Automatic synthesis of hardware accelerator from high-level specifications of physical layers for flexible radio

Ganda Stéphane Ouedraogo

ANNÉE 2014

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du signal et télécommunications

Ecole doctorale Matisse

présentée par

Ganda Stéphane OUEDRAOGO

préparée à l'unité de recherche IRISA (UMR 6074)
Institut de Recherche en Informatique et Systèmes Aléatoires
École Nationale Supérieure des Sciences Appliquées et de Technologie

---

Automatic Synthesis of Hardware Accelerators from High-Level Specifications of Physical Layers for Flexible Radio

Thèse soutenue à Lannion le 10 décembre 2014

devant le jury composé de :

Tanguy RISSET
Professeur, INSA de Lyon / rapporteur

Renaud PACALET
Directeur d'études, Institut Mines-Télécom / Télécom ParisTech / rapporteur

Christophe MOY
Professeur, Supelec Rennes / examinateur

Dominique NOGUET
Ingénieur recherche, CEA-LETI / examinateur

Olivier SENTIEYS
DR INRIA / directeur de thèse

Matthieu GAUTIER
MCF, Université de Rennes 1 / co-directeur de thèse

Remerciements

À mes parents,
à ma famille,
à mes amis.

À mes encadrants.

Aux membres de l'équipe CAIRN.

Contents

1 Introduction **13**
- 1.1 Internet of Things (IoT) and Software-Defined Radio (SDR) . . . 13
- 1.2 Design Methodologies . . . 13
- 1.3 The Proposed Methodology and the Main Contributions . . . 15
- 1.4 The Outline . . . 15

I BACKGROUND **17**

2 Digital Radio and Software-Defined Radio **19**
- 2.1 Introduction . . . 20
- 2.2 Introduction to Digital Communication . . . 21
  - 2.2.1 Symbol Mapping . . . 21
  - 2.2.2 Channel Access Techniques . . . 24
  - 2.2.3 Pulse Shaping . . . 25
- 2.3 Digital Communication Technologies . . . 27
  - 2.3.1 DSP Software Platforms and Programming Environment . . . 27
  - 2.3.2 DSP Hardware Platforms and Programming Environment . . . 28
- 2.4 Flexible Radios . . . 29
  - 2.4.1 Cognitive Radios . . . 30
  - 2.4.2 Adaptive Coding and Modulation (ACM) Technique . . . 31
- 2.5 Software-Defined Radios . . . 31
  - 2.5.1 Motivations and Main Features . . . 32
  - 2.5.2 Survey of SDR Platforms . . . 33
  - 2.5.3 SDR Design Methodologies . . . 35
- 2.6 FPGA Platforms for SDR . . . 37
- 2.7 Conclusion . . . 38

3 The Waveforms of Interest **39**
- 3.1 Introduction . . . 40
- 3.2 The IEEE 802.15.4 Standard . . . 40
  - 3.2.1 ZigBee Generalities . . . 40
  - 3.2.2 The IEEE 802.15.4 PHY . . . 40
  - 3.2.3 State of the art of IEEE 802.15.4 SDR transceivers . . . 43
- 3.3 The IEEE 802.11 Standards . . . 45
  - 3.3.1 Generalities . . . 45
  - 3.3.2 The IEEE 802.11a Physical Layers (PHYs) . . . 45
  - 3.3.3 State-of-the-art of IEEE 802.11a/p SDR transceivers . . . 48
- 3.4 Conclusion . . . 49

4 High-Level Designing of Physical Layers (PHYs) 51
4.1 Introduction . . . 52
4.2 Model-Driven Engineering . . . 52
4.2.1 Generalities . . . 52
4.2.2 Domain Specific Languages (DSLs) . . . 52
4.2.3 Eclipse Modeling Framework (EMF) and Xtext/Xtend . . . 53
4.2.4 Some relevant MDE-based technologies for embedded systems . . . 55
4.3 High-Level Synthesis (HLS) . . . 56
4.3.1 A bit of history . . . 56
4.3.2 High-Level Synthesis Fundamentals . . . 57
4.3.3 Advantages of HLS . . . 58
4.3.4 Examples of mature HLS Tools . . . 60
4.4 Bringing together HLS and MDE for FPGA-SDR . . . 61
4.4.1 Dataflow Model of Computation (MoC) . . . 61
4.4.2 SDR Control Requirements . . . 62
4.5 In a Nutshell . . . 62
4.6 Conclusion . . . 63

II CONTRIBUTIONS 65

5 A Domain-Specific Language (DSL) for FPGA-Based SDRs 67
5.1 Introduction . . . 68
5.2 The Proposed Design Flow . . . 68
5.2.1 Waveform Modeling . . . 68
5.2.2 Waveform Compiling . . . 69
5.2.3 Verification and Validation (V&V) . . . 69
5.2.4 Platform Integration . . . 70
5.3 Conceptual aspects of the proposed DSL . . . 70
5.3.1 Platform Modeling . . . 70
5.3.2 DSL-Based Data-Frame Modeling . . . 71
5.3.3 DSL-Based Dataflow Modeling . . . 73
5.4 Frame-Based Control Unit . . . 76
5.4.1 A Hierarchical FSM (HFSM) for FPGA-based Dataflow Control . . . 76
5.4.2 Frame-based Control Algorithm . . . 77
5.4.3 Simulation of the proposed HFSM on the StateFlow Environment . . . 80
5.5 Library of HLS/RTL-based Functional Blocks . . . 81
5.6 Conclusion . . . 82

6 The DSL-Compiling Framework 83
6.1 Introduction . . . 84
6.2 DSL Implementation . . . 84
6.2.1 Parsing . . . 84
6.2.2 Abstract Syntax Tree (AST) . . . 85
6.3 DSL Compiler Flow . . . 85
6.3.1 AST Verification . . . 85
6.3.2 Waveform Generation . . . 87
6.4 Platform Programming . . . 90
6.5 Conclusion . . . 91

7 A Case Study: DSL-based Specification and Implementation of PHYs 93
7.1 Introduction . . . 94
7.2 Testbed Description . . . 94
7.2.1 Nutaq Perseus 6010 Motherboard . . . 95
7.2.2 Radio420X Radio Front-end Daughterboard . . . 95
7.2.3 Perseus 6010 Software Development Tools . . . 95

7.3 DSL-based Platform Modeling . . . 98
7.4 DSL-based IEEE 802.15.4 PHY . . . 99
7.4.1 IEEE 802.15.4 PHY Data-Frame Modeling . . . 99
7.4.2 IEEE 802.15.4 PHY Transceiver Modeling . . . 100
7.5 DSL-based IEEE 802.11a PHY . . . 103
7.5.1 IEEE 802.11a PHY Data-Frame Modeling . . . 104
7.5.2 IEEE 802.11a PHY Transceiver Modeling . . . 105
7.6 The "adaptive" keyword . . . 108
7.7 Validation and Synthesis Results . . . 109
7.8 A few remarks regarding the development time . . . 111
7.9 Conclusion . . . 112

8 Conclusion and Perspectives **115**

Appendices **121**

A Résumé en français **123**

B HLS Specifications **131**
B.1 IEEE 802.15.4 PHY HLS specification . . . 131
B.1.1 Transmitter . . . 131
B.1.2 Receiver . . . 131
B.2 IEEE 802.11a PHY HLS specifications . . . 141

List of Figures

2.1 Digital transceiver consisting of digital and analog components. 20
2.2 The OSI Model. 21
2.3 BPSK, QPSK, and 16-QAM. 23
2.4 Impulse response and frequency response of raised cosine pulse. 27
2.5 Harvard Architecture. 28
2.6 Abstract FPGA representation (left) and a typical FPGA design flow (right). . . . 29
2.7 Cognitive cycle composed of three major states. 30
2.8 Ideal Software-Defined Radio (SDR). 32
2.9 Realistic Software-Defined Radio (SDR). 33
2.10 KUAR System Diagram. 34
2.11 GNU Radio Companion GUI. 37

3.1 ZigBee PPDU format. 42
3.2 PHY IEEE 802.15.4 transmitter. 44
3.3 PHY IEEE 802.15.4 receiver. 44
3.4 IEEE 802.15.4 SDR Receiver. 45
3.5 IEEE 802.11a PPDU format. 45
3.6 PHY IEEE 802.11a transmitter. 47
3.7 PHY IEEE 802.11a receiver. 47
3.8 GNURadio PHY IEEE 802.11p receiver. 48

4.1 Generic architecture of DSL processing. 53
4.2 EMF possible representations (Java, XML, UML). 54
4.3 Traditional co-design flows *vs.* MOPCOM co-design flow. 55
4.4 Generic High-Level Synthesis Flow. 57
4.5 A for-loop parsed into a CDFG. 59
4.6 A dataflow graph for a digital filter. 62
4.7 An SDF signal flow graph (left) and its possible implementation (right). 63

5.1 Proposed Design Flow. 69
5.2 DSL-based platform description. 71
5.3 Generic data frame. 71
5.4 DSL-based description of a generic data frame. 72
5.5 DSL-based description of an FB. 74
5.6 FB equivalent architecture. 74
5.7 Simplified design constraint management scheme based on a throughput area trade-off. 75
5.8 Transceiver Hierarchical FSM. 77
5.9 Enable distribution for the activation of FB_j operating on F_i. 79
5.10 Control algorithm illustration. 79
5.11 Proposed HFSM modeling and simulation on StateFlow. 80
5.12 IEEE 802.15.4 transceiver matched filter specification in Vivado HLS. 81
5.13 IEEE 802.15.4 transceiver matched filter specification in Catapult. 82

6.1 Generic AST representation. 84

6.2 DSL-compiling framework. 86
6.3 Zoom into the actual AST representation. 86
6.4 Scheduled Design and Datapath Sate Diagram at the block-level. 87
6.5 Example of tcl script generating RTL code with loop constraints. 89
6.6 Hierarchical view of generated RTL for a given FB. 90
6.7 Waveform assembly view. 90

7.1 Testbed Description. 94
7.2 Validation and Testing Environment. 95
7.3 Perseus block diagram 96
7.4 LMS6002D transceiver block diagram. 97
7.5 MBDK-based hand-coded ZigBee transceiver design. 98
7.6 MBDK-based hand-coded ZigBee transceiver design. 99
7.7 DSL-based Perseus 6010 platform description. 100
7.8 DSL-based IEEE 802.15.4 PHY data frame representation. 100
7.9 DSL-based IEEE 802.15.4 PHY data frame description. 101
7.10 DSL-based IEEE 802.15.4 PHY transmitter description. 102
7.11 DSL-based IEEE 802.15.4 PHY transmitter implementation. 103
7.12 DSL-based IEEE 802.15.4 PHY receiver implementation. 104
7.13 DSL-based IEEE 802.15.4 PHY receiver description. 105
7.14 DSL-based IEEE 802.11a PHY data frame representation. 106
7.15 DSL-based IEEE 802.11a PHY data frame description. 106
7.16 DSL-based IEEE 802.11a PHY receiver description. 107
7.17 DSL-based IEEE 802.11a PHY transmitter implementation. 108
7.18 DSL-based IEEE 802.11a PHY receiver implementation. 108
7.19 Transmitted (left) and received (right) IEEE 802.15.4 baseband signals. 109
7.20 Decoding the transmitted IEEE 802.15.4 signal with VSA. 110
7.21 OFDM (left) and ZigBee (right) spectrum. 111
7.22 Throughput/Area trade-off of the CorrBench FB. 112
7.23 Fine estimation: throughput/area tradeoff of the *CorrBench* block. 113
7.24 Resource estimation for the FFT FB. 114

List of Tables

2.1 Summary of state-of-the-art SDR languages. 38

3.1 Frequency bands and data rates. 41
3.2 Channel page and channel number. 42
3.3 Symbol to chip mapping 43
3.4 PHY IEEE 802.11a modulation rate-dependent parameters. 46

4.1 A comparative study between different HLS tools. 60

7.1 Resource estimation for the IEEE 802.15.4 and IEEE 802.11a receivers. 111

Acronyms and Abbreviations

ACM Adaptive Coding and Modulation
ADC Digital to Analog Converter
ASIC Application Specific Integrated Circuit
AST Abstract Syntax Tree
AWGN Additive White Gaussian Noise
BER Bit Error Rate
BNF Backus-Naur Form
BPSK Binary Phase Shift Keying
BSDK Board Software Development Kit
CISC Complex Instruction Set Computer
CP Cyclic Prefix
CPU Central Processing Unit
CR Cognitive Radio
DAC Digital to Analog Converter
DSE Design Space Exploration
DSL Domain Specific Language
DSSS Direct Sequence Spread Spectrum
DSP Digital Signal Processing
FB Functional Block
FDM Frequency Division Multiplexing
FFT Fast Fourier Transform
FHSS Frequency Hopping Spread Spectrum
FIR Finite Impulse Response
FMC FPGA Mezzanine Card
FPGA Field Programmable Gate Array
FSM Finite State Machine
GPL General Purpose Language
GPP General Purpose Processor
HDL Hardware Description Language
HFSM Hierarchical Finite State Machine
HLL High-Level Language
HLS High-Level Synthesis
ICI Inter-Channel Interference
IDE Integrated Development Environment
IFFT Inverse Fast Fourier Transform

IIR Infinite Impulse Response
IoT Internet of Things
ISI Inter-Symbol Interference
ISM Industrial Scientific and Medical
MBDK Model Based Design kit
MAC Medium Access Control
MDE Model Driven Engineering
MoC Model of Computation
OFDM Orthogonal Frequency Division Multiplexing
OQPSK Offset Quadrature Phase Shift Keying
OSI Open System Interconnection
PER Packet Error Rate
PHY Physical Layer
QAM Quadrature Amplitude Modulation
QoS Quality of Service
QPSK Quadrature Phase Shift Keying
RF Radio Frequency
RISC Reduced Instruction Set Computer
RTL Register Transfer Level
RX Digital Receiver
SCA Software Communication Architecture
SDF Synchronous Data Flow
SDR Software-Defined Radio
SNR Signal to Noise Ratio
TRX Digital Transceiver
TX Digital Transmitter
UML Unified Modeling Language
USRP Universal Software Radio Peripheral
VHDL VHSIC Hardware Description Language
XPS Xilinx Platform Studio

Chapter 1

Introduction

1.1 Internet of Things (IoT) and Software-Defined Radio (SDR)

The Internet of Things (IoT) [1][2] is a promising concept which purpose is to connect billions of communicating devices (the things) through an internet-like network. Such things are expected to range from simple RFID tags to powerful smartphones interacting with various types of nodes or even with human beings themselves. Further to this, the IoT is intended to be implemented through wireless links without any predefined standards or infrastructures. However, such technology would require multiple communicating devices to coexist in a spectrum limited environment. To tackle this issue, diverse approaches have been proposed in the literature. Thus, solutions like the Cognitive Radio (CR) [3][4][5], the Cooperative Radio [6][7][8] or the Green Radio [9] are presented as the enabling technologies to implement the IoT.

Traditional radios have shown some limitations regarding the implementation of such brand new technologies. Indeed, they often consist of a set of dedicated hardware which usually implements a single function at a time. However, the cognitive radio, the cooperative radio or the green radio technologies foster flexible architectures that can adapt to their environmental conditions. A promising alternative which is also presented as a key enabling technology for these concepts is the Software-Defined Radio (SDR)[10][11]. SDR fosters the implementation of most of the processing stages in digital form by using programmable devices. Hence, this architecture is much more flexible than using dedicated hardware while requiring fewer components. It also enables to change the functionality of the radio at any time by simply using software updates. However, the SDR approach comes with some drawbacks which are related to the reliability and the security of the software. Moreover, the encountered SDR solutions require more power than dedicated-hardware based solutions.

Whilst, these concepts have come to a certain maturity, the underlying methodologies or tools which support their implementation still represent an open research topic. Indeed, it is important to point out the fact that an efficient toolset would allow to achieve better productivity by considerably shortening the time-to-market.

1.2 Design Methodologies

Methodologies for implementing digital systems often referred to as design flows, have motivated a lot of research work throughout the decades. As a result, the level of abstraction was raised from the transistor-level up to the system-level all the way through gate-level and function-level. Each of these breakthroughs was welcome with a lot of skepticism with respect to the achievable performance even though their main purpose was to allow the designers to focus more on the functionality rather than its implementation. To this aim, several steps within the design process were automated thanks to some tailored algorithms as well as mechanisms for source code or other artifacts generation. Such automation processes is generally supported by a dedicated compiler.

The design methodologies can be split up into two categories. On the one hand, some tools support the development of digital applications on software-based technologies such as microprocessors or DSP microprocessors. Their entry point consists of a high-level description of the intended application written in a high-level language such as C/C++. Afterwards, this description is compiled to a machine language which consists entirely of numbers (bits) that are understood by the target processor (machine). The assembly language is often used for intermediate representation however, it consists of variables and names which are relatively easy to interpret by human beings unlike machine language. On the other hand, some tools support the development of digital systems on hardware-based technologies like Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs). These technologies allow the designer to build customized circuit architectures which are tailored to the application's needs. However, FPGAs provide more flexibility compared to their ASICs counterparts. Indeed, FPGAs enable to program a customized circuit architecture through a configuration file. It can be reprogrammed to implement a different functionality unlike ASICs whose circuits are hardwired. Their underlying design tools usually consider as entry point a description of the application in form of a Hardware Description Language (HDL) such as VHDL or Verilog. The description which is usually performed at the Register Transfer Level (RTL) is first synthesized into a netlist which is the equivalent description at the gate-level in HDL. Following this step, the netlist is further processed and placed and routed afterwards. The placement and routing tool generates the layout or the configuration files which are not in HDL. Such files are used either to program the FPGA or for ASIC manufacturing.

An emerging approach consists in raising the level of abstraction while considering FPGA or ASIC based designs. Referred to as High-Level Synthesis (HLS), it fosters the use of high-level languages as entry point in both FPGA and ASIC design flows. Thus, an application can be first specified in C/C++ for instance and then directly compiled to a configuration file for the target FPGAs or ASICs. Once again, the approach allows the designers to focus on the functionality of the application rather than the underlying hardware. The HLS is supported by a set of academic and industrial tools such as *Catapult* from Calypto and *Vivado HLS* from Xilinx. In summary, HLS flows can be used for rapid prototyping when hardware fabrics are being considered in a digital system implementation process. However, they emphasize more on datapath designing rather than control path. Hence, the control aspects should be handled separately whenever complex control schemes are required. Indeed, HLS was first thought as a processor generator employing native sequential languages such as C/C++. The related compilers have succeeded in extracting some parallelism in the specified applications so as to build an efficient datapath. Control path on the other hand is handled at function-level rather than system-level. For instance, HLS did not properly address the specification of state machines, which are the mainstream structure for defining the control logic of an application at the system-level.

In the context of SDR, the design tools must support the specification as well as the implementation of any applications while being independent of the underlying hardware, which could ensure the *portability* of the solution on different platforms. Furthermore, the tools entry point should be a high-level language for *programmability* purpose and finally, such tools must address the *reconfigurability* of the application at a higher level of abstraction. To this aim, different proposals can be found in the literature and a state of the art of the design methodologies for SDR implementation is provided in this document.

The FPGA technology is often used as a simple hardware accelerator in a typical SDR platform. Paradoxically, it is also presented as a key enabling technology for SDR since it trades-off between design throughput and power consumption while offering reconfigurability capabilities. In effect, the main reason why FPGA platforms have not encountered a lot of success within the SDR community is essentially because of their programming model which relies on HDLs. This programming model requires a deep knowledge in hardware design, which contrasts with the stated goal of SDR that is to say a software intensive platform. To remedy this issue, we believe that the HLS technology can be leveraged so as to define an FPGA-based SDR design flow.

In essence, the design methodologies for implementing digital radios are broadly addressed in the literature. However, the SDR technology has come with some new challenges which require rethinking the development process by integrating some relevant features that could tackle the domain issues.

1.3 The Proposed Methodology and the Main Contributions

Our proposal consists of a design flow for SDR specification and implementation on FPGA-based platforms. It leverages the HLS principle which allows rapid prototyping on FPGA fabrics while addressing the control requirements at the system-level. The entry point of the proposal is a Domain Specific Language (DSL) which is a customized language tailored to a domain's needs. The DSL was entirely developed with the Xtext/Xtend framework which is an Eclipse plugin. It coarsely allows the designers to capture different parts of an SDR waveform, *i.e.* the frame model and the dataflow structure, at a higher level of abstraction while instantiating HLS-based Functional Blocks (FBs). The DSL is featured with a compiler which purpose is to automate as many steps as possible in the implementation process. The compiler further analyzes the description of the waveform made with the DSL and then produces a set of artifacts such as synthesis script and source code. Finally, the proposed flow has been validated on two well-known waveforms, namely the IEEE 802.15.4 PHY and the IEEE 802.11a PHY transceivers which implement the radio communication protocol of the ZigBee and the WiFi technologies respectively.

In sum, our contributions can be listed in the following way:

— A Domain Specific Language (DSL) which provides the primitives to rapidly prototype the dataflow applications meant for SDR. The DSL is combined to HLS tools so as to take advantage of their offerings.

— A frame-based algorithm to automatically generate an appropriate control path capable of handling the reconfiguration requirements of a multi-rate complex dataflow specification.

— A design space exploration scenario which enables the appropriate selection of the blocks composing the final design. Such scenario was made possible by the usage of HLS for rapid prototyping.

— A compiler which implements the proposed algorithms by further analyzing the DSL-based descriptions and generating the required artifacts such as synthesis scripts as well as source code.

— A library of functional blocks specified in HLS and compatible with *Catapult* or *Vivado HLS*. The library includes some blocks which are part of either the IEEE 802.15.4 PHY or the IEEE 802.11a PHY.

1.4 The Outline

This document is basically divided into two major parts. The first part, Chapters 2 to 4, entitled *Background* provides a comprehensive overview of the SDR throughout the different notions involved. The second part, Chapters 5 to 7, entitled *Contributions* covers the details of our proposal which consists of a design flow for implementing FPGA-based SDRs. More detailed descriptions of the chapters follow.

— Chapter 2 discusses some digital radio principles as well as the underlying technologies. Furthermore, the chapter presents the "big picture" of SDR including platforms and design methodologies.

— Chapter 3 provides the details of two PHYs that we have considered in this work for validation purpose. The two PHYs are the IEEE 802.15.4 PHY and the IEEE 802.11a PHY which implement the radio communication protocols of the ZigBee and the WiFi technologies respectively.

— Chapter 4 covers the methodologies which purpose is to raise the level of abstraction for PHY designing. It introduces some concepts such as the Model Driven Engineering (MDE) or the HLS.

— Chapter 5 provides an overview of our proposal. The chapter discusses the conceptual aspects of the DSL through generic examples. In addition, a discussion on the automatically generated control logic is provided.

— Chapter 6 details the features of the associated compiler. To this aim, it provides more information on the intermediate representation of a DSL-based SDR waveform description and then discusses the mechanisms that we have developed to produce the final waveform.

— Chapter 7 discusses the specification and the implementation of the two aforementioned

waveforms with the proposed flow. The ensuing results are interpreted and some conclusions are drawn.

— Chapter 8 first summarizes the entire work depicted in this document and discusses the perspectives afterwards.

A appendix chapter is provided at the end of the document. It includes HLS specifications, which are intended to illustrate a typical description of a PHY with HLS tools.

Part I

BACKGROUND

Chapter 2

Digital Radio and Software-Defined Radio

Contents

2.1 Introduction 20
2.2 Introduction to Digital Communication 21
2.2.1 Symbol Mapping 21
2.2.2 Channel Access Techniques 24
2.2.3 Pulse Shaping 25
2.3 Digital Communication Technologies 27
2.3.1 DSP Software Platforms and Programming Environment 27
2.3.2 DSP Hardware Platforms and Programming Environment 28
2.4 Flexible Radios 29
2.4.1 Cognitive Radios 30
2.4.2 Adaptive Coding and Modulation (ACM) Technique 31
2.5 Software-Defined Radios 31
2.5.1 Motivations and Main Features 32
2.5.2 Survey of SDR Platforms 33
2.5.3 SDR Design Methodologies 35
2.6 FPGA Platforms for SDR 37
2.7 Conclusion 38

2.1 Introduction

Digital radio systems [12][13][14][15] resulted from an increasing demand in terms of efficiency and control over the electronic applications. Indeed, analog designs which physically operated the signal turned out to be less efficient when high data rates and low power consumption were required. In addition, pure analog systems gave uncertain performance in production in the sense that they did not guarantee accuracy and perfect reproducibility of the designs [16].
The advent of the transistor and the limitations of the analog systems have led the designers to rethink the overall electronic designing processes by introducing some new concepts. Thus, electronic systems have evolved from analog devices to hybrid fabrics composed of both analog and digital components as illustrated in Figure 2.1. Figure 2.1 shows the architecture of a current digital radio transceiver which comprises a baseband processing module to perform the digital processing and a front-end module to modulate a signal on a career frequency. An antenna transmits/receives the signal into/from the propagation channel. A mainstream approach consists now in designing flexible radio transceivers which are capable to adapt to the environmental conditions. Such transceivers are intended to improve the usage of the spectral resource which tends to be more and more scarce owing to an under optimized utilization. Moreover, the evolution of the digital platforms has enabled to reach a certain flexibility at the price of a lesser efficiency in terms of power consumption. Software-Defined Radio (SDR) [10][11] can be presented as a potential implementation of a flexible radio. In addition to the flexibility, the SDRs also address the programmability and the portability of a waveform. In this chapter, we will first emphasize on the baseband transceivers architectures which are mainly ruled by some signal processing principles. Flexible radios and SDRs will be discussed afterwards.
Section 2.2 and 2.3 introduce the digital communication principles and the underlying technologies respectively. In Section 2.4, the concept of flexible radio is discussed and illustrated throughout some examples. Section 2.5 introduces the SDR concept while Section 2.6 discusses the usage of the Field Programmable Gate Array (FPGA) fabrics as physical hardware for SDR development. Conclusions are drawn in Section 2.7.

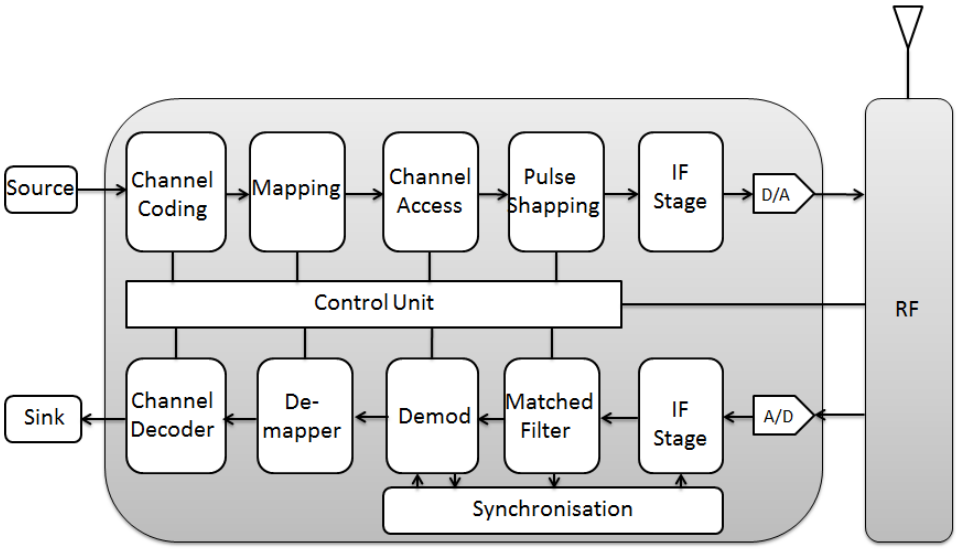

Figure 2.1 – Digital transceiver consisting of digital and analog components.

2.2 Introduction to Digital Communication

Generally speaking, a radio transceiver can be defined as any device that is used to exchange an information signal from point A to point B through wired or wireless channels. It is usually referred to as the Physical Layer (PHY) which is composed of hardware transmission technologies and represents the lowest layer in the OSI model. The OSI model, shown in Figure 2.2, characterizes the internal functions of a communication system. The PHY layer plays an important role within the OSI model since it provides a medium access to the higher layers. The information theory introduced by Claude Shannon [17], partly introduced the concept of digital communication by defining some of its major principles. Indeed, the document teaches us the theoretical capacity of a channel as well as the performance which can be expected. For instance, by considering an Additive White Gaussian Noise (AWGN) channel and a passband transmission, it was shown that the maximum capacity achievable, also known as the channel capacity, is given by

$$C = W \log_2(1 + SNR), \tag{2.1}$$

where C is the bit rate (in bits per second), W the width of the spectral band in which the signal is transmitted and SNR, the Signal to Noise Ratio in this band. These results later guided the research toward source coding, channel coding and algorithmic complexity theories.
The communication theory [18][19] on the other side focuses on how the information transits from point A to B under the constraints dictated by the information theory. In other words, it deals with how to condition the signal so as to ensure its integrity throughout the channel. It is usually well-specified by telecommunication standards [20][21][22][23][24].

Our research work emphasizes on PHY specifications and implementations and therefore, the following sections provide some discussions over the main aspects of a PHY from a baseband perspective namely, the symbol mapping, the channel access techniques and the pulse shaping or signal conditioning. The following equations are taken from [12].

2.2.1 Symbol Mapping

Symbol mapping is usually the starting stage of a PHY definition depending on whether a channel coding scheme is required or not. It literally takes a sequence of streaming bits and

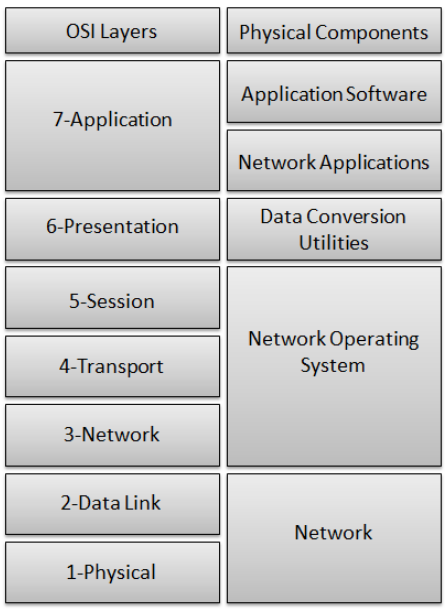

Figure 2.2 – The OSI Model.

converts (maps) them into a set of symbols predefined in an alphabet of symbols. The symbols can be either real or complex. Complex symbols are composed of two components, namely an in phase (I) component and a quadrature phase (Q) component. The choice of a symbol mapping scheme for a given standard depends on its requirements in terms of bit rate and probability of error. Thereby, several symbol mapping techniques have been proposed and each of them allows achieving some theoretical data rate and Bit Error Rate (BER). In the following paragraphs, we will discuss some of the most encountered symbol mapping techniques by highlighting their mathematical formalization together with the associated theoretical error rates.

M-ary Pulse Amplitude Modulation (M-PAM)

The M-PAM modulation consists in mapping the input set of bits into a sequence of symbols according to an M-ary alphabet or constellation. Thus, the number of possible transmitted symbols is equal to M and a unique symbol represents a set of $\log_2(M)$ bits. The alphabet is composed of the symbols s_m given by (2.2), where E_g is the energy per symbol.

$$s_m = \sqrt{E_g}(2m - 1 - M), \qquad m = 1, 2, ..., M, \tag{2.2}$$

One can note from (2.2) that the distance between two consecutive symbols is $2\sqrt{E_g}$. This distance impacts the distinction of the symbols at the receiver since the metric that is used to differentiate them is based on setting a threshold between neighboring symbols. In the context of an AWGN channel, it leads to a theoretical probability of error at the symbol-level given by (2.3).

$$P_{SER} = \frac{2(M-1)}{M} Q(\sqrt{\frac{6P_{av}T_s}{(M^2-1)N_0}}), \tag{2.3}$$

where P_{av} is the average power, N_0 the spectral density of the noise T_s the symbol period and $Q(.)$ the Gaussian density function given by

$$Q(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2},$$

where μ is the mean and σ^2 the variance of the function.

M-ary Phase Shift Keying (M-PSK)

The M-PSK modulation is a slightly different approach in the sense that in this symbol mapping technique, the information is embedded in the phase component of a signal with constant amplitude. It contrasts with the M-PAM modulation where the amplitudes vary with a constant phase. An M-PSK constellation fits onto the circumference of a circle where the symbols lay at equal angular distances. Thereby, the symbols are the phase information whose values are given by

$$\theta_m = \frac{2\pi m}{M} + \frac{\pi}{M}, \qquad m = 0, 1, ..., M-1, \tag{2.4}$$

where M is the number of symbols.

The Quadrature Phase Shift Keying (QPSK) is a particular case of an M-PSK modulation, where $M = 4$, that has found several applications in wireless communication such as in the IEEE 802.11a standard. The Offset Quadrature Phase Shift Keying (OQPSK) is a slightly modified version of the QPSK which introduced a delay of half a symbol in order to prevent fluctuations in the constant envelop of QPSK. It was employed in the definition of the PHY of the Zigbee technology [20]. For the QPSK modulation and its variant OQPSK, the distinction of the symbols at the receiver is performed by assigning a region of decision around each symbol of the alphabet. It leads to a theoretical expression of the probability of symbol error, in the context of an AWGN channel, given by

$$P_{SER} = 2Q(\sqrt{\frac{E_g}{N_0}}) - Q^2(\sqrt{\frac{E_g}{N_0}}), \tag{2.5}$$

where $\frac{E_g}{N_0}$ is the symbol SNR.

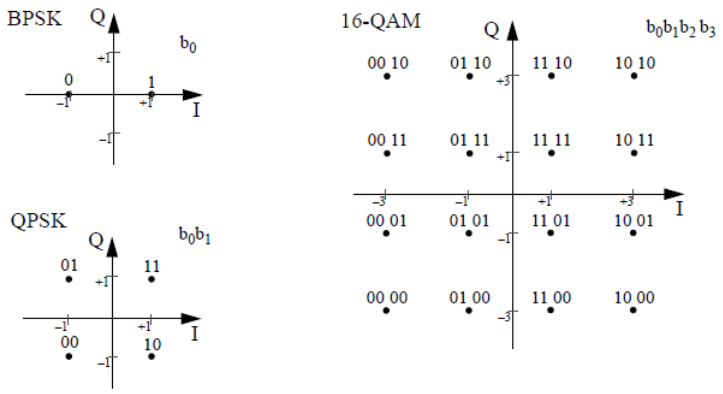

Figure 2.3 – BPSK, QPSK, and 16-QAM.

M-ary Quadrature Amplitude Modulation (M-QAM)

The M-QAM modulation is a combination of the two aforementioned modulation techniques. Indeed, it modulates the information signal into both phase and amplitude information. Doing so, the two dimensions are simultaneously exploited, which enables to achieve higher data rates. It is formalized through a complex mathematical representation that exhibits both the amplitude and the phase information. Equation (2.6) gives the complex representation of an M-QAM modulation where the amplitude corresponds to an M-PAM modulation and the phase corresponds to an M-PSK modulation. The probability of error is given in (2.7) where E_b is the energy per bit. Figure 2.3, which is taken from [21], shows the constellation for $M = 2$, 4 and 16 respectively. One can notice that for $M = 4$ it results in the QPSK modulation that was discussed in the previous paragraph.

$$s_m = \sqrt{E_g}(2m - 1 - M)e^{j(\frac{2\pi m}{M} + \frac{\pi}{M})}, \qquad m = 0, 1, ..., M - 1, \tag{2.6}$$

$$P_{BER} = 4(\frac{\sqrt{M} - 1}{\sqrt{M}})(\frac{1}{\log_2 M}) \sum_{i=0}^{\frac{\sqrt{M}}{2}-1} Q((2i + 1)\sqrt{\frac{E_b}{N_0}\frac{3\log_2 M}{M - 1}}), \tag{2.7}$$

M-ary Frequency Shift Keying (M-FSK)

The M-FSK is a frequency modulation that turns a set of $K = \log_2(M)$ input bits into a frequency information. The set of possible frequencies is given by (2.8), where Δf is the frequency separation between two consecutive frequency symbols.

$$f_m = (2m - 1 - M)\frac{\Delta f}{2}, \tag{2.8}$$

Other symbol mapping techniques have been developed and deployed throughout the decades. Among them, one can mention the Minimum Shift Keying whose variant, the Gaussian-MSK[12] has been employed in the GSM [25] telecommunication standard. Remembering that the choice of a given mapping technique depends on the expectation in terms of data rate and probability of error, accurate simulations must be carried out in order to find the appropriate symbol mapping technique. These simulations must include the distortions of the expected channel which can vary in the context of wired, wireless, satellites or underwater communication.

In addition to information signal modulation, it is important to decide how the allocated bandwidth resource will be utilized by the modulated symbols. This is known as channel access techniques that are developed in the next section.

2.2.2 Channel Access Techniques

The channel access techniques have been developed to optimize the usage of the allocated spectral bandwidth under certain conditions. Indeed, the spectral resource is quite a scarce and costly resource that is generally managed by the local authorities and at some point coveted by the other applications. Thus, several techniques have been proposed to increase the overall communication performance given both a spectral resource and a propagation condition, while ensuring the signal integrity. Each of these techniques has some relevant advantages however they may exhibit some shortcomings deeply related to the properties of the channel.

In this section, we will essentially discuss two of those techniques namely, the spread spectrum and the Orthogonal Frequency Division Multiplexing (OFDM) techniques that have been employed in many of nowadays applications. They are also relevant for the PHY waveforms that we will use to validate our approach.

Spread Spectrum Systems

Spread spectrum is a channel access technique in which the information signal is transformed to a signal of a higher bandwidth before transmission. It is declined in two variants namely, the Direct Sequence Spread Spectrum (DSSS) and the Frequency Hopping Spread Spectrum (FHSS). In DSSS systems, the spreading is performed by multiplying the information signal by a known code while in FHSS this code indicates the carrier frequency on which the information signal is modulated. These two techniques both result in a wider signal bandwidth that protects the signal from narrowband interference. Moreover, these techniques show some improvements in the context of multipath transmission.

Equations 2.9 and 2.10 formalize the information signal $S(t)$ and the spread signal respectively (in the context of the DSSS). The spread signals are commonly called chip sequences and should include some properties (e.g. orthogonality) that enable to efficiently differentiate them at the receiver. Indeed, at the receiver, the transmitted symbols are recovered by a synchronization, which implies a multiplication by the synchronized spreading code and detection. Finally, performance vary depending on the channel under consideration.

$$S(t) = \sum_{m=-\infty}^{+\infty} s_m g_T(t - nT_s), \tag{2.9}$$

where s_m is the transmitted symbol sequence and $g_T(t)$ the modulation pulse of period T_s.

$$S(t)s_C(t) = \sum s_n g_T(t - nT_s) \sum c_{PN}(n) p(t - nT_c), \tag{2.10}$$

where $U(t)$ is the information signal and $c_{PN}(n)$ takes the values ± 1 of the desired code. T_c is the duration of a chip and $p(t)$ represents the unit pulse.
On the other hand, FHSS operates by loading the information signal $S(t)$ onto a carrier, the frequency of which is indicated by the spreading code. The transmitted signal remains in such a way at a specific carrier frequency for a period of T_f. Thus, the hopping process can be mathematically formalized through (2.11). At the reception, the signal is de-hopped by combining mixing operation and bandpass filtering.

$$f(t) = f_m, \;\; mT_f \leq t \leq (m+1)T_f. \tag{2.11}$$

To conclude, it is important to note that the spread spectrum technique has been employed in well-known telecommunication protocols and also for many other ad hoc networks. For instance, the IEEE 802.15.4 radio protocol [20] uses the DSSS within one of its PHY definitions that will be further discussed in this document. Furthermore, both the Bluetooth protocol and the UMTS standard leverage FHSS and DSSS respectively.

Orthogonal Frequency Division Multiplexing (OFDM)

Some classes of modulation techniques operate by dividing the allocated bandwidth into a set of subchannels that are modulated independently. This approach is called Frequency Division

Multiplexing (FDM) and results in a lower data rate per subchannel. However, it was shown that this technique reduces the effect of the impulse response of the channel but requires being further improved so as to reduce the Inter-Channel Interference (ICI). In fact, it was also shown that by selecting orthogonal subchannels the ICI effect can be considerably reduced. Thus, the Orthogonal FDM (OFDM) has become a mainstream channel access technique in which the orthogonality of the subchannels is achieved by making all subcarriers be an integer multiple of a fundamental frequency.

In practice, the OFDM is achieved by splitting the incoming complex data symbols (issued from a symbol mapping function) into N parallel streams corresponding to the subcarriers. Then, for each parallel stream, the complex symbols are modulated by complex sinusoids with frequencies corresponding to the subcarriers. The modulated symbols are then added to form an OFDM symbol. In the digital domain, this operation is known as the Digital Fourier Transform (DFT) whose equation is given by (2.12) and the inverse operation is known as the Inverse Digital Fourier Transform (IDFT) given by (2.13). It was shown that this operation can be digitally realized with a Fast Fourier Transform (FFT) or an Inverse Fast Fourier Transform (IFFT) which are two faster (reduced complexity) implementation of the DFT and the IDFT respectively.

$$s_k = \frac{1}{N} \sum_{k=0}^{N-1} S_n e^{j(\frac{2\pi kn}{N})}, \tag{2.12}$$

$$S_n = \sum_{k=0}^{N-1} s_k e^{-j(\frac{2\pi kn}{N})}, \tag{2.13}$$

Several telecommunication standards employ the OFDM channel access technique. Coarsely, at the transmitter the IFFT is used to convert the complex symbol in time domain as aforementioned. Furthermore, a Cyclic Prefix (CP) is appended to each OFDM symbol before transmission. It was shown that the CP enables to protect the signal from the channel impairments. Signal windowing is also part of the transmitter as it enables to squeeze the signal into the spectral recommendation. The receiver consists essentially in synchronization and symbol recovery through a DFT calculation. Many other challenges are encountered at the receiver making it a burden in OFDM design.

OFDM systems will be further depicted through an example later in this document where an OFDM-based telecommunication standard will be discussed. Finally, OFDM has encountered a lot of success and it has been employed in many standards such as the Digital Audio Broadcasting (DAB) [23], the Digital Video Broadcasting (DVB) [24], the Long Term Evolution (LTE) [26] and many others.

2.2.3 Pulse Shaping

The pulse shaping technique has been developed to counter the Inter-Symbol Interference (ISI) damaging effects. Indeed, pioneers transmission systems have faced severe interference effects. These interference were due to the leakage of the energy of the previously transmitted symbols into the current symbol. It ended up with strong distortions which made it very complex to decode the current symbol. To tackle this issue, a first solution has been to decrease considerably the symbol rate so as to tell the received symbols apart. This solution was not sustainable since the trend was fostering to move toward the opposite way, *i.e.* to increase the transmission rate.

As a result, pulse shaping techniques have been proposed to strengthen the information signal prior to the transmission. Say, $g(t)$ is a pulse shape function, $g^*(-t)$ its transform conjugate and $c(t)$ the impulse response of the channel. The received pulse, formalized as $p(t) = g(t) * c(t) * g^*(-t)$ should satisfy the *Nyquist criterion* which states that $p(t)$ must be equal to zero at optimal sampling times. In the next paragraphs, three useful pulse shaping techniques are discussed.

Rectangular Pulse

Rectangular pulse shaping consists in shaping the incoming symbols with a time-domain rectangular filter. It is quite an intuitive shape that unfortunately leads to some important shortcomings in practice. Indeed, the rectangular pulse is difficult to create in time domain because of its rise

time and its decay time. In addition, its frequency response is a sinc function ($\sin(x)/x$) that has a zero amplitude at integer multiples of the symbol rate. Furthermore, the sinc frequency response goes on forever and might interfere with others, which is not allowed by the frequency regulators. It exhibits some second lobes that are only 13 dB lower than the main one. Equation 2.14 gives the time domain rectangular pulse function where T_s is the symbol period.

$$g(t) = \sqrt{\frac{2}{T_s}}, 0 \leq t < T_s, \tag{2.14}$$

From then on, one can argue that a filter with a rectangular frequency response could solve this problem since both second lobes and going forever issues would have been tackled in such a way. Such a filter corresponds to a sinc function in time domain that is unfortunately no more possible to build than a rectangular pulse. Thus, alternative solutions have been proposed to trade-off between these two classes of filters.

Cosine Pulse

Cosine pulse shaping presents some interesting spectral gain as compare to the rectangular pulse. For instance, it has some 10 dB lower side lobes in comparison, with a constant amplitude. Equation 2.15 gives the time domain expression of the pulse.

$$g(t) = \sin(\frac{\pi t}{T_s}), 0 \leq t < T_s, \tag{2.15}$$

Raised Cosine Pulse

The raised cosine filters came up as a modification of the sinc pulse. They have an adjustable bandwidth that can be adapted through a parameter called the roll-off factor. Equation 2.16 gives the time domain representation of the raised cosine pulse where the roll-off factor a ranges from 0 to 1. This equation shows that the filter ranges from a pure sinc function ($a = 0$) to a rectangular function ($a = 1$) depending on the value of a. The most encountered values given to the parameter a are 0.2, 0.4 and 0.6. Figure 2.4 gives both the impulse response and the frequency response of a raised cosine pulse for different values of the roll-off factor a. In practice, the raised cosine filter is implemented as a Finite Impulse Response (FIR) filter. In addition, it requires a minimum number of taps since the filter get better as the number of taps increases.

$$g(t) = \frac{\sin(\frac{\pi t}{T_s})}{\frac{\pi t}{T_s}} \frac{\cos(\frac{a\pi t}{T_s})}{1 - \frac{4r^2t^2}{T_s^2}}, 0 \leq t < T_s, \tag{2.16}$$

Further Digital Signal Processing (DSP) blocks/algorithms can be involved in the definition of a digital transceiver, however covering all these blocks features is out of the scope of this document. For instance, the channel coding and decoding techniques are relevant mechanisms for ensuring the integrity of the transmitted information signal throughout a noisy channel. They are declined into block codes [27] and convolutional codes [28]. The former submit k bits in their inputs and forward n bits in their outputs. The latter induces the notion of memory by considering the preceding bits for computing the ongoing code. Both techniques are usually characterized by a coding rate (r) which denotes the redundancy induced by the coding technique. In addition to channel coding, one can mention the synchronization blocks which purpose is to lock the receiver components with the appropriate processing parameters. They are employed in coherent receivers which are opposed to non-coherent receiver where a blind reception is performed. Synchronization has actually a crucial importance on a given transceiver architecture since its performance can limit the request for data re-transmission in the network. Channel estimation is optionally implemented on receivers. It aims at estimating the channel's distortions by using blind techniques or known sequences that are inserted in the transmitted data.

In the next section, the technologies that are proposed to support these algorithms are discussed. Their choice can have a significant impact on the final product, so it is important to have an idea of the expected performance when considering a given technology.

