N_{bytes} \times T_{clk}$, where N_{bytes} is the number of Bytes that compose the received packet, and T_{clk} is the clock period. The scheduler continuously check if data is present in the *Queue*, and when it receives a positive response, it forces the transmission of the packet to the output *FIFO*, which takes only $\frac{T_{rx}}{2}$ since the packet is transferred 16 bits at a time.

5.3.3 Considerations on the implementation

The size of the system, even if it could be reduced with some optimizations on the *Queue*'s code, is small: up to 8 input and 8 output ports can be instantiated on a small size FPGA such as the Xilinx XC6SLX45T used for the prototype. The interfaces are generic FIFOs and they consequently can be connected to both an Ethernet MAC core or the PCIe controller to be communicate with the PC. In both cases, a simple wrapper should be developed to adapt the basic FIFO interface with the MAC interface, or with the PCIe controller; in the second case, a simple protocol should be implemented to converge the packets coming from different ports in the single PCIe channel, and then distribute them towards the corresponding Eth port of the PC. Ethernet MACs usually have separated receive and transmit buffers, and consequently the *Filter* module should read from the first, while the second can directly replace the output *FIFO*.

The system performance satisfies the specification and the project requirements: the switch can handle 10 and 100Mbit/s communications on the network without creating congestion, therefore bringing to a minimum the number of frames lost because of buffer overload. The scheduling algorithm is located entirely in the *Scheduler*, and it can be easily modified to test other solutions without requiring any modification to the other parts of the core.

To complete the realization of the entire switch, it is necessary to implement the monitoring functions and the configuration tables required by the specification, and the End System discussed in Chapter 4 must be adapted to be interfaced with the switch fabric. The *Manager* will need some modifications when external tables will be available instead the internal registers currently used. Even if these modules are not implemented yet, it is possible to use the switch fabric alone for frame routing in this prototyping stage, since reconfigurability and monitoring are not required.

5.4 Test and validation

In order to validate the implemented system functionality, each module has been individually debugged and verified, and a series of possible scenarios has been successively used to analyse the overall system behaviour under standard and critical conditions. For a detailed description of the most interesting test cases and of the resulting information derived from each one of them, see Appendix C.1. For all the tests, the output FIFO has been configured so that it start transmitting the received frames as soon as possible, in order to study the system behaviour without the need for the Ethernet MAC to retrieve the packets from the output ports; this configuration allows the measurement of the latency of the frames in the core, independently from the system it will be connected to. In this section, an overview of

the results and the considerations deriving from these tests is given, in order to provide an analysis of the core performance and behaviour. In order to generate readable and comprehensible results, a 5-port switch fabric has been used in the simulations. Another reason for choosing this number of ports is that it is a reasonable size for the core that will be used in the prototype.

5.4.1 Testbenches

System verification has been performed using logical simulations, run with the ModelSim simulation tool, that highlight specific features of the system. All modules have been independently tested to verify that they provide all the required features, and to analyze their behaviour when input signals do not reproduce expected situations; on the other hand, such a detailed verification is not possible when they are integrated in the complete switch fabric due to the complexity of the system. It has been consequently decided to perform two types of test on the switch fabric:

1. a generic traffic, including valid and erroneous frames, has been generated and sent to the input ports, and the outputs of the core have been observed to determine if each packet reached the desired destinations;

2. some specific scenarios have been recreated to analyze not only the overall functionalities, but also the internal behaviour of the switch fabric, when critical and stressful situations occur.

While the first kind of test only validated the general routing and filtering features of the system, the latter is required to perform some specific performance measurements, and to observe how the system works in detail, to detect potential design weaknesses. The tests developed to realize these peculiar scenarios are listed in Table 5.4 and then described in detail in Appendix C.1; these scenarios concern routing of frames concurrently received, error detection (CRC, frame size, and BAG), scheduling under high loads, priority management, and latency and throughput measurement.

In the testbenches, the generated frames are transmitted to the system directly writing them in the input filters using the `data_in`, `data_in_valid` and `data_in_EOP` (End of Packet) signals. In the smaller tests, where only a specific functionality is analysed, the internal signals and outputs are individually observed on the resulting waveforms. For the first type of test, where multiple frames are processed by the system, a module has been added in the testbench to each output port to save the packets transmitted by it, and successively compare them with the expected ones on that output. A simulation for a 1ms time lapse requires around 1min to be executed.

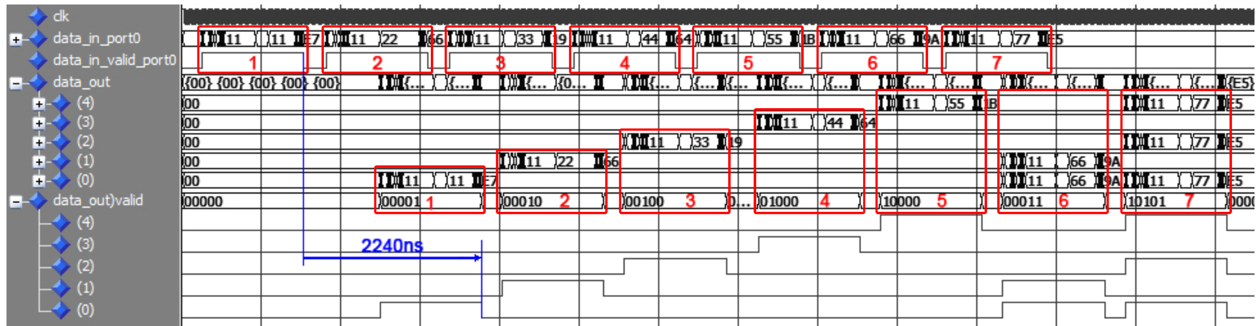

Figure 5.8 Basic routing functionalities

Table 5.4 List of the most significant tests

# Test	Objectives
1	Measure the technological latency for a frame with minimum size
2	Measure the technological latency for a frame with maximum size
3	Verify the routing functionalities of the switch when a single input is considered
4	Analyse how the scheduling of concurrently received frames is performed
5	Verify the priority management when a concurrent reception occurs
6	Analyse the behaviour of the system in presence of input data burst at a single input; congestion must be avoided
7	Analyse how buffer overload occurs for high traffic loads at all the input ports
8	Verify that frames that do not respect their BAG are dropped
9	Verify that frames with an erroneous frame control sequence are dropped
10	Verify that frames with non conforming size are dropped
11	Verify that frames with an excessive latency in the internal buffers are dropped

The requested routing functionalities can be observed in Figure 5.8, related to the test case 3, as given in Table 5.4, whose description can be found in Appendix C.1, where packets with different destinations are received only at the `input_port_0` and transmitted at all the output ports depending on each frame's virtual link. It can be observed that transmission starts before the subsequent reception is completed, avoiding data accumulation in the internal

buffers, and that all the outputs can be reached by the incoming frames. The *Scheduler* behaviour of the core can be observed in test cases 4 and 5, where routing features are further explored as well by concurrently sending frames at the 5 input ports of the core and by observing how they are forwarded to the corresponding outputs. In Figure 5.11, at the end of this chapter, it can be observed how the frames with higher priority are forwarded first (a description of the destination and priority of each VL is provided in the Appendices), and the remaining packets are forwarded as soon as possible. In the same figure, it can be observed how the `crossbar_config` array is used to configure the crossbar that connects each *Queue* to the required destinations: each element of the array corresponds to one output, "-1" means that is unconnected, otherwise the specified value corresponds to the *Queue* that is going to write on that *FIFO*.

5.4.2 System behaviour

The testbenches designed to study error detection proved that the specification [10] is completely met, and all the frames that do not respect it are correctly detected and discarded by the switch. When an error is detected by the Filter, the `drop` signal sent to the related *Queue* is set to '1', forcing the elimination of the packet from the storage memory. While frames that do not respect the BAG allocated for their VL are detected as soon as the *Manager* can process their header, all the other types of error can be identified only once the packet is completely received. One example of the performed filtering can be observed at the end of this chapter in Figure 5.12, where the packets received at ports 1 and 2 are correct, while the other three contain erroneous bits, which causes a mismatch between the CRC included in the frame and the one computed by the switch. The *Filters* corresponding to the input ports 3, 4, and 5 detects this mismatch and set the `drop` flag immediately after the end of the reception. Only the two valid packets are forwarded by the switch fabric. When no problem occurs, frames are correctly forwarded, as shown in Figure 5.8, following the execution flow previously described in Section 5.2.2.

The *Scheduler* behaviour can be observed in Figure 5.11, corresponding to the test case #5, where multiple frames with different priorities are concurrently received. The last byte of each packet is received at the same time, and the *Queues* are ready to answer to the *Scheduler* request two clock cycles later, when it is sure that the frames are valid. In the waveforms, it can be noticed how the `scheduler_request` signal is periodically asserted, while the `queue_response` is not null only after the reception of the last byte of the packet. Since all the frames share one destination (the output port number 0) they can't be forwarded concurrently, and a scheduling must be performed considering the frames priority. In this test, only the packets from ports 2 and 3 are critical, while the others have low priority. At

Table 5.5 Technological latency for frames of minimum and maximum length

Frame length	Technological latency
64 Bytes	2.24μs
1518 Bytes	45.9μs

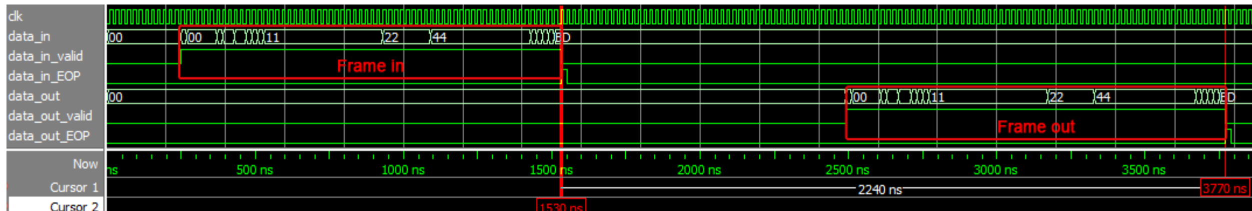

Figure 5.9 Minimal technological latency measure

the first round of the scheduling algorithm, frames 0 and 1 are ignored and, accordingly to the expectations, the critical frame 2 is transmitted first. Once its transmission is completed and the output *FIFO* is available again, the other critical packet is forwarded, and only after the other three frames are considered. The scheduler polling of *FIFOs* and *Queues* is repeated each 7 clock cycles (only 140ns for a 50MHz clock) thus adding a variable component to the latency of frames in the input buffers, that depends on when they are available and when the *Scheduler* tests that status. The time lapse between the time the `send` signal is asserted and the time the corresponding packet can be observed at the output corresponds to the time necessary to write that frame in the output buffers, and it depends on the packet's length.

5.4.3 Performance measurement

Some tests have been conceived to stress the switch and evaluate the performance it can provide in these situations; others are intended to measure parameters that characterize the core, such as the technological latency. A 50MHz clock has been used in all the simulations because it is achieved by the 5-ports switch fabric synthesized for the prototype.

The ARINC 664 specification [10] does not describe precisely where the latency must be measured; therefore, it has been decided that, for this core, it would be measured as the time lapse between the reception of the last byte of the frame by the input *Filter* and the moment the same byte leaves the output *FIFO*. Figure 5.9 represents the waveform corresponding to test case 1 presented in Appendix C.1, where only one frame is received and routed, thus recreating the situation specified in [10] for the technological latency measurement. It can be observed that 2240ns pass between the reception of the frame and its complete transmission,

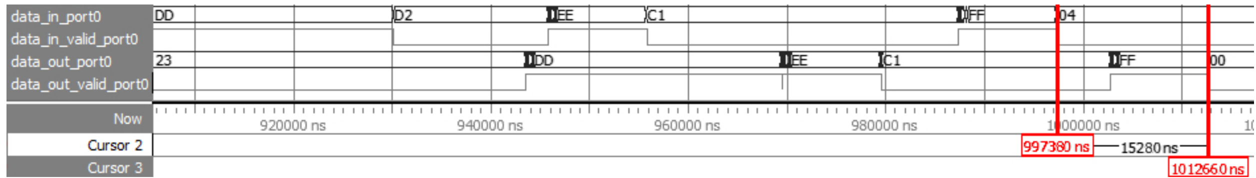

Figure 5.10 Latency of the last byte for a 1ms single-port burst

and that there is an empty space between the two processes, where the scheduling and forwarding to the corresponding outputs is performed by the core. In this case, a frame of minimum length is considered, but the latency is clearly influenced by this parameter, as already discussed in Section 5.3.2. In Table 5.5, the technological latencies resulting from the routing of a minimum-sized and a maximum-sized frames are shown, highlighting the great difference in the results due to the frame length.

Another important test evaluated the system performance under highly stressful traffic loads: frames are sent at maximum wire-speed (100Mbit/s) at the switch to analyse if there is congestion, where it occurs, and if buffer overflow can cause packet loss. Two different types of simulations have been conceived: one-port traffic and multi-port traffic. In the first case, frames are received only at one input; therefore, output contention is not possible and the core should be able to handle this traffic avoiding congestion and frame accumulation in the internal buffers. Since the time required to forward a packet in the output *FIFO* is half the time necessary to receive a new frame, the first can be completely transferred to the output buffer before the following one is completely received, when no contention is present; therefore no accumulation occurs in the input buffers, and since only one-port traffic is considered and the wire speed is the same at the two sides, no output congestion occurs either. This results can be observed in the test case 6 in the Appendix C.1 and in Figure 5.10, where the last packet (that has a size of 500 bytes) leaves the core with a delay of 15.28μs, that correspond to the technological latency of a 500 bytes frames (this delay is equal to 764 clock cycles, that correspond to 500 + 250 cycles for the forwarding and 14 for the scheduling). The same result is obtained both when incoming frames are directed towards different outputs or to the same one.

A different situation occurs for communications at maximum wire speed at all the input ports, since if there is output contention there will also be an inevitable accumulation of frames in the internal buffers. The most critical situation occurs when all the incoming packets (N*100Mbit/s) need to be routed towards the same output, which has a 100Mbit/s bandwidth available. In this case, congestion will occur and buffer overflow is inevitable if this burst is too long. The burst length that can be handled without buffer overflow

Table 5.6 Output FIFO overflow

# FIFO depth [16 bits words]	Time elapsed before FIFO out is full
2^{12}	433μs
2^{13}	1.56ms
2^{14}	3.73ms
2^{15}	8.9ms
2^{16}	17.3ms

depends on the depth of the FIFO of the output where all the frames are destined, while no significant problem is noticed in the *Queue*'s buffers. Maximum burst lengths for different *FIFO* depths are presented in Table 5.6. These results have been measured observing when the `fifo_full` signal of the most stressed output buffer becomes '1' for the first time. To evaluate the additional latency introduced by the scheduling and output contention with this kind of input load, a 1ms burst of frames with the same destination has been generated on each input (with an output FIFO size of 2^{16} words), and the time required for the routing of all these frames has been measured, as it has been done for the single-port traffic in Figure 5.10. The routing latency, equal to the entire delay minus the technological latency corresponding to the size of the last transmitted packet, is about 40μs; this value depends on the composition of the traffic.

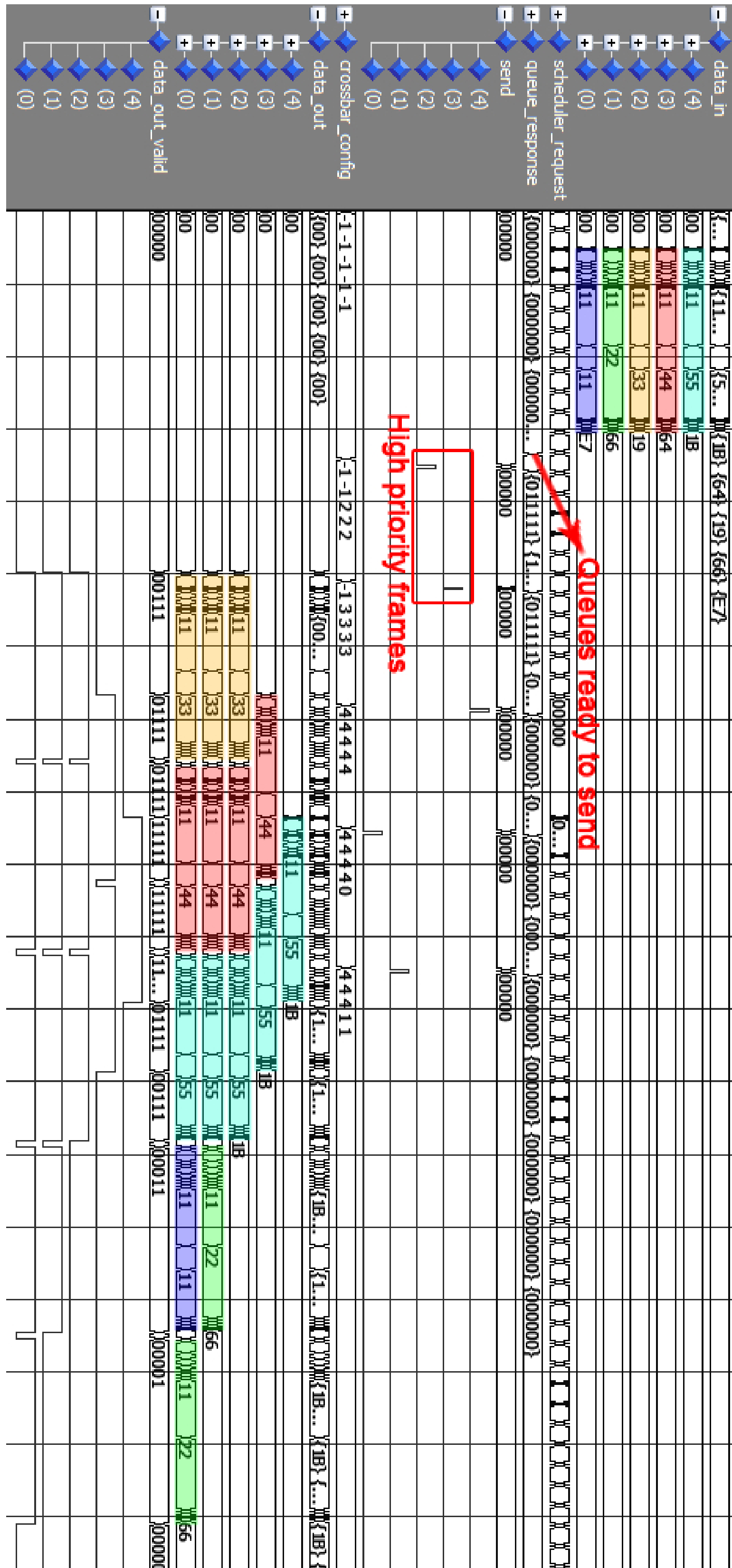

Figure 5.11 Routing of concurrently received frames and priority management

Figure 5.12 Example of error detection: bad CRC

CHAPITRE 6

CONCLUSION AND FUTURE WORK

The work presented in this thesis addressed the design and development of three fundamental modules of the hardware platform that will support the realization of a prototype for the communication network proposed by the AVIO 402 project. These three systems are the CAN bus controller, the End System and the Switch fabric of an AFDX network. Due to the different nature of these systems, their study and development has been separately and independently executed, and different approaches have been used for their realization. The objectives for each part of this thesis were slightly different because of the role of the module in the final prototype and its relations with other tasks of the AVIO 402 project.

The presented work constitutes a contribution to the progress of the Avio 402 project, thanks to its strong practical aspects and to the consideration on the implementation of physical systems. In the three cases the development stage required a previous study to determine the specification for each module, since the literature provides only limited information related to this topic. The cores have been also conceived to simplify their integration in a generic prototyping platform, thus addressing portability and reuse, in order to simplify their adaptation to the different needs of this early stage in the development of the network prototype. By designing the three systems from scratch, not only we provided the means to integrate and test different solutions to improve the network performance, but we also explored design challenges and constraints. This process allowed the evaluation of how the adaptation for avionic systems of two popular technologies, such as CAN and Ethernet, can impact the architecture of the modules required for their physical implementation, even when these differences mainly concerns higher levels of the ISO/OSI model..

