POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Acceleration of an OFDM modulator on a Xilinx FPGA

Supervisor

Prof. Luciano LAVAGNO

Prof. Mihai Teodor LAZARESCU

Candidates

Giovanni Luca AMATO

April 2023

Abstract

Today's technology rush and the rise of the IoT leads us to find new solutions to achieves significant speedup and resource savings over the traditional implementation. In fact, this can result computationally intensive when we talk about digital signal processing, Simulation and modeling or machine learning. Hardware acceleration is dedicate hardware designed to perform specific functions more efficiently when compared to software running on a general-purpose CPU. Central Processing Unit are projected to be used in different tasks, but to archive that the trade-off is a loss of performance in terms of in terms of latency, throughput, and resource utilization respect to dedicate hardware limited to a single task.

Using High Level Synthesis (HLS) to generate hardware designs enables the use of a high-level programming language instead of traditional hardware description languages. HLS tool analyzes the design specification and automatically generates hardware implementation based on the performance requirements.

The objective of our research group is 3GPP 5G Channel Model Acceleration. This task was mainly focused on the acceleration of the channel model using HLS tools for Xilinx and Intel FPGA platforms. Given the complexity of this project each member of the team focusing on a particular component of the channel.

My role was to implement orthogonal frequency-division multiplexing (OFDM) modulator and demodulator designs to be inserted as accelerated channel model modules. The module is characterized by Direct or Inverse Fast Fourier Transform (FFT/IFFT) calculation on a large asset of data, implemented using a dedicated IP. Using Vitis HLS, starts from a Matlab algorithm, was implemented a C++ implementation of the OFDM algorithm, optimized for the HLS (High Level Synthesis). The generated Register Transfer Level (RTL) design was tested with simulation and on board before to be nested in the channel chain.

All modules developed are described in a dedicated chapter and each one with all the test performed, followed by a comparison between different implementation.

Table of Contents

List of Tables IV

List of Figures VI

Acronyms VII

1 Introduction 1
- 1.1 5G Protocol Stack . . . 1
  - 1.1.1 5G Physical Layer . . . 4
  - 1.1.2 Channel simulation . . . 5
  - 1.1.3 OFDM Modulation . . . 7
- 1.2 Board description . . . 10
  - 1.2.1 FPGA . . . 12
- 1.3 Vitis Unified Software Platform . . . 15
  - 1.3.1 Vitis HLS . . . 18
- 1.4 Thesis structure . . . 20

2 Modulator Block 21
- 2.1 Matlab reference . . . 21
- 2.2 Kernel description . . . 24
  - 2.2.1 IP description . . . 24
  - 2.2.2 C++ code and C-Simulation . . . 27
  - 2.2.3 C-Synthesis results . . . 30
  - 2.2.4 Co-Simulation results and waveform . . . 36
  - 2.2.5 Implementation results . . . 37

3 Demodulator Block 40
- 3.1 Matlab reference . . . 40
- 3.2 Kernel description . . . 42
  - 3.2.1 C++ code and C-Simulation . . . 42
  - 3.2.2 C-Synthesis results . . . 42

3.2.3 Cosimulation results and waveform . . . 45
3.2.4 Implementation results . . . 47

4 1D SSR FFT IP 49
4.1 IP description . . . 49
4.2 Code adjusstment . . . 52
4.3 Reports Analysis . . . 53

5 On Device Deployment 57
5.1 Host code . . . 57
5.2 Modulator . . . 60
5.2.1 SW Emulation . . . 60
5.2.2 HW Emulation . . . 61
5.2.3 Hardware implementation . . . 63
5.3 Demodulator . . . 65
5.4 SSRModulator . . . 66
5.5 SSRDemodulator . . . 67

6 Next steps 68
6.1 Conclusion . . . 68
6.2 Further Optimizations . . . 69

Bibliography 71

List of Tables

1.1 Alveo U280 Data Center Accelerator Card Specifications 11

2.1 Synthesis Report of Timing (modulator) 33
2.2 Synthesis Report of Performances (modulator) 33
2.3 Synthesis Report of Resources (modulator) 35
2.4 The Resource Usage and the Final Timing (modulator) 37
2.5 Fail Fast Report (modulator) 38

3.1 Synthesis Report of Performances (demodulator) 43
3.2 Synthesis Report of Resources (demodulator) 44
3.3 Cosimulation II Estimates (demodulator) 45
3.4 Cosimulation Latency Estimates (demodulator) 46
3.5 The Resource Usage and the Final Timing (demodulator) 47

4.1 Input arrays order 50
4.2 Output arrays order 50
4.3 Synthesis Report of Timing (SSRmodulator) 53
4.4 Synthesis Report of top-level Resources (SSRmodulator) 54
4.5 Synthesis Report of top-level Resources (SSRdemodulator) 54
4.6 The Resource Usage and the Final Timing (SSRmodulator) 55

6.1 Calculated throughput for different modules 68

Listings

2.1 OFDM parameters 22
2.2 OFDM implementation 22
2.3 FFT IP paramenters structure 26
3.1 Demodulator implementation 40
4.1 SSR IP FFT paramenters structure 50
4.2 Simplified structure for float data 51

List of Figures

1.1 NR radio interface protocol 3
1.2 Link Level Simulator for 5G NR 6
1.3 Typical OFDM transmission workflow 7
1.4 AMD Alveo U280 Data Center accelerator card 10
1.5 UltraScale FPGA with Columnar Resources 12
1.6 Vitis Development Flow 16
1.7 Vitis HLS Development Flow 18

2.1 ifftshpe function explanation 28
2.2 CP function explanation 28
2.3 Place & Route Timing Paths Report (modulator) 39

3.1 Place & Route Timing Paths Report (demodulator) 48

4.1 Modules/Loops for FFT block in SSRmodulator 53
4.2 Place & Route Timing Paths Report (SSRmodulator) 56
4.3 Place & Route Timing Paths Report (SSRmodulator) 56

5.1 Host/Device interface 58
5.2 Vitis unified software development flow 59
5.3 System Diagram (modulator) 61
5.4 Vivado utilization report (modulator) 63
5.5 Vivado utilization report (demodulator) 65
5.6 Vivado utilization report (SSRmodulator) 66
5.7 Vivado utilization report (SSRdemodulator) 67

Acronyms

3GPP The 3rd Generation Partnership Project

HLS High-level synthesis

OFDM Orthogonal Frequency-Division Multiplexing

FPGA Field Programmable Gate Array

GPU graphics processing unit

DFT Discrete Fourier Transform

FFT Fast Fourier Transform

IFFT Inverse Fast Fourier Transform

IP Intellectual Property

HDL Hardware description language

LUT Lookup table

DSP Digital Signal Processor

CP Cyclic prefix

RTL Register-transfer level

CUDA Compute Unified Device Architecture

TTI Transmission Time Interval

CSI-RS Channel State Information Reference Signal

XRT Xilinx Runtime

LTE Long Term Evolution

FF Flip-Flop

MMCM Mixed-Mode Clock Manager

PLL Phase-locked loop

Chapter 1

Introduction

1.1 5G Protocol Stack

Nowadays, there is a need of new network infrastructure that can ensure services to all devices. The infrastructure will provide stable connections, increased bandwidth and minimal lag. All this requirements set a base for fifth generation of wireless network called 5G. The usage of higher frequencies enable us to communicate at higher data rates but at lesser distance. This means that they need to implement multiple antennas to boost its capacity and signal quality. 5G will also enable operators to divide a physical network into multiple virtual networks depending on its usage.

The Third Generation Partnership Project (3GPP) has specified a new radio interface for 5G which is referred as New Radio (NR). This new model reuses some of features and structures from LTE but this new technology is not required to maintain backward compatibility being able to exploit much higher-frequencies, which gives us more spectra for wide bandwidth and higher data-rates. However, communication at higher frequencies is also affected by higher radio-channel attenuation which results in limiting network coverage. This problem is tackled by using multiple antennas for communication which also favors the beam-centric design of NR.

First specifications for 5G NR have been published in Dec. 2017, which supports NSA (Non Standalone) where in 5G compliant UE relies on existing LTE for initial access and mobility. In June 2018, SA versions of 5G NR specifications have been finalized which works independent of LTE.

I would not describe much details on each element of the protocol stack in this page. The purpose of this chapter is to provide some big picture to been able to understand the radio protocol stack. Most of the fundamental idea in this page comes from 3GPP 38.300 [1].

There are two main components in 5G NR network:

UE (mobile subscriber) and **gNB (base station)**.

gNBs are connected with 5G Core in the backend. The connection from gNB to UE is known as downlink which uses PBCH, PDSCH and PDCCH channels for carrying different data/control informations. The connection from UE to gNB is known as uplink which uses PRACH, PUSCH and PUCCH channels.

In 5G NR there are various physical channels in the downlink (from gNB to UE) and uplink (from UE to gNB):

Downlink channels: PDSCH, PDCCH, PBCH.

Uplink channels: PRACH, PUSCH, PUCCH.

There are specific physical signals present in both downlink and uplink for various purposes. Front loaded DMRS (Demodulation Reference signal) is used for both PDSCH and PUSCH channels. Orthogonal Frequency Division Multiplexing (OFDM) with CP is used for both downlink and uplink chain. Uplink also uses DFT Spread OFDM with CP for improved coverage. Accordingly CP length is choosen. 5G NR Supports two frequency ranges FR1 (Sub 6GHz) and FR2 (millimeter wave range, 24.25 to 52.6 GHz). NR uses flexible subcarrier spacing derived from basic 15 KHz subcarrier spacing used in LTE.

