Towards hardware synthesis of a flexible radio from a high-level language

Mai-Thanh Tran

THESE DE DOCTORAT DE MAI-THANH TRAN

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

Mai Thanh TRAN

Towards Hardware Synthesis of a Flexible Radio from a High-Level Language

Thèse présentée et soutenue à Lannion, le 13 novembre 2018
Unité de recherche : IRISA (UMR 6074), Institut de Recherche en Informatique et Systèmes Aléatoires, École Nationale Supérieure des Sciences Appliquées et de Technologies
Thèse N° : RENNI 13903563

Rapporteurs avant soutenance :
Lillian BOSSUET, Professeur des Universités, Université Jean Monnet, Saint-Etienne
Guillaume VILLEMAUD, Maître de Conférences HDR, INSA Lyon

Composition du Jury :

Président : Christophe JEGO, Professeur des Universités, ENSEIRB-MATMECA Bordeaux
Examinateurs : Lillian BOSSUET, Professeur des Universités, Université Jean Monnet, Saint-Etienne
Sylvie KEROUEDAN, Maître de Conférences HDR, IMT Atlantique
Guillaume VILLEMAUD, Maître de Conférences HDR, INSA Lyon
Dir. de thèse : Emmanuel CASSEAU, Professeur des Universités, Université de Rennes 1
Co-enc. de thèse : Matthieu GAUTIER, Maître de Conférences, Université de Rennes 1

Table of Contents

Chapter 1. Introduction 13

1.1. 5G, Internet of Things (IoT) and Software-Defined Radio (SDR) 14

1.2. Proposed Methodology and Main Contributions 16

1.3. Outline 16

Part I - BACKGROUND 19

Chapter 2. Flexible Radio and Software-Defined Radio 21

2.1. Introduction 22

2.2. Digital Radio System 22

2.2.1. Symbol Mapping 23

2.2.2. Channel Access 25

2.3. Flexible Radio 26

2.3.1. Cognitive Radio 27

2.3.2. Introduction to Software-Defined Radio Architecture 28

2.4. Waveform of Interest: The 3GPP-LTE Standard 29

2.4.1. The 3GPP-LTE Physical Layer 30

2.4.2. Fast Fourier Transform 33

2.5. Software –Defined Radio (SDR) Platforms 35

2.5.1. General Purpose Processor (GPP) Approach 35

2.5.2. Co-processor Approach 36

2.5.3. Multiprocessor Approach 37

2.5.4. FPGA Approach 38

2.6. FPGA-based SDRs 39

2.6.1. Multi-waveform Configuration 39

2.6.2. Separated Configurations 40

2.6.3. Partial Reconfiguration 40

2.7. Conclusions 41

Chapter 3. High Level Synthesis and Hardware Reconfiguration on FPGAs. 43

3.1. Introduction 44
3.2. High Level Synthesis 44
3.2.1. HLS Fundamental 44
3.2.2. Advantages of HLS 46
3.2.3. HLS Tools 47
3.3. Hardware reconfiguration on FPGAs 53
3.3.1. Introduction 53
3.3.2. Dynamic Partial Reconfiguration 56
3.4. Conclusions 58
Part II - CONTRIBUTION 61
Chapter 4. Design Flow for Flexible Radio on FPGA-based SDRs 63
4.1. Introduction 64
4.2. Reconfiguration Methods for Software Defined Reconfiguration 64
4.2.1. Software Reconfiguration 64
4.2.2. Hardware Reconfiguration 65
4.2.3. Algorithmic Reconfiguration 66
4.3. Proposed Design Flow and System Architecture for Flexible Radio on FPGA s 66
4.3.1. System Architecture for Flexible Radio on FPGA s 67
4.3.2. Design Flow for Flexible Radio on FPGAs 69
4.3.3. Verification and Validation 70
4.4. Conclusions 73
Chapter 5. Design and Exploration of a Flexible FFT for LTE standard 75
5.1. Introduction 76
5.2. Radix-2 DIT FFT/IFFT in Vivado HLS 76
5.2.1. C code re-writing for Vivado HLS 78
5.2.2. Fixed-point FFT/IFFT in Vivado HLS 79
5.2.3. Conclusions 82
5.3. Reconfigurable Blocks 83
5.3.1. Software Reconfigurable Block 83
5.3.2. Hardware Reconfigurable Block 84
5.3.3. Algorithmic Reconfigurable Block 85

5.4. Latency Estimation 86
5.4.1. Computing an Estimate of The Latency 87
5.4.2. Experiment and Results for Latency Formula 88
5.5. Design Space Exploration in the Power-Of-Two Point FFT 90
5.5.1. DSE at Clock Frequency of 100 MHz 90
5.5.2. DSE at Other Clock Frequencies 93
5.5.3. Comparisons and Conclusions 94
5.6. Conclusions 97
Chapter 6. Designing a flexible FFT for LTE standard as a use case 99
6.1. Introduction 100
6.2. Proposed Flexible FFT Implementations for LTE Standard 100
6.2.1. Generator 101
6.2.2. X-QAM Modulator 101
6.2.3. Implementation for Software Reconfiguration 102
6.2.4. Implementation for Hardware Reconfiguration 102
6.3. Results of the Proposed Flexible FFT Implementation 103
6.3.1. Implementation for Software Reconfiguration. 103
6.3.2. Implementation for Hardware Reconfiguration. 104
6.3.3. Comparisons 104
6.4. Demonstration of the FFT Implementation for LTE Standard 105
6.4.1. Testbed Description 105
6.4.2. Virtex 6 ML 605 Evaluation Board 106
6.4.3. Required Software Development Tools 106
6.4.4. Experiments 107
6.5. Conclusions 107
Chapter 7. Conclusions and Perspectives 111
Publication 117
Bibliography 119

List of Figures

Figure 2.1 Digital radio system. 22
Figure 2.2 OSI model. 23
Figure 2.3 The constellations for BPSK, QPSK and 16-QAM. 24
Figure 2.4 The cognitive radio's cognition cycle. 27
Figure 2.5 Basic software-controlled radio. 28
Figure 2.6 Ideal software-defined radio: (a) transmitter, (b) receiver. 29
Figure 2.7 Realistic software-defined radio: (a) transmitter, (b) receiver. 29
Figure 2.8 LTE generic frame structure (Zyren and McCoy 2007). 30
Figure 2.9 Physical layer coding rate as a function of channel conditions and modulation scheme (Johnson 2012). 32
Figure 2.10 A basic SC-FDMA transmitter and receiver (Zyren and McCoy 2007). 32
Figure 2.11 An eight-point DFT, divided in four two-point DFTs with bit reversal on inputs (Jones 2006). 34
Figure 2.12 Multi-processor approach. 37
Figure 2.13 FPGA-based approach. 38
Figure 2.14 Separated configurations. 40
Figure 2.15 Partial reconfiguration. 41
Figure 3.1 Generic HLS design flow. 45
Figure 3.2 The Vivado HLS design flow. 50
Figure 3.3 RTL hierarchy after HLS synthesis. 50
Figure 3.4 Dataflow optimization (Xilinx 2016). 51
Figure 3.5 Pipeline optimization (Xilinx 2016). 51
Figure 3.6 Unrolled optimization example. 52
Figure 3.7 The reconfiguration chain for a Virtex FPGA. 54
Figure 3.8 Configuration uses 1 address dimension (left) and 2 address dimensions (right) (Bolchini, Miele and Sandionigi 2011). 54
Figure 3.9 Two methods of delivering a partial bit file (with Xilinx devices) (Dye 2012). 56
Figure 3.10 Partial reconfiguration software flow (Xilinx 2016). 57
Figure 4.1 Design approach based on hardware reconfiguration. 65
Figure 4.2 Tradeoff between resources and reconfiguration time for the different reconfigurations. 66
Figure 4.3 HLS-based design flow for a reconfigurable module using Software-Defined Reconfiguration. 67
Figure 4.4 System architecture for FPGA-based SDR. 68
Figure 4.5 Proposed design flow for Flexible Radio using FPGA-based SDR. 69
Figure 4.6 Step-by-step development flow using Xilinx tools. 70
Figure 4.7 Verification and validation tools for the flexible radio design flow. 71
Figure 4.8 Example of code level verification in Vivado HLS based on a Matlab-provided reference. 71
Figure 4.9 Example of baseband level verification using ChipScope Pro Analyzer for 64-QAM data. 72

Figure 4.10 Example of RF signal analysis: an OFDM-based RF spectrum. 73
Figure 5.1 Virtex-6 FPGA DSP48E1 Slice (Xilinx 2011) 81
Figure 5.2 Structure of the 1536-point FFT 86
Figure 5.3 The chronogram of the different steps of Algorithm 5. 87
Figure 5.4 FFT latency with unrolling factor U = 1 a) FFT 2048; b) FFT 128. 89
Figure 5.5 The latency vs number of DSP slices (S is for Simulation after synthesis and F is for the Formula estimate) (frequency = 100 MHz). 91
Figure 5.6 Timing chronogram of the third loop based on Algorithm 6. 92
Figure 5.7 Timing chronogram of the third loop based on Algorithm 7: a) U = 4; b) U = 8. 93
Figure 5.8 Latency vs number of DSPs for clock frequencies 200Mhz (left) and 500MHz (right). 93
Figure 5.9 Latency vs number of DSPs at frequency 50MHz. 94
Figure 5.10 The third loop computation time vs clock frequency for U = 1 (= time to compute a radix-2 FFT) 95
Figure 5.11 Time to compute 1 FFT vs clock frequency, U = 1 95
Figure 5.12 The third loop computation time vs clock frequency for U = 4 96
Figure 5.13 Timing chronogram of the third loop at 50 MHz: a) U = 1; b) U = 4. 96
Figure 5.14 Time to compute 1 FFT vs frequency, U = 4 97
Figure 6.1 Overview of the architecture of the multi-mode FFT. 100
Figure 6.2 The 64-QAM constellation with ChipScope Pro. 101
Figure 6.3 The architecture of the multi-mode FFT with software reconfiguration. 102
Figure 6.4 The architecture of the multi-mode FFT with hardware reconfiguration. 103
Figure 6.5 The testbed description. 105
Figure 6.6 The demonstration platform. 106
Figure 6.7 Received spectrum and 16-QAM received constellation for an AWGN channel. 107

List of Tables

Table 2.1 Downlink OFDM modulation parameters 31
Table 2.2 A comparative study between different approaches for SDR. 39
Table 3.1 General information about HLS tools..... 49
Table 3.2 Maximum bandwidths for configuration ports with Virtex architectures. 56
Table 5.1 Resources required (after synthesis) for a radix_2 block for different input data lengths. 81
Table 5.2 Output errors with different data lengths. 82
Table 5.3 Software reconfiguration synthesis results for a FFT with 2 modes (128/2048). 83
Table 5.4 Hardware reconfiguration synthesis results for a FFT with 2 modes (128/2048). 85
Table 5.5 The latency of loop number 3 depending on unrolling factor 89
Table 5.6 Estimated latency and latency obtained by simulation after HLS. 90
Table 6.1 Performance of the multimode FFT for LTE standard with software reconfiguration 104
Table 6.2 Performance of the multimode FFT for LTE standard with hardware reconfiguration 104

List of algorithms

Algorithm 1 Software reconfiguration for the automatic generation of a multi-mode processing block..59
Algorithm 2 C program for the radix-2 DIT FFT…… …..71
Algorithm 3 Sine lookup table based code……..72
Algorithm 4 Two point FFT code……..73
Algorithm 5 Overview of the algorithmic reconfiguration based code for the power-of-two point FFT….79
Algorithm 6 Shortcut of the third loop in the FFT……..84
Algorithm 7 Shortcut of the updated third loop in FFT……..85

Acronyms and Abbreviations

ADC	Analog-to-Digital Conversion
ARM	Advanced RISC Machines
ASIC	Application Specific Integrated Circuits
ASK	Amplitude-shift keying
AWGN	Additive White Gaussian Noise
AXI	Advanced Extensible Interface
BER	Bit Error Rate
CDMA	Code Division Multiple Access
CLB	Configurable Logic Block
CP	Cyclic Prefix
CR	Cognitive Radio
DFT	Digital Fourier Transform
DIT	Decimation In Time
DPR	Dynamic Partial Reconfiguration
DSE	Design Space Exploration
DSL	Domain Specific Languages
DSP	Digital Signal Processor
FDMA	Frequency Division Multiple Access
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Arrays
FSK	Frequency-shift keying
GPL	General-purpose Programmable Language
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLL	High Level Language
HLS	High Level Synthesis
ICI	Inter-Carrier Interferences
ICT	Information and Communications Technologies
IDFT	Inverse Digital Fourier Transform
IoT	Internet of Things
LB	Logic Blocks
LTE	Long-Term Evolution
LUT	Look Up Table
NoC	Network on Chip
OFDM	Orthogonal Frequency Division Multiplexing
OFDM	Orthogonal Frequency-Division Multiple Access
OSI	Open Systems Interconnection
PAPR	Peak-to-Average Power Ratio

PER	Packet Error Rate
PHY	Physical Layer
PLB	Processor Local Bus
PRB	Physical Resource Block
PSK	Phase-shift keying
QPSK	Quadrature Phase Shift Keying
RAM	Random Access Memories
RF	Radio Frequency
RTL	Register-Transfer Level
SC-FDMA	Single-Carrier Frequency Division Multiple Access
SDK	Software Development Kit
SDR	Software-Defined Radio
TDMA	Time Division Multiple Access
VLIW	Very Large Instruction Word
VSA	Vector Signal Analyzer

Chapter 1.Introduction

1.1. 5G, Internet of Things (IoT) and Software-Defined Radio (SDR)

Wireless communication devices have been developed and raised rapidly last 20 years. They have completely changed modern lifestyle, enabling modern societies to operate more efficiently, and had a major impact on modern politics, economy, education, health, entertainment, logistics, travel and industry among others. Wireless communication enables the connection of people for content exchange, be it data and/or multimedia. The fifth generation of wireless communication, namely the 5G, addresses a new kind of users. In addition to connect people each other, the next breakthrough is the connectivity of machines with other machines, referred to as "M2M", which is the basis of the Internet of Things (IoT) paradigm [1] [2]. IoT refers to an Internet-like network, in which millions of embedded devices are connected together. These devices are various, and range from simple RFID tags to powerful smartphones. All of them are connected to wireless links more or less without any predefined infrastructures or communication standards. This network is opening an extreme breakthrough in the way people interact with traditional objects. Therefore, in addition to a massive increase in the number of accesses and throughput constraints, 5G standard has to take into account new requirements.

First, the use of M2M for virtual reality or human-to-machine interaction rely on steering/control communications with a cloud infrastructure. Hence, the required latency of such communications must be low enough to enable a round-trip delay from devices through the network back to devices of approximately 1ms [3]. As a comparison, LTE frame duration is about 5 or 10 ms.

In addition to the latency requirements, reducing energy consumption is an on-going major challenge for two reasons: the first one is the total energy consumption of ICT (Information and Communications Technologies) infrastructure that must be reduced; the second one is the duration of the activity time of the devices (*e.g.* Wireless Sensor Network nodes, IoT devices, etc.) that must significantly increase to be massively deployed and used.

The last requirement we want to highlight is the flexibility the devices must provide to satisfy the number of modes related to the different communication standards. Indeed, in order to meet the time and space fluctuations of a service, in order to make several classes of objects be connected together and in order to answer the spectrum scarcity issue, it is expected that the new generation of wireless communications should be able to change their own features in real time.

Hence, a new 5G standard is needed to tackle these challenges (throughput, latency, energy consumption and flexibility). Focusing on the baseband processing of the physical layer, two main issues are raised by these new requirements: (#i) what are the signal processing techniques that could help improving the quality of the link, spectrum usage and energy efficiency? (#ii) what kind of hardware could associate flexibility, energy efficiency and high-performance computing of these signal processing techniques?

A huge effort is currently spent on proposing new physical layers and many digital communication techniques have been widely studied to tackle issue (#i) such as the Cognitive Radio (CR) [4], the Cooperative Radio [5] or the Green Radio [6]. However, few studies address issue (#ii). The goal of this PhD thesis is to study new architectures for the implementation of such innovations as well as new

design methodologies. To this aim, two contradictory issues must be faced. The first one is the flexibility required, with the important number of modes for a single protocol and the number of protocols to be supported by a single device. The second concern is related to performance and power consumption requirements.

In the past, radio engineers used to focus on implementing and optimizing a single communication protocol (or waveform) at a time. Application Specific Integrated Circuits (ASICs) were most of the time used because theses circuits allow both high throughput and low power consumption. However, this kind of circuits is not flexible and requires the application to be described at Register-Transfer Level (RTL) using a Hardware Description Language (HDL) such as VHDL or Verilog, which is quite a tough task. This approach is quite obsolete while current radio devices must be flexible *i.e.* must be able to reconfigure their protocol among different numbers of modes for a given waveform or among different waveforms. A promising technology, which can bring these needs into practice, is SDR [4] [7] [8]. SDR allows the fast prototyping of a waveform from its high-level description, specified in C/C++ for instance, on a hardware device that can be reconfigured. Unlike other approaches that use a dedicated hardware for a waveform, SDR concept relies on a single software-based technology such as microprocessors, which can be reprogrammed to implement signal processing for all possible waveforms. The SDR device is controlled by a software program and can be easily configured at different levels: from simple processing blocks to full waveforms whenever necessary. However, in practice, if this approach is clearly more flexible than earlier ones and allows fast prototyping, it also usually decreases performance and increases power consumption.

An interesting trade-off between performance and power consumption could be achieved using Field Programmable Gate Arrays (FPGAs) circuits. FPGAs enable the design of customized circuit architectures through a configuration file. It can be reprogrammed using other configuration files to implement different functionalities. FPGA-based SDR is an old paradigm [9] offering in theory a good tradeoff between flexibility and processing power. The programmability of such platform is however a bottleneck as FPGAs need a HDL description as an entry in the design flow.

This PhD aims at improving the programmability of FPGA-based SDR through the use of high-level specifications instead of HDL ones. Fast prototyping capability is achieved by leveraging High-Level Synthesis (HLS) techniques and related tools to generate RTL descriptions from high-level specifications [10] [11]. If HLS fast prototyping capability enables the compile-time flexibility of a FPGA-based SDR, run-time flexibility is still an open issue. This thesis addresses consequently the FPGA-based implementation of a run-time hardware reconfiguration of a flexible waveform from its high-level description. The proposed methodology mainly aims at analyzing the performance of using a multi-mode processing block with control signals or Dynamic Partial Reconfiguration (DPR) to provide in both cases flexibility.

1.2. Proposed Methodology and Main Contributions

The proposed methodology is a guideline to completely build a flexible radio on FPGA-based SDR, which can be reconfigured at run-time. The flow goes from the description of the waveform to the implementation, verification and validation steps of the system. Its entry is a description with a high-level language that leverages HLS techniques to build a radio system. Two kinds of approaches are used to address the run-time flexibility of a FPGA- based SDR. The first one proposes to automatically design multi-mode RTL components with control signals to switch between the different modes (similar to [12]). The other approach is based on dynamic partial reconfiguration (DPR). Run-time DPR, referred to as *Hardware reconfiguration* in the following of this PhD report, is the ability to reconfigure a part (or parts) of the FPGA (*e.g.* a functionality at the hardware level) while the rest of the FPGA continues to work. It has been a research topic since the 90s [13] and it can be currently used in FPGAs, since Xilinx and Altera companies promote such technology [2] [5]. The main advantage of the hardware reconfiguration is to provide both hardware flexibility and hardware area reuse, allowing power consumption and device cost reductions.

Here HLS concept is complementary to DPR as it allows the rapid generation of the different configurations that can then be implemented in the reconfigurable partitions of the FPGA. Moreover, the different configurations can be derived from a single functional block for which the modes have different properties (*e.g.* low power consumption/low performance or high performance/high power consumption). The use of a HSL tool is also very useful as most HLS tools enable a fast exploration of the design space (DSE, for Design Space Exploration) using several compile-time optimization techniques. They can implement for instance latency or throughput, power, memory or area optimizations. HLS enables performing such optimizations from high-level specifications, which considerably accelerates the design process.

To summarize, our contributions are:

- Proposing a complete design flow for FPGA-based SDR that embeds functional blocks or waveforms that can be reconfigured at run-time. This flow leverages both HLS and DPR technologies to enhance both prototyping time and hardware resource utilization. The flow also allows the DSE of the functional blocks.
- Applying this proposal to design, as an example, a flexible Fast Fourier Transform (FFT) for LTE standard that requires different FFT sizes. A DSE of the FFT is also performed with several frequencies and directives at the HLS step. Based on this case study, tradeoffs between used resources, reconfiguration time, design effort and performance are discussed.
- Designing LTE-like waveforms with run-time reconfiguration capabilities at both modulation and FFT levels for demonstration. In our case, the system is implemented and validated on a Xilinx Virtex 6 based board.

1.3. Outline

This PhD report is divided into two major parts. The first part is the background, presented in chapters 2 and 3. This part provides an overview of SDR concepts and related hardware platforms, HLS principles,

hardware reconfiguration and other relevant concepts. The second part, from Chapter 4 to Chapter 6, is related to the contributions. It includes all the details towards our proposal of a flexible radio on FPGA-based SDRs and presents the implementation of LTE-like waveforms on a FPGA following our proposal.

The detailed description of the chapters is:

- Chapter 2 provides the basic concepts of digital radio and SDR concept. In addition, this chapter presents an overview of 3GPP-LTE standard, a standard which is relevant to our work.
- Chapter 3 talks about HLS and hardware reconfiguration on FPGAs. In this chapter, the principles of HLS are presented and both the advantages and disadvantages of HLS are discussed. This chapter also provides a short survey of some HLS tools. This chapter also discusses a feature which has been developed on FGPA technology for a long time but has not yet got enough concern: dynamic partial reconfiguration.
- Chapter 4 provides an overview of our proposal for FPGA-based SDR based on HLS. It presents our system architecture as well as our proposed design flow for flexible radio on FPGAs.
- Chapter 5 discusses the implementation and design space exploration of the FFT tuned for the LTE standard. Synthesis results are analyzed and some conclusions are drawn.
- Chapter 6 is dedicated to the implementation of a flexible FFT for LTE standard. Based on all previous results, two flexible FFT components are investigated. In this chapter we also present a demonstration platform with run-time reconfiguration capabilities for experimenting a LTE-like waveform using one of the flexible FFT components.
- Chapter 7 is a conclusion chapter, which reviews the entire works depicted in this document and discusses perspectives.

Part I – BACKGROUND

Chapter 2.Flexible Radio and Software-Defined Radio

2.1. Introduction

Nowadays, communication systems increasingly play an important role in our society. Although first radio systems were analog ones, digital radio systems [14] [15] [16] gradually phase out analog ones with their advantages in terms of data rate and range. Most of current digital communication systems are usually composed of a baseband processing module to encode the bit stream into a symbol stream suitable to the propagation channel and a front-end module to transpose the baseband signal on a carrier wave. The diagram in Figure 2.1 shows the main basic blocks in a typical digital radio system. In practice, there are diverse types of digital radios and every block of Figure 2.1 is not always used.

More and more new standards born for different applications lead to various modulation techniques and specific requirements. Those new requirements bring a demand for flexible radio communication systems, which can support multiple communication protocols. In this chapter, we first introduce digital radio system fundamentals in Section 2.2. Then, the concept of flexible radio systems is introduced in Section 2.3 and the waveform we are mainly interested in is introduced in Section 2.4. Section 2.5 discusses the main platform-based approaches for SDR and Section 2.6 specifically presents FPGA-based SDR. Finally, section 2.7 concludes this chapter.

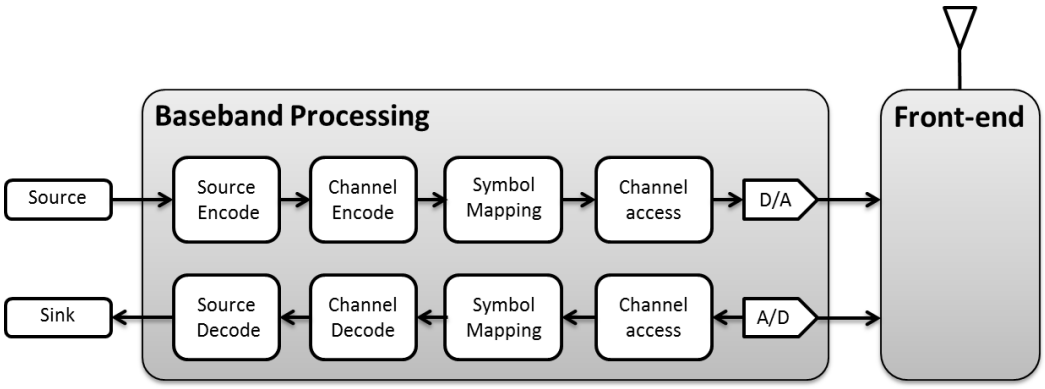


Figure 2.1 Digital radio system.

2.2. Digital Radio System

The goal of a digital radio system is to transmit information between two distant devices through a propagation channel (wired or wireless). The internal functions of a telecommunication system are usually referred to a standard, the Open Systems Interconnection (OSI) model [17], shown in Figure 2.2. In this model, the Physical Layer (PHY) is the lowest layer, which is composed of hardware transmission technology and plays an important role within the OSI model since all higher layers access to the channel through it. Because our research work focuses on PHY specifications and implementations, two main aspects of a PHY are discussed in the following sections: the symbol mapping and the channel access.

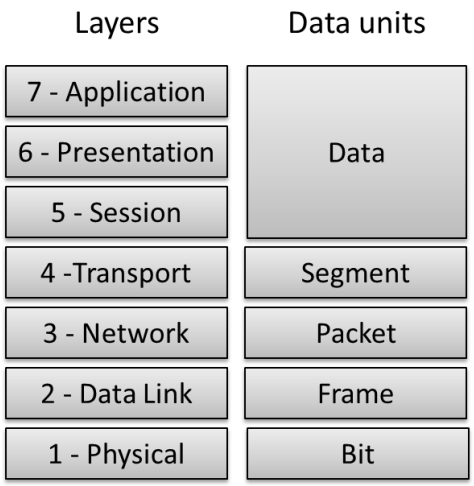


Figure 2.2 OSI model.

2.2.1. Symbol Mapping

Digital modulation [18] [19] is the step where a stream of data bits is converted into a stream of symbols which are suitable to transmission through the channel. The symbols are predefined in an alphabet of symbols being either real or complex. The choice of symbol mapping techniques depends on the transmission requirements *i.e.* mainly data rate and Bit Error Rate (BER). The three most fundamental modulation techniques are Phase-Shift Keying (PSK), Amplitude-Shift Keying (ASK) and Frequency-Shift Keying (FSK).

PSK modulation

The symbols of this modulation are provided using different phases of the carrier. The amplitude of the symbols is constant. So the alphabet symbols are mapped on the circumference of a circle at a regular angular distance. The different phases are given by:

$$\theta_m = \frac{2\pi m}{M} + \frac{\pi}{M}, \qquad m = 0, 1, .., M-1, \tag{2.1}$$

where M is the number of alphabet symbols. A symbol represents therefore a set for $\log_2(M)$ bits.

In the context of an Additive White Gaussian Noise (AWGN) channel, the symbol-error probability can be computed by knowing exactly the bit-mapping. In the case of Quadrature Phase Shift Keying (QPSK), a particular case of an M-PSK modulation where M = 4, the theoretical bit error probability is given by:

$$\mathrm{P_{BER}} = \mathrm{Q}\left(\sqrt{\frac{\mathrm{E_S}}{\mathrm{N_0}}}\right), \tag{2.2}$$

with E_S the energy per transmitted symbol, N_0 the spectral density of the noise and Q (.) the Gaussian density function given by:

$$Q(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}, \tag{2.3}$$

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

The symbol error rate is given by:

$$\mathrm{P_{SER}} = 1 - \left(1 - \mathrm{Q}\left(\sqrt{\frac{\mathrm{E_S}}{\mathrm{N_0}}}\right)\right)^2 \tag{2.4}$$

ASK modulation

In the ASK modulation, digital data are represented by the variation in the amplitude of a carry wave. The distance between two consecutive symbols is constant and is equal to $2\sqrt{E_S}$ and the amplitude s_m of the symbols is given by:

$$s_m = \sqrt{E_S}(2m - 1 - M), \qquad m = 0, 1, .., M - 1, \tag{2.5}$$

The theoretical probability of symbol error in the case of a AWGN channel is given by:

$$\mathrm{P_{SER}} = \frac{2(\mathrm{M}-1)}{M} Q\left(\sqrt{\frac{6P_{av}T_s}{(M^2-1)N_0}}\right), \tag{2.6}$$

where P_{av} is the average power, T_s is the symbol period.

Quadrature amplitude modulation (QAM)

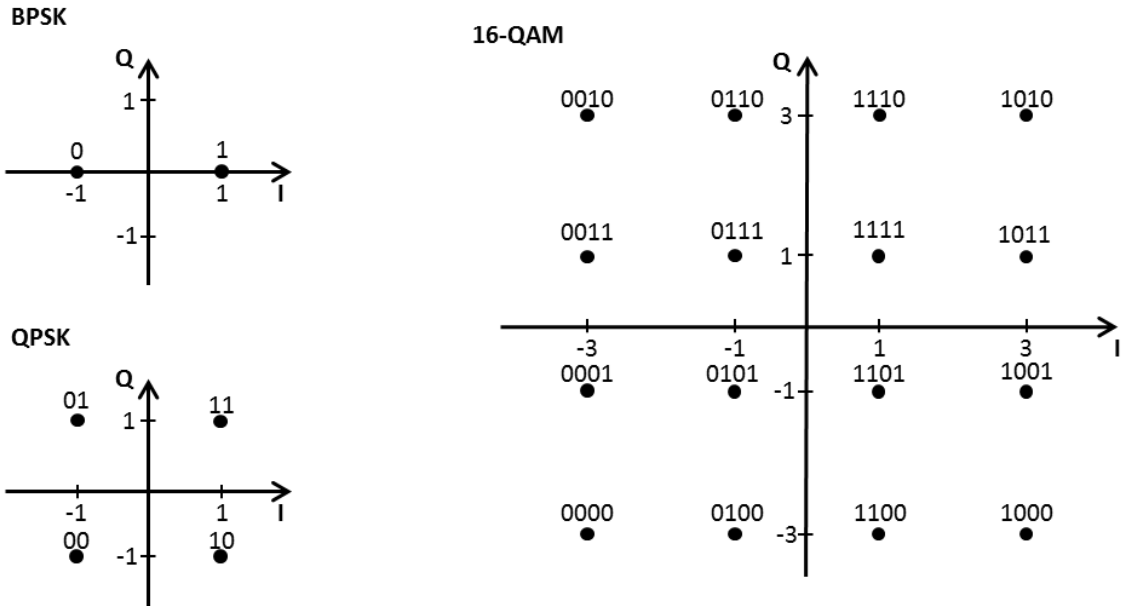


Figure 2.3 The constellations for BPSK, QPSK and 16-QAM.

Combining PSK and ASK techniques leads to QAM modulation. Indeed, the alphabet symbols in QAM are distinguished by both amplitude and phase and enable achieving higher data rates than PSK and ASK modulations. The complex symbols of this modulation are given by:

$$s_m = \sqrt{E_S}(2m - 1 - M)e^{j(\frac{2\pi m}{M} + \frac{\pi}{M})}, \qquad m = 0, 1, .., M - 1. \tag{2.7}$$

In the context of AWGN, the theoretical symbol error probability is given by:

$$\mathrm{P_{SER}} = 4\left(\frac{\sqrt{M}-1}{\sqrt{M}}\right)\left(\frac{1}{log_2 M}\right)\sum_{i=0}^{\frac{\sqrt{M}}{2}-1} Q((2i+1)\sqrt{\frac{E_b}{N_0}\frac{3log_2 M}{M-1}}). \tag{2.8}$$

Figure 2.3 shows the M-QAM constellations for M = 2, 4, 16.

FSK modulation

The FSK symbols are provided based on several frequencies. In other words, M alphabet symbols are represented by M separated frequencies. Δf is the frequency separation between two consecutive symbols. The set of possible frequencies is:

$$\mathrm{f}_m = \mathrm{f}_c + (2\mathrm{m} - 1 - \mathrm{M})\frac{\Delta f}{2}, \qquad m = 0, 1, .., M - 1, \tag{2.9}$$

with f_c the carrier frequency.

There are other mapping techniques developed and deployed throughout the decades such as the Minimum Shift Keying [20] used for the GMS telecommunication standard and most of them relies on PSK, ASK or FSK.

Besides signal modulation, channel access is one of the most important parts in a digital telecommunication system. This technique is discussed in the next section.

2.2.2. Channel Access

Since the spectral resource is scarce and therefore limited per standards, different channel access methods have been developed to optimize its use. Multiple access techniques allow the signal from different sources to be combined in order to share the usage of a communication channel. There are some fundamental types of channel access schemes [21] such as Frequency Division Multiple Access (FDMA), Time Division Multiple Access (TDMA), Spread Spectrum Multiple Access, Space Division Multiple Access and Power Division Multiple Access.

In this section, we will focus on the Orthogonal Frequency Division Multiplexing (OFDM) technique, an advanced form of FDMA, which has been widely used in recent years. This technique is relevant to our work since it is used in the standard we target.

Orthogonal Frequency Division Multiplexing (OFDM)

OFDM divides the incoming data symbols into N parallel streams corresponding to the sub-channels. Each sub-channel is modulated independently. This channel access technique belongs to FDMA and results in a lower data rate per sub-channel. In OFDM technique, the sub-channels are orthogonal to each other, meaning theoretically the Inter-Carrier Interferences (ICI) is eliminated unlike conventional

FDMA. Thus, the OFDM approach reduces inter-carrier guard bands and simplifies the design for transmitter. Indeed, the Digital Fourier Transform (DFT) and the Inverse Digital Fourier Transform (IDFT) are used to modulate and demodulate the symbols respectively. Equation (2.10) gives the modulation operation while (2.11) gives the inverse process.

$$x_k = \frac{1}{N}\sum_{k=0}^{N-1} X_n e^{j(\frac{2\pi kn}{N})}. \quad (2.10)$$

$$X_n = \sum_{k=0}^{N-1} x_k e^{-j(\frac{2\pi kn}{N})}. \quad (2.11)$$

In practice, Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) algorithms are used to reduce the complexity of the implementation of the DFT and the IDFT respectively.

The data in OFDM is transmitted on many sub-channels. Each sub-channel transmits a subset of the data at a reduced rate. This reduction incurs a long OFDM symbol duration that reduces the effects of multi-path propagations. In addition, a guard interval named Cyclic Prefix (CP) is added at the beginning of the transmitted symbol. CP consists in the copy of the end part of the OFDM symbol at the beginning of it. It allows the total annulation of inter-symbol interferences and thus enable the use of a very simple equalization at the reception, with only one coefficient per sub-channel in the frequency domain, etc.

Nowadays, OFDM modulation is widely used for wide-band communications because of its advantages. It has resilience to interference, high spectrum efficiency and it is less sensitive to frequency selective fading than single carrier systems. In addition, the channel equalization of OFDM is much simple than the one of single carrier or CDMA systems. However, OFDM also has some disadvantages such as its sensitivity to Doppler shift, to frequency synchronization and so on.

In conclusion, OFDM is parameterized by the number M of subcarriers. The choice of M depends on the requirements in terms of bandwidth and sensitivity to multi-path channel. Therefore, the value of M can be different between two OFDM-based standards and many standards such as LTE specify multiple values for M. A detailed example of OFDM modulation, LTE standard specification as well as FFT transform will be deeply discussed in the Section 2.4.

2.3. Flexible Radio

First generations of radio systems were designed for fixed environment using specific waveforms. A single-purpose device is quite simple to optimize for both performance and power consumption. Gradually, requirements for multi-purpose devices have grown. There is an increasing need for multi-task devices, which are able to support multiple communication protocols. In this context, flexible radios then refer to radio systems that are capable for instance to switch their waveform according to different environments or to follow user specifications.