CAN is a mature technology but, although an exhaustive literature is available for the implementation of the bus controller, its exploitation in an avionic environment is nothing has been published to discuss what modifications are required in the Data Link and Physical layers to adapt it to the CANaerospace and ARINC 825 specifications. The controller developed for this thesis was designed to minimize the core area, and to facilitate a potential migration towards ARINC 825. An architecture based on a central manager, where all the differences between these technologies are included, has been chosen instead of the traditional two-paths solution adopted, for example, by the HurriCANe core. After ARINC 825 has been chosen to implement the local field bus in the AVIO 402 network instead of CAN, the core was quickly adapted to this protocol by changing the manager's behaviour, eliminating overload frame

generation. The resources occupied by the IP core on the XC6SLX45T FPGA used for the prototype are less than those of the HurriCANe and MARIA cores used as a reference; this corresponds to a 2% utilization of device resources.

AFDX, on the other hand, is a recent technology, whose potential and limits are still being explored; research on various topics concerning this protocol is part of the AVIO 402 project as well. The ES development is strictly related to those studies since potential improvements and solutions identified by them will be eventually included in this system; considering this aspect, the complexity of this system, and the inevitable presence of a processor in the NCAP, a software approach has been preferred for its development. The proposed solution is based on the modification of the existing Ethernet protocol stack included in the Linux kernel to add the specific features of AFDX. An embedded system has been realized using a Microblaze processor, that runs a minimal configuration of the Linux 2.6.37 kernel. A first implementation of the BAG controller has also been obtained modifying the Ethernet MAC drivers and the Data Link layer. Major problems concerning the need for an ARINC 653-compliant OS and the recognition of some peripherals suggested a change of strategy: the embedded solution has been abandoned, and a PC has been connected to the FPGA via PCIe to realize the software part of the ES. The implementation of the AFDX protocol in the Linux kernel is still on progress, following the same design initially intended for the embedded processor. This integration of the ADFX protocol stack in the kernel protocol stack makes it easily portable to other platforms, it exploits standard Linux sockets as the interface with the NCAP applications, and it allows to easily modify the scheduling algorithm. This last characteristic will be a solid starting point for the implementation and testing for the optimal scheduling techniques proposed, for example, by Tawk *et al.* [31], who suggest a BAG-based policy as the best way to reduce ES delay.

Finally, the development of the switching core of the network router has been addressed. Differently from the ES, this module is more self-contained and it has been possible to implement it thoroughly. Since no study concerning its development has been found in the existing literature, an original architecture has been presented in this thesis: starting from an Ethernet switch fabric, functionalities required by AFDX have been identified and integrated in it. To perform a parallel processing of incoming frames, thus minimizing the delay introduced by the system to each communication, a space-division switching architecture provided with combined input/output buffering has been implemented in VHDL, and synthesized on the same XC6SLX45T FPGA used for the rest of the prototype. Synthesis results shows that up to 5 in/out ports can be supported using half of the device capacity, with an operating frequency that can go up to 50MHz. With a clock frequency of 50Mhz, the system can provide an 800Mbit/s throughput (if no output contention is considered) thanks to its 16-bit wide

datapath. AFDX specific features have been added to the design, originally intended for an Ethernet switch, including input filtering, latency control, and priority management. Input queues have been doubled to separate high priority from low priority traffic, preventing the latter from increasing the latency of critical frames, and to create a redundant path for them whenever a problem occurs to the high priority buffer.

While the CAN core has already been completely transformed in a dedicated ARINC 825 bus controller by other colleagues, the AFDX End System is still under development. The switching core as well needs some additional work to interface it with the Ethernet ports of the PC, and to complete the switch by developing a monitoring module and a properly managed configuration table, but its switching functions are completely implemented and tested. Once all the parts of the prototype are ready, the integration and testing stage will follow. When the prototype is functional, measurements on the network performance and reliability can be performed, and it will be possible to evaluate if the mathematical models developed in other tasks of the AVIO 402 project properly represent the network behaviour; it will be also interesting to see if the solutions identified thanks to these models can be practically integrated in the real network, and how this can be done. For example another paper by Tawk *et al.* [32] shows that BAG-based scheduling could be an optimal choice also for handling the traffic flows in all the network nodes, both ES and Switches; thanks to the prototype it will be possible to evaluate if this algorithm can bring the expected improvements.

Hopefully, the contributions of this thesis will not be limited to the products delivered to the AVIO 402 project for the realization of the prototype, but they will also help the improvement of the technology readiness level of the considered protocols, thanks to the design solutions proposed in this work. The three IP cores have been conceived considering portability not only of the developed code, but also of the ideas behind it, that consequently can be potentially reused when developing commercial avionic products. Not only the positive results and successful solutions obtained with this work, but also the problems and the related lessons learned will constitute an important reference when approaching the development of one of the considered systems.

BIBLIOGRAPHY

[1] J. Rosero, J. Ortega, E. Aldabas, and L. Romeral, “Moving towards a more elecric aircraft,” *IEEE A&E System Magazine*, 2007.

[2] IEEE, “Ieee std. 1451 standard for a smart transducer interface for sensors and actuators,” standard, Institute of Electrical and Electronical Engineers, September 1997.

[3] Xilinx, “Sp605 hardware user guide.” White paper UG526 (v1.5), 2011.

[4] Avnet, “Ism networking fmc module user guide,” 2010.

[5] H. Bauer, J. Scharbarg, C. Fraboul, *et al.*, “Applying trajectory approach with static priority queuing for improving the use of available afdx resources,” 2010.

[6] J. Zhang, S. Qiao, D. Li, and G. Shi, “Modeling and simulation of edf scheduling algorithm on afdx switch,” in *Signal Processing, Communications and Computing (ICSPCC), 2011 IEEE International Conference on*, pp. 1–4, IEEE, 2011.

[7] K. Wang, S. Wang, and J. Shi, “Integrated network reliability research for afdx,” in *Fluid Power and Mechatronics (FPM), 2011 International Conference on*, pp. 973–976, IEEE, 2011.

[8] R. Bosh, “Can specification version 2.0,” specification, Robert Bosh GmbH, 1991.

[9] AEEC, “Arinc specification 825, general standardization of can for airborne use,” standard, Areonautical Radio INC., 2007.

[10] ARINC, “Aircraft data network part 7, avionics full-duplex switched ethernet network,” standard, ARINC, September 2009.

[11] J.-S. Young, C.-Y. Hua, and K.-H. Chang, “The feasibility study of sru/lru for air vehicle communication network based on can bus,” *Journal of Aeronautics, Astronautics and Aviation, Series A*, vol. 42, no. 1, pp. 57–66, 2010.

[12] L. Zhang, M. Zhu, and Z. Wu, “Can bus application layer protocol design for unmanned helicopter system,” *Journal of Beijing University of Aeronautics and Astronautics*, vol. 37, no. 11, pp. 1264–1270, 2011.

[13] T. Lv, N. Hu, Z. Wu, and N. Huang, "The analysis of end-to-end delays based on afdx configuration," in *Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on*, pp. 1296–1300, IEEE, 2011.

[14] HOLT, "Avionics can controller with integrated transceiver." HI-3110 Data Sheet, September 2011.

[15] L. Stagnaro, "Hurricane - free vhdl can controller core," white paper, European Space Agency, March 2000.

[16] J. Reges and E. Santos, "A vhdl can module for smart sensors," in *Programmable Logic, 2008 4th Southern Conference on*, pp. 179–182, IEEE, 2008.

[17] F. Carvalho, I. Jansch-Porto, E. Freitas, and C. Pereira, "The tinycan: an optimized can controller ip for fpga-based platforms," in *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, vol. 1, pp. 4–pp, IEEE, 2005.

[18] P. Prisaznuk, "Arinc 653 role in integrated modular avionics (ima)," in *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pp. 1–E, IEEE, 2008.

[19] L. Kinnan, J. Wlad, and P. Rogers, "Porting applications to an arinc 653 compliant ima platform using vxworks as an example," in *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, vol. 2, pp. 10–B, IEEE, 2004.

[20] I. Khazali, M. Boulais, and P. Cole, "Afdx software network stack implementation—practical lessons learned," in *Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th*, pp. 1–B, IEEE, 2009.

[21] B. Yu, T. Zhang, and D. Liu, "Low cost afdx end system based on system on a programmable chip," *Applied Mechanics and Materials*, vol. 29, pp. 2308–2311, 2010.

[22] X. Chen, X. Xiang, and J. Wan, "A software implemetation of afdx end system," in *New Trends in Information and Service Science, 2009. NISS'09. International Conference on*, pp. 558–563, IEEE, 2009.

[23] E. Erdinç, *Soft AFDX (Avionics Full Duplex Switched Ethernet) End System Implementation With Standard PC and Ethernet Card*. PhD thesis, Middle East Technical University, 2010.

[24] P. Pusik, J. Hangyun, S. Daekyo, L. Kitaeg, and Y. Jongho, "Implementation of the hardwired afdx nic." 1st Asia NetFPGA Developers Workshop, June 2010.

[25] Xilinx, “Architecting arinc 664 part 7 (afdx) solutions.” White Paper XAPP1130, 2009.

[26] H. Chao and B. Liu, *High performance switches and routers*. Wiley-IEEE Press, 2007.

[27] F. Ajmone Marsan, A. Bianco, P. Giaccone, E. Leonardi, and E. Neri, “Packet scheduling in input-queued cell-based switches,” in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 1085–1094, IEEE, 2001.

[28] N. McKeown, *Scheduling algorithms for input-queued cell switches*. PhD thesis, University of California, 1992.

[29] D. Serpanos and P. Antoniadis, “Firm: A class of distributed scheduling algorithms for high-speed atm switches with multiple input queues,” in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 548–555, IEEE, 2000.

[30] M. Karol, M. Hluchyj, and S. Morgan, “Input versus output queueing on a space-division packet switch,” *Communications, IEEE Transactions on*, vol. 35, no. 12, pp. 1347–1356, 1987.

[31] M. Tawk, G. Zhu, L. Jian, X. Liu, Y. Savaria, and F. Hu, “Optimal scheduling and delay analysis for afdx end-systems,” 2011.

[32] M. Tawk, G. Zhu, Y. Savaria, X. Liu, J. Li, and F. Hu, “A tight end-to-end delay bound and scheduling optimization of an avionics afdx network,” in *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pp. 7B3–1, IEEE, 2011.

[33] M. Yao, Z. Qiu, and K. Kwak, “Leaky bucket algorithms in afdx,” *Electronics Letters*, vol. 45, no. 11, pp. 543–545, 2009.

ANNEXE A

IMPLEMENTATION OF THE MANAGER OF THE CAN CORE

The VHDL code of the Finite State Machine that control the execution flow of the *Manager* of the CAN controller, and consequently of the entire core, is given in this appendix.

VHDL code of the Manager's FSM

```
------------------------------------------------------------------------------------------
-- Title         : Manager
-- Project       : AVIO 402
------------------------------------------------------------------------------------------
-- File          : fsm.vhd
-- Author        : Davide Trentin
-- Created       : 2010/07/01
-- Last Modified : 2010/08/15
------------------------------------------------------------------------------------------
-- Description   : this state machine must manage the execution flow of both reception
--         and transmission of data on a CAN bus. It must also manage the error and
--         overload routines.
------------------------------------------------------------------------------------------
-- Modification history :
-- 2010/07/01 : created
-- 2010/07/23 : modified => the error signal is checked before entering the "case"
-- 2010/08/15 : modified => - counter must be reset when an error routine starts
--                          - to avoid loosing a cycle changing state when using the
--         counter the  state is changed when cnt = n-1 AND t_sync = '1'
------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.states.all;
use work.states_error.all;

entity fsm is
port(
    clk            : in std_logic;
    rst            : in std_logic;
    -- The following signals come from the synchronization modules and are used to
    -- synchronize the state machine with the communication on the bus
    t_sync         : in std_logic; -- It goes high at the beginning of each bit time
    t_sample       : in std_logic; -- It goes high when the value of the bis is sampled

    -- The following signals are related to error and overload situations
    ovl_req        : in std_logic; -- When it is '1' the node need to send an overload frame
    err            : in std_logic; -- It is '1' when an error is detected
    err_state      : in std_logic_vector(1 downto 0); -- It specifies the error state

    -- Control signals
    check          : in std_logic; -- It comes from the bitcheck module and it is '1' when
                                   -- the bit sent and the received one are different
    data_out_ready : in std_logic; -- Specifies when there are data available to be sent
    tx_ctl         : in std_logic; -- It is used to synchronize the FSM with a transmission
    end_rec        : in std_logic; -- It goes high when the CRC sequence has been received

    -- Outputs
    send_bit       : out std_logic; -- It forces a dominant bit on the bus
```

```
51      reg_available  : out std_logic; -- It tells the bitstream gen that data have been sent
52      state          : out std_logic_vector(4 downto 0) -- It is the state of the FSM
53  );
54  end fsm;
55
56  architecture behav of fsm is
57
58  -- signals
59  signal current_state    : std_logic_vector(4 downto 0);
60  -- signal next_state    : std_logic_vector(4 downto 0);
61  signal cnt              : std_logic_vector(7 downto 0);
62  signal tr_err_pass      : std_logic;
63  signal tr_successful    : std_logic;
64  signal rd_bus           : std_logic;
65
66  begin
67      state <= current_state;
68
69      next_state_process : process(clk, rst)
70      begin
71          if (rst = '1') then
72              current_state <= STATE_RESET;
73          elsif (clk'event and clk = '1') then
74
75      -- All the states where the presence of an error must be checked, and where consequently
76      -- there could be a transition to an error state have the bit current_state[4]='0'.
77      -- For all the other states the error flag isn't verified.
78              if(err = '1' AND current_state(4) = '0') then
79                  if(err_state = ERR_ACT_STATE) then
80                      cnt <= x"00";
81                      current_state <= STATE_ERR_ACT;
82                  -- Only err_state="00" and "01" are expected
83                  elsif(err_state = ERR_PASS_STATE)then
84                      cnt <= x"00";
85                      current_state <= STATE_ERR_PASS;
86                  else
87                      current_state <= STATE_BUS_OFF;
88                  end if;
89              else
90                  case current_state is
91                  when STATE_RESET =>
92  -- RST: Whene the system is turned on or when it is reinitialized by the reset signal the
93  -- state machine starts its execution flow from a reset state. In this state all the internal
94  -- registers and the outputs are set as zeros.
95                      send_bit <= '0';
96                      cnt <= x"00";
97                      current_state <= STATE_WAIT_IDLE;
98                      tr_err_pass <= '0';
99                      rd_bus <= '0';
100                     tr_successful <= '0';
101                     reg_available <= '0';
102
103                 when STATE_WAIT_IDLE =>
104 -- WAIT_IDLE: When the reset state is over the FSM must check is if the bus is in an
105 -- idle state or if there is a transmission going on. To do it it must verify that the
106 -- value on the bus remains recessive for at least 10 bit times. This time correspond
107 -- to the length of the End of Frame and Intermission  fields of a frame. The error flag
108 -- is the signal with the highest priority and it can force the FSM to move to the error
109 -- active or error passive state (it depends on the "error state" of the error controller).
110 -- If no error is present the FSM then moves to the state IDLE.
111                     if (check = '1') then
112                         cnt <= x"00";
113                         current_state <= STATE_WAIT_IDLE;
114                     else
115                         if (cnt = x"0a") then
116                             cnt <= x"00";
117                             current_state <= STATE_IDLE;
118                         elsif (t_sync = '1') then
```

```
119                             cnt <= cnt + 1;