NR Radio Protocol Stack Architecture is almost same as LTE Radio Protocol Stack Architecture. As in LTE/WCDMA, NR radio protocol stack has two different stack depending on the type of data that is processed by the stack. If the data is signaling message, it goes through the C-plane stack and if it is user data, it goes through U-Plane stack. Both U-Plane and C-Plane is made up of a common structure as represented in Figure 1.1, but the components sitting on top of PHY/MAC/RLC/PDCP gets different between C-Plane and U-Plane.

The 5G Protocol Stack can be divided in layer 1, layer 2 and layer 3:

- The 5G layer-1 is PHYSICAL layer.
- The 5G layer-2 include MAC, RLC and PDCP.
- The 5G layer-3 is RRC layer.

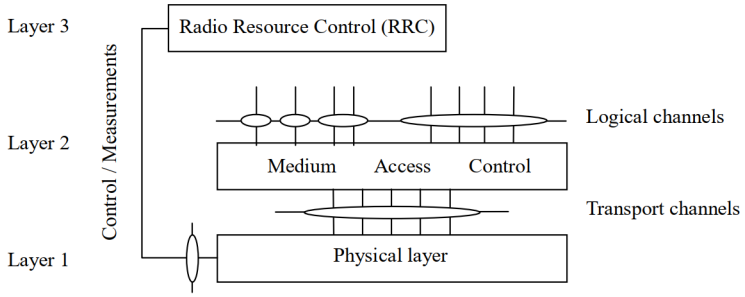

Figure 1.1: NR radio interface protocol

Figure 1.1 shows the NR radio interface protocol architecture around the physical layer (Layer 1). The physical layer interfaces the Medium Access Control (MAC) sub-layer of Layer 2 and the Radio Resource Control (RRC) Layer of Layer 3.The circles between different layer/sub-layers indicate Service Access Points (SAPs).

The physical layer offers a transport channel to MAC. The transport channel is characterized by how the information is transferred over the radio interface. MAC offers different logical channels to the Radio Link Control (RLC) sub-layer of Layer 2. A logical channel is characterized by the type of information transferred.

In case of U-Plane, a layer called SDAP is sitting at the top of the radio stack and the SDP is connected to UPF (User Plane Function). In case of C-Plane, the two layers RRC and NAS are sitting at the top of the stack. NAS layer gets connected to AMF (Access and Mobility Management Function).

1.1.1 5G Physical Layer

The 5G PHY layer is specified in 3GPP TS 38.200 series of documents [2].

The physical layer offers data transport services to higher layers. The access to these services is through the use of transport channels via the MAC sub-layer.

A transport block is defined as the data delivered by MAC layer to the physical layer and vice versa. The Layer 1 is defined in a bandwidth agnostic way based on resource blocks, allowing the NR Layer 1 to adapt to various spectrum allocations.

Following are the functions of 5G layer 1 i.e. PHYSICAL (PHY) Layer [3].

- Error detection on the transport channel and indication to higher layers
- FEC encoding/decoding of the transport channel
- Hybrid ARQ soft-combining
- Rate matching of the coded transport channel to physical channels
- Mapping of the coded transport channel onto physical channels
- Power weighting of physical channels
- Modulation and demodulation of physical channels
- Frequency and time synchronisation
- Radio characteristics measurements and indication to higher layers
- Multiple Input Multiple Output (MIMO) antenna processing
- Transmit Diversity (TX diversity)
- Digital and Analog Beamforming
- RF processing

The multiple access scheme for the NR physical layer is based on Orthogonal Frequency Division Multiplexing (OFDM) with a cyclic prefix (CP).

The downlink transmission waveform is conventional OFDM using a cyclic prefix. The uplink transmission waveform is conventional OFDM using a cyclic prefix with a transform precoding function performing DFT spreading that can be disabled or enabled. For uplink, Discrete Fourier Transform-spread-OFDM (DFT-s-OFDM) with a CP is also supported.

1.1.2 Channel simulation

Channel simulation is used for functional and performance verification of such models in the network planning phase. By channel model is meant the model of the medium interposed between typical transmission and receiving stations consists of two groups antennas, one transmitting and one receiving each alongside a propagation channel. The propagation channel is the environment in which radio waves propagate from the transmitting antenna to the receiving antenna.

Channel simulation is important for evaluating performance of communication systems. What a model aims to do is to reflect physical reality with a level of detail appropriate to the purpose, balancing accuracy and complexity. Channel simulators are used to model the routing protocol performance, traffic flows and evaluate the efficiency of the communication system using real-life parameters in a virtual environment the impact of a physical channel in real environment is very important.

The design of a channel model starts with the requirements of the target system to describe. For wireless systems it is even more crucial because of high variations in propagation medium with respect to space, time and frequency.

The results of the simulations are used as input by planning tool (every network operator use these tools) in order determine the network infrastructure (network nodes) and its configuration Channel simulation allows to create a theoretical model which can be used to assess the performance of real devices.

Many network operators develop these models by running simulations and the test commercial devices in lab and in the field in order to verify that nominal performances are achieved 3GPP and other standardization bodies use channel simulator in order to develop theoretical model of communication system. Accurate 5G channel model simulations require very high computational effort and take a very long time to execute on general purpose processors.

Hardware acceleration of such functions is an option to speed up the execution, hence to reduce the simulation time. It was required to have it working in a homogeneous environment to take benefits of High-Level-Synthesis optimizations and use them in order to derive technological requirements or event to choose among multiple solutions, architectures, and systems to optimize the design.

In order to accelerate the channel functions using an FPGA which supports complex simulations with higher numbers o antenna elements and more UE speeds, the design is split into two major parts: **host** and **kernel** code.