2.3.1. Cognitive Radio

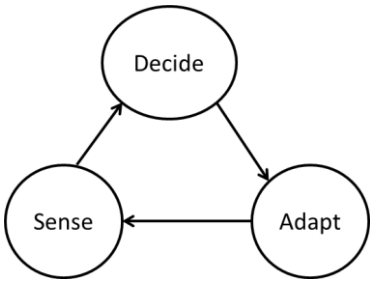


Figure 2.4 The cognitive radio's cognition cycle.

Cognitive Radio (CR) [22] is a concept proposed by Joseph Mitola. This radio system is aware of the environment in which it is being operated and automatically changes its parameters to optimize performance (*e.g.* data rate, spectrum usage, power consumption) according to the changing context. In other words, such system is theoretically an autonomous system which can know and decide the most effective way to transform the data to set up the configuration. Actually, the ambitions for cognitive radio are mainly dealing with spectrum-aware radios, which can change the frequency bandwidths they use according to other systems (*e.g.* legacy primary standards or interference systems). A cognition cycle is shown in Figure 2.4. It is composed of three states which illustrate the main features of a cognitive radio. They are the *Sense*, the *Decide* and the *Adapt* states. The *Sense* state collects all information from environment. Based on this information, the *Decide* state gives the decision to choose the suitable configuration and the *Adapt* state controls the system to be reconfigured for the new selected configuration. Spectrum-Aware radio and Multi-Standard radios are two key examples of CR. They will be introduced in the following paragraphs.

Spectrum Aware Systems

In the telecommunication domain, spectrum is quite a scare resource. Thus, spectrum-aware radios turn out to be one of the essential research directions of cognitive systems. Indeed, there has been a growing need for taking advantage of the under-used allocated spectrum. Radio systems target to achieve better performance from a clever usage of the spectrum. For instance, there is a lot of interest in cognitive systems focusing on the white spaces in the TV band (470 to 862 MHz in Europe). In this context, opportunistic systems should operate without disturbing the incumbents (*i.e.* the TV broadcast communications). Part of the *Sense* step, [23] [24] [25] study spectrum sensing techniques that mainly rely on signal processing methods to detect the incumbents. Among those methods, [23] makes use of the energy detector to detect the presence of an incumbent by thresholding the energy in the channel. Another solution uses the cyclostationarity detector [24] [25] which detects digital modulations through their cyclostationarity properties. The cyclostationarity detector is based on the fact that most of the digital modulations have cyclic frequencies due to the periodical digital computations.

Multi-Standard Systems

Another type of cognitive system is multi-standard systems [26] [27]. This kind of cognitive system focuses on the capability to operate with different telecommunication standards. This type focuses on

the *Adapt* step of the cognitive cycle. It would be more efficient if the radio system has the capacity to switch between a lower data rate standard and a high data rate standard based on the specific application before initiating the communication. For instance, a high data rate standard (*e.g.* used for video applications) may not be appropriate to transmit low data rate signal such as voice and it would be more efficient in such case to switch to a lower data rate standard before initiating the voice communication. Moreover, because of the limitation of coverage deepening on the telecommunication operators, multi-standard systems are necessary for different geographical regions. For example, our current mobile phone often has to change between different mobile standards in order to ensure a permanent network access. This is currently achieved mostly by integrating a dedicated chip for each standard and a software control is in charge of switching at run-time between standards. From an implementation perspective, this current approach is not efficient and not suitable for the long term.

2.3.2. Introduction to Software-Defined Radio Architecture

SDR is another concept proposed by Joseph Mitola [7] [8]. It is a concept of a flexible radio system which has the ability to support different waveforms without changing the hardware. The term software-defined means that the waveform can be modified from software or firmware directives while the hardware of the radio is not modified. SDR presents several advantages and is actually a promising direction for IoT. Indeed, SDR has high interoperability, great capacity in frequency reuse as well as efficiency in using limited resources under varying conditions [28]. These features are important points to build an IoT system.

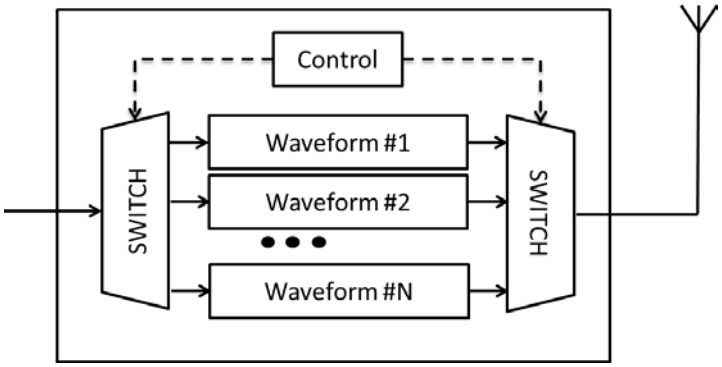


Figure 2.5 Basic software-controlled radio.

In the OSI model in Figure 2.2, PHY is the lowest layer of the model in charge of the bit transmission over the channel. A multi-waveform radio requires a PHY suitable to all the waveforms. Nowadays, many devices support multi-waveform radio but most of them have a dedicated hardware module for each waveform and use software to control which waveform is used. It is called software-controlled radio and its basic form is illustrated in Figure 2.5. An ideal SDR is composed of a single hardware module which can be fully reprogramed according to the needed waveform. The ideal SDR is given in Figure 2.6. A single micro-processor is processing the binary data that are modulated and digitally transposed to a Radio Frequency (RF) signal before being sent to the antenna.

The ideal SDR hardware has therefore the capacity to support any waveforms, frequencies and

bandwidths. However, we are still far from implementing an ideal SDR and many technical challenges must be faced to achieve such ideal radio in practice [28]. First, the ideal SDR antenna must be able to capture a wide range of frequencies, from very low frequencies to very high frequencies (*e.g.* between 1 MHz and 6 GHz). However, most antennas are currently based on mechanical structures making those wide frequency ranges out of their capability. The wide band also imposes stringent requirements on the analog-to-digital conversion (ADC) of the receiver and on channel selection. Converting the RF signal just after the antennas requires high sampling frequency ADCs. Furthermore, the high rate data stream generated by the ADCs cannot be supported by current microprocessors. Indeed, if programmability is eased when considering microprocessors, their computation speed is limited.

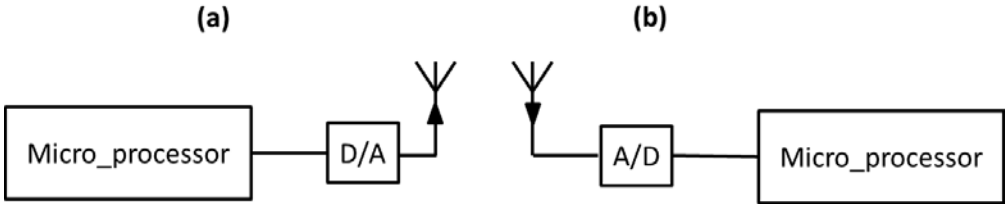


Figure 2.6 Ideal software-defined radio: (a) transmitter, (b) receiver.

Because an ideal SDR is difficult to achieve right now, a RF Front-End can be inserted after the antennas to reduce the target bandwidth and to translate the RF signal to baseband. This model of more realistic SDR is illustrated in Figure 2.7. The baseband processor computes most steps such as filtering, modulation, channel equalization and so on. This processing uses software intensive approach as much as possible to ease programmability.

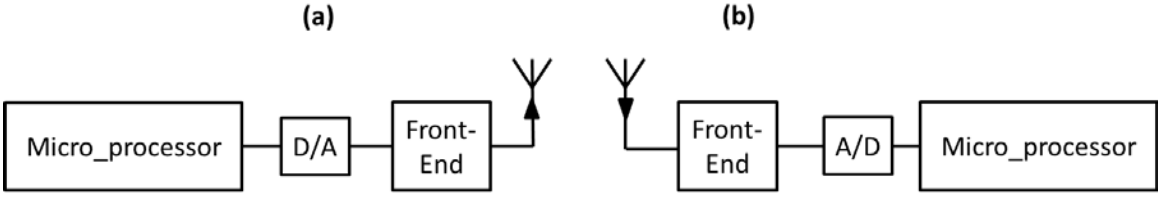


Figure 2.7 Realistic software-defined radio: (a) transmitter, (b) receiver.

2.4. Waveform of Interest: The 3GPP-LTE Standard

As introduced in Section 2.2.2, OFDM technique is an advanced form of FDMA that uses orthogonal sub-carriers. It means ICI is eliminated and inter-carrier guard bands are no longer necessary as for a traditional FDMA, therefore increasing spectral efficiency. Because of its advantages, it is widely used in current wideband communications. Thus, there are plenty of different OFDM-based standards such as the wireless LAN (IEEE 802.11a), the terrestrial digital TV systems (DVB-T and ISDB-T), the terrestrial mobile TV systems (DVB-H, T-DMB and ISDB) and the downlink of the 3rd Generation Partnership Project Long Term Evolution (3GPP-LTE). Since OFDM technique is based on parallel streams, many OFDM standards use various numbers of streams as a parameter in the design. For example, DVB-T2 has six modes ranged from 1k (1024) to 32k (32768) sub-carriers. This leads to the requirement of flexible configurations in the device and FPGAs with dynamic partial reconfiguration, as we will see later, can be a good solution for this requirement.

In this section, we will discuss about the 3GPP-LTE, another OFDM standard using various number of sub-carriers. In this standard, there are six modes with various numbers of streams from 128 to 2048. This is the standard we used to apply our proposal as a case study. In addition, this section also introduces the FFT algorithm, a popular and effective way to create orthogonal frequency in OFDM.

2.4.1. The 3GPP-LTE Physical Layer

3GPP-LTE is a standard used for mobile phones for high-speed wireless communication. This standard is developed by the 3GPP providing for an uplink speed up to 50 Mbps and a downlink up to 100 Mbps. OFDM and MIMO are two key technologies for LTE which establish the major advantages over 3G systems using Code Division Multiple Access (CDMA). LTE uses different modes for downlink and uplink. The downlink physical layer of LTE is based on Orthogonal Frequency-Division Multiple Access (OFDMA) and the uplink is based on Single-Carrier Frequency Division Multiple Access (SC-FDMA).

2.4.1.1. Generic frame structure

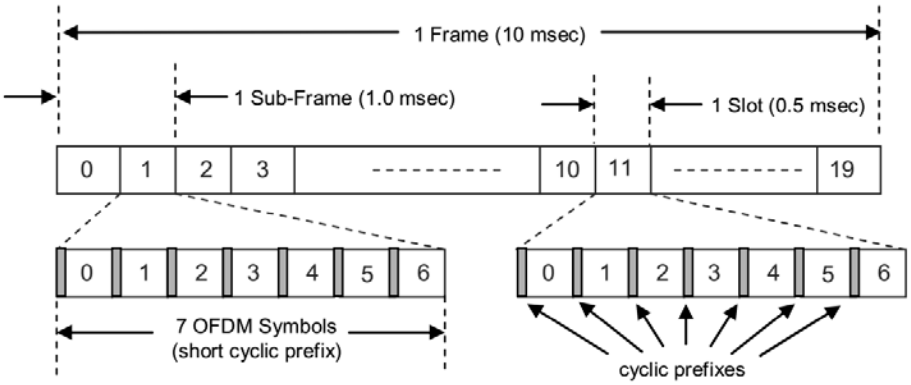


Figure 2.8 LTE generic frame structure [29].

LTE transmissions are segmented into frames. The generic frame structure is the same with both uplink and downlink for Time-Division Duplex (TDD) operation. A frame takes 10ms as shown in Figure 2.8. There are 20 slot periods of 0.5 ms in a frame. One sub-frame contains two slot periods and lasts 1 ms.

The use of normal CP or extended CP is decided depending on the channel delay spread. A slot has 6 or 7 OFDM symbols. The number of symbols depends on the extended or normal CP as shown in Figure 2.8.

2.4.1.2. Downlink

3GPP-LTE PHY specifications were firstly designed with bandwidths from 1.25 MHz to 20 MHz. Then, the bandwidth of 1.25 Mhz was changed to 1.4 MHz to include guard bands and to help with emission [30]. Currently, the specification has six bandwidths: 1.4, 3, 5, 10, 15, and 20 MHz.

Modulation parameters: OFDM is designed for the basic modulation scheme. Table 2.1 summarizes OFDM modulation parameters. There are six transmission bandwidths 1.4, 3, 5, 10, 15, and 20 MHz as said previously. The number of sub-carriers ranges from 128 to 2048, depending on the channel bandwidth. In OFDM, the number of sub-carriers corresponds to the FFT size. Thus, a transceiver which supports all LTE bandwidth has to capacity to deal with various FFT sizes.

Downlink multiplexing: OFDMA method is used as the multiplexing scheme in the LTE downlink. This method is interesting in terms of efficiency and latency. In OFDMA, users are allocated a specific number of subcarriers for a predetermined amount of time. In the LTE specifications, they are called as Physical Resource Blocks (PRBs). PRBs are divided in both time and frequency domains. The number of PRBs ranges from 6 to 100 depending on the bandwidth.

Transmission BW		1.4 MHz	3 MHz	5 MHz	10 MHz	15 MHz	20MHz
Sub-frame duration		0.5 ms					
Sub-carrier spacing		15 kHz					
Sampling frequency		1.92 MHz	3.84 MHz	7.68 MHz	15.36 MHz	23.04 MHz	30.72 MHz
FFT size		128	256	512	1024	1536	2048
OFDM sym per slot (short/long CP)		7/6					
CP length (usec/samples)	**Short**	(4.69/9)x6, (5,21)/10)x1	(4.69/18)x6, (5,21)/20)x1	(4.69/36)x6, (5,21)/40)x1	(4.69/72)x6, (5,21)/80)x1	(4.69/108)x6, (5,21/120)x1	(4.69/144)x6, (5,21/160)x1
	Long	(16.67/32)	(16.67/64)	(16.67/128)	(16.67/256)	(16.67/384)	(16.67/512)

Table 2.1 Downlink OFDM modulation parameters

In contrast to packet-oriented networks such as 802.11a, 3GPP-LTE does not use a PHY preamble to ease carrier offset estimation, channel estimation, timing synchronization, etc. 3GPP-LTE uses special reference signals embedded in the PRBs. Reference signals are transmitted from the first to the fifth OFDM symbols of each slot with normal CP and from the first to the fourth OFDM symbols with the extended CP.

2.4.1.3. Uplink

The uplink of the LTE PHY uses SC-FDMA as the basic transmission scheme. The basic transmitter and receiver architecture of SC-FDMA is quite similar to OFDMA. However, SC-FDMA has a great advantage over conventional OFDM by reducing the Peak-to-Average Power Ratio (PAPR). Indeed, SC-FDMA can reduce PAPR by approximately 2 dB compared to conventional OFDMA. In TDD, the uplink uses the same frame structure as for the downlink (Figure 2.8). The subcarrier spacing in uplink is also 15 kHz.

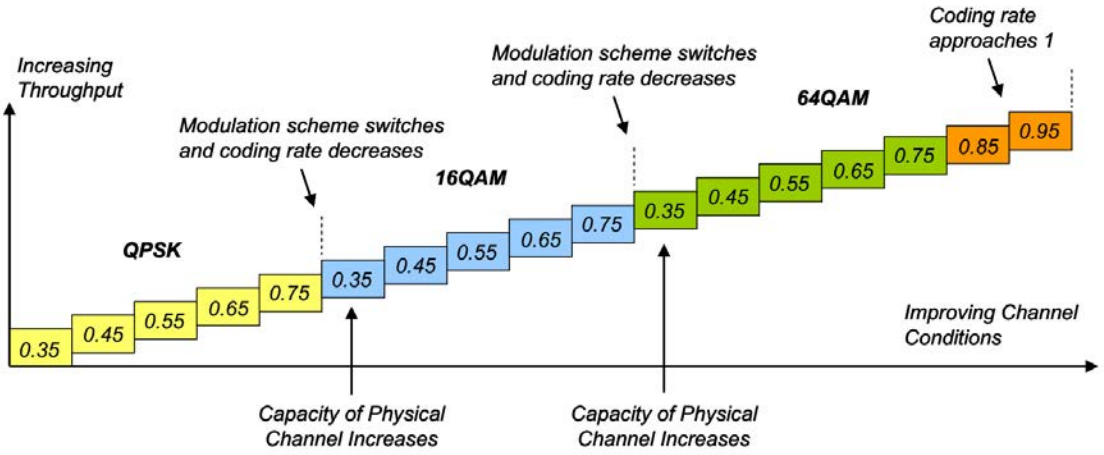


Figure 2.9 Physical layer coding rate as a function of channel conditions and modulation scheme [31].

Modulation parameters: The uplink modulation parameters (including normal and extended CP length) are the same as the downlink ones. However, the subcarrier modulation is very different.

In the uplink, depending on the channel quality, the data is mapped into a different signal constellation such as QPSK, 16 QAM or 64 QAM. Figure 2.9 shows the physical layer coding rate as a function of channel conditions and modulation scheme. If the conventional OFDM uses the QPSK/QAM symbols to directly modulate subcarriers, the LTE PHY uplink feds serial uplink symbols into a serial/parallel converter. The data, then, is given to a FFT block. Thus, the result of the FFT block is a discrete frequency domain representation of the QPSK/QAM symbols. These outputs are mapped to subcarriers before being fed to IFFT block. This process is illustrated in Figure 2.10.

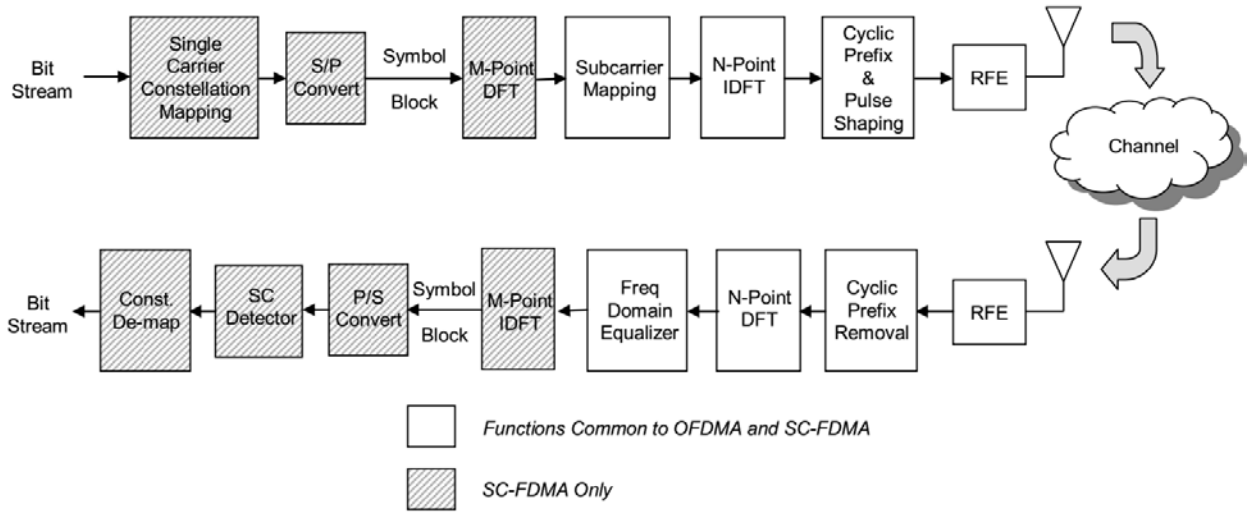


Figure 2.10 A basic SC-FDMA transmitter and receiver [29].

SC-FDMA: SC-FDMA is a misleading term. Indeed, SC-FDMA is a multi-carrier scheme which shares many similar function blocks with conventional OFDMA as shown in Figure 2.10.

LTE uplink requirements have some differences with downlink requirements. Indeed, for a user equipment terminal, power consumption is a key consideration. Power consumption is a vital requirement for the uplink. Unfortunately, OFDM has high PAPR and then does not suit LTE uplink. As a result, it is replaced by SC-FDMA which well suits the uplink requirements because the underlying waveform is essentially a single-carrier with lower PAPR.

The similar function blocks in SC-FDMA and OFDMA are interesting because the user equipment terminal can share the common parts between the uplink and downlink. Figure 2.10 shows a basic SC-FDMA transmitter and receiver arrangement. The functional blocks in the transmitter chain are:

1. Constellation mapper: Takes a sequence of streaming bits and converts them into single carrier symbols (BPSK, QPSK, 16QAM or 64 QAM depending on channel conditions).
2. Serial/parallel converter: Converts serial symbols into blocks to feed DFT blocks.
3. M-point DFT: Converts symbols from time domain into discrete frequency domain.
4. Subcarrier mapping: Maps outputs from DFT block into specified subcarrier before transmission. In SC-FDMA, sub carriers can be mapped in two ways: either localized or distributed mode.
5. N-point IDFT: Converts subcarriers into time domain.
6. Cyclic prefix and pulse shaping: Cyclic prefix provides a guard interval to eliminate ISI and multipath immunity with the same manner as in OFDM. Pulse shaping is used to prevent spectral regrowth.
7. RFE: Converts digital signal to analog signal and thus converts the data to the radio frequency domain for transmission.

In summary, the 3GPP-LTE has gathered some interest in the SDR community. It has various bandwidths which can be applied for many different use cases. However, this also brings concerns about the transceiver. One of them is the capacity to support full bandwidth in real-time. Our work has considered the 3GPP-LTE as a case study within an SDR development flow. Our point of view mainly focuses on a flexible FFT design for full bandwidth 3GPP-LTE.

2.4.2. Fast Fourier Transform

The formula of DFT and IDFT were previously given in equations (2.9) and (2.10) respectively. DFT and IDFT are widely used in signal processing but require a lot of calculations. For an N-point DFT, a direct DFT calculation requires at least N^2 complex multiplications. This computation is therefore slow and requires many resources. Many efforts have been made in the past to reduce the number of multiplications in order to compute faster the same results, leading to FFT algorithms.

There are many strategies for FFT algorithms but one of the most popular algorithms is Cooley–Tukey FFT algorithm [32]. This algorithm factorizes an N-point DFT into smaller DFTs to reduce complexity. By this way, a length-N can be decomposed to prime factors and calculations are done with each of prime-factor DFT before computing the final results.

The most common Cooley–Tukey FFT algorithm is used for power-of-two FFTs which lengths are a power of two $N=2^M$. In the power-of-two FFT, the radix-2 FFT is widely used due to its simplicity. In this

FFT, the N-point DFT is converted to a set of 2-point DFT computations. Thus, a two-point DFT elementary block can be re-used many times allowing an architectural exploration between resources and computation speed. Consequently, we chose this algorithm for our implementation using a high-level synthesis based design flow.

Decimation in time Radix-2 FFT

There are two algorithms to implement power-of-two FFT: Decimation-In-Time (DIT) and Decimation-In-Frequency algorithms. In this section, DIT algorithm will be discussed. The DIT algorithm splits DFT into the even-numbered part and the odd-numbered part.

$$X(k) = DFT_N\big[[x(1), x(2), \dots, x(N-1)]\big] = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi nk}{N}}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\frac{j2\pi(2n)k}{N}} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-\frac{j2\pi(2n+1)k}{N}}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\frac{j2\pi nk}{\frac{N}{2}}} + e^{-\frac{j2\pi k}{N}} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-\frac{j2\pi nk}{\frac{N}{2}}}$$

$$= DFT_{\frac{N}{2}}\big[[x(0), x(2), \dots, x(N-2)]\big] + W_N^k DFT_{\frac{N}{2}}\big[[x(1), x(3), \dots, x(N-1)]\big] \qquad (2.12)$$

with $W_N^k = e^{-\frac{j2\pi k}{N}}$.

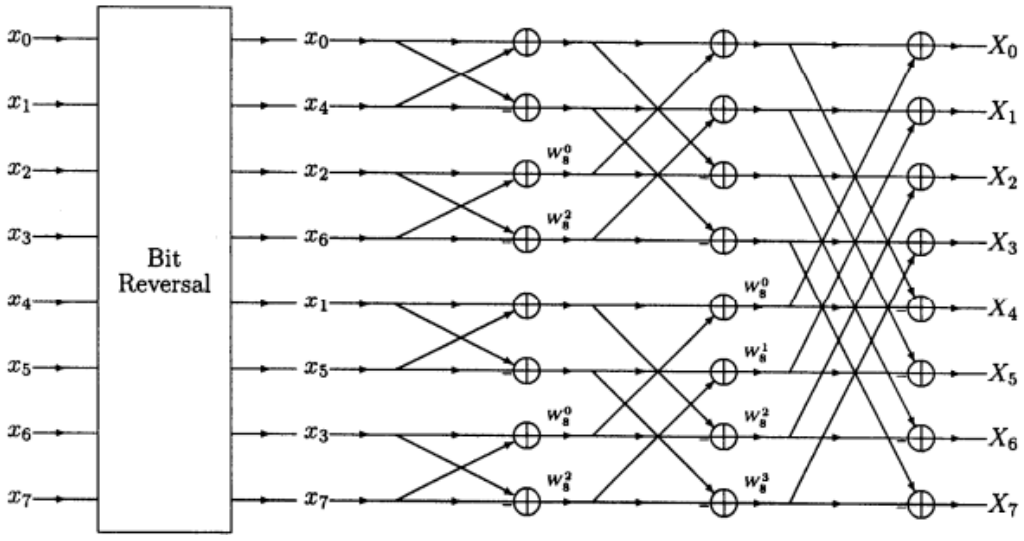


Figure 2.11 An eight-point DFT, divided in four two-point DFTs with bit reversal on inputs [33].

All DFT frequency outputs *X(k)* can therefore be calculated as the sum of the outputs of two N/2-point DFTs that process independently even-indexes and odd-indexes. With power-of-two DFT, this splitting process can be repeated until the length of DFT blocks is short enough for simple computing. In DIT

radix-2 technique, the length N is decomposed with a factor of 2. Once the DFT is fully split, the elementary block is a two-point FFT that can be used many times. Using this method, the order of the inputs needs to be rearranged before the calculation. The bit reversal block therefore re-orders the position of the inputs.

Figure 2.11 shows the full radix-DIT decomposed into $M=log_2(N)$ stages. Each stage uses N/2 two-point DFTs. Each two-point DFT requires 1 complex multiplication and 2 complex additions. The total cost of the algorithm is therefore:

- $\frac{N}{2} log_2(N)$ complex multiplications,
- $N log_2(N)$ complex additions.

This is a significant reduction compared to direct DFT computation.

2.5. Software –Defined Radio (SDR) Platforms

Choosing a computing platform for SDR terminal is an important decision. A specific hardware platform defines the computation capacity, efficiency of the system as well as several essential features of the system. Besides, the platform also decides the programing/development environment. For example, with a hardware platform using a processor, a designer can use C language to build the system. On the other hand, he may need to know VHDL or Verilog languages to design a system with a FPGA.

Because most platforms are heterogeneous, the classification and evaluation of them can be a tricky task. In this document, the platforms are introduced following four features: programmability, flexibility, energy consumption and computing power. The programmability stands for the capacity to modify the "software" in the radio system. The flexibility stands for the capacity to support various architecture designs.

There are many kinds of SDR platform [34] which have been proposed to implement SDR. Each of them has different advantages and disadvantages based on the approach it relies on. In this section, SDR platforms are divided into four main approaches:

- General Purpose Processor approach
- Co-processor approach
- Multi-processor approach
- FPGA approach

This section aims at providing an overview of different platforms used to implement SDR.

2.5.1. General Purpose Processor (GPP) Approach

A general Purpose Processor (GPP) is a typical microprocessor which is widely used in personal computer or smart phone. GPP provides a high flexibility and an easy development environment. A wide range of operating systems and qualified developers make GPP the easiest platform to develop SDR. However, GPP is only suitable for applications for which power consumption and computation capability

is not critical. In addition, the operating system running in GPP normally creates latency. Thus, GPP approach has high latency and is not suitable for hard real-time applications.

The USRP platform (Universal Software Radio Peripheral) [35] is a famous example of GPP approach in SDR. In this platform, the signal processing is performed mostly on GPP and a FPGA is used to store output data from the ADC. It is developed to work mainly with GNU Radio [35] but is also supported by Labview or Matlab.

SORA [36] is another example of GPP approach of Microsoft. In this platform, a PCIe (Peripheral Component Interconnect Express) bus is used to accelerate the connection between the platform and the computer. Thus, the latency is reduced and the data rate is increased. By this way, it is possible to decode the Wi-Fi 802.11 b/g in real time.

The ongoing development of microelectronics technology pushes to believe that future computers will be able to decode real-time protocols. However, because the data flow to be processed is increasing faster than the computing power [37], it is difficult to support advanced protocols using such platforms.

2.5.2. Co-processor Approach

Co-processor platforms are composed of a GPP and additional hardware resources to enhance computation capability. The additional resources are usually Advanced RISC Machines (ARMs), Digital Signal Processors (DSPs), FPGAs or a combination of them. They are used to perform heavy calculations of the applications. Unlike GPP platforms that have high latency, the high processing power of co-processors allows to run real-time applications. Such an approach can also be used to reduce the power consumption while trying to maintain a high flexibility and a simple programmability required by SDR platform. However, a combined architecture makes the programming model of this approach dependent of the hardware and reduces the flexibility.

The Kuar platform (Kansas University Agile Radio) [38] uses a GPP combined with a FPGA. To use this platform, the developer can design the application using GNURadio. When available in a predefined library of VHDL (VHSIC Hardware Description Language) IPs, a processing block can be run either on the GPP or on the FPGA.

Horrein, Hennebert and Pétrot [39] proposed a SDR platform that uses a Graphics Processing Unit (GPU) as a coprocessor for heavy computations. They use the GNURadio as a computing environment. The platform is divided between a host (GPP) and computing devices (GPU). The final system can gain three to four times the GPP computing power.

IMEC developed the ADRES platform (Architecture for Dynamically Reconfigurable Embedded Systems) [40]. It includes a central CPU and the ADRES accelerator. ADRES is a grid of 16 functional units, seen by the processor as a VLIW (Very Large Instruction Word) co-processor. This architecture helps the platform to process data in parallel and its compiler DRESC enables the design of reconfigurable 2D array processors making a powerful reconfigurable architecture. The ADRES platform targets both the 108 Mbps 802.11n and 18 Mbps LTE standards with an average consumption of 333 mW.

However the above architectures still have limited capability on parallel processing. This may reduce their computing power. The next approaches offer a better task parallelism.

2.5.3. Multiprocessor Approach

In this approach, a main processor is used for control. The other processors are dedicated processors for signal processing which can run in parallel. In some platforms, in order to reduce energy consumption, dedicated processors are only configurable units. Although the flexibility of this approach is reduced, it provides high programmability. This approach is presented in the Figure 2.12. In this figure, a group of processors which are connected to each other responds to the needs for signal processing tasks before sending the data to the Frond-End.

Tomahawk [41] developed by the University of Dresden is a platform for LTE and WiMAX. Two processors Tensilica RISC are used for control while six vector DSPs and two scalar DSPs are used for signal processing. The C language is used to program this platform. According to the authors' estimation, the power consumption of the platform is about 1.5 W.

The University of California at Davis develops smart ASAP (Asynchronous Array of Simple Processors) [42]. The objective of the project is to achieve signal processing computations on small processors within limited power budgets. All processors can communicate with their nearest neighbors on a grid. This platform offers a good compromise between programmability and performance. A 54 Mbps 802.11a/g standard was fully implemented in this platform with a consumption power of about 198 mW.

Magali [43] is a software radio platform developed by CEA LETI. This platform uses configurable units for dedicated processors. It is based on a Network on Chip (NoC) controlled by an ARM processor. The computation is performed by specialized DSPs and reconfigurable IPs for OFDM modulation. The chip runs the LTE protocol in 4x2 MIMO transmission mode with a power consumption of about 236 mW [44].

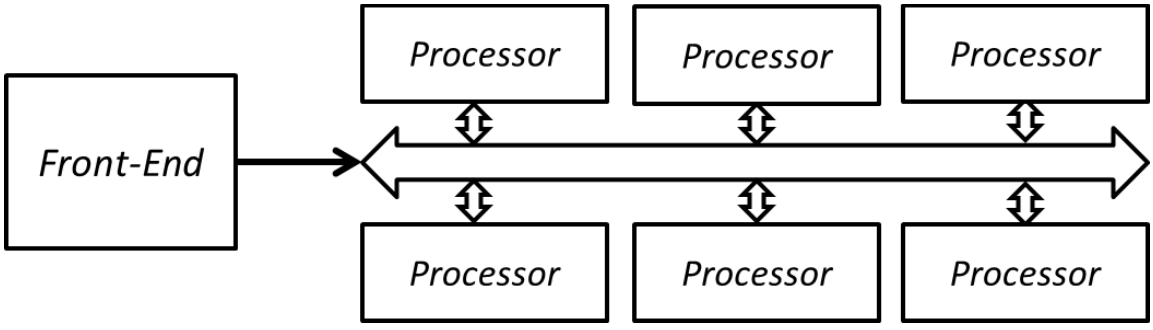


Figure 2.12 Multi-processor approach.

The ExpressMIMO platform [45] is built by EURECOM. It has configurable units which share a common network interface. The platform implements MIMO OFDM encoding method used in several protocols such as Wi-Fi and LTE. The open-source OpenAirInterface framework [46] is used for this platform.

2.5.4. FPGA Approach

The last type of platform relies on FGPA hardware which uses many configurable logic blocks to build the system. Unlike other approaches, the FGPA approach may not use processor(s). FPGAs make use of many low grain logic components, called Logic Blocks (LBs), and many interconnections as shown in Figure 2.13. LBs can be configured by the user for various purposes such as implementing combinational logic. Current FPGAs are also composed of macro blocks to support specific functions such as RAMs, DSP slices and interfaces to external devices. Because of their architecture allowing parallel processing, FPGAs have a high computation capability for moderate energy consumption. In addition, the capacity to implement and to run only the necessary parts makes FPGAs quite efficient in power saving mode.

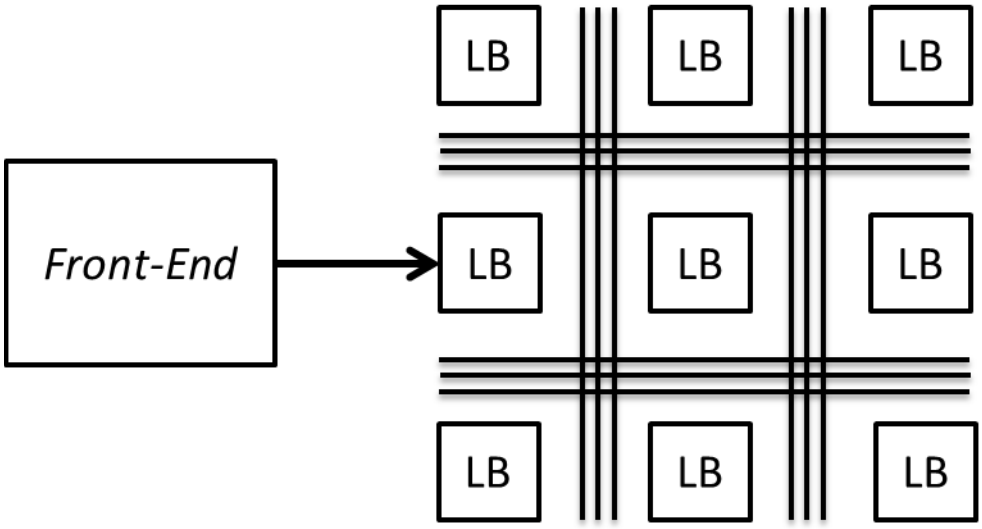


Figure 2.13 FPGA-based approach.

WARP [47] is an open SDR platform developed by Rice University. It provides a MIMO OFDM reference design and is used in many research projects. This platform is based on a Xilinx Virtex II Pro FPGA and is designed to be programmed using VHDL language.

The Nutaq company [48] provides various development tools and a software radio platform based on FPGAs. The development is carried out on Simulink or VHDL. These platforms can support MIMO WiMAX.