120                             current_state <= STATE_WAIT_IDLE;
121                         end if;
122                     end if;
123
124                 when STATE_IDLE =>
125 -- IDLE: Starting from this state there are three possible destination state for the next cycle:
126 --  - If the error state is "bus off" the next state will be BUS_OFF;
127 --  - Else if there is an overload request the next state will be OVERLOAD;
128 --  - Else if there is a Start of Frame (the signal check becomes '1') a RECEPTION will start;
129 --  - Else if there is a frame ready to be sent a TRANSMISSION will start;
130 --  - Else the next state will still be IDLE.
131 -- NB: the reception has the priority over the transimission
132                     if (err_state = BUS_OFF_STATE) then
133                         current_state <= STATE_BUS_OFF;
134                     elsif (ovl_req = '1') then
135                         current_state <= STATE_OVL;
136                     elsif (check = '1') then
137                         current_state <= STATE_REC;
138                     elsif (data_out_ready = '1') then
139                         current_state <= STATE_ARB;
140                     else
141                         current_state <= STATE_IDLE;
142                     end if;
143
144                 when STATE_REC =>
145 -- RECEPTION: The signal with the highest priority is the error signal that can force the
146 -- controller to send an error frame immediately. If the reception is correct the FSM remains
147 -- in this state until the destuffer receives the whole CRC sequence and sets the signal
148 -- "end reception" at '1'. When the FSM detects tx_ctrl = '1' it moves to te state CRC_DELIM
149                     if (end_rec = '1') then
150     -- I can eliminate CRC_DELIM asking the destuffer to send me the signal AFTER the delimiter
151                         current_state <= STATE_CRC_DELIM;
152                     else
153                         current_state <= STATE_REC;
154                     end if;
155
156                 when STATE_CRC_DELIM =>
157 -- CRC_DELIM: The FSM remains in this state for one bit time and the next state depends on
158 -- the result of the CRC computation:
159 -- - If a CRC error is detecte the error routine will be initiated (active or passive);
160 -- - Else, that means that there is correpondance between the received and the computed CRCs,
161 --   the next state is SEND_ACK.
162                 if (cnt = x"00") then
163                     if(t_sync = '1') then
164                         cnt <= x"01";
165                     end if;
166                     current_state <= STATE_CRC_DELIM;
167                 else
168                     if(t_sync = '1') then
169                         current_state <= STATE_SEND_ACK;
170                         send_bit <= '1';
171                         cnt <= x"00";
172                     end if;
173                     current_state <= STATE_CRC_DELIM;
174                 end if;
175
176                 when STATE_SEND_ACK =>
177 -- SEND_ACK: If a frame is received with no errors an acknowledge must be sent during the
178 -- acknowledge slot. This state consequently lasts one bit time and force the serializer to
179 -- send a dominant bit on the bus. The following state is ACK_DELIM.
180                     send_bit <= '1';
181                     if (t_sync = '1') then
182                         send_bit <= '0';
183                         current_state <= STATE_ACK_DELIM;
184                     else
185                         current_state <= STATE_SEND_ACK;
186                     end if;
```

```
187
188                 when STATE_ACK_DELIM =>
189  -- ACK_DELIM: The FSM must one bit time before moving to the End Of Frame state. The error
190  -- flag still has the priority and can make an error routin to start. The serializer must
191  -- now be forced to send a recessive bit.
192                     send_bit <= '0';
193                     if (t_sync = '1') then
194                         current_state <= STATE_EOF;
195                     else
196                         current_state <= STATE_ACK_DELIM;
197                     end if;
198
199                 when STATE_EOF =>
200  -- END_OF_FRAME: This state must last seven bit times and it is followed by the INTERMISSION
201  -- state. Again, the error flag has the priority and can force the start of an error routine.
202                     if(cnt = x"06" AND t_sync = '1') then
203                         cnt <= x"00";
204                         if(tr_successful = '1')then
205                             reg_available <= '1';
206                             tr_successful <= '0';
207                         end if;
208                         current_state <= STATE_INTER;
209                         elsif (t_sync = '1') then
210                             cnt <= cnt + 1;
211                             current_state <= STATE_EOF;
212                         else
213                             current_state <= STATE_EOF;
214                         end if;
215
216                 when STATE_ARB =>
217  -- ARBITRATION: The error flag has always the priority to start an error routine (the next
218  -- state will be ERR_ACTIVE or ERR_PASSIVE in function if the error state). During this
219  -- state if the bit check detect a difference between the bit read and the bit sent it means
220  -- that the arbitration has been lost (signal "check" = '1') and the FSM must pass immedialtely
221  -- to the RECEPTION state. If check remains '0' the FSM remains in this state until the
222  -- serilizer ends the arbitration part o the frame (and puts the signal "tx controller" at '1').
223  -- At this point the FSM moves to the state TRANSMISSION.
224                     if (t_sample = '1' AND check = '1') then
225                         current_state <= STATE_REC;
226                     elsif (tx_ctl = '1') then
227                         current_state <= STATE_TR;
228                     else
229                         current_state <= STATE_ARB;
230                     end if;
231
232                 when STATE_TR =>
233  -- TRANSMISSION: The FSM remains in this state until the serilizer sends the last bit of the
234  -- CRC frame and puts "tx controller" high ('1') again. When this happens the state changes to
235  -- WAIT_ACK. The error flag has always the priority and is it is '1' the next state will be ERR_ACT
236  -- or ERR_PASS.
237                     if (tx_ctl = '1') then
238                         current_state <= STATE_WAIT_ACK;
239                     else
240                         current_state <= STATE_TR;
241                     end if;
242
243                 when STATE_WAIT_ACK =>
244  -- WAIT_ACK: This state lasts one bit time that corresponds to the acknowledge slot of a
245  -- transmission. If a recessive bit is read on the bus an acknowledge error is detected.
246  -- If the acknowledge is receive the machine moves to the SUCCESSFUL state.
247                     if (cnt = x"01") then
248                         if (t_sync = '1') then
249                             current_state <= STATE_SUCC;
250                             cnt <= x"00";
251                         else
252                             current_state <= STATE_WAIT_ACK;
253                         end if;
254                     else
```

```
255                         if (t_sync = '1') then
256                             cnt <= x"01";
257                         end if;
258                         current_state <= STATE_WAIT_ACK;
259                     end if;
260
261                 when STATE_SUCC =>
262 -- SUCCESSFUL: This state lasts one bit time and corresponds to the acknowledge delimiter.
263 -- If the error state of the node os passive a signal "error_passive_transmission" becomes
264 -- '1', otherwise it remains '0'. Successively the FSM goes to the state END_OF_FRAME.
265                     if (err_state = ERR_PASS_STATE) then
266                         tr_err_pass <= '1';
267                     end if;
268
269                     if (t_sync = '1') then
270                         tr_successful <= '1';
271                         current_state <= STATE_EOF;
272                     else
273                         current_state <= STATE_SUCC;
274                     end if;
275
276                 when STATE_INTER =>
277 -- INTERMISSION: The FSM remains in this state for two bit times:
278 -- - If a dominant bit is read on the bus an overload frame must be generated;
279 -- - Else the FSM goes to the END_INTERMISSION state.
280                     reg_available <= '0';
281
282                     if (check = '1') then
283                         rd_bus <= '1';
284                     end if;
285
286                     if (cnt = x"01" AND t_sync = '1') then
287                         cnt <= x"00";
288                         if(rd_bus = '1') then
289                             rd_bus <= '0';
290                             current_state <= STATE_OVL;
291                         else
292                         current_state <= STATE_END_INTER;
293                         end if;
294                     elsif (t_sync = '1') then
295                         if(rd_bus = '1') then
296                             rd_bus <= '0';
297                             cnt <= x"00";
298                             current_state <= STATE_OVL;
299                         else
300                             cnt <= cnt + 1;
301                             current_state <= STATE_INTER;
302                         end if;
303                     else
304                         current_state <= STATE_INTER;
305                     end if;
306
307                 when STATE_END_INTER =>
308 -- END_INTERMISSION: It lasts one bit time:
309 -- If a dominant bit is read it's interpreted as a SoF and the FSM goes to the state RECEPTION;
310 -- Else if it is recessive and error_passive_transmission is '0' the next state is IDLE;
311 -- Else if it is recessive and error_passive_transmission is '1' the next state is SUSPEND;
312                     if (check = '1') then
313                         rd_bus <= '1';
314                     end if;
315
316                     if (t_sync = '1') then
317                         if (rd_bus = '1') then
318                             rd_bus <= '0';
319                             current_state <= STATE_REC;
320                         elsif (tr_err_pass = '1') then
321                             tr_err_pass <= '0';
322                             current_state <= STATE_SUSPEND;
```

```
323                             else
324                                 current_state <= STATE_IDLE;
325                             end if;
326                         else
327                             current_state <= STATE_END_INTER;
328                         end if;
329
330                     when STATE_SUSPEND =>
331     -- SUSPEND: This state is necessary to let the nodes in an error active state to have a
332     -- higher transmission priority over the ones in an error passive state. It lasts 8 bit
333     -- times and force the serilizer to send recessive bits. If a dominant bit is read on the
334     -- bus the machine moves immediately to the RECEPTION state and error_passive_transmission
335     -- signal is set to '0', otherwise it will go back to the IDLE state after the eight bit times.
336                         if (check = '1') then
337                             cnt <= x"00";
338                             current_state <= STATE_REC;
339                         elsif (cnt = x"07" AND t_sync = '1') then
340                             cnt <= x"00";
341                             current_state <= STATE_IDLE;
342                         elsif (t_sync = '1') then
343                             cnt <= cnt + 1;
344                             current_state <= STATE_SUSPEND;
345                         else
346                             current_state <= STATE_SUSPEND;
347                         end if;
348
349                     when STATE_BUS_OFF =>
350     -- BUS_OFF: The FSM remains in this state as long as the error state is "bus off".
351     -- When it chenges to "error active" the FSM will go back to the IDLE state.
352                         if (err_state = BUS_OFF_STATE) then
353                             current_state <= STATE_BUS_OFF;
354                         else
355                             current_state <= STATE_IDLE;
356                         end if;
357
358                     when STATE_ERR_ACT =>
359     -- ERR_ACT: The machine is in this state if an error has been detected        and the error
360     -- state is active. This state lasts  six bit times and force the serilizer to send six
361     -- dominant bits on the bus. After that the state becomes EXTENSION.
362                         send_bit <= '1';
363
364         -- There is a seventh count because the first one is the beginning of the error frame
365         -- The error frame is consequently synchronized with the bus starting from the first
366         -- bit time after the error signal becomes '1'
367                         if (cnt = x"06" AND t_sync = '1') then
368                             cnt <= x"00";
369                             current_state <= STATE_EXT;
370                         elsif (t_sync = '1') then
371                             cnt <= cnt + 1;
372                             current_state <= STATE_ERR_ACT;
373                         else
374                             current_state <= STATE_ERR_ACT;
375                         end if;
376
377                     when STATE_ERR_PASS =>
378     -- ERR_PASS: The machine is in this state if an error has been detected and the error
379     -- state is passive. The FSM remains in this state until six consecutive recessive bits
380     -- have been detected on the bus. When this happens the state becomes DELIM.
381     -- Each time a dominant bit is read on the bus the count ot the 6 recessive ones is reset.
382                         if (check = '1') then
383                             cnt <= x"00";
384                             current_state <= STATE_ERR_PASS;
385                         elsif (cnt = x"06" AND t_sync = '1') then
386                             cnt <= x"00";
387                             current_state <= STATE_DELIM;
388                         elsif (t_sync = '1') then
389                             cnt <= cnt + 1;
390                             current_state <= STATE_ERR_PASS;
```

```
391                     end if;
392
393                 when STATE_OVL =>
394 -- OVERLOAD: When an overload frame must be sent the FSM arrives in this states and
395 -- remains here for six bit times. The serializer is forced to send a dominant bit
396 -- for all this time. After the 6 bit times the machine moves to the state EXTENSION.
397                     send_bit <= '1';
398
399                     if (cnt = x"06" AND t_sync = '1') then
400                         cnt <= x"00";
401                         current_state <= STATE_EXT;
402                     elsif (t_sync = '1') then
403                         cnt <= cnt + 1;
404                         current_state <= STATE_OVL;
405                     else
406                         current_state <= STATE_OVL;
407                     end if;
408
409                 when STATE_DELIM =>
410 -- DELIM: It lasts seven bit times before changing to DELIM_LAST. If in the meanwhile
411 -- a dominant bit is read on the bus an error routine will be started immediately
412 -- (the state changes to ERR_ACT or ERR_PASS).
413                     if (cnt = x"06" AND t_sync = '1') then
414                         cnt <= x"00";
415                         current_state <= STATE_DELIM_LAST;
416                     elsif (t_sync = '1') then
417                         cnt <= cnt + 1;
418                         current_state <= STATE_DELIM;
419                     else
420                         current_state <= STATE_DELIM;
421                     end if;
422
423                 when STATE_DELIM_LAST =>
424 -- DELIM_LAST: It corresponds to the last bit of the error/overload delimiter.
425 -- If a dominant bit is received from the bus an overload frame will be generated,
426 -- otherwise the FSM will move to the IDLE state after one bit time.
427                     if (check = '1') then
428                         rd_bus <= '1';
429                     end if;
430
431                     if (t_sync = '1') then
432                         if (rd_bus = '1') then
433                             rd_bus <= '0';
434                             current_state <= STATE_OVL;
435                         else
436                             current_state <= STATE_IDLE;
437                         end if;
438                     else
439                         current_state <= STATE_DELIM_LAST;
440                     end if;
441
442                 when STATE_EXT =>
443 -- EXTENSION: The machine needs this state to wait for the end of the transmission
444 -- of error flags by all the other nodes. When a recessive bit is finally read it
445 -- moves to the state DELIM.
446                     send_bit <= '0';
447
448                     if (check = '1') then
449                         rd_bus <= '1';
450                     end if;
451
452                     if (t_sync = '1') then
453                         if (rd_bus = '1') then
454                             rd_bus <= '0';
455                             current_state <= STATE_EXT;
456                         else
457                             current_state <= STATE_DELIM;
458                         end if;
```

```
459                     else
460                         current_state <= STATE_EXT;
461                     end if;
462
463                 when others =>
464     -- This situation shold never occur
465                 end case;
466             end if;
467         end if;
468 end process;
469
470 end behav;
471
```