In our case, the link between trasmitter and receiver is modelled as follows:

```
% OFDM signal generation
[ tb10 ] = ofdm_modulation_ul(env,tb9);
% tb10 = OFDM signal in time domain including CP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transmitter Front-End (Oversampling and filtering)
[ tb11 ] = oversample_filter_ul(env,tb10);
% tb11 = OFDM signal oversampled and filtered
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Phase Noise (PN) insertion
[ tb12, Jmf ] = nr_phase_noise_insertion_ul(env,tb11);
% tb12 = OFDM signal affected by phase noise
% Jmf = phase noise spectrum
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rayleigh fading channel
[ tb13, sinr_pre, Hid ] = fading_channel(env,ch,tb12);
% tb13 = OFDM signal at radio channel output
% sinr_pre = pre-detection SINR per antenna in dB
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Receiver Front-End (Filtering and downsampling)
[ tb14 ] = downsample_filter(env,tb13);
% tb14 = received sig  nal at RX Front-end output
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Analog Beamforming (Grid of Beams)
[ tb15, beam_id ] = nr_analog_beamforming_ul(env,tb14);
beam_stat(:,SINR_ITER,counter+1) = beam_id;
% tb15 = OFDM signal after Analog Beamforming
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OFDM signal demodulation
[ tb16 ] = ofdm_demodulation_ul(env,tb15);
% tb16 = received signal after FFT and CP cut
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 1.2: Link Level Simulator for 5G NR

The host code performs tasks related to control and data movement such as allocating space on the device memory, receiving data via the socket from a client, launching the kernel and copying results back to the client via another socket. The kernel code is the part which is highly data parallel and computationally intensive, optimized to be executed on the target FPGA.

1.1.3 OFDM Modulation

Orthogonal Frequency Division Multiplexing (OFDM) is a specialized frequency-division multiplexing (FDM) method, with the additional constraint that all subcarrier signals within a communication channel are orthogonal to one another, meaning that crosstalk between the sub-channels is eliminated and inter-carrier guard bands are not required [4].

The block supports scalar and vector inputs. You can use a vector input to increase the data throughput and achieve an high throughput (GSPS). The block supports windowing for scalar and vector inputs to reduce the spectral regrowth, or adjacent channel leakage ratio (ACLR), of an OFDM signal. Almost the whole available frequency band can be used. OFDM generally has quasi-white spectrum, giving it favourable electromagnetic interference properties with respect to other co-channel users.

The orthogonality allows high spectral efficiency, with a total symbol rate near the Nyquist rate for the equivalent baseband signal. This simplifies the design of both the transmitter and the receiver because it does not need a separate filter for each sub-channel, where each subcarrier is modulated with a classic modulation at a low symbol rate. This maintains total data rates similar to conventional single-carrier modulation schemes considering same bandwidth.

The orthogonality also allows for efficient modulator and demodulator implementation using the FFT algorithm on the receiver side, and inverse FFT on the sender side.The time to compute the inverse-FFT or FFT transform has to take less than the time for each symbol.

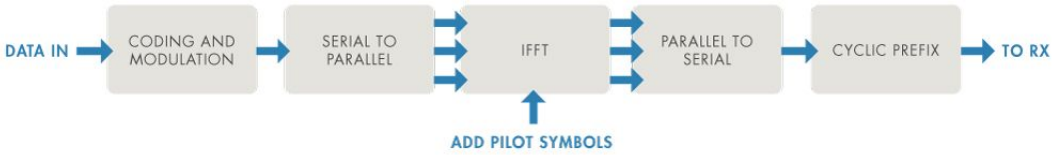

Figure 1.3: Typical OFDM transmission workflow

The most of the computational effort in the OFDM algorithm is the FFT calculation. A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT).

The DFT is obtained by decomposing a sequence of values into components of different frequencies, converting from its original domain (often time or space) to a representation in the frequency domain and vice versa. This operation is useful in many fields, but computing it directly from the definition is often too slow to be practical. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of mostly zero factors. As a result, it manages to reduce the complexity of computing the DFT from $2O\left(N^2\right)$, which arises if one simply applies the definition of DFT, to $logO(N \log N)$, where N is the data size.

The difference in speed can be enormous, especially for long data sets where N may be in the thousands or millions. In the presence of round-off error, many FFT algorithms are much more accurate than evaluating the DFT definition directly or indirectly. Let x_{N-1} be complex numbers.

The DFT is defined by the formula

$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \qquad k = 0, \ldots, N-1;$

where $e^{i2\pi/N}$ is a primitive N_{th} root of 1.

Evaluating this definition directly requires $O(N^2)$ operations: there are N outputs X_k, and each output requires a sum of N terms. An FFT is any method to compute the same results in $O(N \log N)$ operations.

To illustrate the savings of an FFT, consider the count of complex multiplications and additions for $N = 4096$ data points. Evaluating the DFT's sums directly involves N^2 complex multiplications and $N(N-1)$ complex additions, of which $O(N)$ operations can be saved by eliminating trivial operations such as multiplications by 1, leaving about 30 million operations. In contrast, the radix-2 Cooley–Tukey algorithm, for N a power of 2, can compute the same result with only $(N/2)\log_2(N)$ complex multiplications and $log2N \log_2(N)$ complex additions, a thousand times less than with direct evaluation.

In practice, actual performance on modern computers is usually dominated by factors other than the speed of arithmetic operations and the analysis is a complicated subject, but the overall improvement from $O\left(N^2\right)$ to $O(N \log N)$ remains.

OFDM was improved with the introduction of a guard interval to achieve a better orthogonality in transmission channels affected by multipath propagation. The low symbol rate makes the use of a guard interval between symbols beneficial, making it possible to eliminate intersymbol interference (ISI) and to obtain a signal-to-noise ratio improvement.

The main advantage of OFDM over single-carrier schemes is its ability to work in harsh channel conditions without the need for complex equalization filters.

The cyclic prefix, which is transmitted during the guard interval, consists of the end of the OFDM symbol copied into the guard interval, and the guard interval is transmitted followed by the OFDM symbol. The reason that the guard interval consists of a copy of the end of the OFDM symbol is so that the receiver will integrate over an integer number of sinusoid cycles for each of the multipaths when it performs OFDM demodulation with the FFT.

1.2 Board description

The board using as target platform for the project exposed in these thesis is the Alveo U280. AMD Alveo U280 Data Center accelerator cards are designed to meet the constantly changing needs of the modern Data Center. [5].

Figure 1.4: AMD Alveo U280 Data Center accelerator card

Built on the AMD 16nm UltraScale+ architecture, Alveo U280 offers 8GB of HBM2 460 GB/s bandwidth to provide high-performance, adaptable acceleration for memory-bound, compute intensive applications including database, analytics, and machine learning inference.

This boards consist of an Virtex UltraScale+ FPGA and two HBM2 stacks (8 HBM2 dies). The FPGA and the HBM2 dies are connected through 32 independent PCs. Each PC has 256MB of capacity (8 GB in total). The U280 acceleration card includes PCI Express 4.0 support to leverage the latest server interconnect infrastructure for high-bandwidth host processors.

Modern memory interfaces provide access through multiple banks with dedicated access channels like high bandwidth memory (HBM) lanes or double data rate (DDR) channels. Hence, access bandwidth of an array can be increased by striping it across the different memory interfaces (banks) available on the board.

The same considerations also apply to on-chip BRAM banks, in order to increase the on-chip memory bandwidth to match the requirements of the data computations.

Card Specifications	**U280**
DRAM Memory	
HBM2 Total Capacity	8GB
HBM2 Total Bandwidth	460GB/s
DDR Format	2x16GB 72b DIMM DDR4
DDR Memory Capacity	32GB
DDR Total Bandwidth	38GB/s
SRAM Memory	
Internal SRAM Capacity	41MB
Internal SRAM Total Bandwidth	30TB/s
Interfaces	
PCI Expresss	Gen4x8 with CCIX
Network Interfaces	2xQSFP28 (100GbE)
Logic Resources	
Look-up Tables (LUTs)	1,079,000
Power	
Maximum Total Power	225W

Table 1.1: Alveo U280 Data Center Accelerator Card Specifications

On the Xilinx device, the platform consists of two physical FPGA partitions: Shell and User. The Shell partition is a static region and provides basic infrastructure for the platform like PCIe connectivity, board management, sensors, clocking, and reset. The User partition is a dynamic region that contains user compiled binary called .xclbin which is loaded by XRT during execution.

RTL kernels are the custom logic created by the developer and programmed into the dynamic region. In this document, kernels refer to the functions that the designer is implementing into the dynamic region of the Alveo accelerator card.

1.2.1 FPGA

Xilinx has developed the Alveo family of PCIe Data Center accelerator cards using FPGAs at its core [6].

An FPGA (field-programmable gate array) is an integrated circuit (IC) equipped with configurable logic blocks (CLBs) and other features that can be programmed by a user through a bitstream generated by a synthesis tool to implement any type of digital function as a physical circuit. The term "field-programmable" indicates that the FPGA's abilities are adjustable and not hardwired by the manufacturer like other ICs.

By creating multiple copies of these functions, FPGAs are particularly well suited at implementing functions in parallel, making them extraordinarily good at serving as hardware accelerators for applications that contain high levels of parallelism. Arrays of hardware blocks, each configurable, can be connected as needed, allowing highly efficient, domain-specific architectures to be built for any application.

FPGAs come in different sizes with different quantities of programmable logic resources. Larger devices contain more resources, allowing designers to implement more parallel circuits, leading to higher levels of acceleration. The variety of devices provides designers with multiple cost/performance trade-offs [7].

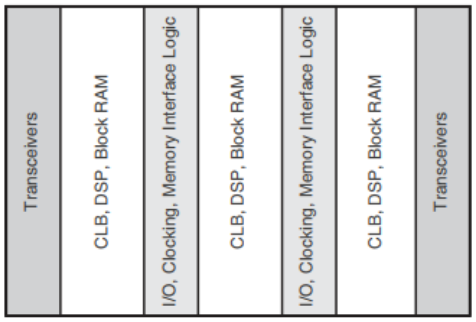

Figure 1.5: UltraScale FPGA with Columnar Resources

The process is similar to programing software in that you write code that is turned into a binary file and loaded onto the FPGA, but the outcome is that the HDL makes physical changes to the hardware, rather than strictly optimizing the device to run software. It's also possible to adjust basic functions such as memory or power usage depending on the task.

FPGAs give programmers and designers the ability to adapt and update the compute architecture with greater flexibility, resulting in domain-specific architectures that are more specific to their requirements. FPGAs are not new, but are becoming more necessary due to the speed of innovation in areas like artificial intelligence. Because the elements and the routing resources that connect them are configured after power-up, the FPGA can be repeatedly programmed to implement any set of functions required.

FPGA programming uses an HDL to manipulate circuits depending on what capabilities you want the device to have. The process is different from programming a GPU or CPU, since you aren't writing a program that will run sequentially. Rather, you're using an HDL to create circuits and physically change the hardware depending on what you want it to do. CPUs are highly flexible, but their underlying hardware is fixed. Once a CPU is manufactured the hardware cannot be changed. It relies on software to tell it which specific operation (arithmetic function) to perform, on which data in memory. The hardware must be capable of performing all possible operations, which are called using software instructions and can generally only execute one instruction at a time. FPGAs in contrast can process massive amounts of data in parallel. The benefit of adaptive hardware over CPUs varies by application largely depending on the nature of the computation and its ability to be parallelized, but it's not uncommon to see a 20X performance improvement respects a CPU implementation of functions that can be highly parallelized.

The architecture of FPGAs makes them an efficient solution for hardware acceleration. Devices such as ASICs and GPUs use an antiquated method of jumping between programming and memory. They also don't accommodate applications where real-time information is needed, since the high amount of power required for storage and retrieval tasks causes performance lags. As FPGAs are re-configurable according to required functionality, they are considered more flexible than ASICs (Application Specific Integrated Circuits) but at the cost of power and area. Also they are more efficient than general-purpose processors but the cost to pay is more complex programming and lower flexibility.