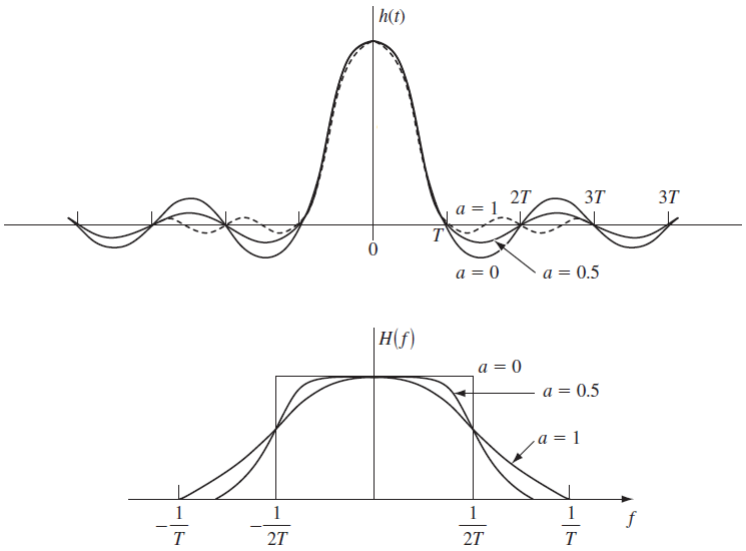

Figure 2.4 – Impulse response and frequency response of raised cosine pulse.

2.3 Digital Communication Technologies

The advent of the digital communication has started a new era for communication systems in general. This paradigm has been supported by the proposal of a set of technologies which purpose is to increase the overall productivity in digital systems designing. In the signal processing domain, these technologies allow implementing most of the algorithms discussed in the previous section and they are often referred to as the baseband technology. As the complexity of the communication systems increases, technologies turn out to be rapidly obsolete and may require to be further enhanced so as to support more complex algorithm implementations. As a result, a bunch of technologies, most of them issued from research programs, for digital systems designing have appeared throughout the decades. These technologies can be classified into two main groups namely, the software technologies and the hardware technologies. They both enable implementing digital systems and are supported by specific design flows.

In this section we briefly review the technologies that are involved into digital signal processing systems, which essentially consist in design flows and running platforms. Designs flows, on the one hand, are generally composed of textual or graphical programming languages and their associated compilers. The platforms on the other hand, represent the physical device on which the application is run. In DSP, the choice of the underlying platform is governed by the target performance. Those performance are diverse, in the sense that they can be estimated in terms of processing speed, platform area, power consumption or flexibility. Further to this, DSP applications are described as computationally intensive, data independent, exhibiting a high level of parallelism and requiring low arithmetic architectures. As a result, different types of platforms have been proposed, some of which tending to be more software-oriented in contrast to the hardware-oriented platforms. Coarsely, software-oriented platforms emphasize on the programmability issues whereas the hardware-oriented platforms focus more on the resulting circuit architecture. Each of these two platforms is depicted in the following two paragraphs.

2.3.1 DSP Software Platforms and Programming Environment

Microprocessors are generally characterized by their Instruction Set Architecture whose instructions are sequentially evaluated on a fixed hardware. They are declined into two types namely, the Complex Instruction Set Computer (CISC) [29] machines that have some complex instruction formats and the Reduced Instruction Set Computer (RISC) [29] machines with regular and simple instruction formats. Microprocessors are widespread electronic components that are employed in

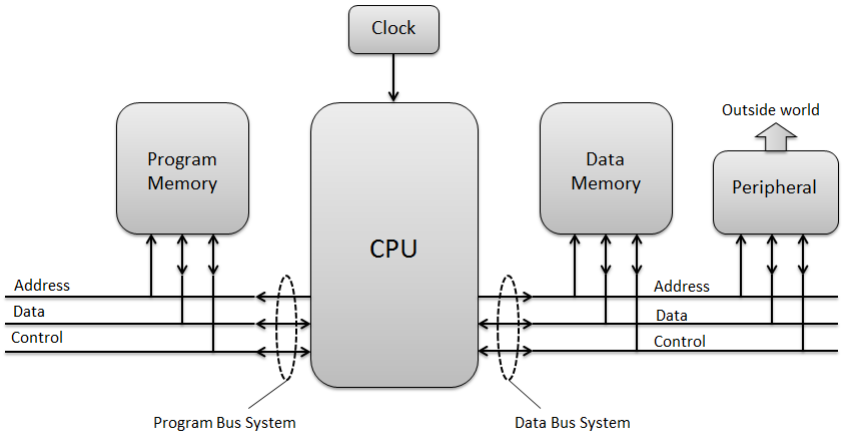

Figure 2.5 – Harvard Architecture.

many industrial fields such as the automotive , the aerospace, the home/building automation or the telecommunication industry. However, their sequential nature can make them unsuitable for an efficient implementation of computationally intensive DSP applications.

These limitations have led in the 80s to the proposal of microprocessors that are optimized for DSP, those DSP microprocessors [30] are capable to perform multiplication and accumulation operations by consuming less power. They rely on the Harvard architecture [30] which separates the data memory and the program memory as shown in Figure 2.5. Indeed, a common data and program memory leads to a memory bottleneck that limits the performance of microprocessors. DSP microprocessors have been enhanced with dedicated complex functions and also with some of the fixed-point operators that are often required in most of DSP applications.

Furthermore, parallelism has been also addressed in DSP microprocessors by introducing instruction level parallelism through instruction pipelining or *Very Long Instruction Word* (VLIW) [31] architecture for instance. Thus, DSP microprocessors have become an interesting alternative for implementing DSP applications and they are usually associated with General Purpose Processors (GPPs) or other type of general microprocessors, such as the Advanced RISC Machine (*ARM*) microprocessors, to implement the rest of the network stack.

As mentioned before, software platforms offer a good trade-off between programmability and performance. Thereby, DSP microprocessors and general purpose microprocessors are supported by software tools that enable specifying and implementing an application from a high-level of abstraction with an *a priori* knowledge of the underlying architecture. Those tools usually consider a C-based specification of the application and rely on a compiling framework to produce a run-time code that is optimized for the microprocessor architecture. Moreover, some of these compilers are featured with some *intrinsic* functions that are used to optimally target, from the C-based description, a dedicated resource.

2.3.2 DSP Hardware Platforms and Programming Environment

Whilst DSP microprocessors have enabled achieving certain performance in terms of application throughput, they still restrict the application designers to a pre-defined architecture. This restriction limits the achievable level of parallelism, which makes DSP microprocessors unsuitable for several high data rate DSP applications. FPGA and ASIC technologies enable customizing circuit architecture for a given application. From a DSP point of view, these platforms allow defining the architecture that are tailored to an application, thereby optimized in terms of processing speed, resource area and power consumption. FPGAs can be coarsely presented as a set of programmable logic components, interconnections and input/output (I/O) pins at their outer edge as shown in Figure 2.6. Nowadays, they are featured with large memory and data processing resources. Logic

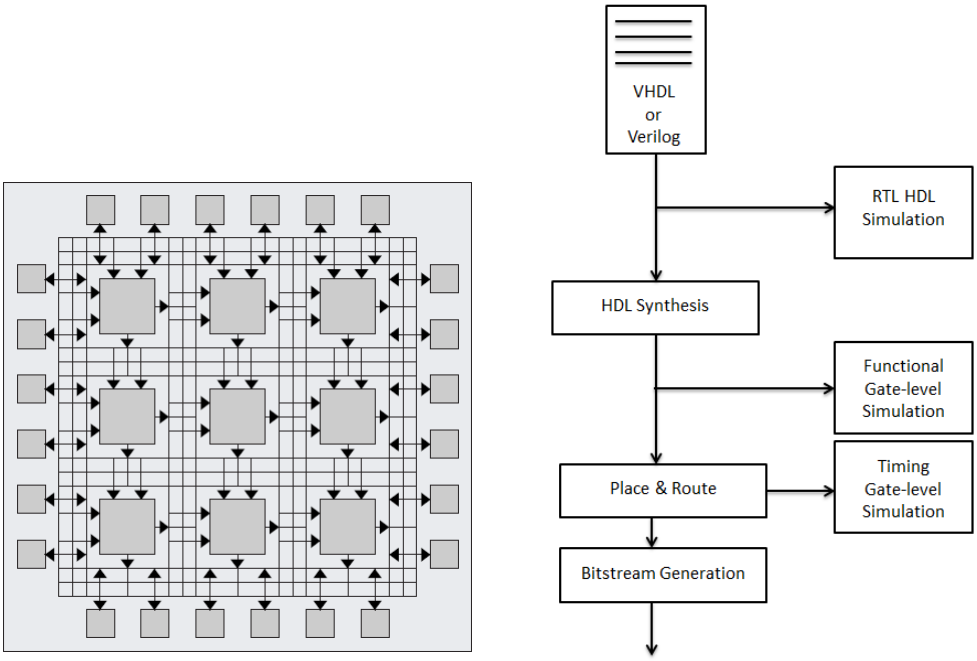

Figure 2.6 – Abstract FPGA representation (left) and a typical FPGA design flow (right).

components can be programmed to implement simple or complex DSP functions while the interconnection is used to program a desired functionality. FPGAs are declined into different types based on the way they are configured. Actually, FPGAs are programmed via configuration files stored into memories. Thus, SRAM-based FPGAs, FLASH-based FPGAs or antifuse-based FPGAs can be found. However, this programmability at the architectural level implies an overhead in terms of power consumption and speed as compared to ASICs fabrics.

ASICs are fully customized architectures that are generally employed for large market technologies. They consist in providing a hardware that is suitable for a given DSP application. By doing so, the provided hardware (referred to as an ASIC) can be optimized to meet all the requirements regarding processing speed, resource area and power consumption. As a result, ASICs are not flexible fabrics and they are usually associated with microprocessors to perform some others task such as control.

The design methodologies employed to prototype an FPGA-based or an ASIC-based DSP application usually relies on Hardware Description Languages (HDLs). Those languages such as Verilog or VHDL are the main entry points for most of the available FPGA and ASIC synthesis tools. HDLs are programming languages that allow an accurate description of circuit architecture at the RTL-level for instance. In the context of FPGA, a HDL description is compiled down to a bitstream which is used to program the FPGA as shown in the design flow illustrated in Figure 2.6. An ASIC development methodology is quite similar to the FPGA ones, however it requires additional steps for the final circuit manufacturing.

2.4 Flexible Radios

In the previous section we have introduced the DSP concept by discussing some of its core principles together with its underlying technologies. A high degree of flexibility is now expected in DSP applications since communication protocols must support multiple modes and transceiver must support multiple communication protocols. Flexible radios are then employed when it comes to incorporate adaptive capabilities. Such radios must be capable for instance to switch between different configurations so as to take advantage of their operating environment. This approach aims at increasing the spectral efficiency or can be used to save the overall energy. Indeed, the

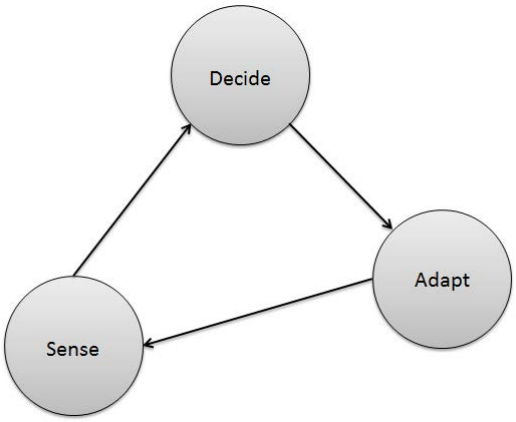

Figure 2.7 – Cognitive cycle composed of three major states.

spectral resource which is managed by the local authorities tends to be scarce because of all the operating radios. One way to tackle this issue is to undertake a cognitive usage of the spectrum by these radios.

2.4.1 Cognitive Radios

Cognitive Radios (CRs) [5] can be defined as radios that are aware of the context in which they are being operated. They observe a set of environmental parameters before selecting the optimal communication scheme. The cognition refers to the fact that cognitive radios monitor their operating environment in order to improve their performance. A cognition cycle as shown in Figure 2.7 is usually employed so as to illustrate the main features of a cognitive radio. It is composed of three major states, namely the *Sense* state where the system monitors a set of metrics, the *Decide* state where the system makes up a decision on which configuration to select and the *Adapt* where the system is reconfigured into the selected configuration. In the following two paragraphs, we will discuss two implementations of a cognitive radio that are the Spectrum-Aware radios and the Multi-Standard radios.

Spectrum Aware Systems

The research on cognitive systems has been essentially oriented toward spectrum aware radios. As spectrum turns out to be quite a scarce resource, there has been a growing need for spectrum management techniques so as to take advantage of the underutilized allocated spectrum. Such opportunistic systems aim at achieving better performance from a clever usage of the spectrum. For instance, the white space in the TV band (470 to 790 MHz in Europe) has gathered a lot of interest in the community and an important part of the research in cognitive systems focuses on how to exploit these spectral bands without disturbing the incumbents. The detection of an incumbent is performed by using different signal processing methods that can be found in the literature [32][33][34]. Among those methods, one can mention the *energy detector* [32] which typically detects the presence of an incumbent by thresholding the energy that is sensed in the channel. A relevant detection technique that is also discussed in the literature is the *cyclostationnarity detector* [33][34][35] which detects digital modulations through their cyclostionnarity properties. Indeed, most of the digital modulations imply cyclic frequencies in the transmitted signal due to the periodical digital computations performed over random source data. Thus, the *cyclostationnarity detector* aims at finding a periodicity within the mean and the autocorrelation function of a signal $x(t)$. Equations (2.17) and (2.18) give the mean and the autocorrelation functions of a signal $x(t)$ and (2.19) gives the periodicity of these functions.

$$m_x(t) = E[x(t)] \tag{2.17}$$

$$r_{xx}(t) = E[x(t - \frac{\delta}{2})x^*(t + \frac{\delta}{2})] \tag{2.18}$$

$$m_x(t + T_0) = m_x(t) \qquad r_{xx}(t + T_0, \delta) = r_{xx}(t + T_0). \tag{2.19}$$

Multi-Standard Systems

Multi-Standard systems [36][37] can also be viewed as another type of cognitive systems where the cognition is related to the capability to operate with different telecommunication standards. Indeed, several telecommunication standards were released in order to fulfill the increasing demand in terms of data rate. However, given an application (voice, data, video...), the required data rate may vary in the sense that a high data rate standard may not be appropriate to transmit low data rate signal such as voice. It would be more efficient in such cases to switch to a lower data rate standard before initiating the communication. Furthermore, owing to the limited coverage that is offered by the telecommunication operators, multi-standard systems enable, given a geographical region, virtualizing global network coverage. An example of such scenario that we often face is the switching between mobile standards operated by most of the current mobile phones in order to ensure a permanent network access to the users. From an implementation perspective, this is currently achieved by integrating a dedicated chip for each standard and then using a software control to switch at run-time between standards. Some mechanisms are provided to handle the handover between the standards as well. This approach implies a duplication of radio chips, which is not economically sustainable in the long term.

2.4.2 Adaptive Coding and Modulation (ACM) Technique

The Adaptive Coding and Modulation (ACM) [38] technique is used in many of the recent radio protocol. Its main purpose is to dynamically improve the overall spectral efficiency, *i.e.* increasing the number of bit per second in a given spectral band. This technique employs some cognition to tailor both the coding and the modulation schemes to the environmental conditions. Thus, depending on the value of the *SNR*, the system can select which coding or modulation technique to employ. As a result, such a system would be capable to higher the data rate when *SNR* is high and lower it when *SNR* is low.

ACM has brought some interesting perspectives by enabling a dynamic exploitation of the spectrum. However, it also came up with some new challenges by making DSP, in the context of ACM, more data dependent. Indeed, the dynamic coding and modulation information are now embedded in the transmitted data frame so as to inform the receiver which decoder or demodulator to employ. Thus, these information must be decoded prior to data decoding. It leads to some implementation challenges, which can be addressed with either multi-mode functions or reconfigurable functions. In this work, we have studied the possibility of implementing ACM-based waveforms on FPGA platforms.

2.5 Software-Defined Radios

Software-Defined Radio (SDR) is the generic terminology that is employed to depict the flexible DSP architectures with very high reconfiguration capabilities so as to adapt themselves to various air-interfaces. This concept was first introduced by Joseph Mitola and then turned out to be a sustainable implementation of Cognitive Radios [10] [11]. Indeed, in the early *90s* the unavoidable shift from hardware radios to software intensive radios was portended. In his paper, Dr. Mitola also pointed out that such software radios should consist in a set of DSP primitives and a metamodel system for combining these primitives into communications systems functions such as transmitters, channel models or receivers. In the following sections, we propose an insight of SDR by discussing its major principles.

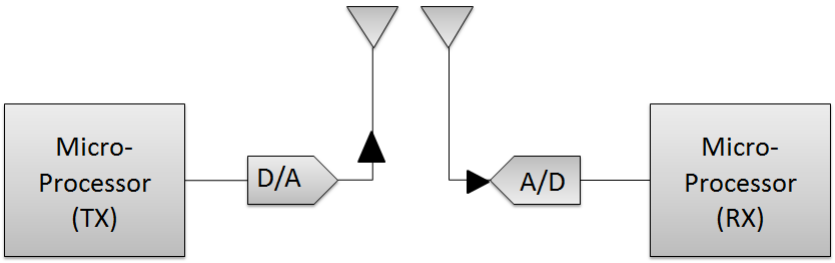

Figure 2.8 – Ideal Software-Defined Radio (SDR).

2.5.1 Motivations and Main Features

Motivations

SDR actually came up from a military need in insuring the inter-operability of the equipment through platforms that could run any type of waveforms by simply being reprogrammed [38]. Indeed, military applications have quite a long life cycle and it is crucial to maintain them and insure their inter-operability with the upcoming applications. Thus, an SDR platform could enable to deploy various types of application in a given geographical location and make it possible to communicate whatever the situation. Say a jammer is trying to disturb the communication, it would be easy for SDR platforms to agree on a different channel/protocol and then switch the communication into that new configuration.

Later on, SDR has appeared to present several advantages. Indeed, it allows among others, to make an efficient use of resources under a key metric such as low-power or high-throughput for instance. Opportunistic frequency reuse can be easily sketched with an SDR platform so as to virtually increase the amount of available spectrum. Moreover, SDR can significantly reduce the equipment obsolescence in the sense that SDR can be upgraded so as to support the latest communications standards. Finally, SDR has opened some perspectives for the research and development by enabling with a certain comfort to implement many different waveforms for real-time analysis.

Main Features

An ideal SDR (Software Radio) can be illustrated as in Figure 2.8. In this ideal representation, the signal is converted at the transmitter (TX) antenna and receiver (RX) antenna by a Digital to Analog Converter (DAC) and an Analog to Digital Converter (ADC) respectively. Such architecture could, in theory, support any types of waveforms since it could be easily reprogrammed for a desired waveform implementation. Furthermore, this approach ensures both the programmability and the portability of the SDR solutions over multiple microprocessor-based platforms. Indeed, it would merely suffice to recompile the same application code for the other platforms.

The ideal SDR would have been the holy grail for digital radio systems however some limitations appear when it comes to the practical requirements. Actually, digitizing the signal right after the antennas requires high sampling frequency ADCs and DACs technologies capable to support the high rate incoming data stream. ADCs and DACs are essentially characterized by their bandwidth, data precision and sampling frequency [39]. Sampling in an ideal SDR context would require the ADCs to operate at up to twice the carrier frequency when sub-sampling technique is not employed.

Furthermore, microprocessors exhibit some limitations that are related to the power consumption and the achievable throughput. As aforementioned, microprocessors evaluate the instructions in a sequential order, which limits the achievable parallelism. However, most of the DSP algorithms are highly parallel and require computation intensive solutions. Whilst programmability is well addressed when considering microprocessors, their overall performance in terms of computation speed does not make them such a good candidate for an ideal SDR implementation. Timing performance is then a critical aspect for microprocessor-based SDRs.

Further to this, power consumption is also limiting the development of microprocessor-based

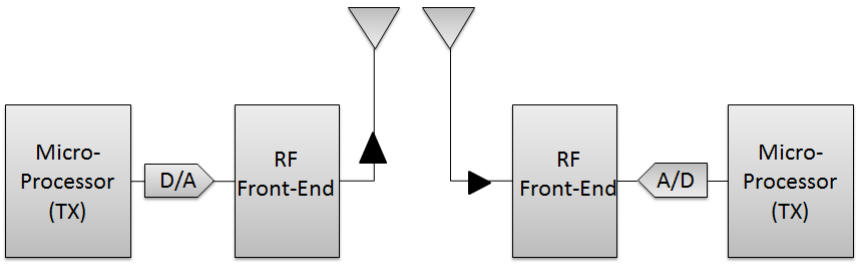

Figure 2.9 – Realistic Software-Defined Radio (SDR).

SDR. It is known that microprocessor-based solutions consume much more power compare to hardware counterparts. Thus, in such a context where some restrictions exist on power consumption, microprocessors would be unsuitable for SDR implementation. A typical illustration of such a power constrained SDR could be in cellular networks where base stations could afford a greedy (in terms of power) SDR implementation whereas mobile phones are extremely constrained in terms of power. However, many efforts are being done so as to reduce the power consumption at the base station. In summary, the ideal implementation of SDR is dramatically limited by the technologies that are available on the market. Nonetheless, research on SDR domain has identified three major aspects that must be properly tackled when it comes to implementation. Thus, *programmability*, *portability* and *reconfiguration* are the key elements of an SDR and most of the proposal emphasize on at least one of those three features.

Given the limitation factors of the ideal SDRs, the proposals in this domain have so far consisted in inserting an agile RF front-end after the antennas and building the baseband processing as software-defined as possible. Thence, the SDR solutions that are encountered in the literature use a software intensive approach to perform most of their baseband computation such as filtering, modulation, channel equalization and so on. The mainstream SDR platform can be illustrated as in Figure 2.9 where the challenge consists in pushing the converter as close as possible to the antennas.

Finally, the implementation of SDR platforms is subject to exploitation constraints. Thus, some implementations are more applicable to laboratory conditions where space and power-consumption are not that critical. These implementations rely on General Purpose Processors (GPPs) which offer maximum flexibility and easy development flow while suffering from low-throughput and high power-consumption. On the other hand, SDRs which rely on signal processing specific hardware such as FPGAs, are more suitable for high data rate and resource-constrained applications. In the following section, we give an insight of SDRs platforms by emphasizing on their underlying architecture technology.

2.5.2 Survey of SDR Platforms

Since the advent of the SDR concept, several platforms intended to implement SDR waveforms have been proposed. These platforms aim at providing a hardware environment for programming SDR waveforms while making an abstraction of the underlying hardware. Such platforms can be classified into three major groups depending on their hardware resources [40] [41]. Thus, some platforms were developed around a single or a cluster of GPPs. Such platforms do not consider constraints like space or power consumption. Another group of platforms were developed in a co-processing style where a single GPP or a general microprocessor, which provides the control and the upper layers, is augmented with accelerators for the PHY requirements. The third group considers programmable hardware fabrics as the central components on the platform.

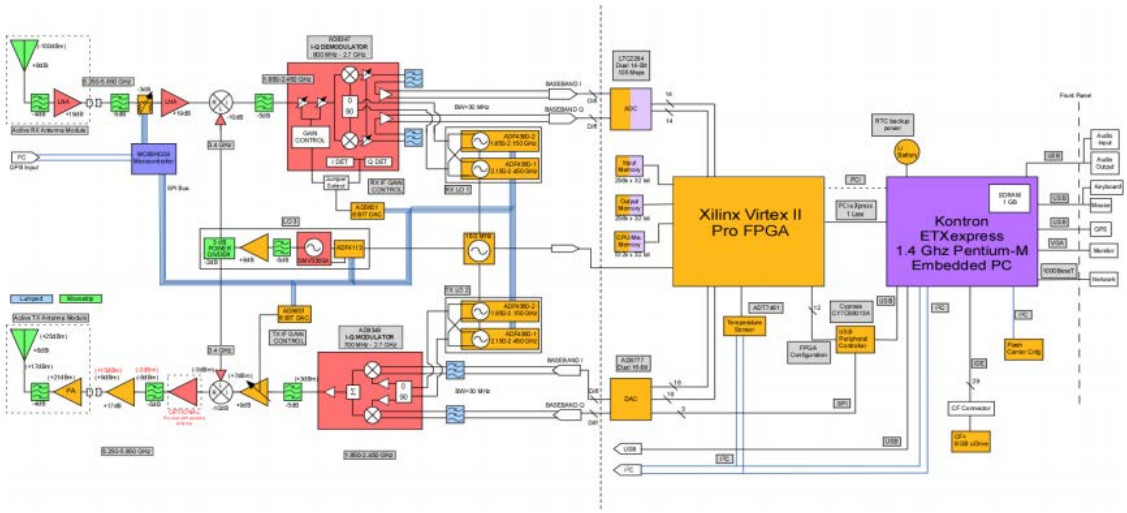

Figure 2.10 – KUAR System Diagram.

GPP-based SDRs

GPP-based SDR platforms offer a high flexibility and an easy development flow. They are suitable for laboratory environment where the size and the power consumption are not a major concern. They enable rapid prototyping of new waveforms by using specification languages such as C. However, latency is a major concern for such platforms. Indeed, GPPs are more suitable to work with block of data rather than on one sample at the time. Furthermore, the operating system introduces some latency which makes GPPs inappropriate for certain real-time applications.

The most popular GPP-based SDR platform is certainly the pair USRP/GNU Radio. USRP [42] stands for Universal Software Radio Peripheral and is employed as an RF front-end. It is composed of ADC/DAC which convert the signal from/at the intermediate frequency and an FPGA which converts the signal form the intermediate frequency down to the baseband. GNU Radio on is software framework that enables the baseband processing on GPPs, typically on a host computer. It provides a Graphical User Interface (GUI), referred to as the GNU Radio Companion (grc), which enables modeling dataflow graphs by using a block-based approach. The pair USRP/GNU Radio has encountered a lot of success and it has been intensively used for demonstration purpose in the SDR community.

Co-Processor based SDRs

Co-processor based SDR platforms came out from the limitations and unsuitability of the GPP-based SDR for real-time and low power implementation. They aim essentially at reducing the platform power consumption while increasing the achievable throughput. Such platforms accelerate the signal processing of the GPP by considering additional resources referred to as accelerators. On these platforms, GPPs are sometimes substituted for general microprocessors such as the ARM family microprocessors and the accelerators vary between FPGAs, DSP microprocessors, Graphics Processing Units (GPUs) or a combination of these technologies. This approach is often employed to develop SDR platforms, however it makes the programming model dependent on the platform.

The Kansas University Agile Radio (KUAR) [43], shown in Figure 2.10, is an SDR development platform whose DSP part is composed of an embedded PC and an FPGA. It was designed to address the need in wireless networking and radio frequency research. The Imec ADRES [44] is an SDR platform that is built around a main CPU and the ADRES accelerator which is used for signal processing and which also leverages data parallelism. The NXP EVP16 [45] is built around an ARM processor which provides the control and the LINK/MAC layers. It is featured with a conventional DSP and several hardware accelerators for the signal processing. The Tomahawk SDR chip [46]

was developed by the University of Dresden and uses two Tensilica RISC processors for control and eight DSP microprocessors for the signal processing. Along the same lines, the Embb [47] is a generic hardware and software architecture dedicated to dataflow applications. To this aim, it combines some DSP units and a General Purpose Control Processor. Each of these DSP units is dedicated to a family of signal processing algorithms and the available set of DSP units include an interleaver, a general purpose modulator, a general purpose interface (up to 4 ADCs and DACs) as well as a general purpose vector processor.

Reconfigurable Hardware-based SDRs

As mentioned before, hardware fabrics such as FPGAs or ASICs enable to gain more performance compared to DSP microprocessors. Thus, they have gathered a lot of interest in the SDR community when it came to increase the computation power available on SDR platforms. As a result, different architectures which employed specialized hardware fabrics were proposed. However, one can argue that this approach takes the SDR concept far away from its initial goal by decreasing the programmability and making the portability of the solutions more burdensome.

The Magali SDR chip [48] is a platform that was developed by the CEA-Leti for telecommunication demonstration purposes. It has a Network-on-Chip configuration controlled by an ARM processor. It is featured with reconfigurable IPs for OFDM, decoding and de-interleaving functions. The ExpressMIMO SDR platform [49] was developed as a configurable units approach on an FPGA by EUROCOM. It targets essentially MIMO implementations and relies on the OpenAirInterface open-source framework. The WARP SDR platform [50] was developed by the Rice University as an open SDR platform programmed entirely in VHDL (RTL). It is built around an FPGA fabric and supported by a community whose goal is to offer some open source implementations on the platforms. The Nutaq platform [51] is an example of commercial platforms for FPGA-based SDR development. It is also presented as supporting MIMO transceivers and the WIMAX protocol.

2.5.3 SDR Design Methodologies

The programmability constitutes a major factor in the rise of SDR. Indeed, SDR has come to maturity and a common development process is the need of the hour. In the previous section, we have discussed different SDR platforms and we have also highlighted that one important thing while considering these platforms is the designing methodology that they offer. GPP-based SDRs are the easiest to program since they are usually programmed in C-based languages and leverage a well-supported compiling framework. Heterogeneous SDR platforms like co-processor based and reconfigurable hardware based platform are more complex to program because of their programming model which is tailored to their architecture. This also limits the portability of an SDR application, specified for these platforms, on a different platform.

The lack of a common flow to specify SDRs has implied two mainstream approaches. On the one hand, some research programs focused on defining standards for SDR development so as to unify the development approaches. On the other hand, some researches have addressed the PHY layer implementation by proposing programming language to model and implement PHYs in the context of SDR. In the following paragraphs, we discuss the SDR standards and SDR-PHY languages that can be found in the literature.

SDR standards

SDR standardization proposals can be viewed as middlewares which provide an interface between the hardware and the application. The underlying idea is to propose Application Specific Interfaces (APIs) through which an SDR can be specified, while giving an emphasis on the portability of the solution on different platforms. Standardization was supported by the US Army essentially and two major works have been realized in this domain. First on is the Software Communication Architecture (SCA) [52] that was motivated by the diversity of the waveforms employed by the US Army. The stated goal of the SCA was the development of a standard to facilitate the reuse of waveform code between different radio platforms. SCA specifies how waveform components are defined, created, connected together, and the environment in which they operate. To this end,

SCA defines a Real-Time Operating System (RTOS) that manages the hardware, a middleware layer which takes care of connecting different parts of the radio and handles data transfer between them, a set of interface definition language (IDL), an eXtensible Markup Language (XML) based ontology to describe all the components that make up a radio and how these components are to be interconnected and finally APIs for the many frequently used interfaces.

Another attempt for SDR standardization is the Space Telecommunication Radio System (STRS) [53] developed by the NASA. Indeed, satellites and deep space missions have quite a long life cycle and could leverage SDR to extend the life of a mission. NASA investigated on SDR and the previously mentioned SCA turned out to be too heavy for NASA missions' applications. STRS explicitly addresses the signal processing modules built upon ASIC or FPGA fabrics and employed GPP for control only. STRS relies essentially on APIs that handle control, application setup, memory, messaging and timing.

SDR PHY languages

The second set of approaches consists of textual/graphical languages which purpose is to specify and implement the PHY of an SDR. Given that SDR was thought as software-centric platform, these languages mainly target GPP or DSP-based SDRs. The Waveform Description Language (WDL) [54] is an example of such languages that was proposed by the Programmable Digital Radio (PDR) research project in the UK. It enables to implement and SDR-PHY from a hierarchical decomposition. Each processing element is viewed as a block within which a state machine locally handles both scheduling, thanks to handshake protocols, and communication with the other blocks. SPEX [55] is language developed by the Michigan University SDR Group to specify SDR-PHYs on a Single Instruction Multiple Data (SIMD) processor by using vector data type like Matlab and also data types borrowed from SystemC. SPEX is declined into three sub-languages namely, the Kernel SPEX to define the processing algorithm, the Stream SPEX to handle both dataflow and interconnection and finally the Synchronous SPEX for real-time constrains consideration. DiplodocusDF [56] is a modeling language that was proposed for implementing SDR PHYs on software-based platforms. It leverages a Unified Modeling Language (UML)-like representation to model the SDR PHY and generates an executable to be run on a software-based platform. The Prismtech Spectra Core Framework [57] is a SCA-compliant framework that supports the deployment of waveform components on any mix of General Purpose Processor (GPP), DSP microprocessors and FPGAs. In this framework, FPGA functions are essentially programmed in VHDL-RTL. In the same way, the Platform and Hardware Abstraction Layer (P-HAL) [58] aims at designing specific radio applications independently of the hardware context. The underlying approach consists in abstracting the hardware platform by software functional units. Thus, it manages radio process real time constraints, processing elements communication issues and enables the software functional units to be configured. The P-HAL defines four services namely, the *BRIDGE* that handles real time constraints, the *SYNC* that synchronizes the concurrent processes, the *KERNEL* schedules the software functional units and the *STATS* analyses the statistics of the functional units.
SDRPHY [59] is another language that was proposed to cover the *sample-to-bit* part of the SDR. It enables an XML-based description of the waveform and provides an *interpreter* as an interface between the description and a specific SDR implementation. This approach makes it possible to allow multiple applications to operate on different SDR hardware. The SDRPHY language aims at satisfying the following goals:

— Completeness: most of the waveforms should be describable using the language.
— Lightweight: the interpreters should require low processing and memory.
— Consistency: similar functionality should not require different descriptions.
— Compactness: low number of keywords to support the description.
— Accessibility: making the language open to any user wishing to utilize or improve.

The GNU Radio [60] framework is an environment for designing GPP-based SDRs. It is featured with a GUI referred to as grc, which enables to draw the flow graph of an SDR. The GNU Radio's major components are:

— A framework of arbitrary signal processing blocks that can be connected together.
— A scheduler to sketch the processing and the transfer of the data.

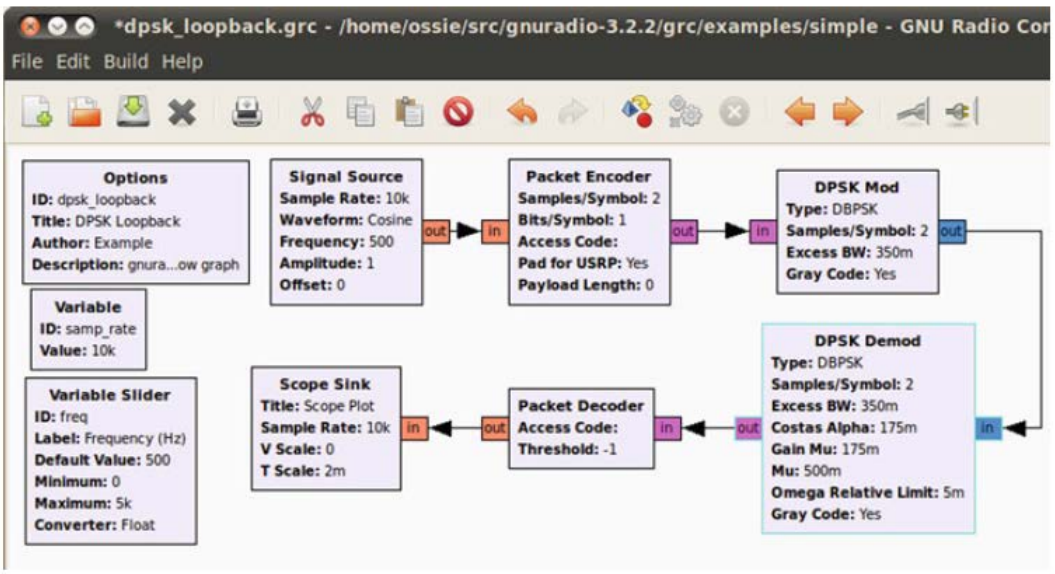

Figure 2.11 – GNU Radio Companion GUI.

— C++/Python infrastructure to build the flow graph.
— A GUI called *grc* to draw the flow graph.
— An interface to commercial front-ends such as USRP.

Most of the signal processing blocks are implemented in C++ and they can be interconnected through a Python-based infrastructure. Indeed, GNU Radio is intended for rapid prototyping of radio waveforms. By using C++/Python as entry point, it enables the waveforms to be rapidly developed and easily integrated into the framework. Programmability and portability is ensured in such a way on different GPP-based platforms and GNU Radio has been extensively adopted by the SDR community especially by the academic research groups. The GUI-based approach illustrated in Figure 2.11 has also contributed in SDR popularization by providing an intuitive way to develop the flow graphs of SDRs.

The GNU Radio framework leverages an interface to commercial front-end. Thus, one can implement a whole transceiver chain and perform some real-time analysis. As mentioned before, the pair GNU Radio/USRP is extensively used for SDR demonstration purpose. However, GNU Radio remains unsuitable for implementations which require low power performance. Whilst, programmability and portability is ensured, an open question would be how to port such solution onto low power signal processing hardware? Table 2.1 summarizes the previously discussed languages for SDR. It details each of the features of the aforementioned SDR design methodologies, namely their programmability, their flexibility as well as their portability.

2.6 FPGA Platforms for SDR

FPGA is a promising technology that is expected to play a key role in the development of SDR experimental platforms. It has so far been employed as a hardware accelerator on which computation intensive functions can be deployed at run-time. This has enabled low timing to be achieved while making the platform more complex to program because of its heterogeneity. The WARP SDR platform [50] is an example of SDR platform that has centered the processing on an FPGA fabrics. It employs a large FPGA device that can support complex waveform implementation. However, WARP-SDR is programmed entirely with a low-level language that is the VHDL language. This approach does not ensure the programmability of the platform since the program model relies on low level language that requires a strong knowledge in circuit designing. Thenceforward, we are facing a trade-off between programmability and performance by using the FPGA technology. In

Table 2.1 – Summary of state-of-the-art SDR languages.

Proposals	Programming language	Flexibility	Portability
WDL [54]	UML-based representation	Constrained Specifications	n/a
SPEX [55]	Subset of C++	n/a	DSP (VLIW & SIMD)
DiplodocusDF [56]	UML-based representation	Constraint profile	GPP & DSP
P-HAL [58]	Object-oriented C++	Real-time adaptation	GPP & DSP & FPGA
GNU Radio [60]	C++ & Python	Compile-time flexibility	GPP
Prismtech Spectra Core [57]	Model-based design & RTL IP Cores	n/a	GPP & DSP & FPGA
SDRPHY [59]	XML-based description	Configuration files	GPP & FPGA

this thesis, we have investigated the feasibility of programming SDR on FPGA platforms by raising the level of abstraction of the programming model.

2.7 Conclusion

In this chapter, we have reviewed the digital radios and introduced the SDR paradigm. Digital radios have been depicted in both signal processing requirements as well as the technologies that are involved. SDR, which implements digital radios, was presented as a promising technology that has motivated a lot of research since its introduction. Its three major aspects are its programmability, its portability and its reconfiguration capabilities on the end platform. Most of the proposals have traded-off between these three aspects and it winded up with SDR platforms lacking either of programmability, portability or reconfiguration.

In the forthcoming two chapters, we will first depict two PHYs that have considered in this research work for demonstration purpose. Afterwards, we will discuss some approaches that aim at improving the programmability in PHY designing by raising the level of abstraction. They enable to speed up the development process by hiding the complexity of programming a waveform.

Chapter 3

The Waveforms of Interest

Contents

3.1 Introduction 40
3.2 The IEEE 802.15.4 Standard 40
3.2.1 ZigBee Generalities 40
3.2.2 The IEEE 802.15.4 PHY 40
3.2.3 State of the art of IEEE 802.15.4 SDR transceivers 43
3.3 The IEEE 802.11 Standards 45
3.3.1 Generalities 45
3.3.2 The IEEE 802.11a Physical Layers (PHYs) 45
3.3.3 State-of-the-art of IEEE 802.11a/p SDR transceivers 48
3.4 Conclusion 49

3.1 Introduction

In the first chapter, we have briefly introduced some of the DSP principles together with one of its applications which is the Software-Defined Radio (SDR). It was shown that SDR is meant to operate with different Physical Layers (PHYs), which implement by definition some DSP algorithms. An implementation of such a DSP algorithm in the telecommunication domain is referred to as a waveform and it is usually well-specified by the released standards [20][21][22]. Such standards are issued from international institutions like the Institute of Electrical and Electronics Engineers (IEEE) [61] or the European Telecommunications Standards Institute (ETSI) [62] or the 3GPP. Their stated goal is to propose some standards that will ensure the interoperability of the associated equipment worldwide. In this chapter, we will review two of these standards, namely the IEEE 802.15.4 [20] and the IEEE 802.11 [21], from a PHY perspective. Our research work focuses on PHY layers modeling and implementation in the context of SDR. Coarsely, the IEEE 802.15.4 is intended for low data rate applications where energy consumption is the key metric, whereas the IEEE 802.11 supports higher data rate applications where the achievable throughput is the key metric. Each of these standards will be depicted by starting from generalities down to compliant transceiver architectures that were proposed in the literature with respect to SDR. Sections 3.2 and 3.3 discuss the IEEE 802.15.4 and IEEE 802.11 respectively. Conclusions on the implementation of the two waveforms are drawn in Section 3.4.

3.2 The IEEE 802.15.4 Standard

The IEEE 802.15.4 standard specifies both the MAC and the PHY layers of the IEEE 802.15.4 protocol. It is a standard for low-rate Wireless Personal Area Networks (WPANs) [63], which are used to convey information over relatively short distances. The ZigBee technology actually came up from an increasing demand in a low consumption and low data rate wireless communication standards. The specification of the standard has enabled among others achieving the interoperability between equipment released by different vendors and therefore the ZigBee technology has been largely deployed for several applications.

3.2.1 ZigBee Generalities

The ZigBee protocol enables to reach up to 250 kbit/s of data rates for relatively short (around 100m) distances with low power consumption. ZigBee is mainly promoted by the *ZigBee Alliance* [64] which is an open non-profit association of members that was established in 2002. ZigBee has been deployed for several applications which implement some Wireless Sensor Networks (WSN) [65] [66] such as:

- Building automation: to offer interoperable products that enable secure and reliable monitoring and control of commercial building systems.
- Remote control: to provide a global standard for advanced, greener and easy-to-use RF remotes.
- Health care: to offer a global standard for secure and reliable monitoring and management of non-critical and low-acuity healthcare services.
- Smart energy: to enable products that monitor, control, inform and automate the delivery and use of energy and water.
- Telecom services: to provide the means for a wide variety of added-value telecom services including information delivery, location-based services or secure mobile payment.

ZigBee was defined to operate on the Industrial Scientific and Medical (ISM) radio bands depending on the geographical location. However, the ZigBee technology can be deployed around the 2.4 GHz ISM band worldwide. In the following section, we discuss the features of the ZigBee PHY.

3.2.2 The IEEE 802.15.4 PHY

Our research focuses on PHY layers modeling and implementation, therefore we will only discuss the baseband PHY layer in the subsequent lines. The PHY layer of the IEEE 802.15.4 Standard is responsible for the activation and the deactivation of the transceiver, the Energy Detection

Table 3.1 – Frequency bands and data rates.

PHY	**Frequency band**	**Spreading parameters**		**Data parameters**		
	(MHz)	***chip* rate** (kchip/s)	**Modulation**	**Bit rate** (kbit/s)	**Symbol rate** (ksymbol/s)	**Symbols**
868/915	868-868.6	300	BPSK	20	20	Binary
	902-928	600	BPSK	40	40	Binary
868/915	868-868.6	400	ASK	250	12.5	20-bits PSSS
(optional)	902-928	1600	ASK	250	50	5-bits PSSS
868/915	868-868.6	400	O-QPSK	100	25	16-ary Orthogonal
(optional)	902-928	1000	O-QPSK	250	62.5	16-ary Orthogonal
2450	2400-2483.5	2000	O-QPSK	250	62.5	16-ary Orthogonal

(ED) within the current channel, the Link Quality Indicator (LQI) for the received packets, the Clear Channel Assessment (CCA) for the Carrier Sense Multiple Access with Collision Avoidance (CSMA-CA), the channel frequency selection and finally the data transmission and reception. In this work, we emphasize more on the data transmission and reception process, especially on the underlying architecture. As aforementioned, the PHY was specified to operate in several location-based ISM bands namely, 868-868.6 MHz in Europe, 902-928 MHz in North America and 2400-2483.5 MHz worldwide. Each of these channels exhibits some drawbacks together with some advantages regarding the signal propagation. The underlying modulations must then be robust enough to protect the transmitted signals from those channel distortions.

Table 3.1 gives the theoretical bit rates according the frequency bands. It also provides the type of modulation that is employed in each of these bands. The frequency bands are divided into sub-channels defined in the standard. To this end, 32 bits are allocated for the identification of the channels through a dedicated paging and number management system. Among those 32 bits, the 5 most significant bits (MSB) designate the page and the 27 remaining bits designate the channel. This technique enables forecasting some extensions of the standard to other frequency bands as illustrated in Table 3.2 where some pages are reserved for extension purpose; and for each page the supported frequency bands are given. On page zero for instance, the channels center frequencies can be formalized as:

$$F_c = 868.3 \quad \text{MHz for k= 0} \tag{3.1}$$

$$F_c = 906 + 25(k-1) \quad \text{MHz for k= 1,...,10} \tag{3.2}$$

$$F_c = 2405 + 5(k-1) \quad \text{MHz for k= 11,...26} \tag{3.3}$$

where k is the channel number.

In a ZigBee transmission process, the useful data are gathered into packet or frame. Each frame includes some data fields that are appended for various purposes. The IEEE 802.15.4 defines the PHY frame, also called PHY protocol data unit *PPDU*, which consists of a synchronization header (SHR), a PHY header (PHR), and a variable length payload as shown in Figure 3.1. The SHR allows the receiving devices to synchronize and lock onto the bit stream. In the 2400-2483.5 MHz O-QPSK PHY definition, the synchronization field includes *Preamble* and the Start-of-Frame Delimiter (SFD), which are used to perform the synchronization at the sample-level and at the symbol-level respectively. A symbol in the ZigBee protocol corresponds to a group of four bits. Thus, there are 16 different symbols that are numbered from 0 to 15. The synchronization field consists of eight symbols 0 and the SFD field consists of symbols 10 and 7. The PHR field informs the length of the current frame on 7 bits plus one reserved bit. The variable payload (DATA) carries the useful information bits issued from the upper layers and its size ranges from 4 to 128

Table 3.2 – Channel page and channel number.