ANNEXE B

IMPLEMENTATION OF THE AFDX SWITCH

The VHDL implementation of the most important modules of the switch fabric core is provided here. The code that realizes the *Queue*, the *Manager*, the *Scheduler*, and the *Filter* is reported in this appendix to show how the requested features have been included in the design.

B.1 Queue implementation

```
1  -------------------------------------------------------------------------------------------
2  -- Title        : Queue
3  -- Project      : AVIO 402
4  -------------------------------------------------------------------------------------------
5  -- File         : Queue.vhd
6  -- Author       : Davide Trentin
7  -------------------------------------------------------------------------------------------
8  -- Description   : the queue module must stock incoming frames waiting for them to be
9  --  forwarded to the output FIFOs. Frames arrive from the Filter module using 16-bit
10 --  words and are delimited by an EndOfPacket(EOP) signal. If the drop signal is
11 --  asserted the last received frame must be discarded. A frame is considered valid only
12 --  when the manager sends the destination and priority information, if a new frame is
13 --  received before this moment the previous one is overwritten. High priority frames
14 --  must be sent before low priority ones. When the scheduler gives the start input the
15 --  fisrt frame of the queue is sent to the FIFO.
16 -------------------------------------------------------------------------------------------
17 library IEEE;
18 use ieee.std_logic_1164.all;
19 use ieee.std_logic_unsigned.all;
20 use ieee.numeric_std.all;
21 -- Project Libraries
22 use work.Router_Types.all;
23
24 entity Queue is
25     port(
26     -- Inputs
27         clk                     : in        std_logic;
28         reset                   : in        std_logic;
29     -- From Input Filter
30         data_in                 : in        std_logic_vector(PORT_IN_LENGTH-1 downto 0);
31         data_in_valid           : in        std_logic;
32         data_in_EOP             : in        std_logic;
33         data_in_drop            : in        std_logic;
34     -- From Manager
35         destination_in          : in std_logic_vector(NBR_PORTS-1 downto 0);
36         priority                : in std_logic;
37         destination_valid       : in std_logic;
38     -- From Scheduler
39         request                 : in std_logic;
40         to_send                 : in std_logic;
41     -- Outputs
42     -- To Scheduler
43         destination_out         : out std_logic_vector(NBR_PORTS downto 0);
44     -- To Crossbar
45         data_out                : out std_logic_vector(15 downto 0);
46         data_out_valid          : out std_logic;
```

```
47          data_out_EOP            : out std_logic;
48      -- additional signal for the timer used for latency control
49          tick                    : in std_logic_vector(timer_range-1 downto 0)
50          );
51  end Queue;
52
53  architecture behaviour of Queue is
54  -- TYPES -----------------------------------------------------------------------------
55  type state_repack_type is (INIT, IDLE, RECEIVE);
56  type state_tx_type is (INIT, IDLE, SEND_L, SEND_H, END_SEND);
57  type dest_port_type is array (INTEGER range <>) of std_logic_vector(NBR_PORTS-1 downto 0);
58  type addr_type is array (INTEGER range <>) of std_logic_vector(QUEUE_RAM_DEPTH-1 downto 0);
59  type priority_type is array (INTEGER range <>) of std_logic;
60  type latency_type is array (integer range <>) of std_logic_vector(timer_range-1 downto 0);
61  -- SIGNALS ---------------------------------------------------------------------------
62  -- Possible states for the communication with the repack, the scheduler, and the FIFO
63  signal l_state_repack: state_repack_type := INIT;
64  signal h_state_repack: state_repack_type := INIT;
65  signal state_tx: state_tx_type := INIT;
66  -- Counters
67  signal l_frame_count      : integer := 0; -- It counts how many frames are in the L queue
68  signal h_frame_count      : integer := 0; -- It counts how many frames are in the H queue
69  signal l_frame_rx         : integer := 0; -- Received frames in the L queue
70  signal l_frame_tx         : integer := 0; -- Transmitted frames in the L queue
71  signal h_frame_rx         : integer := 0; -- Received frames in the H queue
72  signal h_frame_tx         : integer := 0; -- Transmitted frames in the H queue
73  signal l_table_count      : integer := 1; -- Used to keep track of the current frame
74  signal l_table_count_old  : integer := 1;
75  signal h_table_count      : integer := 1;
76  signal h_table_count_old  : integer := 1;
77  signal l_next             : integer := 1; -- pointer to the next frame to be sent (L queue)
78  signal h_next             : integer := 1;
79  -- Full/Empty
80  signal l_queue_empty      : boolean := false;
81  signal l_queue_full       : boolean := false;
82  signal h_queue_empty      : boolean := false;
83  signal h_queue_full       : boolean := false;
84  -- Tables
85  signal l_start_addr       : addr_type(QUEUE_DEPTH downto 1);
86  signal l_end_addr         : addr_type(QUEUE_DEPTH downto 1);
87  signal l_dest_port        : dest_port_type(QUEUE_DEPTH downto 1);
88  signal l_frame_priority   : priority_type(QUEUE_DEPTH downto 1);
89  signal h_start_addr       : addr_type(QUEUE_DEPTH downto 1);
90  signal h_end_addr         : addr_type(QUEUE_DEPTH downto 1);
91  signal h_dest_port        : dest_port_type(QUEUE_DEPTH downto 1);
92  signal h_frame_priority   : priority_type(QUEUE_DEPTH downto 1);
93  signal l_latency          : latency_type(QUEUE_DEPTH downto 1);
94  signal h_latency          : latency_type(QUEUE_DEPTH downto 1);
95  -- Signals for the communication with the internal RAM to stack frames
96  signal l_wren         : std_logic := '0';
97  signal l_rdaddress    : std_logic_vector(QUEUE_RAM_DEPTH-1 downto 0) := (others => '0');
98  signal l_wraddress    : std_logic_vector(QUEUE_RAM_DEPTH-1 downto 0) := (others => '0');
99  signal data           : std_logic_vector(PORT_IN_LENGTH-1 downto 0) := (others => '0');
100 signal l_q            : std_logic_vector(PORT_IN_LENGTH-1 downto 0) := (others => '0');
101 signal h_wren         : std_logic := '0';
102 signal h_rdaddress    : std_logic_vector(QUEUE_RAM_DEPTH-1 downto 0) := (others => '0');
103 signal h_wraddress    : std_logic_vector(QUEUE_RAM_DEPTH-1 downto 0) := (others => '0');
104 signal h_q            : std_logic_vector(PORT_IN_LENGTH-1 downto 0) := (others => '0');
105 -- ERROR
106 signal null_vector    : std_logic_vector(NBR_PORTS downto 0) := (others => '0');
107 -- Output registers
108 signal destination_out_reg : std_logic_vector(NBR_PORTS downto 0) := (others => '0');
109 signal send_priority       : std_logic := '0';
110
111 -- COMPONENTS ------------------------------------------------------------------------
112 component ram_fifo
113 generic(ADDR_SIZE : integer := QUEUE_RAM_DEPTH;
114     PORT_IN_LENGTH : integer := PORT_IN_LENGTH);
```

```
115 port(
116     clka : in  STD_LOGIC;
117     wea: in std_logic;
118     addra: in std_logic_vector(ADDR_SIZE-1 downto 0);
119     dina: in std_logic_vector(PORT_IN_LENGTH-1 downto 0);
120     clkb: in std_logic;
121     addrb: in std_logic_vector(ADDR_SIZE-1 downto 0);
122     doutb: out std_logic_vector(PORT_IN_LENGTH-1 downto 0));
123 end component;
124
125 begin
126     Lmem : ram_fifo
127     port map (
128         clka    => clk,
129         wea     => l_wren,
130         addra   => l_wraddress,
131         dina    => data,
132         clkb    => clk,
133         addrb   => l_rdaddress,
134         doutb   => l_q
135     );
136
137     Hmem : ram_fifo
138     port map (
139         clka    => clk,
140         wea     => h_wren,
141         addra   => h_wraddress,
142         dina    => data,
143         clkb    => clk,
144         addrb   => h_rdaddress,
145         doutb   => h_q
146         );
147
148 -- COMBINATORIAL ----------------------------------------------------------------------
149     destination_out <= destination_out_reg;
150     l_frame_count <= l_frame_rx - l_frame_tx;
151     h_frame_count <= h_frame_rx - h_frame_tx;
152     data_out <= l_q when (send_priority = '0') else h_q;
153
154 -- PROCESS ----------------------------------------------------------------------------
155 -- Old_counter: keeps track of the previous table_count value in case the currently
156 -- received frame is dropped
157 ---------------------------------------------------------------------------------------
158 Old_counter: process(clk, reset)
159 begin
160         if(l_table_count = 1)then
161             l_table_count_old <= QUEUE_DEPTH;
162         else
163             l_table_count_old <= l_table_count - 1;
164         end if;
165
166         if(h_table_count = 1)then
167             h_table_count_old <= QUEUE_DEPTH;
168         else
169             h_table_count_old <= h_table_count - 1;
170         end if;
171 end process;
172
173 -- PROCESS ----------------------------------------------------------------------------
174 -- Full_FIFO: checks if the H and L FIFOs are full. In that case the corresponding flag
175 -- is set high. No need for an empty signal since the frame counter replace that function
176 ---------------------------------------------------------------------------------------
177 Full_FIFO: process(clk, reset)
178     variable v_l_free_space : integer := 0;
179     variable v_l_rd_addr : integer := 0;
180     variable v_l_wr_addr : integer := 0;
181     variable v_h_free_space : integer := 0;
182     variable v_h_rd_addr : integer := 0;
```

```
183     variable v_h_wr_addr : integer := 0;
184 begin
185     if(reset = '1')then
186         l_queue_full <= false;
187         h_queue_full <= false;
188 
189     elsif ( clk'event and clk = '1') then
190         v_l_rd_addr := conv_integer('0'&l_rdaddress);
191         v_l_wr_addr := conv_integer('0'&l_wraddress);
192         v_h_rd_addr := conv_integer('0'&h_rdaddress);
193         v_h_wr_addr := conv_integer('0'&h_wraddress);
194                 -- LOW PRIORITY - Free space computation
195         if(v_l_wr_addr >= v_l_rd_addr)then
196             v_l_free_space := QUEUE_RAM_WORDS - (v_l_wr_addr - v_l_rd_addr);
197         else
198             v_l_free_space := v_l_rd_addr - v_l_wr_addr;
199         end if;
200 
201         -- FIFO full if there is no place for a frame of maximum size
202         -- Smax = 1518; 16bit words --> 1518/2 = 759
203         if(l_frame_count > 0 AND v_l_free_space <=  759)then
204             l_queue_full <= true;
205         else
206             l_queue_full <= false;
207         end if;
208         -- HIGH PRIORITY - free space computation
209         if(v_h_wr_addr >= v_h_rd_addr)then
210             v_h_free_space := QUEUE_RAM_WORDS - (v_h_wr_addr - v_h_rd_addr);
211         else
212             v_h_free_space := v_h_rd_addr - v_h_wr_addr;
213         end if;
214 
215         if(h_frame_count > 0 AND v_h_free_space <=  759)then
216             h_queue_full <= true;
217         else
218             h_queue_full <= false;
219         end if;
220     end if;
221 end process;
222 
223 -- PROCESS ---------------------------------------------------------------------------
224 -- LowPriority: handles the reception of low priority frames from the filter module.
225 -- The frame is saved in any case, but if at the end its priority is H it is dropped.
226 -- If the drop signal is received the packet is droppes also.
227 --------------------------------------------------------------------------------------
228 LowPriority: process(clk, reset)
229     variable v_infoManagerSent : boolean := false; -- TRUE when Manager sends info
230 begin
231     if(reset = '1')then
232 -- All signals are reset to the default value
233         l_frame_rx <= 0;
234         l_table_count <= 1;
235         l_wren <= '0';
236         l_wraddress <= (others => '0');
237         data <= (others => '0');
238         for j in 1 to QUEUE_DEPTH loop
239             l_start_addr(j) <= (others => '0');
240             l_end_addr(j) <= (others => '0');
241             l_dest_port(j) <= (others => '0');
242             l_frame_priority(j) <= '0';
243             l_latency(j) <= (others => '0');
244         end loop;
245 
246     elsif ( clk'event and clk = '1') then
247         data <= data_in;
248 
249         case l_state_repack is
250 -- INIT: is a sort of reset state that can be reached in case of error. It is also the
```

```
251 -- first state executed by the FSM when it is booted
252         when INIT =>
253             l_wren <= '0';
254             l_wraddress <= (others => '0');
255             l_frame_rx <= 0;
256             l_table_count <= 1;
257             v_infoManagerSent := false;
258             for j in 1 to QUEUE_DEPTH loop
259                 l_start_addr(j) <= (others => '0');
260                 l_end_addr(j) <= (others => '0');
261                 l_dest_port(j) <= (others => '0');
262                 l_frame_priority(j) <= '0';
263                 l_latency(j) <= (others => '0');
264             end loop;
265             l_state_repack <= IDLE; -- State moves to IDLE
266
267 -- IDLE: Queue waits for the start of a reception. If it`s not full and a reception
268 -- starts the FSM moves to RECEPTION otherwise it ignores the received frame.
269         when IDLE =>
270             v_infoManagerSent := false;
271
272             if(l_queue_full AND data_in_valid = '1')then
273     -- TODO: since the received frame is ignored when the queue is full an error signal
274     -- could be generated. For the moment nothing is done.
275                 l_state_repack <= IDLE;
276     -- If the FIFO is not full and a reception starts the current wr_addr is saved as
277     -- start_addr and the new state is RECEIVE.
278             elsif(data_in_valid = '1')then
279                 l_wren <= '1';
280                 l_start_addr(l_table_count) <= l_wraddress;
281                 l_state_repack <= RECEIVE;
282             else
283                 l_state_repack <= IDLE;
284             end if;
285 -- RECEIVE: received data are saved in the internal FIFO (L priority). If the drop flag
286 -- is set the frame is dropped
287         when RECEIVE =>
288     -- When the manager provides destination and priority they are saved in the
289     -- corresponding table and v_infoManagerSent is set to TRUE
290             if(destination_valid = '1')then
291                 v_infoManagerSent := true;
292                 l_dest_port(l_table_count) <= destination_in;
293                 l_frame_priority(l_table_count) <= priority;
294             end if;
295     -- data_in is saved in the FIFO and wr_addr incremented
296             if(data_in_valid = '1' and data_in_EOP = '0')then
297                 l_wraddress <= std_logic_vector(unsigned(l_wraddress) + 1);
298                 l_wren <= '1';
299                 l_state_repack <= RECEIVE;
300             elsif(data_in_EOP = '1')then
301     -- When End of Packet arrives state goes back to IDLE, and the last written address
302     -- is saved in the corresponding table. if the data from the manager have been
303     -- received the frame is valid and the counter incremented otherwise it is dropped
304                 l_wren <= '0';
305                 if(v_infoManagerSent)then
306     -- if priority is L it keeps the frame, saves the timer value and increment the table
307     -- counter. RX counter is incremented as well.
308     -- NB: the frame is saved also if it has H priority but the H FIFO is full (redundancy)
309                     if(h_frame_priority(h_table_count) = '0' OR (
310                     h_frame_priority(h_table_count) = '1' AND H_queue_full))then
311                         l_wraddress <= std_logic_vector(unsigned(l_wraddress) + 1);
312                         l_frame_rx <= l_frame_rx + 1;
313                         l_latency(l_table_count) <= tick;
314                         if(l_table_count < QUEUE_DEPTH)then
315                             l_table_count <= l_table_count + 1;
316                         else
317                             l_table_count <= 1;
318                         end if;
```

```
319                     else
320     -- Otherwise the frame is dropped
321                         l_wraddress <= l_start_addr(l_table_count);
322                     end if;
323                 else
324                     l_wraddress <= l_start_addr(l_table_count);
325                 end if;
326                 l_state_repack <= IDLE;
327                 l_end_addr(l_table_count) <= l_wraddress;
328             end if;
329         -- If at any moment the drop flag is set the frame is dropped
330             if(data_in_drop = '1')then
331                 l_wren <= '0';
332                 l_wraddress <= l_start_addr(l_table_count);
333                 l_state_repack <= IDLE;
334             end if;
335
336         when others =>
337             l_state_repack <= INIT;
338         end case;
339     end if;
340 end process;
341
342 -- PROCESS ------------------------------------------------------------------------------
343 -- HighPriority: handles the reception of high priority frames from the filter module.
344 -- The frame is saved in any case, but if at the end its priority is L it is dropped.
345 -- If the drop signal is received the packet is droppes also.
346 -- Most of this process is identical to the previoous one, refer to the commentaries of
347 -- the corresponding code for information, only different features are explained here
348 -----------------------------------------------------------------------------------------
349 HighPriority: process(clk, reset)
350     variable v_infoManagerSent : boolean := false;
351 begin
352     if(reset = '1')then
353         h_state_repack <= INIT;
354         h_frame_rx <= 0;
355         h_table_count <= 1;
356         h_wren <= '0';
357         h_wraddress <= (others => '0');
358         for j in 1 to QUEUE_DEPTH loop
359             h_start_addr(j) <= (others => '0');
360             h_end_addr(j) <= (others => '0');
361             h_dest_port(j) <= (others => '0');
362             h_frame_priority(j) <= '0';
363             h_latency(j) <= (others => '0');
364         end loop;
365
366     elsif ( clk'event and clk = '1') then
367         case h_state_repack is
368 -- INIT: reset state
369         when INIT =>
370             h_wren <= '0';
371             h_wraddress <= (others => '0');
372             h_frame_rx <= 0;
373             h_table_count <= 1;
374             v_infoManagerSent := false;
375             for j in 1 to QUEUE_DEPTH loop
376                 h_start_addr(j) <= (others => '0');
377                 h_end_addr(j) <= (others => '0');
378                 h_dest_port(j) <= (others => '0');
379                 h_frame_priority(j) <= '0';
380                 h_latency(j) <= (others => '0');
381             end loop;
382             h_state_repack <= IDLE;
383
384 -- IDLE: waits the start of a new reception, that is performed only if H FIFO
385 -- is not full
386         when IDLE =>
```

```
387             v_infoManagerSent := false;
388             if(H_queue_full AND data_in_valid = '1')then
389                 h_state_repack <= IDLE;
390             elsif(data_in_valid = '1')then
391                 h_wren <= '1';
392                 h_start_addr(h_table_count) <= h_wraddress;
393                 h_state_repack <= RECEIVE;
394             else
395                 h_state_repack <= IDLE;
396             end if;
397 
398 -- RECEIVE: reception is performed. The frame is dropped if the corresponding
399 -- flag is set or if it has L priority
400         when RECEIVE =>
401             if(destination_valid = '1')then
402                 v_infoManagerSent := true;
403                 h_dest_port(h_table_count) <= destination_in;
404                 h_frame_priority(h_table_count) <= priority;
405             end if;
406 
407             if(data_in_valid = '1' and data_in_EOP = '0')then
408                 h_wraddress <= std_logic_vector(unsigned(h_wraddress) + 1);
409                 h_wren <= '1';
410                 h_state_repack <= RECEIVE;
411             elsif(data_in_EOP = '1')then
412                 h_wren <= '0';
413                 if(v_infoManagerSent)then
414                     if(h_frame_priority(h_table_count) = '1')then
415                         h_wraddress <= std_logic_vector(unsigned(h_wraddress) + 1);
416                         h_frame_rx <= h_frame_rx + 1;
417                         h_latency(h_table_count) <= tick;
418                         if(h_table_count < QUEUE_DEPTH)then
419                             h_table_count <= h_table_count + 1;
420                         else
421                             h_table_count <= 1;
422                         end if;
423                     else
424                         h_wraddress <= h_start_addr(h_table_count);
425                     end if;
426                 else
427                     h_wraddress <= h_start_addr(h_table_count);
428                 end if;
429                 h_state_repack <= IDLE;
430                 h_end_addr(h_table_count) <= h_wraddress;
431             end if;
432 
433             if(data_in_drop = '1')then
434                 h_wren <= '0';
435                 h_wraddress <= h_start_addr(h_table_count);
436                 h_state_repack <= IDLE;
437             end if;
438 
439         when others =>
440             h_state_repack <= INIT;
441         end case;
442     end if;
443 end process;
444 
445 -- PROCESS ------------------------------------------------------------------------------
446 -- Transmit: communication with the scheduler is performed to determine when a
447 -- frame is available for forwarding and when it is scheduled for transmission
448 ---------------------------------------------------------------------------------------
449 Transmit: process(clk, reset)
450     variable tick_now       : integer := 0; -- present time
451     variable h_tick_old     : integer := 0; -- time of reception of H head of FIFO
452     variable l_tick_old     : integer := 0; -- time of reception of L head of FIFO
453 begin
454     if(reset = '1')then
```

```
455     send_priority   <= '0';
456     data_out_valid  <= '0';
457     data_out_EOP    <= '0';
458     l_rdaddress     <= (others => '0');
459     h_rdaddress     <= (others => '0');
460     l_next          <= 1;
461     h_next          <= 1;
462     destination_out_reg <= (others => '0');
463     -- destination_out "000...0" means that there is no frame ready for transmission
464     null_vector <= (others => '0');
465     state_tx <= INIT;
466
467     elsif ( clk'event and clk = '1') then
468     case state_tx is
469 -- INIT: reset state
470     when INIT =>
471         destination_out_reg <= (others => '0');
472         send_priority         <= '0';
473         data_out_valid  <= '0';
474         data_out_EOP    <= '0';
475         l_rdaddress     <= (others => '0');
476         h_rdaddress     <= (others => '0');
477         l_next          <= 1;
478         h_next          <= 1;
479         state_tx        <= IDLE;
480 -- IDLE:
481     when IDLE =>
482         data_out_EOP <= '0';
483         tick_now := conv_integer('0'&tick);
484         h_tick_old := conv_integer('0'&h_latency(h_next));
485         l_tick_old := conv_integer('0'&l_latency(l_next));
486
487     -- MAXIMUM LATENCY CONTROL ---------------------------------------------------------
488     -- If there is a frame in the H FIFO its reception time is compared with the "now"
489     -- If the difference is greater than the MAX_LATENCY it is dropped
490     -- To drop it, the Queue consider it transmitted even if it is not the case
491         if(h_frame_count > 0 AND tick_now /= h_tick_old)then
492             if(tick_now > h_tick_old )then
493                 if(tick_now - h_tick_old > MAX_LATENCY)then
494                     h_frame_tx <= h_frame_tx + 1;
495                     if(h_next < QUEUE_DEPTH)then
496                         h_next <= h_next + 1;
497                     else
498                         h_next <= 1;
499                     end if;
500                 end if;
501             else
502                 if((tick_now + 255) - h_tick_old > MAX_LATENCY)then
503                     h_frame_tx <= h_frame_tx + 1;
504                     if(h_next < QUEUE_DEPTH)then
505                         h_next <= h_next + 1;
506                     else
507                         h_next <= 1;
508                     end if;
509                 end if;
510             end if;
511         end if;
512     -- The same process is executed for the Head of Queue of the L FIFO
513         if(l_frame_count > 0 AND tick_now /= l_tick_old)then
514             if(tick_now > l_tick_old )then
515                 if(tick_now - l_tick_old > MAX_LATENCY)then
516                     l_frame_tx <= l_frame_tx + 1;
517                     if(l_next < QUEUE_DEPTH)then
518                         l_next <= l_next + 1;
519                     else
520                         l_next <= 1;
521                     end if;
522                 end if;
```

```
523             else
524                 if((tick_now + 255) - l_tick_old > MAX_LATENCY)then
525                     l_frame_tx <= l_frame_tx + 1;
526                     if(l_next < QUEUE_DEPTH)then
527                         l_next <= l_next + 1;
528                     else
529                         l_next <= 1;
530                     end if;
531                 end if;
532             end if;
533         end if;
534
535         -- TRANSMISSION ------------------------------------------------------------------
536         -- If there is a H frame ready to be sent the queue answers to the Scheduler
537         -- request specifying a H priority
538         if(h_frame_count > 0)then
539             send_priority <= '1';
540             if(request = '1')then -- Scheduler request
541         -- The MSB of the destination corresponds to the priority
542                 destination_out_reg <= '1' & h_dest_port(h_next);
543             elsif(destination_out_reg /= null_vector AND to_send = '1')then
544         -- destination_out_reg /= null_vector to be sure that a transmission is not started
545         -- if the answer to the Scheduler was not sent.
546                 h_rdaddress <= h_start_addr(h_next);
547                 state_tx <= SEND_H;
548             end if;
549
550         -- If the H queue is empty the frames are fetched from the L queue
551         elsif(l_frame_count > 0)then
552             send_priority <= '0';
553             if(request = '1')then
554                 destination_out_reg <= '0' & l_dest_port(l_next);
555             elsif(destination_out_reg /= null_vector AND to_send = '1')then
556                 l_rdaddress <= l_start_addr(l_next);
557                 state_tx <= SEND_L;
558             end if;
559         end if;
560
561 -- SEND_L: transmission of a frame from the L queue is performed
562     when SEND_L =>
563     -- Answers to Scheduler are reset
564         send_priority <= '0';
565         destination_out_reg <= (others => '0');
566         data_out_valid <= '1';
567     -- rd_addr is incremented until it reached the end_address
568         l_rdaddress <= std_logic_vector(unsigned(l_rdaddress) + 1);
569         if(l_rdaddress = l_end_addr(l_next))then
570             state_tx <= END_SEND;
571             l_frame_tx <= l_frame_tx + 1;
572     -- pointer to the next frame to be sent is updated
573             if(l_next < QUEUE_DEPTH)then
574                 l_next <= l_next + 1;
575             else
576                 l_next <= 1;
577             end if;
578         end if;
579
580 -- SEND_H: transmission of a frame from the H queue is performed
581     when SEND_H =>
582         send_priority <= '1';
583         destination_out_reg <= (others => '0');
584         data_out_valid <= '1';
585         h_rdaddress <= std_logic_vector(unsigned(h_rdaddress) + 1);
586         if(h_rdaddress = h_end_addr(h_next))then
587             state_tx <= END_SEND;
588             h_frame_tx <= h_frame_tx + 1;
589
590             if(h_next < QUEUE_DEPTH)then
```

```
591                 h_next <= h_next + 1;
592             else
593                 h_next <= 1;
594             end if;
595         end if;
596 -- END_SEND: state necessary to indicate to the destination FIFOs that the
597 -- packet has been completely transmitted
598     when END_SEND =>
599         data_out_valid <= '0';
600         data_out_EOP <= '1';
601         state_tx <= IDLE;
602
603     when others =>
604         state_tx <= INIT;
605     end case;
606
607     end if;
608 end process;
609
610 end behaviour;
```