Unlike ASICs and GPUs, FPGAs don't need to jump between memory and programming, which makes the process of storing and retrieving data more efficient. And since FPGA architecture is more flexible, you can customize how much power you'd like an FPGA to utilize for a specific task. Unlike GPUs, which contain processing cores that must fetch and execute instructions, FPGAs have a flexible architecture that maps code to physical logic circuitry. Like GPUs, however, it will be necessary for you to understand some of the basics of how this is done to architect your code for best results.

That flexibility can help offload energy-consuming tasks to one or several FPGAs from a conventional CPU or another device. And since many FPGAs can be reprogrammed, you can easily implement upgrades and adjustments to a hardware acceleration system. Though FPGAs had exclusively existed in the domain of hardware engineers, AI scientists and software programmers can now access new platforms that make the process feel the same as writing software. With the right tools, you will find a solution for programming FPGAs that meets you at your current knowledge level of software and hardware.

The FPGAs considered in this study, despite being both aimed at data center applications, do not have optimized support for double-precision floating-point adders or multipliers, so as already discussed for both the IP used it was preferred to use float data instead of double. To create the design, Xilinx provides a suite of tools that can aid software developers in every step of the FPGA programming process.

1.3 Vitis Unified Software Platform

The Vitis unified software platform is a development environment for heterogeneous applications supporting Xilinx devices such as Alveo Data Center Accelerator cards. The Vitis unified software platform combines all aspects of Xilinx hardware and software development into one unified environment using standard C/C++ for both software and hardware components.

The Vitis tools provide compilation, linking, profiling and debug capabilities for heterogeneous systems in a number of different design flows including Data Center application acceleration, RTL kernel design, Embedded System design, and traditional embedded hardware and software design.

In the Vitis environment, heterogeneous systems include software applications running on x86 host processors or Arm embedded processors, compute kernels running in programmable-logic (PL) regions or Versal AI Engine arrays and extensible platform designs that provide the foundation for building and running the heterogeneous systems.

The software development tool stack, such as compilers and cross-compilers to build your software application. Debuggers to help you locate and fix any problems in your system design. Program analyzers to let you profile and analyze the performance of your application. Xilinx Runtime (XRT) provides an API and drivers for your software program to connect with the target platform, and handles transactions and data transfers between the software application and the hardware design.

The development flow works in parallel on two ways:

- **Application Compilation** using G++ to generate the host.exe file.
- **Kernel Compilatio**n to obtain the .xclbin file needed.

The host program written in C/C++, and using XRT native API, is compiled using a g++ compiler to create a host executable file to run on the x86 processor. The host program interacts with kernels in the PL region on the FPGA device.

Vitis HLS is a compiler that takes C/C++ source code as an input and synthesizes it into an RTL design that is optimized for Xilinx FPGA products. Each C++ kernel must be synthesized using Vitis HLS to produce a Xilinx object (.xo) file.

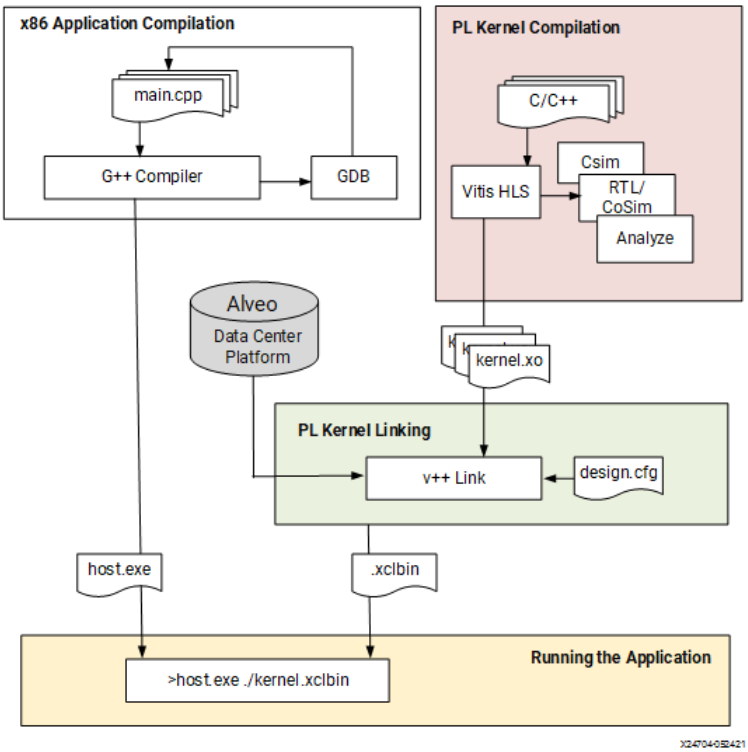

Figure 1.6: Vitis Development Flow

One or more .xo files can be paired for linking using Vitis linker to produce the .xclbin file. Xilinx object (.xo) files are linked with the target hardware platform by the Vitis linker to create a device binary file (.xclbin) that is loaded for execution on the Alveo accelerator card.

PL kernel (.xo) is a hardware function that can be added to the PL region of an extensible platform to define custom hardware. PL kernels can be defined using C++ code in Vitis HLS, or using RTL code and the IP packager feature of Vivado. Device Binary (.xclbin) file contains the programmable device image (PDI) for Versal ACAP or the bitstream for Zynq MPSoC, and metadata needed to control the hardware design.

Vitis accelerated libraries provide performance-optimized hardware functions with minimal code changes, and without the need to re-implement your algorithms to harness the benefits of Xilinx adaptive computing. Vitis accelerated libraries are available for common functions of math, statistics, linear algebra and DSP, as well as for domain specific applications, like vision and image processing, quantitative finance, database, data analytics, and data compression.

To help define the architecture of the device binary, a configuration file can be created specifying option like how many instances of a kernel (or Compute Unit) should be built in the device binary, how are the kernels connected to the global memory, or to other kernels, etc. This configuration file is passed to the Vitis linker to generate the .xclbin.

There are three different build targets of the Vitis Compiler that defines the nature and contents of the generated .xclbin file. Two emulation targets used for validation and debugging purposes: software emulation for C-based simulation, and hardware emulation for RTL cosimulation; and one hardware target for building the final project output to run on the Alveo card. The same host program can be used to run any of the .xclbin targets. Finally, when you run the application the host program loads the .xclbin file generated by Vitis Compiler. The host application always runs on the CPU and can be run in emulation mode on x86, or run on the actual physical accelerator platform.

An RTL language design, synthesis, and implementation tool that enables hardware designers to create and export hardware designs as Xilinx Support Archive (.xsa). It is a hardware container exported for multiple uses, including in a fixed or extensible platform (.xpfm). Fixed Platform includes a completed hardware design (.xsa) and supporting software files defining the operating system, libraries, and boot files. In this context, "fixed" simply means that the hardware design is complete. Extensible Platform is the target platform of the Vitis heterogeneous system design flow. In this context, the "extensible" design can be further customized by adding programmable content such as PL kernels and AI Engine graph applications to the platform to build the embedded system. Extensible Platform can also be used to develop software like the fixed platform.

1.3.1 Vitis HLS

Xilinx HLS tools provided an interface from software design to hardware design, with the objective as in our case to deploy accelerated application [8].

In the Vitis application acceleration flow, the Vitis_HLS tool automates much of the code modifications required to implement and optimize the C/C++ code in programmable logic and to achieve low latency and high throughput.

While FPGAs can be programmed using lower-level Hardware Description Languages (HDLs) such as Verilog or VHDL, there are now several High-Level Synthesis (HLS) tools that can take an algorithmic description written in a higher-level language like C/C++ and convert it into lower-level hardware description languages such as Verilog or VHDL (RTL) for implementation in the programmable logic (PL) region of a Xilinx FPGA device.

Moreover, FPGA architectures permit finer-grained control over the implementation parallelism, e.g. between tasks within a kernel or iterations of an inner loop.

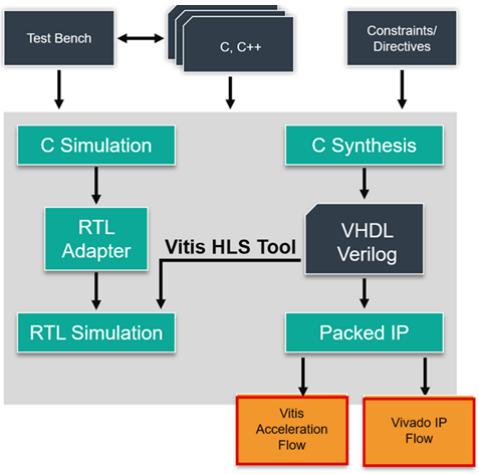

Figure 1.7: Vitis HLS Development Flow

It essentially increases the effectiveness and flexibility of Hardware design by accepting software languages that facilitate the logic design and description of complex computation as a starting point The main benefit of this type of flow is the advantages of the programming language like C/C++ to write efficient code that can then be translated into hardware.

However, achieving acceptable performance will require additional work such as rewriting the software to help the HLS tool achieve the desired performance goals.

The steps for kernel development in Vitis HLS are as follows:

1. Write the C/C++ code for the function
2. Verify the C/C++ code using C-simulation.
3. Build the kernel using C-synthesis.
4. Verify the kernel generated with C++ outputs (Co-Simulation)
5. Review the HLS synthesis reports and co-simulation reports to analyze performance
6. Repeat previous steps until performance goals are met.

Vitis HLS generates Vivado IP or Vitis Kernel based on the target flow, design constraints, and any optimization pragmas or directives you specify. You can use optimization directives to modify and control the implementation of the internal logic and I/O ports, overriding the default behaviors of the tool.

Several pragmas are supported to optimize the RTL design and explore the design space to find an optimal solution according to specific space and throughput requirements.The inference of required pragmas to produce the right interface for your function arguments and to pipeline loops and functions within your code is the foundation of Vitis HLS.

Vitis HLS is tightly integrated with both the Vivado Design Suite for synthesis, place, and route, and the Vitis core development kit for heterogeneous system-level design and application acceleration. In the Vivado IP flow, Vitis HLS also supports customization of your code to implement broader interface standards to achieve your design objectives. The RTL generated can be used as an IP directly within the Vivado tool or Model composer. The Vitis Kernel flow, which is the bottom-up design flow for the Vitis Application Acceleration Development flow instead, supports a specific set of interfaces and is more restrictive.

This more structured flow allows correct-by-construction integration of HLS blocks with Vitis extensible platforms and enables seamless integration with the Xilinx Run Time (XRT) software stack, greatly simplifying the hardware/software integration process.

1.4 Thesis structure

After seeing these key points, it's possible to understand the main objective archieve during this thesis:

Chapter 2 describe the OFDM modulator implementation, starting from the reference algorithm provided by TIM to an overview of the FFT IP provided by Xilinx; next it's discussed the C++ code in its structure but also in the usage of the pragmas to modify how the code is synthesized.

At the end of the chapter are shown the results in different reports in terms of performance (Cosimulation step) and resources allocated on board (Implementation/place and route step).

Chapter 3 describe in the same way the OFDM demodulator, underlining the main difference in the algorithm and in the code, but also in terms of performance and resources.

In Chapter 4 it is possible to see the two design previously discussed implemented using a different IP that, using Super Sample Rate, calculate the same outputs faster but using more resources.

After a quickly presentation of the IP, it's discussed how the code is being modified in both the .h and .cpp files. At the end of the chapter it's possible to analyze the advantage and disadvantage of this implementation into respect the previous one.

In Chapter 5 is explained the host code, followed by the three target option to run the design and the differences between the results in the previous step and the results from the on board tests for the different implementation analyzed.

Chapter 6 is the last chapter, where are discussed further implementation of the algorithm (GPU) and possible optimization related to the channel model implementation.

Chapter 2

Modulator Block

2.1 Matlab reference

Since the primary task of this research is the acceleration of the channel model, the main focus is the reduction of overall latency of the kernel execution. The accelerated function can be either defined in OpenCL or in C/C++.

These kernel modeling styles differ mainly in the way of defining the kernel input parameters and optimization pragmas. The same kernel can be executed by all the PEs inside a CU. For this thesis, the kernel was developed following the Vitis HLS flow.

The starting point of thesis was a model developed in C environment and co-simulated with matlab. As seen before, the modulator is the first block in the chain, followed by the FIR filter module. This function takes two arguments, the env structure (with all the parameters calculated for the model) and the input matrix, to write the result in an output matrix of different dimensions.

The parameters taken from the env structure are:

```
%   Parameters

    NFFT = env.NFFT;                % FFT size
    POL = env.POL;                  % Number of polarizations
    FS = env.FSAMPLE;               % Sampling Frequency [MHz]
    TTI_LENGTH = env.TTI_LENGTH;    % TTI duration [ms]
    LOUT = FS*TTI_LENGTH*1E3;       % Output vector size
    NSYMB_TTI = env.NSYMB_TTI;      % Number of OFDM symbols per TTI
```

```
    N1 = env.N1;
    % Number of CSI-RS positions in azimuth
    N2 = env.N2;
    % Number of CSI-RS positions in elevation
    CP_LUT = env.ICP_LUT;
    % Load look-up table for fast CP insertion
    CHANNEL_FREQ = env.CHANNEL_FREQ;
    % Application of the propagation channel in frequency domain
    P_CSI_RS = POL*N1*N2;
    %Total number of CSI-RS antenna ports
```

Listing 2.1: OFDM parameters

The OFDM modulation function starts from a matrix NxM, where N is the number of the transmitting antennas in the physical array, M is the number of symbols transmitted. For each antenna elements, it's performed the same operations as shown below:

```
for m1 = 1 : P_CSI_RS     % for each antenna port
    sig1 = tb_in(m1,:);  % extract port signal in freq domain
    sig2 = reshape(sig1,NFFT,NSYMB_TTI).';  % shape it to a matrix
                                              to exploit matrix
   capability                                 of Matlab fft
    sig3 = ifftshift(sig2,2);  % each column is a symbol
    sig4 = ifft(sig3,NFFT,2)*sqrt(NFFT);
    sig5 = reshape(sig4.',1,NFFT*NSYMB_TTI);  % from matrix to array
    tb_in(m1,:) = sig5;
    tb_out(m1,:) = tb_in(m1,CP_LUT);  % adds CP using CP_LUT
end
```

Listing 2.2: OFDM implementation

This cycle takes for each iteration a single row of the input matrix, dividing it in 14 block (Number of OFDM symbols per TTI) of 4096 elements, the FFT size. The output of this reshape it's a transposed matrix of dimension 4096x14. At this point, this matrix pass through a function called ifftshitf, just before the calculation of the Inverse Fast Fourier Transform.

$X = ifftshift(Y, dim)$ operates along the dimension dim of Y: if Y is a matrix whose rows represent multiple 1-D transforms, then ifftshift(Y, 2) swaps the halves of each row of Y. The effort of the function under analysis can be found in the FFT calculation or IFFT in case of the modulator. Matlab function $X = ifft(Y, n, dim)$ returns the inverse Fourier transform along the dimension dim, if Y is a matrix, then ifft(Y, n, 2) returns the n-point inverse transform of each row.

For OFDM processing, the necessary padding for the circular convolution is provided by adding a Cyclic Prefix rather than zero-padding the signals. In the Matlab implementation, a LUT was used to add to each block resulted from the FFT function a copy of the last N elements at the beginning of the block.

As result we have in the final matrix output of the modulator block each row of 16 block instead of the initial 14. The number of values N that the LUT copy at the beginning of each one changes block by block, thus requiring a check of the block index to select the correct value of N from the LUT.

2.2 Kernel description

2.2.1 IP description

The Xilinx LogiCORE IP Fast Fourier Transform (FFT) core implements the Cooley-Tukey FFT algorithm, a computationally efficient method for calculating the Discrete Fourier Transform (DFT) [9].

It allows Forward and inverse complex FFT (runtime configurable), fixed-point or floating-point interface, rounding or truncation after the butterfly, Block RAM or Distributed RAM for data, Bit/digit reversed or natural output order, Bit accurate C model and MEX function for system modeling. Four architecture options are available: Pipelined Streaming I/O, Radix-4 Burst I/O, Radix-2 Burst I/O, and Radix-2 Lite Burst I/O. In this thesis is considered the first one.

The FFT core computes an N-point forward DFT or inverse DFT (IDFT) where N can be $2m$, where $m = 3 - 16$. For single-precision floating-point inputs, the input data is a vector of N complex values represented as dual 32-bit floating-point numbers with the phase factors represented as 24 or 25-bit fixed-point numbers. The N element output vector is represented using the same number of bits for each of the real and imaginary components of the output data. All memory is on-chip using either block RAM or distributed RAM.

Input data is presented in natural order and the output data can be in either natural or bit/digit reversed order. The complex nature of data input and output is intrinsic to the FFT algorithm, not the implementation. The inverse FFT (IFFT) is computed by conjugating the phase factors of the corresponding forward FFT. The FFT core does not implement the 1/N scaling for inverse FFT. The scaling is therefore applied in the forward FFT, simply with conjugated phase factors (twiddle factors) [10].

Xilinx recommends always using the IFFT function in a region using dataflow optimization, because this ensures the arrays are implemented as streaming arrays. To be sure of that, the input and output arrays of the ifft are setted as stream using the HLS STREAM pragma.

To use the FFT in a C++ code :

1. Include the hls_fft.h library in the code.
2. Set the default parameters using the struct hls::ip_fft::params_t
3. Define the runtime configuration
4. Call the FFT function
5. Optionally, check the runtime status

The Xilinx FFT IP block can be called within a C++ design simply using the library hls_fft.h. Following that flow, the next step was to set the parameters structure. In our case, was preferred to create a different structure with all the static parameters settable, to be able to tune them to obtain the best performance for our goal.

```
struct param1 :  hls::ip_fft::params_t {
    static const unsigned input_width = FFT_INPUT_WIDTH;
 //Data input port width
    static const unsigned output_width = FFT_OUTPUT_WIDTH;
 //Data output port width
    static const unsigned status_width = FFT_STATUS_WIDTH;
 //Output status port width
    static const unsigned config_width = FFT_CONFIG_WIDTH;
 //Input configuration port width
    static const unsigned max_nfft = FFT_NFFT_MAX;
 //The size of the FFT data set is specified as 1 and max nfft
    static const bool has_nfft = FFT_HAS_NFFT;
 //Determines if the size of the FFT can be runtime configurable
    static const unsigned channels = FFT_CHANNELS;
 //Number of channels
    static const unsigned arch_opt = hls::ip_fft::
   pipelined_streaming_io;
 //The implementation architecture
    static const unsigned phase_factor_width = 24;
 //Configure the internal phase factor precision
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
 //The output ordering mode
    static const bool ovflo = true;
 //Enable overflow mode
    static const unsigned scaling_opt = hls::ip_fft::unscaled;
 //Define the scaling options. The Scaling option integrated in the
   IP is only available when the data format is of fixed-point type,
   so the multiplication of each element by the square root of the
   number of points is implemented outside the FFT IP
```

```
    static const unsigned rounding_opt = hls::ip_fft::truncation;
//Define the rounding modes
    static const unsigned mem_data = hls::ip_fft::block_ram;
 //Specify using block or distributed RAM for data memory
    static const unsigned mem_phase_factors = hls::ip_fft::block_ram;
 //Specify using block or distributed RAM for phase factors memory
    static const unsigned mem_reorder = hls::ip_fft::block_ram;
 //Specify using block or distributed RAM for output reorder memory
    static const unsigned stages_block_ram =
        (max_nfft < 10) ? 0 : (max_nfft - 9);
 //Defines the number of block RAM stages used in the implementation
    static const bool mem_hybrid = false;
 //When block RAMs are specified for data, phase factor, or reorder
   buffer, mem hybrid  specifies where or not to use a hybrid of
   block and distributed RAMs to reduce block RAM count in certain
   configurations
    static const unsigned complex_mult_type =
        hls::ip_fft::use_mults_performance;
 //Defines the types of multiplier to use for complex multiplications
    static const unsigned butterfly_type = hls::ip_fft::use_luts;
 //Defines the implementation used for the FFT butterfly
};
```

Listing 2.3: FFT IP paramenters structure

It's important to highlight that from the Matlab function which support matrix as input, the IP used arrays 4096x1 as input of the fft function.

In fact, both the input and output data are supplied to the function as arrays to the fft call, to be implemented on the FFT RTL block as AXI4-Stream ports. The conversion to AXI4-Stream interfaces brings standardization and enhances interoperability of Xilinx IP LogiCORE solutions. Other than general control signals such as aclk , aclken and aresetn, and event signals, all inputs and outputs to the core are conveyed on AXI4-Stream channels.

Due to the limitation of the IP, complex<float> datatype was used for inputs and outputs. In the case of floating point data, the data widths will always be 32-bit and any other specified size will be considered invalid. The internal architecture of the IP works with pseudo-floating point values, using a 24 bit mantissa and a fixed exponent to best match the available DSP unit support on the FPGA, thus losing precision in the calculation of the outputs with respect to using true floating point arithmetic.

2.2.2 C++ code and C-Simulation

The project is mainly composed by 2 source files:

modulator.hpp and **modulator.cpp**.

In the modulator.hpp file we can find the parameters structure used by the FFT IP, the typedef used in the modulator function and the definition of that function.

In the modulator.cpp file we can find different functions as shown below:

- **modulator**: This is the top level function of the project. It takes the input matrix, the number of blocks and antennas used in the simulation, optional Cyclic Prefix parameter and a destination matrix where the outputs will be write.