Rutgers University has developed WINC2R [49], a SDR platform built on a Xilinx Virtex 4 FPGA chip. It uses both softcore processors *i.e*. processor cores that are wholly implemented using LBs and dedicated hardware accelerators. Depending on the constraints, the processing can be balanced between softcore processors and accelerators. Unlike the previous approaches, with this platform the system can choose to use accelerators or not. An 802.11a waveform has been implemented on the platform.

The FPGA approach provides a flexible platform for prototyping using FPGA technology. It offers high computing power for moderate energy consumption. We will discuss in more details this approach on the next section.

	GPP	Co-processor	Multiprocessor	FPGA
Programmability	+ +	+ / -	+	- -
Flexibility	+ +	+ / -	-	+
Energy consumption	- -	-	+ / -	+ / -
Computing power	- -	+ / -	+ / -	+ +

Table 2.2 A comparative study between different approaches for SDR.

To sum up, Table 2.2 compares the four approaches based on the four criteria previously mentioned: programmability, flexibility, energy consumption and computing power. However, as previously noticed, most platforms are heterogeneous and a specific platform may need a detailed evaluation.

2.6. FPGA-based SDRs

Using FPGAs to implement SDR has many advantages. Due to its high computing power combined with parallel processing capability, FPGAs can support high data rates and consequently a wide variety of waveforms. Although it does not provide high programmability compared to other approaches, a FPGA is very flexible and suitable to support various architectures. Thus, FPGA-based SDR is expected to play an important role in SDR.

There are 3 ways to implement various waveforms on a FPGA:

- Multi-waveform configuration
- Separated configuration
- Partial reconfiguration

2.6.1. Multi-waveform Configuration

In this configuration, the bitstream (*i.e.* the file to configure the LBs and interconnection switches) keeps all the expected waveforms. According to the used waveform, all the features are set and configured by appropriate registers at runtime. This approach has the ability of instantaneously switching among configurations. However, because the bitstream is designed to support all the waveforms, it usually costs a lot of memory and resources. This solution becomes therefore impossible when a large number of waveforms are targeted. In most cases, the designer optimizes the code by grouping the common parts among waveforms and also tries to share common resources among waveforms but it costs time and can make the code significantly complex. One other disadvantage of this approach is the difficulty to upgrade or implement a new waveform that needs to make a fully new bitstream. Moreover, this approach normally achieves lower throughput because the bitstream has to support multiple waveform.

This approach is quite suitable with similar waveforms. Indeed, there are several related waveforms and the firmware needs only few changes to support other kind of waveforms. Massouri [50] developed a SDR a using this configuration on the Nutaq family of board Perseus 6011 (Virtex 6 FPGA). The options with different frequency bands of IEEE 802.15.4 have been accomplished on this platform. Zlydareva [51] develops a multi-standard wimaz/umts system using this approach too.

2.6.2. Separated Configurations

In this approach, each waveform is designed in a separated bitstream and is stored in an external memory as illustrated in Figure 2.14. According to the used waveform, the associated bitstream is loaded to the FPGA whenever it is required. This configuration does not need a large number of resources and easily suits various architectures. Moreover, it is very easy and fast if the designer needs to change or upgrade a specific waveform because every configuration is designed independently. In addition, only the required resources are configured and run for one waveform at a time. This saves energy and resources. However, unlike a GPP, it takes time for a FPGA to load a new configuration. It is a drawback of this approach. In practice, this approach is not often applied. Actually, current FPGAs can implement enough resources so that the multi-waveform configuration is preferred even if it is usually more complex to design.

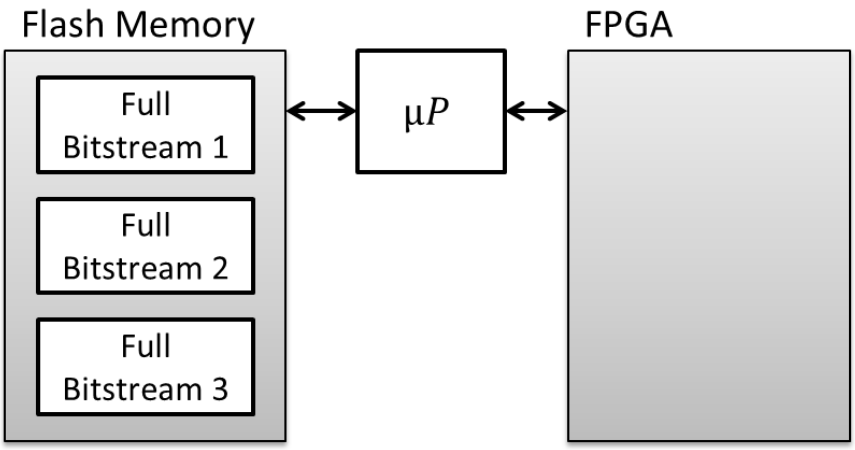


Figure 2.14 Separated configurations.

2.6.3. Partial Reconfiguration

An interesting feature of some FPGAs is partial reconfiguration (PR). Such FPGAs can load a new bitstream to only a small part of the hardware device while the other parts normally run. Although PR technique has been introduced for a long time, the associated design tools were quite limited and not very well supported by major FPGA manufacturers. Thus, PR was not easy to use and only few SDR platforms include this feature [52] [53].

When using PR, FPGA resources are split in several partitions:

- one or more reconfigurable partitions, where different configurations or different blocks of a new targeted waveform can be loaded at run time,
- one or more static partitions, where the processing are initially loaded, continuously run while reconfigurable partitions are modified in a PR process.

This method is presented in the Figure 2.15. In this method, a full bitstream, *i.e.* a bitstream which contains the configuration for a waveform, is first loaded. Whenever the firmware needs to change the waveform, it will load the partial bitstream which contains only the different configuration parts of a new waveform.

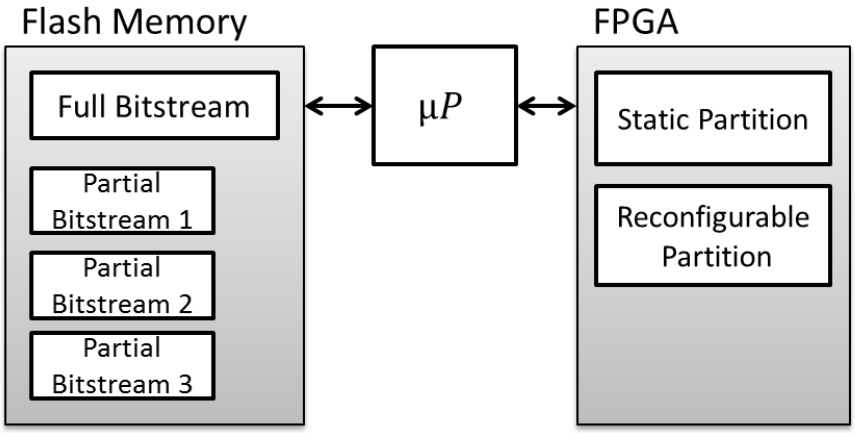


Figure 2.15 Partial reconfiguration.

Because the size of a partial bitstream is expected to be much smaller than the full bitstream, the loading time (time to reconfigure) is reduced and memory use is also usually reduced compared with the separated configuration based method. One very interesting point is the resources used in the FPGA at running time are kept minimum. In addition, this method is also quite simple for upgrading or adding a new waveform.

Recently, FPGA manufacturers increasingly noticed interest to PR. As a consequence, PR tools are more and more convenient to use and complete and PR currently seems to be an indispensable part in the most innovative FPGA Xilinx families. Thus, PR can be seen as a promising technology for SDR.

2.7. Conclusions

The increasing demand in high-speed connectivity both creates many different telecommunication standards and promotes various wireless technologies. In this chapter, after introducing digital radio and SDR which can afford high flexibility needs, we reviewed SDR system's main requirements. 3GPP LTE standard has been introduced as a case study for our work. Indeed this standard has six modes with different bandwidths. In each mode, the length of the FFT is different. Actually, flexibility is a major concern in such standard.

We also discussed the four basic approaches of SDR platforms. In each approach, there is a trade-off among four essential features: programmability, flexibility, energy consumption and computing power/throughput. The FPGA approach has some advantages in flexibility and computing power but it is also quite limited in programmability. That is the reason why in our research we want to use the advantages of HLS to improve the programmability in order to promote FPGA-based SDR.

In the next chapter, we will introduce the fundamentals of HLS and we will also provide an overview of several HLS tools. Hardware reconfiguration on FPGAs which can be of great importance for SDR will be also discussed.

Chapter 3.High Level Synthesis and Hardware Reconfiguration on FPGAs

3.1. Introduction

A SDR platform needs to be able to support various communication standards. It also needs to have powerful computation capability to support high demand of innovative waveforms. That is the reason why our research focuses on FPGA-based systems. A FPGA-based system can provide interesting computing power with moderate energy consumption. One minus point of a FPGA system is that it does not provide high programmability. Developing a FPGA-based system costs time and requires much knowledge in hardware. However, this issue may be solved by using HLS, which may help designers to build digital systems with limited effort. In this chapter, we first detail the basic principles of HLS as well as some popular HLS tools. The second part of this chapter provides knowledge about hardware reconfiguration on FPGAs, *i.e.* the important related terminologies and basics.

3.2. High Level Synthesis

In order to implement an application on a hardware target, most designs usually start with descriptive specifications. These can be simple texts or programs written in ANSI C, C++ or Matlab. These specifications essentially describe the function of the application or "what" the application does. The engineers, then, design the architecture of the application or "how" the application does and code it in a HDL language such as VHDL or Verilog for RTL. This transformation is not an easy task. It costs a lot of time and efforts. Moreover, because of implementation constraints, the final application can differ from the initial one. Thus, HLS [54] [55] has been developed to support engineer smoothly overcome the transformation from high-level specification to HDL with minimum errors and efforts. The idea of HLS came early from the years 1970s. At this time, HLS existed just in pioneer researches and had very little impact on industrial world. In the early generations of HLS design flows (1980s – early 2000s), HLS was still not used because of a lack of mature tools [56]. Recently, the new generation of HLS tools has obtained many attentions because of the significant improvements of the tools and performance of the generated applications. Indeed, HLS tools, now, play an important role to shorten the time from algorithm design to architecture design with limited efforts.

3.2.1. HLS Fundamental

HLS is a process which automatically generates RTL descriptions from high-level specifications. It helps engineers with minimum knowledge in hardware design to create quality RTL descriptions. A HLS process begins with the high-level specification, that is to say an algorithm written in an algorithmic fashion, written with some High-Level Languages (HLL) such as C/C++, Matlab or sometimes Domain-Specific Languages (DSL). This specification is first tested with a test-bench usually written in the same language before being parsed by the HLS tool. A typical HLS tool, then, creates the Control and Data Flow Graph (CDFG), an intermediate representation of the application. Based on design constraints and the target technology, all the operations of the intermediate representation are scheduled and mapped into hardware resources. This step is the most important step of a high-level synthesis and many techniques are used to optimize the design. Then the RTL description is generated and, finally, the RTL description is checked again with the test-bench for verification. A generic HLS design flow is shown in Figure 3.1.

High-Level Specifications for HLS

A traditional hardware design flow is divided into two separated steps. The first one is system modeling with specifications in a high-level language. These specifications served as a reference system. The designer can test and execute the reference system under various constraints. Test metrics in the telecommunication domain are usually BER or Packet Error Rate (PER). This work can be done by software designers having little knowledge in hardware design. Then, the second step is building the realistic hardware system. This step is done by the hardware designers. These two steps are separated and the specifications only provide a model for the hardware design.

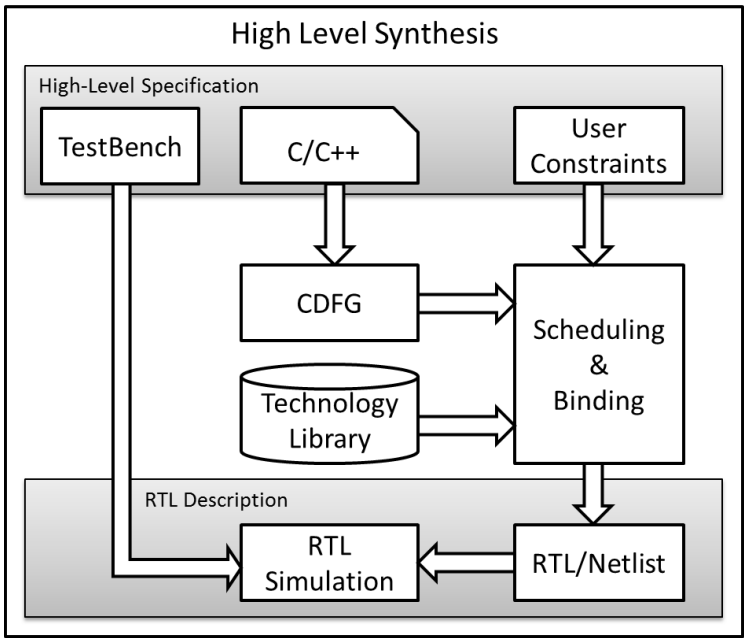


Figure 3.1 Generic HLS design flow.

On the other hand, HLS provides a direct solution from the specifications to the hardware description. The high-level specifications can be automatically converted to RTL descriptions. To do this, the specifications have to be written with certain (hardware related) rules. Indeed, software designs cannot be immediately turned into hardware design. All HLS tools have coding guides to include hardware considerations into the high-level specifications. The designers have to follow these guidelines to obtain well-generated RTL descriptions. However, these HLS instructions are tool-dependent. Most of the tools support high-level languages such as C, C++, SystemC or Matlab. The others have their own languages (DSL).

From Specifications to RTL Descriptions

From high-level specifications, a HLS tool, first, parses and converts them into CDFGs, the intermediate representation. A CDFG is a graph. It shows all the nodes, which represent operations, and their dependencies and it also shows the controls in the application. CDFGs are, then, fed to the scheduling

and binding processes. Scheduling step determines when an operation is executed. There are several scheduling algorithms. The basic ones are *As Soon As Possible* and *As Late As Possible*. Other algorithms are based on other strategies such as time-constrained or resource-constrained. Binding step determines on which operator an operation is mapped. It thus decides to share or not to share hardware for operations. The suitable scheduling and binding algorithms are automatically selected depending on the goal to optimize the performance or area. This is one of the greatest advantages of HLS. Indeed, the hardware designers normally need deep knowledge and experience to optimize the system to balance between the performance and used resources. However, theoretically, with HLS, the designers just need to focus on what they want (performance for example) to adjust directives and the HLS tool optimizes the design according to these directives. This interesting feature provides a great capability in optimization and design space exploration (DSE).

Co-simulation in HLS

After having generating RTL descriptions from high-level specifications, the HLS tool makes it possible to execute a last verification. The final RTL description (usually in VHDL or Verilog) is verified with the test-bench. The last verification confirms the final RTL design has the right properties and meet the constraints for the chosen technology target (ASIC, FPGA).

3.2.2. Advantages of HLS

HLS has ability to generate automatically RTL implementation from high-level specifications. The automation of the different design steps eliminates many errors and reduces time to market. In the following section, the main advantages of HLS are discussed.

Reducing design and verification efforts

Reducing design and verification efforts is the main advantage of HLS, sometimes referred to as a kind of rapid prototyping. Indeed, HLS reduces a lot of burdens for designers. Firstly, designers do not need the deep knowledge in hardware design to implement details. For example, in a hardware language, the designer must explicitly insert the clock cycles in the design. However, the clock cycles do not appear in specifications at HLL. In most of HLS tools, the user just chooses the clock frequency and then such clock statements will be automatically inserted into the generated component. Hence, the designer focusses on the behavior design only. Moreover, the HLL code is simpler to write. The fewer lines of code are written, the fewer errors occur. For example, a 1M-gate design requires less than 20% code lines using HLS tools [57] than using a RTL language.

Although the HLL code is shorter, the designer still takes control of the system. He can use directives to guide the tool to synthesize according to his desire. Indeed, the designer decides the value of the clock cycle, the level of parallelism and the constraints. The designer keeps engineering intervention while the HLS tools implements the designer commands in an efficient and productive way into RTL level. All of the details such as scheduling and resource allocation are decided by designer but are performed by the HLS tools. Moreover, the verification, usually a time-consuming process, is also automated reducing

another big burden for designers. Thus, HLS is a valuable tool for reducing design and verification efforts.

Optimization and DSE

HLS provides a great capacity in optimization and Design Space Exploration (DSE). While designing at RTL level takes a lot of time to optimize a dedicated hardware, HLS supports an easy way to optimize the system and shorten the time to market of a design. Indeed, HLS implements many techniques in order to generate the architecture which suits the requirements. For example, latency and throughput optimizations can be achieved through techniques such as pipelining or unrolling. The HLS tool can help the designer detecting the bottleneck and overcoming it. For example, with the FPGA technology, the designer can choose DSP blocks to accelerate the computation instead of using Look Up Tables (LUT). Random Access Memories (RAM) or Read Only Memories can also be easily selected. It may take time and it usually requires code-change in RTL design but it is fast and easy to do using HLS.

More effective reuse

One of another benefit of HLS is the ability of effective reuse. For years, RTL designers have difficulty reusing RTL code. Indeed, most RTL designs are optimized for specific constraints and technology target with dedicated processes and state machine for example. Because the RTL descriptions are targeted to specific technology and clock frequency, even a small change can cause many bugs and unexpected problems. With HLS, on the other hand, it is much more easy. High level of abstraction brings a generic and versatile architecture. There are no details such as clocks and technology in specifications written in HLL. These kinds of details are just information the designer has to provide before starting the HLS process.

3.2.3. HLS Tools

The history of HLS began in the 1970s. In the early time, most HLS tools targeted ASIC designs and were only used for research purposes. Until the early 1990s, HLS tools were not mature yet because of lack of attention from the designers. Indeed, most HLS tools provided poor results and it was hard to validate the results. Therefore, HLS research in this time was difficult to transfer to the market. However, since 2000, many CAD vendors offer high-quality HLS tools such as Mentor Catapult C Synthesis, Forte Cynthesizer, Celoxica Agility compiler, Bluespec, Synfora PICO Express and Extreme, ChipVision PowerOpt, NEC CyberWorkBench, AutoESL AutoPilot [56]. They have created a new generation of HLS tools which were much more mature and which provide good quality results. This section provides a short overview of various HLS tools. Because most of our proposed work relies on Vivado HLS tool, some optimizing techniques available in this tool are also presented.

There are several approaches for HLS and related design flows, leading to various types of HLS tools. They can be divided into two main categories: the ones that use DSLs and the ones that use General-purpose Programmable Languages (GPLs) [58]. DSL-based tools use a dedicated language which is specially designed for parsing and converting specifications to HLS. In that case, the designer needs to study the syntax of the language but this language is expected to provide a better support for HLS

process (actually, this kind of tools are usually dedicated to a specific application domain and implement optimization techniques accordingly). On the other hand, the GPL-based tools provide a familiar language along with directives and directions to use for HLS purpose.

Because some HLS tools are not supported anymore, this section only focuses on several HLS tools being in use. We divide them into academic tools and commercial tools. Some general information about the tools is provided in Table 3.1.

Academic tools: These tools are mostly developed by Universities. They are easy to access and generally free. However, due to limit in budget and resources, these tools are usually not fully mature and have limited support.

- **Bambu** [59] is an HLS tool released in 2012. It is developed at the Politecnico di Milano. It has the option to adjust implementation to balance between latency and resource requirements. It supports complex construct of C language such as pointers, multidimensional array as well as floating point arithmetic. It is a quite complete HLS tool and supports most common simulators.
- **DWARV** [60] is an HLS compiler developed at the Delft University of Technology. This compiler is constructed on the CoSy compiler, the highly flexible compiler development system from ACE Associated Compiler Experts. DWARV provides the basic features of an HLS tool and inherits from advanced features of using CoSy infrastructure such as the ability to easily exploit standard and custom optimizations.
- **LegUp** [61] is a research compiler developed at the University of Toronto. It first realized in 2011 and introduced several unique techniques. Indeed, most of C language features are supported in LegUp and it also supports Pthreads, a parallel execution model and OpenMP, a parallel execution model to support multithreading. Thus, it is possible to synthesize a parallel software thread to parallel-operating hardware.
- **GAUT** [62] is an HLS tool developed by the University de Bretagne-Sud. It targets digital signal processing applications. It supports C/C++ specifications and has ability to automatically verify RTL results for validation purpose. GAUT mostly targets data-path design and supports many optimization techniques for loops with constant boundaries.

Commercial tools:

- **Vivado HLS** [63]**,** formerly AutoPilot, was firstly developed by AutoESL. Xilinx acquired AutoESL in 2011 and developed a tool based on AutoPilot with a lot of additional features. A whole new product was released in 2013 named Vivado HLS. Vivado HLS is based on a low-level virtual machine compiler framework. It provides a friendly environment for designer with sufficient features for a mature HLS tool. It supports C, C++ and SystemC languages for high-level specifications and can generate hardware modules in VHDL, Verilog and SystemC. Moreover, it also proposes many different optimizing techniques. As most of our proposed work relies on this tool, it will be introduced with more details in the next subsection.

- **Cyber-WorkBench** [64] is a set of tools for HLS design developed by NEC Company. All of these tools work together to create a fulfill environment for HLS from synthesis to verification. It provides therefore a C-based design environment to target FPGA or ASIC components.
- **Catapult-C** [65] was firstly introduced in 2004 by Mentor Graphics before being acquired by Calypto Design Systems. Catapult-C has some interesting techniques for power optimization by automating two design techniques: multi-level clock gating and interfacing to dynamic power and clock management units. This tool firstly targeted ASIC component but is now supporting FPGA targets. It supports almost C/C++ and SystemC as high-level description.
- **DK Design Suite** is a HLS tool also developed by Mentor Graphics. This tool targets C and C++ designers who do not have in-depth hardware design tool experience to utilize the parallel logic in FPGA prototyping hardware. It uses Handel-C as a design language based on a subset of C and extended with hardware-specific language constructs. However, in Handel-C language, the designer needs to manually perform data mapping to memory. The designer also has to specify timing requirements.

Compiler	Owner	Input	Output	Year	Test-Bench	Floating Point	Fixed Point
DWARV	Deft UT.	C subset	VHDL	2012	Yes	Yes	Yes
Bambu	Poli. Milano	C	Verilog	2012	Yes	Yes	No
LegUp	U. Toronto	C	Verilog	2011	Yes	Yes	No
GAUT	U. Bretagne Sud	C/C++	VHDL	2010	Yes	No	Yes
Vivado HLS	Xilinx	C/C++ SystemC	VHDL/Verilog SystemC	2013	Yes	Yes	Yes
Cyber-WorkBench	Nec	BDL	VHDL/Verilog	2011	Cycle/ Formal	Yes	Yes
Catapult-C	Calyto Design System	C/C++ SystemC	VHDL/Verilog SystemC	2004	Yes	No	Yes
DK Design Suite	Mentor Graphic	Handel-C	VHDL/Verilog	2009	No	Yes	No

Table 3.1 General information about HLS tools.

Vivado HLS tool

Vivado HLS tool is a complete HLS tool integrated in Xilinx's CAD framework (Vivado Design). It provides many options and features to control and optimize the synthesized design. This section discusses some typical features of Vivado HLS. Figure 3.2 shows Vivado HLS design flow. The system is built on C/C++ or SystemC and is first checked at this high abstraction level. Then, combining constraints and directives, Vivado HLS synthesizes to VHDL, Verilog or/and SystemC languages. The results are then checked for a second time with test-benches based on a RTL simulation. The final results can be packed in a IP core to be used with the other tools.

Vivado HLS interface: Vivado HLS provides command line interface with Tcl commands. It is powerful but not very simple to use. With the version we use (2017.3), Vivado does not support the automatic

makefile generation. Every command in Vivado HLS can also be started in Vivado HLS graphical user interface. In the graphical user interface, the user can also see and analyze the performance of the implementation.

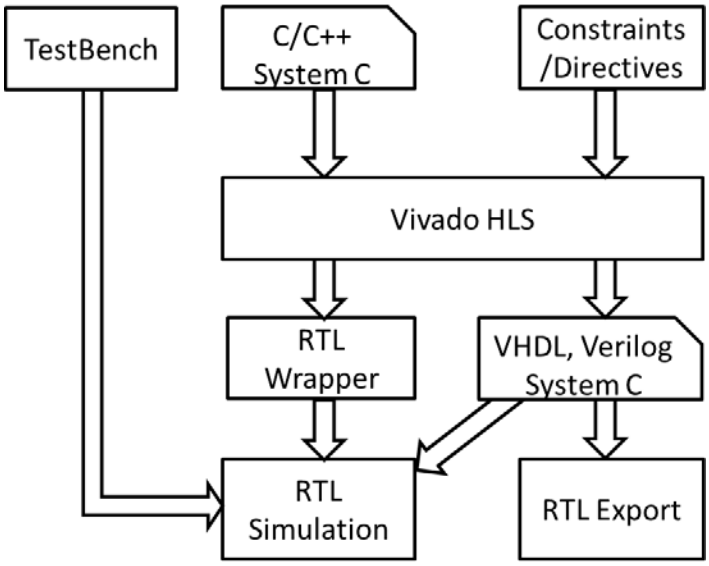


Figure 3.2 The Vivado HLS design flow.

Optimization techniques with Vivado HLS: Vivado HLS offers many types of optimizations. One single specification can create several RTL architectures with various performance and area depending on the type of optimization. These techniques can be used by the designer according to the design requirements. Most useful techniques operate on commonly used instructions of the high-level description that are the functions, the loops and the arrays.

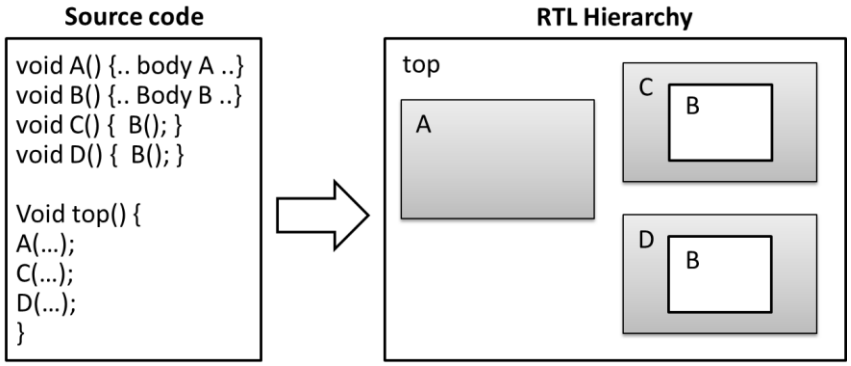


Figure 3.3 RTL hierarchy after HLS synthesis.

- **Optimization techniques for functions**: Vivado HLS provides “inline”, “dataflow” and “pipelining” directives for functions. Default Vivado HLS configuration synthesizes the source code to RTL one in the same hierarchy. Each function is implemented into an independent RTL block as illustrated in Figure 3.3**.** However, the tool provides “inline” option to remove function hierarchy to get better optimization without boundaries. By this way, the tool has more options

to optimize the resource usage. For example, if the tool detects an unconnected port, it can eliminate that port and reduce the resource usage. The “dataflow” directive is a directive for a top function**.** It optimizes the dataflow among functions to get better throughput. Figure 3.4 is an illustration of how “dataflow” directive works. For a program with functions sequentially executed, the “dataflow” directive exploits parallelism if these functions do not share same variables. The “pipelining” directive in function level arranges consecutive operations in serial to parallel to get higher throughput. Indeed, an operation is executed in serial by default. If the “pipelining” directive is used, the tool will try to cut an operation to smaller units and execute them in parallel. This directive is illustrated in Figure 3.5.

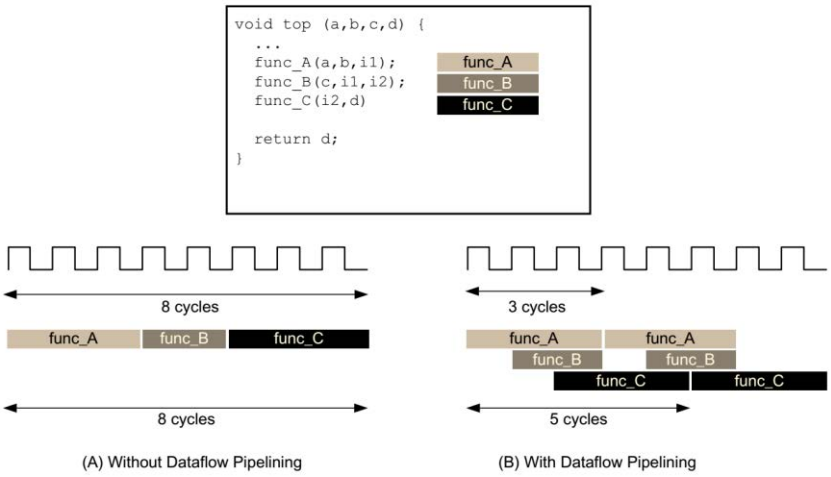


Figure 3.4 Dataflow optimization [66].

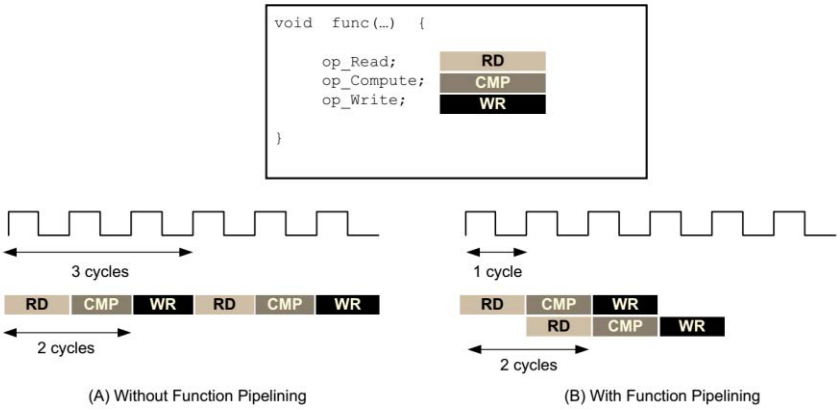


Figure 3.5 Pipeline optimization [66].

- **Optimization techniques for loops:** Vivado HLS provides “unrolled”, “flattening”, “merging” and “pipelining” directives for loops. By default, loops are rolled in Vivado HLS but they can be “unrolled” to reduce latency while using more resources. An example is shown in Figure 3.6. To balance this, Vivado HLS provides the ability to partially unroll a loop so that the designer can exactly decide the number of operations that run in parallel. ”Flattening” directive converts

inner loops into outer loop and “merging” directive combines several loops in a function into single loop without changing the source code. ”Pipelining” directive first totally unrolls the loop if possible and then applies pipeline directive.

- **Optimization techniques for arrays**: Arrays after synthesis are typically stored in a memory. This can create speed bottlenecks when the number of memory ports limits the access to the data. Thus, “array partition” directive allows the designer to split an array into smaller blocks or individual elements to have a faster access to the data. On the other hand, “array reshaping” combines different arrays into a single one.

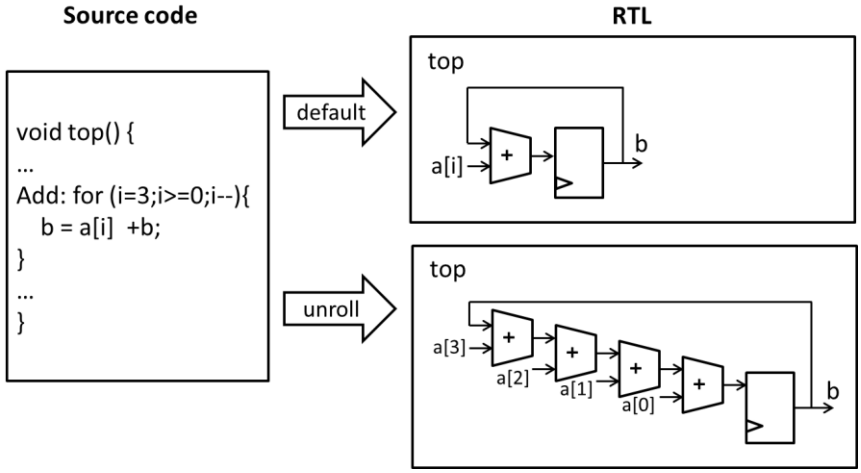


Figure 3.6 Unrolled optimization example.

I/O interfaces: Vivado HLS provides several options for I/O interfaces. Besides the basic ports automatically added such as clock and reset, it also provides protocols at both block level and port level. At block level, it has a handshake protocol. In this protocol, the data port is combined with a valid port and an acknowledge port. The valid port indicates the data is valid for reading or writing and the acknowledge port notices the data has been read or written. At port level, multiple interface protocols are available such as no I/O protocol, wire handshake protocol, memory protocol and bus protocol. Moreover, it is also possible to add adapters to the RTL and create a bus interface. The bus interface supports for Advanced Extensible Interface (AXI) [63].

Controlling the resources: Controlling the resources in Vivado HLS is quite convenient. The designers can totally control the binding process using directives. They can exactly choose the specific resources for an operation. For example, they can choose if a multiplication is implemented in a DSP slices or not. They can also choose to save the data in ROM, RAM or look-up table. By this way, the resources can be specifically managed.

There are some other features of Vivado HLS which are not mentioned in this part. More details on Vivado HLS can be found in [66].

3.3. Hardware reconfiguration on FPGAs

3.3.1. Introduction

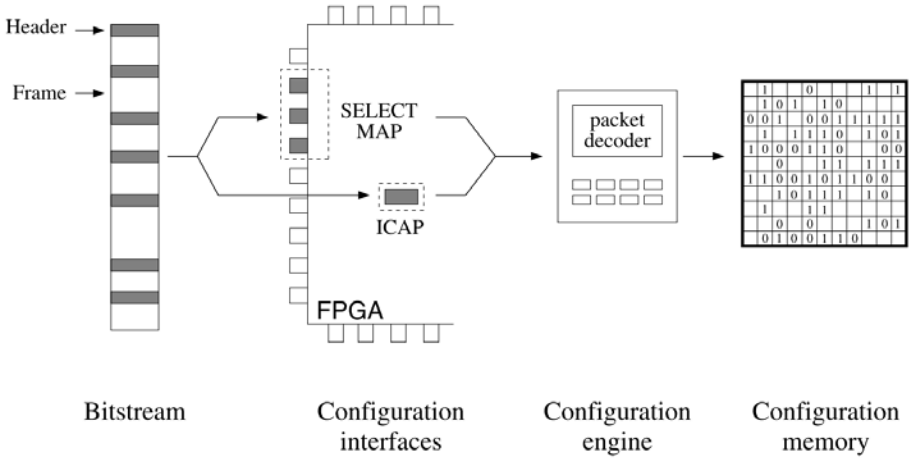
The smallest available unit in a conventional FPGA is the LUT (Loo-Up Table). An n-bit LUT is an n-input Boolean function provided by storing all the output values in the LUT. Most of LUT in FPGAs now have four to six bit inputs. According to Xilinx terminology[1], a slice is a group of LUTs, storage elements, carry logics and multiplexers. In the higher hierarchy, Configurable Logic Blocks (CLB) contains several slices. In FPGA Virtex 7 series for example, a CLB has four slices. Each slice has four 6-bit input LUTs. FPGAs are organized by arrays of CLBs.

In a conventional design flow, the designer writes the program using a hardware description language such as VHDL or Verilog. The compiler makes the logic synthesis so that the netlist can be generated. The netlist is the description of used electronic components (*e.g.* logic gates) and their connectivity. The netlist is verified and optimized according to the FPGA target and then the compiler creates a binary file named bitstream which is used to configure the specific FPGA. A FPGA needs to load the full bitstream for configuration before running.

Current FPGAs implement reconfiguration capabilities and this reconfiguration can be performed in two manners: full reconfiguration and partial reconfiguration (PR).