B.2 Manager implementation

```
1  ------------------------------------------------------------------------------------------
2  -- Title         : Manager
3  -- Project       : AVIO 402
4  ------------------------------------------------------------------------------------------
5  -- File          : Manager.vhd
6  -- Author        : Davide Trentin
7  ------------------------------------------------------------------------------------------
8  -- Description  : The manager wait until an input port received a complete header and
9  --  send it to it. Depending on the VL specified by the destination address included in
10 --  the header, the manager retrieves the corresponding information about Smin and Smax
11 --  and sends them back to the filterand sends also their destinations and priority
12 --  to the corresponding queue. A control over the bandwidth allocated for each VL is
13 --  performed as well using a token bucket algorithm to perfrom the frame based traffic
14 --  policing described in the specification.
15 ------------------------------------------------------------------------------------------
16 library ieee;
17 use ieee.std_logic_1164.all;
18 use ieee.std_logic_unsigned.all;
19 use ieee.numeric_std.all;
20 -- Project Libraries
21 use work.Router_Types.all;
22
23 entity Manager is
24 port (
25     clk                 : in  std_logic;
26     reset               : in  std_logic;
27     clk_low             : in std_logic;          -- 1MHz clock
28     -- From and to Repack
29     header              : in  Mngr_ports_in ( Nbr_ports - 1 downto 0 );
30     header_valid        : in  std_logic_vector ( Nbr_ports - 1 downto 0 );
31     Smin                : out Smin_array( Nbr_ports - 1 downto 0 );
32     Smax                : out Smax_array( Nbr_ports - 1 downto 0 );
33     bad_BAG_jitter      : out std_logic_vector( Nbr_ports - 1 downto 0 );
34     Manager_out_valid   : out std_logic_vector( Nbr_ports - 1 downto 0 );
35     -- To queue
36     destination         : out Destination_array( Nbr_ports - 1 downto 0 );
37     priority            : out std_logic_vector( Nbr_ports - 1 downto 0 )
38     -- To Routing Table - TODO: connection with external routing table
39     --ram_data          : inout std_logic_vector( 26 downto 0 );
40     --ram_oe_n          : in std_logic;
41     --ram_address       : out std_logic_vector( 18 downto 0 );
42     --ram_cen_n         : out std_logic;
43     --ram_ce_n          : out std_logic_vector( 2 downto 0 );
```

```
44      --ram_adv           : out std_logic;
45      --ram_we_n          : out std_logic
46      );
47  end Manager;
48
49  architecture behaviour of Manager is
50  -- TYPES --------------------------------------------------------------------------------
51  type state_mgr_type is ( INIT_STATE, WAIT_STATE, SEND_STATE);
52  type AC_array is array (NATURAL range <>) of std_logic_vector(28 downto 0);
53  type delta_array is array (natural range <>) of std_logic_vector(10 downto 0);
54  -- SIGNALS ------------------------------------------------------------------------------
55  -- Traffic Policing
56  signal AC               : AC_array (Nbr_VLs-1 downto 0);    -- ACcount for each VL
57  signal delta_AC         : delta_array (Nbr_VLs-1 downto 0); -- Delta of the AC ramp
58  signal AC_max           : AC_array (Nbr_VLs-1 downto 0);    -- ACmax = Smax*N*(1+J/BAG)
59  signal AC_min           : AC_array (Nbr_VLs-1 downto 0);    -- ACmin = Smax * N
60                                                  --(N is the number of steps in one BAG)
61  -- Frame Filtering and Routing
62  signal array_Smin       : Smin_array( Nbr_VLs-1 downto 0 );
63  signal array_Smax       : Smax_array( Nbr_VLs-1 downto 0 );
64  signal array_Ports      : Destination_array( Nbr_VLs-1 downto 0 );
65  signal priority_table   : std_logic_vector( Nbr_VLs-1 downto 0 );
66  -- Execution flow control
67  signal frame_received   : boolean := false; -- true when a frame header has been received
68  signal VL_received      : integer := 0;            -- VL of the currently received frame
69  signal manager_state    : state_mgr_type;
70  signal null_vector      : std_logic_vector ( Nbr_ports - 1 downto 0 );
71
72  begin
73  -- PROCESS ------------------------------------------------------------------------------
74  -- Initialization: all the registers that should be retrieved from the configuration
75  -- table are here initialized; in a final version this process should initialize them
76  -- by reading the table. The considered signals are: Smin, Smax, priority, DeltaAC,
77  -- ACmax, and ACmin
78  -----------------------------------------------------------------------------------------
79      Initialization: process(clk, reset)
80      begin
81      null_vector <= (others => '0');
82      if(clk'event AND clk = '1')then
83          if(manager_state = INIT_STATE)then
84              for j in 0 to Nbr_VLs - 1 loop
85      -------------------------------------------------------------------------------------
86      -- ACcounts initialization: they include information both about BAG and Jitter, as
87      -- well as about Smax. Registers dedicated to BAG and Jitter can be eliminated, they
88      -- must be used at configuration time to assign the required value to AC_min, AC_max
89      -- and delta_AC. Smin and Smax still need to be provided in the configuration table
90      -- N = 1ms * 1Mhz is the number of clock cycles per minimum BAG = 1000
91      -------------------------------------------------------------------------------------
92      -- NB: in this case all parameters, for all VLs, have the same value to reduce code,
93      -- but for the simulation various possible values have been tested
94                  -- Smax = 1538, Jmax/BAG = 1/5, BAG = 1ms
95                  delta_AC(j) <= "11000000010"; -- 1538 = Smax i the amplitude of each step
96                  AC_max(j) <= "00000000111000010100101100000"; -- 1845600
97                  AC_min(j) <= "00000000101110111011111010000"; -- 1538000
98
99                  -- Another example: Smax = 1538 and BAG = 128ms
100                 --        delta_AC(j) <= "11000000010";        -- 1538*128*1000
101                 --        AC_max(j) <= "10001100110011101110000000000";
102                 --        AC_min(j) <= "01011101110111110100000000000";
103             end loop;
104
105     -------------------------------------------------------------------------------------
106     -- Destination and Priority initialization
107     -------------------------------------------------------------------------------------
108             priority_table <= "00000000000001111100"; -- for 20 VLs
109
110             --        5 ports -- various possible destinations for testing
111             array_Ports( 0 ) <= "00001";
```

```
112             array_Ports( 1 ) <= "10100";
113             array_Ports( 2 ) <= "11111";
114             array_Ports( 3 ) <= "11000";
115             array_Ports( 4 ) <= "00011";
116             array_Ports( 5 ) <= "01101";
117             array_Ports( 6 ) <= "11101";
118             array_Ports( 7 ) <= "00010";
119             array_Ports( 8 ) <= "01000";
120             array_Ports( 9 ) <= "11000";
121             array_Ports( 10 ) <= "10000";
122             array_Ports( 11 ) <= "10101";
123             array_Ports( 12 ) <= "01100";
124             array_Ports( 13 ) <= "00110";
125             array_Ports( 14 ) <= "01010";
126             array_Ports( 15 ) <= "11111";
127             array_Ports( 16 ) <= "11000";
128             array_Ports( 17 ) <= "01110";
129             array_Ports( 18 ) <= "00100";
130             array_Ports( 19 ) <= "00001";
131
132     ---------------------------------------------------------------------------------------
133     -- Smin and Smax
134     ---------------------------------------------------------------------------------------
135             for j in 0 to Nbr_VLs -1 loop
136     -- Like for ACs, all VLs have the same Smin and Smax in this case only to reduce code
137     -- size, but various values have been used for simulations
138                 array_Smin(j) <= "00001010100";   -- 84
139                 array_Smax(j) <= "11000000010";        -- 1538
140             end loop;
141
142         end if;
143     end if;
144 end process;
145
146 -- PROCESS ------------------------------------------------------------------------------
147 -- MainManager: it's the process that handles the flow of the FSM that controls this
148 -- module, it also set the values of the signals used to communicate with the Filter
149 -- and with the Queue
150 ------------------------------------------------------------------------------------------
151 MainManager: process(clk, reset)
152 variable cpt_round_robin    : integer := 0;
153 variable current_VL         : integer := 0;
154 variable index              : Integer := 0;
155 variable last_port          : integer := 0;
156
157 begin
158     if(reset = '1') then
159         Manager_out_valid <= ( others => '0' );
160         bad_BAG_jitter <= ( others => '0' );
161         for j in 0 to Nbr_ports - 1 loop
162             destination( j ) <= ( others => '0' );
163             Smin( j ) <= ( others => '0' );
164             Smax( j ) <= ( others => '0' );
165         end loop;
166         frame_received <= false;
167         VL_received <= 0;
168         priority <= (others => '0');
169         index := 0;
170         last_port := -1;
171         current_VL := 0;
172         manager_state <= INIT_STATE;
173
174     elsif ( clk'event and clk = '1') then
175
176         case manager_state is
177 -- INIT_STATE: all the variables and signals are initialized to the required values
178         when INIT_STATE =>
179             Manager_out_valid <= ( others => '0' );
```

```
180             bad_BAG_jitter <= ( others => '0' );
181             for j in 0 to Nbr_ports - 1 loop
182                 destination( j ) <= ( others => '0' );
183                 Smin( j ) <= ( others => '0' );
184                 Smax( j ) <= ( others => '0' );
185             end loop;
186             frame_received <= false;
187             VL_received <= 0;
188             index := 0;
189             last_port := -1;
190             current_VL := 0;
191             manager_state <= WAIT_STATE;
192
193 -- WAIT_STATE: The Manager waits for a Filter module to ask its intervention by
194 -- sending a header to analyze
195         when WAIT_STATE =>
196             frame_received <= false;
197             bad_BAG_jitter <= ( others => '0' );
198             Manager_out_valid <= ( others => '0' );
199             -- If one of the header_valid signal is high the analysis routine is started
200             -- more than one Filter can send the request
201             if( header_valid /= 0 ) then
202                 manager_state <= SEND_STATE;
203             end if;
204
205 -- SEND_STATE: the received header is studied to determine the corresponding VL and
206 -- send the relative information to the Filter that is receiveing the frame
207         when SEND_STATE =>
208     -- header_valid signals from all ports are polled to check who's asking
209     -- the manager intervention. A round robin algorithm is used to poll each
210     -- time starting from the port that follows the one treated in the last cycle
211             for j in 0 to Nbr_ports - 1 loop
212                 index := ( last_port + j );
213                 if(index > Nbr_ports - 1)then
214                     index := index - Nbr_ports;
215                 end if;
216
217     -- When the first port asking for the manager intervention is found the loop
218     -- is broken and the index value used for the rest of the analysis
219                 if(index /= -1)then
220                     if header_valid(index) = '1' then
221                         EXIT;
222                     end if;
223                 end if;
224             end loop;
225
226     -- If only one port has set header_valid = '1' it doesn't have the time to put
227     -- it to '0' before a new SEND is started. if this is the case traffic policing
228     -- would recognize this as a new frame on the same VL and signal a bad BAG.
229     -- To avoid this if the port is the same as before nothing is done and the Round
230     -- Robin set back to port 0
231             if(index = last_port - 1)then
232                 last_port := -1;
233             else
234     -- Retrieve current VL from the current header
235                 current_VL := conv_integer(header(index)( 79 downto 64 ) );
236                 VL_received <= current_VL;
237
238 -- TODO -----------------------------------------------------------------------------
239 -- A more refined algorithm must be implemented to ensure that only acceptable VL are
240 -- considered valid for now this is done only for consecutive VLs.
241 -----------------------------------------------------------------------------------
242                 if((current_VL < Nbr_VLs) and (current_VL >= 0)) then
243
244     -- TRAFFIC POLICING: if the AC for the current VL is lower than Smax the bandwidth
245     -- alloc gap is not respected and the frame must be dropped,otherwise it is valid
246                     if(AC(current_VL) < AC_min(current_VL))then
247                         bad_BAG_jitter(index) <= '1';
```

```
248                         frame_received <= false;
249                     else
250                         bad_BAG_jitter(index) <= '0';
251                         frame_received <= true;
252                     end if;
253
254     -- The outputs towards the input port that sent the header are updated
255                     priority(index)       <= priority_table( current_VL );
256                     Smin(index) <= array_Smin( current_VL );
257                     Smax(index) <= array_Smax( current_VL );
258                     destination(index)  <= array_Ports( current_VL );
259                     Manager_out_valid(index)  <= '1';
260                 else
261     -- The VL is not acceptable and the frame must be dropped
262                     bad_BAG_jitter(index) <= '1';
263                 end if;
264
265     -- The port next to the one treated in this cycle is saved so that in the next
266     -- cycle the polling will start from there, assuring a Round Robin polling of
267     -- the request of the input ports
268                 last_port := index +1;
269             end if;
270
271             manager_state <= WAIT_STATE;
272
273         when others =>
274             manager_state <= INIT_STATE;
275             end case;
276     end if;
277 end process;
278
279 --- PROCESS ---------------------------------------------------------------------------
280 -- TrafficPolicing: it handles the ACcounts for each VL, incrementing them constantly
281 -- and decreasing them when a frame from the corresponding Vl is received
282 ---------------------------------------------------------------------------------------
283 TrafficPolicing: process(clk, reset)
284 begin
285     if(reset = '1')then
286     -- All the ACcounts are set to their maximum value = Smax * N * (1 + J/BAG)
287         for j in 0 to Nbr_VLs - 1 loop
288             AC(j) <= AC_max(j);
289         end loop;
290
291     elsif(clk'event AND clk = '1')then
292         case manager_state is
293 -- INIT_STATE: all the AC are reset to their maximum value
294         when INIT_STATE =>
295             for j in 0 to Nbr_VLs - 1 loop
296                 AC(j) <= AC_max(j);
297             end loop;
298
299 -- OTHERS: in any other case they are incremented continuously, or decremented if a
300 -- valid frame is received on that VL
301         when others =>
302     -- If a frame is received the corresponding AC is decremented by the relative AC_min
303     -- AC > AC_min is not verified because that's already done in the main process
304             if(frame_received)then  -- the main process detected a valid frame
305                 AC(VL_received) <= AC(VL_received) - AC_min(VL_received);
306             end if;
307
308             if(clk_low = '1')then
309                 for j in 0 to Nbr_VLs - 1 loop
310                     if(AC(j) < AC_max(j))then
311                         AC(j) <= AC(j) + delta_AC(j);
312                     end if;
313
314                 end loop;
315             end if;
```

```
316             end case;
317
318         end if;
319 end process;
320
321 end behaviour;
```

B.3 Scheduler implementation

```
1   -------------------------------------------------------------------------------------
2   -- Title        : Scheduler
3   -- Project      : AVIO 402
4   -------------------------------------------------------------------------------------
5   -- File         : Scheduler.vhd
6   -- Author       : Davide Trentin
7   -------------------------------------------------------------------------------------
8   -- Description  : It controls which output ports have free space in their FIFO to receive
9   --  new frames for transmission, and continuously polls the Queues to determine if any of
10  --  them has a frame ready to be forwarded to the available ports. If more than one queue
11  --  can send data, a RoundRobin algorithm is used to decide which one should transmit.
12  -------------------------------------------------------------------------------------
13  library ieee;
14  use ieee.std_logic_1164.all;
15  use ieee.std_logic_unsigned.all;
16  use ieee.numeric_std.all;
17  -- Project Libraries
18  use work.Router_Types.all;
19
20
21  entity Scheduler is
22  port(
23      -- Inputs
24      clk             : in  std_logic;
25      reset           : in  std_logic;
26      response        : in  Reponse_queues (Nbr_ports - 1  downto 0); -- From Queues
27      port_available  : in  std_logic_vector (Nbr_ports-1 downto 0);  -- From FIFOs
28      -- Outputs
29      request         : out std_logic_vector (Nbr_ports-1 downto 0); -- To Queues
30      to_send         : out std_logic_vector (Nbr_ports-1 downto 0); -- To Queues
31      selection       : out slctn (Nbr_ports-1 downto 0 );  -- Crossbar configuration
32      be_ready        : out std_logic_vector(Nbr_ports - 1 downto 0) -- To FIFOs
33     );
34  end Scheduler;
35
36  architecture behaviour of Scheduler is
37
38  -- TYPES ----------------------------------------------------------------------------
39  type state_type is (idle, rep_queue, R_Robin_H, R_Robin_L ,Send_Queue_OK,Etat_tampon);
40  -- SIGNALS --------------------------------------------------------------------------
41  signal state                : state_type:= idle ;
42  signal sent                 : std_logic;
43  signal response_reg         : Reponse_queues (Nbr_ports - 1 downto 0);
44  signal request_reg          : std_logic_vector(Nbr_ports-1 downto 0) := (others => '0');
45  signal to_send_reg          : std_logic_vector(Nbr_ports-1 downto 0) := (others => '0');
46  signal port_available_reg   : std_logic_vector (Nbr_ports-1 downto 0);
47  signal waitBool             : boolean := false;
48  signal Selection_tampon     :slctn (Nbr_ports-1 downto 0);
49  signal null_vector          : std_logic_vector(Nbr_ports -1 downto 0) := (others => '0');
50  signal start_queue          : integer; -- Used for Round Robin
51
52  begin
53  -- COMBINATORIAL---------------------------------------------------------------------
54  to_send<=to_send_reg;
55
56  -- PROCESS --------------------------------------------------------------------------
57  -- Scheduling: it handles the polling of the queues, the analysis of their response, and
```

```
58  -- it determined which ones should start a transmission
59  ---------------------------------------------------------------------------------------
60  Scheduling: process(reset,clk)
61  -- VARIABLES --------------------------------------------------------------------------
62  variable queue_send         : std_logic_vector(Nbr_ports -1 downto 0);
63  variable v_FIFO_available   : std_logic_vector(Nbr_ports -1 downto 0);
64  variable send_possible      : boolean;
65  variable index              : integer;        -- Used for Round Robin
66  variable select_buffer      : slctn (Nbr_ports-1 downto 0);
67
68  begin
69      if (reset='1')then
70      -- All the outputs and control signals are reset
71          waitBool <= false;
72          state <= IDLE;
73          request_reg(Nbr_ports-1 downto 0) <=(others =>'0');
74          to_send_reg(Nbr_ports-1 downto 0) <=(others =>'0');
75          queue_send := (others => '0');
76          start_queue <= 0;
77          for j in 0 to Nbr_ports - 1 loop
78              select_buffer(j) := -1;
79              be_ready(j) <= '0';
80          end loop;
81          request <= (others => '0');
82
83      elsif (clk='1' and clk'event) then
84      case state is
85  -- IDLE: It checks which outputs are available
86      when IDLE =>
87          waitBool <= false; -- required because REP_QUEUE needs to last 2 cycles
88          queue_send := (others => '0');
89      -- if there is at least one available FIFO the scheduling routine is initialized
90      -- available ports are saved and the request signal sent to the queues
91          if(port_available /= null_vector) then
92              v_FIFO_available := port_available;
93              request <= (others => '1');
94              for j in 0 to Nbr_ports - 1 loop
95                  be_ready(j) <= '0';
96              end loop;
97              state <= REP_QUEUE;
98          end if;
99
100 -- REP_QUEUE: It waits for the queues' response. since they come after 2 clock cycles
101 -- there is one wait cycle
102     when rep_queue  =>
103         waitBool <= not(waitBool);
104         request<=  (others => '0');
105     -- After one cycle the answer of the queue is read
106         if(waitBool) then
107             response_reg <= response;
108             state <= R_ROBIN_H;
109         end if;
110
111 -- R_ROBIN_H: the round robin algorithm is performed to determine which queues can send
112 -- their packet because their destinations are available.
113     when R_ROBIN_H   =>
114     -- The queues destinations are analysed to determine which ones have high priority
115         for j in 0 to Nbr_ports - 1 loop
116     -- Round Robin: the polling starts from the queue immediatelly after the last one
117             index := start_queue + j;
118             if(index > Nbr_ports - 1)then
119                 index := index - Nbr_ports;
120             end if;
121     -- If the considered queue has a high priority frame and a non-zero destination
122             if(response_reg(index)(Nbr_ports) =  '1' AND
123             response_reg(index)(Nbr_ports-1 downto 0) /= null_vector)then
124     -- send_possible is set back to false if the required outputs are not available
125                 send_possible := true;
```

```
126                 for i in 0 to Nbr_ports - 1 loop
127                     -- If the required FIFO is not available the frame will not be sent
128                     if(response_reg(index)(i) = '1' AND v_FIFO_available(i) = '0')then
129                         send_possible := false;
130                     end if;
131                 end loop;
132 
133     -- If the queue can start a tranmsission the "send" vector is updated to force this transmission
134     -- and the output ports that will be used are now considered unavailable
135                 if(send_possible)then
136                     queue_send(index) := '1';
137                     for i in 0 to Nbr_ports - 1 loop
138                         if(response_reg(index)(i) = '1')then
139                             v_FIFO_available(i) := '0';
140                             select_buffer(i) := index;
141                             be_ready(i) <= '1'; -- it tells to the corresponding FIFO that it is
142                                                 -- going to receive a packet
143                         end if;
144                     end loop;
145                 end if;
146             end if;
147         end loop;
148 
149         state <= R_ROBIN_L;
150 -- R_Robin_L: The remaining queues, the ones with low priority, are now considered
151 -- The same routine used for high priority queues is adopted here
152     when R_ROBIN_L =>
153         for j in 0 to Nbr_ports - 1 loop
154             index := start_queue + j;
155             if(index > Nbr_ports - 1)then
156                 index := index - Nbr_ports;
157             end if;
158 
159             if(response_reg(index)(Nbr_ports) =  '0' AND
160             response_reg(index)(Nbr_ports-1 downto 0) /= null_vector)then
161                 send_possible := true;
162                 for i in 0 to Nbr_ports - 1 loop
163                     if(response_reg(index)(i) = '1' AND v_FIFO_available(i) = '0')then
164                         send_possible := false;
165                     end if;
166                 end loop;
167 
168                 -- The outputs used by high priority queues are not available anymore
169                 if(send_possible)then
170                     queue_send(index) := '1';
171                     for i in 0 to Nbr_ports - 1 loop
172                         if(response_reg(index)(i) = '1')then
173                             v_FIFO_available(i) := '0';
174                             select_buffer(i) := index;
175                             be_ready(i) <= '1';
176                         end if;
177                     end loop;
178                 end if;
179             end if;
180         end loop;
181 
182         state <= SEND_QUEUE_OK;
183 
184 -- SEND_QUEUE_OK: the pointer to the current queue is incremented for the round robin
185     when SEND_QUEUE_OK =>
186     -- When at least one frame is forwarded the round robin is incremented
187         if(queue_send /= null_vector)then
188             if(start_queue = Nbr_ports -1 )then
189                 start_queue <= 0;
190             else
191                 start_queue <= start_queue + 1;
192             end if;
193         end if;
```

```
194
195     -- The to_send output is updated with the result of the scheduling
196         to_send_reg <= queue_send;
197         port_available_reg<=port_available ;
198         state<=IDLE;
199     when others =>
200         state <= IDLE;
201     end case;
202     end if;
203
204     -- The configuration signal for the crossbar is always updated depending on the
205     -- scheduling results
206     selection <= select_buffer;
207
208 end process Scheduling;
209
210 end behaviour;
```