Channel page (decimal)	**Channel page** (binary) (b31, b30, b29, b28, b27)	**Channel number** (decimal)	**Channel number** description
0	00000	0	868 MHz, BPSK
		1-10	915 MHz, BPSK
		11-26	2.4 GHz, O-QPSK
1	00001	0	868 MHz, ASK
		1-10	915 MHz, ASK
		11-26	Reserved
2	00010	0	868 MHz, O-QPSK
		1-10	915 MHz, O-QPSK
		11-26	Reserved
3-31	00011-1111	reserved	Reserved

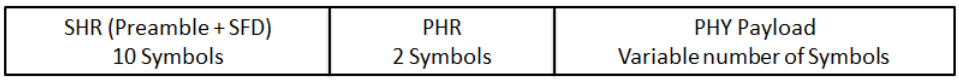

Figure 3.1 – ZigBee PPDU format.

bytes.

At the transmitter whose synoptic is given in Figure 3.2, the data frame is first spread into *chip* sequences and then modulated by an Offset-QPSK modulator. Spreading consists in multiplying each symbol which is composed of 4 bits by a Pseudo Noise (PN) sequence which includes 32 chips. The correspondence between the symbols and the chip sequence is given in Table 3.3. This spreading operation is a variant of the DSSS technique which was introduced in the first chapter. It results in a wider spectrum of 2 Mchip/s which is later split into two channels I and Q and shaped separately with a half sine FIR filter $p(t)$ such that

$$p(t) = \begin{cases} \sin(\pi \frac{t}{2T_c}) & 0 \leq t \leq 2T_c \\ \\ 0 & else \end{cases} \tag{3.4}$$

where T_c is the chip period. A delay of half a chip period is introduced in channel Q to enable continuous phase change which suits to the energy-efficient nonlinear amplifiers usually employed in IEEE 802.15.4 RF transceivers.

At the receiver, whose synoptic is given in Figure 3.3, the main objective is to recover the transmitted bits (data payload). The architecture of the receiver is not specified by the standard therefore the designers are given the freedom to implement a receiver which is tailored to their needs. An IEEE 802.15.4 PHY receiver would require a matched filter at the output of the *ADCs* to maximize the *SNR* in the presence of additive stochastic noise. The matched filter equation is $p^*(-t)$ and it is employed to reshape the incoming stream. After matched filtering, synchronization must be performed on the *Preamble* field. It mainly consists of the time synchronization to recover an optimum sampling period, the phase synchronization to compensate the phase shift due to either

Table 3.3 – Symbol to chip mapping

Data symbol (decimal)	**Data symbol** (binary) (b0 b1 b2 b3)	***chip* values** (c0 c1 ... c30 c31)
0	0000	11011001110000110101001000101110
1	1000	11101101100111000011010100100010
2	0100	00101110110110011100001101010010
3	1100	00100010111011011001110000110101
4	0010	01010010000101110110110011100011
5	1010	00110101001000101110110110011100
6	0110	11000011010100100010111011011001
7	1110	10011100001101010010001011101101
8	0001	10001100100101100000011101111011
9	1001	10111000110010010110000001110111
10	0101	01111011100011001001011000000111
11	1101	01110111101110001100100101100000
12	0011	00000111011110111000110010010110
13	1011	01100000011101111011100011001001
14	0111	10010110000001110111101110001100
15	1111	11001001011000000111011110111000

the transmission delay or the PLL phase at the receiver; and the Carrier Frequency Offset (CFO) which is a frequency offset introduced by the receiver PLL. Once these synchronization elements are determined, they are used to compensate the distortions on the rest of the incoming frame. The remaining of the receiver consists in decoding the rest of the frame through parallel correlation with the known *PN* sequences. Indeed, the received chip sequences must be distinguished within the 16 possible sequences. To this aim, a correlation bench which computes the correlation between the incoming sequence and the 16 known sequences is required. The correlation bench decides of the received symbol from a comparison between the output correlation values. The compared correlation values must be greater than a pre-established threshold.

3.2.3 State of the art of IEEE 802.15.4 SDR transceivers

The ZigBee technology has motivated a lot of research and development works both from a digital and an analog perspectives. Several ad-hoc and professional networks that have been deployed, rely essentially on this technology. In the context of SDR, IEEE 802.15.4 transceivers have been extensively used as prototypes for demonstration purpose as in [67][68][69][70]. Indeed, the standard compliant transceivers are relatively less complex to implement and therefore several SDR research have led to the proposal of an IEEE 802.15.4 baseband transceiver prototypes. In [67], a USRP-based SDR platform is used to implement the transceiver. The USRP is equipped with 14-bit ADC and connected to a host PC via a Gigabit Ethernet connection which means that all baseband signal processing takes place on the host PC by using Matlab or Labview. This

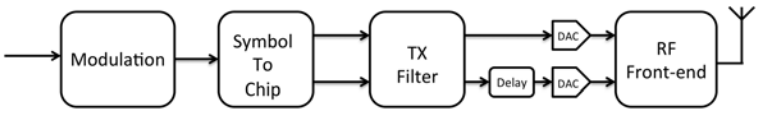

Figure 3.2 – PHY IEEE 802.15.4 transmitter.

transceiver was proposed to enable the computation of error probability parameters at different stages of the transceiver since such architecture makes it possible to access the different stages of the receiver and compute the associate error probability. Thus, performance can be analyzed at the chip-level, the symbol-level and the bit-level in a real-time implementation of the transceiver.

In [68], the authors have developed an IEEE 802.15.4 SDR transceiver whose setup is based on a URSP and the GNU Radio software framework. The software runs on a host PC which is connected to the USRP via USB2.0. The platform, shown in Figure 3.4, was first developed for education and monitoring purpose and also to enable a deeper understanding of the IEEE 802.15.4 PHY. The proposed receiver consists of synchronization, gain adjustment, a differential phase decoder, a symbol correlator and a data frame interface for upper software layers. The prototype was designed to work at the rate of 4 Msamples/s. The paper [69] introduces an all-digital transceiver based on the 868 MHz band of the IEEE 802.15.4. The prototype has first consisted in a behavioral model of the system fully developed in Simulink, which was then used to automatically generate some VHDL file by using Simulink HDL Coder. The generated VHDL files were simulated with ModelSim and synthesized with the help of the Xilinx ISE Suite. The design was implemented in a Xilinx Virtex-5 FPGA [71] and operates at 128 MHz.

In summary, the ZigBee technology has gathered some interest in the SDR community. It is a relatively light protocol which suits to rapid prototyping requirements as fostered by the SDR. However, in practice, ZigBee transceivers have often been implemented as ASICs since its stated goal is first to achieve very low-power performance. Considering different fabrics such as FPGA or CPUs while ensuring low-power requirements would certainly open this technology to more research in the SDR community.

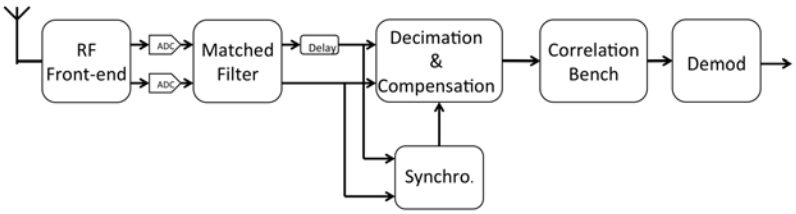

Figure 3.3 – PHY IEEE 802.15.4 receiver.

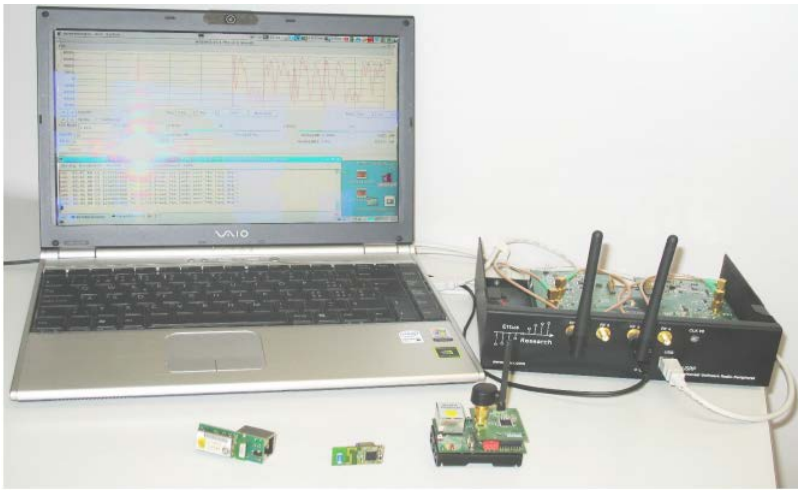

Figure 3.4 – IEEE 802.15.4 SDR Receiver.

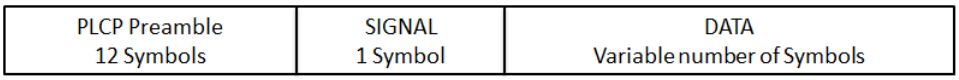

Figure 3.5 – IEEE 802.11a PPDU format.

3.3 The IEEE 802.11 Standards

3.3.1 Generalities

The IEEE 802.11 technology is an international standard which describes the main features of a Wireless Local Area Network (WLAN). It enables creating some high data rate wireless networks ranging from a few dozen meters up to a few hundred meters. Initially designed to support up to 2 Mb/s of data throughput, it later led to multiple revisions of the standard. Some of these revisions are characterized as follows:

— 802.11a: designed to ensure a theoretical rate of up to 54 Mbit/s for a range of a few dozen meters. It operates in the 5GHz frequency band.
— 802.11b: enables to obtain a theoretical rate of 11 Mbit/s for a range of a few hundred meters.
— 802.11g: enables to obtain a theoretical rate of 54 Mbit/s for a range of a few hundred of meters. It operates in the 2.4GHz frequency band.
— 802.11p: defines mechanisms that allow IEEE 802.11 technology to be used in high speed radio environments typical of cars and trucks.
— Some others such as the 802.11d which purpose is to enable an international wireless access, the 802.11e for better Quality of Service (QoS) or the 802.11f for better roaming services.

These technologies have been extensively deployed throughout the years and today they are part of our daily life in the sense that the *Wifi* network coverage tends to be accessible from any geographical points. In this work, we have shown a particular interest for the IEEE 802.11a whose PHY layer is discussed in the subsequent section.

3.3.2 The IEEE 802.11a Physical Layers (PHYs)

The IEEE 802.11a was released and approved in 1999 and it specifies the MAC and the PHY layers of a Wireless LAN. It was designed to support the transmission and reception at data rate of 6, 9, 12, 18, 24, 36, 48 and 54 Mbit/s by using an OFDM-based channel access technique that

Table 3.4 – PHY IEEE 802.11a modulation rate-dependent parameters.

Data rate) (Mbits/s)	Modulation	Coding rate (r)	Coded bits per subcarrier	Coded bits per OFDM symbol	Data bits per OFDM symbol
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24	16-QAM	1/2	4	192	96
36	16-QAM	3/4	4	192	144
48	64-QAM	2/3	6	288	192
54	64-QAM	3/4	6	288	216

we have already introduced in the first chapter. Similarly to the lately introduced ZigBee PHY, the IEEE 802.11a PHY organizes its data into frame prior to the transmission. The transmitted data frame is illustrated in Figure 3.5 and consists of three major fields. The *Preamble* field, which comprises twelve known symbols, is appended for synchronization purposes. It is composed of 10 repetitions of a "short training sequence" with a duration of $8\mu s$ (employed for AGC convergence, diversity selection, timing synchronization and coarse frequency synchronization in the receiver) and two repetitions of a "long training sequence" ($8\mu s$) that are used for channel estimation and fine frequency synchronization in the receiver. The *SIGNAL* field ($4\mu s$) includes frame-specific information like the data rate that can be any of the aforementioned data rates and the length of the data payload. This field is modulated with a BPSK modulator and coded at rate $r = 1/2$. Such modulation and coding scheme ensures the integrity of this field against the channel impairments. The *DATA* field carries the useful information and its rate and length information are embedded in the *SIGNAL* field. In Figure 3.5, one can see that each field is composed of a certain number of OFDM symbols as well. These symbols are computed with an IFFT as it was explained in the first chapter. The IEEE 802.11a defines a 52-subcarrier FFT/IFFT where 48 of them are used for data and the remaining 4 are used for pilot insertion. Table 3.4 summarizes the modulation parameters depending on the targeted data rate.

The IEEE 802.11a transmitter shown in Figure 3.6 consists first in gathering the encoded bits into groups of 1, 2, 4 or 6 bits as shown in Table 3.4. Each group is converted into a complex number, belonging to a given constellation, depending to the selected mapping scheme. These complex numbers (the outputs of the mapper) are then divided into groups of 48 complexes so as to compute a single OFDM symbol. To this end, each complex is mapped to a designated subcarrier and the 52 (48+4) subcarriers are converted into a time domain representation by using an IFFT. Each OFDM symbol is prefixed with a repetition of its end which forms the Cyclic Prefix (CP). Following that, a windowing operation takes place by using a time domain windowing function whose equation is given by

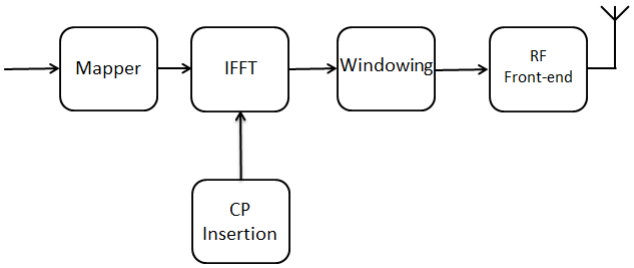

Figure 3.6 – PHY IEEE 802.11a transmitter.

$$W_T(t) = \begin{cases} \sin^2(\frac{\pi}{2}(0.5 + t/T_{TR})) & -T_{TR}/2 \leq t \leq T_{TR}/2 \\ 1 & T_{TR}/2 \leq t \leq T - T_{TR}/2 \\ \sin^2(\frac{\pi}{2}(0.5 - (t-T)/T_{TR})) & T - T_{TR}/2 \leq t \leq T + T_{TR}/2 \end{cases} \tag{3.5}$$

where T_{TR} represents the transition time between two windowing operations and T is the duration of the overall windowing operation. The windowing function can be optionally extended over more than one OFDM symbol period. The resulting windowed symbols or group of symbols are appended one after another to form the final frame.

The receiver, which is shown in Figure 3.7, starts with a windowing function which truncates the incoming frame into subframes of duration T. Afterwards, a synchronization block is appended to process the short training sequences of the *Preamble* field (coarse synchronization). It is followed by a CP removal block which removes the CP of the following OFDM symbols. Each OFDM symbol is then converted from time domain to frequency domain through an FFT operation. This operation produces some complex numbers which may present some distortions due to the channel. To tackle this issue, a fine synchronization block is added to compute the error (thanks to the long training sequences) when it is required and an equalizer block compensates the computed error on each complex number. Finally the complex numbers are de-mapped into group bits. As mentioned before, the *SIGNAL* field is used to recover the incoming symbols (*DATA* field) data rate and length.

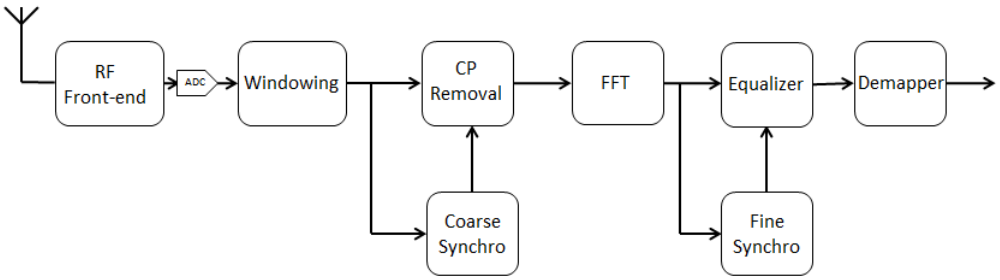

Figure 3.7 – PHY IEEE 802.11a receiver.

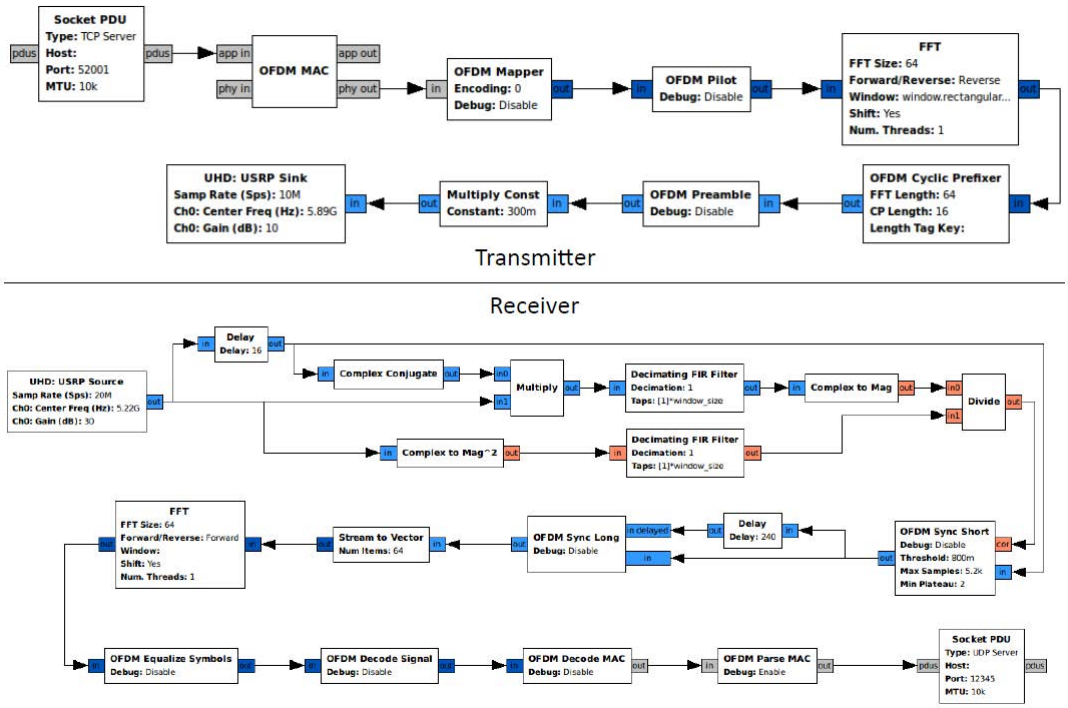

Figure 3.8 – GNURadio PHY IEEE 802.11p receiver.

3.3.3 State-of-the-art of IEEE 802.11a/p SDR transceivers

The IEEE 802.11a compliant transceivers have been extensively developed to provide wireless connectivity in the home, office and commercial establishments. Such transceivers are slightly complex to develop and they require further development efforts in comparison to IEEE 802.15.4 transceivers. This certainly explains why the standard is not often considered when it comes to implement a waveform prototype in a typical SDR validation process. Furthermore, the IEEE 802.11a standard was specified to operate in the 5 GHz whereas most of the SDR RF front-ends such as the USRP operate in the 2.4 GHz band. However, examples of IEEE 802.11a SDR transceivers [72][73][74][75] can be encountered in the literature. The authors of [72] address the implementation an IEEE 802.11a compliant transceiver on the OSSIE SDR platform. Their stated objective was to design and implement a software transceiver which uses the SCA [52] including the Common Object Request Broker Architecture (CORBA) [76] that allows flexibility, performance and maximum potential for software module reuse.

In [73], a full SDR-based IEEE 802.11p transceiver was developed in the GNU Radio framework. The IEEE 802.11p is a variant of the IEEE 802.11a that was developed for Wireless Access in Vehicular Environments. It was shown that the implementation of the overall transceiver including the MAC layer, as illustrated in Figure 3.8, can be run on a low-end desktop PC or a laptop. Moreover, the solution was made available as Open Source to enable further research within the SDR community.

Some other SDR-based implementations of the 802.11a transceiver have been proposed such as in [74] where a receiver was implemented on an array of programmable processors. The computational platform is precisely composed of an array of processors combined with configurable accelerators interconnected in a 2-D mesh network. The proposed receiver includes frame detection, timing synchronization, carrier frequency offset compensation and channel equalization. It is claimed that the implementation of the receiver has required 29 small processors together with some Viterbi and FFT accelerators that operate at the frequency of 590 MHz.

Finally, the authors in [75] have considered the FPGA as an enabling technology for the hardware platform of an SDR system as FPGAs offer the potential of hardware-like performance coupled

with software-like programmability. At the time the paper was written (2004), the authors have considered the relatively recent "partial reconfiguration" feature that was proposed on Xilinx FPGAs to implement an 802.11a baseband transceiver. They have partitioned the transceiver into static and reconfigurable units. Thus, coding and modulation blocks have been clearly identified as reconfigurable units while the FFT block was stamped as a static unit. Further to this, streaming data are handled with input buffers when reconfiguration takes place. The size of the buffer depends on the speed at which the FPGA can be reconfigured and the size of the FPGA being used. However, the authors have pointed out in conclusion that a proficient knowledge in FPGA design techniques is required to produce effective implementations and thus they suggested that automating different steps in the design process would be a relevant contribution. Automation usually implies to raise the level of abstraction of the design process.

Our work has considered both the IEEE 802.15.4 and the IEEE 802.11 standards as case studies within an SDR development flow. To our point of view, they enable to grasp the specificity of a dataflow application while allowing to explore some SDR features such as the flexibility or programmability.

3.4 Conclusion

Wireless technologies are widespread technologies which resulted from an increasing demand in connectivity. However, the requirements in terms of wireless performance appeared to be different depending on the application. Thus, telecommunication standards have been proposed by different institutions to address these communication needs. In this chapter, we have introduced two of these technologies, namely the ZigBee and the Wifi technologies. Their transceivers must be compliant to the IEEE 802.15.4 and the IEEE 802.11 standards respectively.

Actually, ZigBee is meant for low rate and power-efficient wireless applications while Wifi was designed to provide high data rate connectivity to the users. To this end, each of them employs a different modulation scheme namely the DSSS for the ZigBee and the OFDM for the Wifi. These modulations have been discussed in this chapter and some examples of their implementations in the context of SDR have also been detailed.

The research work presented in this document has partly consisted in the specification and implementation of these two waveforms through a proposed SDR design flow. The ensuing results will be discussed later in this document.

Chapter 4

High-Level Designing of Physical Layers (PHYs)

Contents

4.1 Introduction 52
4.2 Model-Driven Engineering 52
4.2.1 Generalities 52
4.2.2 Domain Specific Languages (DSLs) 52
4.2.3 Eclipse Modeling Framework (EMF) and Xtext/Xtend 53
4.2.4 Some relevant MDE-based technologies for embedded systems 55
4.3 High-Level Synthesis (HLS) 56
4.3.1 A bit of history 56
4.3.2 High-Level Synthesis Fundamentals 57
4.3.3 Advantages of HLS 58
4.3.4 Examples of mature HLS Tools 60
4.4 Bringing together HLS and MDE for FPGA-SDR 61
4.4.1 Dataflow Model of Computation (MoC) 61
4.4.2 SDR Control Requirements 62
4.5 In a Nutshell 62
4.6 Conclusion 63

4.1 Introduction

In the DSP domain, one can note that PHY designing has been performed with several methodologies throughout the decades. Each breakthrough in the proposed methodologies has mainly consisted in raising the level of abstraction. Indeed, raising the level of abstraction reduces the amount of detailed design work required. However, each of these evolutions came along with a lot of skepticism which was related to both the achievable performance and the associated learning curve. Each time the complexity of the desired architectures was increasing, because of the expected performance, it turned out to be tedious and time consuming to use the same design methodologies which required considerable efforts. The solution to this was to raise the level of abstraction while automating as much steps as possible in the final architecture synthesis process. Thus, regarding logic synthesis, the design methodologies have evolved as follows:

— 1960s: D-Algorithm [77] applied to Automatic Test Pattern Generation (ATPG) [78] is used for boolean reasoning.
— 1979: IBM uses logic synthesis for Gate Array-based mainframe design. LSS tool [79], followed by BooleDozer [80].
— 1986: Synopys founded and offers a logic remapper between Standard Cell Libraries which was later extended to RTL logic synthesis.
— 1990s to early 2000s: Major EDA companies offer commercial High-Level Synthesis (HLS) [81] [82] tools for system-level design.

Similarly, for DSP software platforms, the abstraction was raised from the assembly language or machine code to C or C++ language which is now mainstream. A generalization of this concept, which is mostly promoted in the software domain, is referred to as the Model-Driven Engineering (MDE) [83]. It fosters the use of models and the generation of code from the models. In this chapter, we discuss the modeling concept/tools that were proposed for software application in general and also PHY designing and then we will emphasize on the proposed environments in the context of FPGA.

4.2 Model-Driven Engineering

4.2.1 Generalities

The growth of the platform complexity exhibits the limitation of the current design methodologies for software-oriented platforms and also for embedded systems. As these platforms evolve, the application code is generally written and maintained manually. As a result, the developers spend considerable efforts to port the application codes to different platforms or newer versions of the same platform. A mainstream approach to address platform complexity is to develop the Model-Driven Engineering (MDE) [83] technology, whose main purpose is to improve the software development process. It shifts the development process from code-centric to model-centric by fostering the reuse of models, the transformation of models and the generation of code from models. The MDE technology combines:

— A Domain-Specific Language (DSL) [84] which purpose is to formalize the application structure, behavior and requirements in a declarative way. A DSL is described from metamodels which define the main concepts of the domain and allow capturing its expressiveness.
— A set of transformation engines and generators to parse the DSL script and generate diverse artifacts such as source code, simulation inputs or intermediate model of representations. Such approach is claimed to ensure a "correct-by-construction" development of the final application.

In the following sections, we will briefly review the existing software technologies that enable implementing an MDE environment and then we will discuss some of the existing MDE technologies that were proposed for embedded systems.

4.2.2 Domain Specific Languages (DSLs)

Domain-Specific Languages, as opposed to General Purpose Languages (GPLs) such as C/C++ or Java, are computer programming languages of limited expressiveness and tailored to a particular

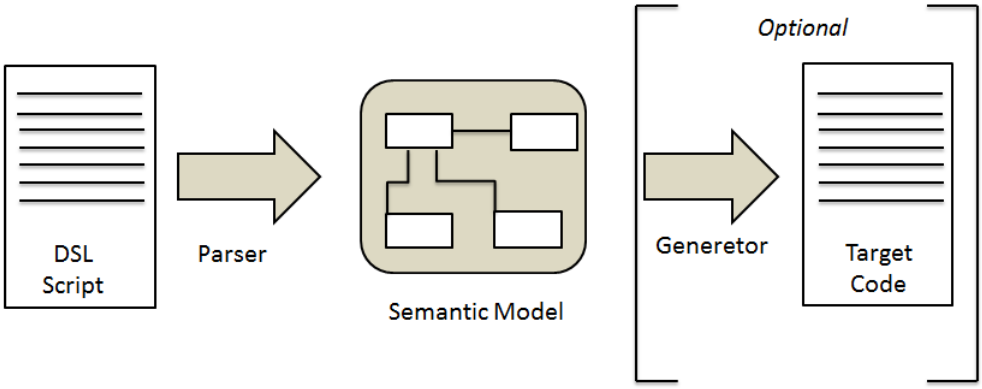

Figure 4.1 – Generic architecture of DSL processing.

domain. They can be viewed as very specific tools for very particular conditions. Their stated goal is among others to improve the development productivity, to ease the communication between domain experts and to provide an alternative computational model. Indeed, DSLs enable to raise the level of abstraction in a software or hardware development process. Doing so, they allow the automation of several steps throughout the process. Their expressiveness, through a tailored syntax, eases the communication between domain experts and facilitates it for arbitrary users to quickly grasp the main idea behind the proposed DSL. A generic DSL processing architecture is presented in Figure 4.1. It starts with a DSL-based description of the application which is then fed to a parser to generate an intermediate model referred to as a semantic model. This model is used to generate optional artifacts such as source code.

In [84], DSLs are declined into three types, namely the internal DSLs, the external DSLs and the language workbenches. Internal DSLs are usually hosted by a GPL and they are therefore constrained by their host language since any expression must be a legal expression in the host language. However, internal DSLs benefit from the host compiling framework which is generally well-developed and supported by an active developer community. Examples of internal DSLs are the Lisp language [85] or the Rails language [86] which was developed as a framework of the Ruby language [87]. External DSLs are usually developed from scratch thereby, they are not limited in terms of expressiveness compared to internal DSLs. They require defining a specific compiling framework that can leverage parsing programming languages techniques. Unlike internal DSLs, they might require more effort to be developed and may lead to a higher learning curve. Some of the external DSLs that the reader may have come across are for instance the Structured Query Language (SQL) [88] or XML [89]. Finally, the language workbenches are specialized Integrated Development Environments (IDEs) for defining and building DSLs. They proposed a custom environment for editing DSL scripts intended to model the target application. In the next section, we will first discuss the Eclipse Modeling Framework (EMF) [90] that is a framework intended for the development of language workbenches and then we will review the Xtext/Xtend framework that is an EMF-based language workbench that we have used in our research work.

4.2.3 Eclipse Modeling Framework (EMF) and Xtext/Xtend

The Eclipse Modeling Framework

EMF [90] was initiated by the Eclipse Modeling Project to enable developers to rapidly build robust applications based on simple models. EMF relates modeling concepts directly to their implementations in three important technologies that are Java, XML and UML as illustrated in Figure 4.2. It relies on the *Ecore* metamodel which defines the following components:

— *EClass*: enables representing a modeled class. It includes a name, some attributes and references.

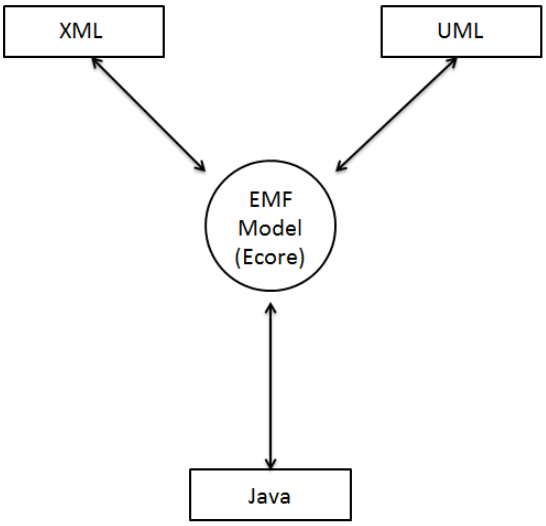

Figure 4.2 – EMF possible representations (Java, XML, UML).

— *EAttribute*: with a name and a type to represent a modeled attribute.
— *EReference*: used to represent some associations between classes.
— *EDataType*: employed to represent the type of an attribute.

An *Ecore* model is usually the entry point to an EMF project and EMF itself includes a simple tree-based sample editor for Ecore modeling. Ecore models have a canonical representation in form of XML Metadata Interchange (XMI) [91] serialization which does not add any extra information. Finally, the EMF project has inspired several other projects which leverage the ENF facilities to offer some modeling environments. Among them, the Xtext/Xtend framework enables defining the structure of DSL through a Backus-Naur Form (BNF) [92] syntax while offering an environment to edit the DSL scripts or models.

Xtext/Xtend Framework

Xtext [93] is an open EMF-based framework for implementing DSLs as well as their integration in the Eclipse IDE. It covers all the aspects of a language implementation starting from the parser, code generator or interpreter up to a full Eclipse IDE integration. It also enables building the entire DSL from its ANTLR (ANother Tool for Language Recognition) [94] like grammar specification and most of the intermediate steps like Abstract-Syntax Tree (AST) representation are handled automatically by Xtext itself. Actually, the entry point to the Xtext framework is a grammar specification which is composed out of rules. Xtext generates the lexer, the parser, the AST model, the construction of the AST to represent the parsed program and the Eclipse editor with all the IDE features. The AST is generated from an ANTLR specification which is derived from the aforementioned Xtext grammar. An AST is composed of nodes whose classes are generated using the EMF framework. Throughout an Xtext design process, the generators and model checkers, implemented by the designers, will have to traverse and analyze the AST in order to produce the required artifacts.

To this end, Xtext is enhanced with the Xtend programming language that is sugared version of the java programming language. Xtend enables customizing the Xtext framework so as to produce a desired output from a DSL specification. For instance, validation, tests, and code generation are entirely written in Xtend. Indeed, apart from the grammar definition, the main efforts while implementing a DSL will consist in visiting the AST and compiling it down to the desired artifacts.

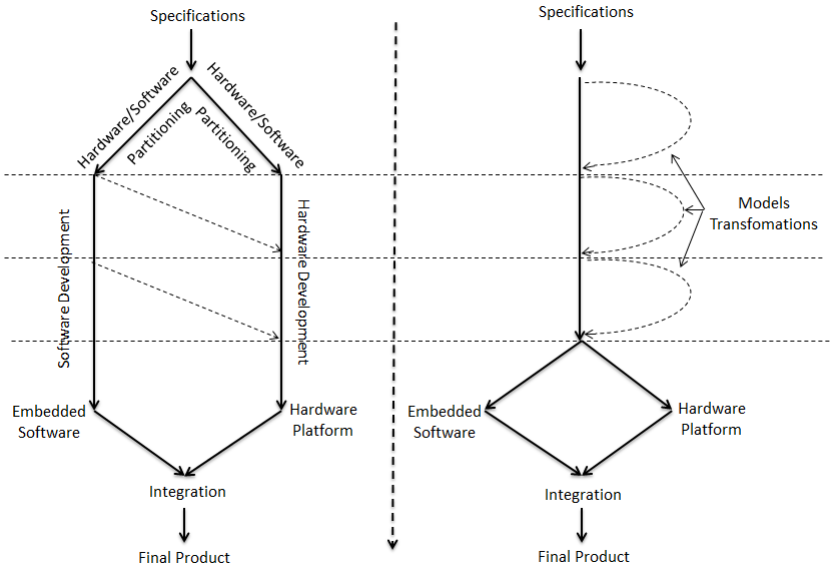

Figure 4.3 – Traditional co-design flows *vs.* MOPCOM co-design flow.

4.2.4 Some relevant MDE-based technologies for embedded systems

The MDE technology was essentially fostered by the software communities with a stated goal to raise the level of abstraction for implementing software applications. It winded up with several projects whose purpose was to standardize this approach into a common framework. The most active group toward such standardization is the Object Management Group (OMG) which conducted the projects that led to the definition of relevant standards such as the UML [95], the Meta-Object Facility (MOF) [96], the XML Metadata Interchange (XMI) [91] or the Model-Driven Architecture (MDA) [97], which is a refinement of the MDE for embedded systems.

These concepts are extensively employed by the software communities and they also gathered a lot of attention in the hardware design community. Indeed, the limitations of the methodologies for implementing hardware-oriented embedded systems have led the developers to consider raising the level of abstraction in their design process. Thus, some solutions were proposed to address this issue and most of these solutions are implemented as a subset or an extension of the UML standard.

SysML

The Systems Modeling Language (SysML) [98] is a general-purpose modeling language for system engineering. It reuses a subset of UML2 and provides additional extensions to address the requirements in UML for Systems Engineering. Moreover, SysML proposes both structural and dynamic diagram to model a system. It supports the specification, analysis, design, verification and validation of a broad range of embedded systems. SysML development environment usually consists of *plugins* and some commercial and open source modeling tools have extended their functionalities to support SysML development.

UML-MARTE Profile

Modeling Architecture Real-Time Embedded (MARTE) [99] is a UML profile intended for model-based Real-Time Embedded Systems (RTES) that was accepted by the OMG in June 2007. It mainly consists in a set of specializations of general UML so as to enable modeling real-time embedded applications. MARTE was proposed to replace the UML SPT (Schedulability, Performance and Time) profile [100] that was considered as too complex to implement by the community.

MARTE enables a high-level description of both the application and the platform. It stated goals are the followings [101]:

— Providing a common way of modeling both hardware and software aspects of an RTES in order to improve communication between developers.
— Enabling interoperability between development tools used for specification, design, verification, code generation, etc.
— Fostering the construction of models that may be used to make quantitative predictions regarding real-time and embedded features of systems taking into account both hardware and software characteristics.

MARTE is supported by stand-alone academic, commercial and open source development tools. Some *plugins* have also been proposed to support the MARTE project.

The MOPCOM Project

The MOPCOM (Modélisation et spécialisatiOn de Plates-formes et Composants MDA) [102] was defined to tackle the issue of co-designing complex systems composed of both software and hardware components. It stated goal is to offer an environment for the development of heterogeneous *Systems on Programmable Chip (SoPC)* by using a co-designing approach. Thus, it addresses the limits of traditional design flows where the partition between hardware and software components occurred early in the design process. Figure 4.3 illustrated the two approaches. Furthermore, MOPCOM relies on the MDA concept which implements the UML MARTE profile to cover the *Electronic System-Level (ESL)* domain, to automatically generate source code and provide some documentation. Indeed, the MOPCOM flow generates some source code for DSP microprocessors, microcontrollers, and FPGAs.

4.3 High-Level Synthesis (HLS)

4.3.1 A bit of history

As mentioned before, the methodologies that are involved in circuit architecture design have shifted considerably throughout the decades. These changes are related to the increasing complexity of the desired architectures. Thus, the methodologies that were usually employed turned out to be inefficient and time consuming. Each of these evolution consisted in raising the level of abstraction in order to enable rapid design time and better performance. This approach has the advantage of shortening the time-to-market while making it possible to explore more and more complex architectures.

Until the 60s, the circuits were manually designed at the transistor-level. Around the 70s, the first gate-level (AND, OR, NAND) design and simulation tools appeared. Subsequently, in the 80s the level of abstraction was raised to the schematic-level where coarser grained components such as adders and multipliers were employed to design circuit architectures. Hardware Description Languages (HDLs) such as Verilog and VHDL were proposed in the late 80s and popularized in the 90s. HDLs have enabled Register-Transfer Level (RTL) description of circuit architectures. An RTL description consists in describing the circuit's registers and the sequence of transfers between these registers but does not describe the hardware used to carry out these operations. Today HDLs are supported by mature synthesis tools which compile HDL-based descriptions down to binary executable (for FPGA designing) or netlist (for ASIC designing).

Once again, we are at the turning point in the history of circuit designing with more complex applications that require a shorter time-to-market. HDLs synthesis tools have addressed this issue with the concept of reusable IPs which fosters the reuse of pre-developed and well debugged functions [103] [104]. This approach is still on the mainstream but an alternative approach has consisted in raising the level of abstraction. Referred to as High-Level Synthesis (HLS) [81] [82], its main features and advantages are discussed in the following sections.

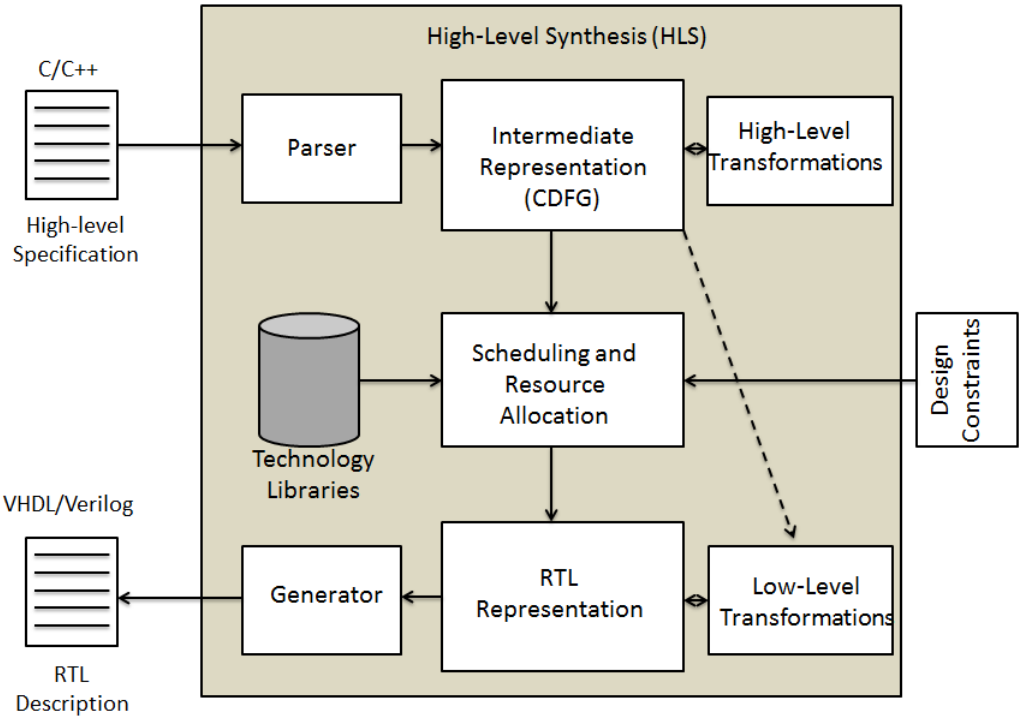

Figure 4.4 – Generic High-Level Synthesis Flow.

4.3.2 High-Level Synthesis Fundamentals

The HLS can be defined as the process of automatically generating quality RTL descriptions from high-level specifications. It has now come to maturity and gathers a lot of attention in the circuit designers community. HLS bridges the gap between algorithm designers and architecture designers however, it requires some knowledge in hardware design as well. It allows, all along the design process, to focus essentially on what the end-system does rather than how the system is implemented. Thus, most of the designing effort is put on specifying the application while the underlying architecture (RTL) is automatically generated. A high-level specification is usually performed with some High-Level Languages (HLL) [105] such as C/C++, SystemC or Matlab, which enable modeling an untimed representation of the application. A typical HLS tool first parses the provided high-level description of the application to extract an intermediate representation. Then, depending on the target technology, the design constraints and the structure of the intermediate representation, each operation is mapped on a dedicated resource and scheduling mechanism in form of state machine is decided for the computation. Finally, the RTL description of both the datapath and the state machine are generated. Figure 4.4 represents a synoptic of an HLS synthesis flow. The following lines discuss each of these steps.

High-Level Specifications for HLS

An HDL-based design flow often starts with specification of the application in a high-level language such as C/C++, SystemC or Matlab. These specifications serve as reference designs with which performance analysis, such as Bit Error Rate (BER) or Packet Error Rate (PER) in the telecommunication domain, can be performed under various constraints. Further refinements are then applied to these specifications so as to achieve more realistic ones. These refinements consist mostly in data quantization (data bit-width determination) which severely impacts the performance of the specification. Then, the associated RTL description is performed by a different team which has some deep knowledge in hardware designing. Finally, the design is validated and tested. This approach has been on the mainstream for many years (since the popularization of HDLs) and it

requires a good communication between all the teams involved in the designing process.

HLS on the other hand offers a direct path from the specifications down to the RTL description while fully automating the validation process. It relies on high-level descriptions and requires some knowledge in hardware designing. In fact, it is important to note that any software designer cannot be turned into a hardware designer since HLS designing requires some good quality specifications that include some hardware considerations. Moreover, the HLS specifications/implementations are written in an untimed language where no clock is specified and doing this can be troublesome for the hardware designers who generally work on a clock basis. Thus, HLS trades-off between the hardware and software engineers to enable rapid and efficient prototyping of the final solution.

An HLS specification is quite tool-dependent. Even though, most of the tools use C/C++ descriptions as entry point, some variations may appear when it comes to write a synthesizable specification. Data sizing, for instance, is supported by different libraries of bit accurate data types which model bit accuracy for *integer*, *fixed* or *complex* data. It contrasts with native C/C++ data types which come with predefined widths such as 1, 8, 16, 32 bits. Thus, the designer can model data with an arbitrary width. HLS tools enable specifying different kinds of annotation, also called *pragmas*, which can be used for optimization or design exploration purpose. In addition, care must be taken while manipulating memory components such as arrays which impact considerably the cost and performance of the resulting hardware.

A complete discussion over the specifications expected by HLS tools is out of the scope of this document. For deeper information, the reader can explore the user guide references provided by each tool.

From Specification to RTL Generation

The high-level specifications are fed to an HLS parser which converts the specification into an intermediate representation also called Control Data Flow Graph (CDFG) [106][107]. The CDFG models the control and the components that are involved in the datapath. It also models data dependencies which are identified within a single function. Figure 4.5 shows the CDFG corresponding to a for-loop. The control is represented at the upper part of the graph while the graph itself is composed of nodes which represent data and operators, and edges to represent the dependencies between data. From the CDFG representation, scheduling and resource allocation are performed with the goal to optimize both the performance and the occupied area. Scheduling consists in determining the order in which each operation is executed. It ensures in such a way consistency in the datapath. Several scheduling algorithms [108][109][110], whose goal is to minimize either latency or resources, are proposed in the literature. Resource allocation is the process which consists in binding the computation, communication or memory storage to some dedicated hardware resources with the objective to maximize the reuse of the hardware within a clock cycle so as to minimize the overall area.

Finally, RTL generation generation process takes place after scheduling and resource allocation processes are successfully performed. The final RTL (VHDL or Verilog) consists of four units namely, the control unit, the processing unit, the memory unit and the I/O unit, which are assembled into a top level design. Thus, the top level function includes a state machine, a datapath, some memory elements as well as some interconnections to the outside world. The RTL is claimed to faithfully implement the specification and most of the tools enable to automatically verify the generated RTL against the C/C++ testbench.

4.3.3 Advantages of HLS

HLS has opened a lot of research perspectives and the available tools can be considered mature. Its primary goal was to speed up the design time however it now comes with a lot of advantages which are specific to each tool. Thus, each tool offers different types of optimization which means that a final architecture's performance and cost may vary depending on the selected tool. In the following sections, we discuss some of the advantages of HLS.

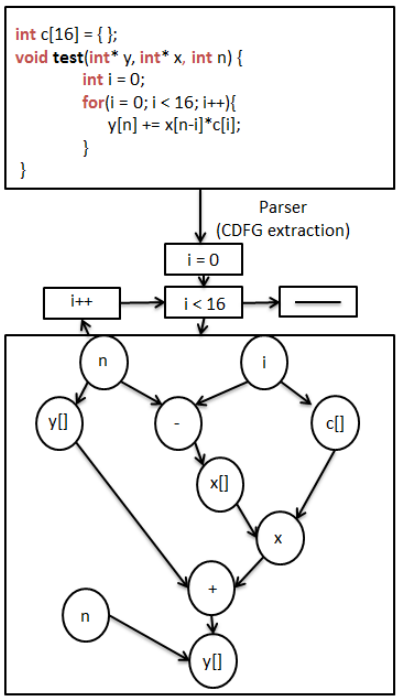

Figure 4.5 – A for-loop parsed into a CDFG.

Rapid Prototyping

One of the key elements of the HLS is undoubtedly its entry language which is an untimed HLL. This HLL, which raises the level of abstraction in comparison with traditional HDLs, is partly responsible of the success encountered by the most of the proposed software programming environment. By leveraging such well known languages, HLS enables somehow to speed up the design time while taking advantage of the existing compiling frameworks. In fact, it was shown in [111] that a 1M-gate design requires approximately 300K lines of RTL code where HLS would require around 40K lines of C/C++ code.