The process for full reconfiguration is similar to the configuration with a full bitstream before running. In full reconfiguration, all CLB configurations are replaced by new ones while, in partial reconfiguration, only a part of the reconfiguration in a specific region is replaced by a new one. A drawback of full reconfiguration is its long reconfiguration time. When a full reconfiguration is performed, the FPGA has to stop its execution to load the new bitstream file that contains the new configuration of all the FPGA components. In contrast, the PR allows the reconfiguration only of a specific region. In this case, only the specific region has to stop its execution, the other part of the FPGA still can normally work. The bitstream size of a specific region is smaller than the full bitstream one, reducing the loading time. For instance, Xilinx provides PR capabilities in its Virtex series. Most concepts introduced in this document are related to Virtex series as it is the technology we target.

[1] Altera terminology is not exactly the same, however meanings are very similar



[67]

Figure 3.7 The reconfiguration chain for a Virtex FPGA.

The reconfiguration chain is illustrated in Figure 3.7. At the beginning of the chain, the bitstream is sent to a configuration port. The bitstream is organized in packets which contain headers and payloads. It is, then, targeted to the right place before being written in the configuration memory of the FPGA.

Some important terminologies about FPGA reconfiguration are described below.

Configuration frames: Configuration frame is the smallest addressable unit in FPGA memory space. A configuration frame can include all reconfigurable logic elements in a column and then has one address dimension. One can notice that the size of a configuration frame has become recently smaller and these frames may have two address dimensions. Small configuration frame are more efficient in reconfiguration. An example is shown in Figure 3.8. The black rectangle is a configuration memory. The figure in the left is based on one address dimension and wastes unused area for nothing. On the other hand, the right figure is based on two address dimensions which provide better area efficiency.

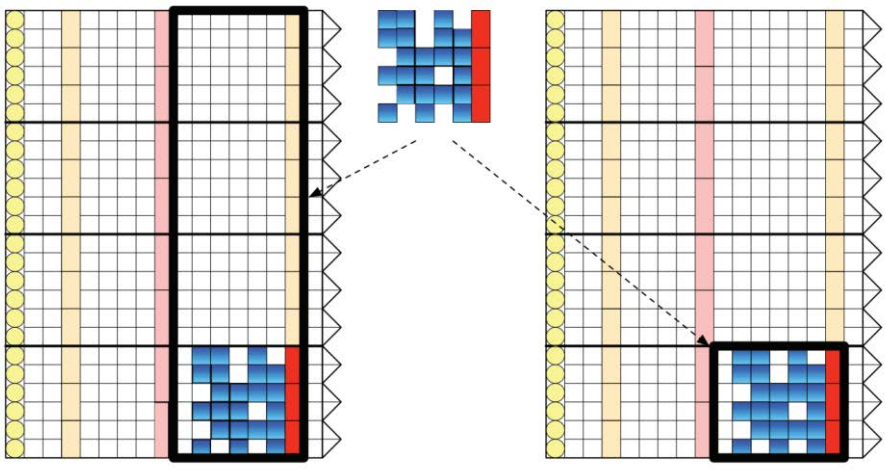


Figure 3.8 Configuration uses 1 address dimension (left) and 2 address dimensions (right) [68].

Configuration memory: A configuration memory is an array of configuration frames. A full configuration memory defines every component in the FPGA from the LUT, memory blocks as well as DSP blocks, IO controller and all other aspects of the FPGA (*i.e.* interconnections, …). Configuration memory has two types: 1D-array reconfiguration and 2D-array reconfiguration depending on the type of the configuration frame.

Bitstream: A bitstream is a binary file used to configure the FPGA. The full bitstream is used to configure all the FPGA for full configuration. In contrast, a partial bitstream keeps information to configure only a specific region of the FPGA for partial configuration. The bitstream is organized in packet with a header and a payload. The header contains information for the target address of the FPGA while the payload contains the configuration frames.

Configuration port: The configuration port is a dedicated FPGA port used to transfer a bitstream to the configuration memory during the configuration process. This port could be internal or external of the FPGA and the bitstream is written following a specific protocol to use the specific configuration port such as JTAG or Slave Serial. The internal port used in Xilinx FPGA is called ICAP for Internal Configuration Access Port.

Configuration chain: The configuration chain is the configuration process to configure a FPGA. It is similar to the reconfiguration chain in Figure 3.7 except the configuration for all the FPGA.

Configuration time: The configuration time is the time to perform the configuration chain. It depends on the bitstream size and the data rate of the configuration port.

Reconfiguration controller: The reconfiguration controller is the device which controls the configuration chain. It can be external or internal of the FPGA. The type of reconfiguration controller does not affect the configuration time.

Communication link: The communication link is an interconnection between a reconfigured region and a fixed one. This link ensures that the data connection between the reconfigured part and the fixed one (static region) correctly works when different reconfigurations are re-placed and re-mapped in the FPGA. There are several solutions for this issue. In Xilinx partial reconfiguration design, in previous versions, this link was performed using Bus Macro but it is no longer used. For now, there are special components named Partition Pins. They are communication links between static region and reconfigurable region. They guarantee that the static logic can connect well with all different versions of a reconfigurable partition. It leads to a fact that all configurations of a reconfigurable partition have similar pins (ports). In other words, the architecture inside the reconfigurable partition can be changed but it must have similar inputs and outputs to ensure the good connection with the others partitions.

3.3.2. Dynamic Partial Reconfiguration

3.3.2.1. Managing Dynamic Device Reconfiguration

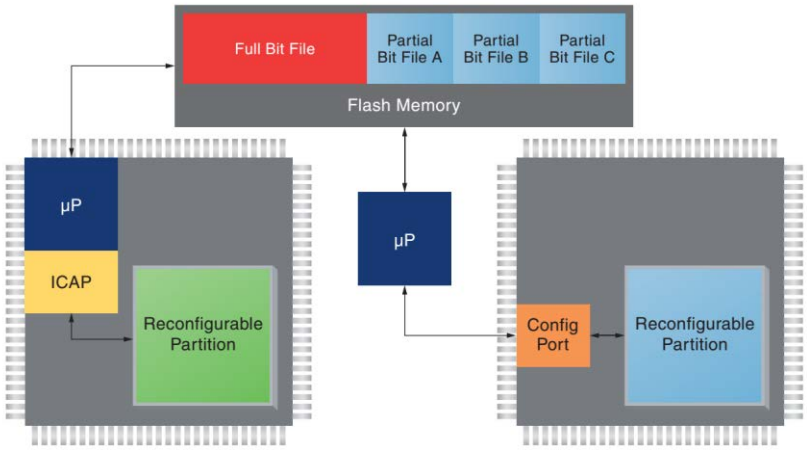


Figure 3.9 Two methods of delivering a partial bit file (with Xilinx devices) [69].

To configure a FPGA, a full bitstream is loaded onto the FPGA. Once initially fully configured, the FPGA can be fully and/or partially reconfigured any time the reconfiguration controller requests a new configuration. With DPR (Dynamic Partial Reconfiguration), reconfigurable partitions are reconfigured while the other partitions remain fully active without interrupting. The designer can choose the reconfiguration controller method to deliver the partial bitstream as illustrated in Figure 3.9. The first method is self-reconfigurable FPGA. A microprocessor inside the FPGA is used to control the partial reconfiguration process. This microprocessor reads the partial bitstream from a flash external memory and sends it to the internal configuration port (ICAP for Xilinx devices). In some cases, the microprocessor is not necessary and a simple state machine can manage the operation. If the system has a microprocessor outside the FPGA, it can be used to control the partial reconfiguration process as shown on the right of the Figure 3.9.

Table 3.2 shows the maximum bandwidths for configuration ports in Virtex architectures. There are four configuration modes for Xilinx FPGAs. They are ICAP, SelectMap, Serial and JTAG. The designer can choose the suitable port to load the partial bitstream depending on designer's purposes. The ICAP and SelectMAP supports the highest bandwidth.

Configuration Mode	Max Clock Rate	Data Width	Maximum Bandwidth
ICAP	100 MHz	32 bits	3.2 Gbps
SelectMAP	100 MHz	32 bits	3.2 Gbps
Serial Mode	100 MHZ	1 bit	100 Mbps
JTAG	66 MHz	1 bit	66 Mbps

Table 3.2 Maximum bandwidths for configuration ports with Virtex architectures.

3.3.2.2. Design Flow for Xilinx FPGAs

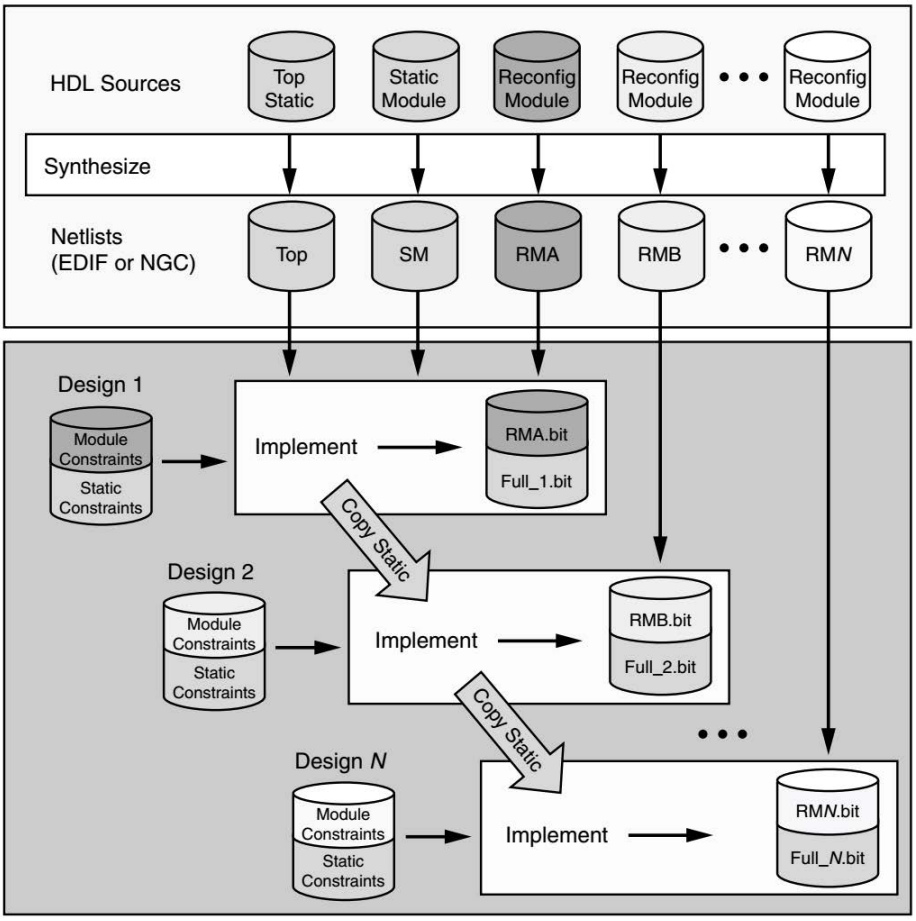


Figure 3.10 Partial reconfiguration software flow [66].

The partial reconfiguration design flow for a Xilinx device is illustrated in Figure 3.10 The top gray box represents the synthesis of HDL sources to netlists for each module. Modules are independently synthesized in the bottom up synthesis way to make sure the integrity of them. Indeed, although the static modules can be synthesized together, each reconfigurable module needs to be synthesized independently. The reconfigurable partitions are presented as black boxes because they have no configuration inside at this step[2]. For each design, the netlists of reconfigurable modules are placed into the black boxes and tested with the constraints. This makes sure that all reconfigurable modules are compatible with static modules.

3.3.2.3. Main interests of Dynamic Partial Reconfiguration

Reduce number of resources: Historically, designers spent a lot of time to optimize the implementation so that it can fit into the smallest possible FPGA. Nowadays, DPR helps the designer to reduce dramatically the hardware used resources by time-multiplexing portion technique. This strategy is

[2] In the same way, the implementations of top and static modules are black boxes when dealing with reconfigurable partitions.

especially useful in SDR where it is required to support various waveforms but only a single waveform is used at a time.

Increase system flexibility: DPR makes the system more flexible to be upgraded. Most often in such a case, only a particular field is required to be reconfigured, the remainder of the design still works normally and all the system can run without interrupting. This feature is vital for many run-time systems.

Reduce power consumption: Besides the FPGA size and cost, power consumption is also a primary concern of hardware designers and using DPR is one of the solutions to reduce FPGA power consumption. Indeed, DPR can help reducing both static and dynamic power. The simple way to reduce power is to use the smaller device. By using DPR, the design can be smaller and static power requirement is less. Moreover, the designers can swap the configuration depending on the performance of the system to optimize the power consumption. For example, the design sometimes must be able to run at maximum performance in a small percent of time. During some other times, the design can run at - let say- normal performance or low performance. Thus, several versions of the system can co-existed to support the system saving dynamic power according to the different performances.

Some other advantages: There are also many other advantages offered by DPR based on the ability to time-multiplex hardware such as the reduction of the overall bitstream size/memory, the acceleration of the time to (re)configure, the providing of real-time flexibility protocols [69], …

3.4. Conclusions

HLS provides a fast and convenient solution for hardware system design. Indeed, it helps designers not only reducing the design time but also exploring design space. Along with the maturity of current HLS tools, HLS plays a more and more important role in hardware system design. Furthermore, DPR offers a powerful solution to extend the capability of FPGAs. It not only enables reducing the size, power and cost of FPGAs but also enables new types of FPGA-based designs. DPR requires designers to create various RTL versions of the reconfigurable partitions and it is required they have the same partition pins. However, thanks to HLS tools with powerful DSE capacity, the designer can quickly create various RTL versions of the reconfigurable partitions. In the next chapter, we will discuss about a specific design flow based on HLS to develop DPR-based systems.

Part II - CONTRIBUTION

Chapter 4.Design Flow for Flexible Radio on FPGA-based SDRs

4.1. Introduction

Several proposals attempted to meet the flexibility requirements of SDR by using software-based approaches. Indeed, software-based approaches provide an abstraction level that enables more control than hardware-based ones. Two complementary approaches have been proposed, namely the SDR-specific languages to design the waveform [70] [71] and the SDR middleware to provide the design environment [72] [73]. Both approaches exploit the abstraction level given by the software to achieve both compile-time and run-time flexibility.

As we previously noticed, among SDR platforms, FPGA-based ones seem to be a good candidate. Indeed, a FPGA platform, as shown in Table 2.2, provides not only flexibility but also computing power for run-time processing.

Our research aims at keeping specifications at high-level while addressing FPGA platforms. To this end, we leverage HLS to achieve such a high abstraction level. The recent availability of mature HLS tools allows the consideration of components described in C/C++ languages. It raises the abstraction level compared to hardware languages like VHDL and Verilog usually used for RTL-based design. While compile-time flexibility has already been studied [74], this work focuses on run-time hardware reconfiguration of a flexible waveform from its high-level description.

There are different ways to achieve a flexible processing block while implementing it onto a FPGA. The first one is to design a multi-mode processing block and the second one is based on DPR. The first one rely on the design of multi-mode RTL components with control signals to switch between the different modes [75]. The other one is run-time DPR, referred to as *Hardware reconfiguration* in the following of this report, has the ability to reconfigure part(s) of the FPGA (*e.g.* functionality at the hardware level) while the rest of the FPGA continues to work. It has been a research topic since the 90s [76] and it can be now used in FPGAs, since Xilinx and Altera provide such functionality in their circuits [69] [77]. The hardware reconfiguration takes advantage of having hardware flexibility as well as reusing hardware area. In addition, it can be used to reduce power consumption and device cost.

In our approach, a multi-mode processing block can be described using dedicated algorithmic modifications of the processing block (referred to as *Algorithmic reconfiguration*) or with an automatic generation using a HLS encapsulation (referred to as *Software reconfiguration*). The goal of our design flow is to choose or combine these two reconfiguration ways while describing the processing block at a high-level of description. This work is based on one commercially available HLS tool: Vivado HLS from Xilinx. It generates a RTL description of an application from its C-like specification. This section details the ways towards the generation of a flexible processing block.

4.2. Reconfiguration Methods for Software Defined Reconfiguration

4.2.1. Software Reconfiguration

This reconfiguration uses HLS encapsulation to generate a `MULTI_MODE_BLOCK`. This method uses the different modes of a block and generates a `MULTI_MODE_BLOCK` with a control input to switch

between the modes. Algorithm 1 describes this encapsulation in the case of two modes BLOCK_A and BLOCK_B.

The advantages of this method are its simplicity, the rapid prototyping capability provided by HLS and the short reconfiguration time (one clock cycle). However, the resources can be important in that case. Actually, the Vivado HLS tool does not efficiently share the resources even when the modes are quite similar.

```
1. function  MULTI_MODE_BLOCK(inputs, outputs, control)
2.   switch control do
3.     case A
4.       BLOCK_A(inputs, outputs)
5.     case B
6.       BLOCK_B(inputs, outputs)
7. end function
```

Algorithm 1 Software reconfiguration for the automatic generation of a multi-mode processing block.

4.2.2. Hardware Reconfiguration

In DPR, the FPGA is divided into several regions being static (the areas that are not modified) or reconfigurable (the area that can be reconfigured) [66]. Each reconfigurable partition can configure for one function with all modes. Each mode has its own partial bitstream. As introduced in Section 3.3, the partial bitstreams are stored in a flash memory and a software processor controls which partial bitstream is loaded into the reconfigurable partition at a particular time. The reconfigurable partition size must cover the area of the largest mode. Figure 4.1 shows the example of the hardware reconfiguration of two modes BLOCK_A and BLOCK_B. The two modes are processed separately using first HLS and then RTL synthesis to generate two partial bitstreams. Parts of this flow can be automated, at least to have an estimation of the performance. The main advantage of this approach is that the modes share the same area. The drawback is the reconfiguration time that depends on the size of the partial bitstream.

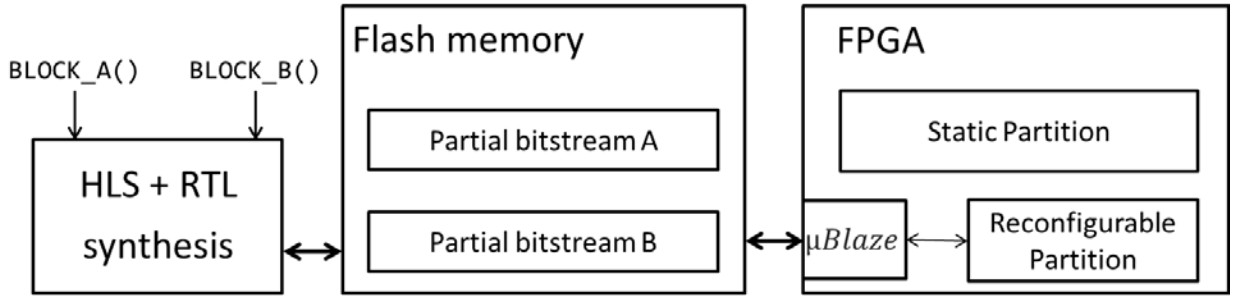


Figure 4.1 Design approach based on hardware reconfiguration.

4.2.3. Algorithmic Reconfiguration

For this kind of reconfiguration, the designer has to hand-code a dedicated processing block being intrinsically flexible. Signals are used to control the modes. Algorithmic optimizations should be done so that the HLS tool can share the resources between the modes. This approach is much more tricky as it relies on the designer's ability.

Figure 4.2 shows the design tradeoff between the resources and the reconfiguration time for the three kinds of reconfigurations. Algorithmic reconfiguration is interesting to decrease the resources compared to software reconfiguration and to decrease the reconfiguration time compared to hardware reconfiguration. It provides the best performance in term of resources/reconfiguration time tradeoff. However, depending on the processing blocks, time to code the algorithmic reconfiguration can be important compared to Software reconfiguration. Moreover, algorithmic optimizations are not always possible and may not be efficient in every case (they may lead to the same performance as software reconfiguration).

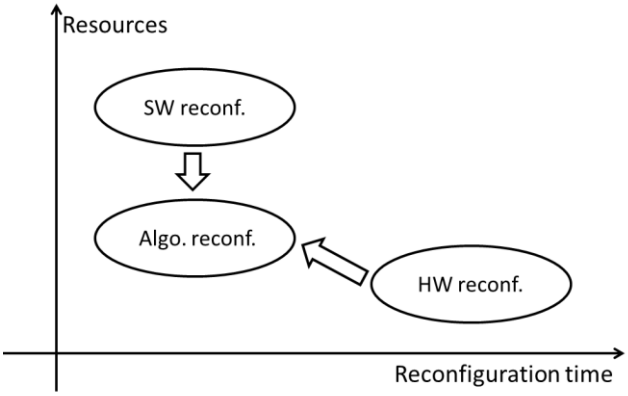


Figure 4.2 Tradeoff between resources and reconfiguration time for the different reconfigurations.

4.3. Proposed Design Flow and System Architecture for Flexible Radio on FPGAs

Based on these three kinds of reconfiguration, the HLS design flow for Software Defined Reconfiguration used in this work is shown in Figure 4.3. This design flow describes the process to design a reconfigurable module. The different modes of a processing block can be provided by hand-coding or using a HLS tool to generate different versions of a processing block by modifying synthesis constraints like throughput, latency, data size, etc. HLS tools make it easy to explore a set of solutions via DSE.

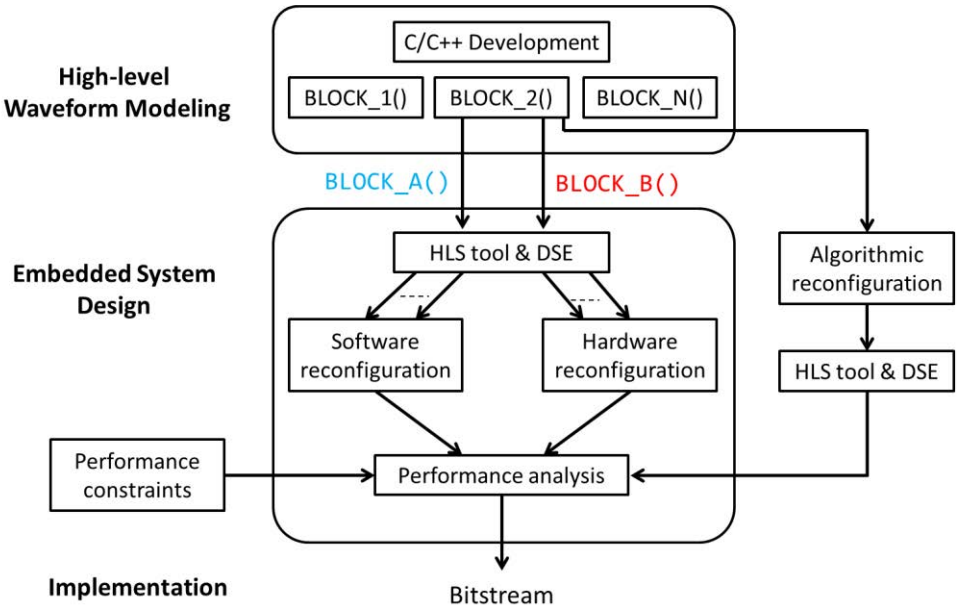


Figure 4.3 HLS-based design flow for a reconfigurable module using Software-Defined Reconfiguration.

In Figure 4.3, performance constraints are user-defined constraints such as resources/area, reconfiguration time, throughput or latency. The performance analysis compares the performance of the different kinds of reconfigurations (if available) against to the user-defined constraints. The basic idea is to first analyze the performance of the software reconfiguration and/or hardware reconfiguration approaches and to use the algorithmic reconfiguration if the performance constraints are not met because this latter is more time consuming.

We have experimented this design flow with the rapid prototyping of a flexible FFT as a use case. An architecture exploration was performed allowing the comparison of the three kinds of reconfigurations. This exploration will be detailed in Chapter 5.

Until now, there is no system architecture standard as well as common design flow to design SDR system making SDR system design a tough work. Therefore, in this section, we propose a general system architecture based on FPGAs and a design flow which is based on HLS to build this architecture.

4.3.1. System Architecture for Flexible Radio on FPGAs

In this section, an architecture, which can support the three above-mentioned reconfiguration schemes, is proposed. A general view of the system we propose is shown in Figure 4.4.

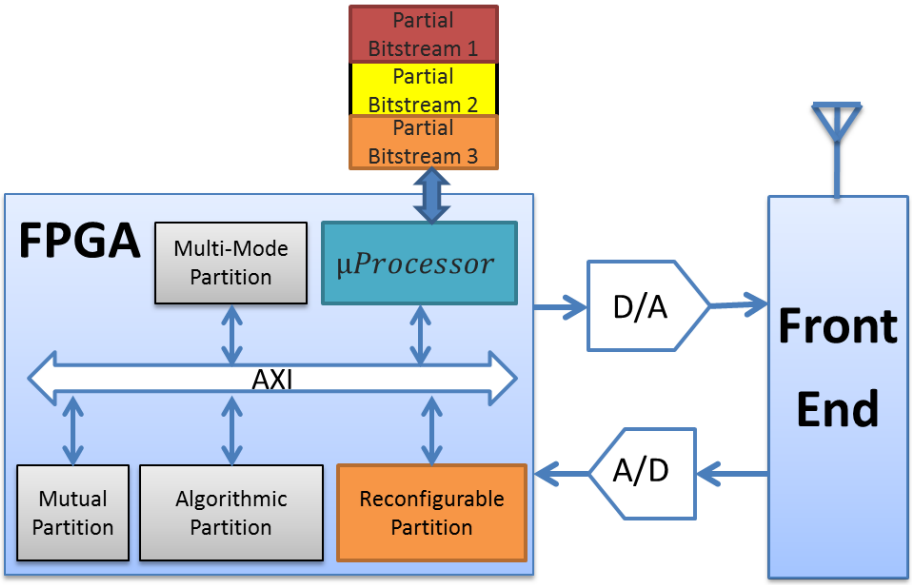


Figure 4.4 System architecture for FPGA-based SDR.

This architecture fits the model of a realistic SDR introduced in Figure 2.7. It is composed of an analog front-end to transpose the signal from RF to baseband frequency and a FPGA to compute most processing blocks (*e.g.* filter, modulation, channel equalization…). The system in the FPGA is composed of three main parts: the control, the connection and the execution partition.

The control: it is the brainpower of the FPGA part. It can be inside or outside the FPGA. Most of FPGAs now support an embedded microprocessor for a more efficient control. In Xilinx FPGAs, Microblaze is a soft-core microprocessor that can be used to manage the FPGA. Microblaze is highly configurable and it can be easily controlled via Xilinx Platform Studio (XPS). The control part manages all the activities of the system. For example, it can decide, according to external signals, to change some processing blocks of the waveforms or the full waveform. It can also turn on or turn off some processing blocks of the system due to power management policies. In our proposal, the control part manages the reconfiguration of the processing blocks for the three types of reconfiguration schemes. For hardware reconfiguration, we use an external flash memory in which full the partial bitstreams are stored as mention in Section 3.3.2.

The connection: it is backbone of the system. It provides all the connection in the system. At first, Xilinx support Processor Local Bus (PLB), a bus standard from IBM. However, Xilinx devices now switch to AXI bus, which has higher bandwidth comparing with PLB. In our system, an AXI-4 connection is used.

The execution partition: it is the part that executes all the computations of the system. There are four kinds of partitions in our proposal. The first one is the Mutual Partition that is used for all waveforms. It contains all the common processing blocks of the waveforms. The three others partitions are Multi-mode Partition, Reconfigurable Partition and Algorithmic Partition. These three partitions are used for the three kinds of reconfigurations discussed in the previous section. Among them, the Reconfigurable Partition only has partial bitstreams stored in flash memory.

4.3.2. Design Flow for Flexible Radio on FPGAs

Figure 4.5 shows the design flow to program a flexible radio on our proposed architecture. The design flow is composed of three different parts: the high-level language level, the embedded system design, and the platform integration level. Keeping a high-level language of description helps the designer to rapidly prototype the modules and to easily explore its design space. Moreover, it also helps for a more effective reuse of the code. HLS is used to implement most processing blocks. The code can be written in C/C++ or SystemC and is checked with design constraints before generating the RTL code.

The embedded system design step consists in packing the RTL codes and in connecting the IP cores into the system. Packing RTL codes is the last step in HLS design and is mostly supported by the HLS tool. The RTL codes from HLS are packaged in IP cores which are compatible and easy to connect to the system via the IP integrator tool. Packages are saved in the library. These packages as well as other processing packages from third party, from others libraries and from other projects can also be used. The designer creates a system with the processor unit and the AXI connection. The static IPs can be created and stored in the library. IPs that have reconfigurable partitions need to be managed by the reconfiguration tools before being added to the system. As said previously, the processor unit controls all the system.

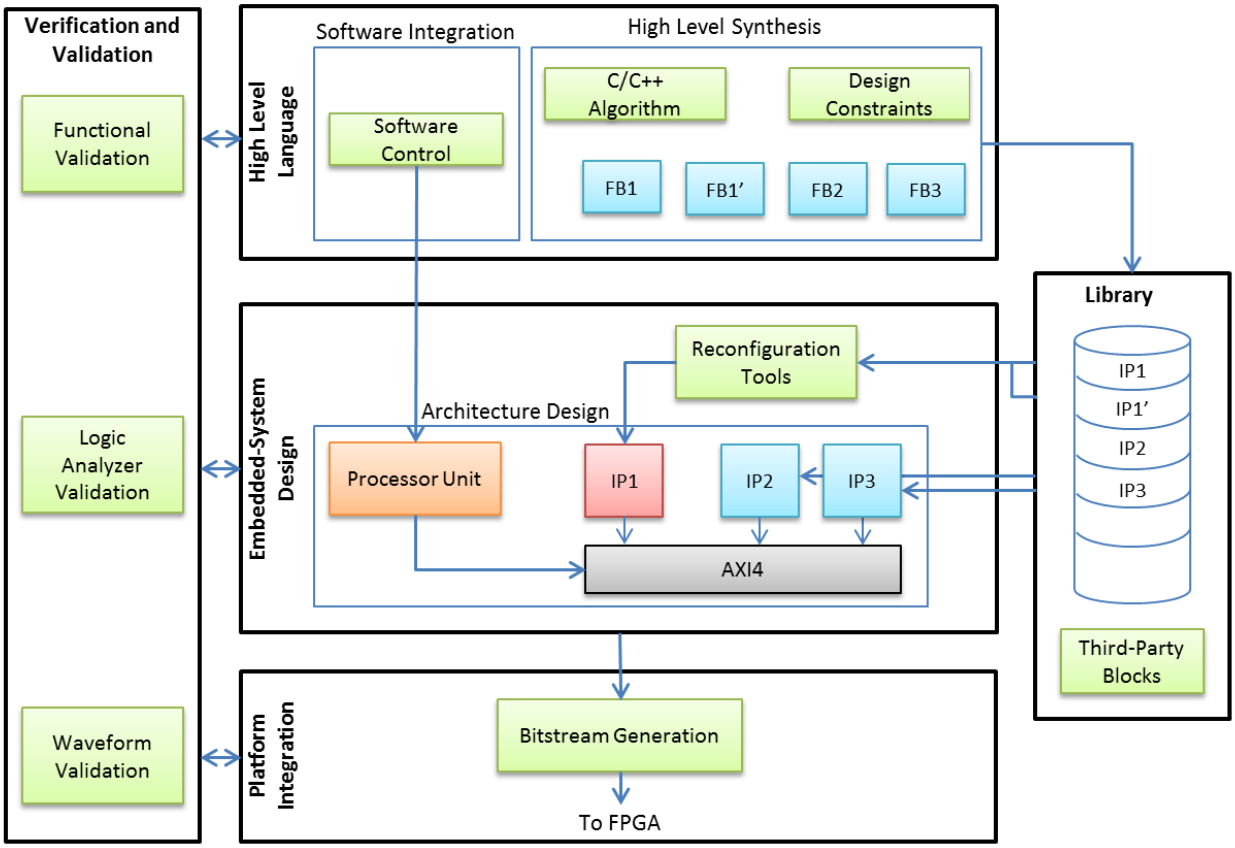


Figure 4.5 Proposed design flow for Flexible Radio using FPGA-based SDR.

Required tools: Our work relies on Xilinx development tools. At the High Level Language level, Vivado HLS tool is used. Vivado HLS can generate IP cores as output. If the output is native RTL, ISE is needed to

pack it to IP cores. XPS is used for embedded system design and Xilinx Software Development Kit (Xilinx SDK) for software integration. SDK uses C language to develop and control the Microblaze (processor unit). PlanAhead is used for managing hardware reconfiguration.

Software Development Kit (SDK)

The connections between the different tools are detailed in Figure 4.6. It shows each step to develop a flexible radio system using HLS tool and reconfiguration capability of FPGAs.

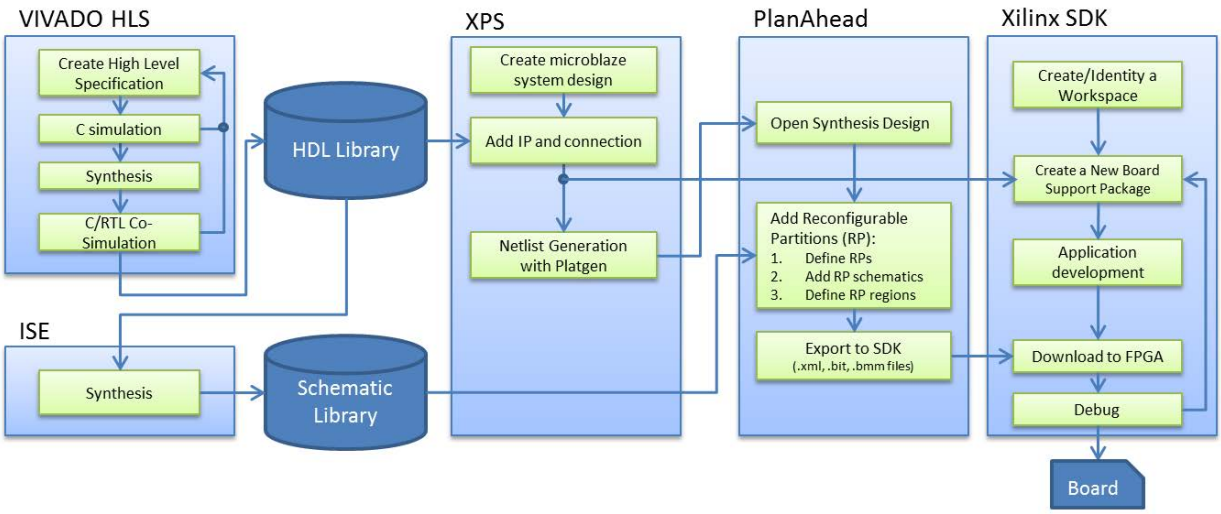


Figure 4.6 Step-by-step development flow using Xilinx tools.

4.3.3. Verification and Validation

Verification and validation is a very important step in a design flow as it allows checking the waveform specifications at different steps of the flow. In this section, the methods we use for verification and validation are discussed.

Using the proposed design flow (Figure 4.5), various tools can be used to verify each level of development. At first, we need to verify if the design is correctly built. This can be checked with the design programs. Secondly, we need to be verify that the functions of the design work as expected. It can be done by using different inputs and comparing outputs with expected values. Because most of the design programs have the auto-check programs, this section focuses on the validation of the functions.

Figure 4.7 summarizes the verification tools that are used in this work. They are declined into three levels: code level, baseband level and RF signal level.

At a code level, input and output references are needed to check all the processing blocks. These references can be quickly generated using Matlab software for instance. Matlab can simulate each processing block needed to build the waveform and therefore provide theoretical inputs and outputs.