B.4 Filter implementation

```
1  ------------------------------------------------------------------------------------------
2  -- Title       : Filter
3  -- Project     : AVIO 402
4  ------------------------------------------------------------------------------------------
5  -- File        : Filter.vhd
6  -- Author      : Davide Trentin
7  ------------------------------------------------------------------------------------------
8  -- Description  : It receives the packets with 8bit words and composes the 16bit words
9  -- that must be saved in the Queue. It extracts the header and passes it to the manager
10 -- for processing. It filters frames that do not respect size, format, or CRC.
11 ------------------------------------------------------------------------------------------
12 library IEEE;
13 use ieee.std_logic_1164.all;
14 use ieee.numeric_std.all;
15 use ieee.std_logic_unsigned.all;
16 -- Project Libraries
17 use work.Router_Types.all;
18
19 entity Filter is
20 port (
21     -- Inputs
22     clk                 : in std_logic;
23     reset               : in std_logic;
24     port_in             : in std_logic_vector ((port_length - 1) downto 0);
25     port_in_valid       : in std_logic;
26     port_in_good_frame  : in std_logic;
27     port_in_bad_frame   : in std_logic;
28     bad_BAG_jitter      : in std_logic; -- From Manager to drop packet
29     Smin                : in std_logic_vector(10 downto 0);
30     Smax                : in std_logic_vector(10 downto 0);
31     Manager_in_valid    : in std_logic;
32     -- Outputs
33     data_out            : out std_logic_vector ((port_in_length - 1) downto 0); -- To Queue
34     data_out_valid      : out std_logic;
35     data_out_drop       : out std_logic;
36     data_out_EOP        : out std_logic;
37     header              : out std_logic_vector (111 downto 0);
38     header_valid        : out std_logic
39     );
40 end Filter;
41
42 architecture behaviour of Filter is
43 -- COMPONENT ------------------------------------------------------------------------
44 component CRCgenerator is
45 port ( data_in : in std_logic_vector (7 downto 0);
46     crc_en , rst, clk : in std_logic;
```

```
47      crc_out : out std_logic_vector (31 downto 0));
48  end component;
49  -- CONSTANTS ----------------------------------------------------------------------------
50  constant buffer_size : integer := 2; -- 16 bit words passed to the Queue
51  constant wordEmpty: std_logic_vector((port_in_length/2 - 1) downto 0):=(others => '0');
52  --TYPES-----------------------------------------------------------------------------------
53  type CRC_buffer_type is array (integer range <>) of std_logic_vector(31 downto 0);
54  type data_in_buffer_type is array (integer range <>) of std_logic_vector(7 downto 0);
55  type repack_state is (idle, receive, EOP, drop1, dropn, iGAP);
56  --SIGNALS---------------------------------------------------------------------------------
57  signal CRC_buffer       : CRC_buffer_type(buffer_size-1 downto 0);
58  signal data_in_buffer   : data_in_buffer_type(3 downto 0);
59  -- Input register
60  signal port_in_valid_reg        : std_logic := '0';
61  signal port_in_good_frame_reg   : std_logic := '0';
62  signal port_in_bad_frame_reg    : std_logic := '0';
63  signal bad_BAG_jitter_reg       : std_logic := '0';
64  signal Smin_reg                 : std_logic_vector(10 downto 0) := (others => '0');
65  signal Smax_reg                 : std_logic_vector(10 downto 0) := (others => '0');
66  signal Manager_in_valid_reg     : std_logic := '0';
67  -- Output registers
68  signal data_out_reg         : std_logic_vector ((port_in_length - 1) downto 0);
69  signal data_out_valid_reg   : std_logic;
70  signal data_out_drop_reg    : std_logic;
71  signal data_out_EOP_reg     : std_logic;
72  signal header_reg           : std_logic_vector (111 downto 0);
73  signal header_valid_reg     : std_logic;
74  -- Others
75  signal CRC              : std_logic_vector(31 downto 0);
76  signal CRC_received     : std_logic_vector(31 downto 0);
77  signal bufferIn1        : std_logic_vector((port_in_length/2 - 1) downto 0);
78  signal bufferIn2        : std_logic_vector((port_in_length/2 - 1) downto 0);
79  signal completeWord     : boolean := false;
80  signal t_state : repack_state   := idle;
81  -- CRC Module
82  signal CRC_in       : std_logic_vector(7 downto 0);
83  signal CRC_en       : std_logic;
84  signal CRC_reset    : std_logic;
85  signal new_CRC      : std_logic_vector(31 downto 0);
86
87  begin
88  CRC_gen: CRCgenerator
89  port map( data_in => CRC_in,
90      crc_en => CRC_en,
91      rst => CRC_reset,
92      clk => clk,
93      crc_out => new_CRC
94  );
95
96  -- COMBINATORIAL -------------------------------------------------------------------------
97      data_out        <= data_out_reg;
98      data_out_valid  <= data_out_valid_reg;
99      data_out_drop   <= data_out_drop_reg;
100     data_out_EOP    <= data_out_EOP_reg;
101     header          <= header_reg;
102     header_valid    <= header_valid_reg;
103     CRC_received    <= data_in_buffer(3)&data_in_buffer(2)
104                         &data_in_buffer(1)&data_in_buffer(0);
105
106 -- PROCESS -------------------------------------------------------------------------------
107 -- RepackProc: it's the main process of the Filter, it handles reception and output
108 -- generation. Filtering is included in this process as well
109 -----------------------------------------------------------------------------------------
110 Repack_proc: process(reset, clk)
111 -- Variables
112 variable drop           : boolean := false;
113 variable cpt_InterGAP   : integer range 0 to 12   := 0;
114 variable cpt_header     : integer range 0 to 112  := 112;
```

```
115 variable cpt_frameSize  : integer range 0 to 1540 := 0;
116 variable manInfo        : boolean := false; -- Manager answered
117
118 begin
119 if(reset = '1') then
120     t_state <= idle;
121     data_out_valid_reg  <= '0';
122     data_out_drop_reg   <= '0';
123     data_out_EOP_reg    <= '0';
124     data_out_reg        <= (others => '0');
125     header_reg          <= (others => '0');
126     header_valid_reg    <= '0';
127     cpt_InterGAP        := 0;
128     cpt_header          := 112;
129     cpt_frameSize       := 0;
130     completeWord        <= false;
131     manInfo             := false;
132     drop                := false;
133 elsif(clk'event AND clk='1')then
134         case t_state is
135 -- IDLE: all the signal are reset and the module waits for the start of a new reception
136 -- when the valid input is set high the reception routine starts
137     when idle =>
138     -- All the signal used for the reception are reinitialized to be ready for a new one
139         data_out_valid_reg  <= '0';
140         data_out_drop_reg   <= '0';
141         data_out_EOP_reg    <= '0';
142         data_out_reg        <= (others => '0');
143         header_reg          <= (others => '0');
144         header_valid_reg    <= '0';
145         cpt_InterGAP        := 0;
146         cpt_header          := 112;
147         completeWord        <= false;
148         manInfo             := false;
149         drop                := false;
150     -- If a receptions starts (port_in_valid = '1') move to reception state
151         if((port_in_valid_reg = '1') AND (port_in_good_frame_reg = '0') AND
152         (port_in_bad_frame_reg = '0')) then
153             t_state <= receive;
154     -- First byte is saved in the 8 LSB of the header
155             for j in 0 to 7 loop
156                 header_reg(j + cpt_header - 8) <= bufferIn1(j);
157             end loop;
158             cpt_header := cpt_header - 8;
159             completeWord <= true;   -- Used to determine if the LSB are received
160     -- If other signal than port_in_valid are set high an error occurred
161     -- and the frame is dropped
162             elsif((port_in_valid_reg = '1') AND
163             ((port_in_good_frame_reg = '1') OR (port_in_bad_frame_reg = '1')))then
164                 t_state <= dropn;
165             end if;
166 -- RECEIVE: received data are saved in the queue and the manager is composed and passed
167 -- to the Manager. When the EoP is received the state changes
168     when receive =>
169         data_out_valid_reg <= '0';
170     -- When the good_frame signal is received it means that the EoP has been reached
171     -- We move to the EOP state. If rhe frame has an odd number of bytes, the last
172     -- 8 bits of the 16-bit word are set to 0
173         if(port_in_valid_reg = '0' AND
174         (port_in_good_frame_reg = '1') AND (port_in_bad_frame_reg = '0')) then
175             if(completeWord) then
176                 data_out_valid_reg <= '1';
177                 data_out_reg <=  bufferIn2 & wordEmpty;
178             end if;
179             t_state <= EOP;
180             drop := drop or not(manInfo); -- drop keeps track of the possible
181                     -- error that can occur, it the Manager still has not answered now
182                     -- the frame is dropped because the system must be ready for
```

```
183                     -- a new reception
184      -- Frame is dropped because the bad_frame signal is set high
185          elsif((port_in_valid_reg = '1') AND
186          ((port_in_good_frame_reg = '1') OR (port_in_bad_frame_reg = '1')))then
187              t_state <= dropn;
188              data_out_drop_reg <= '1';
189      -- Frame is dropped if a non acceptable situation occurs
190          elsif((port_in_valid_reg = '0') and not((port_in_good_frame_reg = '1') and (port_in_bad_frame_reg = '0')))then
191              t_state <= drop1;
192              data_out_drop_reg <= '1';
193      -- Otherwise it means that the reeption is still on progress
194          else
195          -- Each 2 bytes received the 16bit word is saved in the Queue
196              if(completeWord) then
197                  data_out_reg <= bufferIn2 & bufferIn1;
198                  data_out_valid_reg <= '1';
199              end if;
200              completeWord <= not(completeWord);
201              if(cpt_header > 0) then -- for header composition
202                  for j in 0 to 7 loop
203                      header_reg(j + cpt_header - 8) <= bufferIn1(j);
204                  end loop;
205              cpt_header := cpt_header - 8;
206              end if;
207          end if;
208  -- EOP: the end of the packet has been reached, the Filter completed the filtering and drop
209  -- erroneous frames.
210      when EOP =>
211          data_out_valid_reg <= '0';
212      -- all the filtering controls are executed and the results is saved in drop
213          drop := drop OR (CRC_buffer(1) /= CRC_received) -- CRC control
214              OR (header_reg(20 downto 16) /= "00000") -- fixed structure fields
215              OR (header_reg(63 downto 40) /= "000000100000000000000000")
216              OR (header_reg(111 downto 80) /= "00000011000000000000000000000000") -- custom
217              OR (cpt_frameSize < (conv_integer('0'&Smin_reg) - 22)) -- Smin
218              OR (cpt_frameSize > (conv_integer('0'&Smax_reg) - 21)) -- Smax
219              OR (cpt_frameSize < 63) OR (cpt_frameSize > 1538); -- Lmin and Lmax
220  -- TODO: no control over the fixed structure fields of the frame is executed since
221  -- these fields are not defined at the moment
222          if(not(drop)) then
223              t_state <= idle;
224              data_out_EOP_reg <= '1';
225          else
226              t_state <= drop1;
227              data_out_drop_reg <= '1';
228          end if;
229  -- DROP1: currently received frame is dropped and the filter goes immediately
230  -- back to IDLE
231      when drop1 =>
232          t_state <= idle;
233          data_out_valid_reg <= '0';
234          data_out_EOP_reg <= '1';
235  -- DROPN: same as before but this time this state lasts as long as there is a reception
236  -- in progress. Ita waits for its end before going back to IDLE
237      when dropn =>
238          data_out_drop_reg <= '1';
239          if(port_in_valid_reg = '0')then
240              t_state <= idle;
241              data_out_EOP_reg <= '1';
242          end if;
243          data_out_valid_reg <= '0';
244  -- OTHERS: error
245      when others =>
246          t_state <= idle;
247      end case;
248
249  -- When the Manager sends the required information they are saved
250      if((Manager_in_valid_reg = '1') AND (t_state = receive))then
```

```
251         manInfo := true;
252         drop := drop or (bad_BAG_jitter_reg = '1');
253     end if;
254
255     -- When the header is ready it is sent to the Manager
256     -- the header_valid signal remains high until the manager answers
257     if((cpt_header <= 0) AND NOT(manInfo) AND (t_state = receive))then
258         header_valid_reg <= '1';
259     else
260         header_valid_reg <= '0';
261     end if;
262
263     -- The number of bytes of the packet are counted to perform size control
264     if((port_in_valid_reg = '1')) then
265         cpt_frameSize := cpt_frameSize + 1;
266     end if;
267     -- When the state is IDLE the counter is reset to be ready for a new reception
268     if(t_state = idle)then
269         cpt_frameSize := 0;
270     end if;
271 end if;
272 end process Repack_proc;
273
274 -- PROCESS ---------------------------------------------------------------------------
275 -- Reg: all the inputs are registered,
276 -- most of them are used when they are not available anymore
277 ---------------------------------------------------------------------------------------
278 Reg: process(reset, clk)
279 begin
280     if(reset = '1')then
281         bufferIn1              <= (others =>'0');
282         bufferIn2              <= (others =>'0');
283         port_in_valid_reg      <= '0';
284         port_in_good_frame_reg <= '0';
285         port_in_bad_frame_reg  <= '0';
286         bad_BAG_jitter_reg     <= '0';
287         Smin_reg               <= (others => '0');
288         Smax_reg               <= (others => '0');
289         Manager_in_valid_reg   <= '0';
290     elsif(rising_edge(clk))then
291         bufferIn1              <= port_in;
292         bufferIn2              <= bufferIn1;
293         port_in_valid_reg      <= port_in_valid;
294         port_in_good_frame_reg <= port_in_good_frame;
295         port_in_bad_frame_reg  <= port_in_bad_frame;
296         bad_BAG_jitter_reg     <= bad_BAG_jitter;
297         Smin_reg               <= Smin;
298         Smax_reg               <= Smax;
299         Manager_in_valid_reg   <= Manager_in_valid;
300     end if;
301 end process Reg;
302
303 -- PROCESS ---------------------------------------------------------------------------
304 -- CRC_proc: it computed the CRC32 of the received frame one byte at the time
305 -- Since the filter can't know when the frame is going to end, the last two computed
306 -- values of the CRC are stored in a buffer because the second last computed one
307 -- corresponds to the CRC of the frame before the CRC field is received
308 ---------------------------------------------------------------------------------------
309 CRC_proc: process(reset, clk)
310 begin
311     if(reset = '1') then
312         CRC_reset <= '1';
313         CRC_en <= '0';
314         CRC_in <= (others => '0');
315         for j in 0 to buffer_size-1 loop
316             CRC_buffer (j) <= (others => '1');
317         end loop;
318         CRC <= (others => '1');
```

```
319     elsif(clk'event AND clk = '1')then
320         CRC_en <= '1';
321     -- when a byte is received it is passed to the CRC module
322     -- previously computed CRC is saved in the buffer
323         if(port_in_valid = '1') then
324             CRC_in <= port_in;
325             CRC <= new_CRC;
326             CRC_reset <= '0';
327             CRC_buffer(0) <= CRC;
328             for j in 1 to buffer_size-1 loop
329                 CRC_buffer(j) <= CRC_buffer(j-1);
330             end loop;
331     -- if the packet is dropped or the reception is completed the CRC is reset
332         elsif(t_state = idle OR t_state = drop1 OR t_state = dropn)then
333             CRC <= (others => '1');
334             CRC_reset <= '1';
335             for j in 0 to buffer_size-1 loop
336                 CRC_buffer(j) <= (others => '1');
337             end loop;
338         end if;
339     end if;
340 end process;
341
342 -- PROCESS --------------------------------------------------------------------------------
343 -- CRC_received_proc: it saves the last 4 received bytes, that at the end of the frame
344 -- correspond to the received CRC
345 ------------------------------------------------------------------------------------------
346 CRC_received_proc: process(reset, clk)
347 begin
348     if(reset = '1') then
349         for j in 0 to 3 loop
350             data_in_buffer(j) <= (others => '0');
351         end loop;
352     elsif(clk'event AND clk = '1')then
353         if(port_in_valid = '1') then
354             data_in_buffer(0) <= port_in;
355             for j in 1 to 3 loop
356                 data_in_buffer(j) <= data_in_buffer(j-1);
357             end loop;
358         end if;
359     end if;
360 end process;
361
362 end behaviour;
```

ANNEXE C

SWITCH FABRIC TEST VERIFICATION

C.1 Switch Fabric test cases

In Chapter 5.4, the most important results deriving from the switch fabric simulation have been briefly presented. A detailed description of the most important test cases used for the system verification is provided in this appendix. The described scenarios represent only the verification of the most significant functionalities of the switch, to verify that AFDX specific features are respected even for critical situations where the system is subjected to stressful conditions. After the presentation of all the testbenches, some waveforms are presented to show the corresponding results.

For test purposes, and to obtain images easier to read, the output FIFOs have been configured so that they start a transmission as soon as a frame is completely saved in them. UDP and IP fields of the used frames contain dummy bytes since the switch fabric does not analyze their content in any case. In every test where multiple frames are generated, payloads and lengths of these packets are different from each other to recreate a generic situation.

C.1.1 Test test case 1

Objective: Measure the technological latency for a frame with minimum size

Description: The VL of the input packet is set to 1 and this VL has been configured to have only the `output_port_0` as destination. The packet is 64 bytes long, which is the smallest size allowed by the ARINC specification. In Figure C.1, the most significant signals are shown in the waveforms that represent the routing process, and it can be observed that the frame arriving at `input_port_0` is correctly routed only to `output_port_0`. The last byte of the packet is received at 1530ns and it leaves the system after 2240ns, that consequently is the technological latency.

The same waveforms show the internal behaviour of the system. As soon as the header is received, the *Filter* advises the *Manager* that it can be processed setting the `header_valid` signal to '1', and waits for its answer. When `manager_answer` is '1' the values of S_{min} and S_{max} are saved by the *Filter*, which also puts `header_valid` back to '0', and the destination and priority are saved by the *Queue*. While it is extracting the header, the *Filetr* is also saving the incoming bytes in the *Queue* (the related signals are `data_to_queue` and

`write_to_queue`. As soon as the frame is completely saved in the input buffer, if it is not dropped (as is the case here), the *Queue* is ready to answer to the *Scheduler*, that continuously polls them. The polling signal is `scheduler_request`, while `queue_answer` has the form `"priority & destination"` when a frame is ready to be forwarded, and it is an array of zeros otherwise. When the *Scheduler* receives the destinations of the frame saved in the buffer, it prepares the configuration array for the *Crossbar* and initiates the transmission by asserting the `scheduler_force_send` signal. The frame is forwarded 16 bits at a time to the output *FIFO* using the `data_to_FIFO` and `write_to_FIFO` signals. When the packet is completely saved in it, the FIFO begins its transmission, and the output can be observed on the external signal `data_out_0` since it is only forwarded to that port.

The technological latency is composed by the time taken to save the frame in the output *FIFO*, plus the time required to transmit it, plus a variable scheduling component; since the *Scheduler* polls the *Queues* each 7 clock cycles, thus there is a 140ns potential difference in the technological latency from case to case, depending on when the last byte of the frame is saved in the buffer compared to when the *Scheduler* last checked it. 2240ns corresponds to 112 clock cycles, and it can be noticed that 64 of them are due to the transmission of the 64 bytes of the packet, 32 for their storage in the *FIFO*, and the remaining 16 are mostly dedicated to the *Scheduler* polling and processing.

C.1.2 Test test case 2

Objective: Measure the technological latency for a frame with maximum size

Description: This test is identical to the previous one, but this time, the size of the routed frame is of 1518 bytes, which is the maximum size allowed by the AFDX specification. The change of the frame size allows an evaluation of the maximum technological latency that can be expected with this switch, since its largest component, observed in the previous test, depends directly on the packet length. Figure C.2 shows that the last byte is sent by the *FIFO* after 76490ns, while it was received at 30590ns; therefore, the resulting technological latency is equal to 45900ns, i.e to 2295 clock cycles. 1518 cycles are used for transmission, and 759 for the frame storage in the output FIFO; the remaining 18 cycles, resulting from the scheduling process, confirms that there is a small variable component on the latency due to the *Scheduler* behaviour. The advantage of having a 16-bit datapath is evident when dealing with long frames, since an 8-bit datapath would have brought to a technological latency 30% longer than in our case. The same behaviour described for the previous test case can be observed in the waveform corresponding to this test as well.

C.1.3 Test test case 3

Objective: Verify the routing functionalities of the switch when a single input is considered

Description: To check if the core routes incoming frames towards the expected destination, frames of different VLs have been sequentially sent to the same input, after having configured the various VLs to have different destination ports. The destination ports for each VL have been set as shown in Table C.1. To recognize the packets at the output, their payloads have been filled with the value of the corresponding virtual link.

Figure C.3 shows the resulting waveform for this test: each frame reaches the expected output, thus the main functionality of the system is correctly implemented. It can be noticed that each frame reaches its destinations after a time lapse equal to the technological latency previously measured, and no congestion can be observed since each packet can be completely forwarded to the corresponding *FIFO* before the reception of the following one is completed (in fact the transmission of each packet at the output starts before the following one is received).

C.1.4 Test test case 4

Objective: Analyse how the scheduling of concurrently received frames is performed

Description: In this scenario, five frames are received concurrently at the five input ports, and the scheduling process is observed; the goal is to study how the *Scheduler* handles this concurrent reception (the five *Queues* have a frame ready for transmission at the same time). The VLs have been configured to have destinations that can give visually significant waveforms: as illustrated in Table C.2 the virtual links 1, 2, and 3 share one output, while VL 4 and 5 have one destination in common between them but not with the other three. The waveforms obtained with this simulation are presented in Figure C.4, which has also been coloured highlighting each packet with a different colour.