However, HLS compilers must extract a maximum parallelism from native sequential languages first designed for fixed architectures, whereas HDLs enable to explicitly express the parallelism within an application. Thus, HLS tools usually provide some tips whose purpose is to help expressing some parallelism into the specifications.

Optimization and Design Space Exploration (DSE)

Given that HLS employs high-level specifications, it naturally borrows several compile-time optimization techniques from the existing software compiling frameworks. These optimizations enable exploring the design space in order to generate an architecture that fits the best to the requirements. They can be led toward latency, throughput, power, memory or area optimization. HLS makes it possible to trigger such optimizations from high-level specifications which considerably accelerates the design process and enables shorter time-to-market to be achieved.

Latency and throughput optimizations can be addressed by using several techniques that remove performance bottlenecks. Pipelining techniques allow concurrent operations to occur and it has the effect of increasing the overall computation throughput. In the tools that we have experienced with, pipelining techniques can be applied at different levels. Dataflow, function or loop-level pipelining can be easily performed by inserting memory elements (registers) which break the critical path into several stages. Pipelining is generally characterized by an *Initiation Interval (II)*. It is a metric that was introduced to estimate how often, in number clock cycles, a pipelined function or loop

iteration starts. Thus, an II of 1 means that an iteration starts every clock cycle. Generally, the lower is the II, the faster is the function or loop.

Latency on the other hand can be addressed by optimizing memory accesses for instance. Indeed, instantiating an array results in a memory component in hardware that can be implemented as register files, Random-Access Memories (RAMs) or Read-Only Memories (ROMs). Register files enable a parallel access to all the cells of an array. It allows any function or operator, which fetches data from that array, to reduce its computation delays. But such array implementation requires considerable resources. RAMs and ROMs allow an access to a few cells at the time and they are sometimes considered as performance bottleneck. An alternative solution consists in partitioning the arrays into several memories that can be accessed in parallel. Some other techniques based on loop manipulation are also employed by HLS tools for optimizing latency. Those techniques are loop unrolling, loop merging or flattening nested loops. Loop unrolling consists in duplicating the loop body so as to minimize the initial number of loop iteration. This technique is characterized by a factor (the unrolling factor) U whose value tells how many times the loop body has been duplicated. Loop merging enables to improve the locality and reduces the loop overhead. However, care must be taken when using these loop optimization techniques as they might be very sensible to data dependencies.

To conclude, one of the big challenges in HLS resides in the possibility to extract parallelism from C/C++ descriptions that are sequential by nature. Indeed, one of the advantages of HDLs is their ability to express parallelism through concurrent expressions. HLS compilers analyze the provided specifications to extract both data and instruction parallelism. As aforementioned, different optimization techniques can be employed to explore the space of the solutions and different performance can be achieved when feeding a similar algorithm to different HLS tools.

4.3.4 Examples of mature HLS Tools

As it was pointed out in the previous section, parallelism extraction is quite a crucial aspect within each HLS design flow. GAUT [112] is an example of academic HLS tools that is dedicated to DSP applications. It starts from a pure C description and extracts the potential parallelism before selecting, allocating, assigning and scheduling hardware operations. It generates an IEEE P1076 VHDL file which is compatible with the commercial synthesis tools. However GAUT, as many of HLS tools, emphasizes on datapath designing and supports only loops with constant boundaries. Commercial tools are also available on the market. Among them, Catapult [113] from Calypto raises the level of abstraction by using the standard ANSI C++ and SystemC to describe the functions. Catapult speeds the time to RTL by automating the generation of bug free RTL and significantly reduces the time to verify RTL. It targets essentially ASICs and FPGAs and is featured with power optimization techniques.

Formerly known as AutoESL [82] and renamed Vivado HLS [114] after being purchased by

Table 4.1 – A comparative study between different HLS tools.

	Catapult	C-to-Silicon	Impulse-C	AutoESL	GAUT
Pointer Management	++	-	-	-	-
Memory Management	+++	+	-	+++	-
Interfaces Management	++	++	++	++	+
Loop Unrolling	+++	+	+	+	+
Pipelining	+++	+	+	++	-
Dependence Analysis	++	+	-	+	–
Resource Reuse	+++	++	-	++	++

Xilinx and made compatible with the Vivado Design Suite, Vivado HLS is an HLS tool that is tailored to the Xilinx families FPGAs and takes a mix of C/C++ function specifications as entry point. Some other examples of HLS tools are C-to-Silicon [115] from Cadence or Impulse-C [116]. Recently, OpenCL [117] [118] [119] has also been proposed as an abstraction to program FPGAs. It is argued that OpenCL has a native approach to express application parallelism, hence it is a good candidate for designing parallel signal processing applications for FPGA fabrics. Table 4.1 compares the features of some of these HLS tools.

4.4 Bringing together HLS and MDE for FPGA-SDR

In the previous sections, we have discussed the methodologies intended to raise the level of abstraction for implementing embedded systems in general and then, an emphasis was given to the FPGA technology through the HLS concept. Indeed, the HLS technology enables targeting FPGA or ASIC devices from HLLs, which raise considerably the level of abstraction when compared to traditional HDLs. As a result, HLS can be fully leveraged to improve the programmability of FPGAs. In the context of SDR, HLS can then be employed to virtualize the software intensive aspect when FPGA-centric platforms are targeted. However, couple of issues must be identified and discussed before defining such a framework that could enable some SDR PHY implementation while considering FPGA-centric platforms. Such issues would consist in defining an appropriate underlying Model of Computation (MoC) that would be suitable for the intended applications and also identifying the control requirements for such applications. In the next section, we introduce the Dataflow MoC which characterizes most of the current DSP applications.

4.4.1 Dataflow Model of Computation (MoC)

A dataflow program can be modeled like a directed graph which is composed of a set of computational units interconnected by communication channels through ports [120]. Each of these channels, usually implemented as First-In-First-Outs (FIFOs), corresponds to a stream of atomic data objects called tokens. The computational units, often called processes or actors, are expected to first read some tokens from their input channel then to perform a given computation on those tokens and then write the result (in form of tokens) to their output channels. Dataflow programming has been heavily used for the development of signal processing applications because it naturally suits the computation structure of such applications. A MoC is coarsely an abstraction of how the computation is done and it is also a useful representation when defining the semantics of a programming model. However, formalizing the dataflow MoC is quite an old paradigm which has motivated a lot of research. Thus, two main classes of dataflow MoCs arose from this research, namely the *static* dataflow MoCs whose behavior can be predicted (at compile-time) and the *dynamic* dataflow MoCs that exhibit a data-dependent behavior. The Khan Process Network (KPN) [121] is a remarkable proposal which represents a dataflow program as a graph $G = (V, E)$ such that V is a set of vertices modeling computational units or processes which operate concurrently, and E the set of unidirectional edges represented as unbounded FIFO channels. The inter-process synchronization is done by a blocking-read which ensures that every program following this model of concurrency is deterministic. As the control is completely distributed to the individual processes, a KPN implementation does not require a global scheduler. As a result, partitioning a KPN over a number of reconfigurable components or microprocessors is quite a simple task.

The Synchronous Data-Flow (SDF) [122] is a special case of static dataflow models in which the computational units (actors) consume and produce a predetermined number of token each time. In Figure 4.6 an IIR-Filter is modeled through an SDF graph and the number of consumed and produced tokens is explicitly annotated on each node. This prevents the computation from any side effect and most of the signal processing applications can be modeled in such a way. An implementation would require buffering the data tokens into FIFOs and control the nodes so that they are run when data is available. In a pure SDF approach, static *scheduling* (or control for uniprocessor implementation) is usually employed when it comes to implementation. However, a dynamic *scheduling* would be more suitable for flexible radios such as SDRs whose control may vary at run-time. Thus, different extensions of SDF were proposed to adapt the model to various

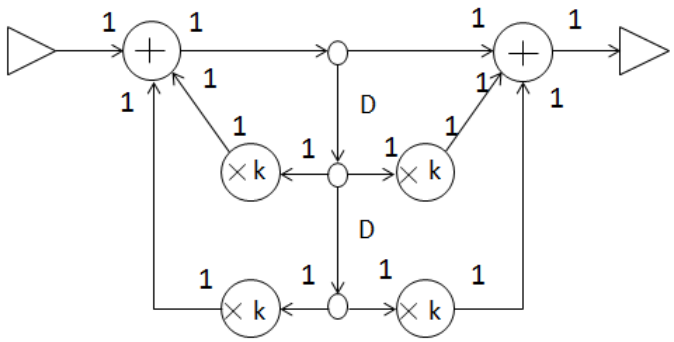

Figure 4.6 – A dataflow graph for a digital filter.

processing requirements. The *Parameterized Synchronous Dataflow (PSDF)* [123] is an example of such an extension which enables to bind the token production and consumption together with the expected delays. PSDF consists coarsely in parameterizing dataflow subsystems within a given dataflow graph so as to enable local configuration to be handled at run time. Finally, SDF and its extensions have been extensively used as the underlying MoC for several widespread DSP modeling tools. As an example, SDF is the spearhead of the Ptolemy project [124] which is dedicated to modeling, simulation, and design of concurrent, real-time embedded systems.

4.4.2 SDR Control Requirements

A dataflow implementation requires a control unit capable to support the data processing throughout the flow graph. In the context of SDF, such control functionality must be determined at compile time so as to set the memory and computation resources. Figure 4.7 shows an SDF signal flow graph and its possible implementation which is composed of memory resources (FIFOs) for interconnection purpose, functional blocks (FBs) to process the streaming data and a control unit to sketch the process. The control appears as a central element which interacts with the functional blocks to produce the desired behavior. Furthermore, statistical analysis must be performed to determine the appropriate depth for each FIFO within the graph since multirate systems can also be modeled with SDF. This task is achieved by analyzing a *topology matrix* which is extracted from the SDF graph and whose properties are used to determine whether the graph can have a valid schedule or not.

An SDR PHY implements signal processing algorithms and the SDF flow graph is a suitable candidate for such implementation. However, SDR PHYs foster reconfigurable computational units or functional blocks to ensure the flexibility of the PHY. To this end, an adapted MoC must be established for SDR-PHYs and the provided control unit must able to support switching between different configurations. At a coarser grain, reconfiguration may be required in SDR at the PHY-level. Typically in a multi-standard SDR, such a scenario must be envisioned and once more the control unit should be capable to handle the handover between two configurations.

4.5 In a Nutshell

Abstraction seems to be the mainstream approach to enhance the productivity and performance in engineering in general. In the software domain, such an abstraction has been implemented and rapidly adopted through technologies like the MDE. Indeed, these technologies are now widely used to developed daily-life software applications. Conversely, raising the level of abstraction in the hardware domain usually comes with a lot of skepticism owing to initial achievable performance. However, some mature tools have been recently proposed to tackle the issue of programming hardware applications from high-level specifications. But, these proposals still require deep knowledge

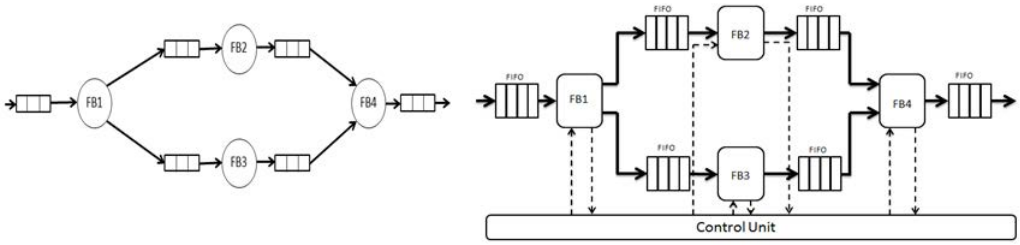

Figure 4.7 – An SDF signal flow graph (left) and its possible implementation (right).

of the underlying hardware. Such tools can be leveraged to rapidly prototype circuit architectures intended for FPGAs or ASICs.

In the SDR domain, an open research work is the FPGA-centric SDR platform development. Indeed, there is an obvious interest in using FPGA fabrics for implementing SDRs however their design methodologies contrast with the paradigm of SDR. Is merging the MDE technology and the tools that enable high-level specifications for FPGA would automatically result in a design flow for FPGA-SDR?

4.6 Conclusion

In this chapter, we have discussed the technologies intended to raise the level of abstraction both in software and hardware domains. These technologies aim at reducing the complexity of developing an application by automating as much steps as possible in the designing flow. It is clearly shown that this approach improves the overall productivity and enables to perform several verification early in the design process. Regarding software development, this approach was rapidly formalized into the MDE technology and widely adopted by the community. On the contrary, hardware communities mostly employed low-level technologies in their design process. However, HLS enabled to bridge the gap between the specifications and the implementation in a hardware design process. It is a promising alternative that mirrors software programmability for FPGAs and ASICs. The chapter ends with a discussion on how MDE and HLS could be unified into a design flow for FPGA-SDR and this represents the heart of our contributions detailed in the second part of this thesis.

Part II

CONTRIBUTIONS

Chapter 5

A Domain-Specific Language (DSL) for FPGA-Based SDRs

Contents

5.1 Introduction 68
5.2 The Proposed Design Flow 68
5.2.1 Waveform Modeling 68
5.2.2 Waveform Compiling 69
5.2.3 Verification and Validation (V&V) 69
5.2.4 Platform Integration 70
5.3 Conceptual aspects of the proposed DSL 70
5.3.1 Platform Modeling 70
5.3.2 DSL-Based Data-Frame Modeling 71
5.3.3 DSL-Based Dataflow Modeling 73
5.4 Frame-Based Control Unit 76
5.4.1 A Hierarchical FSM (HFSM) for FPGA-based Dataflow Control 76
5.4.2 Frame-based Control Algorithm 77
5.4.3 Simulation of the proposed HFSM on the StateFlow Environment . . . 80
5.5 Library of HLS/RTL-based Functional Blocks 81
5.6 Conclusion 82

5.1 Introduction

An ideal SDR development tool should allow capturing any DSP application structures regardless of the underlying hardware. It is also important to allow in such tools, the possibility to express the flexibility of the desired waveform through constraints or annotations. However, flexibility issues must be considered both at compile-time and run-time. Compile-time flexibility should enable exploring the design solution space while run-time flexibility is much more about how to change the functionality of the selected design at run-time. A well-planned SDR development tool should provide a minimum set of waveforms in form of library at launch and then evolve to support multiple waveform constructions. Regarding embedded systems in general, some abstracted modeling environments [99][98], which mostly rely on the Unified Modeling Language (UML) [95] have been proposed. Similarly in the context of SDR, other abstractions have also been proposed and we have already discussed their potential earlier in this document. In this chapter we introduce our proposal, which consists in an FPGA-SDR design flow in form of a DSL [84]. The goal is to provide an environment to model and implement SDR PHY intended to be run entirely on an FPGA fabric. The flow is featured with the HLS technology to enable rapid prototyping of the desired waveforms. An associated compiling framework has been developed to automate different stages of the flow by automatically generating the required artifacts. Throughout this chapter, we will illustrate each step of the design process with some generic examples. Some case studies will be provided later in the document. Section 5.2 introduces the proposal and Section 5.3 provides more details on the conceptual aspects of the DSL. In Section 5.4 we will discuss the features of the automatically generated control unit. Section 5.5 deals with the features of functional blocks composing the HLS library and conclusions are drawn in Section 5.6.

5.2 The Proposed Design Flow

The research work that is presented in this document has mainly consisted in defining and implementing a software-based framework for designing FPGA-SDRs. A synoptic of the proposal is illustrated in Figure 5.1. It comprises four major stages, namely the *Waveform Modeling* stage, the *Waveform Compiling* stage (referred to as DSL-compiler), the *Verification and Validation* stage and finally the *Waveform Programming* stage (Platform integration). Each of these stages covers a specific aspect in an SDR waveform development process and we strongly believe that bringing them together ensures the completeness of the proposal. As the reader can notice in Figure 5.1, the flow is also featured with a library of functions whose main goal is to enable rapid prototyping. The main idea behind this approach is to provide an evolving library of signal processing components that can be used to program any SDR. Such a block-based approach was suggested by Joseph Mitola [10][11] who clearly identified the need for a set of DSP primitives/functions in a software radio development framework. Thus, most of the SDR frameworks encountered in the literature are featured with some libraries of signal processing primitives. Further details on the library which is featured with our proposal will be given later in this chapter. The following sections introduce each of the previously mentioned stages.

5.2.1 Waveform Modeling

The *Waveform Modeling* stage is the entry point of our SDR development flow. It consists of an external DSL, developed with the Xtext/Xtend framework [93], which essentially allows capturing the data types and the dataflow organization of an SDR waveform. The DSL was developed from scratch by defining a meta-model/grammar for the language through a BNF-like (Backus-Naur Form) syntax [92]. Indeed, the Backus-Naur Form is a formal and mathematical way to describe the grammar of a language. It coarsely consists in defining a set of rules that are used to evaluate the correctness of an instance of the language source code.

The Xtext/Xtend framework takes a BNF syntax as entry point. Our DSL definition has then consisted in defining some grammar rules in Xtext so as to model different aspects of an SDR waveform. Thus, these rules enable modeling a typical FPGA-based platform by providing the FPGA device information as well as the ADC and DAC features such as their precision or their

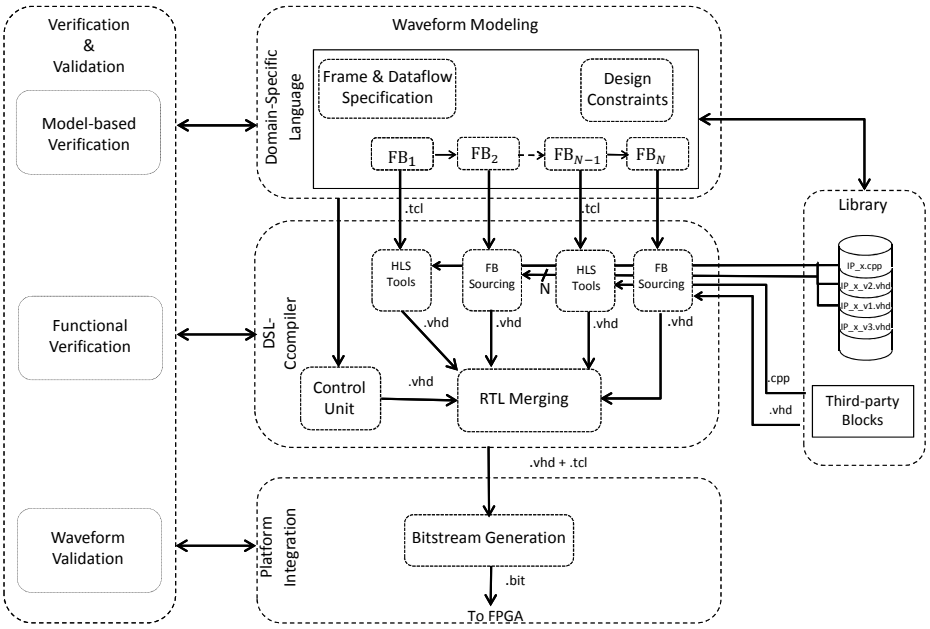

Figure 5.1 – Proposed Design Flow.

operating frequency. The other aspects that are also modeled with the DSL is the data frame structure together with the dataflow organization. These aspects will be further discussed in the following sections. The DSL finally allows specifying a set of design constraints whose goal is to enable both compile-time and run-time waveform flexibility.

5.2.2 Waveform Compiling

A DSL-based description of an SDR waveform is automatically converted into an intermediate representation consisting of an Abstract Syntax Tree (AST). This AST is actually an *Ecore* model which represents the converted code in form of a directed graph whose nodes are coarsely the attributes of rules specified within the source code. The generated AST is made compatible with the ANTLR [94] parser generator which is quite a famous tool in the computer scientist community for generating language parser or compiler. Xtext has been used to model its own parser/compiler generator which is the Xtend framework. Thus, an AST that is issued from a DSL specification can be parsed/compiled into a desired artifact, such as source code, by a set of functions written in Xtend. Xtend itself is a kind of sugared version of the Java language however the parser could have also been written in Java, which is compatible with Xtext.

In this work, we have proposed a DSL compiler which is entirely written in the Xtend language. Its main goal is to automate as much steps as possible in the SDR design process. Thus, a DSL-based SDR specification is first converted into an AST. Following this step, some synthesis scripts are generated for each HLS-based FB instantiated in the DSL specification. The synthesis scripts are used in combination with the HLS description of the block, to generate the RTL-VDHL description of the block through the HLS tools. A control logic tailored to the application is inferred from the AST, as well. Finally, both FBs and the control are assembled into a waveform.

5.2.3 Verification and Validation (V&V)

Verification and validation processes are very important in a design flow since they ensure that the final design will meet with the requirements. They can be time consuming to perform, however, they enable proving the correctness and reliability in the various steps of the design and

implementation processes. In the proposed design flow, as illustrated in Figure 5.1, the V&V is declined into three steps namely, the *Model-Based Verification*, the *Functional Verification* and the *Waveform Validation*. Each of these steps gradually verifies the correctness of the waveform at each stage of its development. Thus, the *Model-Based Verification* relies on the MDE concept to ensure the validity of the specified model. The *Functional Verification* is intended to verify the generated implementation of the waveform and consists of different steps. Finally, the *Waveform Validation* consists in testing the waveform which is programmed on the platform.

5.2.4 Platform Integration

Once the desired SDR waveform implementation has been automatically generated and verified, a platform integration of the solution comes next. This stage relies on the software tools which are provided to synthesize the bitstream for the target FPGA. Indeed, the integration is quite tool-dependent since the target platform(s) is (are) supported by some specific software toolsets. One of our goals was to automate some parts of this stage by leveraging some automatically generated scripts from the high-level descriptions.

We believe that these four stages compose an SDR design flow. In addition, a library of functions is associated to this flow so as to enable rapid implementation in the long term thanks to a block-based designing approach. In the forthcoming sections, each of these stages will be further discussed and illustrated through generic examples.

5.3 Conceptual aspects of the proposed DSL

An SDR development tool/flow must support different requirements regarding the implementation while enabling some system-level simulations. The simulations allow the designer to analyze the big-picture behavior of an architecture while creating its underlying solution. In the literature, system simulation tools fall into two categories [38] namely, the code-based tools and the block-based tools. The first category fosters a description of the system by using specific commands in a specific language while the second category requires the developer to draw block diagrams to describe the algorithm. The major difference between these two approaches resides in the way the dataflow is handled. Indeed, code-based tools would generally require the designer to manage the data transfer between functions with loop constructs for example, whereas block-based tools implicitly provide buffers and flow control between blocks. As a result, block-based tools take more time to simulate compared to code-based tools. However, block-based tools are becoming more and more popular and it is a common sense to recognize that block diagram are more intuitive when compared to equivalent function calls.

We have defined and implemented a code-based tool which consists of a DSL, to model and implement an SDR waveform to be run entirely on an FPGA-based platform. A typical DSL-based description is composed of a platform model description followed by a data frame and a dataflow description. This innovative approach employs the description of the dataflow structure together with the data frame model for an automatic inference of the control unit. Each of these steps is depicted and illustrated with generic examples in the following paragraphs.

5.3.1 Platform Modeling

The proposed DSL enables to provide a description of the intended FPGA platform. To this aim, a description of the FPGA device must be provided together with the features of the available signal converters. Such features comprise the converters bandwidths (in MHz), data precision (in bit), sampling frequency (in MHz) as well as their serial/parallel nature. Recall that, a serial converter requires the data from different channels (I and Q for instance) to be multiplexed before being converted while parallel converters connect directly to the channels. The information enables to tailor the final waveform description to the platform requirements.

Figure 5.2 shows the description of an FPGA-based platform with the proposed DSL. Both ADC and DAC are described by explicitly giving some of their relevant features. For instance, the

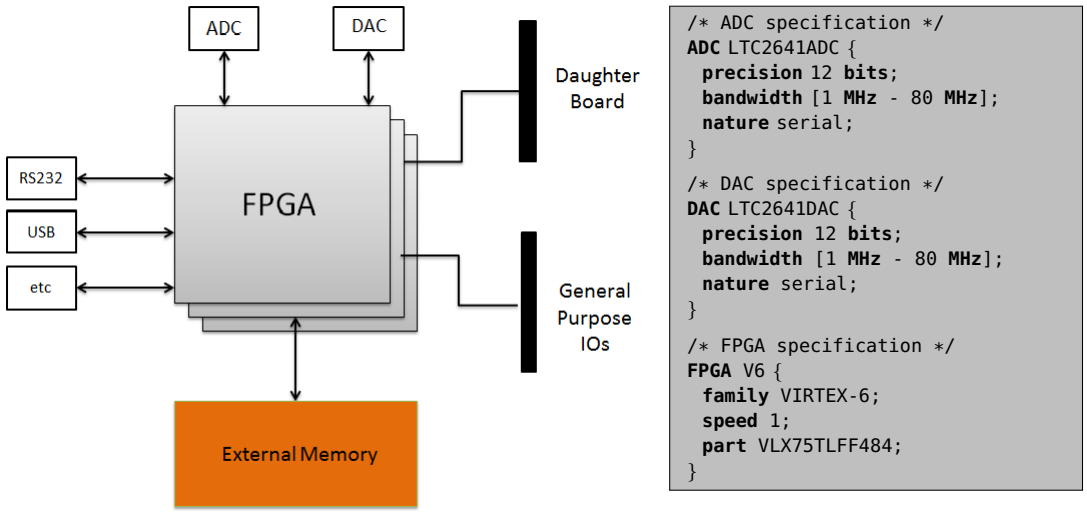

Figure 5.2 – DSL-based platform description.

employed ADC has a data precision of 12 bits with a bandwidth between 1 MHz to 80 MHz. It implies that the information signal bandwidth which can be properly sampled by such an ADC should not exceed 40 MHz. FPGA is also depicted in this description by giving the device family, its speed grade and further information. All these information are later used to guide the compiler so as to produce appropriate RTL from the HLS tools. However, not all types of FPGAs are supported at the time but further extensions to more FPGA families are under consideration.

5.3.2 DSL-Based Data-Frame Modeling

Most of the radio communication standards [20][21][22] organize the transmitted set of data into data frames or data packets. As mentioned previously, this frame structure ensures among others the interoperability of the standard compliant transceivers that are released by different vendors. A data frame at the PHY-level is composed of a set of fields or subframes which carry either synchronization information or upper layers, such as Medium Access Control (MAC), data payload or frame-specific information.

Further to this, the information that is nested in a given field may vary or remain unchanged in all the transmitted frames. For instance, synchronization fields should always respect a regular pattern whereas data payload can vary in terms of content or size. The proposed DSL provides the keywords to describe such organization of a data frame. Thus, each field can be characterized depending on its constant or variable nature. As a result, two types of fields were identified, namely the constant fields and the variables fields. Figure 5.3 shows a generic data frame structure, which is composed of N fields numbered from 1 to N. From this example, both data field and data frame DSL-based descriptions are discussed in the following lines.

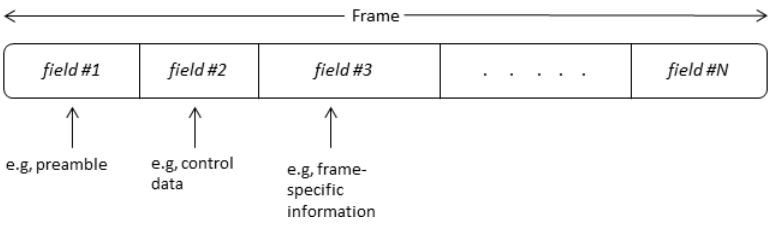

Figure 5.3 – Generic data frame.

```
/* Specification of field#1 */
#fieldC F1 {
    constant dataf1;              /* Constant symbol */
    redundancy 8;                /* Repetition over 8 symbols */
    duration 128 us;              /* Overall duration of the field */
}
                .
                .
                .
/* Specification of field#3 */
#fieldV F3 {
    data dataf3;                  /* Variable data being carried by field#3 */
    duration 32 us;               /* Fixed duration of field#3 */
}
                .
                .
                .
/* Specification of field#N */
#fieldV FN {
    data datafN;                   /* Data payload conveyed by field#N */
    maxsize 128 bytes;              /* Maximum payload size */
    minsize 16 bytes;              /* Minimum payload size */
}
/* Data Frame Specification */
complex frame F {
    F1 F2 F3 ... FN
}    sof after F1               /* F1 is designated as the start of frame */
```

Figure 5.4 – DSL-based description of a generic data frame.

Field specification

In a DSL description, the declaration of constant and variable fields is done with the keywords *#fieldC* and *#fieldV* respectively. They are followed by an arbitrary identifier that will be used to reference the field in the rest of the DSL source code. Following this step, field-specific information such as data redundancy, size or duration are defined in a structure-like specification. The *redundancy* within a given field enables to highlight a repetition of a data structure. The size can be specified as a constant (*size*) or through an interval in *byte* within which lays the size of the field. Specifying the size as an interval is mainly employed for data fields with variable size. The *duration* provides the exact duration (in μs) of a given field. The duration is usually specified by the standards when the field size does not vary. Actually, as it will be discussed later, these definitions aim at adapting the control when each field is transmitted or received through the transmitter or receiver dataflow graph.

Figure 5.4 shows a DSL-based description of the generic data frame illustrated in Figure 5.3. In this description, we assume that the *field#1* denoted F_1 is a synchronization field. Moreover, F_3 and F_N carry frame-specific information and data payload respectively. As a preamble field, F_1 has a constant structure hence being specified with the keyword *#fieldC*. It is composed of a repetition of a regular pattern which is shown by the keyword *redundancy*. Such repetition enables at the transmitter to compute once the data and then store it into a given memory unit. Thus, the constant field can be multiplexed from that memory storage with the rest of the frame at run-time. In the other cases where the constant field does not exhibit any redundancies, the constant data which composes the field can also be entirely stored into memory and multiplexed to the rest of the frame at run-time. F_3 which carries frame-specific information is specified with the keyword *#fieldV*. It has a fixed duration and its content is provided byte-wise through the variable *dataf3*. The data frame payload is conveyed by the field F_N. It is a variable field which carries the useful bits. Its content size generally belongs to an interval which is given by the standard. In the proposed DSL, such an interval can be specified in byte within the definition of the field. Further to this, the useful data that are provided by the upper layers and identified by the variable *datafN*. This variable has a byte size by default.

We believe that such a specification enables to capture the features of each field so as to adapt the control path afterwards. Further discussions on how the description of each field is employed to infer an efficient control path, are given later in this document.

Data frame specification

After each field composing the frame has been specified, a resulting data frame is specified with the keyword *frame*, which is followed by an arbitrary frame identifier. Subsequently, through a structure-like specification, the set of fields composing the frame are listed according to the order with which they are transmitted. Moreover, a frame can be specified either as *complex* or *real*. Remember that complex frames imply both an inphase and a quadrature phase projection of the signal. It results in a baseband signal composed of two channels usually called I and Q. Once a frame has been specified, a *Start-Of-Frame* (*sof*) delimiter is designated after a given field. This *sof* information denotes essentially a set of synchronization elements that must be computed at the receiver. Thus, the detection of the *sof* enables to intuitively sketch the control path of the waveform at the receiver. The description of the data frame which is made up of the N fields is given at the bottom of Figure 5.4. An *sof* is designated after F_1, which implies that synchronization will mainly consist in recovering some information from the field F_1.

The purpose of a frame declaration is twofold. On the one hand, computation resources are optimized with regards to the nature of the field. As mentioned previously, constant fields are one time computed, mapped to memory and inserted in the rest of the frame at run-time. The actual frame is built by consistently multiplexing those fields to the rest of the frame during transmission. On the other hand, the attributes of each field, especially the duration information, enable to build the appropriate control unit tailored to the intended waveform.

5.3.3 DSL-Based Dataflow Modeling

As aforementioned, the proposed DSL provides the primitives to capture the organization of a dataflow structure. Such a dataflow structure is mainly composed of Functional Blocks (FBs) as well as a communication infrastructure. In the context of an FPGA-based SDR, the dataflow structure is intended to be entirely programmed in the FPGA. To this end, the DSL leverages some libraries of HLS-based or RTL-VHDL based FBs that can be instantiated on-the-fly so as to build the desired datapath. In the following sections, we first discuss how to model a dataflow structure with the proposed DSL and then we will discuss the constraints which can be injected within the model so as to enable a flexible implementation of the waveform.

DSL-based dataflow description

The dataflow structure is common to most of the DSP applications. Moreover, its implementation has been extensively discussed in the literature. The Synchronous Data Flow MoC provides an intuitive way for describing a dataflow graph. It fosters an implementation within which FBs are interconnected via FIFOs of pre-determined depth. The SDF MoC has been discussed earlier in this document.

To capture such a structure, the proposed DSL first allows the designer to specify the operating clock as well as some arbitrary data rates which denote the frequencies at which the internal FBs operate. Then, interconnections are specified with the keywords *framedata* or *event*. The *framedata* keyword denotes a dataflow connection (FIFO) and the *event* connection denotes an event-like connection such as a control data. Each of these connections is identified with an arbitrary name, which is provided by the designer. The specification of an FB starts with the block name followed by the function of which it is an instance. It also requires to mention the set of fields that will be processed by the instantiated block after the keyword *processing*. Indeed, FBs do not often process the entire data frame but only a set of fields which composes the frame. Thus, the proposed DSL enables to mention explicitly which fields must be processed by a given block. In addition, it is required to mention the sources of those fields, which can be a preceding FB or an ADC. This step is done with the keyword *from* which is followed by the name of the source. The idea to specify each block on a field-basis is to later infer a control path which can sketch accordingly the computation for each FB. It is important to recall that each FB has been well-debugged prior to its integration into the HLS library. Thus, its instantiation within the DSL implies that it is functionally correct.

Figure 5.5 highlights the specification of the clock and data rates, the interconnections as well as an FB. The clock is specified by explicitly mentioning its value whereas the internal rates can be optionally derived. Thus, the sampling rate f_e is made equal to 10 MHz while the symbol rate f_s

```
/*Clock and rate specification*/
clock   f_clk = 20 MHz;
rate    f_e = 10 MHz;
rate    f_s = f_e/2;
/*Interconnection declaration*/
connect1   framedata on 6 bits;
connect2   framedata on 6 bits;
connect3   framedata on 10 bits;
connect4   event on 2 bits;
/*Specification of a functional block*/
FB1_i:  ip FB1 processing F_1 from FBj_x {
    read   connect1   on port   in1   at   f_e;
    read   connect2   on port   in2   at   f_e;
    write   connect3   on port   out1   at   f_s;
    write   connect4   on port   out2;
    synthesis    catapult;
    constraint    throughput 2;
}
```

Figure 5.5 – DSL-based description of an FB.

is derived from the sample rate. The specification also consists of four interconnections comprising *framedata* connections as well as an *event* connection. Furthermore, their data width is specified in number of bits. The specification shows the instantiation of a FB ($FB1\_i$) that processes a single field (F_1) source from another FB (FBj_x). Following this step, the interfaces of the FB are consistently connected to the interconnections thus building the dataflow graph. The input connections are specified with keyword *read* while the output connections are specified with the keyword *write*. In addition to this, *framedata* connections require specifying the expected data rate as it is shown in Figure 5.5. The resulting FB can be illustrated as in Figure 5.6. Its input data are read from FIFOs interconnections (connect1 and connect2) and output data are written into a FIFO and a wired interconnection. Further connections, such as clock, reset and an enable signal, are added at synthesis time as it can be seen on Figure 5.6 as well.

A key advantage in using HLS-based FBs resides in the fact that they can be further optimized by using some design constraints. We have leveraged this feature in the definition of the DSL by enabling the specification of design constraints while instantiating a given block.

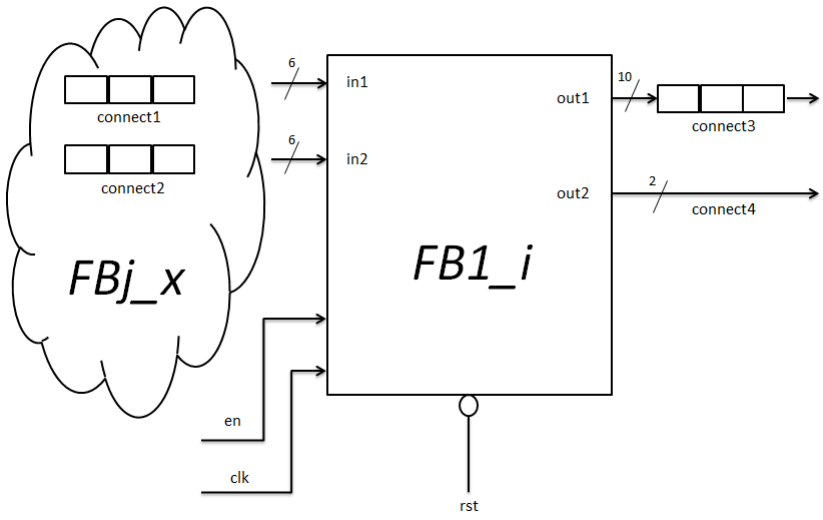

Figure 5.6 – FB equivalent architecture.

Design Constraints

As aforementioned, the HLS tools allow the designer to specify a set of design constraints in a given FB specification. Such constraints are intended to guide the synthesis toward a specific objective. Thus, several optimization can be performed with respect to throughput, area or latency performance. From then on, a FB which was previously seen as a black box in the proposed DSL, can now be considered as a black box whose implementation can be made flexible. HLS tools usually employ some annotations or *pragmas* to guide the optimization process. In the DSL, we have defined some similar annotations that can be used in the specification of a block. Specified with the keyword *constraint* in the DSL, it is an optional feature which can help exploring and optimizing the FB implementation. At the time of writing this report, we have only addressed design throughput and latency constraints in the current version of the DSL. They consist in explicitly mentioning the target throughput or latency in a number of clock cycles. This approach enables to explore the space of a design implementation until the requirements can be met. It is made possible by the use of synthesis scripts which are automatically generated for each FB. In Figure 5.5 such a constraint is specified on the expected throughput. Indeed, the interpretation of this constraint specification is that the underlying FB architecture should have a throughput of two clock cycles.

We have defined a basic allocation strategy for selecting (after synthesis by the HLS tools) the appropriate FB or IP. Basically, the DSL-Compiler chooses the FB with the lowest area, which respects the target throughput. However, latency should be taken care of so as to ensure the correctness of the final design. In Figure 5.7, a block IP_x is connected to two other blocks IP_{x-1} and IP_{x+1} with an input rate of f_{in_x} and an output rate of f_{out_x} must satisfy the following latency constraint in number of clock cycles:

$$Latency_x < \frac{f_{clk}}{max(f_{in_x}, f_{out_x})} \tag{5.1}$$

where f_{clk} is the design clock. By trading-off in such a way, the compiler can ensure some consistency inside the dataflow graph while exploring different solutions.

Moreover, the DSL provides a mean to specify the target HLS tool through the keyword *synthesis*, which is followed by the name of the tool. As different HLS tools are emerging on the market, our proposal aspires to take advantage of each of their offerings into a co-design approach. Thus, synthesis scripts are proposed as an interface to the HLS tool. So far, we have experienced with

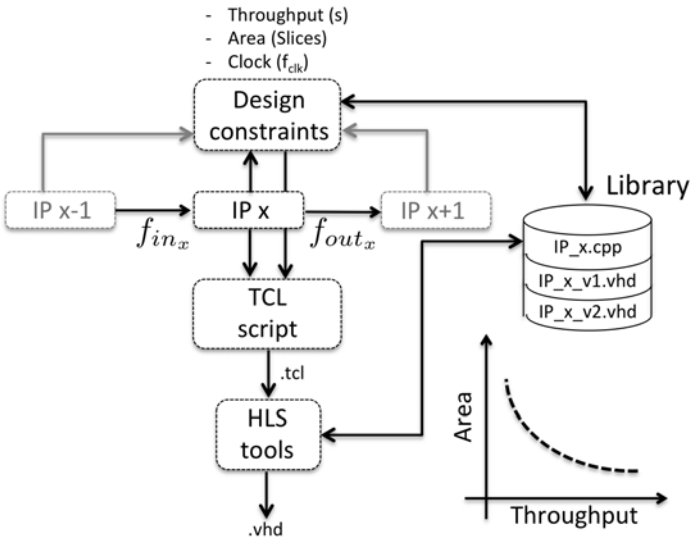

Figure 5.7 – Simplified design constraint management scheme based on a throughput area trade-off.

Catapult [113] and Vivado HLS [114] and the current version of the DSL supports the generation of synthesis scripts for the two of them. However, the HLS-based description of a FB slightly differs depending on whether the synthesis tool is Catapult or Vivado HLS. Indeed, a C/C++ specification intended to be synthesized by Catapult can not automatically be synthesized by Vivado HLS and vice versa. Care must then be taken while specifying the intended synthesis tool.

In this section we have essentially discussed some conceptual aspects of the proposed DSL. Thus, some keywords which enable to model an FPGA-SDR waveform have been introduced. It was shown that the proposed DSL allows the designer to capture several aspects of such a waveform starting from the platform down to the desired datapath. One of the key elements in our proposal is the automatic generation of a control path which suits to the datapath modeled with the DSL. In the next section, we will essentially discuss the features of the control unit that is inferred from a DSL-based description of a given waveform.

5.4 Frame-Based Control Unit

Thus far, the proposed DSL can be viewed as an enabling technology for assembling signal processing FBs that are pre-written in some high-level languages. Indeed, HLS tools are used to synthesize both the RTL implementation of each FB and the communication infrastructure which is mainly composed of FIFO elements. The assembly of FBs into a datapath will be further discussed in the next chapter. However, we have addressed the issue of control at the system-level (waveform-level) by enabling the inference of a control logic from the DSL-based description of a waveform. The control is derived from the data frame and the FB specifications. It is implemented as a Hierarchical Finite State Machine (HFSM) which purpose is to orchestrate the data frame computation through the dataflow graph.

5.4.1 A Hierarchical FSM (HFSM) for FPGA-based Dataflow Control

First of all, a data frame can be considered at distinct levels, namely the bit level, the symbol level and the sample level. In the DSL the content of each field is referenced by an arbitrary variable which default type is the byte. Indeed, this can be seen in Figure 5.4 where a data frame specification was provided. It is also possible to specify the constant fields' content in diverse forms such as a link to a file containing the samples which compose the field or at the symbol level directly in the DSL source code.

As regards the resulting frame, it can be characterized by its duration, its source (e.g. an FB) and represented as composed out of fields. This structure gathers some exploitable information that we have leveraged in our proposal to achieve automatic control path inference. The duration of each field for instance helps generate the read and write clock signals during the appropriate slot of time. In addition, each block within the dataflow graph is meant to perform a given action on a specific set of fields. Once this action terminates, the block may no longer be required and then disabled. For instance, some FBs will address only synchronization blocks while some others will only address data decoding. Such scenarios are quite recurrent in digital transceivers. It is then convenient to control the activation and deactivation of each FB on a per field basis.

The automatically generated controller consists of a HFSM working in both TX and RX modes. Its overall structure is given in Figure 5.8, where dashed lines denote parallel states while solid lines denote sequential states.
In TX mode, the control unit consists of two major states referred to as *super-states*. The first one is the *IDLE* super-state which corresponds to the inactive state of the transmitter. After detecting a start signal from the MAC, it switches from the *IDLE* super-state to the *FRAMING* super-state where data coding is orchestrated. Generally speaking, dataflow transmitters are feed-forward architectures which means less complex to implement as compared to their associate receiver. The *FRAMING* super-state is declined into three parallel sub-states namely, the *CODING* state, the *INSERT* state and finally the *BL-RECONF* state. In the *CODING* state, the dataflow computation which is described in the DSL is performed. The output samples are fed to the DACs (Digital to Analog Converter) prior to an RF modulation. In parallel to the *CODING* state, an *INSERT*

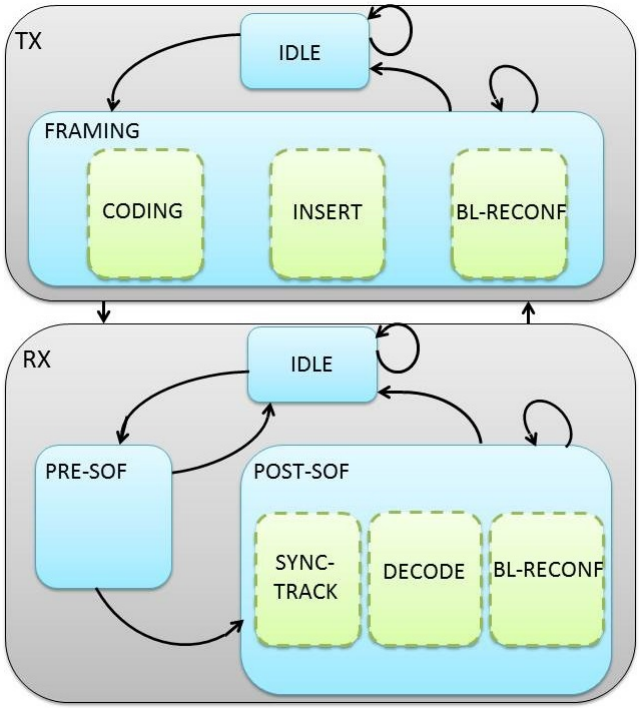

Figure 5.8 – Transceiver Hierarchical FSM.

state manages the run-time insertion of specific data in the frame at the time domain (constant fields). The *BL-RECONF* state handles the block-level (fine-grained) reconfiguration of the transmitter. Indeed, modern standards require certain blocks to be adaptive (ACM), *i.e.* changing their properties on-the-fly. A main stream approach consists in hard-coding all the possible configurations of the block once and then using software controlled switch to select the desired configuration at run-time. A second approach is to reconfigure the block when a given configuration is desired. It is a suitable approach which fits the best to the paradigm of SDR and would require partial reconfiguration capabilities in the context of FPGA.

In RX mode, the FSM is composed of three *super-states*. The *IDLE* state, as in TX mode, denotes the inactive state of the receiver. In this state, the receiver monitors the environment seeking for an incoming signal. Once a signal is detected, by monitoring an RSSI (Received Signal Strength Indicator) for instance, the receiver switches from the *IDLE* state to the *PRE-SOF* state. The *PRE-SOF* state consists essentially of synchronization tasks as imposed by most of the standards. Recall that an *sof* refers to the start of frame delimiter. Once the system enters the *PRE-SOF* state, a set of synchronization elements must be detected and computed within a certain delay. If not, the system returns in the *IDLE* state. The detection of these synchronization elements is referred to as a *sof* event which is defined in the DSL-based frame specification that was lately introduced. An *sof* detection makes the system switch from *PRE-SOF* to *POST-SOF* where a coherent data decoding is sketched. The *POST-SOF* state is declined into three parallel sub-states namely, the *DECODING* state where most of the signal processing is required, the *SYNC-TRACK* state in which the system keeps on tracking synchronization elements and finally the *BL-RECONF* state to handle the run-time block-level reconfiguration as in TX mode.