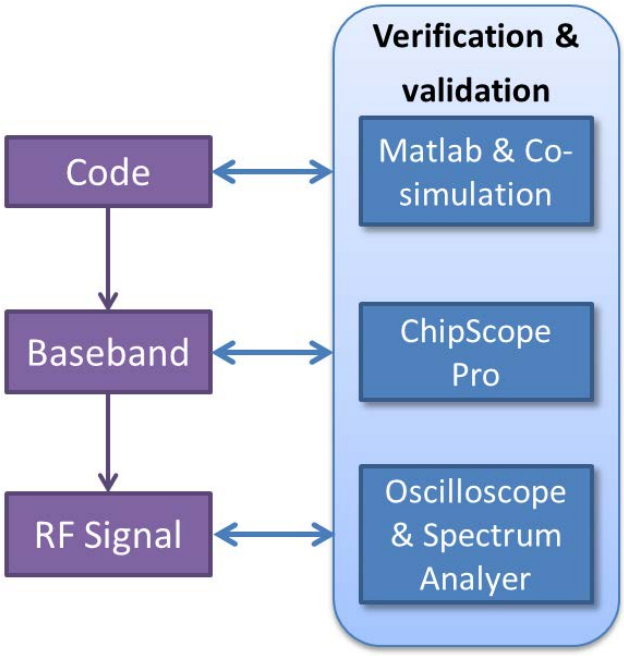


Figure 4.7 Verification and validation tools for the flexible radio design flow.

Figure 4.8 shows how code level verification is done when using Vivado HLS. In this case, Matlab is used to create random inputs. These inputs go through the simulation software to get the theoretical results (which are called as software outputs Sw_out). Both inputs and outputs are then written in an "inout.golden.dat" file and used as input/output references. The input reference is used by Vivado HLS to process using the C/RTL design. The obtained results (which are called hardware output Hw_out) are compared to the ones generated by Matlab. Because the data types in Matlab and Vivado are different and the two blocks are not exactly same, values of the two results are not exactly the same too. The designer must therefore decide the highest acceptable error value as a threshold. If this difference is smaller than the threshold, that program is validated. This process is done at both C and RTL level in Vivado HLS. Vivado HLS also provides the co-simulation function to make sure that the results obtained with the generated RTL code are the same as the ones obtained with the C code.

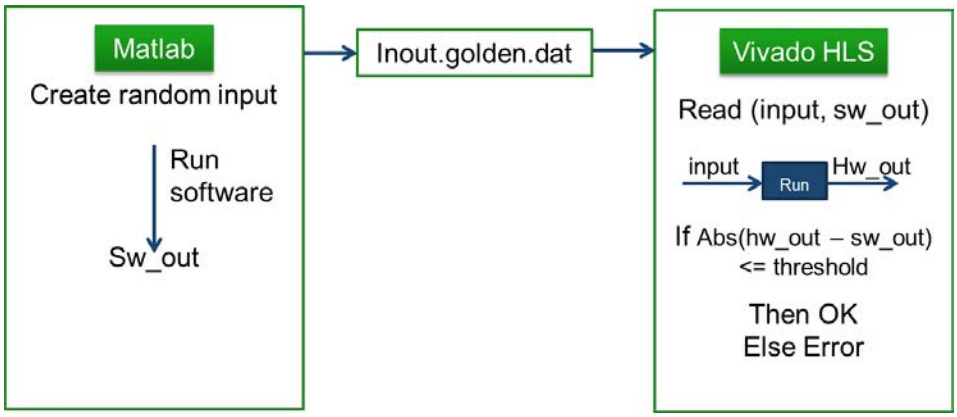


Figure 4.8 Example of code level verification in Vivado HLS based on a Matlab-provided reference.

Verification can also be done at the baseband level once the bitstream is generated and loaded into the FPGA board. To this aim, Xilinx provides a tool to test and verify the system in the FPGA: ChipScope Pro Analyzer. With a connection to a computer (the host), the FPGA board can send all the wanted signals to the host and show the results on the screen. Figure 4.9 illustrates verification using ChipScope Pro Analyzer for the monitoring of two baseband signals: the two outputs of a 64-QAM modulation are plotted, showing the constellation. However, this program does not support real time data acquisition and complex computations to deeply analyze the signals. For further calculations (*e.g.* analyzing the signal in the frequency domain), the designer can export the data to files.

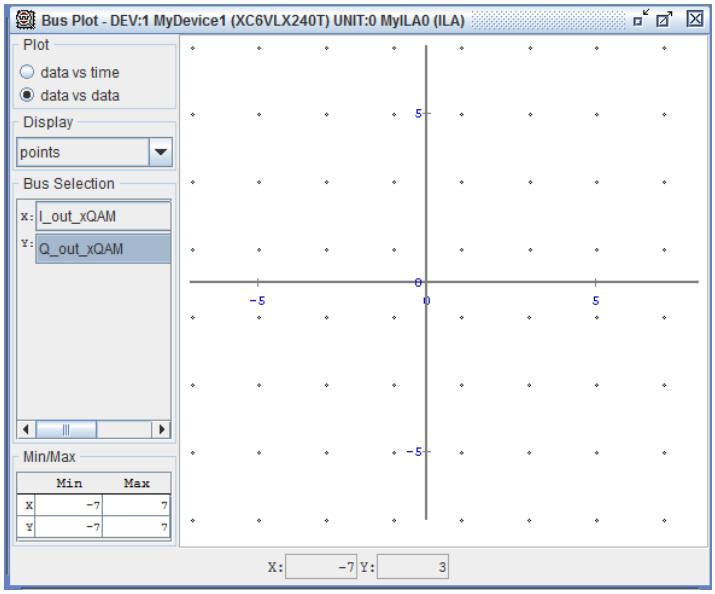


Figure 4.9 Example of baseband level verification using ChipScope Pro Analyzer for 64-QAM data.

At RF signal level, oscilloscope, spectrum analyzer or Vector Signal Analyzer can be used to verify the waveform generated by the design. Via the oscilloscope, the signals in time domain such as waveform and pulse can be verified. A spectrum analyzer enables to verify the signals in frequency domain. A Vector Signal Analyzer (VSA) can be used to demodulate the waveform according to predefined standard and therefore to check if the waveform has been correctly designed. Figure 4.10 shows an example with the RF spectrum for an OFDM-based modulation transmitted at 2.GHz

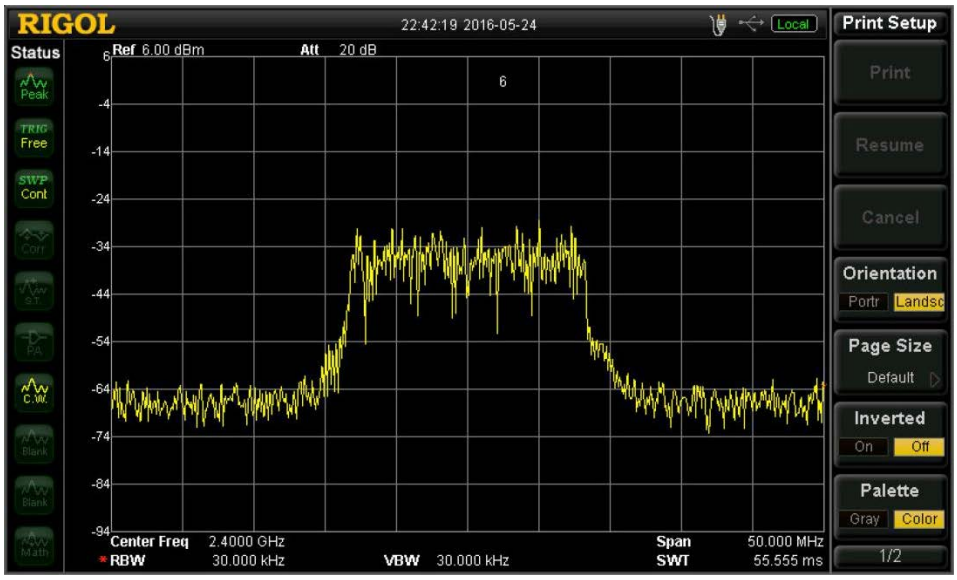


Figure 4.10 Example of RF signal analysis: an OFDM-based RF spectrum.

4.4. Conclusions

Implementing a radio waveform on FPGAs is not an easy task. Implementing a flexible waveform is a more challenging one. However, our work provides a complete solution from the waveform design to the architecture building and validation steps. The flow we propose helps the designer to build step-by-step his own flexible system. To this aim, our proposal leverages high-level languages and tools to benefit from an abstraction level which is more familiar to digital radio designers compared to hardware-level-based languages. For the flexible processing blocks, the approach focuses on three solutions: software reconfiguration, hardware reconfiguration and algorithmic reconfiguration. These three solutions have different advantages and drawbacks and can be used depending on the requirements of the flexible radio system (hardware resources, reconfiguration time, power consumption, ...). Our proposal is expected to reduce the time to market of flexible radio systems. A case study will be presented in the next chapter: the design flow is applied to a LTE-like waveform that must operate with various FFT sizes.

Chapter 5.Design and Exploration of a Flexible FFT for LTE standard

5.1. Introduction

In this section, a flexible FFT is designed using the proposed approach. Addressing LTE standard, the resulting FFT component should have six modes corresponding to these FFT sizes: {128, 256, 512, 1024, 1536, 2048}. Firstly, the facility to reuse a native C code in HLS is discussed in Section 5.2. Compatibility issues between native C code and C code for HLS are addressed and updated solutions are proposed. Each HLS tool has its own specific rules to be applied to re-write a high-level specification to make it synthesizable and these rules can be very different among HLS tools. As we said, in this work, we use Vivado HLS. To introduce and test the different kinds of reconfiguration methods introduced in the previous chapter, preliminary results are given in Section 5.3.1 and 5.3.2 with the design of a flexible FFT with two sizes: {128, 2048}. The design is based on two hand-coded FFT functions using the radix-2 DIT algorithm. We will see that when the number of loop iterations is a variable, Vivado HLS cannot estimate the latency. Thus, in Section 5.4, a latency estimation is proposed to help the designer analyzing the performance of the design. Lastly, a design space exploration is done for a power-of-two point FFT.

All experiments in this chapter are performed with the following setup and device: Vivado HLS 2013.3 is used for the HLS; DPR is setup with PlanAhead 14.6; the xc6vlx240tff1156 FPGA is targeted from Virtex 6 family since a ML605 evaluation board will be used for future demonstration.

5.2. Radix-2 DIT FFT/IFFT in Vivado HLS

The FFT estimates the spectral content of a time-domain sequence of digital signal samples, *i.e.* the results of the FFT are frequency-domain samples. The IFFT (inverse FFT) is the process to convert frequency-domain samples back to time-domain samples. Actually, from a computational point of view, except the normalization factor 1/N, one utilizes the same function[3] to perform both forward and inverse transform. In this chapter, we will basically talk about FFT only but all what we say applies for the IFFT too.

Our work initially relies on a C program dedicated for FFT computing and was written by Douglas L. Jones, 1992. This program is detailed in Algorithm 2 and is based on the radix-2 DIT FFT.

In Algorithm 2, the FFT size is presented as n and must be a power of two. The algorithm splits the FFT into two half-size FFT and repeats the process until the latest element is two-point FFT. This way to compute the FFT has been previously introduced in Section 2.4.2.

```
1. /*************************************************************/
2. /* fft.c                                                     */
3. /* (c) Douglas L. Jones                                      */
4. /* University of Illinois at Urbana-Champaign                */
5. /* January 19, 1992                                          */
6. /*                                                           */
7. /*   fft: in-place radix-2 DIT DFT of a complex input         */
8. /*                                                           */
9. /*   input:                                                  */
```

[3] one is with the complex conjugate compared to the other one

```
/* n: length of FFT: must be a power of two               */
/* m: n = 2**m                                            */
/*   input/output                                         */
/* x: double array of length n with real part of data     */
/* y: double array of length n with imag part of data     */
/*                                                        */
/*   Permission to copy and use this program is granted   */
/*   under a Creative Commons "Attribution" license       */
/*   http://creativecommons.org/licenses/by/1.0/          */
/**********************************************************/

fft(n,m,x,y)
int n,m;
```

```
double x[],y[];
{
int i,j,k,n1,n2;
double c,s,e,s,t1,t2;

j = 0; /* bit-reverse */
n2 = n/2;for (i=1; i < n - 1; i++){  n1 = n2;
  while ( j >= n1 )
  {
    j = j - n1;
    n1 = n1/2;
  }
  j = j + n1;
  if (i < j)
  {
    t1 = x[i];
    x[i] = x[j];
    x[j] = t1;
    t1 = y[i];
    y[i] = y[j];
    y[j] = t1;
  }
}

n1 = 0; /* FFT */
n2 = 1;

for (i=0; i < m; i++)
{
  n1 = n2;
  n2 = n2 + n2;
  e = -6.283185307179586/n2;
  a = 0.0;

  for (j=0; j < n1; j++)
  {
    c = cos(a);
    s = sin(a);
    a = a + e;

    for (k=j; k < n; k=k+n2)
```

```
64.     {
65.       t1 = c*x[k+n1] - s*y[k+n1];
66.       t2 = s*x[k+n1] + c*y[k+n1];
67.       x[k+n1] = x[k] - t1;
68.       y[k+n1] = y[k] - t2;
69.       x[k] = x[k] + t1;
70.       y[k] = y[k] + t2;
71.     }
72.   }
73. }
74.
75. return;
76. }
```

Algorithm 2 C program for the radix-2 DIT FFT.

The Algorithm 2 includes two main parts. The first part is the bit reversal part (from line 28 to 45) and the second part is the FFT computation (from line 50 to 73). When being described at a RTL level, several hardware techniques can be used to efficiently implement the bit reversal process. However, our research focuses on how to implement a processing block from its high-level description and mostly on the different ways to synthesize the computation part into the FPGA.

In the computation of the radix-2-based FFT part, there are three “for” nested loops. In the first loop (lines 50-73), lines 52 to 55 calculate the stage of the radix-2. At the beginning of the second loop (lines 57-72), the two coefficients for radix-2 are computed by using sin() and cos() functions. The third loop (lines 63-71) calculates all the radix-2 with the coefficients determined in the second loop.

The C program in Algorithm 2 does not care of any hardware considerations. Thus, it cannot be directly used as an input of a HLS tool and code re-writing is needed to tackle this issue. Next section discusses the main steps to update this program for Vivado HLS.

5.2.1. C code re-writing for Vivado HLS

In each HLS tool, specific rules have to be applied to write a program that can be synthesized to a RTL description. A native C code consequently needs adjustments before being able to be synthesized. In the case of the C code described in Algorithm 2, the two issues that need to be fixed to be compatible to Vivado HLS are:

- The data type of both inputs and outputs is double floating-point. Using floating-point will cost a lot of resources in hardware. Operations in floats are synthesized so that they are executed on a floating-point operator from Xilinx LogiCORE IP library. In this case, available optimized fixed-point arithmetic components such as DSP blocks can not be used.
- Sine and cosine functions do not exist in hardware language and will thus be unknown by the HLS compiler.

The first issue is typically addressed by using fixed-point instead of floating-point for the data type of both inputs and outputs. However, the number of bits used in the fixed-point representation should be carefully considered and a complete study is detailed in the next section.

Sine and cosine functions can be computed by reading a LUT with twiddle factor. For a given FFT size, the look-up is implemented as a constant array because the values of sine and cosine are predictable and calculations are therefore simplified. For instance, Algorithm 3 shows the code for sine function.

The DATA_SIZE in Algorithm 3 is related to the resolution of the table and is a constant so that `sin_lookup(n)` is equivalent to $\sin(\frac{2\pi n}{\text{DATA\_SIZE}})$. Based on the input n, the value of the index `idx` and sign of sine function are computed. To save memory footprint, the sin table keeps only values on the first quarter. The values for sine function are kept in sin_qtable.txt that contains (DATA_SIZE/4+1) values. The other values can be calculated based on these values by swapping the value and/or computing its opposite in four "if" conditions.

```
1. out_data_t sin_lookup (int n){
2.   out_data_t sin_table[DATA_SIZE/4+1] = {#include "sin_qtable.txt"};
3.   int idx;
4.   bool sign;
5.   if (n == DATA_SIZE){
6.     idx = 0;
7.     sign =0;
8.   }
9.   else if (n<=DATA_SIZE/4){
10.    idx = n;
11.    sign =0;
12.  }
13.  else if (n<DATA_SIZE/2) {
14.    idx = DATA_SIZE/4-n%(DATA_SIZE/4);
15.    sign = 0;
16.  }
17.  else if (n<3* DATA_SIZE/4){
18.    idx = n%(DATA_SIZE/4);
19.    sign =1;
20.  }
21.  else{
22.    idx = DATA_SIZE/4-n%(DATA_SIZE/4);
23.    sign =1;
24.  }
25.  return sign ? (out_data_t)-sin_table[idx] : (out_data_t)sin_table[idx];}
```

Algorithm 3 Sine lookup table based code.

5.2.2. Fixed-point FFT/IFFT in Vivado HLS

In this part, the number of bits used for the fixed-point data type is considered. In the literature, although floating-point is used in [78], most of designs use fixed-point [54] [79] [80]. In [80], integers are used for both input and output of IFFT. In [79], various data widths are evaluated using 18 bits for the integer part and several widths for fractional part, which generate different accuracies of the FFT

output. In [54], the architecture uses 14 bits for both input and output data with 2 bits for the integer part. However, 29-bit length is used for the intermediate data with 5 bits for the integer part. Actually, different data sizes are proposed depending on the input dynamic, the used algorithm and the required accuracy. To set the data size of an IFFT for an OFDM transmitter, we need to consider:

- **Length of IFFT**: For a fixed data size, different IFFT lengths achieve different accuracies. In our study case targeting LTE, 128-point IFFT is the shortest size while 2048-point IFFT is the largest one.
- **Input/Output of IFFT**: In the case of an OFDM transmitter, the IFFT inputs change according to the associated modulation, which can be BPSK, QPSK, 16-QAM or 64-QAM. Thus, the IFFT inputs can be real data encoded on two bits for BPSK or complex data encoded on 4 bits for 64-QAM (two integers between -7 to 7). Five bits are required for the integer part of the output. The accuracy is related to the width of the fractional part.
- **Two-point IFFT**: Two-point IFFT is the smallest unit of calculation in an IFFT. It is the basic block to determine the data range in an IFFT transformation. It is computed in Algorithm 2 in the lines 63 to 71. In a radix-2 DIT FFT transformation with FFT length $N=2^M$, each data goes M times through a two-point IFFT block.

Others parameters that should be considered when choosing the fixed-point data length are the resources used and the accuracy of the computation.

Resources and data length

The number of required resources (memories, LUT, FF or DSP slices) is related to the data lengths. Moreover, resources used for the FFT also depend on the structure of the architecture: pipeline, unrolled, optimized in latency or area, etc. In this section, we only discuss the effect of the data length on the resources in the case of the two-point FFT (Algorithm 4), which is the basic block for FFT calculation that does not depend on the structure of the architecture.

```
1. function  radix_2 (c, s, real[k], image[k], real[k+n1], image[k+n1])
2.   t1 = c*image[k+n1] - s*real[k+n1];
3.   t2 = s*image[k+n1] + c*real[k+n1];
4.   image[k+n1] = image[k] - t1;
5.   real[k+n1] = real[k] - t2;
6.   image[k] = image[k] + t1;
7.   real[k] = real[k] + t2;
8. end function
```

Algorithm 4 Two point FFT code.

First, the effect of data length on the number of DSP slices is considered. In Algorithm 4, there are four multiplications and FPGA synthesis uses DSP slices for multiplication (by default). In Xilinx Virtex 6, the DSP slice is the DSP48E1 structure shown in Figure 5.1. One can multiply two numbers with data lengths of 18 and 25. In Algorithm 4, multiplications are done between inputs and sin/cosine values. So if the data length of sin and cosine is lower than or equal to 18, it is possible to use only one DSP slice per

multiplication when the input data length is lower than or equal to 25. In our study, the data length used for sine and cosine memory values is thus 18 bits with 2 bits for the integer part (including sign). It provides a high accuracy while allowing the use of only one DSP slice. In that case, the number of DSP slices per multiplication only depends on the input data length. Table 5.1 provides the resources needed to implement the two-point FFT for different input data-lengths. Syntheses are based on a 100 MHz clock frequency. With input data lengths of 16 or 24, the multiplications are 18*16 and 18*24. Thus, 4 DSP slices are needed for the 4 multiplications of a radix_2. When the input data length is 32, the required multiplication is 18*32. Thus, it costs 2 DSP slices for each multiplication in this case.

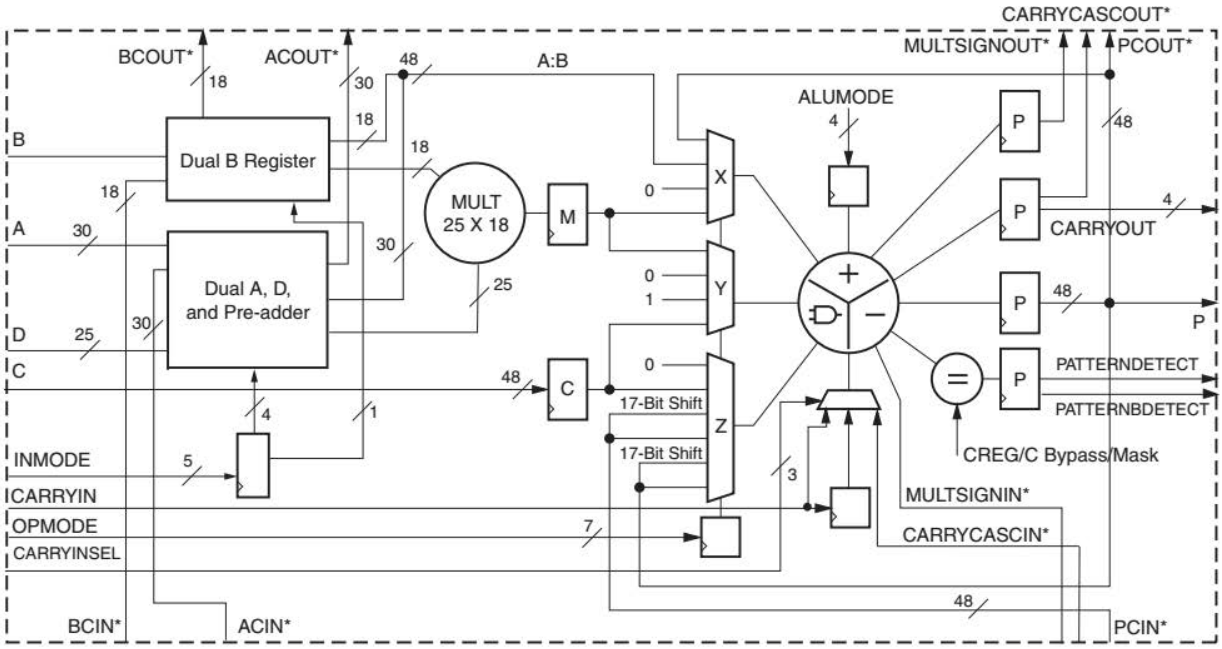


Figure 5.1 Virtex-6 FPGA DSP48E1 Slice [81]

Data-length	16	24	32
DSP slices	4	4	8
FFs	34	50	194
LUTs	69	101	245

Table 5.1 Resources required (after synthesis) for a radix_2 block for different input data lengths.

One can notice that if the input data length was set to 16, the data length of sine and cosine could increase up to 25 with no extra DSP cost. However, in this case, the number of LUT should increase to store these more accurate values of the sine and cosine. It is the reason why we decided to set the data length for sine and cosine to 18 bits.

Accuracy

In order to evaluate the accuracy of the data sizing for the FFT algorithm, N_{sample} floating-point inputs have been randomly generated. The FFT algorithm has been processed using first floating-points and then, using fixed-point with different data lengths. The accuracy is evaluated based on three parameters: the average error $\bar{e}$ [4], the standard deviation σ_e and the maximum error Max_e. The number of samples N_{sample} is equals to 100 000 and five FFT sizes are tested (128, 256, 512, 1024, 2048). Results are shown in Table 5.2.

Size of FFT/IFFT	**Data length = 16**			**Data length = 24**			**Data length = 32**		
	$\bar{e}$	σ_e	Max_e	$\bar{e}$	σ_e	Max_e	$\bar{e}$	σ_e	Max_e
128	2.86E-04	3.78E-07	1.78E-03	9.22E-06	1.28E-07	5.07E-05	3.01E-07	1.67E-08	1.26E-06
256	3.08E-04	1.85E-06	1.92 E-03	7.72E-06	7.87E-08	4.16E-05	2.88E-07	4.62E-09	1.26E-06
512	3.20E-04	2.09E-06	2.29E-03	6.13E-06	8.10E-08	3.28E-05	2.71E-07	1.10E-09	1.16E-06
1024	3.21E-04	3.46E-07	2.45E-03	4.89E-06	1.60E-08	3.87E-05	2.64E-07	3.87E-09	8.83E-07
2048	3.26E-04	9.68E-07	2.81E-03	3.89E-06	8.62E-09	2.25E-05	2.58E-07	1.67E-09	8.57E-07

Table 5.2 Output errors with different data lengths.

Except for sine and cosine values, five bits are used for the integer part in all cases. Thus, the accuracy of the data for data length equals to 16 is 2^{-11} = 4.88E-4. As we can see in Table 5.2, the average error is similar to the accuracy in this case. It means that data length affects the error. The accuracy for data length equals to 24 is 2^{-19} = 1.9E-6. In this case, the average error is still similar to the accuracy. When data length equals to 32, accuracy is 2^{-27} = 7.45E-9. The average error is much higher than the accuracy. It means errors are in this case caused by the FFT algorithm and not by the accuracy of the data.

5.2.3. Conclusions

Most native C codes cannot be directly used in HLS tool. They need to follow tool specific rules to be synthesized. Each HLS tool has its own rules that the designer needs to know. Furthermore, although floating point is very common in HLL, floating point is not supported by some hardware devices or at a cost of (too much) extra resources. The designer needs therefore to consider the accuracy of the application to wisely choose the fixed-point data length. This chapter analyses the accuracy of fixed-point implementations for 128 up to 2048-point FFTs using radix-2 DIT. With data length equals to 32, the errors mostly comes from the FFT algorithm itself whereas with 16 or 24-bit data lengths, errors are related to the size of the data. In this study, we want to design FFT with accurate results. Therefore, in the following of this study, all FFT/IFFT blocks will be based on 32-bit data length.

[4] the error $e[k]$ is defined by : $e[k] = |out_{Fix} - out_{Float}|,\ k= 1, ..., N_{sample}$ where out_{Fix} and out_{Float} are the outputs obtained when data are represented using fixed-points and floating-points respectively

5.3. Reconfigurable Blocks

In this section, the three different approaches for reconfiguring a block are discussed. Each approach is considered with the specific use case of the FFT function. Through experiments on these use cases, advantages and drawbacks of each approach are given. All syntheses in this section are based on a 100 MHz clock frequency and the data length of 32 bits.

5.3.1. Software Reconfigurable Block

First, two functions for 128- and 2048-point FFTs have been written and separately synthesized (`Block_FFT128()` and `Block_FFT2048()` respectively). Then an FFT with 2 modes (128/2048) has been designed using the software reconfiguration approach: based on Algorithm 1 (section 4.2.1), a function `Two_Mode_Block()` has been generated from `Block_FFT128()` and `Block_FFT2048()`.

Table 5.3 shows the synthesis results (number of resources and latency (in number of clock cycles)) of the three processing blocks. The resources are given with logical components such as the number of BRAM, DSP slices, LUT and FF. As expected, `Block_FFT2048()` requires more resources than `Block_FFT128()`: the number of BRAMs to store cosine and sine coefficients and input data is twofold and the number of DSP slices is 4 times more. Indeed, in `Block_FFT128()`, there are 32 cos/sin values stored in the memory. It uses 6 BRAMs for memory. `Block_FFT2048()` needs more cos/sin values and thus it needs 12 BRAMs. As discussed in section 5.2.2, the radix-2 function with data length of 32 uses 8 DSP slices. Moreover, as Block_FFT128() has complex value as inputs, it needs to use two radix-2 functions simultaneously: one for the real part and one for the imaginary one. It also needs one DSP slice for calculating cos/sin index in the second loop. The `Block_FFT2048()` uses eight radix-2 functions at a same time to reduce latency: four for the real part and four for the imaginary one. Thus, it uses 64 DSP slices for radix-2 calculations. Latency of the 2048-point FFT is much higher than the 128-point FFT one because with this architecture the computations are iterated.

Processing block	`Block_FFT128()`	`Block_FFT2048()`	`Two_mode_Block()`
BRAM	6	12	12
DSP Slices	17	65	82
LUT	1017	2522	3459
FF	862	2443	3241
Latency	5362	72410	5491/72411

Table 5.3 Software reconfiguration synthesis results for a FFT with 2 modes (128/2048).

The results for `Two_mode_Block()` show that no resource optimization is obtained uing the software reconfiguration. Except for the number of BRAM, Table 5.3 shows that the resources used by `Two_mode_Block()` are (a little bit less than) the sum of the resources used by the two FFT blocks when synthesized separately. Vivado HLS tool does not share the resources between the two functions although they are not executed at the same time. In this case, from the resources point of view, software reconfiguration appears not to be an efficient solution to implement a flexible block. Latency is similar to mono-mode blocks.

5.3.2. Hardware Reconfigurable Block

The two functions `Block_FFT128()` and `Block_FFT2048()` are now used for hardware reconfiguration. In Xilinx's FPGAs [66], the functions to be dynamically placed are mapped into an area called a reconfigurable partition. Generally speaking, a reconfigurable partition has a rectangle shape dedicated to the function it implements. Every resource in this area may not be used by the implemented function (*i.e.* the resources in this area are private to the partition even if they are not used).

Table 5.4 shows the resources used when implementing hardware reconfiguration. Partial bitstream sizes and reconfiguration time are also given. Two partitions have been first generated using PlanAhead tool: one for the `FFT_128()` only and one for the `FFT_2048()` only. The reconfiguration time depends on the bitstream size so the reconfiguration time for `FFT_128()` is smaller than `FFT_2048()`'s one. Reconfiguration time is computed from PRCC tool (Partial Reconfiguration Cost Calculator) from Technical University of Crete [82] assuming that the reconfiguration controller is an on-chip PowerPC processor at a throughput of 10.297 Mbyte/s [5].

For comparison, Table 5.4 also shows the resources needed by `FFT_128()` and `FFT_2048()` when they are placed as static logic (*i.e.* not as a reconfigurable module using a reconfigurable partition) so resources needed are same as in Table 5.3. It highlights the extra resource cost due to dynamic reconfiguration.

In practice, to perform the DPR of the 2 functions, a third partition called `FFT_128/2048()` in Table 5.3 has been specified. In this case, the two functions are placed on the same partition, *i.e.* on the same area of the FPGA. For each type of logical component, the resources used by this partition are related to the more costly case. As said in Section 3.3.1, FPGAs uses configuration frames to control FPGA memory space and a configuration frame is the smallest addressable unit in the FPGA. Therefore, the number of resources used in a FPGA is always a multiple of configuration frames and the partition used is the smallest partition which can cover all the required resources. In the hardware reconfiguration approach, the partition that is used in practice is the one which covers all necessary resources for each configuration case. In our case, `FFT_2048()` partition always needs the largest number of resources whatever the kind of logical component. Thus the resulting `FFT_128/2048()` partition is mainly based on the `FFT_2048()` partition.

Reconfigurable partition's latency is equal to the latency of the function when synthesized alone onto a static region. The hardware reconfiguration needs 32.9 ms to switch from one mode to the other one (time related to the 416016-Byte bitstream size of the partition whatever the mode) whereas only one clock cycle is required with software reconfiguration. It means in practice, with hardware reconfiguration, many OFDM symbols will be lost when changing the mode.

[5] Higher throughputs up to 400 MBytes/s may be reached using a dedicated controller so that reconfiguration time can be reduced.

	Processing block: Resources needed		Partition: Resources used		
	FFT_128	FFT_2048	FFT_128	FFT_2048	FFT_128/2048
BRAM	6	12	6	17	17
DSP	17	65	24	68	68
LUT	1017	2522	1440	4080	4080
FF	862	2443	2880	8160	8160
Bitstream size	n/a	n/a	138672 Bytes	416016 Bytes	2 x 416016 Bytes
Reconf. Time	n/a	n/a	10.98 ms	32.9ms	32.9 ms
Latency	5362	72410	5362	72410	5362/72410

Table 5.4 Hardware reconfiguration synthesis results for a FFT with 2 modes (128/2048).

5.3.3. Algorithmic Reconfigurable Block

The flexible FFT for LTE standard should have six modes related to the FFT sizes: {128, 256, 512, 1024, 1536, 2048}. Based on the previous results, a software reconfiguration will generate a huge component as the resources are not shared. Hardware reconfiguration makes resource sharing possible but may require a long reconfiguration time since it is based on the more resource consuming partition. In this section, we first present a power-of-two point FFT for algorithmic reconfiguration to share the resources between its different modes. Then, a `BLOCK_FFT_1536()` function, *i.e.* not a power-of-two point FFT, is presented. Sharing the resources between these 2 functions is also discussed. In this section, experimental results for the algorithmic reconfiguration based FFT are not given. Design space exploration with this FFT will be done latter in this chapter and in chapter 6 too.

Power-of-two point FFT:

The power-of-two point FFT has 5 modes {128, 256, 512, 1024, 2048}. Algorithm 5 presents a brief overview of the algorithm of the power-of-two point FFT to show how it works. In this algorithm, FFT length N is presented as `FFT_size`. A dedicated control signal is used to decide the mode. Vivado HLS tool deals with the FFT size as a variable. Indeed, as presented in Algorithm 5, FFT size and FFT stages are calculated based on the control signal value. Once FFT size and FFT stages are determined, a standard three-nested-loop structure for the FFT based on radix-2 is executed. The first loop determines the stages. The second loop chooses butterflies with the same twiddle factor at each stage. Last loop computes all the chosen butterflies. This algorithmic reconfiguration generates a `Block_FFTpow2()` function with only one main FFT core for the five different modes. With this function, the resources should be approximately the ones used by the largest FFT (*i.e.* 2048). Synthesis directives, as we will see latter, will decide these required resources.

```
1. function  BLOCK_FFTpow2 (inputs, outputs, control)  -- 0 <= control <= 4
2.   FFT_size_max = 2048
3.   FFT_stages_max = 11
4.   FFT_size = FFT_size_max >> control
5.   FFT_stages = FFT_stages_max – control
6.   Bit_reverse()                              -- re-order before calculating
7.   Loop1: for i = 0 to FFT_stages do
8.     n1 = n2;                                 -- choose the stage
```

```
9.     n2 <<= 1;
10.    idx >>= 1;
11.    Loop2: for j = 0 to n1 do
12.      c = sin_lookup()                    -- determine coefficients for radix2
13.      s = cos_lookup()
14.      Loop3: for k = j to FFT_size; k = k + n2 do
15.        Radix_2()
16. end function
```

Algorithm 5 Overview of the algorithmic reconfiguration based code for the power-of-two point FFT.

1536-point FFT:

The 1536-point FFT is not a power-of-two FFT so the structure of the algorithm can not be the same. The design of a FFT for LTE standard with six modes {128, 256, 512, 1024, 1536, 2048} will thus be based on two separated algorithms, one for the power of two FFT and the other one for the 1536-point FFT.

By applying the Cooley-Tukey algorithm [83] for a FFT size of 1536, the `BLOCK_FFT_1536()` function can be generated using three `Block_FFT_512()` functions and one radix-3 function [84] as shown in Figure 5.2. First, the 1536 inputs of the FFT are split into three parts. Those parts are computed as three 512-point FFTs independently. Then, while the first part is kept as it is, the second and the third ones are multiplied by twiddles factors. Last, the radix-3 function is applied to compute the final results.

Resource sharing may theoretically be done between `Block_FFT_512()` functions of the `BLOCK_FFT_1536()` and the `Block_FFTpow2()` because they are both based on the radix-2 algorithm. On the contrary, the `BLOCK_Radix3()` function is based on radix-3 thus it can not share resources.