When the *Queues* answer to the *Scheduler* request of information, it can start the scheduling routine. It initially determines that the `Queue_0` can start a transmission towards `FIFO_0`, and `Queue_3` towards `FIFO_2` and `FIFO_3` as well since these outputs are still available, all the others must wait for their destinations to be available again before the *Scheduler* determines that they can start a transmission. As soon as these two buffers have finished forwarding their frame, the corresponding output *FIFOs* advise the *Scheduler* that they are available again, using a `FIFO_available` signal. Since VL 2 and 3 share one destination, the latter will have to wait another round to be forwarded. In the waveforms, it can be observed that

Table C.1 Test 3: VLs and their destinations

# VL	Destinations
1	port 0
2	port 1
3	port 2
4	port 3
5	port 0
6	port 0 and 1
7	port 0, 2, and 4

different jitters are introduced on each output the frame is destined to, depending on its traffic load.

The configuration array `crossbar_config` is composed in the following way: while "-1 -1 -1 -1 -1" means that no output FIFO should be connected to any of the Queues, "3 -1 4 4 2" means that `FIFO_0` is connected to `Queue_2`, that `FIFO_1` and `FIFO_2` are connected to `Queue_4`, `FIFO_4` is connected to `Queue_3`, and finally that `FIFO_3` is unconnected.

C.1.5 Test test case 5

Objective: Verify the priority management when a concurrent reception occurs

Description: In this scenario, each input port receives a frame at the same time, and the behaviour of the scheduler is studied. Differently from the case number 4, VLs have been set so that they have different priorities but one common destination. Two examples are illustrated in Figures C.5 and C.6; in the former, VL 3 and 4 have high priority and the others do not, while, in the second one, the high priority virtual links are 2 and 5. In order to make this situation more significant and also to make the waveform more readable, while VL0 has only `out_port_0` as destination, VL1 has also `out_port_1`, VL2 `out_port_1` and `out_port_2`, and so on. In Figures C.5 and C.6, each frame has been coloured differently so that it can be easily identified when it is transmitted. The waveform shows that not only the frames reach the desired output, but also that the high priority frames are transmitted before the others.

In the second example, even if the *Scheduler* starts transmitting the frame 2 since it is the first with high priority and available destinations identified in the round robin. Since all the other frames need the same destination used by it (`output_port_0` they must wait for

Table C.2 Test 4: VLs and their destinations

# VL	Destinations
1	port 0
2	port 0
3	port 0 and 1
4	port 2 and 3
5	port 1, 2, and 4

that *FIFO* to be available again. When frame 2 is completely saved in the output buffer (in Figure C.5 this moment is identified by the start of its transmission on `data_out_1`, `2`, and `3`) the other critical frame is transmitted. Only after its transmission the three low priority frames are considered, starting from the one saved in *Queue_4*, that is the one immediately after the one that just finished forwarding its packet. Each time a packet must be forwarded, the `crossbar_config` array is reconfigured, and the `send` signal is sent to the *Queue* that must start a transmission.

C.1.6 Test test case 6

Objective: Analyse the behaviour of the system in presence of input data burst at a single input

Description: A 1ms burst of back-to-back frames is sent to the switch to study how it can work when one input is working at maximum wire speed; this test can consequently prove if the system throughput is high enough to support an input burst on a single input port without creating congestion, thus avoiding accumulation of frames in internal buffers.

16 frames of variable length, and payload have been sent to `in_port_0` in 1ms, for a total of 100kbit in 1ms, thus recreating the 100Mbit/s that is the maximum wire speed supported by AFDX. In reality, not all these bits are treated by the system since preambles and interframe gaps are handled by the PHY layer that precedes the switch fabric. Every packet is sent on a different VL in order to avoid filtering due to traffic policing, and they all share the same destination port. Since it is difficult to make the 1ms simulation results readable, a zoom on the last transmitted frame is provided in C.7(b) to show the delay of the last processed packet.

The results are satisfying: packet can be transmitted by the *FIFO* even before the next frame is received and the latency added by the switch to the last packet of the burst is equal to its technological latency, as if no other packet preceded it. The bust could have an

indefinite length and no buffer overflow would occur. In figure C.7(a), the entire simulation can be observed, while in Figure C.7(b) only the transmission of the last frame is highlighted, to show its latency. The latency of the last frame is equal to 15280ns, corresponding to 764 clock cycles; since the last packet has a size of 500 bytes it can be observed once again that this latency is the result of the time of transmission (500 cycles) plus the time to write to the output FIFO (250 cycles) plus the variable time necessary for the scheduling, in this case equal to 14 clock cycles. We can conclude that congestion is avoided and no accumulation of packets in the internal memories is present when a burst is received at a single input, proving that each path of the switch has a throughput effectively higher than the maximum AFDX wire speed.

C.1.7 Test test case 7

Objective: Analyse the buffer overload for high traffic loads at all the input ports

Description: In order to explore the maximum performance that this router can provide when high traffic loads must be managed, a situation similar to the one of the previous test has been reproduced, but this time, bursts are received at all the input ports. As in the previous cases, to realize readable simulations in a reasonable time a 5 ports switch has been considered. A wire-speed traffic, similar to the one used in the previous case, has been generated at each input (frames are sent back-to-back in order to have 100kbits in 1ms of transmission at a single port). To make the test bench easier to realize all the frames received at one input have the same size, but different VL and consequently different CRCs. All the VL used in this simulation have different destinations, but `out_port_0` is shared by all of them in order to create a critical situation where high traffic loads must share the same destination, thus creating output contention. After the 1ms burst (last frame received after 1014.92μs) some frames have been accumulated in the internal memories of the system and need some additional time to be forwarded: a delay of 84970ns is added to the last received frame. This delay is longer than the maximum technological latency previously computed, equal to 45900ns (since the last transmitted byte has a length of 1518 bytes), and the additional delay of 39070ns is due to the unavoidable congestion at in the output *FIFO*. The same test has been run with a different traffic and a slightly different result has been obtained (42100ns of additional delay), thus it must be concluded that this value depends on the incoming traffic. To avoid buffer overflow large buffers have been instantiated in the system for this test.

The congestion caused by high loads coming from multiple ports directed towards the same output inevitably brings to frame accumulation in the internal memories. An analysis of the situation of these memories over time can give important information on the switch

Table C.3 Test 7: Output FIFO overflow

# FIFO depth [16 bits words]	Time elapsed before FIFO out is full
2^{12}	433μs
2^{13}	1.56ms
2^{14}	3.73ms
2^{15}	8.9ms
2^{16}	17.3ms

performance: the test case 7 has been readjusted to generate the same traffic shown in figure C.8 continuously (the same frames are repetitively transmitted in a loop also after 1ms). The memories included in the *Queues* are never going to overflow since the Scheduler can organize transmission towards the outputs fast enough to avoid congestion in this part of the system (at least for a 5 port switch). A depth of 2048 words of 16 bits seems more than enough to guarantee no overflow even for high traffic situations, since it can store two frames of maximum size.

A different condition can be observed at the output: since all the frames need to be transmitted at the `out_port_0`, an accumulation of packet waiting for transmission occurs in the corresponding output *FIFO*. As requested by the specification this buffer must be considered "full" when there is no space for a frame of maximum size of 1518 bytes in it, and if a queue tries to write a new frame in a full FIFO, this is ignored and lost to avoid the corruption of frames already present in the memory; even if a real overflow is avoided this situation is to be considered erroneous and must be avoided. The modified version of the Test case 7 has been run for different sizes of the output buffer to see how this parameter affects the moment the "overflow" occurs, and the results are presented in table C.3. Even if these kind of loads directed towards a single destination port should not be present in an AFDX network, where usually only 20% of the bandwidth is used, an output buffer larger than 4096 words could be adopted to ensure overflow prevention in case of short burst on various inputs especially when the switch has an high number of inputs.

It is important to notice that in a complete switch these results would be worse: in this case the output *FIFO* is configured to start a transmission as soon as a frame is available (in order to measure the technological latency of the switching processing alone), but this means that the output port has a bandwidth of 8-bit*50MHz = 400Mbit/s. Therefore, when only 100Mbit/s are available at the output, as in a real network, the output buffer overflow would occur more rapidly. This is not a problem in previous tests where single-port traffics are used, or where the goal is to evaluate the latency of the switch fabric alone.

C.1.8 Test test case 8

Objective: Verify that frames that do not respect their BAG are dropped

Description: This and the following scenarios are conceived to test the filtering features required by the AFDX specification. In this test, the traffic policing algorithm is verified: frames related to various VLs are sent to the switch respecting their BAG, and some non-conforming frames are included in the traffic as well in order to check if they are recognized and dropped. To make the waveform more readable in Figure C.9, only one VL is used for each input, but more detailed tests have been run with multiple VLs arriving at the same input, and with the same VL received by different inputs as well. In this figure, a transmission with a BAG of 1ms is received on `in_port_0`, another with BAG 2ms on `in_port_1`, 4ms on `in_port_2`, and 8ms, and 16ms on the last two ports. All the frames that arrive after a time equal to or grater than the corresponding BAG are considered valid, also when they have an acceptable jitter. The nonconforming packets, on the other hand, are recognized and the `port_n bad_BAG` flag of their port is asserted to force the *Filter* to discard it (by sending the `drop` signal to the *Queue* that is saving it.

In Figure C.9, the erroneous frames are highlighted with a red box, and it can be observed that the corresponding `bad_BAG` signal is correctly asserted. No other packet has a bad jitter, thus no other communication is dropped. In the image, the vertical lines of the grid mark a 1ms time lapse, consequently it can be observed that the communication on `port_0` is correctly considered non conforming to its traffic policing when a packet is received before 1ms has passed from the last reception.

C.1.9 Test case 9

Objective: Verify that frames with a CRC that does not correspond to the computed one are dropped

Description: Frames that contain errors, whose structure and bit values have been corrupted during transmission, must be identified and discarded. In this scenario, one packet is received at each input at the same time, and while the first two are correct, the ones received at `in_port_2`, `in_port_3`, and `in_port_4` are erroneous. In the first one, an erroneous CRC is included in the packet, in the other two one bit of the frame has been flipped and the CRC is left as it should have been if no error were present, to see if this bit flip is detected. The goal is to check if the system recognizes these three packets as faulty and discards them.

Figure C.10, where the entire simulation is included, shows that only the frames at port 0 and 1 are forwarded to the output, as expected, because a drop signal is set on the other

three paths causing the elimination of the erroneous frames from the network. In figure C.11, the moment when the frames are recognized as erroneous is highlighted to show that when the CRC computed by the input filter corresponds to the one included in the received packet everything is correct, whereas if they are different, the `drop` signal is set high to tell the corresponding queue to ignore the last received frame.

C.1.10 Test case 10

Objective: Verify that frames with non conforming size are dropped

Description: Like in the previous test, a filtering feature is tested here: the objective of this simulation is to show how the system detects and eliminates frames with a non acceptable size. The size of the frames received at each input is given in Table C.4: for the first two ports, S_{min} and S_{max} have not been modified and the minimum and maximum Ethernet sizes are used, but the incoming packet does not respect these boundaries; for `in_port` 2 and 3, frames with a size that would be accepted by the IEEE 802.3 standard are used, but custom boundaries for the frame size are defined in the configuration tables. The communication received at `in_port_4` is the only one conforming to the VL parameters that must consequently be forwarded, all the others should be dropped. In Figure C.12, it can be observed that the required functionality is correctly implemented and that all the non conforming frames are detected and flagged, using the corresponding `drop` signal, in order to them to be dropped by the corresponding Queue. Only the correct packet reached its destination.

C.1.11 Test case 11

Objective: Verify that frames with an excessive latency are dropped

Description: The last test provided was conceived to show how the frames that spend too much time in the internal queues are discarded since they must be considered too old. The specification defines the `max delay` on a per port basis, thus independent from the VL of the considered frame. To check this property, a fault has been inserted in the *Scheduler* in order to skip the `queue_0` while performing the Round Robin algorithm, thus preventing it from forwarding the received frames and consequently adding an excessive delay to them. Two frames are sent to the `in_port_0`, i.e. to the faulty internal path of the system, and two different VLs have been used for them, to ensure that the delay is determined on a per port basis. To make the waveform easier to read, in this thesis the `max_delay` has been set to only 40μs even if this value is not realistic.

Table C.4 Test 10: Frame sizes

in_port	Frame size [bytes]	VL	Smin	Smax
0	83	1	84	1538
1	1539	2	84	1538
2	500	3	700	1538
3	1100	4	84	1024
4	84	5	84	1538

As shown in Figure C.13, the two frames that have low priority are received and saved by the *Queue* in the corresponding memory and, since no `send` signal is received, they remain indefinitely there. The global `latency_timer` is incremented each 1μs and its value is stored when the last byte of a frame is received. `next_frame_arrival` is the arrival time of the first frame in the head of the FIFO, and since no high priority frames are received in this test, the low priority queue is considered. When the difference between the `latency_timer`, that represents the "now", and the `next_frame_arrival` is higher than the `max_delay` set for the input port 0 the "next" frame is discarded; to drop a frame the queue treats it as if it was transmitted (by incrementing the counter `L_frame_tx`, even if no transmission occurred, so that the read pointer can move to the next frame in the memory. This behaviour can be observed in the "drop 1st" and "drop 2nd" boxes: the arrival of the next frame to be sent is 1μs, so when the timer reaches 41μs the latency monitor determines that this frame excessed the allowed maximum latency and mark it as sent, causing the frame counter to decrease by one and the `next_frame` pointer to move to the second frame that has been received after 12μs, which is dropped as well when the timer reaches 53μs