5.4.2 Frame-based Control Algorithm

In both modes (TX and RX) or even at a finer grain *i.e.* for each state, a set of dataflow computations is intended. In the context of FPGA, the datapath associated to each state or super-stated is one-time mapped to dedicated resources which are ready to operate as soon as the FPGA is powered on. We have considered a single clock domain for this work as well. Thus, one of the roles of the inferred control unit is to distribute the signals to consistently activate or deactivate the

FBs. Such signals are the *clock*, the *enable* signals and the *reset* signal. We leverage the properties of the data frame together with the intrinsic structure of the datapath (dataflow described in the DSL) to infer such a control unit. The underlying algorithm can be depicted as it follows:
First, a data frame F is perceived as a collection of fields *i.e.*:

$$F = \cup_{i=1}^{N} F_i, \tag{5.2}$$

where F_i denotes the i-th field and N represents the number of fields composing the frame. Each field F_i is characterized by its duration T_i its *constant* or *variable* nature *State* or its transported data *Payload*:

$$F_i = \{T_i, State, Payload\}. \tag{5.3}$$

The duration T_F of the overall frame F is computed as:

$$T_F = \sum_{i=1}^{N} T_i. \tag{5.4}$$

The data-path, at both the transmitter and receiver sides, is viewed a set of interconnected FBs. Each FB is characterized by its latency (L), throughput (TP) and its input and output data rates (f_{in} and f_{out}). Recall that, the notion of throughput in the HLS tools refers to how often (in number of clock cycles) a function (FB) is called. It is associated to an Initiation Interval (II) whose value is basically the number of clock cycles between two consecutive function calls. For each block, whenever the input and output rates are known, the streaming data can be directly handled by using *enable* signals. Moreover, our scenario consists in activating or deactivating each block on a per-field basis given that the computation may happen to be specific to field. Thus, let FB_j be the j-th FB within the dataflow graph:

$$FB_j = \{f_{in_j}, f_{out_j}, L_j, TP_j\}. \tag{5.5}$$

Assuming that FB_j processes the field F_i of duration T_i at an input rate of f_{in_j} and output rate of f_{out_j}, such a block would require being enabled a number of times equal to:

$$K_{i,j} = T_i f_{in_j} = \frac{T_i}{T_{in_j}}. \tag{5.6}$$

To achieve such a behavior, each FB is stamped with a time slot to process a given field whenever this field traverses the graph. The FBs are therefore activated depending on the ongoing field. To this aim, the control unit decides a starting moment for each block in the graph which is referred to as T_{ref_j}. This starting time is computed by considering both the graph structure and the properties (latency and throughput) of each block composing it. Indeed, each state is associated to a datapath and once the system enters a state, the processing starts with a specific block that is labeled as a reference block. The activation moment of the remaining FBs in the data-path is then estimated by computing their distance compared to the reference block based on the latency and the throughput of the blocks preceding them. Figure 5.9 shows a time chart for illustrating the activation of the block FB_j to compute the field F_i during T_i. Its reference time T_{ref_j} is estimated at compile time and accordingly used to activate the block. Subsequently, the activated block receives some *enable* signals, to control both its input and output streams, from the control unit for a duration of T_i. After this time has elapsed, the block can be disabled or may keep on processing the next field if it was specified to process multiple consecutive fields. This scenario is illustrated throughout Figure 5.10. One can see a data frame which is made up of three fields whose duration are known. The dataflow graph that is intended to process this frame is illustrated, as well. In this path, the block FB_1 processes the whole data frame while FB_2 only processes $Field_1$. The remainder of the graph, *i.e.* from FB_3 to FB_6, is intended to

process both $Field_2$ and $Field_3$. Moreover, FB_1 designated as the reference block FB_{ref} which means that the computation starts with FB_1 therefore, $T_{ref,1} = 0$. The activation time-chart is shown at the bottom of the Figure 5.10. It concerns FB_1, FB_2 and FB_3 and shows the delays that must elapse before the activation of each of these blocks. Subsequently, the FBs are activated for a duration which depends on which part of the data frame is being processed. This approach naturally induces the notion of pipelined architecture as it allows an FB to start processing as soon as a data is available at its inputs. Indeed, thanks to the attributes of the data frame the graph can be scheduled in a pipeline fashion where each block is deactivated when no longer required.

The proposed algorithm has first been modeled and validated on the StateFlow [125] environ-

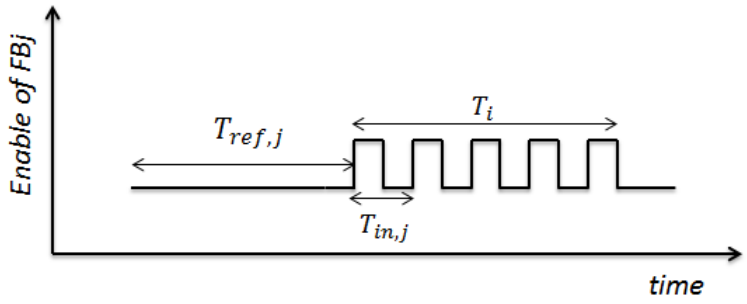

Figure 5.9 – Enable distribution for the activation of FB_j operating on F_i.

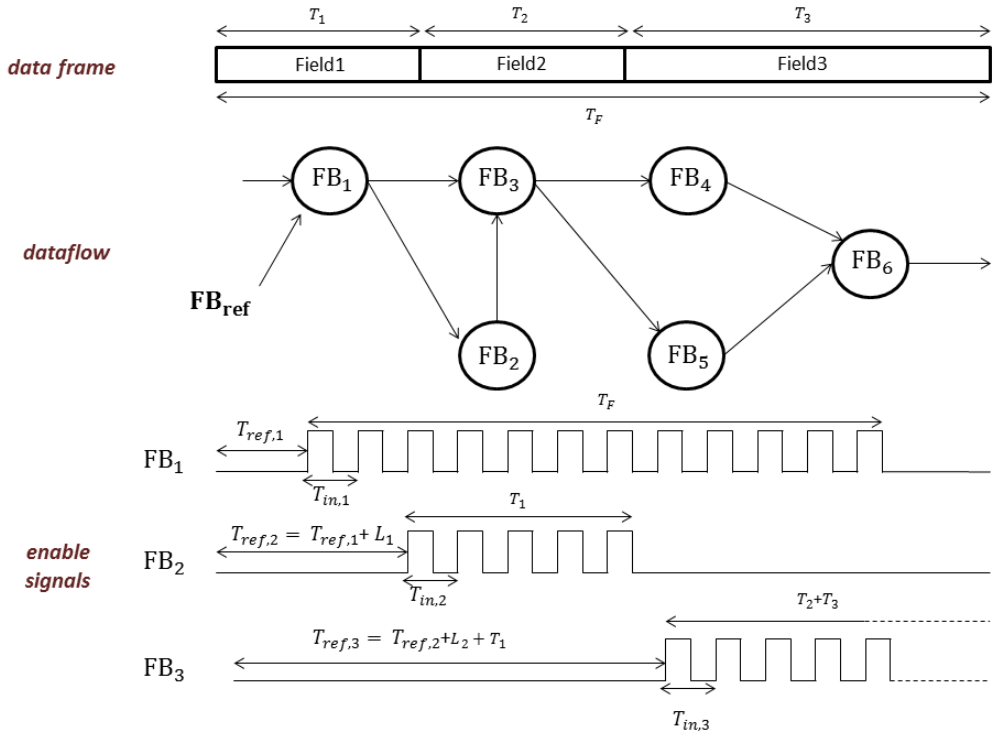

Figure 5.10 – Control algorithm illustration.

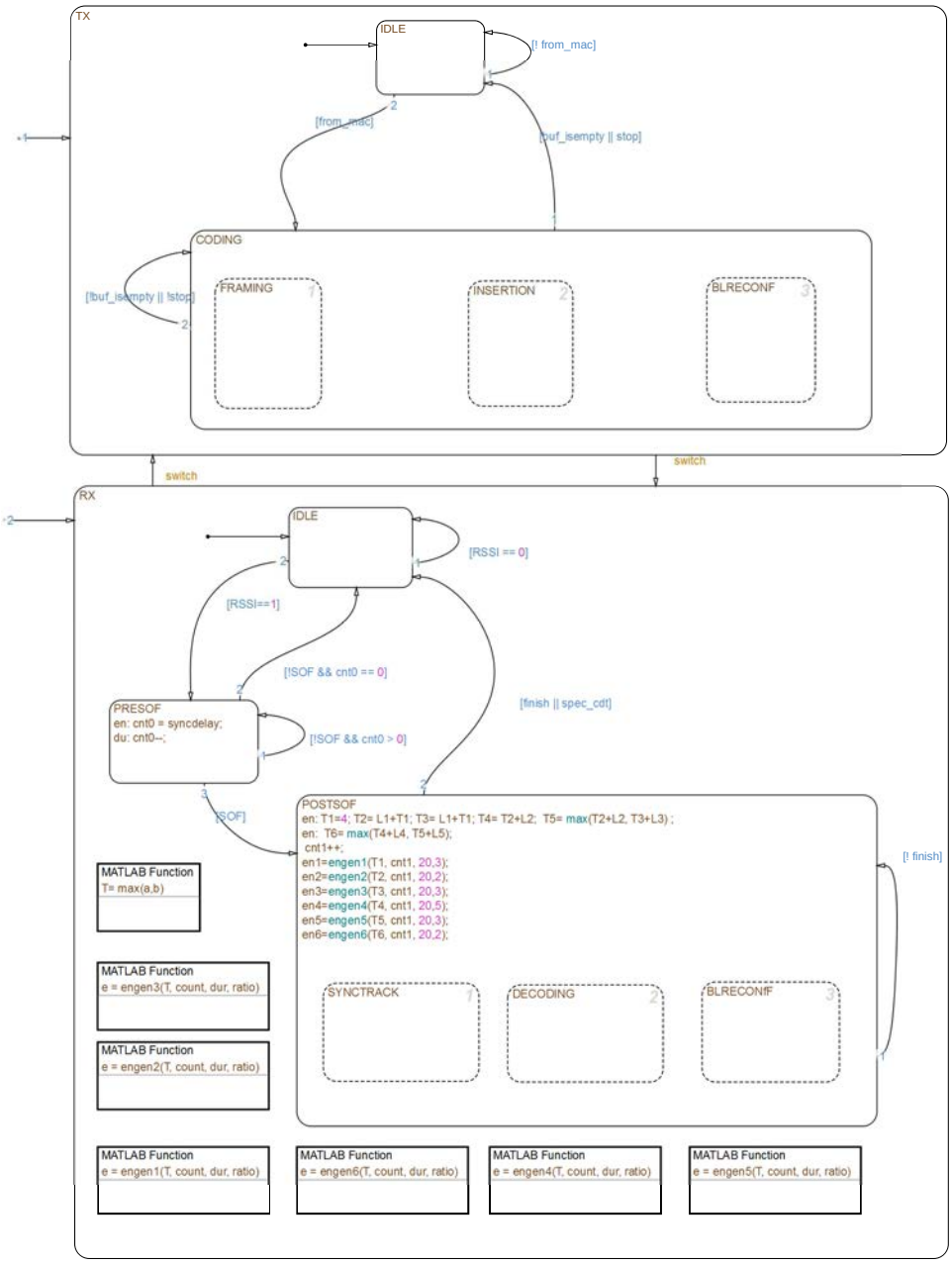

Figure 5.11 – Proposed HFSM modeling and simulation on StateFlow.

ment and subsequently integrated into the proposed SDR PHYs design flow.

5.4.3 Simulation of the proposed HFSM on the StateFlow Environment

The StateFlow environment enables modeling and simulating combinatorial and sequential decision logic based on state machines and flow charts. It is fully integrated to the Matlab/Simulink tool while offering a graphical representation to model state machines. We chose to model and simulate the proposed HFSM so as to further analyze the expected behavior of the control unit regarding the *enable* signals distribution. The model which is shown in Figure 5.11 has been enhanced with some Matlab functions (at the bottom of the diagram) which compute the time references for a set of six FBs. Remember that for each FB, a distance which corresponds to an

activation delay is computed between the block and a reference block.

The architecture of the control unit has thus been modeled and simulated prior to its integration to the proposed SDR design flow. We strongly believe that such an architecture captures the behavior of the frame-based coherent transceivers, especially for an FPGA implementation. In the following section, we will discuss the FBs that have been developed to support our approach.

5.5 Library of HLS/RTL-based Functional Blocks

Our proposal has considered two digital transceivers as case studies. The two transceivers have been depicted earlier in this document, namely the IEEE 802.15.4 and the IEEE 802.11a baseband transceivers. The work has consisted in developing the FBs composing these transceivers with either HLS tools or hand-written VHDL language. We have also leveraged some blocks that were developed by other fellow colleagues or even provided by the HLS tools.

A typical FB has some interfaces which enable to connect with the outside world. We have chosen, in the context of HLS, to define such interfaces as pointers or arrays. Indeed, pointers and arrays are automatically synthesized as memory elements by the HLS synthesis tools. It was then convenient to select these interfaces for implementing dataflow architectures where most of the connections between FBs are done via FIFOs channels. Figures 5.12 and 5.13 show the specification of the matched filter which is required in the receiver of the IEEE 802.15.4 PHY and developed with Vivado HLS and Catapult respectively. In both cases, the filter is developed as an FIR filter whose interfaces are mostly pointers or arrays. They both leverage a particular class of *shift register*, provided by the tool, to buffer the input stream as it is required in a classical FIR implementation. Thus, this specification highlights an advantage of the HLS tools which is the reuse of class of functions which were programmed at a higher level.

The resulting RTL description of a given block usually consists of a top level function which is composed of a core entity, an internal state machine and some interfaces (FIFOs channels). The internal FSM aims at scheduling the function so that it can meet with the requirements at the block-level. Thus, whenever some optimization are selected by the designer, the FSM helps with the actual implementation of the optimization. In other words, loop unrolling or loop pipeling techniques or resource sharing are partly enabled by the internal FSM. The top level function employs classical RTL interfaces, *i.e.*, the *std_logic* interface, the *std_logic_vector* interface and so on. The tools also propose some schematic views of the generated functions which enable to readily grasp they structure. The RTL source code on the other hand is not a readable code even more so because it is usually longer than a dozen of thousands of lines.
We have also implemented the proposed SDR design flow so that it can support some hand written VHDL functions. Such functions must be specified with some input and output interfaces of *std_logic* or *std_logic_vector* types. Further to this, a *clock* signal, an asynchronous *reset* signal

```
 7 void rxfir(sample_rx ich2, sample_rx qch2, sample_tx coeff[OUTPUTFREQ],
 8            data_fir *ich3, data_fir *qch3)
 9 {
10      static ap_shift_reg<sample_rx,OUTPUTFREQ> i_reg, q_reg = {0};
11      i_reg.shift(ich2);
12      q_reg.shift(qch2);
13      data_fir temp_i = 0;
14      data_fir temp_q = 0;
15      for(int i = OUTPUTFREQ-1; i>=0;i-- ){
16           temp_i += (i_reg.read(i))*coeff[OUTPUTFREQ-1-i];
17           temp_q += (q_reg.read(i))*coeff[OUTPUTFREQ-1-i];
18       }
19
20      *ich3 = temp_i;
21      *qch3 = temp_q;
22 }
```

Figure 5.12 – IEEE 802.15.4 transceiver matched filter specification in Vivado HLS.

```
 9 /*Matched filter specification*/
10 void rxfir(sample_rx *ich2, sample_rx *qch2, data_fir *ich3, data_fir *qch3)
11 {
12
13    static shift_class<data_fir, OUTPUTFREQ> i_reg, q_reg;
14    rxfircoeff rxcoeff[OUTPUTFREQ]= {0, 5, 9, 12, 14, 15, 14, 12, 9, 5};
15    i_reg << *ich2;
16    q_reg << *qch2;
17
18    data_fir temp_i = 0;
19    data_fir temp_q = 0;
20    for(int i= OUTPUTFREQ-1; i>= 0; i--){
21       temp_i += i_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
22       temp_q += q_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
23     }
24
25    *ich3= temp_i;
26    *qch3= temp_q;
27 }
```

Figure 5.13 – IEEE 802.15.4 transceiver matched filter specification in Catapult.

and an *enable* signal are mandatory for each block. Our aim is to provide an environment which supports heterogeneous FBs which means that FBs specified by different HLS technologies or hand-written VHDL could be employed in the same project. This was a reason to standardize the interfaces which allow such a scenario. The current version of the DSL-Compiler does not entirely support this heterogeneity however we are still working on its extension. In other words, the DSL editor enables to specify some heterogeneous blocks within the same project but it is not yet entirely supported by its compiler.

5.6 Conclusion

Throughout this chapter we have introduced our proposal for modeling and implementing FPGA-SDRs. The proposal was implemented in form of an external DSL which enables capturing different aspects of an SDR waveform. These aspects range from the platform model down to the datapath of the waveform. Moreover, the conceptual aspects of the proposed DSL have been illustrated with some generic examples. Thus, an FPGA platform model, a data frame model as well as an FB model have been provided.

Afterwards, the architecture of the inferred control path has been introduced and further discussed. It consists of a HFSM which provides the control signal for orchestrating the data computation. Finally, it has been shown how the proposed flow is featured with the HLS technology, which allows rapid prototyping in the context of FPGA.

At the first glance, our proposal can support a conceptual model of an FPGA-SDR. However, the implementation of the waveform should be automatically derived from the model so that it can be claimed to be an SDR design flow. For this purpose, we have defined a DSL compiling framework to support our approach and the compiler's features are discussed in the subsequent chapter.

Chapter 6

The DSL-Compiling Framework

Contents

6.1 Introduction 84
6.2 DSL Implementation 84
6.2.1 Parsing 84
6.2.2 Abstract Syntax Tree (AST) 85
6.3 DSL Compiler Flow 85
6.3.1 AST Verification 85
6.3.2 Waveform Generation 87
6.4 Platform Programming 90
6.5 Conclusion 91

6.1 Introduction

In the previous chapter, we have introduced and discussed the big-picture of our proposal, which consists of an FPGA-based SDR design flow. The proposed flow relies on a DSL that was defined to help capturing the structure of an SDR waveform at a higher level of abstraction. In this chapter, we would like to emphasize more on its compiler by discussing some of its main features. As aforementioned, the Xtext/Xtend framework allows defining a DSL together with its associated compiler. The Xtext/Xtend framework offers a customized editor to specify some DSL-based applications as well. Our design flow has required to develop a compiler which implements some algorithms and also generates several artifacts such as VHDL source code and synthesis scripts. The overall compilation process of a program written in a DSL is composed of two majors steps. The first step consists in producing an intermediate representation of the program also called Abstract Syntax Tree, abbreviated AST. Following the AST generation, it can be further analyzed and later used to produce the desired outputs. Sections 6.2 and 6.3 discuss the two compilation steps while Section 6.4 provides an insight on how a modeled and generated waveform is programmed on the platform. Conclusions are drawn in Section 6.5.

6.2 DSL Implementation

6.2.1 Parsing

A DSL implementation requires first to define an adequate grammar which enables modeling an instance of a domain application. Once a program is written with the DSL, the next step is to make sure that the program respects the syntax of the DSL. To this aim, a lexical analysis is performed by a lexer or a scanner. It consists in converting the program into a set of tokens which represents the atomic elements of the program such as the keywords, the identifiers, variables or the operators. Subsequently, a syntactic analysis takes place to make sure that each sequence of tokens forms a valid statement in the language. These two steps constitute the parsing stage of a DSL and they can be relatively complex to define depending on the features of the DSL. However some tools, referred to as parser generator, generate automatically the parser from a grammar definition. The Xtext/Xtend framework includes such a parser generator.

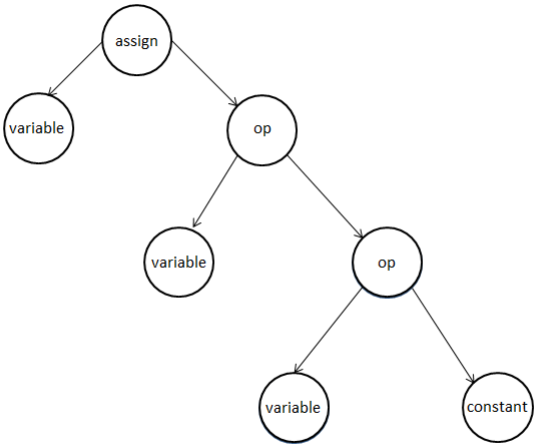

Figure 6.1 – Generic AST representation.

6.2.2 Abstract Syntax Tree (AST)

After a program is parsed and partly checked for correctness, the parsed program is stored in memory with a particular representation. This representation has a tree structure and it is usually referred to as an Abstract Syntax Tree (AST). The AST enables not parsing over and over the same text. With the AST, further verification can be done so as to ensure the consistency of the parsed program. For instance, type checking should be performed after the AST has been generated to make a semantic analysis of the program. In the AST, each node represents a construct of the program. A generic AST is represented in Figure 6.1. It appears as a hierarchical representation of the program (in memory). This example can be interpreted as an assignment operation which includes operators, variables and constant values. Once the AST is successfully checked, it is used for the final step of the implementation which can be either the interpretation of the program or source code generation.

In the Xtext/Xtend framework, the AST is built from two main elements. On the one hand, some Java classes are written for each language construct. Thus, corresponding attributes and methods can be used to customize the interpreter. On the other hand, some annotations which consist in Java code blocks can be added to the grammar specification so as to force some specific actions on the AST.

In summary, the parsing and the AST generation steps represent the early stages of a DSL implementation. They provide a tree-like structure whose nodes correspond, in the specific case of the Xtext/Xtend framework, to a set of Java classes. Following these stages, a DSL designer must implement some methods or mechanisms which allow traversing the AST, to perform some customized analysis and finally produce the desired artifacts. For our FPGA-based SDR design flow, we have developed such a compiler whose features are discussed in the forthcoming sections.

6.3 DSL Compiler Flow

A synoptic of the proposed DSL compiling framework is illustrated in Figure 6.2. Its entry point is an AST which first goes through a customized verification process. After the AST has been successfully checked, it is fed to a compiler which essentially produces some VHDL source codes and some synthesis scripts (tcl scripts). The synthesis scripts are used to interface with HLS tools in order to generate the RTL description of each FB instantiated in the DSL source code. Besides that, we have also implemented the control algorithm that was introduced in the previous chapter and which analyzes the AST and then produces a VHDL description of the control path. Finally the waveform composed of a datapath and a control unit is assembled. The outputs of the compiler can therefore be listed as follows:

— A set of tcl scripts for synthesizing each FB with the HLS tools.
— A datapath in VHDL which instantiates the synthesized FBs.
— A control unit in VHDL.
— A top level VHDL file that instantiates both the generated datapath and control unit.
— At a medium term, a tcl script for programming the FPGA platform.

6.3.1 AST Verification

The AST verification stage is intended to check whether the parsed program (or model) satisfies a set of conditions or not. First of all, a program can be parsed only if it is a valid program. In other words, a program should be grammatically correct so that it can be parsed and used to generate an AST. Some customized verification methods can be performed on the generated AST afterwards. A zoom into a UML-like representation of the AST is illustrated in Figure 6.3. In our design flow, the AST obviously includes information about the platform, the data frame, the datapath together with their interrelations. For each verification, we extract the subgraph representing either the platform or the data frame or the datapath. Thus, we have implemented the following verification methods:

— Platform verification methods which consist partly in checking if the specified FPGA is supported by the provided HLS tools and also checking whether the input or output data

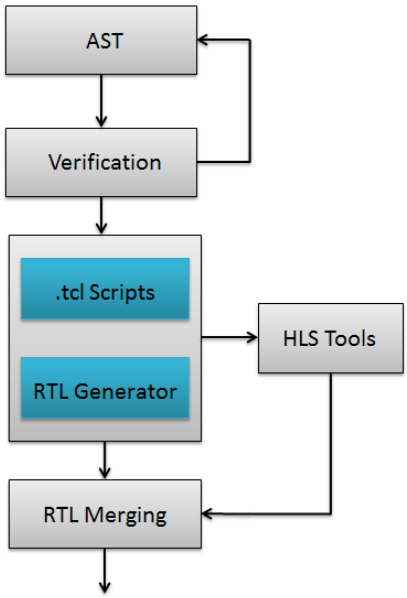

Figure 6.2 – DSL-compiling framework.

samples precision match with the dynamic of the ADCs or DACs precision. If not, an additional re-scaling scenario is performed to interface with the converters.

— Data frame verification methods which consist in checking if constant fields are entirely known at compile-time. Indeed, constant fields can be specified at either symbol level or sample level in the DSL. On the other hand, variable fields are checked to verify whether their duration can be bound at compile-time or not. In the specific case of a variable field with a variable duration a different control scheme is provided for each FB intended to process that field. At the data frame level, the compiler makes sure that all the instantiated fields

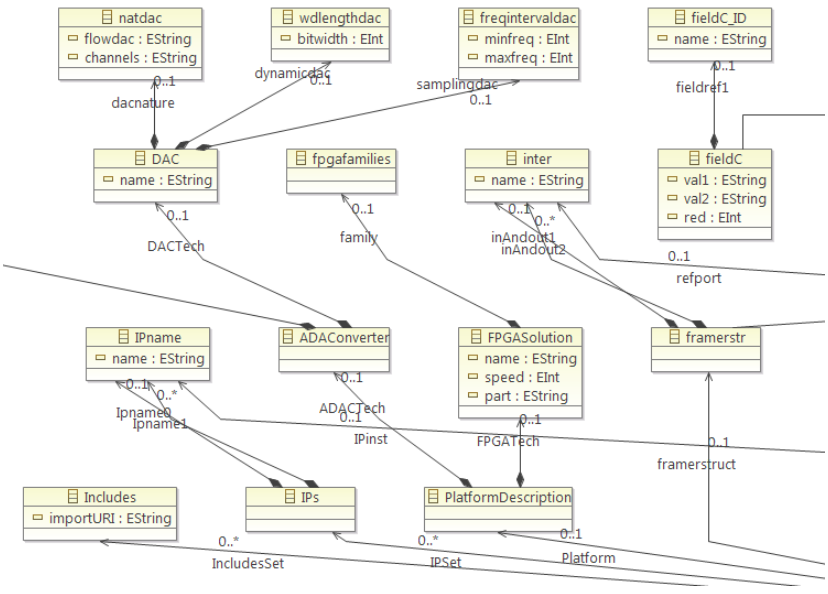

Figure 6.3 – Zoom into the actual AST representation.

are used to build the final frame and also that a start-of-frame (*sof*) has been designated.
— Some methods to check the consistency of the modeled datapath after its extraction from the AST. The methods check among others the data type consistency at the inputs and outputs of each block as well as the possible conflicting interconnections between FBs. For instance, connecting multiple outputs to a single block input is a conflicting situation which results in an error.

We do agree that further verification scenarios should be integrated into the proposed design flow so as to enable a "correct by construction" implementation of the waveform. However, within the time limits allotted to this work, we have not covered all the aspects of this verification stage, and only some of them have been addressed.

6.3.2 Waveform Generation

The implementation of the final waveform consists of different stages. Each of these stages performs a specific aspect of the compiling framework by using the specification provided by the user. The first stage deals essentially with the implementation of the control path which satisfies the specified dataflow graph. After this, each FB composing the flow graph is separately synthesized with the intended HLS tools. Finally the waveform is assembled at the RTL-level.

Control Unit Generation

The automatic control unit generation is the most added value of the proposal. Indeed, we previously argued that HLS tools properly handle the control at the block-level. We have also pointed out the fact that such tools did not address the specification and implementation of state machines which could be used as control units at the waveform level. To tackle this issue, the DSL specification enables the inference of such a state machine.

At a finer-grain, *i.e.*, at the block level, the HLS adds a timing notion into a specified FB through a scheduling process. Scheduling decides when each operation in the DataFlow Graph (DFG) is performed. An example of such a schedule is shown in Figure 6.4 for the computation of $dout = a + b + c + d$. The operation is scheduled in four different clock cycles and registers are inserted between the operations. The control of this schedule is performed with a state machine which is illustrated at the right of the same figure. The state machine requires, in this specific case, four states which correspond to the four clock cycles needed to execute the schedule. These states are often referred to as control steps or c-steps in HLS. Thus, via this representation, one can get a clear idea of the latency and the throughput for each FB. Latency and throughput estimations are usually reported by the HLS tool after synthesis took place.

At a coarser-grain, *i.e.*, at the waveform level, the DSL compiler labels each FB with its reported latency and throughput in clock cycles. Following that, the implementation of the waveform control logic relies on the specification of the data frame which was provided by the user. As a reminder,

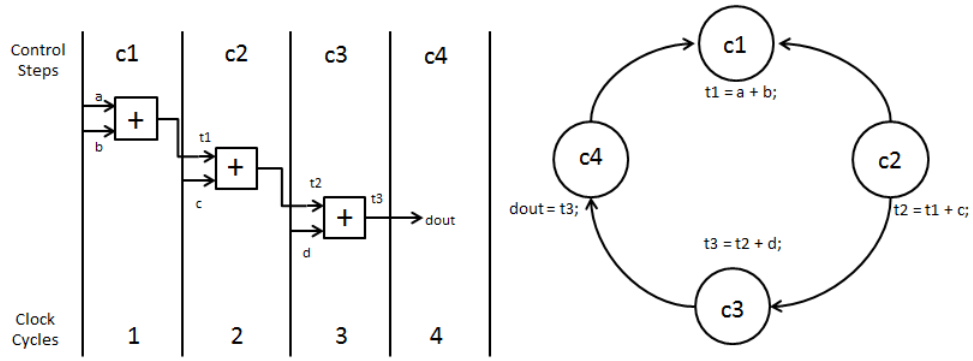

Figure 6.4 – Scheduled Design and Datapath Sate Diagram at the block-level.

the data frame specification within the DSL exhibits a set of features which can be leveraged for inferring control with respect to the specified flow graph. Further to this, the specification of the flow graph uses the frame specification by explicitly mentioning which fields are processed by a given block. Moreover, the origin (an ADC or another FB) of each field inside the graph is mentioned while specifying the block. These information enable to capture the overall structure of flow graph. The compiler uses this specification in addition to the control algorithm that we have lately introduced to build a consistent control path capable of orchestrating the computation. Thus, the current version of the controller generates essentially some *enable* signals within the appropriate slot of time for each FB. Some further extensions so as to handle run-time reconfiguration issues at the block-level are under consideration. Indeed, a *BL-RECONF* state which is part of the control unit and which was introduced in the previous chapter, is intended to manage such run-time flexibility at the block level. However, we have not addressed the precise scenario that would be implemented. Roughly, this state could initiate a data buffering process so as to hold the incoming data during a reconfiguration process.

Functional Block Implementation

Each FB within the flow graph is generated or sourced (for hand-written VHDL blocks) separately as an IP core. To this aim, our scenario consists in generating a synthesis script for each FB. We leverage the tcl scripting language that is usually employed by the Electronic Design Automation (EDA) tools as an alternative entry point. A tcl script is essentially composed of *directive commands*, which are intended to guide the synthesis process. Thus, the synthesis process can be fully automated and tcl scripting can also help quickly generating different solutions for the same design by setting different optimization directives as shown in Figure 6.5. Figure 6.5 is a snapshot of a tcl script generated by our design flow for the matched filter lately introduced. The first part (from the beginning until the *go analyze* directive) informs the name of the FB under consideration together with its associated files. It also sets some default variables which are required for the synthesis process. The second part, from *go analyze* to *go compile*, deals with the underlying technology and the control protocol, including clock, reset or enable signals. The third part, from *go compile* to *go extract*, addresses the FB optimization. For instance, a loop constraint is applied on loop labeled MAC so that it can be unrolled with a factor of five. A constraint on the FB so as to get an initiation interval of 1 (highest throughput) is set as well. One can also note how the FIFOs depths are specified in the same part of the script. Such a scenario enables to achieve local optimization which are mandatory whenever DSE must take place however the depth of the FIFOs should be determined at the compile-time. Figure 6.6 shows a hierarchical view of the generated RTL for a given FB. It is composed of a core function, which comprises a data path and local FSM, and some I/O interfaces which can be implemented as memory storage or simple wires.

Furthermore, we can argue that generating each block in such a way partly ensures the portability of each IP or even the overall waveform over different FPGA solutions. Indeed, HLS tools such as Catapult allow the designer to target different families of FPGAs, thus the specification of the FPGA device within the platform description in the DSL can be used to generate some synthesis scripts that are tailored to the target. On the other hand, for specialized tools like the Vivado HLS tool which only supports Xilinx FPGA, the generated RTL will be definitely further optimized for the target FPGA.

The proposed design flow considers hand-written FB in VHDL RTL, as well. The main goal is to allow the reuse of functional units which have been developed and tested in VHDL. However, no optimization can be performed on such blocks and we have only proposed a selection scenario to select the optimal design. The selection trades-off between area and throughput. Say, we have two distinct hand-written implementations of an FB in VHDL with different features such as latency, throughput or area. Remember that the DSL-based specification of the FB enables to specify some design constraints with respect to throughput at a higher-level. Our selection scenario basically gives the priority to the FB which satisfies the specified constraints. The keyword to specify for hand-written VHDL FBs is *#rtl*. After each FB was synthesized (HSL-based FBs) or sourced (hand-written FBs), the following step addresses the assembly the waveform at the RTL level. This step is discussed in the subsequent section.

```
options set Output OutputVerilog true
options save
project new -name rxfir
flow package require /SCVerify
solution file add {./rx.cpp} -type C++
solution file add {./testrxfir.cpp} -type C++ -exclude true
directive set -REGISTER_IDLE_SIGNAL false
directive set -IDLE_SIGNAL {}
directive set -DONE_FLAG {}
directive set -START_FLAG {}
directive set -FSM_ENCODING none
directive set -REG_MAX_FANOUT 0
directive set -NO_X_ASSIGNMENTS false
directive set -SAFE_FSM false
directive set -RESET_CLEARS_ALL_REGS true
directive set -ASSIGN_OVERHEAD 0
directive set -DESIGN_GOAL area
directive set -OLD_SCHED false
directive set -TIMING_CHECKS true
directive set -PIPELINE_RAMP_UP true
directive set -COMPGRADE fast
directive set -SPECULATE true
directive set -MERGEABLE true
directive set -REGISTER_THRESHOLD 256
directive set -MEM_MAP_THRESHOLD 32
directive set -UNROLL no
directive set -CLOCK_OVERHEAD 20.000000
directive set -OPT_CONST_MULTS -1

go analyze
directive set -CSA 0
directive set -CLOCK_NAME clk
directive set -TECHLIBS {
                        {Xilinx_accel_VIRTEX-6-1.lib Xilinx_accel_VIRTEX-6-1}
                        {mgc_Xilinx-VIRTEX-6-1_beh_psr.lib {{mgc_Xilinx-VIRTEX-6-1_beh_psr part
                        6VLX240TFF1759}}}
                        {ram_Xilinx-VIRTEX-6-1_RAMDB.lib ram_Xilinx-VIRTEX-6-1_RAMDB}
                        {ram_Xilinx-VIRTEX-6-1_PIPE.lib ram_Xilinx-VIRTEX-6-1_PIPE}
                        {ram_Xilinx-VIRTEX-6-1_RAMSB.lib ram_Xilinx-VIRTEX-6-1_RAMSB}
                        {rom_Xilinx-VIRTEX-6-1.lib rom_Xilinx-VIRTEX-6-1}
                        {rom_Xilinx-VIRTEX-6-1_SYNC_regin.lib rom_Xilinx-VIRTEX-6-1_SYNC_regin}
                        {rom_Xilinx-VIRTEX-6-1_SYNC_regout.lib rom_Xilinx-VIRTEX-6-1_SYNC_regout}
                        }
directive set -CLOCKS {
                      clk {-CLOCK_PERIOD 50.0 -CLOCK_EDGE rising -CLOCK_HIGH_TIME 25.00
                          -CLOCK_OFFSET 0.000000 -CLOCK_UNCERTAINTY 0.0 -RESET_KIND sync
                          -RESET_SYNC_NAME rst -RESET_SYNC_ACTIVE high -RESET_ASYNC_NAME
                          arst_n -RESET_ASYNC_ACTIVE low -ENABLE_NAME {} -ENABLE_ACTIVE high}
                      }
directive set -DESIGN_HIERARCHY rxfir
directive set -TRANSACTION_DONE_SIGNAL false

go compile
```

```
directive set /rxfir/ich2:rsc -MAP_TO_MODULE mgc_ioport.mgc_in_wire_wait
directive set /rxfir/qch2:rsc -MAP_TO_MODULE mgc_ioport.mgc_in_wire_wait
directive set /rxfir/ich3:rsc -MAP_TO_MODULE "mgc_ioport.mgc_out_fifo_wait fifo_sz=4"
directive set /rxfir/qch3:rsc -MAP_TO_MODULE "mgc_ioport.mgc_out_fifo_wait fifo_sz=4"
directive set /core/main -PIPELINE_INIT_INTERVAL 1
directive set /core/main/MAC -UNROLL 5
go extract

go allocate
```

Figure 6.5 – Example of tcl script generating RTL code with loop constraints.

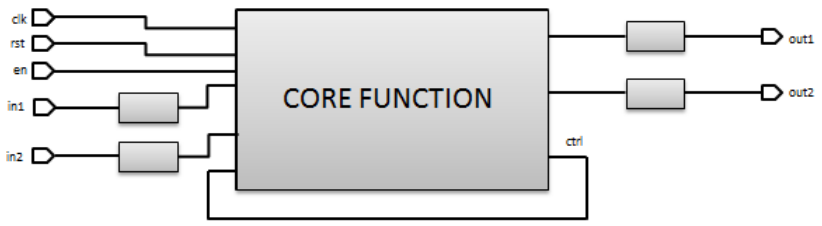

Figure 6.6 – Hierarchical view of generated RTL for a given FB.

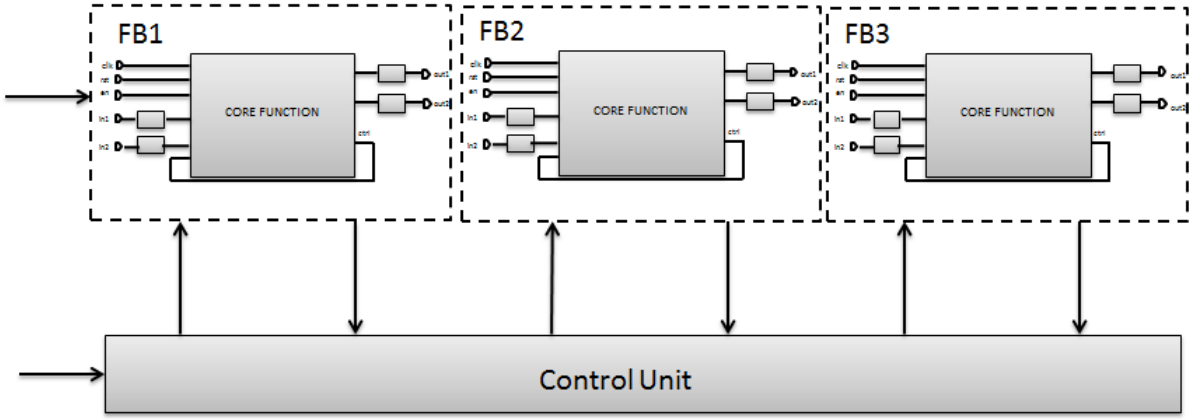

Figure 6.7 – Waveform assembly view.

Waveform Assembly

The final stage of the waveform generation consists in assembling the generated blocks and the control path. After collecting the generated or sourced FBs, the compiler uses the DSL description of the flow graph to perform this step. The datapath is thus built and the control logic is connected afterwards. Some additional checking routines are also performed to make sure of the type consistency between the interconnected blocks. A generic view of the resulting waveform is illustrated in Figure 6.7 where three FBs are assembled into a datapath connected to a generated control logic.

A heterogeneous waveform implementation, *i.e.* made up of FBs generated by different HLS tools or hand-written VHDL, would require defining some VHDL wrappers capable to interface all FBs at the RTL level. To this aim, the wrappers should mainly implement some data exchange protocols. Such VHDL wrappers have not yet been defined and automated in the proposed flow and thus the current version only supports a single HLS tool at a time for a given waveform.

6.4 Platform Programming

The very last stage of the compilation flow consists in programming the waveform on the intended FPGA platform. Indeed, each FPGA family requires specific compilers to synthesize an application bitstream. Thus, this stage turns out to be deeply dependent on the bitstream generation tools that are provided by the FPGA vendors and this creates some limitations since

each tool has its specific compilation flow. However, tcl scripting is also used as entry point for most of these tools but they all come with specific tcl scripts which are usually optimized for each tool. For these reasons, the platform programming stage is so far handled manually in the proposed SDR design flow. We have experienced tcl scripting on our platform however this approach can be rapidly obsolete if we decide to test our solution on a different platform. On the platform which is at our disposal, the waveform testing process is performed with the help of run-time signal analysis tools which are supported by the FPGA compilation flow. Thus, performance analysis can be performed on the final waveform with some analysis tools, such as chipscope. Further details are provided in the next chapter.

6.5 Conclusion

This chapter has introduced the compiling framework which is associated to the proposed DSL. It comprises an additional verification step and a set of functions which purpose is to generate the desired artifacts such as synthesis scripts and synthesizable VHDL-RTL source code. Even though its current version supports such primary aspects of an SDR waveform implementation, in the sense that further extension should be performed, we have nonetheless successfully used it to model and implement two waveforms in demonstration purpose. The experimentation of the flow is discussed in the following chapter.

Chapter 7

A Case Study: DSL-based Specification and Implementation of PHYs

Contents

7.1 Introduction 94
7.2 Testbed Description 94
7.2.1 Nutaq Perseus 6010 Motherboard 95
7.2.2 Radio420X Radio Front-end Daughterboard 95
7.2.3 Perseus 6010 Software Development Tools 95
7.3 DSL-based Platform Modeling 98
7.4 DSL-based IEEE 802.15.4 PHY 99
7.4.1 IEEE 802.15.4 PHY Data-Frame Modeling 99
7.4.2 IEEE 802.15.4 PHY Transceiver Modeling 100
7.5 DSL-based IEEE 802.11a PHY 103
7.5.1 IEEE 802.11a PHY Data-Frame Modeling 104
7.5.2 IEEE 802.11a PHY Transceiver Modeling 105
7.6 The "adaptive" keyword 108
7.7 Validation and Synthesis Results 109
7.8 A few remarks regarding the development time 111
7.9 Conclusion 112

7.1 Introduction

The experimentation of the proposed flow has consisted in modeling and implementing two waveforms. This step aims at proving the correctness of our approach while bringing to light the possible improvements. We leverage a testbed which is made up of FPGA motherboards coupled with radio front-end daughterboards. The FPGA board, referred to as Nutaq Perseus 6010, embeds a large Virtex-6 FPGA claimed to support MIMO transceivers and the Wimax protocol. Furthermore, we leverage some signal analysis equipment such as a spectrum analyzer, an oscilloscope and a Vector Signal Analyzer (VSA) in order to perform some real-time signal analysis. The platform programming is essentially done with the bitstream generation tools provided by Xilinx. Thus, the previously described waveforms (in Chapter 3), namely the IEEE 802.15.4 and the IEEE 802.11a baseband transceivers, have been modeled and programmed on the platform. For each model we emphasize on the model of the intended data frame as well as the model of the flow graph. A DSL-based description of the FPGA platform model of the platform is provided, as well. Section 7.2 introduces the underlying platform while Sections 7.4 and 7.5 discuss the experimentation on the two waveforms. Finally, Section 7.9 makes some conclusions.

7.2 Testbed Description

As mentioned previously, the experimentation testbed consists of FPGA boards associated with RF front-ends. It also involves some signal analysis equipment for a real-time evaluation of the programmed waveform together with some antennas that enable transmission in the 2.4 GHz ISM band. An illustration of the testbed is provided in Figure 7.1 where some of the core elements have been highlighted. In addition to this, Figure 7.2 shows the environmental conditions in which the experimentation was carried out. The subsequent sections outline the FPGA boards, the RF front-ends and finally the software development tools which support the platform programming.

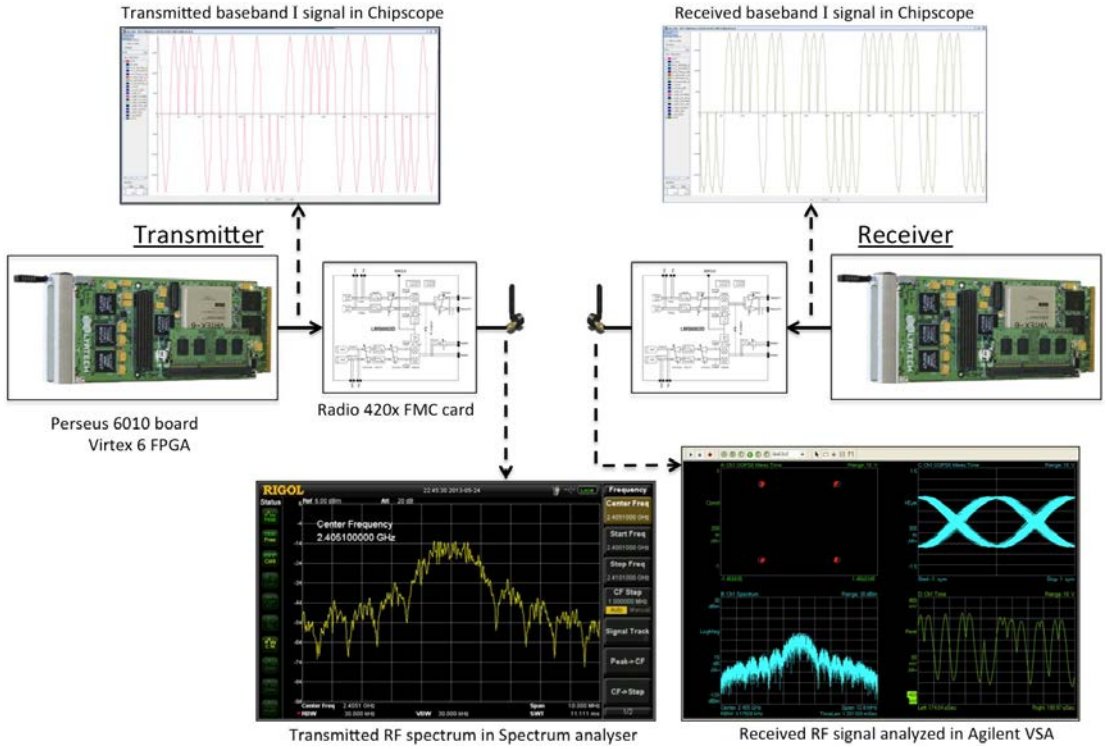

Figure 7.1 – Testbed Description.