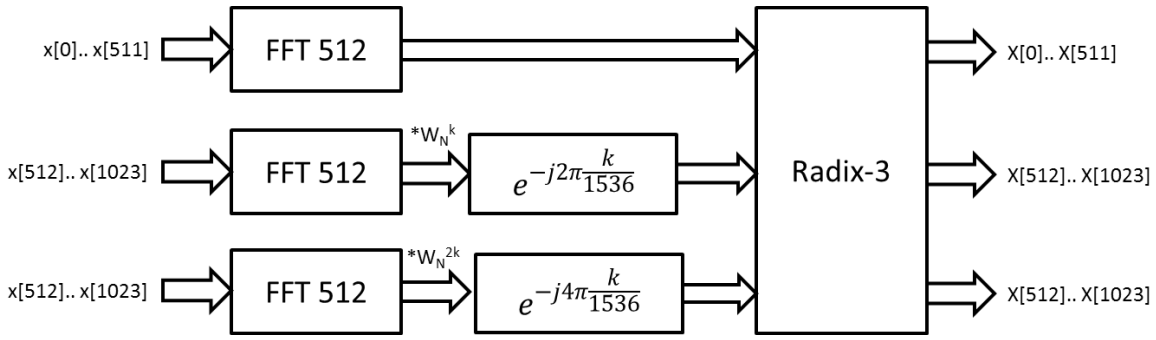


Figure 5.2 Structure of the 1536-point FFT

5.4. Latency Estimation

Vivado HLS estimates the latency of the generated architectures. This functionality works well when the number of loop iterations is a constant. For example, the latency of `Block_FFT512()` and `Block_FFT2048()` are provided in synthesis reports. However, this functionality does not work when number of loop iterations is a variable as it is in our case. However, knowing the latency is important for the designer especially when DSE is performed to find the best tradeoff between latency

and resources. This section discusses a way to overcome this issue and a formula to estimate the latency is proposed.

5.4.1. Computing an Estimate of The Latency

As the latency estimation is provided by Vivado HLS only when the number of the loop iterations is a constant, it is not available for the algorithmic reconfigurable block Block_FFTpow2( ) in which the loop boundaries depend on the FFT size. This drawback prevents from estimating the latency and applying some optimization directives to the system accordingly. A first solution is to perform a test for each configuration by removing the boundary issue, allowing the tool to estimate the latency. However, this solution needs long experimentation time since DSE will be performed to get the best performance for a particular FFT size.

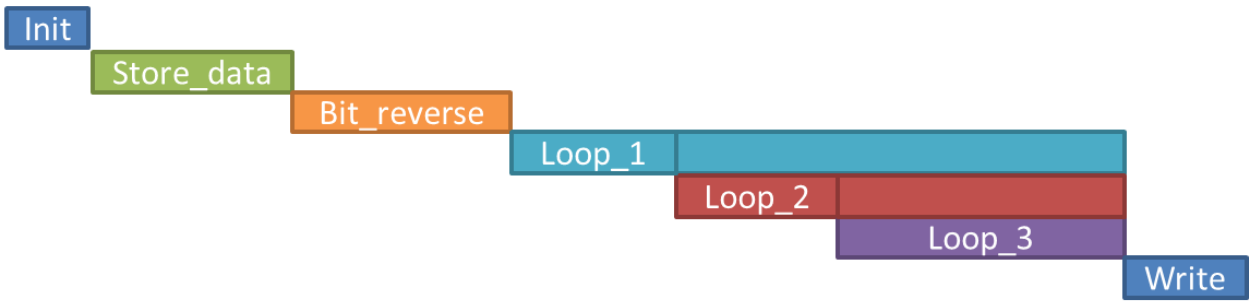


Figure 5.3 The chronogram of the different steps of Algorithm 5.

Another solution is to estimate ourselves the latency using a formula. Actually, using Algorithm 5, the latency of the FFT block can be predicable. However, the number of times loop3 is executed can be hard to determine because variable k depends on two other variables j and n2. The chronogram of the different steps is showed in Figure 5.3. From this figure, we can label the latencies of the different steps: $L_{initialization}$, L_{store_data}, $L_{bit\_reverse}$ and L_{write_data} are the latencies required for initialization, storing data to RAM, bit reverse operation and writing new data back to RAM respectively. For the three-nested-loop FFT core, the loops have latency denoted L_{loop1}, L_{loop2} and L_{loop3} respectively and N_1, N_2 and N_3 are the numbers of loop iterations.

The latency (in clock cycles) can be expressed as:

$$Latency = L_{initialization} + L_{store\_data} + L_{bit\_reverse} + N_1*(L_{loop1} + N_2*(L_{loop2} + N_3*L_{loop3})) + L_{write\_data}. \quad (5.4)$$

By denoting $N_{loop1} = N_1$; $N_{loop2} = N_1*N_2$; $N_{loop3} = N_1*N_2*N_2$; the numbers of times each loop is executed, (5.4) can be rewrited as:

$$Latency = L_{initialization} + L_{store\_data} + L_{bit\_reverse} + N_{loop1}*L_{loop1} + N_{loop2}*L_{loop2} + N_{loop3}* L_{loop3} + L_{write\_data}. \quad (5.5)$$

In the case of Block_FFTpow2( ), the FFT size is 2^n with n the number of stages.

We can determine and verify by using Vivado HLS that:

- $L_{initialization}$ = 1 (simple calculation)
- L_{store_data} = L_{write_data} = 2^n (the pipelined data. So this latency is the size of the FFT)
- $L_{bit\_reverse}$ = Number of reverse * $L_{1\_reverse}$ = 2^n * $L_{1\_reverse}$
  where $L_{1\_reverse}$ is the latency of one reverse input (line 6 in Algorithm 5) that needs to reverse 2^n inputs

Thus (5.5) becomes:

$$Latency = 1 + 2^n*(2 + L_{1\_reverse}) + N_{loop1}*L_{loop1} + N_{loop2}*L_{loop2} + N_{loop3}* L_{loop3}. \quad (5.6)$$

When Loop3 of Algorithm 3 is unrolled, N_{loop3} and L_{loop3} depend on unrolling factor U. Unrolling factor U is a power of 2, so we have $U=2^m$:

- $N_{loop1} = N_{FFT\_stages}$= n (number of stages based on Algorithm 5)
- $L_{loop1} = L_{Calculate\_index}$= 1 (simple calculation)
- $N_{loop2} = N_{twiddles}$= $1 + 2^1 + 2^2 +...+ 2^{n-1} = 2^n - 1$ (number of twiddles based on Algorithm 5)
- $N_{loop3} = 2^{n-m-1}*(n-m) + 2^{n-m} + ... + 2^{n-1} = 2^{n-m-1}*(2^{m+1} + n - m - 2)$ (number of radix-2 in the algorithm using unrolling factor based on Algorithm 5)

Eventually, the latency becomes:

$$Latency = 1 + 2^n*(2 + L_{1\_reverse}) + n + (2^n - 1)*L_{computing\_twiddles} + 2^{n-m-1}*(2^{m+1}+n-m-2)*L_{loop3}. \quad (5.7)$$

Thus, using equation (5.7), the latency of `Block_FFTpow2()` can be estimated for each FFT size. This latency depends on three latencies:

- $L_{1\_reverse}$: it can be determined from Vivado HLS synthesis,
- $L_{computing\_twiddles}$: it can be determined from Vivado HLS synthesis too,
- L_{loop3}: this can be determined by the frequency and the unrolling factor.

To summarize, equation (5.7) is a latency estimation of Block_FFTpow2() latency. It is based on three parameters: $L_{reverse}$, $L_{computing\_twiddles}$ and L_{loop3}. Although Vivado HLS cannot estimate the overall latency when the number of loop iterations is a variable, one can get the values of $L_{reverse}$, $L_{computing\_twiddles}$ and L_{loop3} from one single synthesis[6] [7]just analyzing timings reports after synthesis.

At a same frequency, this latency does not change. So, we just need to synthesize one time to estimate the latency.

5.4.2. Experiment and Results for Latency Formula

First, the test was done for a clock frequency of 100MHz. At this frequency, $L_{1\_reverse}$= 4 and $L_{computing\_twiddles}$ = 5. The latency of loop 3 depends on the unrolling factor and is given in Table 5.5.

[6] The value of the parameters depends on the frequency constraint so if this constraint changes, a new synthesis is required to get the new values.

[7] L_{loop3} depends on the unrolling factor U, so one synthesis is required for one particular value of U .

Unrolling factor U	1	2	4	8
L_{Loop3}	8	10	14	24

Table 5.5 The latency of loop number 3 depending on unrolling factor

Figure 5.4 shows the ratio of each part of the algorithm in the latency for FFT 2048 and FFT 128. As expected analyzing Equation 5.7, the latency of radix-2 takes about ¾ of whole latency. Thus, special attention to this part of the computations should be paid to reduce the latency. In Section 3.2.3, unroll directive in Vivado HLS was presented about how to manage parallelism with "for loop". This feature can be applied in this case to reduce the latency. This will be discussed in more detail on the next section.

An accurate comparison between the estimate and the results after synthesis and simulation is shown in Table 5.6 for different unrolling factors and FFT sizes. Table 5.6 shows that the estimated latency is quite similar with latency obtained when simulating the architecture generated by Vivado-HLS. The difference is lower than 2%. It shows that the latency formula provides accurate estimation of the latency of the power-of-two FFT. Moreover, unroll directive seems to be a good candidate to reduce the latency and should be further analyzed in detail. In the next section, a full DSE will be done to deeply consider the influence of the unroll directive.

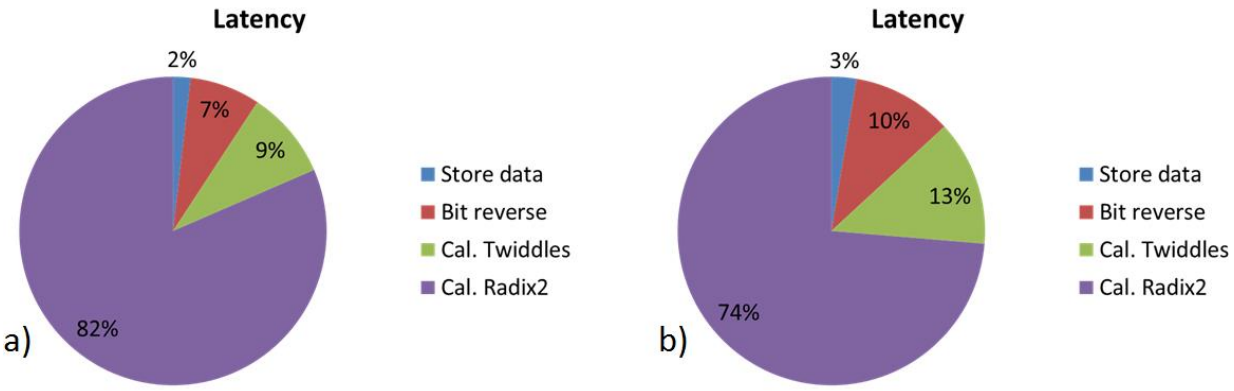


Figure 5.4 FFT latency with unrolling factor U = 1 a) FFT 2048; b) FFT 128.

Unrolling factor U	**FFT**	**Estimation latency**	**Synthesis Latency**	**Difference**
1	128	4995	5042	0.93%
	256	11012	11122	0.99%
	512	24069	24298	0.94%
	1025	52230	52714	0.92%
	2048	112647	113626	0.86%

2	128	3971	4019	1.19%
	256	8580	8690	1.27%
	512	18437	18666	1.23%
	1025	39430	39914	1.21%
	2048	83375	84954	1.86%
4	128	3875	3922	1.2%
	256	8196	8306	1.32%
	512	17285	17514	1.31%
	1025	36358	36842	1.31%
	2048	76295	77274	1.27%

Table 5.6 Estimated latency and latency obtained by simulation after HLS.

5.5. Design Space Exploration in the Power-Of-Two Point FFT

HLS allows the DSE of a processing block by using synthesis directives/constraints. This part presents the DSE of the function `Block_FFTpow2()`. Aim is to know if it is possible to design a flexible FFT that respects design constraints (area, latency, throughput...) using HLS. Several directives are made available with Vivado HLS tool (*e.g.* memory mapping, pipeline, loop unrolling, inline, see section 3.2.3). They enable optimizing the design for area or latency. In this study, because of the loop structures and the data dependencies of the FFT, unrolling loops is used. Loop unrolling reduces the total loop iterations by duplicating (with a U factor) the loop body so that we can tradeoff between area and/or latency. In this section, we first deeply study the DSE at a clock frequency of 100MHz before expanding the analysis to other frequencies.

5.5.1. DSE at Clock Frequency of 100 MHz

Figure 5.5 shows the latency of `Block_FFTpow2()` processing block as a function of the number of DSP slices. In practice, four multi-mode components have been generated by varying the unrolling factor U. Each component is characterized by its numbers of DSP slices. The number of DSP slices increases with the unrolling factor because the FPGA needs more resources to calculate in parallel. For instance, 16 DSP slices are required when U=1 whatever the FFT size. If U=2, twice DSP slices are required, etc. Theoretically, the higher unrolling factor we use, the better calculation capability we get. However, it is not guaranty that the latency is lower. Indeed, in Figure 5.5, the lowest latency is obtained for U = 4. Due to a bond in Equation 5.7, DSP number increase does not always means they are efficiently used in practice. We noticed the same shape is also obtained for the other resources (latency as a function of the number of LUT and FF). Thus, U=4 seems to be a good tradeoff between the number of resources and the latency at frequency 100 MHz.

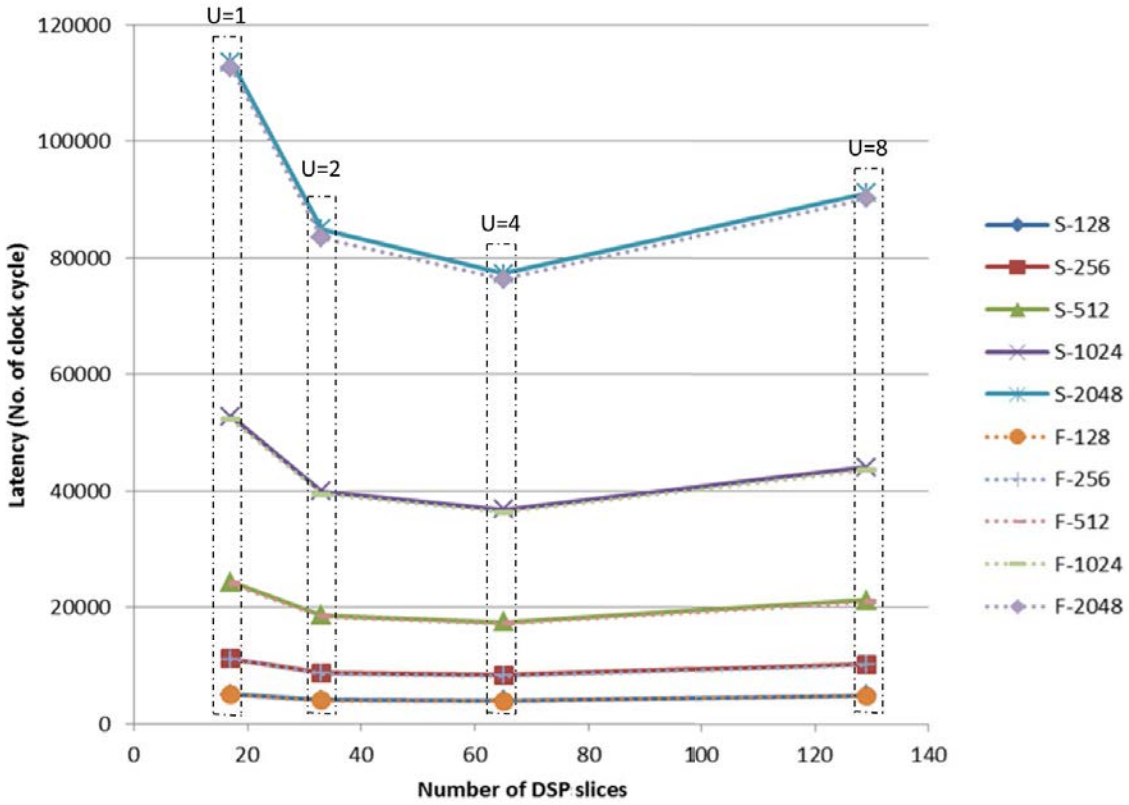


Figure 5.5 The latency vs number of DSP slices (S is for Simulation after synthesis and F is for the Formula estimate) (frequency = 100 MHz).

Memory access bottleneck

To fully understand the reason why the latency does not reduce when the unrolling factor is higher than 4, we need to deeper analyze the code and figure out the reason why.

If the third loop is written like in Algorithm 6 (piece of code taken from Algorithm 5), the latency of the third loop will considerably increase. Indeed, in Vivado HLS, an array is implemented into a RAM. Because inout_1(k) and inout_2(k) are both inputs and outputs, the RAM accesses are blocked until the output is written into the RAM. This process is illustrated in Figure 5.6. In Figure 5.6, "read" stands for RAM reading and "write" stands for RAM writing. The three blocks "mult" represent for multiplication latency in radix_2 computation. This multiplication is implemented in DSP slices. Because of the data length, more than 1 DSP slice is required to calculate. Thus, a cascade structure of DSPs slice is used and it takes three clock cycles to complete this calculation at frequency 100 MHz.

```
1.    Loop3: for k = j to FFT_size; k = k + n2 do
2.               Radix_2(c, s, inout_1(k) , inout_2(k))
```

Algorithm 6 Shortcut of the third loop in the FFT.

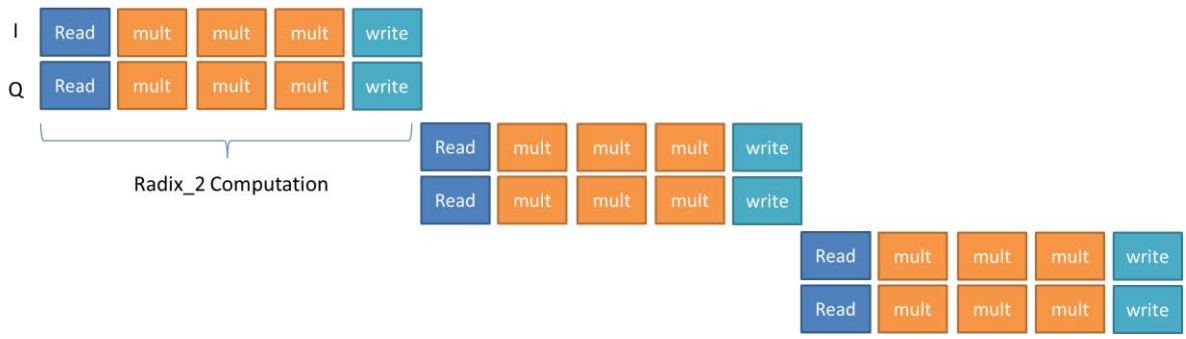


Figure 5.6 Timing chronogram of the third loop based on Algorithm 6.

Figure 5.6 shows that the second Radix_2 can only run when the first one finishes. This limits the capacity of the pipeline directive and increases latency of the third loop dramatically. Actually, Vivado HLS provides a directive to fragment an array into separated memory. However, this directive cannot handle with total fragmentation of a large array. Thus, this solution can not be synthesized in practice.

Actually, Algorithm 6 should be changed to Algorithm 7 using a trick.

```
1.   Loop3: for k= j to FFT_size; k = k + n2 do
2.                  {
3.                       temp_1(p) = inout_1(k);
4.                       temp_2(p) = inout_2(k);
5.                       Radix_2(temp_1(p), temp_2(p), c, s);
6.                       inout_1(p) = temp_1(k);
7.                       inout_2(p) = temp_2(k);
8.                  }
```

Algorithm 7 Shortcut of the updated third loop in FFT.

As temp_1 and temp_2 are small arrays, they can be are easily fully fragmented. By this way, the memory accesses are not blocked and can be used for possible unroll or pipeline. Figure 5.7 a) shows that with U=4, four Radix_2 are computed in parallel[8]. However, unrolling is still limited by the number of data reads and writes at a given time. Indeed, Figure 5.7 b) shows that with unrolling factor U=8, the latency is not reduced due to the memory access bottleneck. Indeed, a RAM has limited access at a given time so that the results cannot be written back just after finishing the calculation: the RAM is "busy" reading new data. Thus, even if more resources are used with U=8, the latency is not reduced compared to U=4.

[8] With of course extra resource cost (48more DSP slices more than U=1)

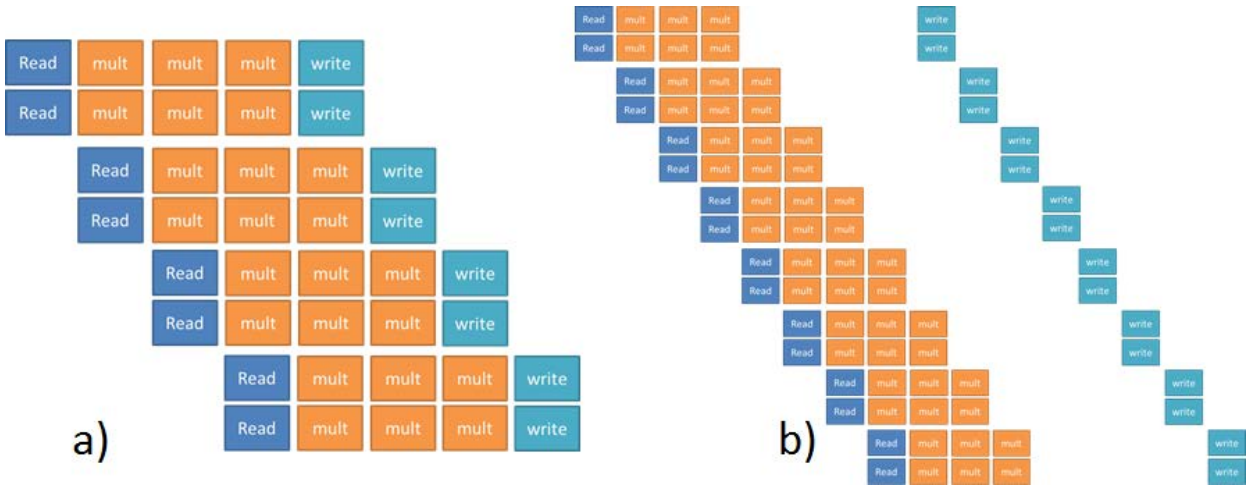


Figure 5.7 Timing chronogram of the third loop based on Algorithm 7: a) U = 4; b) U = 8.

5.5.2. DSE at Other Clock Frequencies

Synthesis results at clock frequencies 200 MHz and 500 MHz are shown in Figure 5.8. The results are quite similar with results at 100 MHz. Both the unrolling factors 2 and 4 are good tradeoffs and can be considered depending on the constraints of the application.

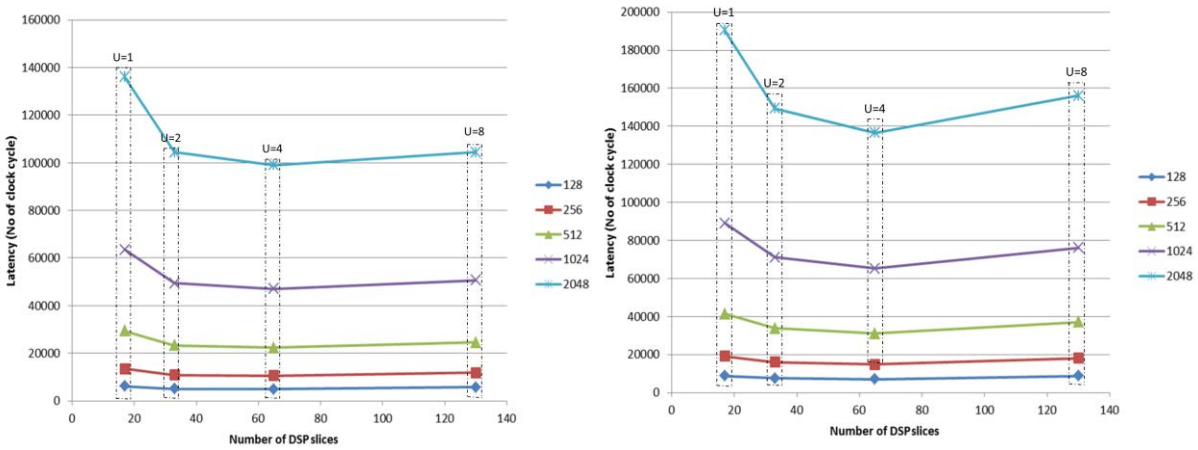


Figure 5.8 Latency vs number of DSPs for clock frequencies 200Mhz (left) and 500MHz (right).

The results of the DSE for a lower frequency, that is to say 50 MHz in our case, is provided on Figure 5.9. This figure shows that the unrolled directive of Vivado HLS is not efficient in this case. It is not what is expected in theory. In practice, at a clock frequency of 50 MHz, the timing constraint is not an issue. Actually, it is possible to do the radix-2 multiplications in one clock cycle (excluding read and write) at 50 MHz (3 cycles are required at 100 MHz or higher frequencies). Thus, the latency of the multiplication is only 1 clock cycle and the latency of third loop is only 3 clock cycles in this case including reads and writes in memory (it is 5 clock cycles at 100 MHz). As a consequence, the unrolled directive does not take advantage to reduce latency while increasing the resources. One can see that overall latency is

even worse. So the unrolled directive is not efficient at this frequency (see also Figure 5.13's comments for more information).

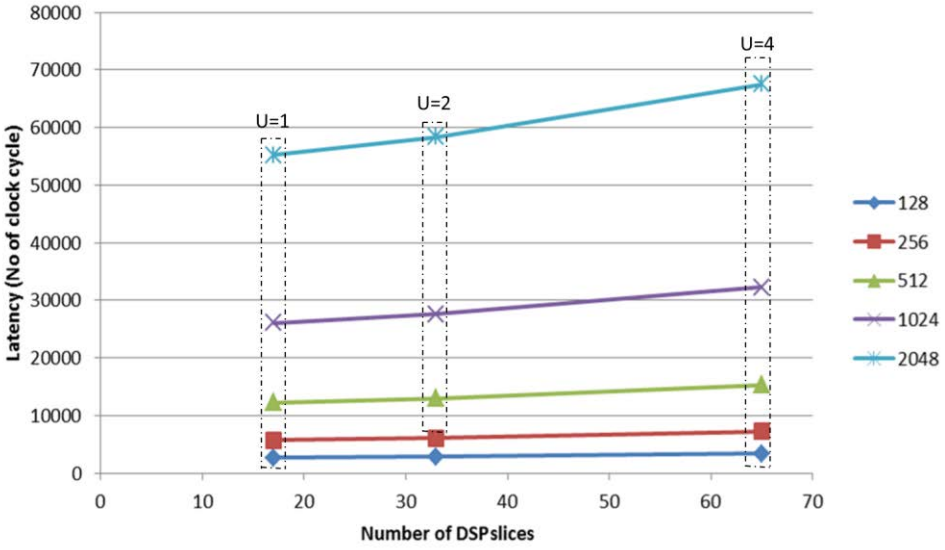


Figure 5.9 Latency vs number of DSPs at frequency 50MHz.

5.5.3. Comparisons and Conclusions

In this section, the DSE will be considered among various frequencies. Figure 5.10 shows the delay to calculate one iteration of Loop 3 iteration when unrolling factor U = 1 (*i.e.* equals the delay to calculate a Radix_2 FFT). In this figure, we can see that for clock frequency higher than 100 MHz, the delay is reduced when the frequency increases. However, at a clock frequency of 50 MHz, the delay is lower than for 100 MHz because DSP slices can calculate two multiplications in one clock cycle[9].

[9] For a 50MHz frequency, it takes three cycles for Loop 3 (one for read data, one for multiply and one for write data). Latency is thus 20ns *3 cycles.

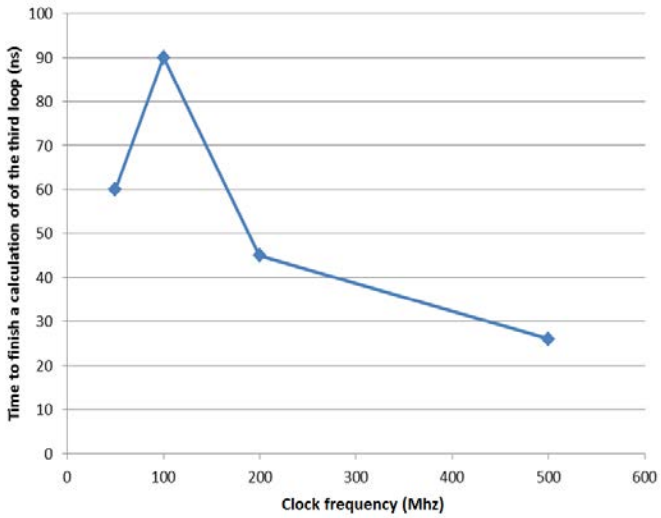


Figure 5.10 The third loop computation time vs clock frequency for U = 1 (= time to compute a radix-2 FFT)

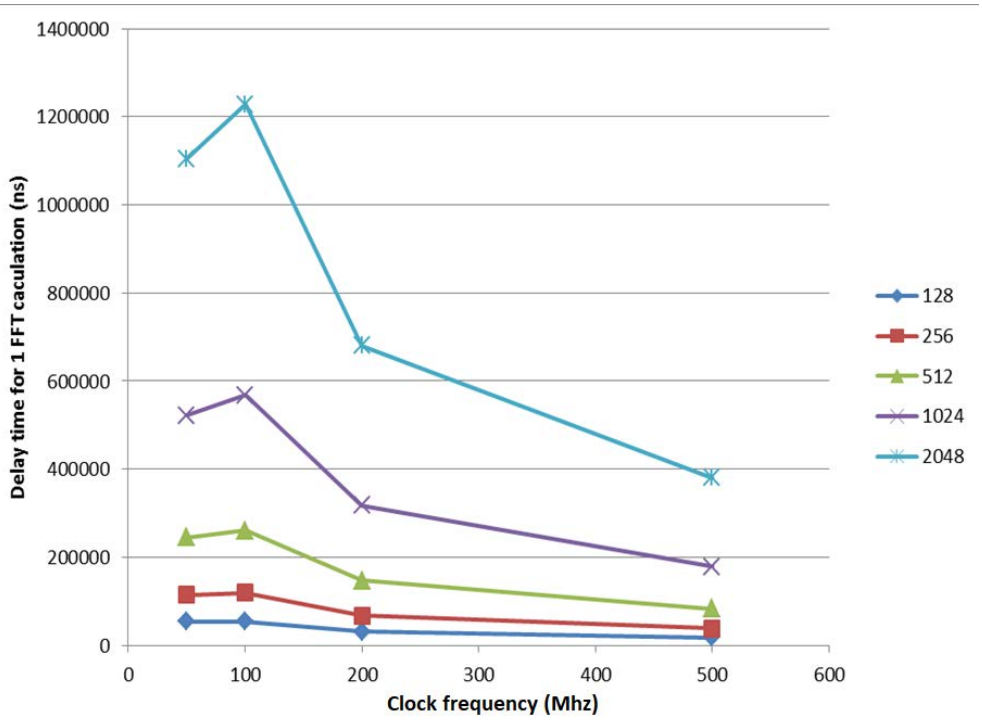


Figure 5.11 Time to compute 1 FFT vs clock frequency, U = 1

Figure 5.11 shows the delay to compute a FFT when the unrolling factor U = 1. The shapes of Figure 5.10 and Figure 5.11 are quite similar. Actually, we can say that the latency of Block FFTpow2() is mostly based on the latency of Loop 3, that is to say the radix-2 FFT computation.

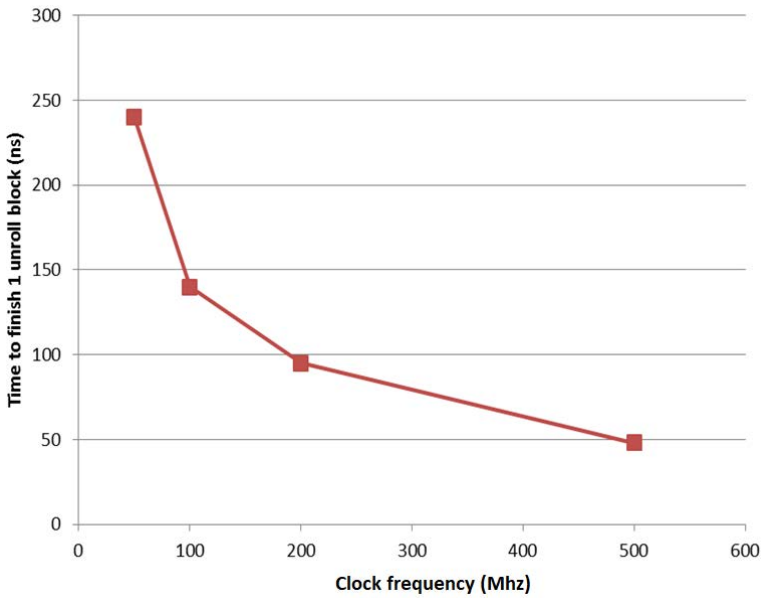


Figure 5.12 The third loop computation time vs clock frequency for U = 4

The same kind of experiments were done with unrolling factor U = 4. Figure 5.12 shows the delay to compute one iteration of Loop 3 with U = 4. The delay reduces when the frequency increases. Even if Figure 5.12 shape seems all right, it should be noticed that at frequency 50MHz, the unrolling factor does not work as expected. Actually, time to calculate four radix-2 in parallel (240 ns) is exactly four times the time to calculate radix-2 with U = 1 (60ns). This is not what is expected with unrolling. Figure 5.13 shows why the unrolling directive in this case is not efficient. Extra resources are used (due to unrolling) but Vivado HLS can not handle with memory accesses so that in practice the four multiplications are not executed in parallel. Fortunately, the unrolling directive works well with other frequencies (100, 200, 500 MHz): the delay of four radix_2 blocks in "parallel" thanks to unrolling factor U=4 is reduced even if it is still higher than one radix_2 block because of memory access bottleneck.

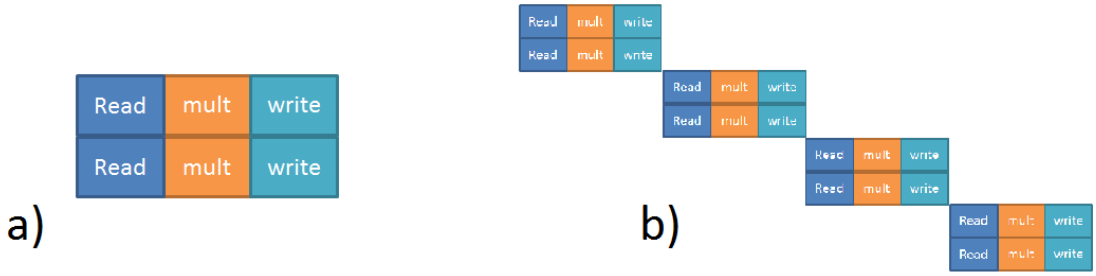


Figure 5.13 Timing chronogram of the third loop at 50 MHz: a) U = 1; b) U = 4.

Figure 5.14 shows the time to compute 1 FFT calculation for various frequencies when the unrolling factor equals to U =4. Shape is different compared to the one for U=1. However, for a particular value of U, one can notice the shape for the third loop computation time and the shape for the time to compute

1 FFT are similar. Actually, it still confirms latency of Block_FFTpow2() is related to the latency of third Loop () (*i.e.* the radix-2 FFT computation) which is quite reasonable.