```
void modulator(
        int antennas,
        int blocks,
        int cp1,
        int cp2,
        Cpx inputs[ANTENNAS][MAX_OUT],
        Cpx total_fft[ANTENNAS][MAX_OUT]);
```

- **IP_config**: This function set the set the runtime configuration with desired parameters to be applied in the fft calculation. In particular the NFFT parameter set the FFT length and the direction parameter set if the operation requested is fft or ifft. A possible parameter settable is the scaling factor, but in our case can't be used because of the float datatype.

```
void IP_config(config_t &fft_config) {
//set up the parameters struct

    config_t *config;
    config = &fft_config;

    config->setNfft(12);
 // Set FFT length to 512 => log2(512) =>9
    config->setDir(false);
 // set FFT/IFFT
    config->setSch(0x77F);
 //scaling factor, valid only in scaled mode fixed point
}
```

- **ifftshape**: This function replicate the same Matlab function, taking the inputs from the matrix block by block (4096 elements per block, FFT length), saving them starting from the second half of the block in the first half of the destination array and the first part in the rest of the array elements.

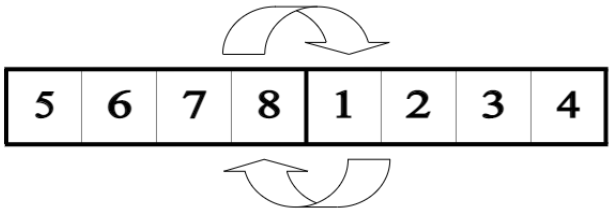

Figure 2.1: ifftshpe function explanation

- **Cyclic Prefix**: This function implement the LUT used in matlab to copy the end section of the resultant block to the beginning of the same block in the result matrix. The number of elements copied per block is decided using the value in the LUT. In this case the values are the same for almost all the block except 2 of them, so it's used 3 fixed values in a switch/case statement. In the same switch we set an offset value to write the block starting from the correct index of the matrix.

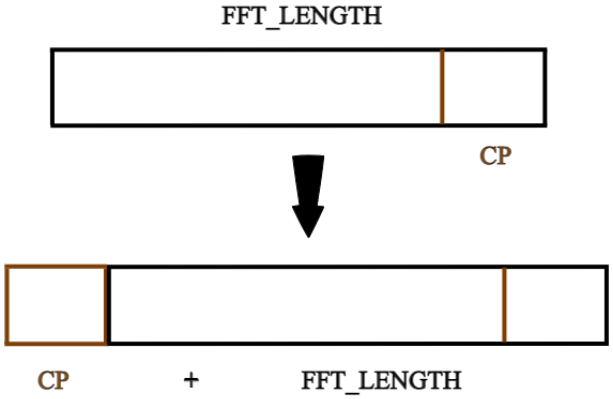

Figure 2.2: CP function explanation

Since the Xilinx FFT IP does not work with a matrix as input, we had to use 2 nested loops to emulate the matrix reshape in the Matlab code.

Inside the nested loop we call in order the ifftreshape function to order correctly the inputs in a single block variable; at this point the fft function was called from the IP library with the parameters structure as argument; the resulting outputs are stored in the output matrix in the Cyclic Prefix function, adding the needed samples to have each row made by 16 block of 4096 samples.

To test the coherence between the fft function used by matlab and the function provided by Xilinx IP was created a specific testbench that works with the same inputs and calculate how many match/mismatch are obtained comparing the outputs with the reference matrix from matlab (absolute error less than 10^{-6}.

It also calculate the average err for real and imaginary part and the maximum error when mismatch happens.

The testbench is made of different files:

- **bin file.h**: This file contain a function used to read the input matrix from a binary file generated on Matlab from a set provided by TIM. The complex numbers are red column by column and saved in a vector; this vector is used to generate the input matrix such as the other matrix, using a nested loop.

- **main.cpp**: This is the testbench file, where the files with the real and imaginary part of the inputs are red and stored in a complex matrix, the top level function is called and then the outputs are compared with a golden reference sequence taken from matlab. In this last part we calculate the matching percentage (using as error value 0.000001, same value used in the Channel implementation) the maximum error and the average error for both real and imaginary part.

- binary files with reference values generated on Matlab.

2.2.3 C-Synthesis results

Design synthesis is the phase where the code is translated into actual circuit with lower level implementation such as gates, flip flops, adders. The input design is converted into net-list which describes components used and interconnections among them.

The process of design synthesis starts with syntax check once it is provided with HDL based design as input. The logic is then optimized by using different optimization techniques such as redundant logic elimination, logic reduction, size reduction and simultaneously making its implementation faster.

This leads to the integration of pragmas for design optimization such as reduction of latency and maximization of throughput, control of the resources of the final RTL code. These pragmas are direct to be inserted in the source code and directly interpreted by the synthesis tool. The optimization directives are embedded into the C source code. If the optimization directives are embedded in the code, they are automatically applied to every solution when re-synthesized.

Pragmas used in the modulator project are:

- **#pragma HLS INTERFACE**:

  Used to define the top level interface of the kernel. Kernels in the Vitis development flow, support four types of interfaces. In our case, we prefer to use the AXI4 Memory Mapped (M_AXI) for access from the kernel to global memory or host memory. Data is accessed by the kernel through memory, in this case DDR.

- **#pragma HLS STREAM**:

  On Vitis HLS 2022, is possible to use a type of variable called stream, that can improve the management of the data and speed up the calculation of the FFT, but in the used version (2021.2) the IP is not compatible with inputs as hls:stream datatype. A possible solution is to use the pragma HLS STREAM to implement the interface of the FFT block as stream.

  A stream is an important abstraction: it represents an unbounded, continuously updating data set, where unbounded means “of unknown or of unlimited size”. A stream can be a sequence of data (scalars or buffers) flowing unidirectionally between a source (producer) process and a destination (consumer) process.

The streaming paradigm forces you to think in terms of data access patterns (or sequences). It lets define the algorithm in a sequential manner and the parallelism is extracted through other means (such as by the compiler).

Complexities like synchronization between the tasks are abstracted away. It allows the producer and the consumer tasks to process data simultaneously, which is key for achieving higher throughput. Another benefit is cleaner and simpler code.

- **#pragma HLS DATAFLOW**:

  The dataflow pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and increasing the overall throughput of the design.

  When the dataflow pragma is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.

  If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, the HLS tool attempts to minimize the initiation interval and start operation as soon as data is available. The dataflow optimization has no hierarchical implementation.

  For the dataflow optimization to work, the data must flow through the design from one task to the next to prevent the HLS tool from performing the dataflow optimization. Other code styles to avoid with the dataflow are:

  - Single producer-consumer violations
  - Feedback between tasks
  - Conditional execution of tasks
  - Loops with multiple exit conditions

  If a sub-function or loop contains additional tasks that might benefit from the dataflow optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function.

- **#pragma HLS TRIPCOUNT**:

  When manually applied to a loop, specifies the total number of iterations performed by a loop. The Vitis HLS tool reports the total latency of each loop, which is the number of clock cycles to execute all iterations of the loop. Therefore, the loop latency is a function of the number of loop iterations, or tripcount.

  The tripcount can be a constant value or a range setting a minimum and maximum value. It can depend on the value of variables used in the loop expression or depend on control statements used inside the loop. In some cases, the HLS tool cannot determine the tripcount, and the latency is unknown.

  This includes cases in which the variables used to determine the tripcount are input arguments or variables calculated by dynamic operation. The LOOP_TRIPCOUNT pragma or directive is for analysis only, and does not impact the results of synthesis.

- **#pragma HLS PIPELINE**:

  This pragma is not used directly in this code, but during the synthesis process there are default optimization applied where possible. Pipeline is one of the most used optimization implemented by default, trying to reduce at the minimum value the Initiation Interval (II).

  A pipelined function or loop can process new inputs every N clock cycles, where N is the II of the loop or function. An II of "1" processes a new input every clock cycle. You can specify the initiation interval through the use of the II option for the pragma.

  As a default behavior, with the PIPELINE pragma or directive Vitis HLS will generate the minimum II for the design according to the specified clock period constraint. If the Vitis HLS tool cannot create a design with the specified II, it issues a warning and creates a design with the lowest possible II.

At the end of the synthesis, can be found a full report with the resulting RTL and how is composed. In particular, the estimated values for Timing, Performance and Resources are reported and analyzed.

The first part gives us an overview of the timing considering the target platform. The uncertainty is because Vitis HLS uses internal models to estimate the delay of each operation. That cannot take into account in advance all the possible increases or decreases of these delays in the real RTL synthesis, with subsequent place and route, unless there is a margin of error.

Target	Estimated	Uncertainly
10.00 ns	7.300 ns	2.700 ns

Table 2.1: Synthesis Report of Timing (modulator)

In following table are reported the values relatives to the estimated performance.