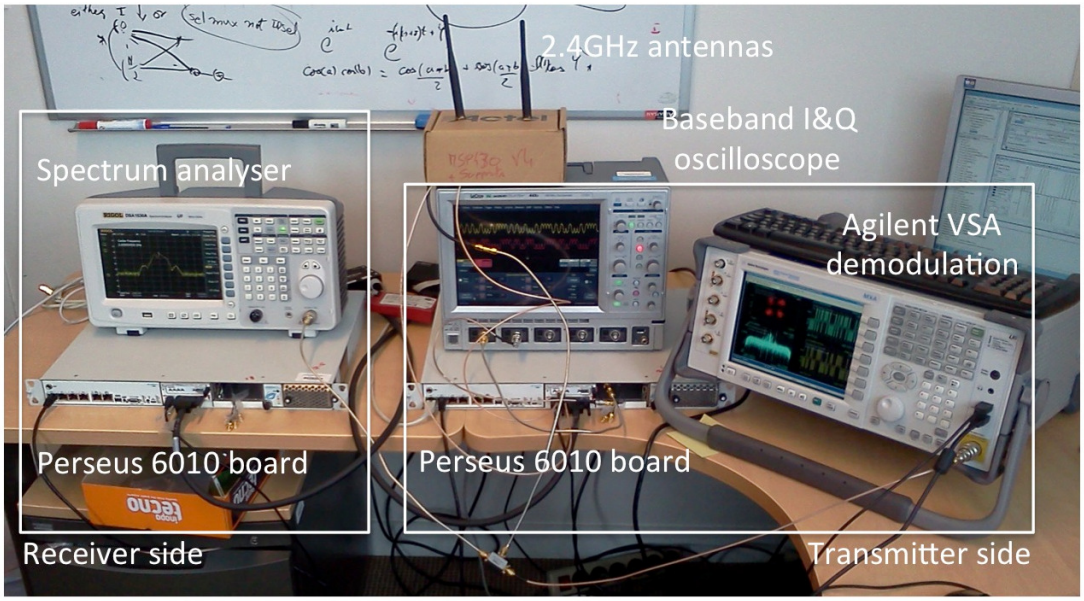

Figure 7.2 – Validation and Testing Environment.

7.2.1 Nutaq Perseus 6010 Motherboard

The Perseus 6010 FPGA board is a commercial platform referred to as Nutaq Perseus 6010 whose block diagram is given in Figure 7.3. It is built around a Xilinx Virtex-6 LXT FPGA and includes some large external memories (1 GB 64-bit DDR3 SODIMM, 128 MB 8-bit DDR3 SDRAM and 64 MB bottom-boot Flash memory) while benefiting from multiple high-pin-count, modular, add-on FMC-based I/O cards. The Perseus 6010 is intended for high-performance, high-bandwidth, low latency processing applications. To this aim, the platform comes with fully compliant AMC backplane connector as well as a high-pin-count VITA 57.1 FMC expension site for I/Os. It includes GTX base clocks (100 MHz, 125 MHz and 156.25 MHz) and a JTAG interface.

7.2.2 Radio420X Radio Front-end Daughterboard

The Perseus 6010 board can be augmented with a RF front-end, for carrier frequency transposition, through its FMC connectors. Our testbed is equipped with two of these RF front-ends referred to as Radio420X, which are claimed to be agile SDR RF transceiver modules. They are designed around the Lime Microsystems LMS6002D RF transceiver which is suitable for multimode SDR and advanced telecommunication. The LMS6002D transceiver is depicted in Figure 7.4. The transmitter demultiplexes the incoming baseband signal and then feeds the samples to a pair of 12-bit DACs. After conversion, the signal is filter on both channels I and Q with a Low Pass Filter (LPF). The transmitter includes some amplifier together with a mixer circuitry for transposition on the required frequency band. Recall that, this work addresses the 2.4 GHz ISM band. The receiver circuitry like the transmitter includes mixers, amplifiers and LPFs. It is featured with a pair of 12-bit ADCs and a multiplexer for multiplexing both channel I and Q samples before transmitting them to the baseband receiver. Finally, the Radio420X relies on Serial Peripheral Interface (SPI) ports and parallel I/Os for controlling the overall behavior.

7.2.3 Perseus 6010 Software Development Tools

The software development tools which support the Perseus board are of two types. The first type is the optional Perseus Model-Based Design Kit (MBDK) which allows to program digital signal processing systems in the FPGA with the Matlab/Simulink design environment and extensive IP libraries from Xilinx. The second software toolset is the Board Software Development Kit (BSDK)

which is a framework for embedded applications development. It includes the Xilinx Platform Studio (XPS) software tool which allows the designer to build, connect and configure embedded processor-based (MicroBlaze) systems.

Model-Based Design Kit (MBDK)

The MBDK flow relies on the Xilinx system generator to generate the low-level FPGA implementation. Thus, with MBDK a design can be developed with the help of specific Xilinx's and Nutaq's blocksets integrated in a Simulink library. Figure 7.5 shows the structure of an IEEE 802.15.4 transceiver implemented with the MBDK flow. The transceiver was first described and tested in handcrafted RTL-VHDL and then the MBDK blocksets were used to encapsulate each function of the transceiver so as to create the graphical representation of the transceiver. As the reader can note, the transmitter is composed of a Symbol-to-Chip FB (ChipGen) followed by a shaping filter (TxFilter). The Symbol-to-Chip FB (ChipGen) reads raw symbols from a memory. A Delay FB (Sample Delay) is inserted on channel Q after the filter and then, channel I and Q are multiplex and fed to the DACs. The receiver first demultiplex the incoming baseband signal in two channels I and Q. Channel I is delayed in order to compensate the delay inserted between the two channels at the transmitter. Subsequently, a matched filter FB (RxFilter) processes the data from channel I and Q and its output data are fed to the synchronization FB (Synchro). The Synchro FB computes the preamble and issues some timing and frequency information. After synchronization, a Decision FB performs a set of sliding correlation to recover the transmitted symbols. In addition, the radio front-ends on both transmitter and receiver can be handily connected to the transceiver through the graphical interface. However, it remains an inactive block which informs about the connection at the time of bitstream generation. A chipscope signal analyzer can be configured through the GUI so as to capture and analyze the real-time baseband signals. Thus, the whole transceiver was graphically represented and then programmed in the FPGA.

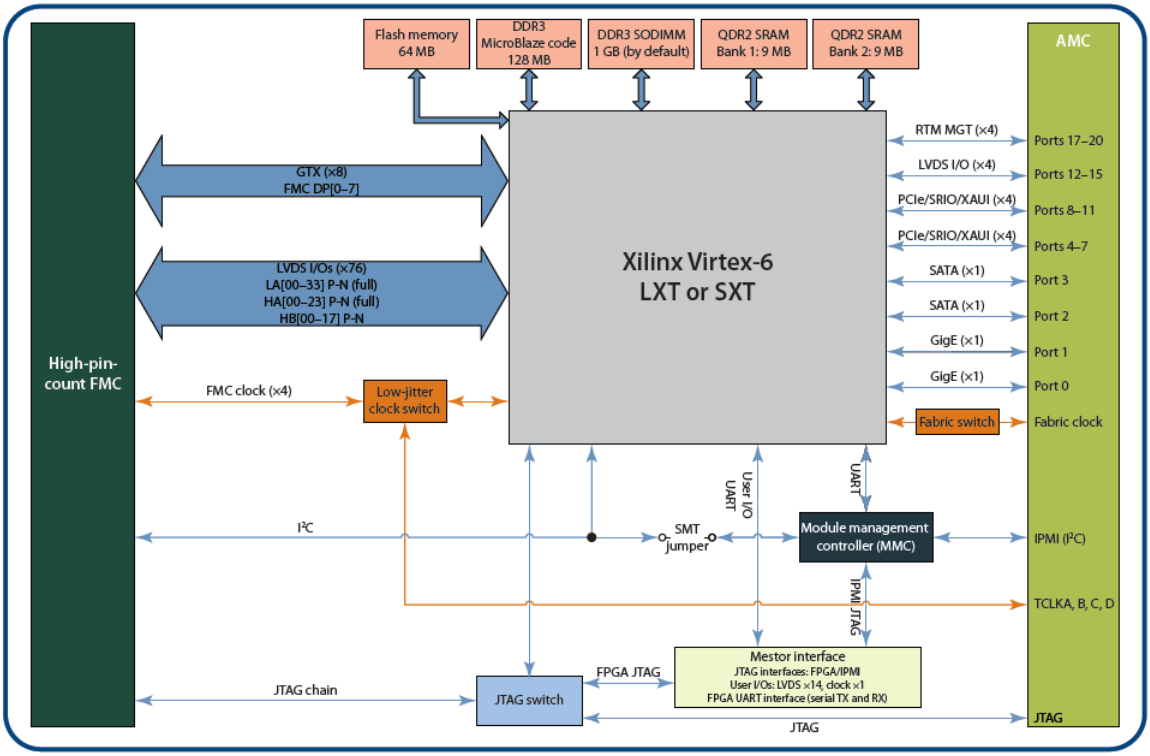

Figure 7.3 – Perseus block diagram

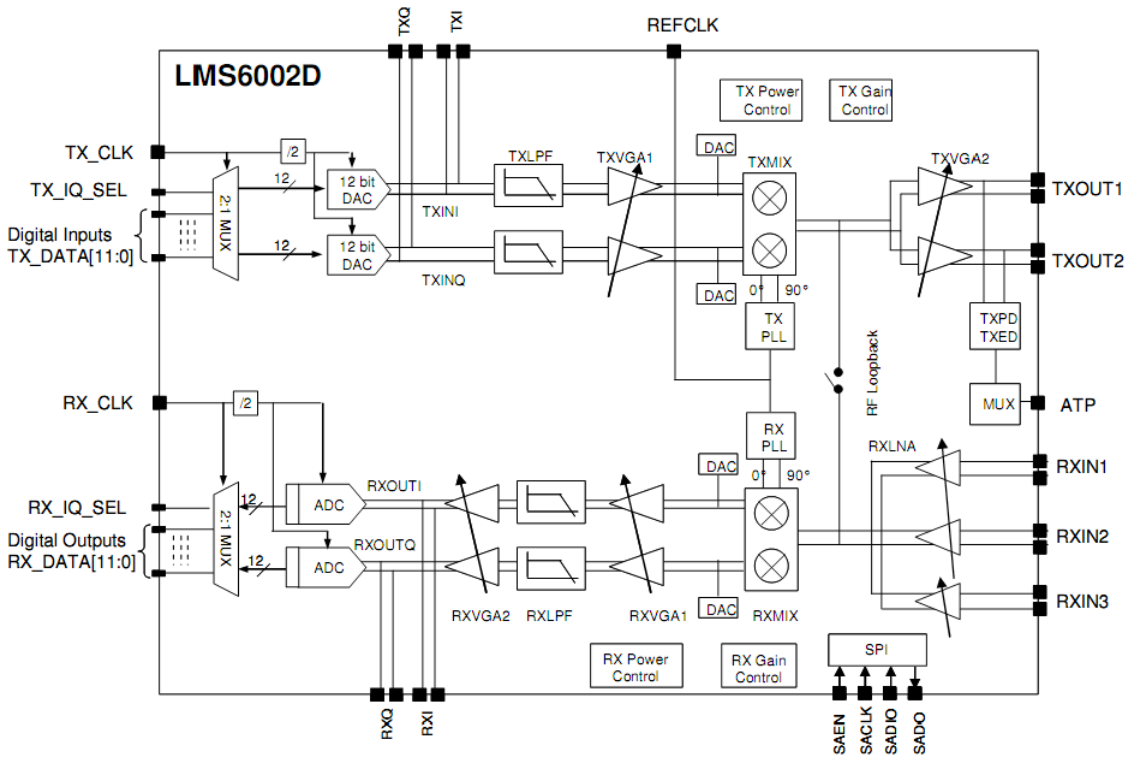

Figure 7.4 – LMS6002D transceiver block diagram.

Board Software Development Kit (BSDK)

The BSDK flow is a script based low-level design kit, which relies on XPS. It enables to program the desired design around a MicroBlaze softcore which can be entirely configured by using C programs or a set of commands/directives through a Command Line Interface (CLI). The instantiated MicroBlaze can significantly help for the development of the rest of the stack protocol while the implementation of the transceiver is done in the reconfigurable logic part connected to the MicroBlaze via a logic bus. This architecture is illustrated in Figure 7.6 which was taken from the Nutaq website. At the left side of the Figure, one can see the user application which can be configured from an external or an embedded PC. Thus, software tools like CLI, SDK or GNU Radio can be used to define the application intended to be run on the platform. At the right side of the same Figure, one can see the internal configuration of the FPGA where the User Logic part is intended to host and run the user's RTL transceiver while being connected to the MicroBlaze softcore through an AXI bus. We have experienced with the BSDK flow by programming the IEEE 802.15.4 transceiver. The MicroBlaze application was specified with both C programs and CLI commands. The RTL transceiver was programmed in the user logic section whit the help of XPS.

MBDK and BSDK in a nutshell

The Xilinx System Generator, which provides system modeling and automatic code generation from Simulink and Matlab, is a key component of MBDK design kit. Furthermore, it provides many features such as resource estimator or hardware co-simulation. The designing is quite straight forward as a drag-and-drop due to Simulink GUI, but it uses a lot of computer resources to compile and run the program. FPGA design is restricted due to available Xilinx blocksets in Simulink library hence for extending the design, Xilinx black box can be helpful, but it may encounter some VHDL integration problem.

On the other hand, BSDK is a low-level designing approach which uses less computer resources.

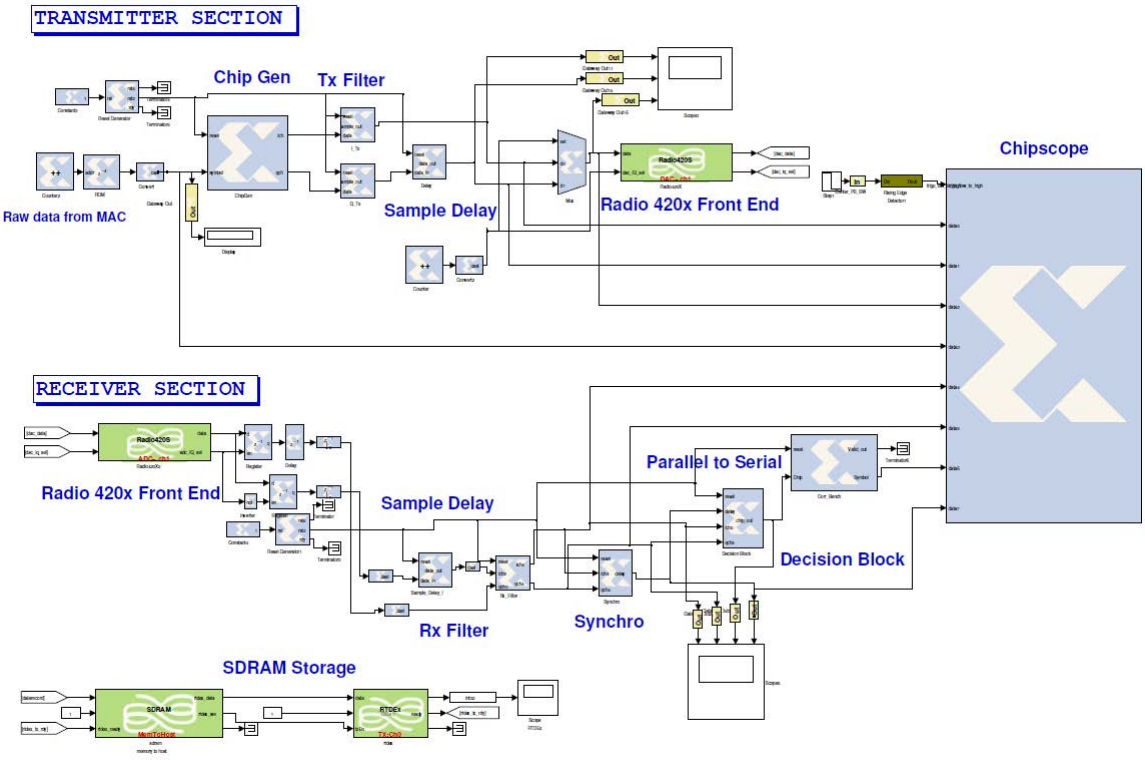

Figure 7.5 – MBDK-based hand-coded ZigBee transceiver design.

However a BSDK designing needs a good understanding of Xilinx tools and HDL programming. BSDK design has a full flexibility of designing due to inclusion of custom IP. Moreover, BSDK design in XPS can also be included to ISE software, which provides a full summary on the FPGA design like resource estimation.

In our opinion, the MBDK flow can be fully leveraged in our design flow since it enables scripting as entry point. Indeed, the user application can be entirely programmed with some C source codes or CLI commands while the user logic programming can be handled with XPS (featured with ISE 13.4). Moreover, XPS takes tcl script as entry point, as a result programming the transceiver could be automated by generation tcl scripts for XPS from the high-level specification of the waveform.

7.3 DSL-based Platform Modeling

The Nutaq platform has served as the underlying hardware platform for validating our design flow. It quickly turned out to be necessary to provide a model of the platform since some details could not be inferred uniquely from the model of the waveform. Such a platform model is intended to provide some descriptions of the key components of the platform. A model of the Perseus 6010 platform is given in Figure 7.7. The model provides a characterization of the converters, namely the data precision, the sampling bandwidth and their serial/parallel nature. The data precision enables to automatically re-scale the data at the outputs and the inputs of the baseband transceiver while the sampling bandwidth informs of the limits of the information signal that can be properly recovered. The nature of the converters tells us whether they are interfaced with the outside world by using serial or parallel connections. In each of these cases, multiplexing and demultiplexing scenarios can be forecast and automatically handled.

A model of the intended FPGA fabric is provided in the specification, as well. It requires providing the device family, its speed grade as well as the exact reference (*part*). The DSL compiler exploits these information when generating the tcl scripts for each FB.

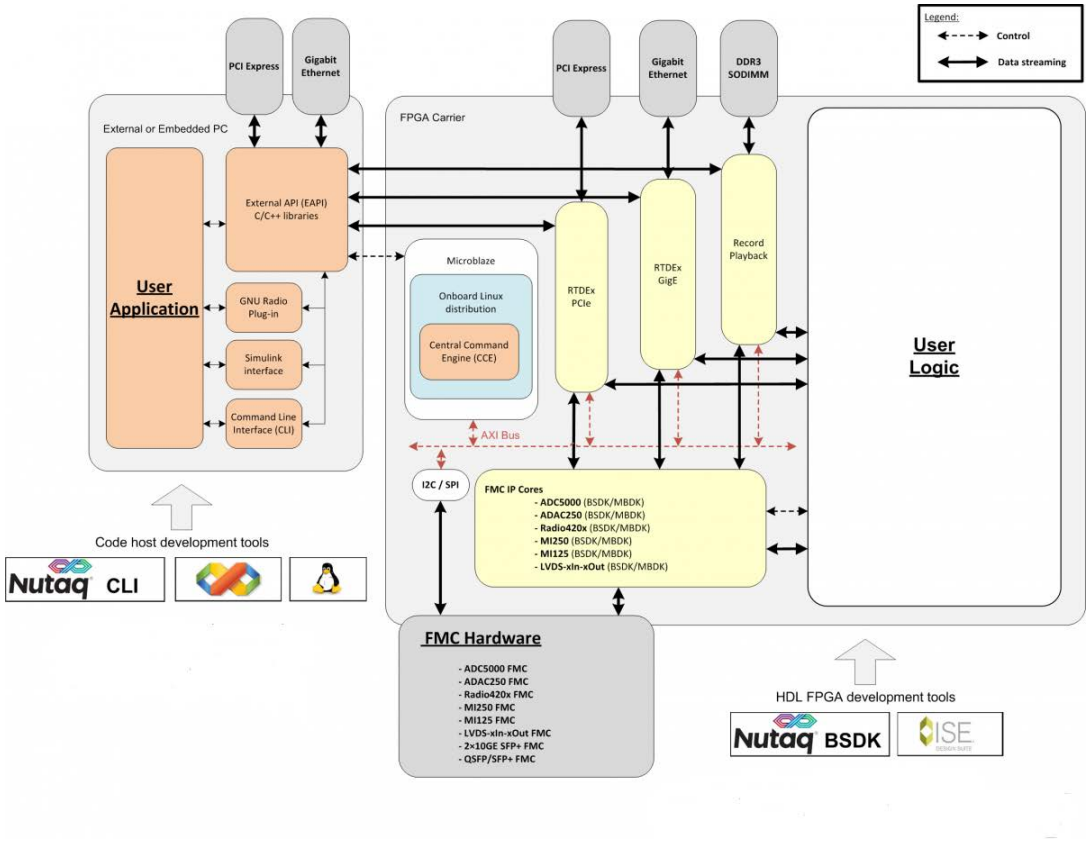

Figure 7.6 – MBDK-based hand-coded ZigBee transceiver design.

Thus, a lightweight description of the platform is given at a high-level of abstraction to enable inferring some hardware element at a lower-level while generating appropriate tcl scripts for the HLS synthesis tools. However, the platform modeling part of the DSL was inspired from the unique Perseus 6010 platform experience. It would be more convenient to validate our approach on a more important variety of platforms. The platform modeling step is followed by the data frame and the datapath models. The following two sections outline those steps for the two previously introduced waveforms, namely the IEEE 802.15.4 and the IEEE 802.11a.

7.4 DSL-based IEEE 802.15.4 PHY

The IEEE 802.15.4 standard has been introduced in the Chapter 3 of this document. As a reminder, it specifies both the MAC and PHY layers of the ZigBee technology which is meant for low rate and low power wireless communications. We have considered its PHY layer for evaluation purpose and we have thus modeled and implemented a compliant baseband transceiver with the proposed SDR design flow.

7.4.1 IEEE 802.15.4 PHY Data-Frame Modeling

The IEEE 802.15.4 PHY data frame is illustrated in Figure 7.8 as it is described with the help of the DSL. It consists first of a *PREAMBLE*, made up of a regular pattern of a repetitive known data symbols. Recall that a symbol refers to a group of 4 bits in this PHY. The *Start of Frame Delimiter (SFD)* field, which follows the *PREAMBLE* is appended for symbol synchronization and consists of a pattern of two known symbols hence a fixed size/length. The *SFD* field is followed by a *PHR* which informs the size in bytes of the conveyed data payload. Its size is fixed but nonetheless its content may vary since the size of the data payload is variable. The *DATA* field

```
/* ADC specification */
ADC LTC2641ADC {                                  /*ADC declaration*/
  precision     12 bits;                           /*ADC input data precision*/
  bandwidth     [1 MHz - 80 MHz];                  /*ADC bandwidth*/
  nature        serial;                            /*ADC's feature*/
}
/* DAC specification */
DAC LTC2641DAC {                                 /*DAC declaration*/
  precision     12 bits;                           /*DAC input data precision*/
  bandwidth     [1 MHz - 80 MHz];                  /*DAC bandwidth*/
  nature        serial;                            /*DAC's feature*/
}
/* FPGA specification */
FPGA V6 {                                        /*FPGA declaration*/
  family     VIRTEX-6;                            /*FPGA's family declaration*/
  speed      1;                                   /*FPGA speed grade*/
  part      VLX75TLFF484;                         /*FPGA precise reference*/
}
```

Figure 7.7 – DSL-based Perseus 6010 platform description.

contains the data payload from the upper layer and its size can vary from 4 bytes up to 128 bytes as specified by the standard. At the transmitter, the fields are transmitted from *PREAMBLE* to *DATA* and obviously received in the same order at the receiver.

The DSL description of the IEEE 802.15.4 PHY data frame is given in Figure 7.9. The PREAMBLE consists of a repetition of 8 symbols zero. It has a fixed duration of 128 μs. Likewise, the SFD field is a constant field with a known payload of a fixed duration of 32 μs. Both fields are specified with the keyword *#fieldC*. Conversely, the PHR field is a variable field which is specified with the keyword *#fieldV*. Its content varies at run-time, however it has a fixed size of 8 bits which is known at compile-time. Duration and size are interchangeable since they both imply the same timing notion in the sense that duration is equivalent size multiply by the appropriate symbol period. In the DATA field specification, the length of its payload is specified by an interval. Indeed, the standard do specifies the payload's length in form of an interval. We have added a default duration information in this context to also enable a fixed payload size default implementation of the control path. Finally the data frame is specified as a collection of the specified fields. It is defined as a complex frame, which implies that the output of the baseband transmitter and the input of the baseband receiver should consist of two channels I and Q. Further to this, the PREAMBLE field is designated as the start-of-frame (*sof*), which implies that its detection will be used for synchronization purpose at the receiver as it was previously explained.

7.4.2 IEEE 802.15.4 PHY Transceiver Modeling

After the frame modeling step, the transmitter is specified by creating the equivalent dataflow graph from the instantiation and consistent interconnection of different blocks. Recall that, each block specification requires mentioning explicitly the set of data fields that it is intended to process as well as their respective sources. The rest of the specification of a given FB consists mainly

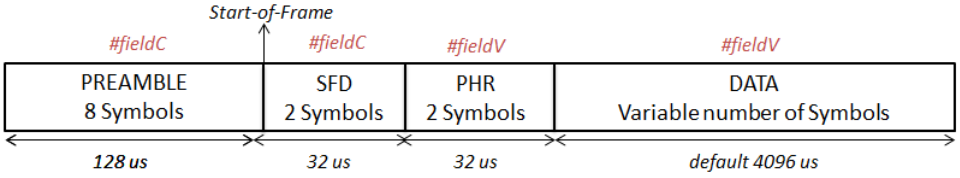

Figure 7.8 – DSL-based IEEE 802.15.4 PHY data frame representation.

in setting up its connections and specifying some design constraints. Thus, the IEEE 802.15.4 PHY transmitter is specified out of four FBs, namely the Modulation FB, the Symbol-to-Chip FB (ChipGen), the TxFilter FB (Shaping filter) and finally the Delay FB. The specification of the transmitter is given in Figure 7.10. It also includes the specification of the clock as well as the specifications of the intended data rates. The interconnections are performed within each block specification and design constraints are defined as well.

A synopsis of the transmitter was first illustrated back in Figure 3.2 where it was depicted as a feed forward architecture, which modulates the incoming bits prior to transmission. At the transmitter, the data frame description helps refining the actual implementation of the intended datapath. Indeed, whenever a frame from the MAC layer is available for transmission, the proposed scenario consists in framing it so as to create a PHY frame. In other words, some additional fields are computed and appended to the frame. Such fields usually convey some frame specific information, which in the case of ZigBee is the size of the data payload (*PHR* field). Moreover, constant fields such as the *PREAMBLE* field, which are also appended at the PHY, are one-time computed and multiplexed to the rest of the frame at run-time. The DSL provides two scenarios for inserting the constant fields. The first scenario inserts the fields at the symbol-level while the second scenario operates the insertion at the sample-level *i.e.* before DACs. For inserting at the symbol-level, the symbols composing each constant field must be provided in their specification. Recall that, a constant field's specification includes a constant term declaration, which are *zero* and *sfdfield* respectively for the *PREAMBLE* and the *SFD* fields. Those constant symbols must be declared prior to the data frame specification in the DSL. In Figure 7.10 the constants *zero* and *sfdfield* are declared in hexadecimal. Their insertion is implicitly performed at the symbol-level since their respective fields are processed from the chipgen_i FB up to the DAC. Indeed, the modulation_i FB only processes the *PHR* and the *DATA* FB. The insertion of the constant fields at the sample-level is discussed and illustrated in the forthcoming sections. The block diagram of the generated IEEE 802.15.4 PHY transmitter through the proposed flow can then be represented as in Figure 7.11. It includes an additional framing FB, which organizes the incoming data into a PHY frame as well as a constant field insertion FB, which inserts the constant fields, at the symbol-level, in the rest of the frame at run-time. In addition, each block is labeled with the set of fields that it is intended to process. The automatically generated control unit is associated to

```
/*Specification of the preamble field*/
#fieldC PREAMBLE {                          /*Preamble field declaration */
    constant zero;                          /* Preamble constant data */
    redundancy 8;                          /*Highlighting redundancy within the preamble*/
    duration 128 us;                          /*Preamble duration */
}
/* Specification of the SFD field */
#fieldC SFD {                                /*SFD field declaration */
    constant sfdfield;                          /* SFD constant data*/
    duration 32 us;                           /*SFD duration specification*/
}
/* Specification of the PHR field */
#fieldV PHR {                                /*PHR field declaration */
    data phrvalue;                          /* PHR field content data*/
   size 8 bits;                                 /*PHR size or length*/
}
/* Specification of the DATA field */
#fieldV DATA {                                 /*DATA field declaration */
    data datasample;                            /*DATA field content data*/
    maxsize 128 bytes;                           /*DATA field maximal length*/
    minsize 4 bytes;                           /*DATA field minimal length*/
    duration 4096 us;                            /*DATA field default duration*/
}
/* Data frame specification */
complex frame PPDU {                           /*Complex frame declaration*/
    PREAMBLE SFD PHR DATA                     /*Listing all the fields composing the frame*/
} sof after PREAMBLE                           /*Start-of-Frame designation*/
```

Figure 7.9 – DSL-based IEEE 802.15.4 PHY data frame description.

```
/*Clock and rate specification*/
clock clk = 16 MHz;
rate fe = 8 MHz;
rate fc = fe/4;
rate fb = fe/32;
rate fs = fe/128;
rate fc2 = fe/8;
/*Constant fields payload*/
constant zero = 0x0;
constant sfdfield = 0xA7;
   ⋮
/*Modulation FB specification*/
modulation_i:  ip modulation processing PHR DATA from MAC {
   read datain on port src at fb;
   write symbol on port sb at fs;
   synthesis catapult;
   constraint latency 4;
}
/*ChipGen FB specification*/
chipgen_i:  ip chipgen processing PREAMBLE SFD PHR DATA from modulation_i {
   read symbol on port sb at fs;
   write chip_i on port ich0 at fc2;
   write chip_q on port qch0 at fc2;
   synthesis catapult;
}
/*TxFilter FB specification*/
txfilter_i:  ip txfilter processing PREAMBLE SFD PHR DATA from spliter_i {
   read chip_i on port ich0 at fc2 ;
   read chip_q on port qch0 at fc2;
   write delay_connect on port ich1 at fe;
   write DAC on port qch1 at fe;
   synthesis catapult;
   constraint throughput 2;
}
/*Delay FB specification*/
delaytx_i :  ip delaytx processing PREAMBLE SFD PHR DATA from txfilter_i {
   read delay_connect on port input at fe;
   write DAC on port output at fe;
   synthesis catapult;
   constraint throughput 2;
}
```

Figure 7.10 – DSL-based IEEE 802.15.4 PHY transmitter description.

the datapath so as to sketch the overall data frame transmission process.

The receiver DSL-based specification comes after the transmitter specification. Each FB composing its flow graph is instantiated and consistently interconnected with the ADC or another FB. A synoptic of the receiver was illustrated in Figure 3.3. It is composed of a matched filter, which reshapes the signal from the ADC while improving the SNR. Its output samples are first fed to a synchronization FB which purpose is to recover timing and frequency information based on the preamble field. Once the synchronization is performed, a decimation and compensation FB is activated to process the rest of the frame issued by the matched filter. Its purpose is to down-sample the incoming samples at the right period while compensating the phase error on each sample. Following this stage, the compensated samples are fed to a correlation bench which recovers the transmitted symbols by computing a sliding correlation between the incoming samples and the 16 possible chip sequences. Finally the demodulator block converts the decoded symbols into bits. Figure 7.13 highlights the specification of the receiver with the DSL. Thus, four of the aforementioned FBs are specified and consistently interconnected. The first FB is the matched filter (rxfir), which processes the whole PPDU frame from the ADC. It reads the input data at the rate *fe* and write its output data at the same rate. It is set up with a throughput constraint of two cycles which implies that a new iteration of this function should be started every two clock cycles. Subsequently, the synchronization FB (synchro) is instantiated as processing the *PREAMBLE*

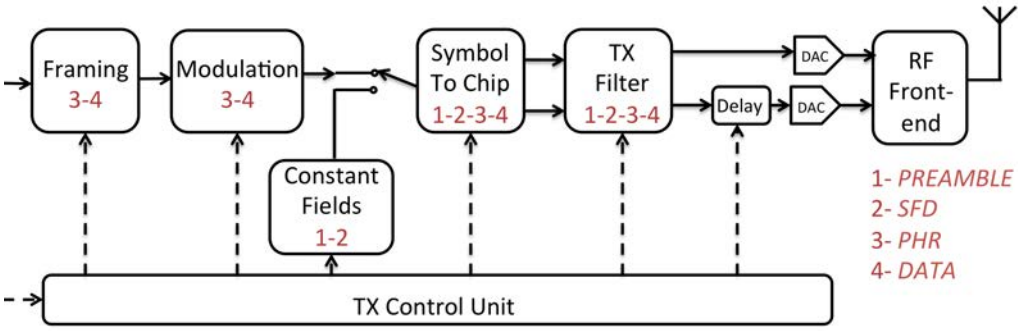

Figure 7.11 – DSL-based IEEE 802.15.4 PHY transmitter implementation.

field issued by the matched filter. Thus, this block reads its input data from the filter's output at the rate *fe* while writing its output data in an *event* interconnection. Indeed, the outputs of the synchronization FB are not part of the data frame, thus they are declared as an interconnection of type *event* which do not require a rate specification. In addition, the synchronization FB is set up with a latency constraint of PBDETECLDL that is a delay after which the block must output the computed synchronization data (optimal sampling period, phase error). Otherwise, the data frame is dropped. When the *PREAMBLE* is successfully detected, the rest of the data frame is fed to the decimation and compensation FB (decimadjust). Indeed, the compensation FB processes the remainder of the data frame, namely the *SFD*, the *PHR* and the *DATA* fields. It reads its input data from the matched filter FB together with the synchronization data issued by the synchronization FB. Roughly, its output data are down-sampled and compensated version of the input data. A throughput constraint of 2 cycles is specified as well. Finally, a correlation bench is performed to recovered the transmitted symbols. Referred to as corrbench_i, this block reads its input data from the decimadjust_i FB at rate f_c and it provides the output symbols at rate f_s. Each of these FBs is intended to be synthesized by the Catapult HLS tool, hence the specification of the tool's reference with the DSL-based instantiation of the FBs. A synoptic of the generated receiver is given in Figure 7.12 where dashed arrows denote control signals and solid arrows denote the *framedata* and *event* like interconnections. In addition, each block is labeled with the set of fields that it is intended to process. The automatically generated control unit is associated to the datapath so as to sketch the overall data frame reception process.

In summary, the data frame specifications have been employed to build an IEEE 802.15.4 compliant transceiver dataflow graph through the proposed DSL. Indeed, each of the FBs composing the graph has been described by including the frame attributes as well as some design constraints. Recall that such a specification intends to enable the inference of the control unit. The ensuing results are discussed later in this chapter.

7.5 DSL-based IEEE 802.11a PHY

The IEEE 802.11a standard specifies both the MAC and the PHY layers of the WiFi technology. It is a widespread technology which is now part of our daily lives. In addition, the standard is intended for high data rate communication with a target throughput of up to a few dozens of Mbits/s. Its PHY layer implements the OFDM modulation as well as the ACM technique which was introduced in the first chapter. Thus, the IEEE 802.11a transceiver appeared to be a good candidate for the evaluation of our proposal.

7.5.1 IEEE 802.11a PHY Data-Frame Modeling

The IEEE 802.11a PHY data frame is composed of three fields as it was shown in Figure 3.5. The first field consists of a preamble field (*PREAMBLE*), which is appended for synchronization purpose. Indeed, the standard suggests using it for both fine grained and coarse grained synchronization at the receiver. The *PREAMBLE* is thus divided into two subfields, namely the short training sequence and the long training sequence intended for coarse grained and fine grained synchronization respectively. Following the *PREAMBLE* field, the *SIGNAL* field conveys frame-specific information, namely data rate and the payload length. The last field is the *DATA* field which carries the useful information *i.e.* the MAC payload. A representation of the data frame as specified in the DSL is given in Figure 7.14 where the short training sequence and the long training sequence are denoted *SHORTPB* and *LONGPB* respectively.

The specification of the IEEE 802.11a PHY data frame with the proposed DSL is illustrated in Figure 7.15. As the reader can see, the specification of the preamble field is twofold. First the short training sequence (*SHORTPB*) subfield is specified as a constant field with the keyword *#fieldC*. It is followed by the long training sequence (*LONGPB*) specification which is a constant field, as well. Their contents are sourced from different files whose links are given in each field's specification. Indeed, this specification illustrates the second type of constant field insertion at the transmitter. The insertion is operated at the sample level as opposed to the first type where the insertion is operated at the symbol level. The short training sequence is specified as a repetition of 10 known symbols with the keyword *redundancy*. It has a fixed duration of 8 microseconds whereas the long training sequence is the repetition of two known sequences with the fixed duration of 8 microseconds as well. The *SIGNAL* field is a variable field which must be computed at run-time. It is specified as a variable field with the keywords *#fieldV* of a fixed duration of 4 microseconds. The *DATA* field is also specified as a variable field of a variable size specified through an interval. Default duration of 256 microseconds is set as well. The resulting data frame is specified as a complex frame which is composed of the aforementioned data fields. Moreover, the Start-of-Frame (*sof*) is designated after *SHORTPB*.

The IEEE 802.11a PHY data frame specification resemble the IEEE 802.15.4 PHY one, expect for the specification of the constant fields which are directly sourced from a sample file. Indeed, this approach is intended to enable constant field insertion at the sample-level. The rest of the data frame specification is similar on both standards. Another path that we would also like to explore is the automatic pilot insertion which is often required in OFDM-based standards. Indeed, we believe that such a data frame specification can be leveraged to handle pilot insertion at a higher level of abstraction. At the time of writing this report, the author had not yet come to a mature approach for automating this step.

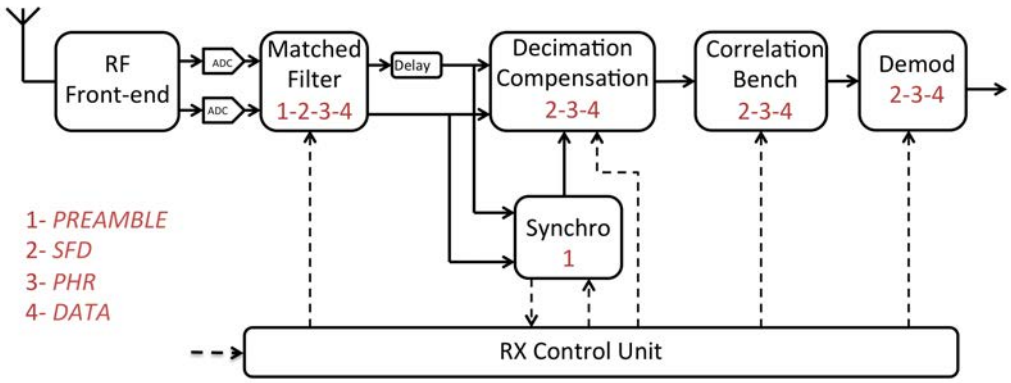

Figure 7.12 – DSL-based IEEE 802.15.4 PHY receiver implementation.

```
/*RxFilter FB Specification*/
rxfir_i:  ip rxfir processing all PPDU from ADC {
    read from_adcI on port ich2 at fe;
    read from_adcQ on port qch2 at fe;
    write ich_rxfilter on port ich3 at fe;
    write qch_rxfilter on port qch3 at fe;
    synthesis    catapult;
    constraint    throughput 2;
}
/*Synchronization FB Specification*/
synchro_i:  ip synchro processing PREAMBLE from rxfir_i {
    read ich_rxfilter on port ich4 at fe;
    read qch_rxfilter on port qch4 at fe;
    write topt_connect on port topt;
    write phiI_connect on port phiI;
    write phiQ_connect on port phiQ;
    write detect_connect on port pbdetect;
    synthesis    catapult;
    constraint    latency PBDETECTDL;
}
/*Compensation FB Specification*/
decimadjust_i:  ip decimadjust processing SFD PHR DATA from rxfir_i {
    read ich_rxfilter on port ich5 at fe;
    read qch_rxfilter on port qch5 at fe;
    read topt_connect on port topt;
    read phiI_connect on port phiI;
    read phiQ_connect on port phiQ;
    read detect_connect on port pbdetect;
    write ich_decim on port ich6 at fc;
    write qch_decim on port qch6 at fc;
    constraint    throughput 2;
}
/*CorrBench FB specification*/
corrbench_i:  ip corrbench processing SFD PHR DATA from decimadjust_i {
    read ich_decim on port ich7 at fc;
    read qch_decim on port qch7 at fc;
    write sb_connect on port sb_rx at fs;
    synthesis    catapult;
}
```

Figure 7.13 – DSL-based IEEE 802.15.4 PHY receiver description.

7.5.2 IEEE 802.11a PHY Transceiver Modeling

Following the data frame modeling, the transmitter dataflow graph is created by instantiating and consistently interconnecting all the blocks composing the graph. A block diagram of the transmitter was previously shown on Figure 3.6. For implementation purpose, an additional framing stage is added as the first stage of the transmitter. A synoptic of the generated transmitter is shown in Figure 7.17 where each FB is labeled with the set of fields that it is intended to process. Thus, once a data frame is available for transmission, framing consists in computing the frame specific information which compose its *SIGNAL* field as it was done on the IEEE 802.15.4 transmitter for its *PHR* field. Indeed, the *SIGNAL* field carries the data rate and the payload length which are known only at run-time. Following the framing step, a Mapper FB converts the data bits into complex numbers representing BPSK, QPSK, 16-QAM or 64-QAM constellation points. This conversion is made according to Gray-coded constellation mappings. The resulting complex numbers are then fed to the IFFT block in groups of 48 for conversion into time-domain. The IFFT FB is intended to insert the pilot symbols, too, which are BPSK modulated in the IEEE 802.11a standard. The standard also requires the insertion of a CP in each OFDM symbol generated by the IFFT FB. Remember that, CP insertion is intended to tackle the ISI issue during an OFDM symbol reception. Finally, an additional FB is integrated for inserting the constant fields, namely the short preamble and the long preamble sequences, at the sample level *i.e.* before the DACs. The whole transmission process is orchestrated by a control unit which is inferred from the DSL specification.

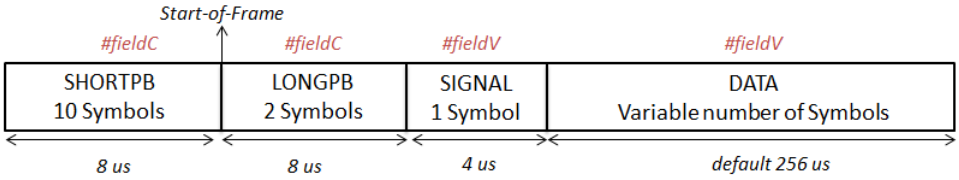

Figure 7.14 – DSL-based IEEE 802.11a PHY data frame representation.

```
/*Short training field specification*/
#fieldC SHORTPB {
    constant ShTrainingSb:  ./shpb.dat;
    redundancy 10;
    duration 8 us;
}
/*Long training field specification*/
#fieldC LONGPB {
    constant LgTrainingSb:  ./lgpb.dat;
    redundancy 2;
    duration 8 us;
}
/*Header(SIGNAL) field definition*/
#fieldV HEADER {
    data hdpayloal;
    size 3 bytes;
    duration 4 us;
}
/*DATA field specification*/
#fieldV DATA {
    data dtpayload;
    maxsize 2312 bytes;
    minsize 0 bytes;
    duration 256 us;
}
/*OFDM PPDU specification*/
complex frame PPDU {
    SHORTPB LONGPB HEADER DATA
 } sof after SHORTPB
```

Figure 7.15 – DSL-based IEEE 802.11a PHY data frame description.

The receiver dataflow graph is also defined through the DSL by instantiating and consistently interconnecting the FBs. A synoptic of the receiver was previously given in Figure 3.7. Its DSL-based specification is provided in Figure 7.16. Ideally, this specification should include a Windowing FB which purpose is to reshape the incoming signal. As a result, this block should process the entire data frame, thus it should be specified with the keyword *processing all* followed by the reference of the frame issued by the ADC. We did not include such an FB in the provided DSL specification and we assume that the receiver starts with a coarse synchronization FB (ctfs_i). The synchronization FB processes the first part of the preamble field in the incoming frame, namely the short training sequence (*SHORTPB*). To this aim, this block is specified as *processing* the *SHORTPB* field from the ADC. The block reads its input data from the ADC and writes the results on two interconnections of type *event*. It is specified with a latency constraint of *SHORTPBDEL* cycles, which is the required delay for computing the synchronization outputs. Once the coarse synchronization FB has successfully performed, the incoming data from the ADC are fed to the CP Removal and Compensation block (cpremoval_i). This FB is intended to remove the CP from each OFDM symbol and compensate these data thanks to the output data of the synchronization FB. It is followed by the FFT FB (fft_i) which converts the OFDM symbols from the time domain to the frequency domain. It deals with the remainder of the data frame, namely from the *LONGPB*

```
/*Coarse Time-Frequency Synchronization*/
ctfs_i:  ip CoarseTimeFreqSync processing SHORTPB from ADC
{
   read adc_data on port frame_in at fe;
   write sync0 on port sync_data0;
   write sync1 on port sync_data1;
   synthesis catapult;
   constraint latency SHORTPBDL;
}
 /*Cyclic Prefix Removal*/
cpremoval_i:  ip CpRemoval processing LONGPB HEADER DATA from ADC
{
   read adc_data on port frame_in at fe;
   read sync0 on port sync_data0;
   read sync1 on port sync_data1;
   write cpRemov on port frame_out at fe;
   synthesis catapult;
   constraint latency 1;
}
/*FFT instantiation*/
fft_i:  adaptive ip FFT processing LONGPB HEADER DATA from cpremoval_i
{
   read cpRemov on port fft_in at fe;
   write cplx_sbl on port fft_out at fs;
   synthesis catapult;
   constraint latency N;
}
/*Fine Time synchronization*/
fts_i:  ip FineTimeSync processing LONGPB from fft_i
{
   read cplx_sb on port cplx_fts at fe;
   write sync1 on port sync_data1;
   synthesis catapult;
}
/*LongPreamble-based Channel Estimation*/
equa_i:  ip ChannelPhaseTrack processing LONGPB from fft_i
{
   read cplx_sb on port fft_in_eq at fs;
   write cplx_sb_eq on port eq_out at fs;
   synthesis catapult;
}
/*De-mapping*/
demapper_i:  adaptive ip DeMapper processing HEADER DATA from fft_i
{
   read cplx_sb_eq on port cplx_in at fs;
   write coded_bit on port bit_out at fc;
   synthesis catapult;
}
```

Figure 7.16 – DSL-based IEEE 802.11a PHY receiver description.

field to the *DATA* field. The FFT FB is set with a latency constraint of N cycles, where N is the number of FFT points. The specification of the fine synchronization FB (fts_i) comes after the FFT specification. It is intended to process the long preamble field (*LONGPB*) so as to make a finer frequency and timing acquisition. This block is left without any constraint specification. After this, the equalizer FB (equa_i) is specified to process the rest of the frame, namely the *SIGNAL* and the *DATA* fields. This block operates the channel equalization based on both the pilot symbols and the fine synchronization outputs. Finally, the demapper FB (demapper_i) converts the compensated complex data into group of bits according to the mapping scheme employed at the transmitter. The receiver's FBs are intended to be synthesized by the Catapult HLS tool, hence the specification of Catapult preceded by the keyword *synthesis*. A synoptic of the generated receiver is provided in Figure 7.18 where dashed arrows denote control signals and solid arrows denote the *framedata* and *event* like interconnections. The receiver is featured with a control unit which distributes the control signals responsible in activating or deactivating each block.