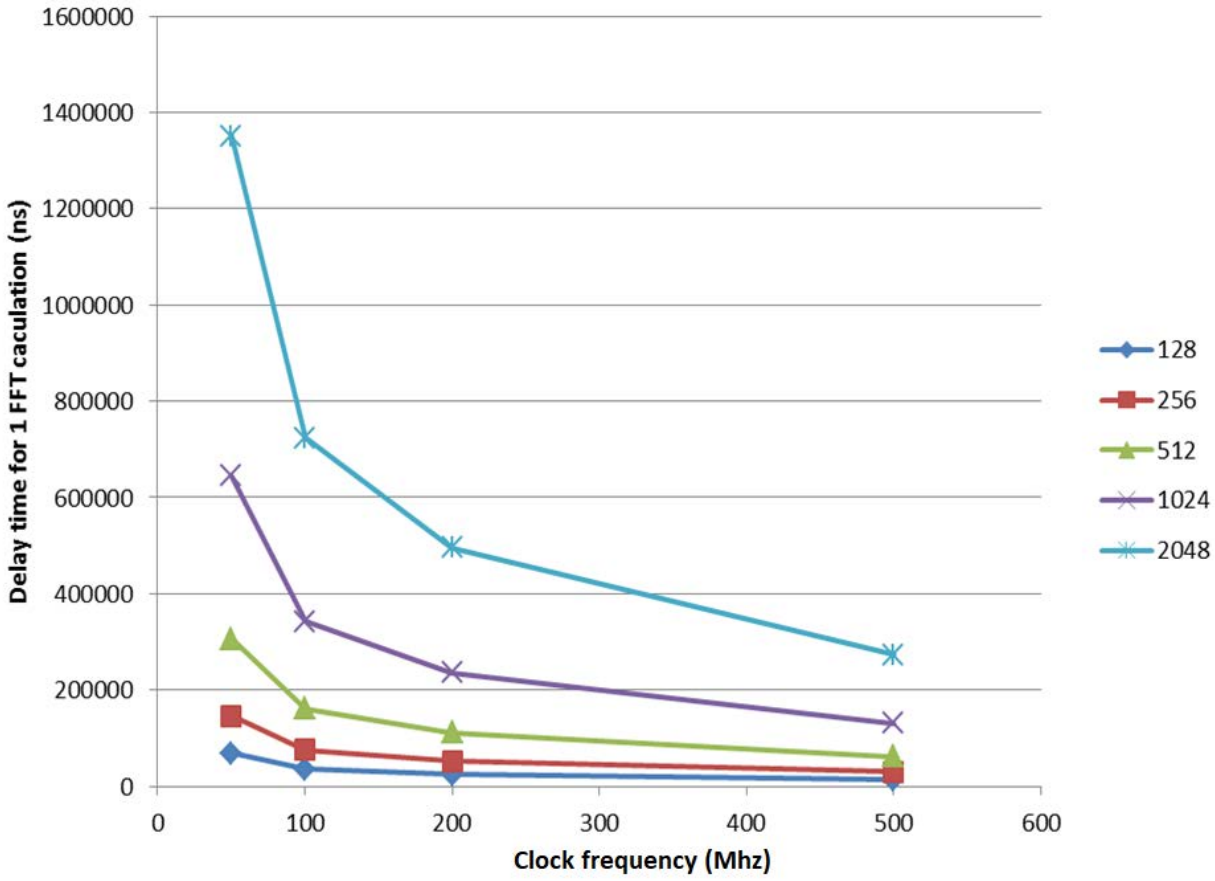


Figure 5.14 Time to compute 1 FFT vs frequency, U = 4

5.6. Conclusions

In this chapter, the design of a flexible FFT for LTE standard to be implemented on a FPGA target is investigated. At first, a native C program for FFT is adapted to Vivado HLS. Although the native C program cannot immediately run with Vivado HLS and needs to be updated, HLS reveals the great advantages to synthetize a high-level based language program into a RTL design. In this chapter, three different ways to implement a reconfigurable block onto a FPGA are also presented. Each of them has its own various advantages and disadvantages. A DSE of the flexible FFT is carried on. By using a HLS tool, DSE can be done quite quickly. However, this case study shows that the designer needs to have few knowledge in hardware design to understand some results and to use HLS tools efficiently. Optimizing the design is sometimes tricky. Actually, these experiments show that although HLS is basically a very useful tool, hardware knowledge is still required for better utilization.

Chapter 6.Designing a flexible FFT for LTE standard as a use case

6.1. Introduction

In this chapter, the implementation of a flexible FFT is proposed. We apply our design flow based on high-level synthesis for the design of two flexible FFT components. Six LTE-based FFT sizes are investigated: {128, 256, 512, 1024, 1536, 2048}. To this aim, we take advantage of the 3 reconfiguration approaches presented in Chapter 4 (software reconfiguration, hardware reconfiguration and algorithmic reconfiguration). A demonstration platform with run-time reconfiguration capabilities for experimenting a LTE-like waveform using one of the two flexible FFT components is also presented.

6.2. Proposed Flexible FFT Implementations for LTE Standard

Based on Chapter 5's results, the implementation of a flexible FFT for LTE standard is proposed and implemented on a Xilinx ML 605 board. Target frequency is 100 MHz. To achieve full OFDM-like waveforms, addition blocks are associated to the flexible FFT: a generator of random binary sequence and an X-QAM modulator. The additional blocks used to manage the reconfigurations for both software and hardware approaches will be introduced. The system is built as shown in Figure 6.1 and will be discussed latter in this Chapter. It should be noticed that as we focus on the hardware implementation of the transmitter, IFFTs are implemented rather than FFTs. However, to be consistent with previous chapters, block names based on FFT (like `Block_FFTpow2()`, etc.) are still used in this section.

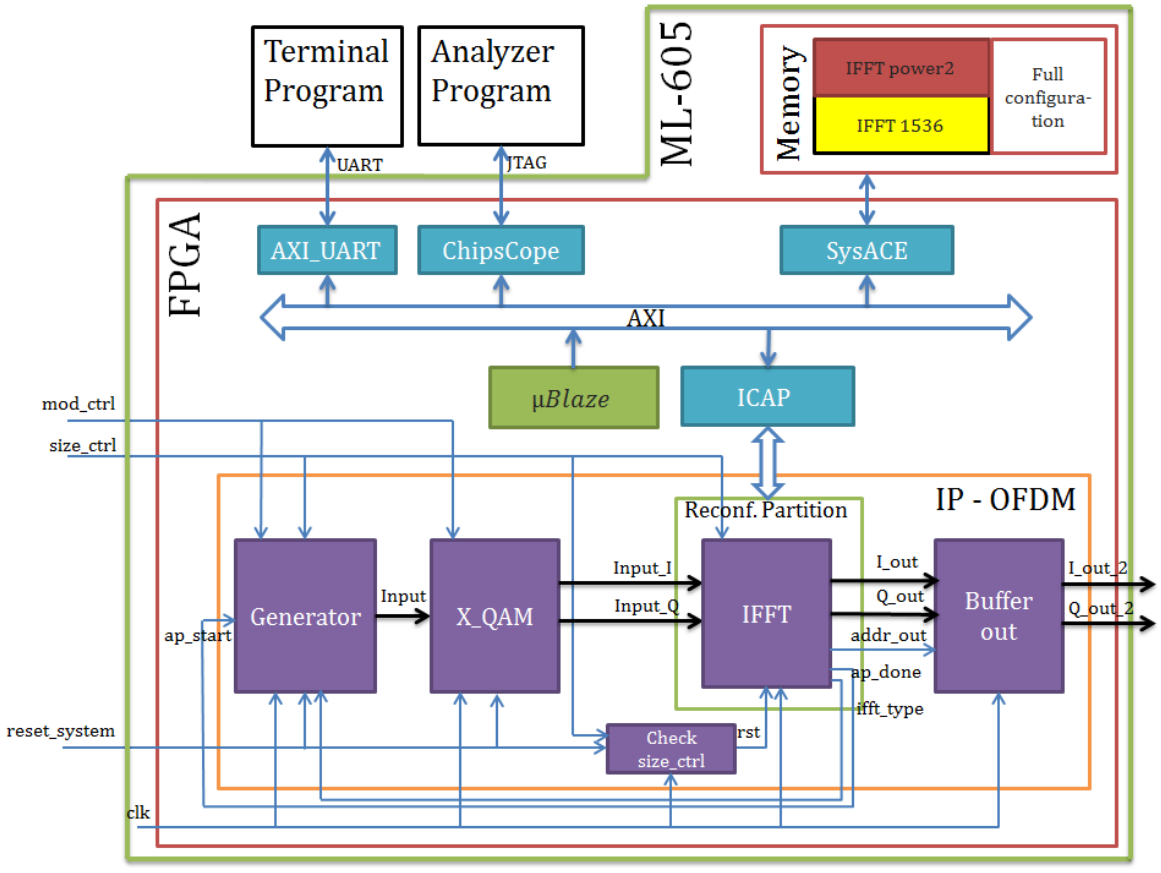


Figure 6.1 Overview of the architecture of the multi-mode FFT.

6.2.1. Generator

As presented in Figure 6.1, the random generator block produces the input data for the modulation (*i.e.* the data information that will be transmitted). This block has mainly five inputs and one output. The two inputs *reset_system* and *clk* are reset and clock common signals for all the system. The inputs *size_ctrl* and *mod_ctrl* are external control signals which are set by the user. *size_ctrl* is a 3-bit input that allows setting the six different IFFT sizes {128, 256, 512, 1024, 1536, 2048}. In the same way, *mod_ctrl* is a 2-bit signal that sets the used modulation among {BPSK, QPSK, 16-QAM, 64-QAM}. *ap_start* is a feedback signal from the reconfigurable IFFT processing block. It gives the information the IFFT processing is finished or not. If an IFFT computation is completed, the generator will generate the specific number of inputs knowing the IFFT size and modulation type (*mod_ctrl*). The output is a serial bit stream used by X_QAM modulator.

6.2.2. X-QAM Modulator

X_QAM block generates the complex I/Q symbols for the IFFT block among BPSK, QPSK and 16-QAM and 64-QAM modulations. It has four inputs: in addition to the reset and clock signals, the other inputs are *Input*, the stream of bits, and *mod_ctr* to control the modulation. The 4-bit I and Q outputs are the real part and image part of complex number. Their values range from -7 to 7. Such as the Generator, X-Qam Modulator block is implemented using HLS. Both blocks are low complexity blocks and could have been described at RTL level too with few lines of code. However, HLS makes it very easy to describe flexible (multi-mode) blocks as required for Generator and X-Qam Modulator.

For instance, Figure 6.2 shows the results of X-QAM modulator with ChipScope Pro in the case 64-QAM.

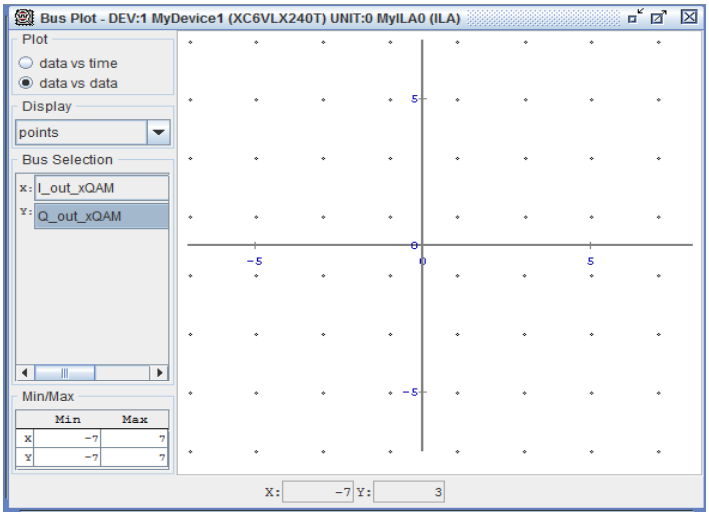


Figure 6.2 The 64-QAM constellation with ChipScope Pro.

6.2.3. Implementation for Software Reconfiguration

The multi-mode FFT with software reconfiguration is used in this implementation. Software reconfiguration is applied to design the FFT with 6 modes for LTE. Based on Algorithm 1, a `Multi_Mode_Block_LTE()` function is generated from the two functions `Block_FFTpow2()` and `Block_FFT1536()`. U = 4 is used for `Block_FFTpow2()`. As discussed in Chapter 5, the resources used by `Multi_Mode_Block LTE()` should be in this case almost the sum of the resources used by `Block_FFTpow2()` and `Block_FFT1536()` when synthesized separately. The architecture of multi-mode FFT with software reconfiguration is illustrated in Figure 6.3. Because there is no hardware reconfiguration, it is a simplified architecture compared to Figure 6.1.

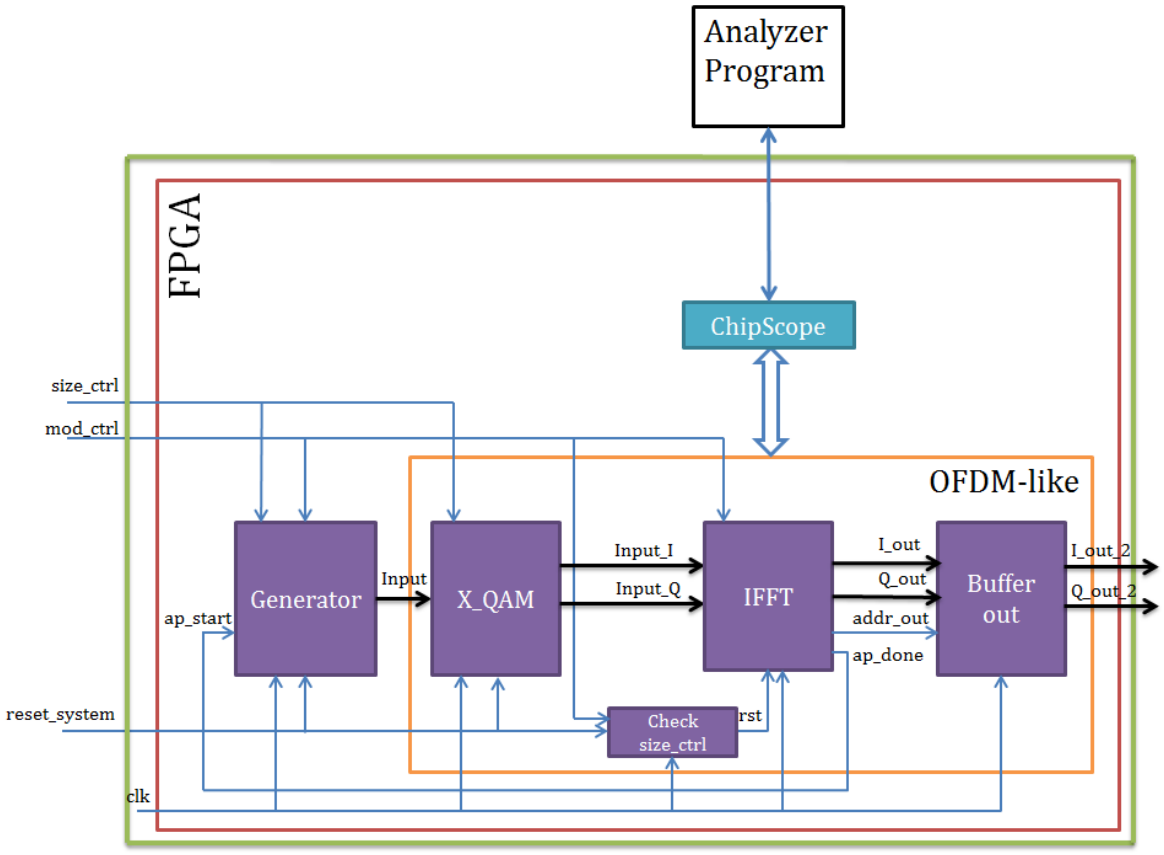


Figure 6.3 The architecture of the multi-mode FFT with software reconfiguration.

6.2.4. Implementation for Hardware Reconfiguration

In this case, the multi-mode FFT with hardware reconfiguration is used. Hardware reconfiguration is now applied on the two blocks `Block_FFTpow2()` and `Block_FFT1536()`. Thus, two partitions are first generated: one for the power-of-two point FFT only and one for the FFT 1536 only. Then, a partition is finally created for the DPR of the 2 FFTs.

The architecture of multi-mode FFT with hardware reconfiguration is illustrated in Figure 6.4. The Microblaze needs to be used to control the hardware reconfiguration process through AXI4 connection. A memory is used to keep the bitstream for full configuration of the FPGA for first time running as well as for the IFFT power 2 and IFFT 1536 . The user can control the hardware reconfiguration process via a terminal program. The terminal program sends the command to the Microblaze to determine the time to reconfigure. When reconfigurable partition IFFT is for IFFT 1536 configuration, *size_ctrl* is not considered. *size_ctrl* is used only when IFFT power 2 configuration runs.

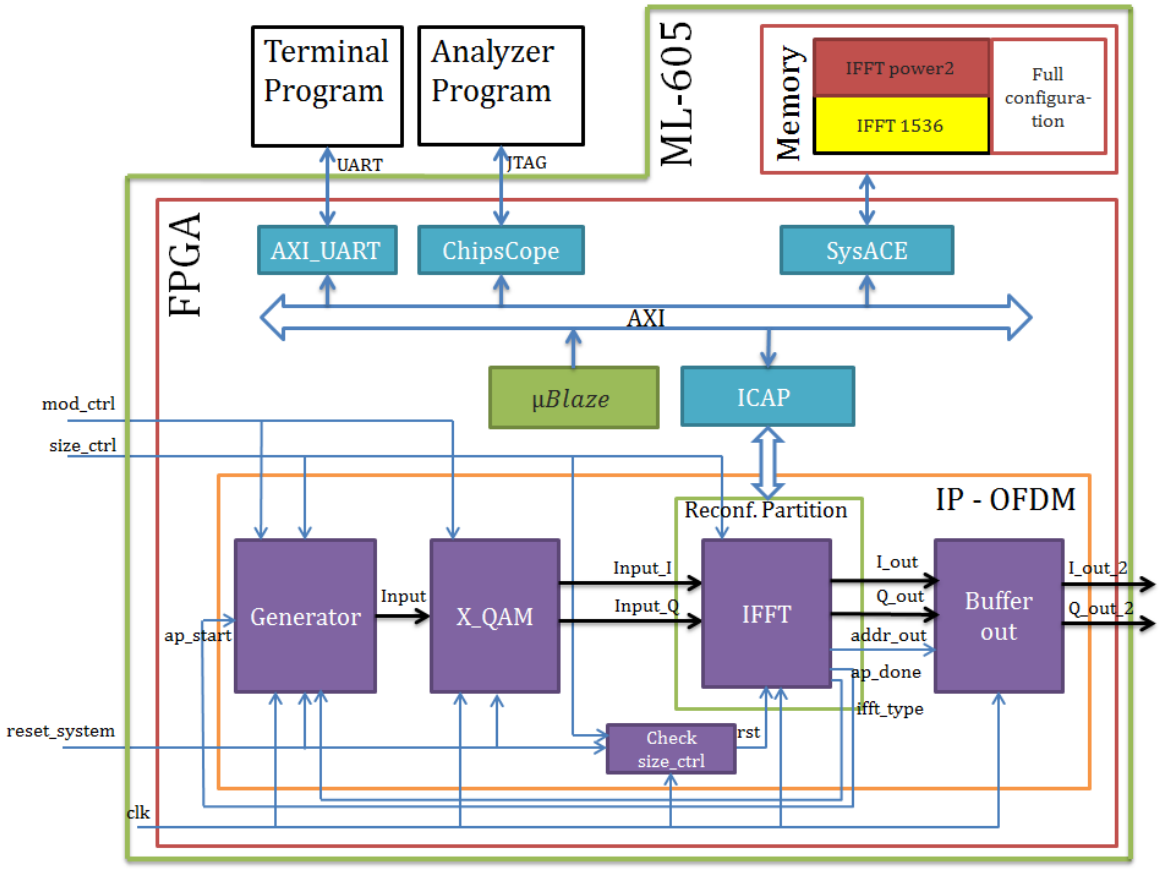


Figure 6.4 The architecture of the multi-mode FFT with hardware reconfiguration.

6.3. Results of the Proposed Flexible FFT Implementation

6.3.1. Implementation for Software Reconfiguration

Table 6.1 shows the synthesis results and latency (in number of clock cycles) of the three processing blocks for the software reconfiguration (`Block_FFTpow2()` is for the power of two point FFT alone, implementing algorithmic reconfiguration, `Block_FFT1536()` is for the 1536-point FFT alone, and `Multi_Mode_Block_LTE()` is for the multi-mode FFT {128, 256, 512, 1024, 1536, 2048} with software reconfiguration).

Processing block	Block_FFTpow2()	Block_FFT1536()	Multi_Mode_Block_LTE()
BRAM	12	14	26
DSP Slices	65	40	103
LUT	2553	3054	5256
FF	2497	2010	4299
Latency	Cf. Table 5.6-U=4	52198	Cf. Table 5.6-U=4/52198

Table 6.1 Performance of the multimode FFT for LTE standard with software reconfiguration

In this table we can see that the multi_Mode_Block_LTE approximately requires as many resources as the sum of the two other blocks. It is consistent with section 5.3.1's result.

6.3.2. Implementation for Hardware Reconfiguration

Table 6.2 shows the synthesis results for the hardware reconfiguration. When no hardware reconfiguration is targeted (resources needed in table 6.2), number of BRAMs is higher for function Block_FFT1536() than for Block_FFTpow2(). However this latter uses more DSP slices. After generating both related partitions, number of resources are always higher for Block_FFTpow2() one due to the shape of the configuration frames with this FPGA target. Unsurprisingly, when combining the 2 FFTs partitions into one partition to build the multi-mode FFT based on hardware reconfiguration, the resulting partition is based on the power-of-two point FFT's partition.

	Processing block: Resources needed		Partition: Resources used		
	Pow.-of-two FFT	FFT_1536	Pow.-of_two FFT	FFT_1536	FFT for LTE
BRAM	12	14	17	14	17
DSP	65	40	68	56	68
LUT	2553	3054	4080	3360	4080
FF	2497	2010	8160	6720	8160
Bitstream size	n/a	n/a	416016 Bytes	277344 Bytes	2 x 416016 Bytes
Reconf. Time	n/a	n/a	32.9 ms	21.96 ms	32.9 ms
Latency	Cf. Table 5.6-U=4	52198	Cf. Table 5.6-U=4	52198	Cf. Table 5.6-U=4/52198

Table 6.2 Performance of the multimode FFT for LTE standard with hardware reconfiguration

6.3.3. Comparisons

Compared with software reconfiguration, the multi-mode FFT based on hardware reconfiguration uses fewer resources (BRAM and DSP are the more costly resources in a FPGA). From the user point of view, when the FFT size has to be modified but is still a power of two, in both cases only one clock cycle is required to reconfigure. However, 32.9 ms are required to reconfigure when switching from a 1536-point FFT and a power-of-two point FFT (or vice versa) with hardware reconfiguration whereas only one clock cycle is required with software reconfiguration.

6.4. Demonstration of the FFT Implementation for LTE Standard

In this section, the implementation of the proposed IFFT architecture on the FPGA Xilinx board is presented. We use it as a demonstration platform.

6.4.1. Testbed Description

Virtex 6 ML605 evaluation board is a standard evaluation board so it does not embed any RF front-end. Thus, we decided to implement on the FPGA the baseband processing of the transmitter only (data generator, modulator, IFFT). The I and Q output signals are then simulated to the high band using a Python program and noise is also added (an additive white Gaussian noise in our case). The receiver (FFT, demodulator) is implemented as a Python program too to receive and analyze the signal. Chipscope is used to records all transmitted signals on the FPGA for comparisons with decoded signals on the receiver side. The illustration of the test bed is given in Figure 6.5. A photo of the testing environment is shown in Figure 6.6.

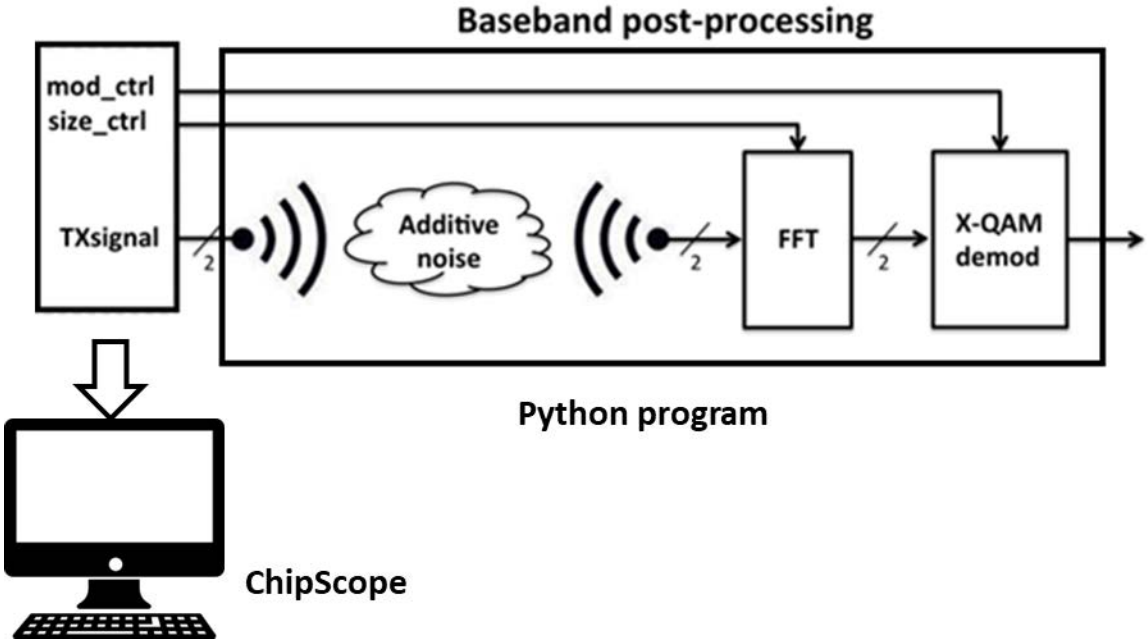


Figure 6.5 The testbed description.

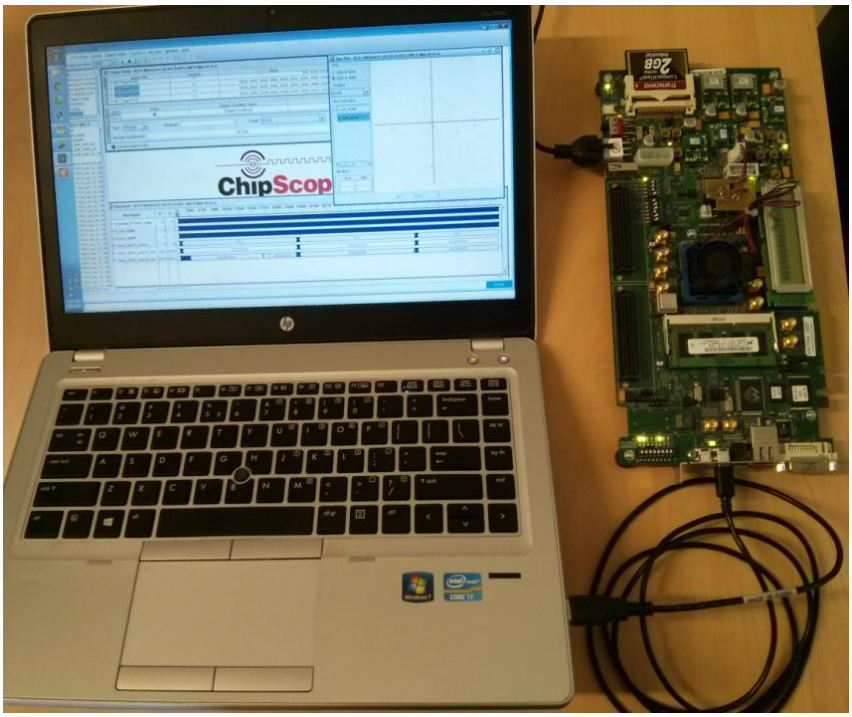


Figure 6.6 The demonstration platform.

6.4.2. Virtex 6 ML 605 Evaluation Board

ML 605 Evaluation board is a Xilinx Virtex 6 board. A we said, it uses a Virtex-6 xc6vlx240t FPGA that has capacity for hardware reconfiguration. Moreover, the board supports embedded processing with, among others, MicroBlaze (soft 32 bit RISC-based processor) and DDR3 memory. Except the front-ends, this board is a good platform to demonstrate our proposed design flow based on different approaches for reconfiguration that can be used in the context of a 3GPP-LTE waveform and OFDM symbols.

6.4.3. Required Software Development Tools

To build our testbed we use development tools provided by Xilinx. Because partial reconfiguration is not natively integrated in Xilinx framework, main tools we use are listed here (we have already presented most of these tools in Chapter 3 so only few comments are given here. See also section 4.3.2 for additional details):

Vivado HLS: to convert the high-level specification to RTL.

Integrated Synthesis Environment (ISE): This most common Xilinx tool is used for synthesis and analysis of HDL designs. It also provides a simulation environment (ISim) or can be combined with ModelSim.

Embedded Development Kit: this is an integrated development environment for the design of embedded systems. It enables to quickly connect the different modules in Xilinx ecosystem such as IPs, bus, peripherals and Microblaze. This design kit also includes some other tools. In our case, we also use Xilinx Platform Studio (XPS) tool suite and Software Development Kit (SDK) for MicroBlaze. Basically, XPS

allows the designer to create his own IP and configures the embedded system architecture while SDK provides a GNU C/C++ design environment for microprocessors.

PlanAhead: PlanAhead is a tool to support hardware reconfiguration. The designer can choose the specific area for the reconfigurable partition, check the compatibility of reconfigurable partitions, etc.

6.4.4. Experiments

The demonstration platform implements the hardware reconfiguration, *i.e.* one reconfigurable partition for the algorithmic reconfiguration based power-of-two point FFT and one partition for the 1536-point FFT. Using the demonstration platform, we checked every FFT size and every modulation to validate the compliance of the OFDM like transceiver with the standard.

One kind of results the demo can provide is shown in Figure 6.7. In this case, the control signals set a 16-QAM modulation and a 512 FFT. Figure 6.7 left shows the received spectrum that was recovered from the transmitted signal. Figure 6.7 right shows the received constellation with 16 clear positions (signal to noise ratio was quite high in the case of this experiment).

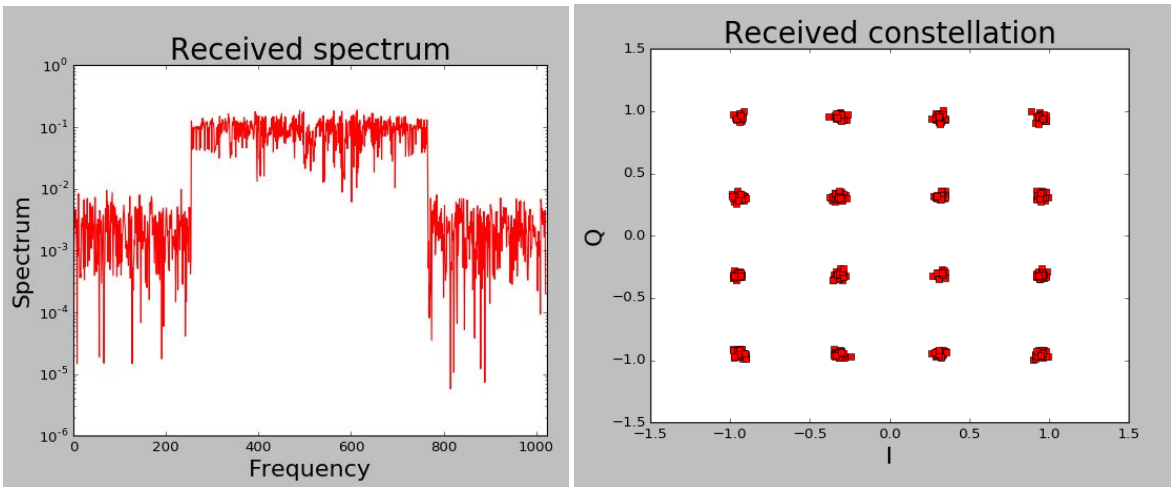


Figure 6.7 Received spectrum and 16-QAM received constellation for an AWGN channel.

6.5. Conclusions

To demonstrate our proposed design flow based on high-level synthesis for the design of a FPGA-based software defined radio, we have implemented using Vivado HLS a flexible FFT for LTE standard as a case study. FFT sizes are {128, 256, 512, 1024, 1536, 2048}. Two flexible FFT components were designed. One relies on software reconfiguration and combines an algorithmic reconfiguration based power-of-two point FFT and a 1536-point FFT. The other one is based on hardware reconfiguration and combines the algorithmic reconfiguration based power-of-two point FFT again but as a reconfigurable partition and a 1536-point FFT for the second reconfigurable partition. Even if the results show the user shall choose between reconfiguration time and FPGA resource's utilization, the experiments enable validating our approach. We also developed a demonstration platform supporting run-time reconfiguration capabilities and implementing a LTE waveform and OFDM symbols. The transmitter was implemented

onto a ML 605 evaluation board and rely on the hardware reconfiguration based flexible FFT (IFFT actually) whereas the front-ends, the channel and the receiver were pre and post-processed using a Python program.

Chapter 7.Conclusions and Perspectives

Context of our work

Software defined radio (SDR) is a promising technology to tackle flexibility requirements of new generations of communication standards. It can be easily reprogrammed at a software level to implement different waveforms. When relying on a software-based technology such as microprocessors, this approach is clearly flexible and quite easy to design. However, it usually provides low computing capability and therefore low throughput performance. To tackle this issue, FPGA technology turns out to be a good alternative for implementing SDRs. Indeed, FPGAs have both high computing power and reconfiguration capacity. Thus, including FPGAs into the SDR concept may allow to support more waveforms with more strict requirements than a processor-based approach. However, main drawbacks of FPGA design are the level of the input description language that basically needs to be the hardware level, and, the reconfiguration time that may exceed run-time requirements if the complete FPGA is reconfigured. To overcome these issues, this PhD thesis proposes a design methodology that leverages both HLS tools and dynamic reconfiguration. The proposed methodology is a guideline to completely build a flexible radio for FPGA-based SDR, which can be reconfigured at run-time.

Summary of the report and conclusions

Except the introduction which provides a general view of our work, this report is divided into two parts: background and contribution.

The background part is composed of two chapters (chapters 2 and 3) and provides underlying information related to our work. First, general knowledge on digital radio systems, cognitive radio and SDR have been provided focusing on the flexibility required by new generation of waveforms such as the 3GPP-LTE standard. Then implementation issues have been discussed by introducing SDR platforms and FPGA-based SDR. After that, we discussed two main techniques/technologies that are used in this work: high-level synthesis and FPGA hardware reconfiguration. Main HLS fundamentals have been presented with a specific focus on HLS tools and their optimization techniques. Finally, hardware reconfiguration on FPGAs and the related design flow have been discussed.

The contribution part is composed of chapters 4, 5 and 6 which details our propositions and the results we obtained in order to validate theses propositions. First, the reconfiguration approaches have been classified in three types (software, hardware and algorithmic reconfiguration) and a design flow has been proposed to explore the use of theses reconfigurations according to performance metrics in terms of used resources, reconfiguration time and time to design. To implement a flexible radio, a FPGA-based system architecture has been introduced as well as the design steps for verification and validation of the system. To evaluate the proposed approaches, a flexible radio for LTE standard has been designed using native C codes as inputs. C-code rewriting, HLS-based design optimizations and design space exploration have been discussed. The three reconfigurable approaches are used to achieve flexibility. Finally, a full LTE-like system has been designed and implemented on a Virtex 6 FPGA board. The different modes of

the LTE standard can operate on the platform by modifying both the FFT size and the modulation. Flexibility is achieved by combining both HLS optimizations and dynamic hardware reconfiguration.

Designing and implementing a radio system is always a challenging task. If this task can be quite easily performed when targeting processor-based SDR systems, it remains a tough job when addressing FPGA-based SDR systems. Our goal was to propose a methodology for the implementation of run-time reconfiguration in the context of FPGA-based SDR. The generic HLS-based design flow for flexible radio on FPGAs we propose is expected to help the designer to shorten the design time. It allows the exploration between dynamic partial reconfiguration and multi-mode design using control signals.