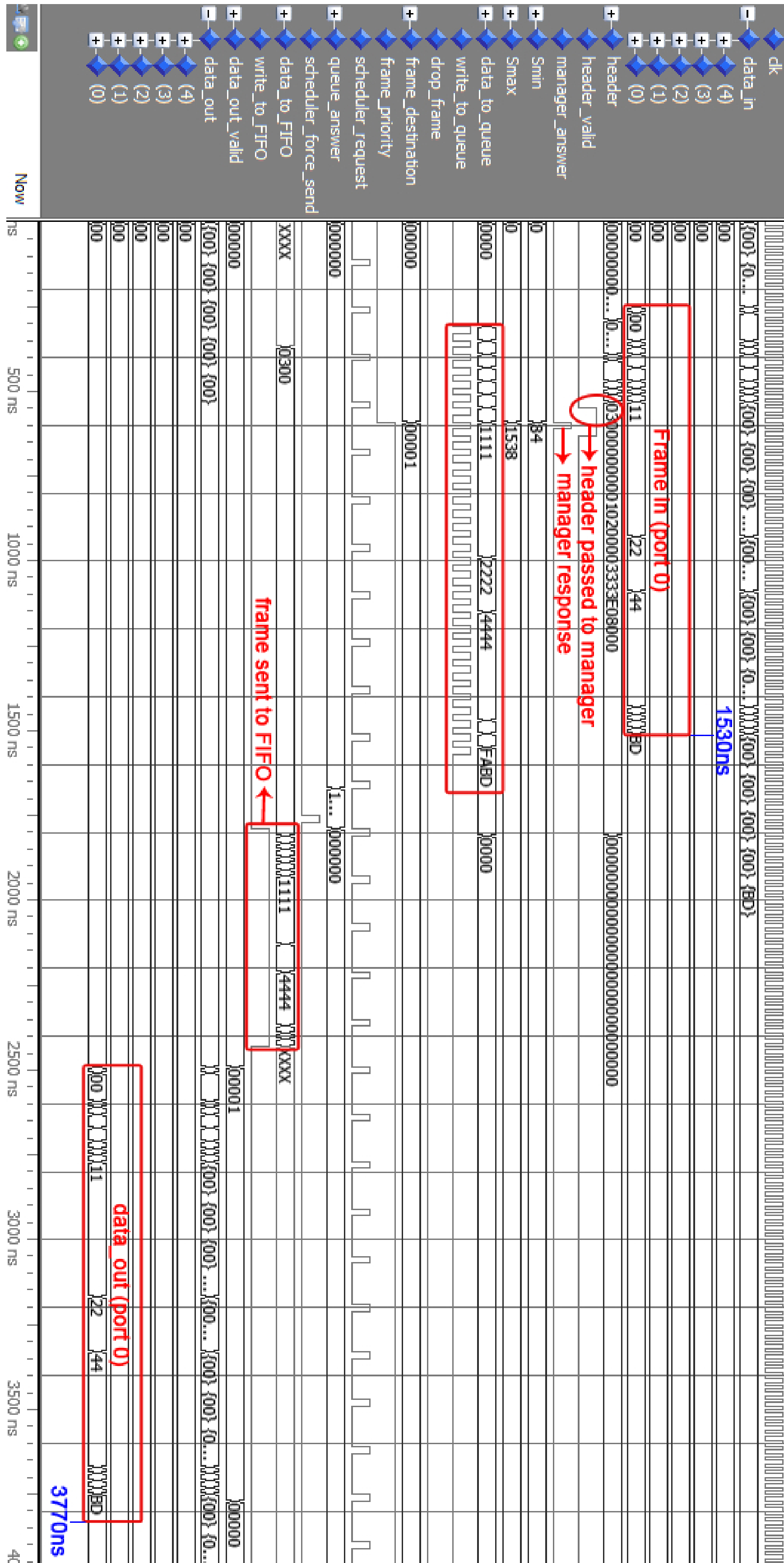

Figure C.1 Test Case 1

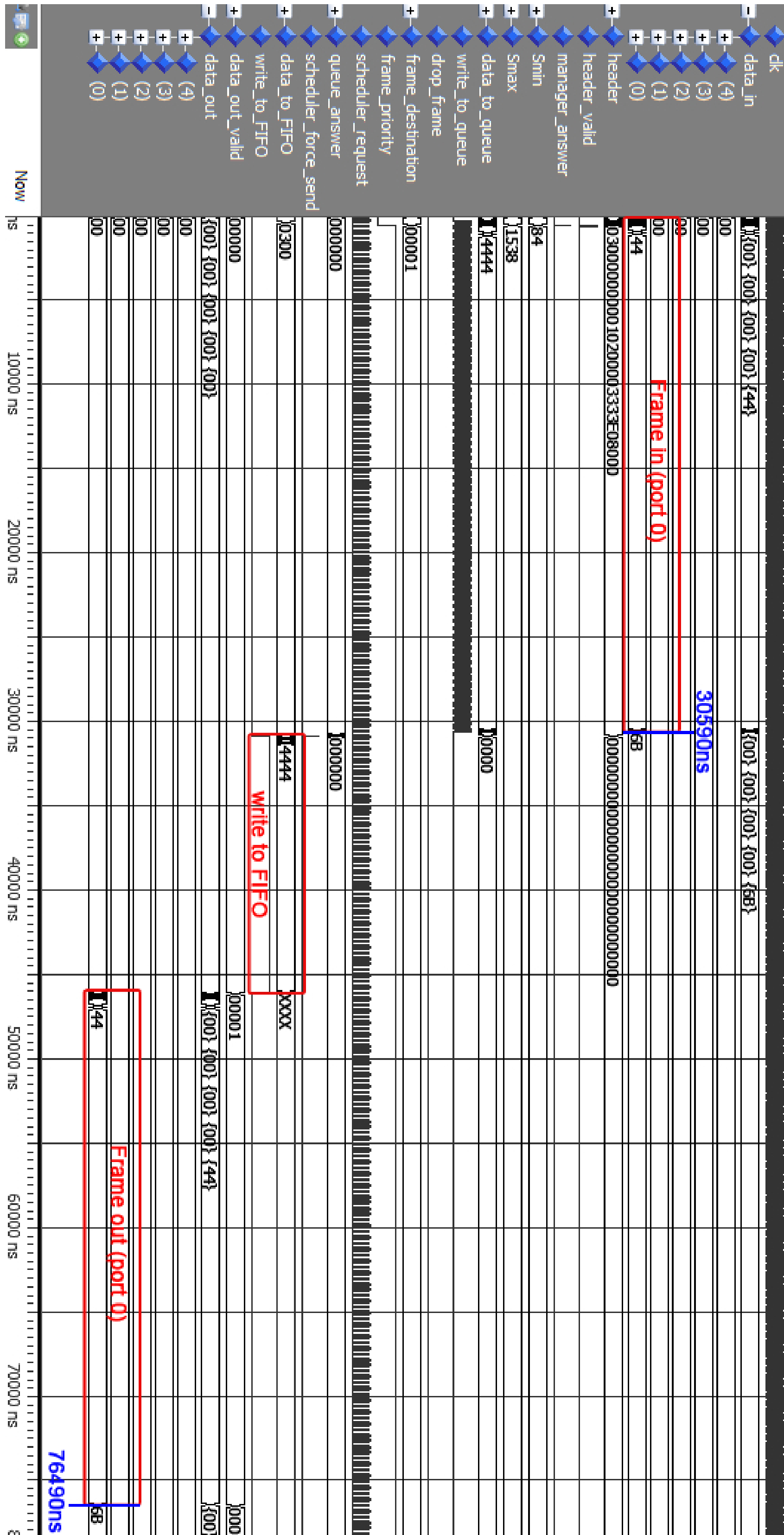

Figure C.2 Test Case 2

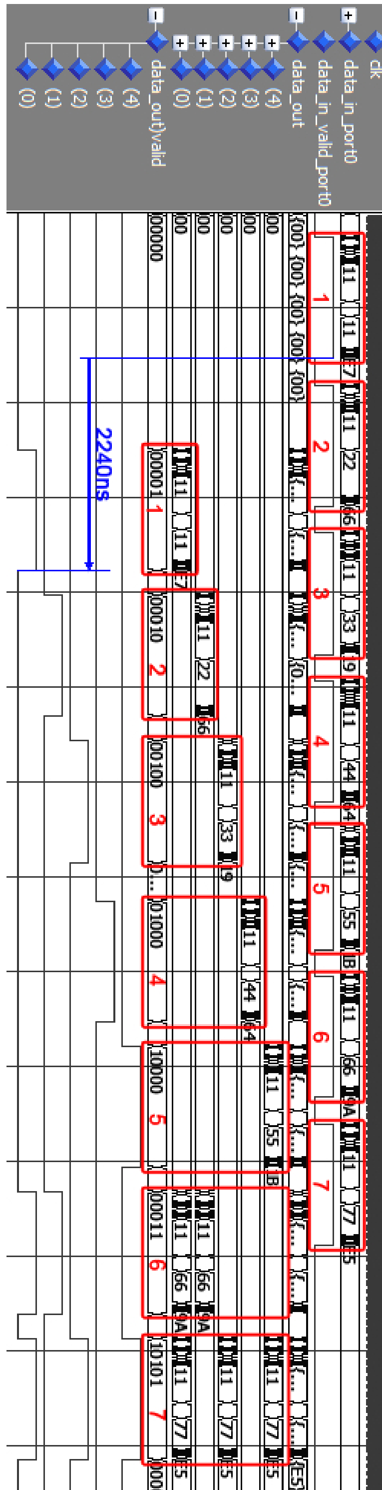

Figure C.3 Test Case 3

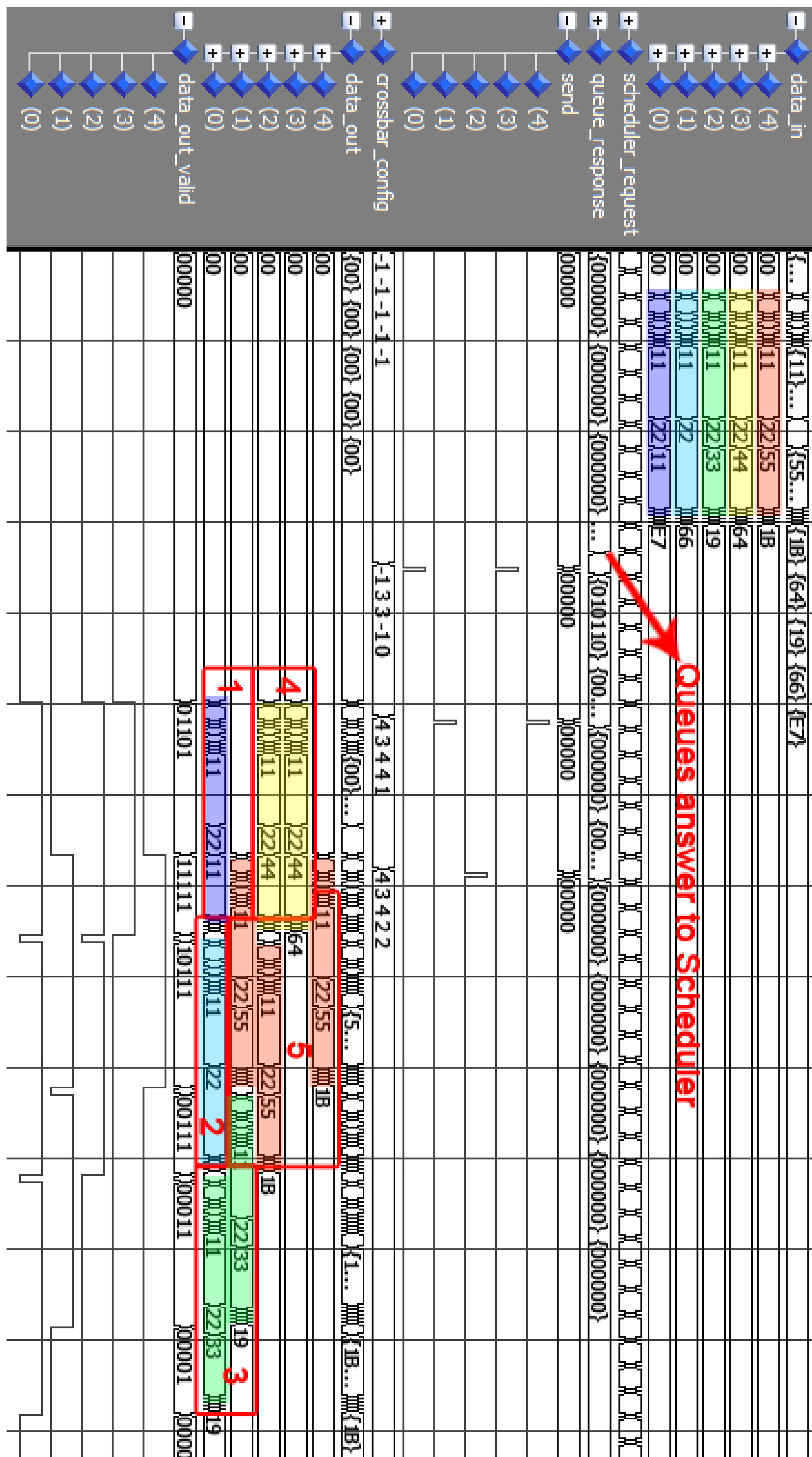

Figure C.4 Test Case 4

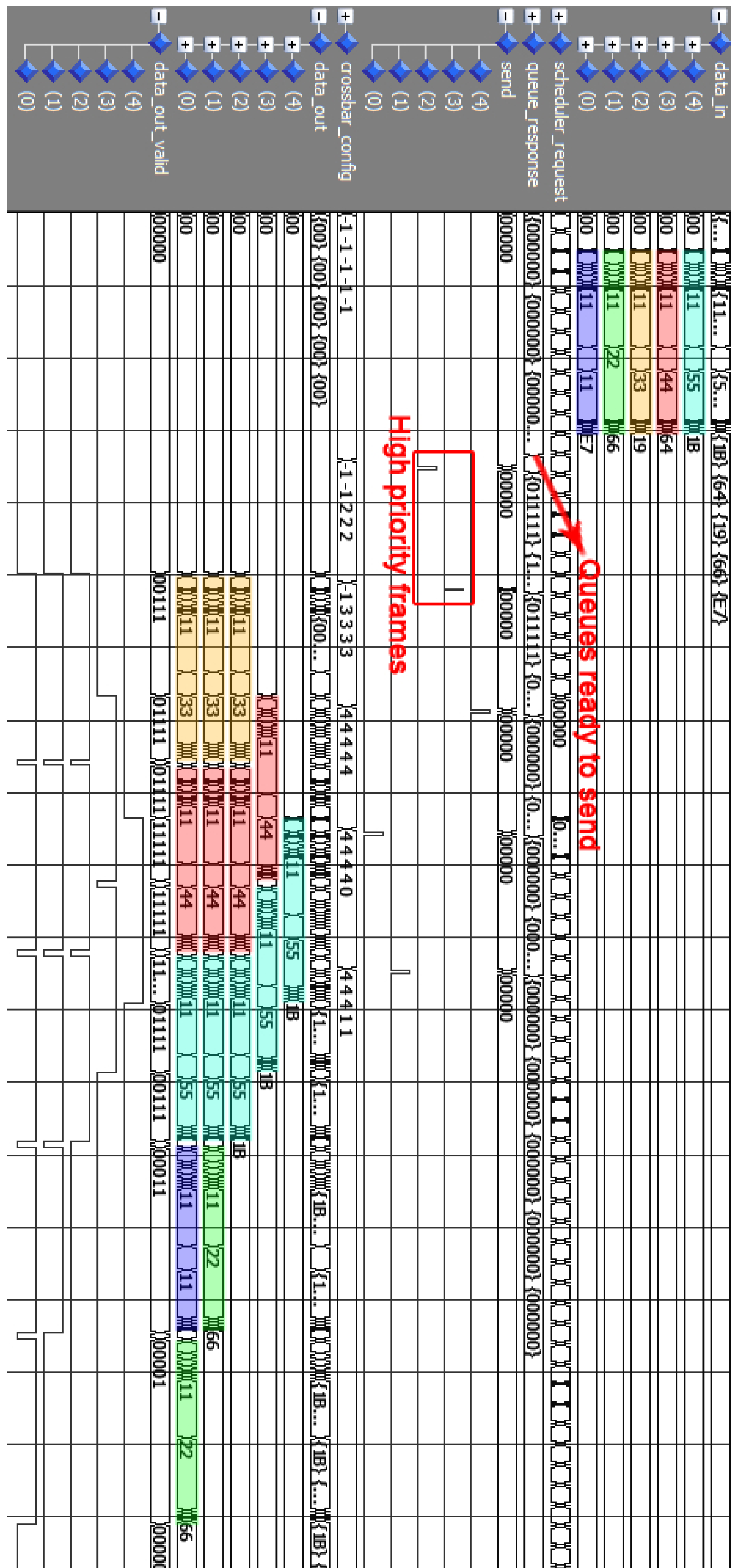

Figure C.5 Test Case 5 - first example

Figure C.6 Test Case 5 - second example

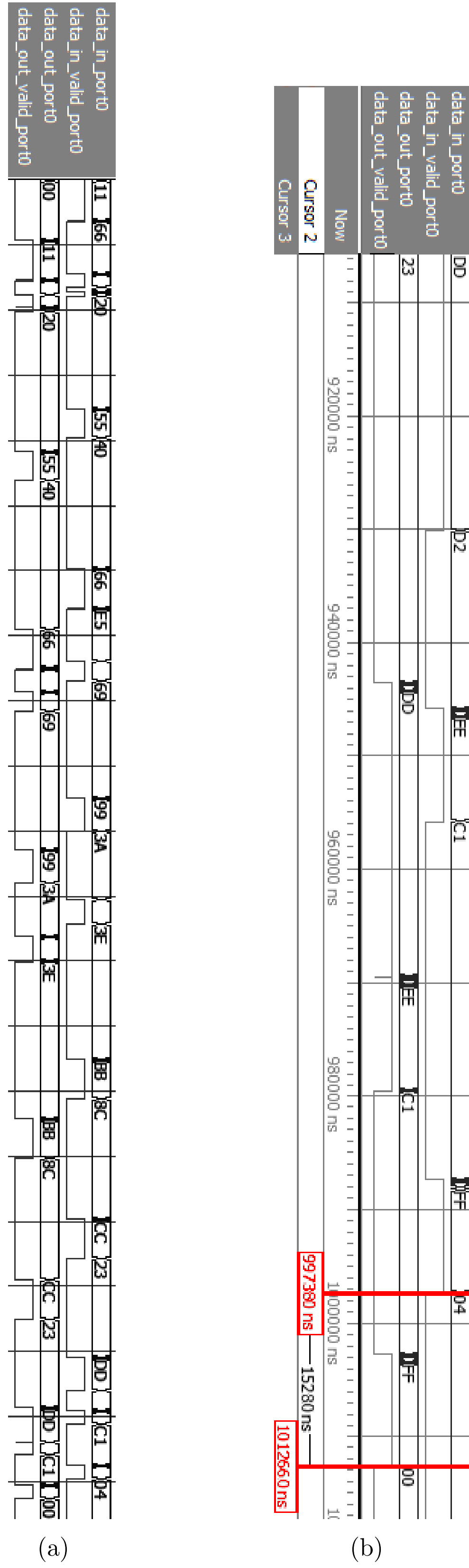

Figure C.7 Test Case 6: (a) Overview; (b) Detail on the last received frame.

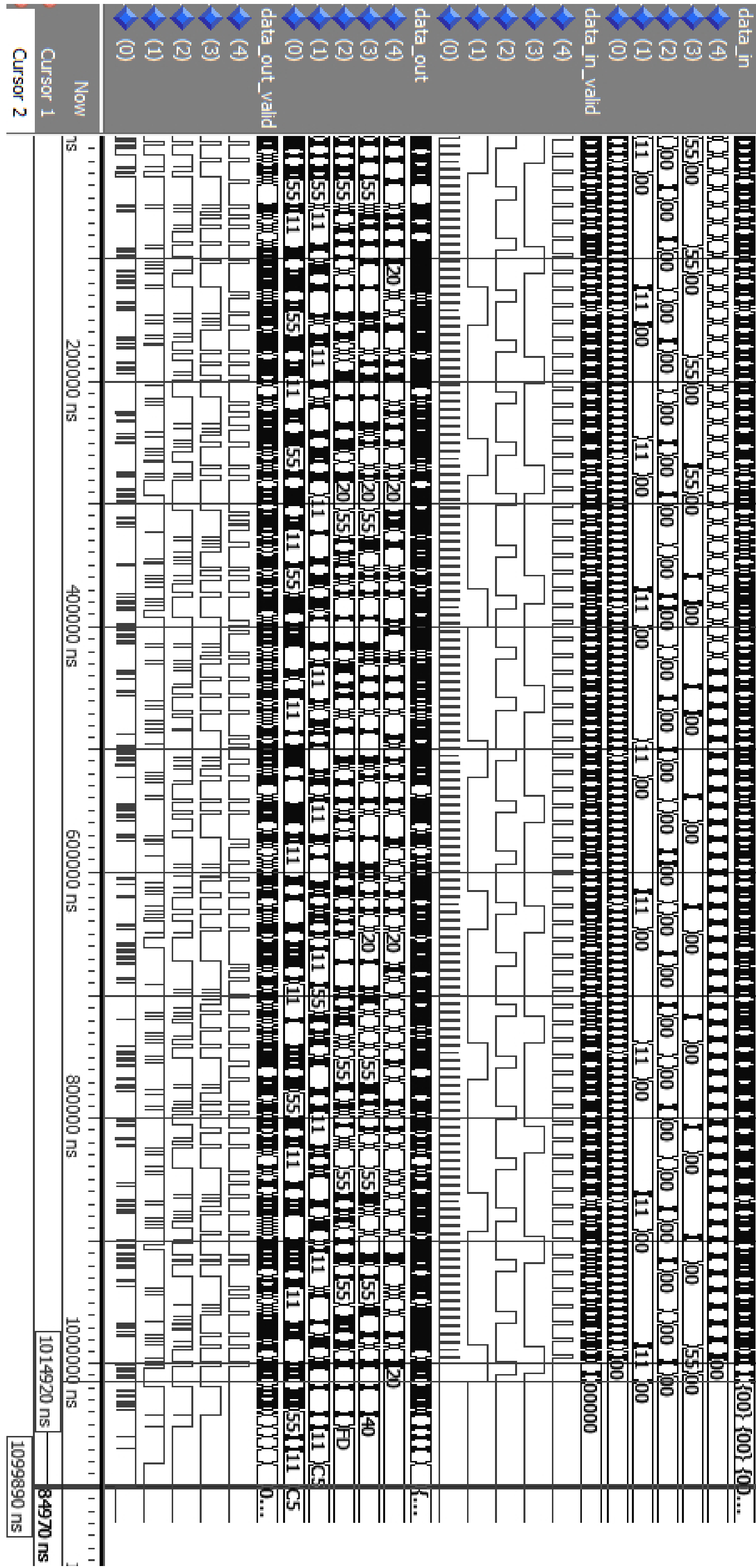

Figure C.8 Test Case 7

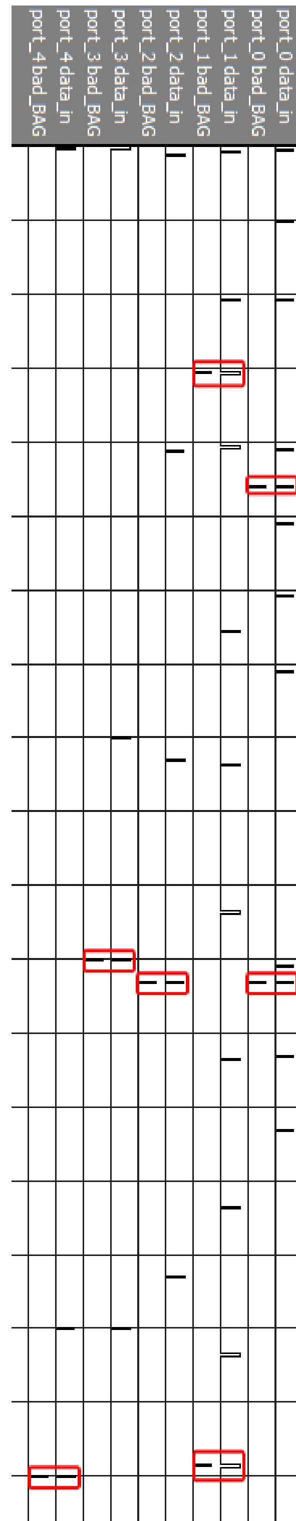

Figure C.9 Test Case 8

Figure C.10 Test Case 9 - overview

Figure C.11 Test Case 9 - CRC control

Figure C.12 Test Case 10

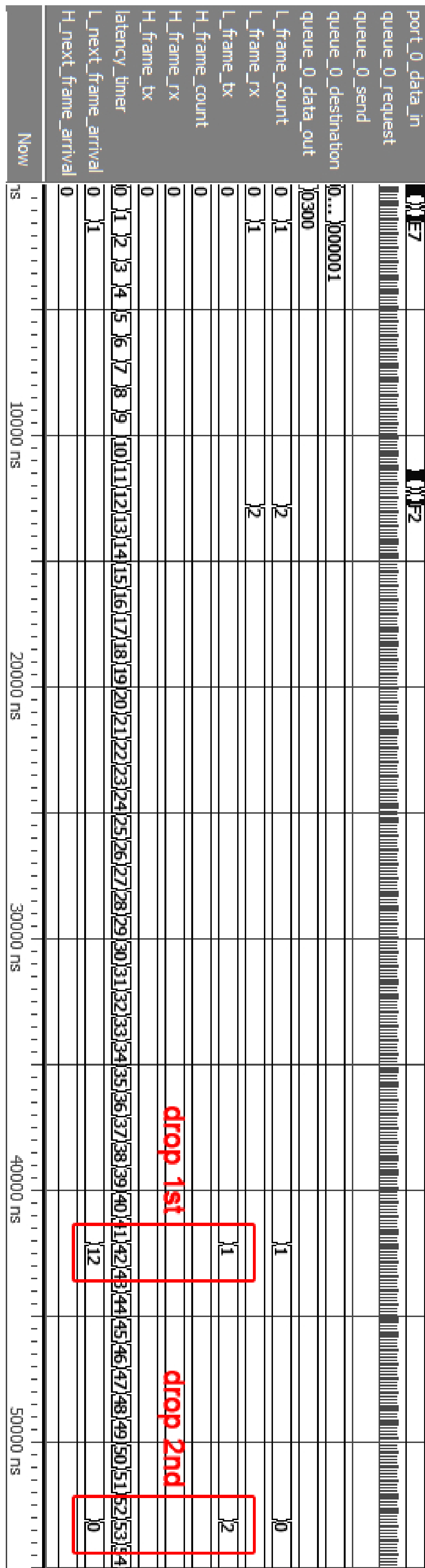

Figure C.13 Test Case 11