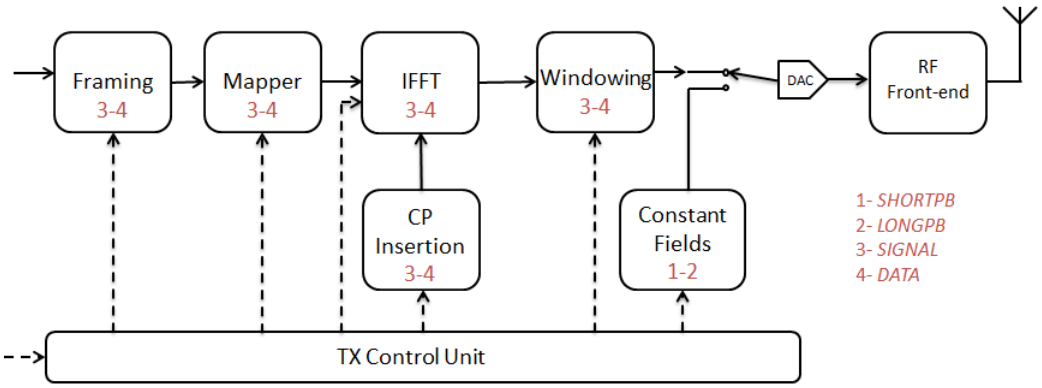

Figure 7.17 – DSL-based IEEE 802.11a PHY transmitter implementation.

7.6 The "adaptive" keyword

We have defined the *adaptive* keyword so as to handle run-time flexible FBs. Indeed, the SDR fosters high reconfiguration capabilities in the intended waveforms both at the block and the waveform (standard) levels. To this end, we have introduced the keyword *adaptive* in the DSL in order to highlight all the blocks which require to be reconfigured at run-time. The ACM technique which was introduced in the first chapter is such an example where run-time reconfiguration of a unique block can be needed. It is usually implemented through multi-mode blocks which can switch from one mode to another at run-time. Conversely, our goal was to leverage the dynamic and partial reconfiguration capabilities of the FPGAs to automate such run-time reconfiguration scenarios in the design flow. At the time of writing this report, we did not come to some valuable results which would deserve to be published. An ongoing research work has been set up to address this aspect of an SDR through the proposed flow. As an example, two FBs have been declared *adaptive* in Figure7.16, namely the fft_i and the demapper_i FBs. Regarding the FFT, its size (number of FFT points) may be required to change on the fly, for instance switching from the 64-point FFT to a 128-point FFT. This behavior could be captured in advance in the DSL specifications. The demapper_i FB could also be required to operate in different modes, for instance switching from BPSK to QPSK or 16-QAM demodulation. The *adaptive* keyword could allow capturing

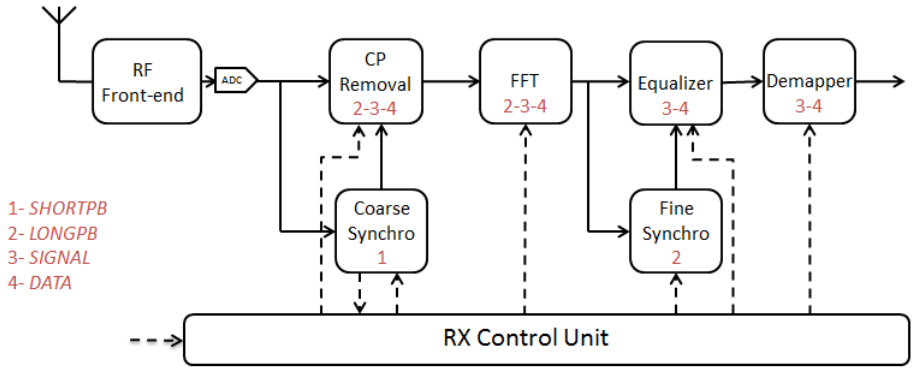

Figure 7.18 – DSL-based IEEE 802.11a PHY receiver implementation.

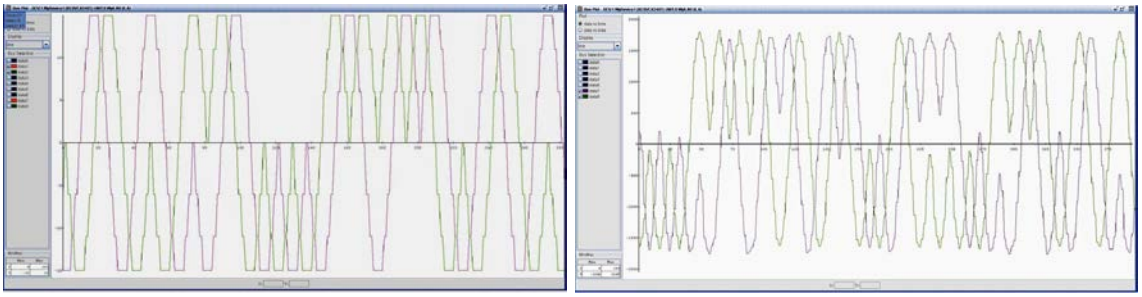

Figure 7.19 – Transmitted (left) and received (right) IEEE 802.15.4 baseband signals.

such behavior in the DSL. The underlying idea is to generate a synthesis script for each working mode of an *adaptive* FB and then generating the equivalent RTL description with the HLS tool. Subsequently, partial bitstreams could be synthesized out of the generated FB and dynamically employed at run-time to switch between two modes of a given *adaptive* block.

7.7 Validation and Synthesis Results

The DSL-based specifications of the two transceivers are compiled by the DSL compiler which first generates a synthesis script (tcl) for each FB composing the transceiver dataflow graph. After this, the scripts are fed to the HLS tool together with the underlying C specification of the FB, in order to produce the equivalent VHDL-RTL description. Subsequently, the DSL compiler assembles the generated VHDL description into a datapath by following the dataflow model specified in the DSL. A control unit is generated for both the transmitter and receiver with the help of the data frame specification and the datapath specification, as well. Control units and datapaths for both the transmitter and the receiver are assembled into a waveform architecture by the compiler.

The waveforms are separately compiled into a bitstream with the help of the XPS tool, which is featured with the version 13.4 of ISE. The IEEE 802.15.4 baseband signals, at both transmitter and receiver, were captured with the help of a chipscope routine and highlighted in Figure 7.19. It is also made possible to observe the same baseband signals directly from the platform by using an oscilloscope connected the UFL connectors that are available on the platform. Furthermore, the VSA equipment as shown in Figure 7.20 has been used to decode the transmitted signal. Indeed, this experiment intended to prove the compliance of the transceiver with the standard. In Figure 7.20, one can observe the constellation that was recovered from the transmitted signal. The corresponding eye diagram is represented as well. Thus, the experiment has enabled validating the transmitter architecture. The associated bandpass signal which is around the carrier frequency (2.4 GHz) is shown on the spectrum analyzer at the right of Figure 7.21. Figure 7.21 also provides a snapshot of the IEEE 802.11a spectrum which was captured with the help of the spectrum analyzer. These results primarily intend to prove the correctness of the synthesized waveforms. However, owing to lack of time we did not get to complete further analysis such as bit error rates or packet error rates.

Table 7.1 gives the synthesis results obtained for a given solution of the two waveforms receivers. These results are collected after place and route and such results aim at validating the proposed flow. Besides that, some resource optimization have been performed at a higher-level while specifying the waveforms with the help the HLS tools given that the DSL can fully leverage the automatic Design Space Exploration (DSE) capabilities offered by the tools on a FB. The exploration aims essentially at meeting performance requirements and represents an important add value to the HLS tools. They make it possible in a few clicks to explore various solutions of the same design. In Figure 7.22, a DSE is performed on a *CorrBench* block, part of the PHY IEEE 802.15.4, in order to trade-off between the area and the throughput of the block. Each curve on this figure corresponds to a level of internal loop pipelining, which represents how often, in clock cycles, a loop iteration is started. Thus, the less the Initiation Interval (II) is the deeper the pipeline will be. One can see from these curves that low throughput in cycles (high in frequency) is achieved when the design is

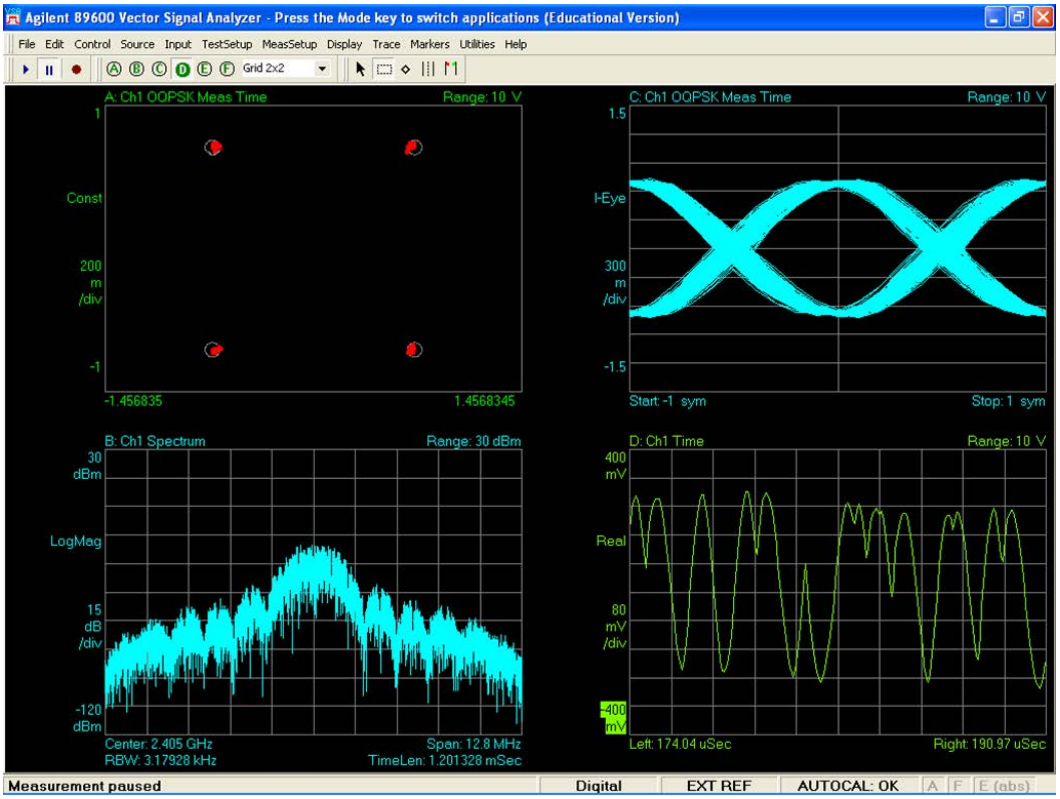

Figure 7.20 – Decoding the transmitted IEEE 802.15.4 signal with VSA.

pipelined to a maximum (II=1). Loop unrolling (U) impacts considerably the design throughput. These results were obtained from the Catapult area estimation feature. Further area estimation were performed with the help ISE tool from Xilinx so as to get a more precise estimation of the area. Thus, Figure 7.23 shows the results regarding the number of LUTs, the number of Flip-Flop registers and finally the number of slices. These results simply confirm the previous one in the sense that they similarly show an increase of resource as the design is pipelined or internal loops are unrolled. In addition to this, Figure 7.24 shows a DSE which was performed for a 256-point FFT with Catapult so as to compare different solutions for same FFT FB. The goal was to trade-off between the achievable throughput (in number of cycle) and the total area. To this end, the DSL-Compiler generates a tcl script for each value of the II an then extracts the area estimation from the synthesis reports which are released by the HLS tools. The exploration of the FFT block similarly shows an increasing area versus the achievable throughput in number of cycles. A suitable solution is automatically selected in such a way to build the desired waveform. This solution exploration scenario can be performed repeatedly for each FB prior to assembling the final waveform. Thus, the designer selects the appropriate solutions which can satisfy the requirements while trading-off between the throughput and area performance.

In sum, the current version of the flow handles block-level configuration, it enables to select the architecture of an *adaptive* block depending on the constraints. This compile-time reconfiguration takes advantage of the HLS capabilities. An extension would be to make the generated controller able to manage at run-time the handover between two configurations of an *adaptive* block. In practice, it could leverage the dynamic and partial reconfiguration features available on recent FPGA devices. To this end, each *adaptive* block could be interfaced with large memory resources to store the streaming data when a reconfiguration is required.

Figure 7.21 – OFDM (left) and ZigBee (right) spectrum.

Table 7.1 – Resource estimation for the IEEE 802.15.4 and IEEE 802.11a receivers.

	Slices	FF	LUT	DSP	BRAM
IEEE 802.15.4	543	1630	1058	1	0
IEEE 802.11a	961	803	2832	8	5

7.8 A few remarks regarding the development time

Throughout this section, we would like to give a feedback on the design methodologies that we have experienced all along this research work. Indeed, we have started this work by developing PHY descriptions from handcrafted RTL-VHDL specifications which required a few thousands of lines of source code. This approach allowed us to provide precise description of the intended waveforms however, it led to some inflexible specifications which were quite complex to modify or optimize. Afterwards, we undertook the development of the same PHY by employing the HLS technology which required a few hundreds of line of source code. This approach has offered more flexibility over the design process since it we could modify or optimize the underlying architectures in a few clicks. The VHDL code generated by the HLS tools has consisted of tens of thousands of lines of source code which are not easily readable by human beings. The development time required for each of these approaches was significantly different. Indeed, hand-written VHDL turned out to be error-prone hence requiring multiple iteration before reaching the target performance. HLS, on the other hand was quite straightforward and required much less time so as to come up with the first working prototypes.

We have built a DSL on top of HLS so as to leverage this technology (HLS) in an FPGA-SDR design flow. The DSL raises the level of abstraction of the design process while adding some new features such as frame modeling. The development time required to specify a waveform with the DSL is relatively short once the FBs are developed. Indeed, the specifications that we have performed through the DSL lie between a hundred or two hundred of lines of source code. Furthermore, the tcl scripts generated by the DSL compiler for each FB have an approximate length of a few dozen of lines of source code.

The validation platform offered two entry points, namely the MBDK and the BSDK. The MBDK flow can be limiting due to its association to the Simulink library which can lack of some signal processing blocks. We have chosen to only use the BSDK flow since it allows scripting for programming the platform. Our goal being to automate the platform programming stage from the DSL specification.

7.9 Conclusion

In this chapter, we have presented the underlying testbed which is composed of two FPGA boards associated with two RF front-end. In addition to this, some real-time signal analysis equipment are employed in the testbed. After this, the modeling process of two waveforms with the proposed DSL has been depicted. This process involves data frame and the transceiver modeling by instantiating and interconnecting the required blocks as well as setting some implementation constraints. The results of the experimentation have been illustrated in form of real-time signals snapshots as well as some synthesis results.

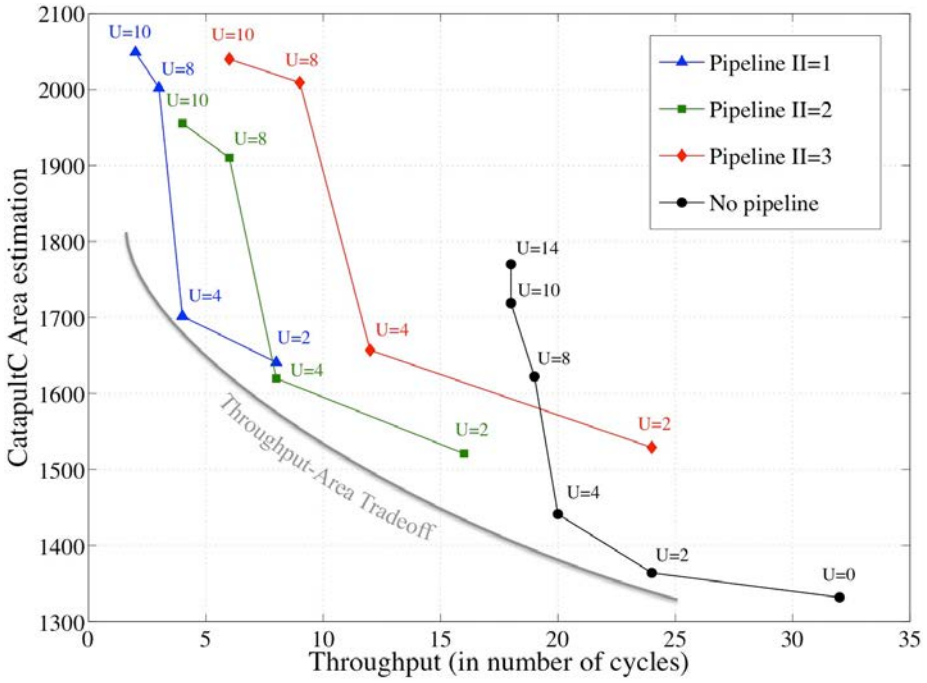

Figure 7.22 – Throughput/Area trade-off of the CorrBench FB.

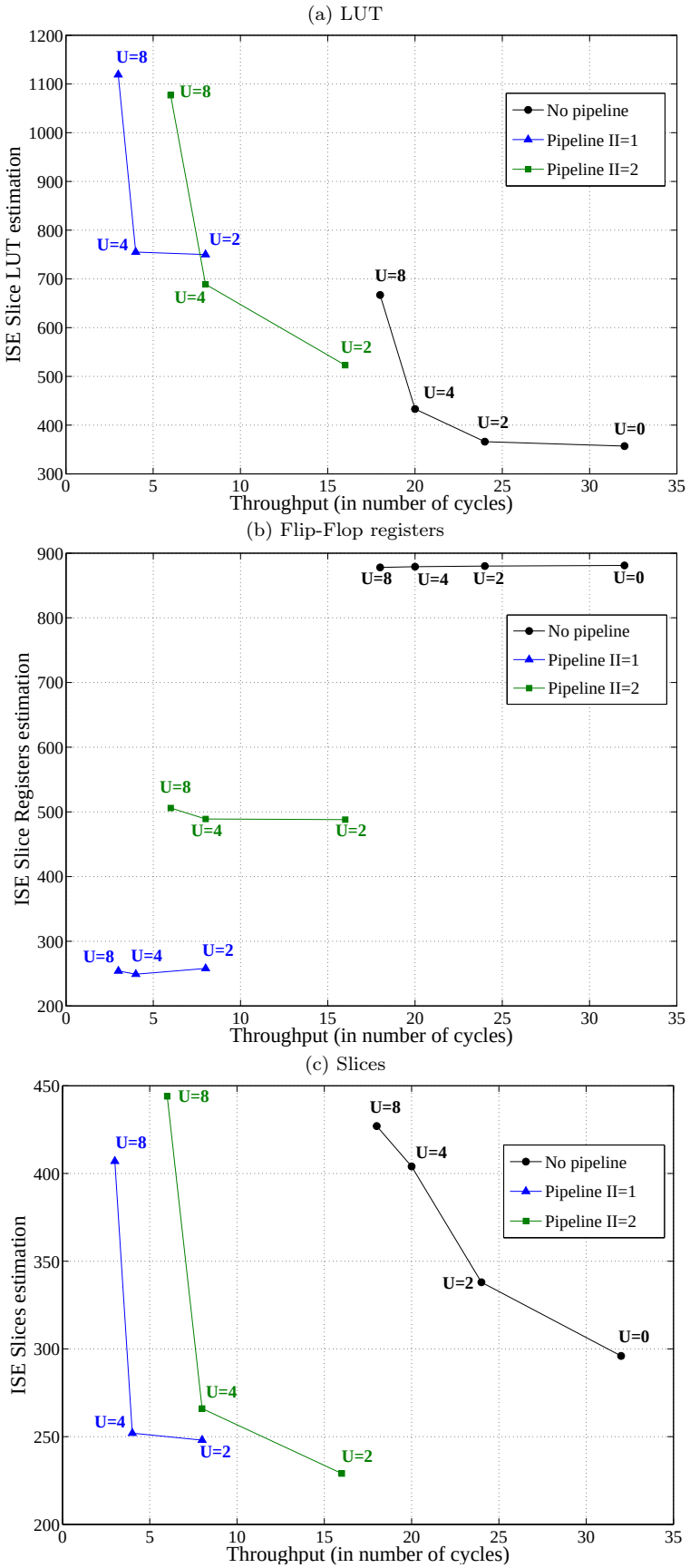

Figure 7.23 – Fine estimation: throughput/area tradeoff of the *CorrBench* block.

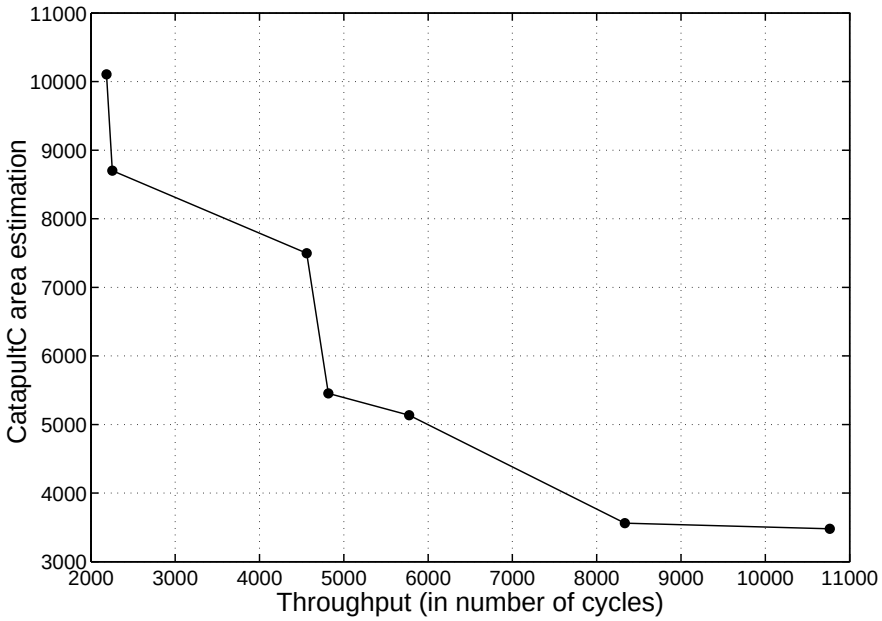

Figure 7.24 – Resource estimation for the FFT FB.

Chapter 8

Conclusion and Perspectives

As the FPGA technology evolves, it turns out to be a good candidate for implementing SDRs. Indeed, it is more and more claimed that recent FPGAs can support complex waveforms implementation which requires considerable resources and throughput. However, the FPGA programming model, which relies essentially on HDLs, has always been a serious impediment to its usage on SDR platforms. Recent proposals have tackled this issue by raising the level of abstraction of the programming model. Referred to as HLS, they enable to program FPGA-based IPs from a HLL such as C/C++ or Matlab. The work presented in this document has consisted in defining and developing a system-level design flow on top of HLS for FPGA-based SDR waveforms.

Summary

Chapter 1 consists of an introduction of the work depicted in this document. To this aim, it provides the general context of this work while listing our contributions.

Chapter 2 has introduced the notion of digital radio from a signal processing perspective as well as the underlying technologies. Thus, some important aspects of digital transceivers have been covered so as to illustrate both the theoretical principles and the requirements at the system-level. Furthermore, the flexible radio, which can operate thanks to the digital radio, has been discussed and illustrated through some examples. Finally, a big picture of the SDR concept was presented together with both the existing platforms and the design methodologies.

In Chapter 3, we have depicted two digital transceivers namely an IEEE 802.15.4 and IEEE 802.11a compliant transceivers. They are intended for low-rate and high-rate wireless communication respectively. The chapter has also provided the generalities of the two standards together with a brief survey of their implementation in an SDR context.

The fourth chapter has dealt with the design methodologies for implementing digital transceivers. Thus, the concepts of MDE and HLS, which both consist in raising the level of abstraction of a design flow, have been broadly discussed as well as examples of MDE projects and HLS tools. Finally, the chapter provided a brief discussion on how to bring together the two concepts into an FPGA-based SDR design flow.

Chapter 5 has introduced our proposal, which consists of a DSL for modeling SDR waveforms intended to be run on FPGA platforms. Thus, each stage of the proposal has been conceptually explained and depicted through generic examples. Furthermore, willing to automate the implementation of the waveform's control path, a data frame based algorithm which has been proposed was covered in the chapter.

The sixth chapter has discussed the underlying mechanisms of the proposal namely, the compiling framework. It was shown that a modeled waveform is represented in form of an AST that is further processed so as to perform some verification or generate some artifacts such as source code or synthesis scripts for the HLS tools. Moreover, the current approach which is used to program the FPGA platform with the generated waveform architecture has been discussed.

The experimentation of the proposed design flow on two well-known waveforms has been discussed in Chapter 7. First, the testbed with which the experimentation took place has been intro-

duced along with its programming methodology. Following this, the modeling stages for each of the two waveforms have been covered. The ensuing results have been analyzed and some conclusions have been drawn.

Throughout this research work, we have defined and implemented an environment for developing FPGA-SDRs. A novel approach, which leverages a data frame model, has been proposed. Its main goal is to enable automatic control inference by considering the intrinsic structure of the frame. Furthermore, this environment has addressed the programmability of the FPGA by taking advantage of the nascent HLS capabilities. A customized compiler has been associated to the proposed flow so as to automate as many steps as possible in the FPGA-SDR design process. Two waveforms have been considered for validation purpose, namely the IEEE 802.15.4 and the IEEE 802.11a transceivers. The waveforms have consisted of FB which have been entirely developed in HLS/C++ and synthesized with the help of the Catapult HLS tool. The experimentation was carried out in the Nutaq Perseus 6010 FPGA board which was connected the Radio420X RF front-end for transposition around the carrier frequency. Thus, we were able to assess the performance of the proposed methodology on a commercial platform while bringing to light some interesting perspectives.

In conclusion, our proposal has contributed in enhancing the programmability of FPGA-based SDRs through a tailored DSL. Moreover, the use of the HLS technology in form of a library of FBs has enabled achieving a rapid prototyping of the actual waveforms. Willing to automate the design flow, we have featured the DSL with a compiling framework which produces the required artifacts. Finally, the proposed flow was validated on two waveforms.

Perspectives

The SDR is an outstanding technology and throughout this PhD thesis we have proposed an SDR design methodology based on the HLS principles. However, different perspectives arise with this brand new approach.

As regards the DSL, the model of the FPGA platform was essentially inspired from the Nutaq Perseus 6010 FPGA board. Thus, the platform modeling stage was first thought to support the integration of the solution in this specific platform. We believe that this model can be further improved with respect to its style and its content. In fact, the current version of the platform model is limited to the definition of the converters along with the FPGA device. Such model could be enriched with the specification of the memory units that are available on the platform or the frequency bands supported by the RF transceiver but still with keeping in mind not to over-specify the platform. Besides that, the specification of the intended data frame must be augmented so as to support dynamic pilot symbol insertion which is common requirement in OFDM-based transceivers. Such a specification should enable to enhance the control unit's capabilities in order to handle the pilot insertion at the transmitter whenever it is required. Moreover, the data frame structure could also be extended to support more complex frame such as downlink frames which may convey data packets for multiple users. The waveform model leverages the data frame attributes while enabling the specification of some design constraints. It is quite a novel approach combining those aspects of a waveform however, we strongly believe that a graphical specification of the waveform would be more intuitive thus requiring a lower learning curve.

The DSL compiling framework supports the generation of different artifacts so as to automate several steps in the FPGA-SDR design process. Thus far, the platform programming step is manually performed since the platform was released with some software dependencies. Nonetheless, scripting could be employed to automate some of the steps in this stage and we are investigating the feasibility of such an approach. Furthermore, with the capability of instantiating a microblaze softcore on the FPGAs under consideration, the waveform's control path could be deployed on the softcore rather than the programmable logic. This scenario would enable to devote more computing resources to the datapath while the control could leverage a software architecture. To this aim, the DSL compiler must be extended so as to support the generation of C scripts intended to program the softcore and the generation of a C-based description of the control path on the

other side. An interesting perspective would be to support the other types of platforms such as DSP microprocessors or GPPs from a DSL-based specification. This would require extending the platform model and supporting such processors from the DSL-based specification.

We are working on enriching the library of HLS-based signal processing blocks. Indeed, it was shown that the HLS technology enables a rapid prototyping of the final waveform. Our proposal takes advantage of the HLS capabilities so as to allow the users to focus only on the waveform specification. To this aim, such library should include further blocks so that it could support the specification any waveforms. In addition to this, each HLS tool fosters a specific HLL-based description. An interesting perspective should be to unify the description of the HLS FB, by using macros for instance, so that a given block could be synthesized by any of the available HLS tools. This would undoubtedly help achieving better performance since each tool has its specific approach when it comes to optimize the design. Thus, the user could easily trade-off between different solutions issued from diverse HLS tools. The dynamic reconfiguration of the final waveform has not properly been addressed in this work. However, we have featured the generated control units with a block level reconfiguration state that is intended to support the run-time reconfiguration requirements at the block-level. FPGAs dynamic and partial reconfiguration capabilities were also identified as the key enabling technologies for implementing reconfiguration scenarios in FPGA-SDRs. Standard level reconfiguration should be addressed too.

A long term perspective should address the evolution of the proposed design flow. One could argue that the flow could be further enhanced to support multi-platform SDRs or, on the contrary, to support low power applications, which are in essence two distinct visions. The former would require extending the flow to different platform architectures while the latter would foster augmenting the waveform generation process with low power design techniques by considering a unique type of platform, in our case, the FPGA-based platforms. In our opinion, both options should be considered given that SDR is intended to support any kinds of waveforms.

The Author's Publications

The work presented in this document was subject to different publications which can be listed as follows:

Journal Paper

G. S. Ouedraogo, M. Gautier and O. Sentieys, *"A Frame-Based Domain-Specific Language for Rapid Prototyping of FPGA-Based Software Defined Radios"*, in EURASIP Journal on Advances in Signal Processing, December 2014.

International Conferences

G. S. Ouedraogo, M. Gautier and O. Sentieys, *"Frame-based Modelling for Automatic Synthesis of FPGA-Software Defined Radio"*, in International Conference on Cognitive Radio Oriented Wireless Networks (CrownCom 2014), Finland, June 2014.

M. Gautier, G. S. Ouedraogo and O. Sentieys, *Design Space Exploration in an FPGA-Based Software Defined Radio*, in Euromicro Conference on Digital System Design (DSD 2014), Italy, August 2014.

V. Bhatnagar, G. S. Ouedraogo, M. Gautier, A. Carer and O. Sentieys, *An FPGA Software-Defined Radio Platform with a High-Level Synthesis Design Flow*, in IEEE International Conference on Vehicular Technology (VTC Spring 2013), Germany, June 2013.

Book Chapter

M. Gautier, E. Casseau, H. Yviquel, G. S. Ouedraogo, M. Raulet and O. Sentieys, *Rapid Prototyping for Video Coding over Flexible Radio Links*, In Multimedia over Cognitive Radio Networks: Algorithms, Protocols, and Experiments. Edited by Fei Hu and Sunil Kumar, CRC Press, December 2014.

National Conferences

G. S. Ouedraogo, M. Gautier, and O. Sentieys, *Description de haut niveau de forme d'ondes pour la radio logicielle sur des architectures reconfigurables*, Colloque GRETSI 2013, Brest, sept. 2013.

G. S. Ouedraogo, M. Gautier and O. Sentieys, *Vers un langage spécialisé pour la radio logicielle sur FPGA*, Colloque du GDR SOC-SIP 2013, Lyon, juin 2013.

Appendices

Appendix A

Résumé en français

Introduction

L'objectif de cette thèse est la définition et la mise en oeuvre d'un flot de conception de formes d'ondes radio logicielles s'exécutant sur des cibles matérielles de types *Field Programmable Gate Array (FPGA)*. En effet, la radio logicielle est une technologie qui adresse la conception de formes d'ondes hautement flexibles et capables de s'adapter à l'environnement dans lequel elles opèrent. Pour ce faire, la radio logicielle s'appuie sur des plateformes matérielles capables d'assurer les besoins en flexibilité ainsi qu'en puissance de calcul. Dans ce contexte, les FPGAs apparaissent naturellement comme un support "idéal" pour la mise en oeuvre de ces formes d'ondes tant pour leur puissance de calcul que pour leur capacités en matière de reconfiguration. Cependant, les FPGAs disposent d'un modèle de programmation qui contraste sévèrement avec l'esprit même de la radio logicielle en ce sens qu'ils requièrent l'utilisation de langages de programmation bas niveau.

Dans cette thèse, nous avons ainsi traité le problème de la conception de forme d'ondes radio logicielles sur FPGA en proposant un flot de conception innovant, partant de modèles haut niveau des formes d'ondes jusqu'à leur implantation sur la plateforme ciblée. Ce flot contourne les limites liées au modèle de programmation des FPGAs en employant la synthèse de haut niveau (*HLS pour High-Level Synthesis*). En effet, la HLS est une approche qui permet de programmer les FPGAs à partir de langages haut niveau comme le C/C++, relachant ainsi la contrainte liée à son modèle de programmation. Afin de bénéficier de cette nouvelle approche dans le contexte de la radio logicielle, nous avons defini au dessus de la HLS un langage à usage spécifique (*DSL pour Domain-Specific Language*) dans l'optique de faciliter la mise en oeuvre de forme d'ondes radio logicielles sur FPGA.

La radio logicielle

La radio logicielle a été introduite au début des années 90 par Joe Mitola qui à travers des articles pointait du doigt l'inélucatable évolution des plateformes radio vers des plateformes comprenant de plus en plus de composantes logicielles. Ainsi, il prévoyait que les approches traditionnelles de conception radio, qui consistaient essentiellement en l'implémentation d'architectures dédiées, seraient peu à peu abandonnées au profit d'une conception centrée sur des composantes logicielles.

Une plateforme radio logicielle idéale est illustrée à travers la Figure 1 dans laquelle les données sont échantillonnées directement après les antennes. Cette architecture permettrait de réaliser tout le traitement en radio fréquence (RF) et en bande de base sur un processeur ou sur un microprocesseur généraliste. Une telle architecture est qualifiée d'idéale du simple fait que la technologie actuelle ne permet pas la réalisation d'un tel scénario. Les solutions radio logicielles rencontrées à ce jour intercalent un *front-end* radio entre les convertisseurs et les composantes numériques (processeur, microprocesseurs) afin de démoduler le signal en fréquence intermédiaire ou directement en bande base.

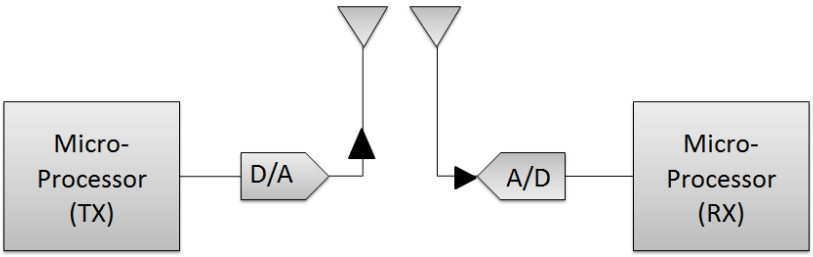

FIGURE 1 – Plateforme radio logicielle idéale.

Les travaux sur la radio logicielle ont conduit principalement à définir d'une part un ensemble de méthodologies de conception dédiées à la radio logicielle et d'autre part un certain nombre de familles de plateformes radio logicielles. Nous proposons une classification des familles de plateformes radio logicielles comme suit :

- Les plateformes se basant sur un processeur généraliste : ces plateformes réalisent le traitement en bande de base sur un ordinateur associé à un *front-end* radio pour la transmission du signal dans le canal.
- Les plateformes hybrides : elles associent au processeur généraliste des composants dédiés à certaines opérations du traitement du signal.

— Les plateformes se basant sur le FPGA : elles réalisent l'ensemble du traitement en bande de base sur des gros FPGA capables de supporter des formes d'ondes relativement complexes.

D'autre part, les recherches sur les méthodologies de conception dédiées à la radio logicielle ont conduit aux deux grandes catégories de méthodes qui sont les suivantes :

— Les **standards** qui peuvent être présentés comme un ensemble d'intergicielles faisant office d'interface entre le matériel et le logiciel.

— Les **langages** qui ont pour objectif principal de faciliter la spécification et la programmation de couches physiques radios logicielles.

La synthèse de haut niveau

La synthèse de haut niveau peut être définie comme étant le processus permettant de générer automatiquement une description RTL d'une forme d'onde ou d'une fonction à partir de sa spécification à haut niveau. Elle entend ainsi réduire le fossé existant entre les équipes chargées de spécifier les algorithmes et celles chargées de leur implémentation. Les spécifications de haut niveau sont réalisées dans des langages tel que le C/C++, le SystemC ou Matlab. La Figure 2 présente un flot de conception HLS qui commence par une spécification haut niveau du système qui est ensuite analysée afin de produire une représentation intermédiaire sous forme d'un graphe de contrôle et de flot de données (CDFG). Ce graphe peut éventuellement subir des transformations de haut niveau avant d'être utilisé pour réaliser l'allocation de ressources matérielles selon la technologie choisie par l'utilisateur. Les contraintes spécifiées par l'utilisateur sont prises en compte dans cette étape et il s'en suit la génération du code RTL décrivant le système.

La synthèse de haut niveau présente de nombreux avantages en ce sens qu'elle permet un prototypage rapide des formes d'ondes du fait de l'utilisation de langages de haut niveau. Ainsi, il a été montré que la HLS réduit le nombre de ligne de code utile à la description d'une application de près de 80%. D'autre part, la HLS permet d'explorer assez aisément l'espace des solutions possibles pour une application donnée. En effet, elle propose un ensemble d'optimisation permettant d'aboutir à différentes solutions sans avoir à rééditer le code source. Ces optimisations peuvent entre autres être faites sur des structures de boucles ou encore sur des accès mémoires.

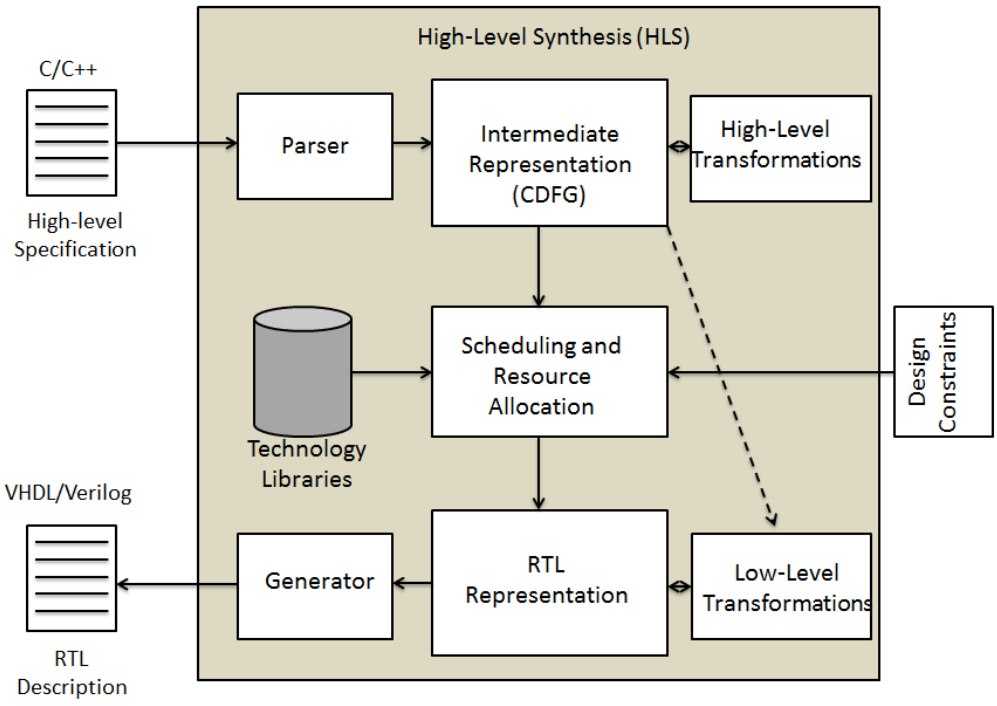

FIGURE 2 – Flot de conception HLS.

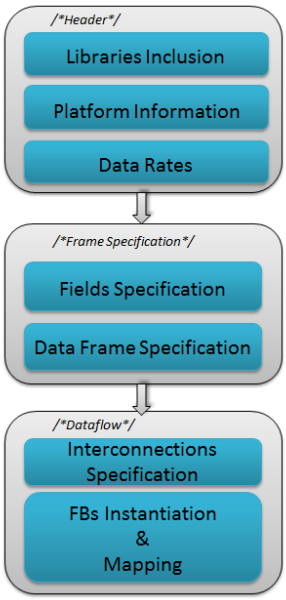

FIGURE 3 – Description d'une forme d'onde via le DLS proposé.

Principales contributions

Nos travaux ont porté sur la définition ainsi que la mise en oeuvre d'une flot de conception radio logicielle s'appuyant sur la synthèse de haut niveau, l'idée de base étant de tirer profit des innovations de la HLS dans le contexte de la radio logicielle. Pour ce faire, nous avons défini un langage à usage spécifique (DSL) permettant de modéliser les différents aspects d'une forme d'onde. En effet, le flot que nous proposons permet de spécifier un modèle de plateforme d'exécution comprenant une spécification des convertisseurs et de la cible FPGA. Ensuite, l'utilisateur spécfie dans le DSL le modèle de la trame de donnée émise (en émission) ou traitée (en réception) par la forme d'onde. Enfin, le graphe flot de données de la forme d'onde est spécfié en instanciant chaque bloc fonctionnel composant le graphe et en les interconnectant de manière cohérente. Chacun de ces blocs à été préalablement décrit et verifié par des outils HLS. Des contraintes d'architectures peuvent être indiquées pour chaque bloc dans le DSL, permettant ainsi d'explorer un ensemble de solutions pour la forme d'onde. Outre les contraintes architecturales, l'instanciation de chaque bloc requière de mentionner explicitement les parties de la trame de donnée que ce bloc sera emmené à traiter. La structure de la description d'une forme d'onde à travers le DSL est donnée dans la Figure . Elle comprend, une en-tête, un modèle de trame et un modèle du graphe flot de données.

Une vue d'ensemble du flot de conception proposé est présentée sur la Figure 4. Une fois les élements mentionés ci-dessus modélisés, ils sont par la suite analysés afin d'en déduire une représentation intermédiaire sous la forme d'un arbre abstrait (*AST pour Abstract Syntax Tree*). Cette représentation constitue le principal point d'entrée du compilateur que nous avons associé à ce flot de conception. Le dit compilateur est chargé d'extraire de la représentation les sous graphes modélisant la plateforme, la trame de données ainsi que la structure flot de données de la forme d'onde. Les interrelations entre ces différentes parties sont aussi prises en compte. Pour chacun des blocs instanciés, le compilateur génère un script de synthèse destiné à guider la synthèse de ce bloc par les outils de HLS. Aussi, l'ensemble des blocs synthétisés est assemblé de sorte à constituer la structure flot de données préalablement définie dans le DSL. Le modèle de la trame de données est utilisé par le compilateur afin de déduire l'unité de contrôle utile au fonctionnement de la forme d'onde. La description d'une trame générique est donnée à travers la Figure 5 dans laquelle une trame comprenant N champs, numérotés de 1 à N, est décrite. Cette description inclue la spécification de tous les champs composant la trame et, pour finir, la spécification de la trame comme étant une concaténation de ces champs. Parallèlement, dans le flot que nous proposons, un ensemble de vérifications permettent de s'assurer du fonctionnement de la forme d'onde finale.

La dernière étape du flot que nous proposons consiste en la programmation de la forme d'onde sur la plateforme cible. Il s'agit principalement de la génération de bitstream à travers les outils proposés par les différents acteurs du marché des FPGAs. Ces outils, bien que incontournable, rajoutent une dépendance supplémentaire à notre flot car ils ne sont accessibles qu'à travers des licences propriétaires. Dans la version

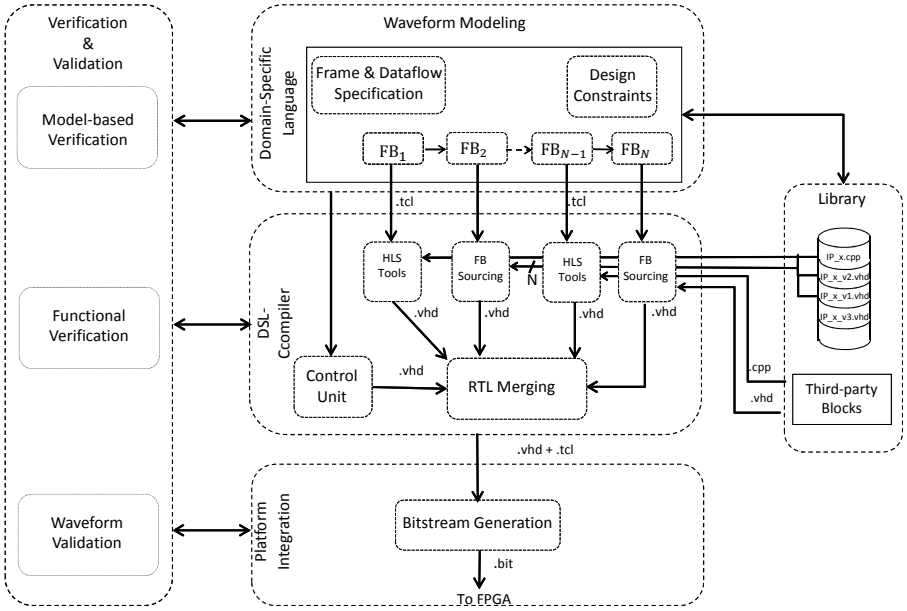

FIGURE 4 – Flot de conception proposé.

actuelle de notre flot, la programmation de la forme d'onde se fait manuellement à partir de sa description RTL et des outils de gérération de bitsream.