Modules/Loops	Latency	II	Trip Count	Pipelined
modulator	5572929	5572930	-	no
VITIS_LOOP_84_1	5572928	-	32	dataflow
dataflow_parent_loop	174153	174153	-	no
VITIS_LOOP_86_2	174152	-	14	dataflow
dataflow_in_loop	12574	12429	-	dataflow
ifftshape	4246	4246	-	no
ifftshape_Pipeline_1	2051	2051	-	no
VITIS_LOOP_43_1	2049	1	2048	yes
ifftshape_Pipeline_2	2051	2051	-	no
VITIS_LOOP_46_2	2049	1	2048	yes
fft_param1_s	12428	12429	-	dataflow
fft_param1_proc5	0	0	-	no
fft_syn_param1_32	12428	12428	-	no
fft_param1_proc6	0	0	-	no
entry_proc	0	0	-	no
Cyclic_Pre	4577	4577	-	no
Cyclic_Pre_Pipeline_1	4105	4105	-	no
VITIS_LOOP_29_1	4103	1	4096	yes
Cyclic_Pre_Pipeline_2	329	329	-	no
VITIS_LOOP_33_2	327	1	320	yes

Table 2.2: Synthesis Report of Performances (modulator)

To have a better comprehension of the report, are explained the meaning of each parameter shown in the table:

- **Latency (Max)**

  This is the maximum latency relative to the loop in terms of clock cycles. The minimum latency is not reported because not considered as significant. This can be found in the full synthesis report generated by Vitis_HLS. For each estimate, the worst case (so the max latency value) will always be the reference one. The number incorporates the latency of the innermost loop.

- **Initiation Interval (II)**

  This is a fundamental parameter in the concept of pipelining. Usually abbreviated with II, it represents the number of cycles necessary for the pipeline to be able to take a new input and process it. The default target is 1: an II = 1 means that the pipeline can be fed by a new input at each clock cycle. That represents perfection in terms of execution.

  Every resource within the pipeline will always and continuously be used without any waste of clock cycles. For waste, it is meant the failure to use a resource for one or more cycles. If there is no number, but the character - means that the loop is not pipelined but executed regularly.

- **Trip Count**

  The number of iterations a loop completes

- **Pipelined**

  Indicates that the function or loop are pipelined in the RTL design.

The second part of the report shows the estimated resources to implement the design on the target FPGA.

Modules/Loops	BRAM	DSP	FF	LUT	URAM
modulator	86	28	28596	25240	0
VITIS_LOOP_84_1	-	-	-	-	-
dataflow_parent_loop_proc	52	28	25620	22168	0
VITIS_LOOP_86_2	-	-	-	-	-
dataflow_in_loop	52	28	24963	22001	0
ifftshape	0	0	2227	1599	0
ifftshape_Pipeline_1	0	0	981	124	0
VITIS_LOOP_43_1	-	-	-	-	-
ifftshape_Pipeline_2	0	0	981	124	0
VITIS_LOOP_46_2	-	-	-	-	-
fft_param1_s	48	0	19967	16308	0
fft_param1_proc5	0	0	3	29	0
fft_syn_param1_32	48	0	19874	16196	0
fft_param1_proc6	0	0	2	20	0
entry_proc	0	0	3	47	0
Cyclic_Pre	0	28	2421	3655	0
Cyclic_Pre_Pipeline_1	0	0	351	126	0
VITIS_LOOP_29_1	-	-	-	-	-
Cyclic_Pre_Pipeline_2	0	0	369	151	0
VITIS_LOOP_33_2	-	-	-	-	-

Table 2.3: Synthesis Report of Resources (modulator)

As done before, follows a brief explanation of the type of resource reported:

- Block RAMs (BRAM) are used for storing large amounts of data inside FPGA.
- The Digital Signal Processors (DSP) are usually distributed blocks of very wide hardware accumulators and multipliers which are able to perform MAC and SIMD operations very quickly on wide data inputs.
- Flip-flops (FF) are used to create registers, which store data
- The LUT in an FPGA holds a custom truth table, which is loaded when the chip is powered up
- URAM is a special kind of memory that is wider and deeper than the BRAM and can be used to store large data structures

2.2.4 Co-Simulation results and waveform

The Cosimulation step check that the generated RTL has the same output of the C++ code when stimulated with the same inputs. When we run the cosimulation, there are some option to set the desired dump trace (none, port or all) and extra option for Dataflow. Using the waveform viewer it's possible also to check the flow of the signal in case of failed test.

Cosimulation to verify the correctness of the generated code comparing the result of source C++ code with the generate HDL. If you added a C testbench to the project for simulation purposes, you can also use it for C/RTL cosimulation to verify that the RTL is functionally identical to the C source code.

The C/RTL verification process consists of three phases:

The C simulation is executed and the inputs to the top-level function, or the Design-Under-Test (DUT), are saved as "input vectors". The "input vectors" are used in an RTL simulation using the RTL created by Vitis HLS in Vivado simulator, or a supported third-party HDL simulator.
The outputs from the RTL, or results of simulation, are saved as "output vectors." The "output vectors" from the RTL simulation are returned to the main() function of the C test bench to verify the results are correct.

The C test bench performs verification of the results, in some cases by comparing to known good results. Additionally, C/RTL co-simulation automatically verifies aspects of the DATAFLOW and DEPENDENCE pragmas or directives. If those outputs are the same and the other checks are ok, the Test is passed and it's possible to go on with the Implementation step.

These results differs from values reported after HLS synthesis, which are based on the absolute shortest and longest paths through the design. The results provided after C/RTL co-simulation show the actual values of latency and II for the given simulation data set (and may change if different input stimuli is used).

The next step is to map this circuit description to the actual locations on the target device to reduce the length of the critical paths. The final stage is encoding the circuit description into a binary format (bitstream), which is then used to configure the FPGA on-chip resources and define the initial on-chip static random-access memory (SRAM) contents.

2.2.5 Implementation results

The last step in Vitis HLS is the Implementation and Place and Route.

The Vitis HLS tool is limited in terms of the estimations it can provide about the RTL design that it generates. It can project resource utilization and timing of the end result, but these are just projections. The Implementation Report contains the results of Synthesis and Place and Route.

Resources	Verilog
SLICE	0
LUT	13072
FF	19790
DSP	70
BRAM	65
URAM	2
LATCH	0
SRL	3039
CLB	3003
Timing	
CP required	10.000
CP achieved post-synthesis	6.567
CP achieved post-implementation	7.416

Table 2.4: The Resource Usage and the Final Timing (modulator)

The Resource Usage and the Final Timing sections show a quick summary of the resources and timing achieved by either the RTL Synthesis run or the Place & Route run. These sections give a very high-level overview of the resource utilization and status on whether timing goals were met or not.

The information in the succeeding sections provide details useful in debugging timing issues. A detailed per-module split up of resources is provided to in-deep optimization.

The **fail fast** reports that Vivado provides can guide your investigation into specific issues encountered by the tool. In the fail fast report, you should look into anything with the Status of REVIEW to improve the implementation and timing closure. Different sections of the fail fast report are shown in the follow table:

Criteria	Guideline	Actual	Status
LUT	70%	1.00%	OK
FD	50%	0.76%	OK
LUTRAM+SRL	25%	0.55%	OK
CARRY8	25%	0.36%	OK
MUXF7	15%	0.05%	OK
DSP	80%	0.78%	OK
RAMB/FIFO	80%	1.61%	OK
URAM	80%	0.21%	OK
DSP+RAMB+URAM (Avg)	70%	0.87%	OK
BUFGCE + BUFGCTRL	24	0	OK
DONT_TOUCH (cells/nets)	0	0	OK
MARK_DEBUG (nets)	0	0	OK
Control Sets	24444	196	OK
Average Fanout for modules > 100k cells	4	0	OK
Non-FD high fanout nets > 10k loads	0	0	OK
TIMING-6 (No comm primary clock)	0	0	OK
TIMING-7 (No comm node)	0	0	OK
TIMING-8 (No comm period)	0	0	OK
TIMING-14 (LUT on the clock tree)	0	0	OK
TIMING-35 (No comm node in the paths)	0	0	OK
Number of paths above max LUT bud (0.300ns)	0	0	OK
Number of paths above max Net bud (0.208ns)	0	0	OK

Table 2.5: Fail Fast Report (modulator)

This table can be divided in 3 parts:

- Design Characteristics: The default utilization guidelines are based on SSI technology devices and can be relaxed for non-SSI technology devices. Designs with one or more REVIEW checks are feasible but are difficult to implement.
- Clocking Checks: These checks are critical and must be addressed.
- LUT and Net Budgeting: Use a conservative method to better predict which logic paths are unlikely to meet timing after placement with high device utilization.

The Timing Paths reports show the timing critical paths that result in the worst slack for the design. By default, the tool will show the top 10 worst negative slack paths. Each path in the table has detailed information that shows the combination path between one flip-flop to another.

Breaking these long combinational paths will be required to address the timing issues. So you need to analyze these paths and reason where they are coming from and map these paths back to the user's C code. Using both these paths and the resources table presented earlier can help in determining and correlating the path back to your source code.

In the next figure it's possible to see the timing report for different path of the modulator:

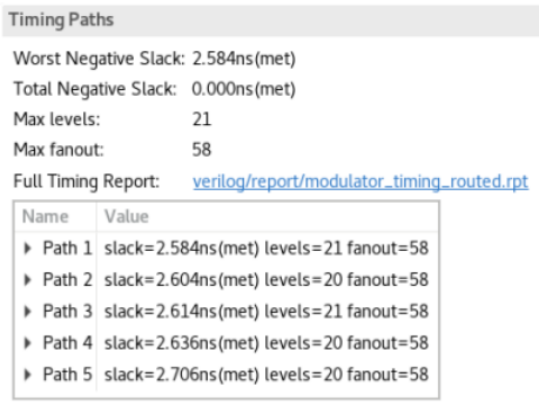

Figure 2.3: Place & Route Timing Paths Report (modulator)

Chapter 3

Demodulator Block

3.1 Matlab reference

Move to the other end of the channel link, it's needed a demodulation step to obtain the original data sent. In the Matlab implementation, the demodulator is the last block in the chain.

This function proceeds inversely with respect to the modulator seen before, using the same env structure for the parameters needed and matrices of variable dimension based to the number of receiving antennas.