HLS provides a kind of shortcut to generate RTL descriptions from high-level specifications. It has a huge potential to change the traditional way hardware systems are built. Indeed, HLS tools can reduce the numerous amount of work a hardware designer/team has to do and then accelerates design time. However, besides these advantages, HLS tools need some improvements to fulfill themselves. For instance, enabling efficient use of .tcl commands for the user is one of them to ease and automate design space exploration (DSE). Moreover, each HLS tool currently has its own specific rules to be applied to write a high-level specification and to make it able to be synthesized and optimized. These rules are more or less very different among HLS tools. HLS tools also need to share the resources more efficiently. For example, Vivado HLS does not take advantage when two different modules do not operate at the same time and thus the tool does not share the resources when implementing these modules. We also remarked in some cases, for data with no dependencies, when applying parallelization techniques so that the tool allocates more resources, there is no performance improvement but the tool does not detect and does not give advices for the user. Having that said, in the near future, we believe HLS will step-by-step replace the traditional way to design a hardware system.

Besides HLS, DPR is an important concern of our research. DPR is an interesting feature of FPGA devices. However, it is not very used by many FPGA engineers. There are several reasons for this fact. Two of them are probably designers are not used to DPR and the difficulty to use the related tools. However, things are changing along with the improvement of DPR tools and many FPGA engineers are now paying attention to such an approach. In the IoT era for which the requirements in energy, weight and size of the devices are very strict, DPR is really a good candidate. It provides the capability to fit more logic into an existing device and makes the system more flexible to be upgraded. Moreover, by reducing redundancy, it also reduces power consumption. One drawback of DPR is the time for reconfiguration. There are some techniques to reduce this time but they usually limit the range of DPR usability. However, the potential of DPR is huge and along with more user-friendly tools soon, we believe DPR will gradually convince the users to change their design methodology.

Perspectives

In this PhD thesis, we have proposed a design flow for FPGA-based SDR and different reconfiguration methods have been analysed. However, many works are still pending and perspectives arise from this thesis. This section presents the issues that we want to highlight, from short-term to longer term.

Implementation of other processing blocks: In this work, we focused on the design of a flexible FFT as a use case. However, the proposed design flow can be also evaluated with other functional blocks. For example other modulations, filters, channel coding, source coding, etc. can be implemented in the FPGA-based SDR. The reconfiguration of full waveforms can also envisaged by adding into the top of the design flow a primary search of common functional blocks.

Complete system FPGA implementation for LTE standard: The current version of the LTE-like system performs the baseband processing of the transmitter on the FPGA only. The rest of the system is implemented as a software program in Python. To fulfill the implementation, the receiver can be designed using our approach too and a radio front-end can be added to the system. In this way a complete FPGA-based system can be evaluated.

Design framework target technology: Currently, our design framework is dedicated to Xilinx FPGAs and thus it is based on Xilinx tools. However, the design flow we propose is not dedicated to particular FPGA targets[10]/tools so that it could also be applied for instance to Altera FPGAs using Altera tools.

Energy consumption evaluation: Energy consumption is an issue for SDR due to devices that sometimes need to operate during very long times. In this report, we did not discuss about power and energy consumption (for example, dynamic reconfiguration requires extra energy during reconfiguration step). Main reason was we did not succeed in solving power consumption measurement issues (in practice we were not able to measure particular block's power consumption but overall board mean power consumption only). We assume energy consumption for each component should also be provided to help the designer making design choices during DSE.

Automation: Currently, the design framework we use based on the proposed design flow is mostly unautomated. Automating the design framework so that the user does not need to manually perform tool invocation, set optimization parameters/directives, etc. any more should be a real extra value for a design team.

[10] assuming they implement DPR

Publication

Mai-Thanh Tran, Matthieu Gautier, and Emmanuel Casseau, "On the FPGA-based implementation of a flexible waveform from a high-level description: Application to LTE FFT case study", *EAI International Conference on Cognitive Radio Oriented Wireless Networks (Crowncom)*, Grenoble, France, May 30 - June 1, pp. 545-557, 2016.

Mai-Thanh Tran, Emmanuel Casseau, Matthieu Gautier, "FPGA-based implementation of a flexible FFT dedicated to LTE standard", *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Demo Night, Rennes, France, pp.1-2, October 12-14, 2016.

Bibliography

[1] J. Gubbi, R. Buyya, S. Marusic and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future direction," *Future generation computer systems 29.7,* pp. 1645-1660, 2013.

[2] F. Wortmann and K. Flüchter, "Internet of things," *Business & Information Systems Engineering 57.3,* pp. 221-224, 2015.

[3] G. P. Fettweis, "The tactile internet: Applications and challenges," *IEEE Vehicular Technology Magazine 9, no. 1,* pp. 64-70, 2014.

[4] J. Mitola and G. Q. Maguire, "Cognitive radio: making software radios more personal," *IEEE personal communications 6.4,* pp. 13-18, 1999.

[5] H. F. Fitzek and M. D. Katz, "Cooperation in wireless networks: principles and applications," New York: Springer, 2006.

[6] H. Congzheng, T. Harrold, S. Armour, I. Krikidis, S. Videv, P. M. Grant and H. Haas, "Green radio: radio techniques to enable energy-efficient wireless networks," *IEEE communications magazine 49.6,* 2011.

[7] J. Mitola, "Software radio," *John Wiley & Sons, Inc,* 2003.

[8] J. Mitola, "The software radio architecture," *IEEE Communications magazine 33.5,* pp. 26-38, 1995.

[9] M. Cummings and S. Haruyama, "FPGA in the software radio," *IEEE communications Magazine 37, no. 2,* pp. 108-112, 1999.

[10] M. G. O. S. G.-S. Ouedraogo, " A frame-based Domain-Specific Language for rapid prototyping of FPGA-based Software-Defined Radios," *EURASIP Journal on Advances in Signal Processing,* p. 164, November 2014.

[11] M. G. O. S. G. S. Ouedraogo, "Frame-based Modeling for Automatic Synthesis of FPGA-Software Defined Radio," *IEEE International Conference on Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM),* June 2014.

[12] A. Massouri and T. Risset, "FPGA-based Implementation of Multiple PHY Layers of IEEE 802.15. 4 Targeting SDR Platform," *SDR-WInnComm. Wireless Innovation Forum,* 2014.

[13] E. Lemoine and D. Merceron, "Run time reconfiguration of FPGA for scanning genomic databases," *FPGAs for Custom Computing Machines,* pp. 90-98, 1995.

[14] T. T. Ha, "Theory and design of digital communication systems," *Cambridge University Press,* 2010.

[15] E. A. Lee and D. G. Messerschmitt, "Digital communication," *Springer Science & Business Media,* 2012.

[16] A. Das, in *Digital Communication: Principles and system modelling*, Springer Science & Business Media, 2010.

[17] D. Wetteroth, "OSI reference model for telecommunications," *McGraw-Hill Professional,* 2001.

[18] S. G. Wilson, "Digital modulation and coding," *Prentice-Hall, Inc.,* 1995.

[19] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE transactions on Information Theory 13, no. 2,* pp. 260-269, 1967.

[20] G. Kalivas, in *Digital radio system design*, John Wiley & Sons, 2009.

[21] T. S. Rappaport, "Wireless communications: principles and practice," *Vol. 2. New Jersey: Prentice Hall PTR,* 1996.

[22] J. Mitola and G. Q. Maguire, "Cognitive radio: making software radios more personal," *IEEE personal communications 6.4,* pp. 13-18, 1999.

[23] Z. Xuping and P. Jianguo, "Energy-detection based spectrum sensing for cognitive radio," *Wireless, Mobile and Sensor Networks IET Conference,* pp. 944-947, 2007.

[24] V. Turunen, M. Kosunen, A. Huttunen, S. Kallioinen, P. Ikonen, A. Parssinen and J. Ryynanen, "Implementation of cyclostationary feature detector for cognitive radios," *Cognitive Radio Oriented Wireless Networks and Communications,* 2009.

[25] M. Gautier, M. Laugeois and P. Hostiou, "Cyclostationarity detection of DVB-T Signal: testbed and measurement," *The First International Conference on Advances in Cognitive Radio,* 2011.

[26] C. Kuo and J. Wong, "Multi-standard DSP based wireless system," *Signal Processing Proceedings,* pp. 1712-1728, 1998.

[27] S. T. 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.

[28] E. Grayver, "Implementing software defined radio," *Springer Science & Business Media,* 2012.

[29] J. Zyren and W. McCoy, "Overview of the 3GPP long term evolution physical layer," in *Freescale Semiconductor*, 2007.

[30] "The LTE standard: Developed by a global community to support paired and unpaired spectrum deployments," 2014. [Online]. Available: http://www.signalsresearch.com.

[31] C. Johnson, "LTE in Bullets," Createspace Independent Pub, 2012.

[32] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation 19, no. 90 ,* pp. 297-301, 1965.

[33] D. L. Jones, "OpenStax CNX," Decimation-in-time (DIT) Radix-2 FFT, 15 9 2006 . [Online]. Available: http://cnx.org/contents/ce67266a-1851-47e4-8bfc-82eb447212b4@7..

[34] M. Dardaillon, K. Marquet, T. Risset and A. Scherrer, "Software defined radio architecture survey for cognitive testbeds," in *Wireless Communications and Mobile Computing Conference (IWCMC)*, 2012.

[35] M. Ettus, "Universal software radio peripheral," 2009. [Online]. Available: https://www.ettus.com.

[36] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang and G. M. Voelker, "Sora: high-performance software radio using general-purpose multi-core processors," *Communications of the ACM 54.1,* pp. 99-107, 2011.

[37] T. Ulversoy, "Software defined radio: Challenges and opportunities," *IEEE Communications Surveys & Tutorials 12, no. 4,* pp. 531-550, 2010.

[38] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Petty, R. Rajbanshi and T. Newman, "Kuar: A flexible software-defined radio development platform," *New Frontiers in Dynamic Spectrum Access Networks, 2nd IEEE International Symposium,* pp. 428-439, 2007.

[39] P.-H. Horrein, C. Hennebert and F. Pétrot, "Integration of GPU computing in a software radio environment," *Journal of Signal Processing Systems 69.,* pp. 55-65, 2012.

[40] B. Bougard, B. D. Sutter, D. Verkest, L. V. d. Perre and R. Lauwereins, "A coarse-grained array accelerator for software-defined radio baseband processing," *IEEE micro 28.4,* pp. 41-50, 2008.

[41] O. Arnold, E. Matus, B. Noethen, M. Winter, T. Limberg and G. Fettweis, "Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs," *ACM Transactions on Embedded Computing Systems (TECS) 13.3s,* pp. 107:1-107:24, 2014.

[42] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge and M. J. Meeuwsen, "A 167-processor computational platform in 65 nm CMOS," *IEEE Journal of Solid-State Circuits 44.4,* pp. 1130-1144, 2009.

[43] F. Clermidy, R. Lemaire, X. Popon, D. Ktenas and Y. Thonnar, "An open and reconfigurable platform for 4g telecommunication: Concepts and application," *Digital System Design, Architectures, Methods and Tools, 12th Euromicro Conference on. IEEE,* 2009.

[44] C. Jalier, D. Lattard, A. A. Jerraya, G. Sassatelli, P. Benoit and L. Torres, "Heterogeneous vs homogeneous MPSoC approaches for a mobile LTE modem," *Proceedings of the Conference on Design, Automation and Test in Europe. European Design and Automation Association,* 2010.

[45] C. Schmidt-Knorreck, R. Pacalet, A. Minwegen, U. Deidersen, T. Kempf, R. Knopp and G. Ascheid, "Flexible front-end processing for software defined radio applications using application specific instruction-set processors," *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on. IEEE,* p. 201, 2012.

[46] "OAI Open Air Interface," 2014. [Online]. Available: http://www.openairinterface.org,.

[47] Rice University, "WARP," 2016. [Online]. Available: http://warp.rice.edu.

[48] "Nutaq," 2017. [Online]. Available: https://www.nutaq.com/.

[49] Z. Miljanic, I. Seskar, K. Le and D. Raychaudhuri, "The WINLAB network centric cognitive radio hardware platform—WiNC2R," *Mobile Networks and Applications 13.,* pp. 533-541, 2008.

[50] A. a. T. R. Massouri, "FPGA-based Implementation of Multiple PHY Layers of IEEE 802.15. 4 Targeting SDR Platform," *SDR-WInnComm. Wireless Innovation Forum,* 2014.

[51] C. S. Olga Zlydareva, "Multi-standard wimax/umts system framework based on sdr," in *Aerospace Conference , IEEE*, 2008.

[52] K. Papadimitriou, A. Dollas and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost mode," *ACM Transactions on Reconfigurable Technology and Systems (TRETS,* pp. 36:1-36:24, 2011.

[53] J.-P. Delahaye, G. Gogniat, C. Roland and P. Bomel, "Software radio and dynamic reconfiguration on a DSP/FPGA platform," *Frequenz 58.5-6,* pp. 152-159, 2004.

[54] M. Fingeroff, in *High-level synthesis: blue book*, Xlibris Corporation, 2010.

[55] D. D. Gajski, in *High—Level Synthesis: Introduction to Chip and System Design*, Springer Science & Business Media, 2012.

[56] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers 26.4,* pp. 18-25, 2009.

[57] K. Wakabayashi, "C-based behavioral synthesis and verification analysis on industrial design examples," *Proceedings of the 2004 Asia and South Pacific Design Automation Conference. IEEE Press,* 2004.

[58] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis and Y. T. Chen, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35.1,* pp. 1591-1604, 2016.

[59] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on. IEEE,* 2013.

[60] R. Nane, V. M. Sima, B. Olivier, R. Meeuws, Y. Yankova and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," *Field Programmable Logic and Applications (FPL), 22nd International Conference on. IEEE,* 2012.

[61] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays. ACM,* 2011.

[62] P. Coussy, G. Lhairech-Lebreton, D. Heller and E. Martin, "GAUT–a free and open source high-level synthesis tool," *IEEE DATE,* 2010.

[63] Xilinx, "Vivado HLS," [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[64] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19.12,* pp. 1507-1522, 2000.

[65] Calypto Design Systems, "Catapult-C," [Online]. Available: https://www.mentor.com/hls-lp/.

[66] Xilinx, in *Vivado Design Suite User Guide - High Level Synthesis*, 2016.

[67] B. Rousseau, P. Manet, T. Delavallée, I. Loiselle and J. D. Legat, "Dynamically reconfigurable architectures for software-defined radio in professional electronic applications," in *Design technology for heterogeneous embedded systems*, Springer, 2012, pp. 437-445.

[68] C. Bolchini, A. Miele and C. Sandionigi, "A novel design methodology for implementing reliability-aware systems on SRAM-based FPGAs," *IEEE Transactions on Computers 60, no. 12 ,* pp. 1744-1758, 2011.

[69] D. Dye, "Partial reconfiguration of Xilinx FPGAs using ISE," in *Xilinx white paper WP374*, 2012.

[70] "The free and open software radio ecosystem," GNU Radio, [Online]. Available: https://www.gnuradio.org/.

[71] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid and K. Flautner, "SPEX: A programming language for software defined radio," *Software Defined Radio Technical Conference and Product Exposition,* 2006.

[72] A. Gelonch, X. Revès, V. Marojevik and R. Ferrús., "P-HAL: a middleware for SDR applications," *SDR Forum Technical Conference,* 2005.

[73] G. Jianxin, Y. Xiaohui, G. Jun and L. Quan, "The software communication architecture specification: evolution and trends," *Computational Intelligence and Industrial Applications,* pp. 341-344, 2009.

[74] G. S. Ouedraogo, M. Gautier and O. Sentieys, "A frame-based domain-specific language for rapid prototyping of FPGA-based software-defined radios," *EURASIP Journal on Advances in Signal Processing,* p. 164, 2014.

[75] E. Casseau and B. L. Gal, "Design of multi-mode application-specific cores based on high-level synthesis," *INTEGRATION, the VLSI journal 45,* pp. 9-21, 2012.

[76] E. Lemoine and D. Merceron, "Run time reconfiguration of FPGA for scanning genomic databases," *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 90-98, 1995.

[77] "Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28nm FPGAs," *Altera white paper, WP-01137-1.0, www.altera.com.*

[78] P. Mankar, L. P. Thakare and A. Y. Deshmukh, "Design of Reconfigurable FFT/IFFT for Wireless Application," *International Journal of Engineering Research and General Science Volume 2, Issue 3,* pp. 292-297, 2014.

[79] E. Oruklu, R. Hanley, S. Aslan, C. Desmouliers, F. M. VallinaOruklu and J. Saniie, "System-on-chip design using high-level synthesis tools," *Circuits and Systems 3, no. 01,* 2012.

[80] M. A. Mohamed, "FPGA synthesis of VHDL OFDM system," *Wireless personal communications,* pp. 1-25, 2013.

[81] Xilinx, "Xilinx UG369 Virtex-6 FPGA DSP48E1 Slice, User Guide," 2011.

[82] K. Papadimitriou, "Partial Reconfiguration Cost Calculator," Microprocessor and Hardware Laboratory, Technical University of Crete, [Online]. Available: http://user.sisc.tuc.gr/~kpapadimitriou/prcc.html.

[83] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation 19.90,* pp. 297-301, 1965.

[84] Freescale Semiconductor Incorporated, "Software Optimization of DFTs and IDFTs Using the

StarCore SC3850 DSP Core," in *Application Note AN3980*, 2009.

[85] H. Congzheng, T. Harrold, S. Armour, I. Krikidis, S. Videv, P. M. Grant and H. Haas, "Green radio: radio techniques to enable energy-efficient wireless networks," *IEEE communications magazine 49.6,* pp. 46-54, 2011.

[86] B. Rousseau, P. Manet, T. Delavallée, I. Loiselle and J. D. Legat, "Dynamically reconfigurable architectures for software-defined radio in professional electrionic applications," in *Design technology for heterogeneous embedded systems*, Springer, 2012, p. 445.

[87] O. Arnold, E. Matus, B. Noethen, M. Winter, T. Limberg and G. Fettweis, "Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs," *ACM Transactions on Embedded Computing Systems (TECS) 13.3s,* p. 107, 2014.

[88] K. Papadimitriou, A. Dollas and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost mode," *ACM Transactions on Reconfigurable Technology and Systems (TRETS,* p. 36, 2011.

[89] Ericsson; Qualcomm, "The LTE standard: Developed by a global community to support paired and unpaired spectrum deployments," Signals Research Group, 2014.

Synthèse matérielle d'une radio flexible et reconfigurable depuis un langage de haut niveau

Mai-Thanh TRAN

La radio logicielle (SDR - Software Defined Radio) est une technologie prometteuse pour répondre aux exigences de flexibilité des nouvelles générations de standards de communication. Elle peut être facilement reprogrammée au niveau logiciel pour implémenter différentes formes d'onde. En s'appuyant sur une technologie dite logicielle telle que les microprocesseurs, cette approche est particulièrement flexible et assez facile à mettre en œuvre. Cependant, ce type de technologie conduit généralement à une faible capacité de calcul et, par conséquent, à des débit faibles. Pour résoudre ce problème, la technologie FPGA s'avère être une bonne alternative pour la mise en œuvre de la radio logicielle.
En effet, les FPGAs offrent une puissance de calcul élevée et peuvent être reconfigurés. Ainsi, inclure des FPGAs dans le concept de radio logicielle peut permettre de prendre en charge plus de formes d'onde avec des exigences plus strictes qu'une approche basée sur la technologie logicielle.
Cependant, les principaux inconvénients d'une conception à base de FPGAs sont le niveau du langage de description d'entrée qui doit typiquement être le niveau matériel, et le temps de reconfiguration qui peut dépasser les exigences d'exécution si le FPGA est entièrement reconfiguré. Une solution possible pour surmonter ce problème est l'utilisation de la synthèse de haut niveau (HLS pour High Level Synthesis), technique récemment intégrée dans les environnements de conception de FPGA et qui permet de générer des descriptions RTL à partir de spécifications haut niveau. Cette thèse propose une méthodologie de conception qui exploite à la fois la synthèse de haut niveau et la reconfiguration dynamique. La méthodologie proposée donne ainsi un cadre pour construire une radio flexible pour la radio logicielle à base de FPGAs et qui peut être reconfigurée pendant l'exécution.

Méthodologie proposée et principales contributions

La méthodologie proposée est un guide pour concevoir entièrement une radio flexible sur une plateforme SDR avec un FPGA, radio qui peut être reconfiguré en temps réel au cours de l'exécution. Le flot de conception va de la description de la forme d'onde aux étapes d'implémentation, de vérification et de validation du système. L'entrée est une description du système radio avec un langage de haut niveau qui exploite les techniques HLS pour cibler un FPGA. Deux types d'approches sont considérés pour traiter la flexibilité à l'exécution de la plateforme. Le premier propose de concevoir automatiquement des composants RTL (Register Transfert Level) multimodes avec des signaux de contrôle pour basculer entre les différents modes. L'autre approche est basée sur la reconfiguration dynamique partielle (DPR pour Dynamic Partial Reconfiguation). La DPR, appelée *reconfiguration matérielle* par la suite, est la capacité de reconfigurer une partie (ou des parties) du FPGA (par exemple, une fonctionnalité au niveau matériel) pendant que le reste du FPGA continue à fonctionner. C'est un sujet de recherche datant des années 90 qui est maintenant couramment utilisé dans les FPGA, puisque Xilinx et Altera fournissent de tels circuits. Les principaux avantages de la *reconfiguration matérielle* sont de permettre une flexibilité matérielle et de réutiliser

une zone matérielle, permettant ainsi de réduire la consommation d'énergie et les coûts de production.
Le concept de la HLS est complémentaire à la DPR car il permet la génération rapide des différentes configurations qui peuvent être ensuite implémentées dans les partitions reconfigurables du FPGA. De plus, les différentes configurations peuvent être dérivées d'un seul bloc fonctionnel pour lequel les modes ont des propriétés différentes (par exemple faible consommation d'énergie/faible performance ou haute performance/consommation d'énergie élevée). L'utilisation de la HLS est également très utile car la plupart des outils actuels permet une exploration rapide de l'espace de conception (DSE pour Design Space Exploration) en utilisant des techniques d'optimisation à la compilation. Ces techniques permettent par exemple des optimisations de latence, de débit, de puissance, de mémoire. La HLS permet d'effectuer de telles optimisations à partir de spécifications haut niveau, ce qui accélère considérablement le processus de conception.
Pour résumer, nos contributions sont les suivantes:

- La proposition d'un flot de conception complet pour des SDR utilisant des FPGA qui intègrent des blocs fonctionnels ou des formes d'onde pouvant être reconfigurés au cours de l'exécution. Ce flot exploite à la fois les technologies de HLS et de DPR pour améliorer à la fois le temps de prototypage et l'utilisation des ressources matérielles. Le flot permet également l'exploration rapide de l'espace de conception des blocs fonctionnels.
- L'application de cette proposition pour concevoir, à titre d'exemple, une transformée de Fourier rapide (FFT pour Fast Fourier Transform) flexible pour la norme LTE qui nécessite différentes tailles de FFT. Une DSE de la FFT est également effectuée avec plusieurs fréquences et directives d'optimisation HLS. Sur la base de cette étude, les compromis entre les ressources utilisées, le temps de reconfiguration, les efforts de conception et les performances sont discutés.
- La conception d'une forme d'onde de type LTE avec des capacités de reconfiguration à l'exécution, à la fois pour la modulation et la FFT, pour la démonstration. Dans notre cas, le système est implémenté et validé sur une carte Xilinx Virtex 6.

Flot de conception de reconfiguration dynamique

Il existe deux types de travaux qui traitent de la flexibilité temps-réelle d'une SDR à base de FPGA. Les premiers proposent de concevoir des composants RTL multi-modes avec des signaux de contrôle pour basculer entre les différents modes. Dans notre approche, un bloc de traitement multi-mode peut être décrit en utilisant des modifications algorithmiques dédiées du bloc de traitement (*reconfiguration algorithmique*) ou avec une génération automatique utilisant une encapsulation HLS (*reconfiguration logiciel*). Les autres sont basés sur la reconfiguration dynamique partielle DPR (*reconfiguration matérielle*).

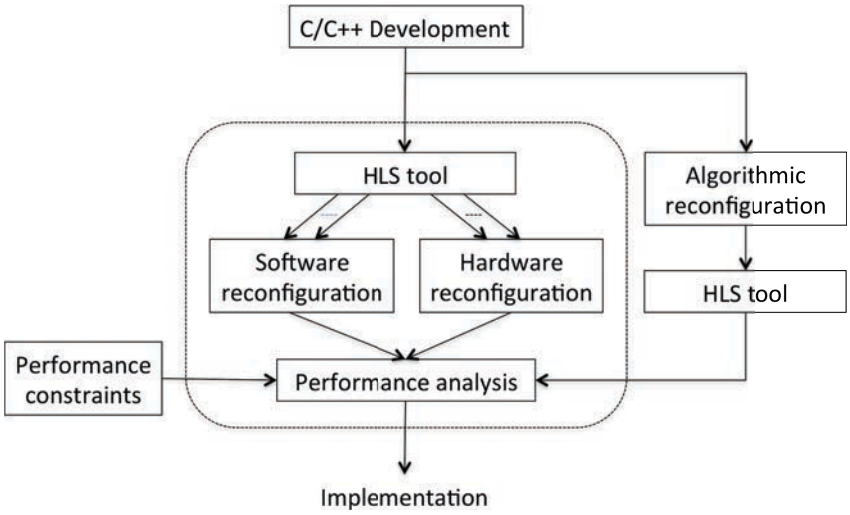


Figure 1 - Flot de conception de la reconfiguration définie logiciellement.

L'objectif de notre flot de conception, décrit dans la Figure. 1 et détaillé dans l'article [1], est de choisir ou de combiner ces reconfigurations tout en décrivant le bloc de traitement à un niveau de description élevé. Les différents modes d'un bloc de traitement peuvent être fournis soit par un codage de niveau RTL (Register Transfer Level) ou soit en utilisant un outil HLS pour générer différentes versions d'un bloc de traitement en modifiant les contraintes de synthèse telles que le débit, la latence, la taille des données, …

Dans la Figure 1, les contraintes de performances sont définies par l'utilisateur telles que les ressources/surfaces, le temps de reconfiguration, le débit ou la latence. L'analyse des performances compare les performances des trois solutions aux contraintes définies par l'utilisateur.
La Figure 2 montre le compromis de conception entre les ressources utilisées et le temps de reconfiguration pour les différents types de reconfiguration. La *reconfiguration algorithmique* permet de diminuer les ressources par rapport à la *reconfiguration logicielle* et de réduire le temps de reconfiguration par rapport à la *reconfiguration matérielle*. Elle offre donc un meilleur compromis ressources/temps de reconfiguration. Cependant, en fonction des blocs de traitement, le temps de description (c'est-à-dire le temps de codage) de la *reconfiguration algorithmique* peut être important par rapport à *reconfiguration logicielle*.

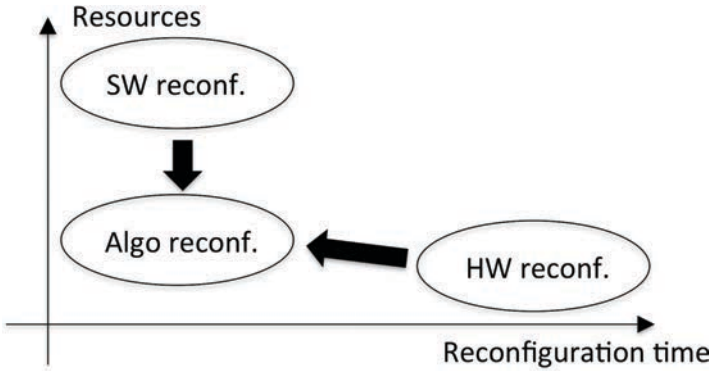


Figure 2 - Compromis ressources utilisées et temps de reconfiguration pour les différents types de reconfiguration.

Performances de la FFT multi-mode pour le standard LTE

Un bloc de traitement FFT flexible est conçu en utilisant l'approche proposée. Pour la norme LTE, le composant FFT résultant doit avoir six modes correspondant à différentes tailles de FFT : {128; 256; 512; 1024; 1536; 2048}. L'architecture proposée repose sur deux descriptions haut-niveau de FFT. D'une part, une FFT pour les puissances de 2 est obtenue par reconfiguration algorithmique afin de partager les ressources entre les différents modes. Ensuite, une fonction FFT pour la taille 1536 est décrite. Dans ce résumé, nous présentons brièvement l'étude du partage des ressources entre ces 2 fonctions pour les *reconfigurations matérielle* et *logicielle*.

La *reconfiguration logicielle* est appliquée en premier pour concevoir une FFT réalisant les 6 modes LTE. Une fonction Multi_Mode_Block_LTE( ) est générée à partir des deux fonctions Block_FFTpow2( ) et Block_FFT1536( ). La *reconfiguration matérielle* est ensuite appliquée aux deux fonctions Block_FFTpow2( ) et Block_FFT1536( ). Deux partitions séparées sont d'abord générées : une pour la FFT puissance de 2 et une pour la FFT 1536. Enfin, une partition commune est créée pour la DPR des 2 FFT.

La Figure 3 donne les résultats de synthèse et la latence (en nombre de cycles d'horloge) pour les deux reconfigurations. Les ressources nécessaires à chaque fonction sont précisées. Pour la *reconfiguration logicielle*, les ressources utilisées par Multi_Mode_Block_LTE( ) sont presque la somme des ressources utilisées par Block_FFTpow2( ) et Block_FFT1536( ) lorsqu'ils sont synthétisés séparément. L'outil HLS ne partage donc malheureusement pas les ressources entre les deux fonctions même si elles ne sont pas exécutées en même temps. Pour la *reconfiguration matérielle*, la partition de la FFT 1536 est plus petite que celle de la FFT puissance de 2. Ainsi, lors de la combinaison des 2 FFTs en une seule partition, la partition résultante est basée sur la partition de la FFT puissance de 2 (principe du « pire » cas, à savoir la plus grande partition). Par rapport à la *reconfiguration logicielle*, la FFT multimode basée sur la *reconfiguration matérielle* utilise moins de ressources (les blocs mémoire BRAM et de traitement DSP sont les composants logiques les plus coûteux). Lorsque la taille de la FFT doit être modifiée mais que la puissance est toujours une puissance de deux, un seul cycle d'horloge est nécessaire pour la reconfiguration dans les deux cas. Toutefois, 32,9 ms sont nécessaires à la reconfiguration lors de la commutation d'une FFT 1536 vers une FFT puissance de deux (ou vice versa) avec une *reconfiguration matérielle*, alors qu'un seul cycle est requis avec la reconfiguration logicielle.

	Processing block : Resources needed		**Hardware reconfiguration** Partition : Resources used			**Software reconfiguration**
	Pow.-of-two FFT	FFT 1536	Pow.-of-two FFT	FFT 1536	FFT for LTE	FFT for LTE
BRAM	12	14	17	14	17	26
DSP	65	40	68	56	68	103
LUT	2553	3054	4080	3360	4080	5256
FF	2497	2010	8160	6720	8160	4299
Bitstream size	n/a	n/a	416016 Bytes	277344 Bytes	2 x 416016 Bytes	n/a
Reconf. time	n/a	n/a	32.9 ms	21.96 ms	32.9 ms	n/a
Latency	$L_{FFTpow2}$	52198	$L_{FFTpow2}$	52198	$L_{FFTpow2}$& 52198	$L_{FFTpow2}$ & 52198

Figure 3 - Performance de la FFT pour le standard LTE avec les reconfigurations logicielle et matérielle. Les latences pour les FFTs puissance de 2 sont LFFTpow2 = {355 ;7604 ;16172 ;34284 ;72412}.

Ce bloc de traitement reconfigurable a été intégré dans une chaine d'émission complète pour la norme LTE et implémenté sur une carte FPGA Virtex 6. Une démonstration de cette plateforme a été faite dans une conférence internationale [2].

Conclusion

Cette thèse présente une méthodologie pour l'implémentation de la reconfiguration en cours d'exécution dans le contexte des SDR utilisant des FPGA. Le flot de conception proposé permet l'exploration entre la reconfiguration dynamique partielle et la conception multimode basée sur un signal de contrôle. Ce compromis architectural repose sur la HLS et les optimisations de conception associées.

Une FFT flexible pour la norme LTE est mise en œuvre comme cas applicatif. Le bloc fonctionnel proposé combine à la fois la DPR (pour traiter la FFT de taille 1536) et la reconfiguration algorithmique (lorsque la taille de la FFT est une puissance de deux). Les résultats de synthèse montrent le compromis qui peut être obtenu entre le temps de reconfiguration et l'utilisation des ressources FPGA.

Les travaux futurs consistent à explorer la mise en œuvre d'autres blocs de traitement et l'automatisation du flot de conception.

Publications scientifiques

[1] Mai-Thanh Tran, Matthieu Gautier, and Emmanuel Casseau, "On the FPGA-based implementation of a flexible waveform from a high-level description: Application to LTE FFT case study", EAI International Conference on Cognitive Radio Oriented Wireless Networks (Crowncom), Grenoble, France, May 30 - June 1, pp. 545-557, 2016.

[2] Mai-Thanh Tran, Emmanuel Casseau, Matthieu Gautier, "FPGA-based implementation of a flexible FFT dedicated to LTE standard", Conference on Design and Architectures for Signal and Image Processing (DASIP), Demo Night, Rennes, France, pp.1-2, October 12-14, 2016.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Synthèse matérielle d'une radio flexible et reconfigurable depuis un langage de haut niveau

Mots clés : radio logicielle, synthèse de haut niveau, FPGA, reconfiguration dynamique

Résumé :
La radio logicielle est une technologie prometteuse pour répondre aux exigences de flexibilité des nouvelles générations de standards de communication. Elle peut être facilement reprogrammée au niveau logiciel pour implémenter différentes formes d'onde. En s'appuyant sur une technologie dite logicielle telle que les microprocesseurs, cette approche est particulièrement flexible et assez facile à mettre en œuvre. Cependant, ce type de technologie conduit généralement à une faible capacité de calcul et, par conséquent, à des débit faibles. Pour résoudre ce problème, la technologie FPGA s'avère être une bonne alternative pour la mise en œuvre de la radio logicielle. En effet, les FPGAs offrent une puissance de calcul élevée et peuvent être reconfigurés. Ainsi, inclure des FPGAs dans le concept de radio logicielle peut permettre de prendre en charge plus de formes d'onde avec des exigences plus strictes qu'une approche basée sur la technologie logicielle. Cependant, les principaux inconvénients d'une conception à base de FPGAs sont le niveau du langage de description d'entrée qui doit typiquement être le niveau matériel, et le temps de reconfiguration qui peut dépasser les exigences d'exécution si le FPGA est entièrement reconfiguré. Pour surmonter ces problèmes, cette thèse propose une méthodologie de conception qui exploite à la fois la synthèse de haut niveau et la reconfiguration dynamique. La méthodologie proposée donne un cadre pour construire une radio flexible pour la radio logicielle à base de FPGAs et qui peut être reconfigurée pendant l'exécution.

Title : Towards Hardware Synthesis of a Flexible Radio from a High-Level Language

Keywords: Software defined radio, high-level synthesis, FPGA, dynamic reconfiguration

Abstract :
Software defined radio (SDR) is a promising technology to tackle flexibility requirements of new generations of communication standards. It can be easily reprogrammed at a software level to implement different waveforms. When relying on a software-based technology such as microprocessors, this approach is clearly flexible and quite easy to design. However, it usually provides low computing capability and therefore low throughput performance. To tackle this issue, FPGA technology turns out to be a good alternative for implementing SDRs. Indeed, FPGAs have both high computing power and reconfiguration capacity. Thus, including FPGAs into the SDR concept may allow to support more waveforms with more strict requirements than a processor-based approach. However, main drawbacks of FPGA design are the level of the input description language that basically needs to be the hardware level, and, the reconfiguration time that may exceed run-time requirements if the complete FPGA is reconfigured. To overcome these issues, this PhD thesis proposes a design methodology that leverages both high-level synthesis tools and dynamic reconfiguration. The proposed methodology is a guideline to completely build a flexible radio for FPGA-based SDR, which can be reconfigured at run-time.