Expérimentation et validation

Dans un souci de validation, le flot de conception à été expérimenté à travers la modélisation et l'implémentation de deux couches physiques, à savoir la couche physique IEEE 802.15.4 et la couche physique IEEE 802.11a. Ces standards spécifient les couches physiques des technologies ZigBee et Wifi respectivement. Nous disposons à cet effet d'une plateforme matérielle nommée Nutaq Perseus 6010. Elle est équipée d'un FPGA Virtex-6 de chez Xilinx, programmable à partir du logiciel XPS. De plus, la plateforme dispose de convertisseur à fréquence de fonctionnement réglable et pouvant atteindre les 80 MHz. Ces convertisseurs ont une précision de donnée de 12-bit.

En rappel, la modélisation d'une forme d'onde comprend, un modèle de la plateforme, un modèle de la trame de donnée ainsi qu'un modèle du graphe flot de donnée associé. La plateforme étant la même pour ces deux formes d'ondes, ceci résulte donc en un unique modèle de plateforme. La trame de donnée de la technologie ZigBee comprend deux champs de synchronisation, un champ de données spécifiquse à la trame et un champ de données utiles. Le graphe flot de données associé implémente une technique d'accès au canal du type étalement de spectre, plus précisement le DSSS (*Direct Sequence Spread Spectrum*).
La trame de données de la technologie Wifi, considérée à travers le standard IEEE 802.11a, comprend aussi deux champs de synchronisation, un champ de données spécifiques à la trame et, pour finir, un champ de données utiles. Par ailleurs, le graphe flot de données associé à cette technologie implémente un technique d'accès au canal du type multiplexage spectrale, plus précisement l'OFDM (*Orthogonal Frequency Division Multiplexing*).

L'utilisation d'outils de synthèse de haut niveau permet, comme nous l'avons mentionné plus haut, d'explorer l'espace des solutions possibles pour une fonction donnée. La Figure (à gauche) présente un exemple d'exploration d'architecture sur l'un des blocs composant la couche physique IEEE 802.15.4. On peut noter sur la Figure, une variation de la surface en fonction des contraintes appliquées sur des boucles.

Ainsi, ces formes d'ondes ont été modélisées et implantées sur la plateforme Nutaq Perseus 6010. L'interprétation des résultats est faite par le biais d'équipements d'analyse temps-réel des signaux tel que des

```
/* Specification of field#1 */
#fieldC F_1 {
    constant dataf1;                /* Constant symbol */
    redundancy 8;                  /* Repetition over 8 symbols */
    duration 128 us;                /* Overall duration of the field */
}
                .
                .
                .
/* Specification of field#3 */
#fieldV F_3 {
    data dataf3;                    /* Variable data being carried by field#3 */
    duration 32 us;                /* Fixed duration of field#3 */
}
                .
                .
                .
/* Specification of field#N */
#fieldV F_N {
    data datafN;                      /* Data payload conveyed by field#N */
    maxsize 128 bytes;                /* Maximum payload size */
    minsize 16 bytes;                /* Minimum payload size */
}
/* Data Frame Specification */
complex frame F {
    F_1 F_2 F_3 ...F_N
}    sof after F_1                 /* F_1 is designated as the start of frame */
```

FIGURE 5 – Description d'une trame générique à partir du DSL.

oscilloscopes, des analyseurs de spectre ou encore des outils d'analyse de signaux en bande de base tel que Chipscope de chez Xilinx. Ceci s'illustre aussi dans la Figure (à droite), dans laquelle on peut voir le signal ZigBee démodulé avec succès par les équipement VSA. Cette expérience permet de valider la conformité de l'architecture proposée avec le standard.

Conclusions et perspectives

La radio logicielle est une technologie remarquable qui nécessite davantage d'efforts de recherche, notamment en ce qui concerne les outils de conception. À travers cette thèse, nous avons étudié ce besoin en proposant un flot de conception partant de modèles haut niveau de formes d'ondes radio logicielles jusqu'à leur implantation sur des plateformes FPGA. Nous avons ensuite expérimenté ce flot, en modélisant les couches physiques de deux standards que sont le standard IEEE 802.15.4 et le standard IEEE 802.11a. Ces standards spécfient les couches physiques des technologies ZigBee et Wifi respectivement.
Les résultats qui en découlent ont été analysés grâce à des équipements d'analyse temps-réel de signaux, permettant de valider la conformité des transmetteurs ainsi implémentés avec les différentes normes.

Diverses perspectives se dégagent de ce travail de recherche à savoir l'éventualitè d'étendre le flot proposé afin qu'il puisse supporter davantage de plateformes telles que les processeurs et les microcprocesseurs généralistes ou encore les processeurs de traitement du signal. Aussi, il serait intéressant d'étendre le modèle de trame de donnée de sorte à supporter des structures de trames plus complexes telles que celles émisent par les stations de bases, pouvant transporter des données destinées à plusieurs utilisateurs à la fois.

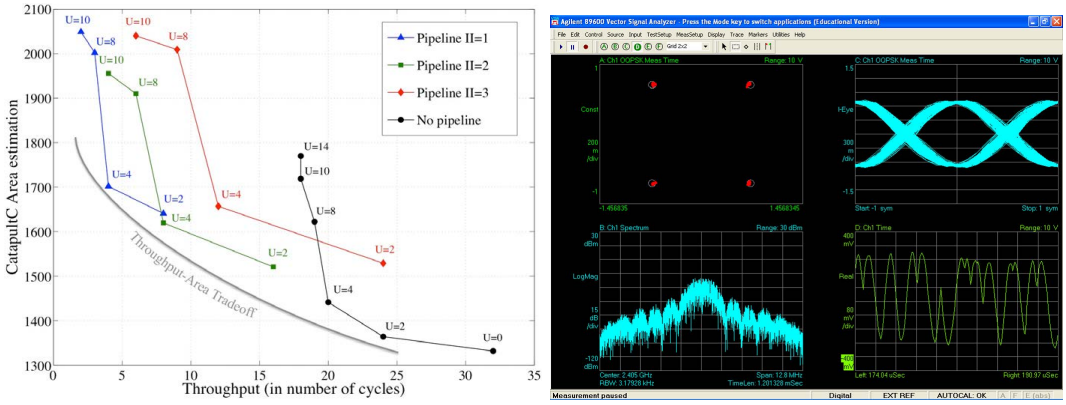

FIGURE 6 – Exploration d'architecture (à gauche) et démodulation ZigBee par les équipements VSA.

Appendix B

HLS Specifications

This chapter is intended to describe the C/C++ source code, which specifies the PHYs that we have considered for this work. The code is compliant with the Catapult synthesis tool and it comprises both the transmitter and the receiver specifications. The IEEE 802.15.4 PHY was entirely specified by the authors whereas the IEEE 802.11a PHY has consisted of some specifications that were provided by the tool or written by some fellow colleagues.

B.1 IEEE 802.15.4 PHY HLS specification

The IEEE 802.15.4 PHY HLS specification comprises three main files, namely the *pack.h* file, the *tx.cpp* and the *rx.cpp* files. The former file is a header file where different macros and functions prototypes are defined. Moreover, some specific data types of an arbitrary precision are specified within the header file. The prototypes consists of *void* functions including some arguments of pointers and arrays type. The arguments represent the input and output connections of the functions. The latter two files specify the transmitter and the receiver of the ZigBee transceiver.

B.1.1 Transmitter

The IEEE 802.15.4 PHY specifies three main blocks, namely the modulation block, the spreading block and the shaping filter block. The former block coverts group of block into 16 different symbols. The second block spreads each symbol into a sequence of 32 chips. The sequence of chip is later split up into two channels I and Q. The latter block shapes the incoming chips (on both channel I and Q) with the half sine filter. Finally, a delay block is inserted on channel Q to enable continuous phase change.

B.1.2 Receiver

The receiver specification includes a delay block, which compensates the delay between channel I and channel Q. Following this stage, a matched filter reshapes the incoming stream by maximizing the SNR. The reshaped samples are then fed to the synchronization block, which computes the sampling period, the phase and the frequency offset. Once these elements are computed, the synchronization block switches off and filter's output samples are fed to the compensation block which downsamples and compensates the phase and the frequency error on the samples. Afterwards, the compensated samples are correlated with the known chips sequences through a sliding correlation bench which enables to recover the transmitted symbols. Each symbol is converted into bits by a de-modulator.

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                       IEEE 802.15.4 PHY LAYER
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%
TRANSMITTER
%%%%%%%%%%%%

    ************      *********** chips ********          samples
    *          *      *         *------>*      *---------------->I
bit *          *symbol*         * 1MHz  *      *           OUTPUTFREQ MHz
--->*modulation*------>*chipgen  *       * txfir*
    *          *      *         * chips *      *    *******
    *          *      *         *------>*      *----*delay*----->Q
    ************      ***********       ********    *******samples
 OUTPUTFREQ MHz


%%%%%%%%%%%
RECEIVER
%%%%%%%%%%%


    *******    ********      ************      *************
--->*delay*--->*      *----->*          *      *           *
    *******    *      *   |  *          *      *           *
               *rxfir *   |  *decimation*----->* corrbench *---->symbol
               *      *   |  *          *      *           *
-------------->*      *---|->*          *      *           *
               ********  ||  ******^*****      *************
                         ||        |
                         ||        |
                         ||   ***********
                         | -->*         *
                         |    * synchro *
                         ---->*         *
                              ***********


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
#ifndef __PACK_ZIGBEE__
#define __PACK_ZIGBEE__
#include <stdio.h>
#include <ac_fixed.h>
#include <ac_int.h>
#include <ac_channel.h>
#include <math.h>
#include "shift_class.h"

using namespace std;

#define  NBCHIP        32
#define  IQNBCHIP      NBCHIP/2
#define  OUTPUTFREQ    10
#define  N             IQNBCHIP*OUTPUTFREQ
#define  d             OUTPUTFREQ/2
#define  DATAWIDTHRX   5
#define  DATAWIDTHTX   5
#define  RXFIRDATAWD   2*DATAWIDTHRX + 4
#define  CORRDATA      RXFIRDATAWD + 5
#define  GARDNER       1
#define  CFO           4
#define  GARDNERDEL    GARDNER*(NBCHIP/2)*OUTPUTFREQ
#define  CFODEL        CFO*(NBCHIP/2)*OUTPUTFREQ
#define  NBPHASE       8
#define  PBDETECT      3
#define  PBDETECTDEL   PBDETECT*(NBCHIP/2)*OUTPUTFREQ
#define  PBTRH         15000

typedef ac_int<1,false>             bit;
```

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                       IEEE 802.15.4 PHY LAYER
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%
TRANSMITTER
%%%%%%%%%%%%

    ************      *********** chips ********          samples
    *          *      *         *------>*      *----------------->I
bit *          *symbol*         * 1MHz  *      *           OUTPUTFREQ MHz
--->*modulation*----->*chipgen  *       * txfir*
    *          *      *         * chips *      *    *******
    *          *      *         *------>*      *----*delay*----->Q
    ************      ***********       ********    *******samples
 OUTPUTFREQ MHz


%%%%%%%%%%%
RECEIVER
%%%%%%%%%%%


    *******    ********      ************      *************
--->*delay*--->*      *----->*          *      *           *
    *******    *      *   |  *          *      *           *
               *rxfir *   |  *decimation*----->* corrbench *---->symbol
               *      *   |  *          *      *           *
-------------->*      *---|->*          *      *           *
               ********  ||  ******^*****      *************
                         ||      |
                         ||      |
                         ||   ***********
                         | -->*         *
                         |    * synchro *
                         ---->*         *
                              ***********


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
#ifndef __PACK_ZIGBEE__
#define __PACK_ZIGBEE__
#include <stdio.h>
#include <ac_fixed.h>
#include <ac_int.h>
#include <ac_channel.h>
#include <math.h>
#include "shift_class.h"

using namespace std;

#define  NBCHIP        32
#define  IQNBCHIP      NBCHIP/2
#define  OUTPUTFREQ    10
#define  N             IQNBCHIP*OUTPUTFREQ
#define  d             OUTPUTFREQ/2
#define  DATAWIDTHRX   5
#define  DATAWIDTHTX   5
#define  RXFIRDATAWD   2*DATAWIDTHRX + 4
#define  CORRDATA      RXFIRDATAWD + 5
#define  GARDNER       1
#define  CFO           4
#define  GARDNERDEL    GARDNER*(NBCHIP/2)*OUTPUTFREQ
#define  CFODEL        CFO*(NBCHIP/2)*OUTPUTFREQ
#define  NBPHASE       8
#define  PBDETECT      3
#define  PBDETECTDEL   PBDETECT*(NBCHIP/2)*OUTPUTFREQ
#define  PBTRH         15000

typedef ac_int<1,false>             bit;
```

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                       IEEE 802.15.4 PHY LAYER
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%
TRANSMITTER
%%%%%%%%%%%%

    ************      *********** chips ********          samples
    *          *      *         *------->*      *----------------->I
bit *          *symbol*         * 1MHz  *      *           OUTPUTFREQ MHz
--->*modulation*----->*chipgen  *       * txfir*
    *          *      *         * chips *      *    *******
    *          *      *         *------->*      *----*delay*----->Q
    ************      ***********       ********    *******samples
 OUTPUTFREQ MHz


%%%%%%%%%%%
RECEIVER
%%%%%%%%%%%


    *******    ********      ************      *************
--->*delay*--->*      *----->*          *      *           *
    *******    *      *   |  *          *      *           *
               *rxfir *   |  *decimation*----->* corrbench *---->symbol
               *      *   |  *          *      *           *
-------------->*      *---|->*          *      *           *
               ********  ||  *****^*****      *************
                         ||         |
                         ||         |
                         ||   ***********
                         | -->*         *
                         |    * synchro *
                         ---->*         *
                              ***********


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
#ifndef __PACK_ZIGBEE__
#define __PACK_ZIGBEE__
#include <stdio.h>
#include <ac_fixed.h>
#include <ac_int.h>
#include <ac_channel.h>
#include <math.h>
#include "shift_class.h"

using namespace std;

#define  NBCHIP        32
#define  IQNBCHIP      NBCHIP/2
#define  OUTPUTFREQ    10
#define  N             IQNBCHIP*OUTPUTFREQ
#define  d             OUTPUTFREQ/2
#define  DATAWIDTHRX   5
#define  DATAWIDTHTX   5
#define  RXFIRDATAWD   2*DATAWIDTHRX + 4
#define  CORRDATA      RXFIRDATAWD + 5
#define  GARDNER       1
#define  CFO           4
#define  GARDNERDEL    GARDNER*(NBCHIP/2)*OUTPUTFREQ
#define  CFODEL        CFO*(NBCHIP/2)*OUTPUTFREQ
#define  NBPHASE       8
#define  PBDETECT      3
#define  PBDETECTDEL   PBDETECT*(NBCHIP/2)*OUTPUTFREQ
#define  PBTRH         15000

typedef ac_int<1,false>             bit;
```

```
/**********************
  CATAPULT C PROJECT
**********************/
#include "pack.h"

/*******************************
Transmitter specification
*******************************/

/*Modulation converts bits into symbols*/

using namespace std;

void modulation(bit src[4], symbol *sb)
{
     symbol tmp= (symbol)src[0];
     for(int i= 1; i < 4; i++){
         if (src[i]== 1){
         tmp += 2<<(i-1);
         }
      }
      *sb = tmp;
}
/* ChipGen generates 32 chips corresponding */
/* to each symbol and split up into two channels*/

void chipgen(symbol *sb, bit ich0[IQNBCHIP], bit qch0[IQNBCHIP])
{
  symbol sbtmp = 0;
  sbtmp = *sb;
  bit chips[16][NBCHIP]=
 {{1,1,0,1,1,0,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0},
  {1,1,1,0,1,1,0,1,1,0,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0},
  {0,0,1,0,1,1,1,0,1,1,0,1,1,0,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0},
  {0,0,1,0,0,0,1,0,1,1,1,0,1,1,0,1,1,1,0,0,1,1,1,1,0,0,0,0,1,1,0,1},
  {0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,1,0,1,1,0,0,1,1,1,1,0,0,0,0,1,1,1},
  {0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0,1,1,0,1,1,0,0,1,1,1,0,0},
  {1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0,1,1,0,1,1,0,0,1},
  {1,0,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0,1,1,0,1},
  {1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,0,1,1,1,0,1,1,1,1,0,1},
  {1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,0,1,1,1,0,1,1},
  {0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,1,1,1},
  {0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0},
  {0,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0},
  {0,1,1,0,0,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1},
  {1,0,0,1,0,1,1,0,0,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0},
  {1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0}};


  for(int i= 0; i< NBCHIP; i++){
        if(i%2==0){
         ich0[i>>1]= chips[sbtmp][i];
        }
        else {
         qch0[(i-1)>>1]= chips[sbtmp][i];
        }
   }

}
/*Tx_fir shapes each chip prior to transmission*/
void txfir(bit ich0[IQNBCHIP], bit qch0[IQNBCHIP], sample_tx ich1[N], sample_tx qch1[N])
{
  const txfircoeff txcoeff[OUTPUTFREQ]= {0, 5, 9, 12, 14, 15, 14, 12, 9, 5};
  for(int i= 0; i< IQNBCHIP; i++){
      bit temp_i = ich0[i];
      bit temp_q = qch0[i];
      for(int j= OUTPUTFREQ*i; j< OUTPUTFREQ*(i+1); j++){
          ich1[j]= (2*temp_i-1)*txcoeff[j-OUTPUTFREQ*i];
          qch1[j]= (2*temp_q-1)*txcoeff[j-OUTPUTFREQ*i];
```

```
/**********************
  CATAPULT C PROJECT
**********************/
#include "pack.h"

/*******************************
Transmitter specification
*******************************/

/*Modulation converts bits into symbols*/

using namespace std;

void modulation(bit src[4], symbol *sb)
{
     symbol tmp= (symbol)src[0];
     for(int i= 1; i < 4; i++){
         if (src[i]== 1){
         tmp += 2<<(i-1);
         }
      }
      *sb = tmp;
}
/* ChipGen generates 32 chips corresponding */
/* to each symbol and split up into two channels*/

void chipgen(symbol *sb, bit ich0[IQNBCHIP], bit qch0[IQNBCHIP])
{
  symbol sbtmp = 0;
  sbtmp = *sb;
  bit chips[16][NBCHIP]=
 {{1,1,0,1,1,0,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0},
  {1,1,1,0,1,1,0,1,1,0,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0},
  {0,0,1,0,1,1,1,0,1,1,0,1,1,0,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0},
  {0,0,1,0,0,0,1,0,1,1,1,0,1,1,0,1,1,1,0,0,1,1,1,1,0,0,0,0,1,1,0,1},
  {0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,1,0,1,1,0,0,1,1,1,1,0,0,0,0,1,1,1},
  {0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0,1,1,0,1,1,0,0,1,1,1,0,0},
  {1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0,1,1,0,1,1,0,0,1},
  {1,0,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,1,1,1,0,1,1,0,1},
  {1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,0,1,1,1,0,1,1,1,1,0,1},
  {1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,0,1,1,1,0,1,1},
  {0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,1,1,1},
  {0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0},
  {0,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1,0,1,1,0},
  {0,1,1,0,0,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,1},
  {1,0,0,1,0,1,1,0,0,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0},
  {1,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,0,0}};


  for(int i= 0; i< NBCHIP; i++){
        if(i%2==0){
         ich0[i>>1]= chips[sbtmp][i];
        }
        else {
         qch0[(i-1)>>1]= chips[sbtmp][i];
        }
   }

}
/*Tx_fir shapes each chip prior to transmission*/
void txfir(bit ich0[IQNBCHIP], bit qch0[IQNBCHIP], sample_tx ich1[N], sample_tx qch1[N])
{
  const txfircoeff txcoeff[OUTPUTFREQ]= {0, 5, 9, 12, 14, 15, 14, 12, 9, 5};
  for(int i= 0; i< IQNBCHIP; i++){
      bit temp_i = ich0[i];
      bit temp_q = qch0[i];
      for(int j= OUTPUTFREQ*i; j< OUTPUTFREQ*(i+1); j++){
          ich1[j]= (2*temp_i-1)*txcoeff[j-OUTPUTFREQ*i];
          qch1[j]= (2*temp_q-1)*txcoeff[j-OUTPUTFREQ*i];
```

```
1  #include "pack.h"
2
3  /*****************************
4  Receiver specification
5  *****************************/
6
7  using namespace std;
8
9  /*Matched filter specification*/
10 void rxfir(sample_rx *ich2, sample_rx *qch2, data_fir *ich3, data_fir *qch3)
11 {
12
13    static shift_class<data_fir, OUTPUTFREQ> i_reg, q_reg;
14    rxfircoeff rxcoeff[OUTPUTFREQ]= {0, 5, 9, 12, 14, 15, 14, 12, 9, 5};
15    i_reg << *ich2;
16    q_reg << *qch2;
17
18    data_fir temp_i = 0;
19    data_fir temp_q = 0;
20    MAC: for(int i= OUTPUTFREQ-1; i>= 0; i--){
21            temp_i += i_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
22            temp_q += q_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
23         }
24
25    *ich3= temp_i;
26    *qch3= temp_q;
27 }
28
29 /*Delay compensation on channel I*/
30 void delayrx(data_fir *input, data_fir *output)
31 {
32      static shift_class<data_fir, d+1> reg;
33      static bool delay = false;
34      if(!delay){
35         reg << *input;
36         for(int i= 1; i<d+1; i++){
37             reg[i] = 0;
38         }
39         *output = reg[d];
40         delay = true;
41      }
42      else{
43         reg << *input;
44         *output = reg[d];
45      }
46 }
47
48 syncwd minindex(int array[OUTPUTFREQ]){
49 int min = array[0];
50 syncwd idxtp = 0;
51
52 for(int i= 1; i < OUTPUTFREQ; i++){
53    if(array[i] < min){
54    min = array[i];
55    idxtp = i;
56    }
57 }
58   return idxtp;
59 }
60
61 syncwd maxindex(corr array[NBPHASE]){
62     corr max = array[0];
63     syncwd idxtp= 0;
64     for(int i= 1; i < NBPHASE; i++){
65        if(array[i] >= max){
66        max = array[i];
67        idxtp = i;
68        }
69     }
```

```
1  #include "pack.h"
2
3  /*****************************
4  Receiver specification
5  *****************************/
6
7  using namespace std;
8
9  /*Matched filter specification*/
10 void rxfir(sample_rx *ich2, sample_rx *qch2, data_fir *ich3, data_fir *qch3)
11 {
12
13    static shift_class<data_fir, OUTPUTFREQ> i_reg, q_reg;
14    rxfircoeff rxcoeff[OUTPUTFREQ]= {0, 5, 9, 12, 14, 15, 14, 12, 9, 5};
15    i_reg << *ich2;
16    q_reg << *qch2;
17
18    data_fir temp_i = 0;
19    data_fir temp_q = 0;
20    MAC: for(int i= OUTPUTFREQ-1; i>= 0; i--){
21            temp_i += i_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
22            temp_q += q_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
23         }
24
25    *ich3= temp_i;
26    *qch3= temp_q;
27 }
28
29 /*Delay compensation on channel I*/
30 void delayrx(data_fir *input, data_fir *output)
31 {
32      static shift_class<data_fir, d+1> reg;
33      static bool delay = false;
34      if(!delay){
35         reg << *input;
36         for(int i= 1; i<d+1; i++){
37             reg[i] = 0;
38         }
39         *output = reg[d];
40         delay = true;
41      }
42      else{
43         reg << *input;
44         *output = reg[d];
45      }
46 }
47
48 syncwd minindex(int array[OUTPUTFREQ]){
49 int min = array[0];
50 syncwd idxtp = 0;
51
52 for(int i= 1; i < OUTPUTFREQ; i++){
53    if(array[i] < min){
54    min = array[i];
55    idxtp = i;
56    }
57 }
58   return idxtp;
59 }
60
61 syncwd maxindex(corr array[NBPHASE]){
62     corr max = array[0];
63     syncwd idxtp= 0;
64     for(int i= 1; i < NBPHASE; i++){
65        if(array[i] >= max){
66        max = array[i];
67        idxtp = i;
68        }
69     }
```

```
1  #include "pack.h"
2
3  /*****************************
4  Receiver specification
5  *****************************/
6
7  using namespace std;
8
9  /*Matched filter specification*/
10 void rxfir(sample_rx *ich2, sample_rx *qch2, data_fir *ich3, data_fir *qch3)
11 {
12
13    static shift_class<data_fir, OUTPUTFREQ> i_reg, q_reg;
14    rxfircoeff rxcoeff[OUTPUTFREQ]= {0, 5, 9, 12, 14, 15, 14, 12, 9, 5};
15    i_reg << *ich2;
16    q_reg << *qch2;
17
18    data_fir temp_i = 0;
19    data_fir temp_q = 0;
20    MAC: for(int i= OUTPUTFREQ-1; i>= 0; i--){
21            temp_i += i_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
22            temp_q += q_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
23         }
24
25    *ich3= temp_i;
26    *qch3= temp_q;
27 }
28
29 /*Delay compensation on channel I*/
30 void delayrx(data_fir *input, data_fir *output)
31 {
32      static shift_class<data_fir, d+1> reg;
33      static bool delay = false;
34      if(!delay){
35         reg << *input;
36         for(int i= 1; i<d+1; i++){
37             reg[i] = 0;
38         }
39         *output = reg[d];
40         delay = true;
41      }
42      else{
43         reg << *input;
44         *output = reg[d];
45      }
46 }
47
48 syncwd minindex(int array[OUTPUTFREQ]){
49 int min = array[0];
50 syncwd idxtp = 0;
51
52 for(int i= 1; i < OUTPUTFREQ; i++){
53    if(array[i] < min){
54    min = array[i];
55    idxtp = i;
56    }
57 }
58   return idxtp;
59 }
60
61 syncwd maxindex(corr array[NBPHASE]){
62     corr max = array[0];
63     syncwd idxtp= 0;
64     for(int i= 1; i < NBPHASE; i++){
65        if(array[i] >= max){
66        max = array[i];
67        idxtp = i;
68        }
69     }
```

```
1  #include "pack.h"
2
3  /*****************************
4  Receiver specification
5  *****************************/
6
7  using namespace std;
8
9  /*Matched filter specification*/
10 void rxfir(sample_rx *ich2, sample_rx *qch2, data_fir *ich3, data_fir *qch3)
11 {
12
13    static shift_class<data_fir, OUTPUTFREQ> i_reg, q_reg;
14    rxfircoeff rxcoeff[OUTPUTFREQ]= {0, 5, 9, 12, 14, 15, 14, 12, 9, 5};
15    i_reg << *ich2;
16    q_reg << *qch2;
17
18    data_fir temp_i = 0;
19    data_fir temp_q = 0;
20    MAC: for(int i= OUTPUTFREQ-1; i>= 0; i--){
21            temp_i += i_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
22            temp_q += q_reg[i]*rxcoeff[OUTPUTFREQ-1-i];
23         }
24
25    *ich3= temp_i;
26    *qch3= temp_q;
27 }
28
29 /*Delay compensation on channel I*/
30 void delayrx(data_fir *input, data_fir *output)
31 {
32      static shift_class<data_fir, d+1> reg;
33      static bool delay = false;
34      if(!delay){
35         reg << *input;
36         for(int i= 1; i<d+1; i++){
37             reg[i] = 0;
38         }
39         *output = reg[d];
40         delay = true;
41      }
42      else{
43         reg << *input;
44         *output = reg[d];
45      }
46 }
47
48 syncwd minindex(int array[OUTPUTFREQ]){
49 int min = array[0];
50 syncwd idxtp = 0;
51
52 for(int i= 1; i < OUTPUTFREQ; i++){
53    if(array[i] < min){
54    min = array[i];
55    idxtp = i;
56    }
57 }
58   return idxtp;
59 }
60
61 syncwd maxindex(corr array[NBPHASE]){
62     corr max = array[0];
63     syncwd idxtp= 0;
64     for(int i= 1; i < NBPHASE; i++){
65        if(array[i] >= max){
66        max = array[i];
67        idxtp = i;
68        }
69     }
```

B.2 IEEE 802.11a PHY HLS specifications

As mentioned previously, regarding the IEEE 802.11a PHY, we have leveraged some OFDM blocks which are provided by the Catapult tool as well as some blocks which were specified by some fellow colleagues. Since this material does not belong to the author, they chose not to publish the underlying HLS specifications for some obvious copyrights reasons. However, for further information we refer the reader to the Catapult documentation which provides the necessary documentation.

Bibliography

[1] H. Kopetz, *Internet of Things.* Real-Time Systems Series, Springer US, 2011.

[2] R. H. Weber and R. Weber, *Internet of Things.* Springer Berlin Heidelberg, 2010.

[3] F. K. Jondral, "Software-defined radio: Basics and evolution to cognitive radio," *EURASIP J. Wirel. Commun. Netw.*, vol. 2005, pp. 275–283, Auugust 2005.

[4] J. Mitola III and G. Maguire Jr., "Cognitive radio: making software radios more personal," *IEEE Personal Communications*, vol. 6, pp. 13–18, Aug 1999.

[5] J. Palicot, *De la radio logicielle à la radio intelligente.* Hermes Science Publications, 2010.

[6] A. Nosratinia, T. Hunter, and A. Hedayat, "Cooperative communication in wireless networks," *Communications Magazine, IEEE*, vol. 42, pp. 74–80, Oct 2004.

[7] L. Xiao, T. Fuja, J. Kliewer, and C. D, "A network coding approach to cooperative diversity," *IEEE Transactions on Information Theory*, vol. 53, p. 3714âĂŞ3722, Oct 2007.

[8] S. Jayaweera, "Virtual mimo-based cooperative communication for energy-constrained wireless sensor networks," *Wireless Communications, IEEE Transactions on*, vol. 5, pp. 984–989, May 2006.

[9] C. Han, T. Harrold, S. Armour, I. Krikidis, S. Videv, P. M. Grant, H. Haas, J. Thompson, I. Ku, C.-X. Wang, T. A. Le, M. Nakhai, J. Zhang, and L. Hanzo, "Green radio: radio techniques to enable energy-efficient wireless networks," *Communications Magazine, IEEE*, vol. 49, pp. 46–54, June 2011.

[10] J. I. Mitola, "Software radios: Survey, critical evaluation and future directions," *IEEE Aerospace and Electronic Systems Magazine*, vol. 8, pp. 25–36, April 1993.

[11] J. Mitola, "The software radio architecture," *Communications Magazine, IEEE*, vol. 33, pp. 26–38, May 1995.

[12] G. Kalivas, *Digital Radio System Design.* A John Wisley and Sons, Ltd, Publication, 2009.

[13] J. G. Proakis, *Digital Communications.* McGraw Hill Higher Education, 2000.

[14] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing.* Prentice-Hall, Inc, 1989.

[15] M. Frerking, *Digital Signal Processing in Communications Systems.* Springer, 1994.

[16] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-based Implementation of Signal Processing Systems.* A John Wisley and Sons, Ltd, Publication, 2008.

[17] C. E. Shannon, "A mathematical theory of communication," *Reprinted with corrections from the Bell System Technical Journal*, pp. 379–423,623–656, October 1948.

[18] H. Nyquist, "Certain factors affecting the telegraph speed," *Bell System Technical Journal*, pp. 324–346, 1924.

[19] H. Nyquist, "Certain topics in telegraph mission theory," *Reprint as classic paper in: Proc. IEEE 90(2)*, February 2002.

[20] "Ieee standard for information technology: Wireless medium acess control (mac) and physical layer (phy) specifications for low-rate wireless personal area networks (wpans)," *IEEE Std 802.15.4*, 2006.

[21] "Ieee standard for local and metropolitan area networks: Part 16: Air interface for fixed broadband wireless acess systems," *IEEE Computer Society*, 1999.

[22] "Supplement to ieee standard for information technology: Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Computer Society and the IEEE Microwave Theory and Techniques Society*, 2001.

[23] "Etsi: Radio broadcasting systems; digital audio broadcasting (dab) to mobile, portable and fixed receivers," *DAB Standard by EBU*, June 2006.

[24] "Etsi: Digital video broadcasting (dvb); second generation framing, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broadband satelite application (dvb-s2)," *DVB Standard by EBU-UER*, August 2009.

[25] "Agilent technologies: Understanding gsm/edge transmitter and receiver measurements for base transceiver stations and their components," *Application Note 1312*, August 2002.

[26] J. Zyren and W. McCoy, "Overview of the 3 gpp long term evolution physical layer," *White Paper, freescale*, July 2007.

[27] M. Franceschini, G. Ferrari, and R. Raheli, *LDPC Coded Modulations*. Springer, 2009.

[28] C. Berrou and A. Glavieux, *Turbo Codes*. John Wiley & Sons, Inc., 2003.

[29] S. Heath, *DSP processor fundamentals: architectures and features*. Newnes, 1995.

[30] P. Lapsley, *Microprocessor Architectures, Second Edition: RISC, CISC and DSP*. IEEE Press, 1997, 2010.

[31] L. W. Fook, *VLIW Microprocessor Hardware Design: On ASIC and FPGA*. McGraw-Hill Professional, 2007.

[32] Z. Xuping and P. Jianguo, "Energy-detection based spectrum sensing for cognitive radio," *Wireless, Mobile and Sensor Networks, 2007. (CCWMSN07). IET Conference on*, pp. 944–947, December 2007.

[33] V. Turunen, M. Kosunen, A. Huttunen, S. Kallioinen, P. Ikonen, A. Parssinen, and J. Ryynanen, "Implementation of cyclostationary feature detector for cognitive radios," *EURASIP Journal on Wireless Communications and Networking*, pp. 1–4, June 2009.

[34] D. Noguet, L. Biard, and M. Laugeois, "Cyclostationarity detectors for cognitive radio: Architectural tradeoffs," *Cognitive Radio Oriented Wireless Networks and Communications, 2009. CROWNCOM '09. 4th International Conference on*, August 2010.

[35] M. Gautier, M. Laugeois, and P. Hostiou, "Cyclostationarity detection of dvb-t signal: testbed and measurement," *In International Conference on Advances in Cognitive Radio (Cocora 2011)*, April 2011.

[36] C. Kuo and J. Wong, "Multi-standard dsp based wireless system," *Signal Processing Proceedings, 1998. ICSP '98. 1998 Fourth International Conference on*, vol. 2, pp. 1712–1728, 1998.

[37] S. Gul, C. Moy, and J. Palicot, "Two scenarios of flexible multi-standard architecture designs using a multi-granularity exploration," *Personal, Indoor and Mobile Radio Communications, 2007. PIMRC 2007. IEEE 18th International Symposium on*, pp. 1–5, September 2007.

[38] E. Grayver, *Implementing Software Defined Radio*. Springer, 2013.

[39] R. Walden, "Analog-to-digital converter survey and analysis," *Selected Areas in Communications, IEEE Journal on*, vol. 17, pp. 539–550, April 1999.

[40] M. Dardaillon, K. Marquet, T. Risset, and A. Scherrer, "Software defined radio architecture survey for cognitive testbeds," *IEEE International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 189–194, August 2012.

[41] O. Anjum, T. Ahonen, F. Garzia, J. Nurmi, C. Brunelli, and H. Berg, "State of the art baseband dsp platforms for software defined radio: A survey," *EURASIP Journal on Wireless Communications and Networking*, June 2011.

[42] http://www.ettus.com.

[43] M. et al., "Kuar: A flexible software-defined radio development platform," *In 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Acess Networks*, pp. 428–439, April 2007.

[44] B. D. S. B Bougard and D. Verkest, "A coarse-grained array accelerator for software-defined radio," *IEEE Micro*, pp. 41–50, 2008.

[45] K. van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss, "Vector processing as an enabler for software-defined radio in handheld devices," *EURASIP J. Appl. Signal Process.*, vol. 2005, January 2005.

[46] T. Limberg, M. Winter, M. Bimberg, R. Klemm, E. Matus, M. Tavares, G. Fettweis, H. Ahlendorf, and P. Robelly, "A fully programmable 40 gops sdr single chip baseband for lte/wimax terminals," *In Solid-State Circuits Conference, ESSCIRC. 34th European*, pp. 466–469, September 2008.

[47] "Embb, a generic hardware and software architecture for digital signal processing." http://embb.telecom-paristech.fr/.

[48] F. Clermidy, R. Lemaire, X. Popon, D. Ktenas, and Y. Thonnart, "An open and reconfigurable platform for 4g telecommunication: Concepts and application.," *In Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pp. 449–456, August 2009.

[49] D. Nussbaum, K. Kalfallah, C. Moy, A. Nafkha, P. Lerary, J. Delorme, J. Palicot, J. Martin, F. Clermidy, B. Mercier, and R. Pacalet, "Open platform for prototyping of advanced software defined radio and cognitive radio techniques.," *In 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pp. 435–440, August 2009.

[50] http://warp.rice.edu.

[51] http://www.nutaq.com.

[52] G. Jianxin, Y. Xiaohui, G. Jun, and L. Quan, "The software communication architecture specification: Evolution and trends," *IEEE Conference on Computational Intelligence and Industrial Applications (PACIIA 2009)*, November 2009.

[53] R. Reinhart *et al.*, "Space telecommunications radio system (strs) architecture standard.," *NASA glenn research center, Clevelend, TM 2010-216809*, 2010.

[54] E. D and Willink, "The waveform description language: Moving from implementation to specification," *IEEE Military Communications Conference (MILCOM 2001)*, pp. 208–212, 2004.

[55] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner, "Spex: A programming language for software defined radio," *In Software Defined Radio Technical Conference and Product Exposition (SDR-Forum 06)*, November 2006.

[56] J. Gonzalez-Pina, R. Ameur-Boulifa, and R. Pacalet, "Diplodocusdf, a domain-specific modelling language for software defined radio applications," pp. 1–8, Sept 2012.

[57] "Prismtech spectra sdr: Spectra cf high performance low footprint sca core framework," *PrismTech Corporation, PrismTech Limited and PrismTech (France) SARL.*

[58] A. Gelonch, X. RevÃĺs, V. Marojevik, and R. Ferrús, "P-hal: a middleware for sdr applications," *SDR Forum Technical Conference*, November 2005.

[59] E. Grayver, H. S. Gree, and J. L. Roberson, "Sdrphy - xml description for sdr physical layer.," *The 2010 Military Communications Conference - Unclassified Program - Systems Perspectives Track*, 2010.

[60] "Gnu radio, the free and open software radio ecosystem." www.gnuradio.org.

[61] "Ieee, advancing technology for humanity." http://www.ieee.org/index.html.

[62] "Etsi, world class standards." http://www.etsi.org/.

[63] T. Cooklev, *Book Chapter: Standards for Wireless Personal Area Networking (WPAN).* Wiley-IEEE Standards Association, 2004.

[64] "Zigbee alliance." http://www.zigbee.org/.

[65] F. L. LEWIS, *Wireless Sensor Networks.* in Smart Environments: Technologies, Protocols, and Applications ed. D.J. Cook and S.K. Das, John Wiley, 2004.

[66] C. S. Raghavendra, K. M. Sivalingam, and T. Znati, *Wireless Sensor Networks.* Springer-Verlag New York Inc, 2004.

[67] U. Pesovic, D. Gliech, P. Planinsiz, Z. Stamenkovic, and S. Randic, "Implementation of ieee 802.15.4 transceiver on software defined radio platform.," *Telecommunications Forum (TELFOR), 2012 20th*, pp. 376–379, November 2012.

[68] S. Knauth, "Implementation of an ieee 802.15. 4 transceiver with a software-defined radio setup.," *Lucerne University of Applied Sciences*, 2008.

[69] J. Sabater, J. Gomez, and M. Lopez, "Towards an ieee 802.15.4 sdr transceiver.," *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pp. 323–326, December 2010.

[70] L. Choong, "Multi-channel ieee 802.15.4 packet capture using software defined radio.," *UCLA Networked & Embedded Sensing Lab*, 2009.

[71] Xilinx, "Virtex-5 fpga user guide," *UG190 (v5.4)*, March 2012.

[72] C. L. W. Kiat, "Software defined radio design for an ieee 802.11a transceiver using open source software communications architecture (sca) implementation::embedded (ossie).," *Master Thesis*, December 2006.

[73] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, "Towards an open source ieee 802.11p stack: A full sdr-based transceiver in gnu radio.," *Vehicular Networking Conference (VNC), 2013 IEEE*, pp. 143–149, December 2013.

[74] A. Tran, D. Truong, and B. Baas, "A complete real-time 802.11a baseband receiver implemented on an array of programmable processors.," *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, pp. 165–170, October 2008.

[75] P. Coulton and D. Carline, "An sdr inspired design for the fpga implementation of 802.11a baseband system.," *Consumer Electronics, 2004 IEEE International Symposium on*, pp. 470–475, September 2004.

[76] "Omg, common object request broker architecture (corba)." http://www.omg.org/spec/CORBA/.

[77] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM J. Res. Develop.*, vol. 10, pp. 278–291, 1966.

[78] M. Schulz, E. Trischler, and T. Sarfert, "Socrates: a highly efficient automatic test pattern generation system," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 7, pp. 126–137, January 1988.

[79] J. Darringer, D. Brand, J. V. Gerbi, W. Joyner, and L. Trevillyan, "Lss: A system for production logic synthesis," *IBM Journal of Research and Development*, vol. 44, pp. 157–165, Jan 2000.

[80] L. Stok, D. Kung, D. Brand, A. Drumm, A. Sullivan, L. Reddy, N. Hieter, D. J. Geiger, H. H. Chao, and P. Osler, "Booledozer: Logic synthesis for asics," *IBM Journal of Research and Development*, vol. 40, pp. 407–430, July 1996.

[81] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design Test of Computers, IEEE*, vol. 26, pp. 18–25, July 2009.

[82] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, pp. 473–491, April 2011.

[83] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[84] M. Fowler and R. Parsons, *Domain-Specific Languages*. The Addison-Wisley Signature Series, 2011.

[85] J. McCarthy, "Lisp for share distribution," *MIT Press*, pp. 93 – 99, 1962.

[86] "Rails: Web development that doesn't hurt." http://rubyonrails.org.

[87] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language*. O'Reilly Media, 2008.

[88] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377–387, June 1970.

[89] E. T. Ray, *Learning XML*. O'Reilly Media, 2003.

[90] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF, Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[91] O. M. Group, "Mof 2.0/xmi mapping, version 2.1.1," *formal/2007-12-01*, 2007.

[92] http://www.garshol.priv.no/download/text/bnf.html.

[93] L. Bettini, *Implementing Domain-Speicific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[94] http://www.antlr.org/.

[95] O. M. Group, "Unified modeling language (uml) superstructure specification, version 2.3," *formal/2010-05-05*, 2010.

[96] O. M. Group, "Meta object facility (mof) core specification, version 2.4 beta 2.0," *ptc/2010-12-08*, 2010.

[97] O. M. Group, "Mda guide version 1.0.1," *omg/2003-06-01*, June 2003.

[98] "Sysml open source specification project." http://www.sysml.org/.

[99] O. M. Group, "Uml profile for marte: Modeling and analysis of real-time embedded systems, version 1.0," *On line: http://www.omg.org/spec/MARTE/*, 2010.

[100] "Obect group management: Uml profile for schedulability, performance, and time." http://www.omg.org/spec/SPTP/.

[101] "The uml profile for marte: Modeling and analysis of real-time and embedded systems." http://www.omgmarte.org/.

[102] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet, "A co-design approach for embedded system modeling and code generation with uml and marte," *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pp. 226–231, April 2009.

[103] "Xilinx core generator system." http://www.xilinx.com/tools/coregen.htm.

[104] "Altera megafunctions." http://www.altera.com/products/ip/altera/mega.html.

[105] S. Edwards, "The challenges of synthesizing hardware from c-like languages," *Design Test of Computers, IEEE*, vol. 23, pp. 375–386, May 2006.

[106] N. Ranganathan, R. Namballa, and N. Hanchate, "Chess: a comprehensive tool for cdfg extraction and synthesis of low power designs from vhdl," *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, pp. 6 pp.–, March 2006.

[107] Q. Wu, Y. Wang, J. Bian, W. Wu, and H. Xue, "A hierarchical cdfg as intermediate representation for hardware/software codesign," *Communications, Circuits and Systems and West Sino Expositions, IEEE 2002 International Conference on*, vol. 2, pp. 1429–1432 vol.2, June 2002.

[108] C.-J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems.," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, 1986.

[109] C. H. Gebotys and M. I. Elmasry, "Vlsi design synthesis with testability.," *in Proc. of the 25th ACM/IEEE Design Automation Conference, DACâĂŹ88*, pp. 16–21, 1988.

[110] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems.," *Communications of the ACM*, vol. 17, pp. 685–690, December 1974.

[111] K. Wakabayashi, "C-based behavioral synthesis and verification analysis on industrial design examples.," *in Proceedings ASPDAC*, pp. 344–348, 2004.

[112] E. Martin, O. Sentieys, H. Dubois, and J.-L. Philippe, "Gaut: An architectural synthesis tool for dedicated signal processors," *In Proceedings of the Design Automation Conference, EURO-DAC'93*, pp. 14–19, 1993.

[113] S. McCloud, "Catapult-c, synthesis-based design flow: speeding implementation and increasing flexibility," *White paper, Mentor Graphics*, 2004.

[114] Xilinx, "Vivado design suite.," *White Paper*, June 2012.

[115] Cadence, "C-to-silicon high-level synthesis tool," *On line: http://www.cadence.com/ products/sd/silicon_ compiler/*.

[116] I. Accelerated, "Impulse-c high-level synthesis tool," *On line: http://www. impulseaccelerated.com/*.

[117] Altera, "Implementing fpga design with the opencl sandard.," *White Paper*, November 2013.

[118] K. Shagrithaya, K. Kepa, and P. Athanas, "Enabling development of opencl applications on fpga platforms.," *Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 26–30, June 2013.

[119] G. Economakos, "Esl as a gateway from opencl to fpgas: Basic ideas and methodology evaluation.," *Panhellenic Conference on Informatics*, pp. 80–85, October 2012.

[120] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Press, 1996.

[121] G. Kahn, "The semantic of a simple language for parallel programming," *Information processing*, pp. 471–475, 1974.

[122] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 31, no. 1, pp. 24–35, 1987.

[123] H.-H. Wu, H. Kee, N. Sane, W. Plishker, and S. Bhattacharyya, "Rapid prototyping for digital signal processing systems using parameterized synchronous dataflow graphs," *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*, pp. 1–7, June 2010.

[124] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, pp. 127–144, January 2003.

[125] "Stateflow: Model and simulate decision logic using state machines and flow charts." http://www.mathworks.com/products/stateflow/index.html.