```
% Vector lengths
    LIN = FS*TTI_LENGTH*1E3;        % Input vector size
    LOUT = NFFT*NSYMB_TTI;          % Output size after CP removal
        % CP removal and FFT operation
        tb_out = zeros(P_CSI_RS,LOUT);
        for m1 = 1 : P_CSI_RS
            temp = tb_in(m1,1:LIN);
            sig1 = temp(CP_LUT);
            sig2 = reshape(sig1,NFFT,NSYMB_TTI).';
            sig3 = fft(sig2,NFFT,2)*(1/sqrt(NFFT));
            sig4 = ifftshift(sig3,2);
            sig5 = reshape(sig4.',1,NFFT*NSYMB_TTI);
            tb_out(m1,:) = sig5;
        end
```

Listing 3.1: Demodulator implementation

The demodulation function starts removing the cyclic prefix from the input matrix, simply creating a smaller matrix to be processed by the FFT function. As before, a reshape is needed before the fft function.

Matlab FFT function, as the Inverse one, performs the Fourier transform of a matrix along the dimension dim in this case. In this case, the result is divided by the scaling factor.At this point, this matrix pass through a function called ifftshitf as last step, to order correctly each block. The output of this it's reshaped and transposed to obtain a matrix of correct dimension.

3.2 Kernel description

3.2.1 C++ code and C-Simulation

The C++ code is quite simpler respects to the modulator one.

The IP used for this project is the same used for the modulator, setting the parameters accordingly to the different task to be performed (direction = true to perform FFT instead of IFFT). Inverting the cyclic prefix function with the fftshape function reduce the complexity of the first function, because this time we simply ignore the cyclic prefix values when the inputs are red, to store the values from the CPst element of each block.

In this case, the scaling factor is moved to the fftshpe function, a symmetrical function respect to the ifftshape discussed previously, where each element of the block is divided by the scaling factor $\sqrt{FFT\_LENGTH}$ and the result directly stored in the correct index of the array, considering the swap as in figure 2.1.

The main difference is in the top-level interface: in the modulator the length of each row in the input matrix is smaller than the output one due to the CP operation; in this case is the exactly opposite, with the input matrix bigger respect the result.

3.2.2 C-Synthesis results

There are no differences regards the pragmas usage in the demodulator code. The top-level function works in a dataflow region, where the top level interface is set as seen before using **#pragma HLS INTERFACE mode=m_axi port=input/output bundle=gmemX**, where the bundle divide the input and the output into two different memory bank (DDR0 and DDR1).

Using the same IP to perform the FFT, it's also used the same stream mode for the input/output arrays of the FFT function. To have a correct estimation in the report, **#pragma HLS loop_tripcount** is used in the nested loop at top-level to have a max and min value for the number of iteration in each loop.

Moving to the synthesis reports, we start with the same timing report of the modulator (default settings are used to set the clock). Due to the different in the algorithm we obtain a top-level latency of 5570689 cycles, same order of magnitude of the modulator result but 2000 cycles less in the estimated maximum latency (cycles).

Modules/Loops	Latency	II	Trip Count	Pipelined
demodulator	5570689	5570690	-	no
VITIS_LOOP_69_1	5570688	-	32	dataflow
dataflow_parent_loop	174083	174083	-	no
VITIS_LOOP_71_2	174082	-	14	dataflow
dataflow_in_loop	12504	12429	-	dataflow
Cyclic_Pre	4173	4173	-	no
Cyclic_Pre_Pipeline_1	4169	4169	-	no
VITIS_LOOP_29_1	4167	1	4096	yes
fft_param1_s	12428	12429	-	dataflow
fft_param1_proc4	0	0	-	no
fft_syn_param1_32	12428	12428	-	no
fft_param1_proc5	0	0	-	no
entry_proc	0	0	-	no
fftshape	4255	4255	-	no
fftshape_Pipeline_1	2057	2057	-	no
VITIS_LOOP_38_1	2055	1	2048	yes
fftshape_Pipeline_2	2057	2057	-	no
VITIS_LOOP_40_2	2055	1	2048	yes

Table 3.1: Synthesis Report of Performances (demodulator)

Similar result are obtained in the resources report, where the FF and LUT usage decrease of 2500 each, as shown in the next figure.

Modules/Loops	BRAM	DSP	FF	LUT	URAM
demodulator	82	29	26882	25992	0
VITIS_LOOP_69_1	-	-	-	-	-
dataflow_parent_loop_proc	52	29	25058	23874	0
VITIS_LOOP_71_2	-	-	-	-	-
dataflow_in_loop	52	29	24401	23707	-
Cyclic_Pre	0	7	987	3292	0
Cyclic_Pre_Pipeline_1	0	0	795	2430	0
VITIS_LOOP_29_1	-	-	-	-	-
fft_param1_s	48	0	19967	16308	0
fft_param1_proc5	0	0	3	29	0
fft_syn_param1_4096	48	0	29381	28472	0
fft_param1_proc6	0	0	2	20	0
entry_proc	0	0	3	29	0
fftshape	0	22	3109	3745	0
fftshape_Pipeline_1	0	0	863	564	0
VITIS_LOOP_43_1	-	-	-	-	-
fftshape_Pipeline_2	0	0	863	564	0
VITIS_LOOP_46_2	-	-	-	-	-

Table 3.2: Synthesis Report of Resources (demodulator)

3.2.3 Cosimulation results and waveform

Cosimulation report shows how the synthesis estimation set the order of magnitude, but are not enough precise to be used. Calling more than one time the top-level function in the testbench we can have a range of II for the different modules, as shown in the following table.

Modules/Loops	Avg II	Max II	Min II
demodulator	7938815	7938815	7938815
VITIS_LOOP_84_1	7938815	7938815	7938815
dataflow_parent_loop_proc	247891	260487	247688
VITIS_LOOP_84_1	247891	260487	247688
dataflow_in_loop	17706	30491	17692
Cyclic_Pre	17706	30491	17692
Cyclic_Pre_Pipeline_1	17706	30491	17692
VITIS_LOOP_29_1	17706	30491	17692
fft_param1_s	17725	35043	13140
fft_param1_proc5	17725	35043	13140
fft_syn_param1_4096	17735	70425	4177
fft_param1_proc6	17735	70426	4177
entry_proc	17706	30491	17692
ifftshape	17735	70502	4252
ifftshape_Pipeline_1	17735	70502	4252
VITIS_LOOP_43_1	17735	70502	4252
ifftshape_Pipeline_2	17669	63837	4252
VITIS_LOOP_46_2	17669	63837	4252

Table 3.3: Cosimulation II Estimates (demodulator)

The main difference is related to the fft module, because the synthesis estimate the latency of a black-box module instead of cosimulation that use the actual RTL netlist to calculated a more accurate value.

The latency is much higher respect the previous value. It's depends from the different estimation of the latency for each module.

Modules/Loops	Avg Latency	Max Latency	Min Latency
demodulator	7938788	7938788	7938788
VITIS_LOOP_84_1	7938789	7938789	7938789
dataflow_parent_loop_proc	299247	300498	260460
VITIS_LOOP_84_1	17706	300499	260461
dataflow_in_loop	70323	70502	30464
Cyclic_Pre	17691	17691	17691
Cyclic_Pre_Pipeline_1	17688	17688	17688
VITIS_LOOP_29_1	17689	17689	17689
fft_param1_s	53011	70426	12795
fft_param1_proc5	17621	35042	0
fft_syn_param1_4096	17719	70425	4177
fft_param1_proc6	17718	70425	4176
entry_proc	17310	17427	0
ifftshape	17719	70501	4251
ifftshape_Pipeline_1	8861	61598	2055
VITIS_LOOP_43_1	17735	61599	2056
ifftshape_Pipeline_2	8717	8762	2055
VITIS_LOOP_46_2	8718	8763	2056

Table 3.4: Cosimulation Latency Estimates (demodulator)

3.2.4 Implementation results

The reports from this step gives us coherent results with the modulator one, where demodulator use less LUT but a bit more FF and BRAM of the modulator due the different dimension of the input matrix.

Resources	Verilog
SLICE	0
LUT	12493
FF	20143
DSP	70
BRAM	79
URAM	2
LATCH	0
SRL	2844
CLB	2842
Timing	
CP required	10.000
CP archieved post-synthesis	6.567
CP archieved post-imolementation	7.905

Table 3.5: The Resource Usage and the Final Timing (demodulator)

Talking about the timing, as seen in the cosimulation step the value post-implementation is higher repsect the modulator one (starting from the same value post-synthesis).

In the timing Paths report we can see that the worst negative slack is lower considering the same levels and fanout. The values for the paths are reported in the following figure.

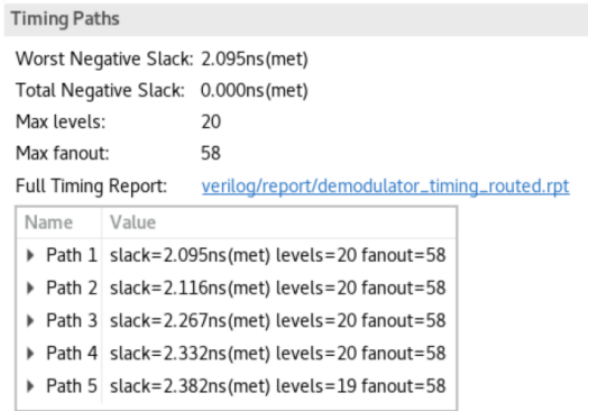

Figure 3.1: Place & Route Timing Paths Report (demodulator)

Chapter 4

1D SSR FFT IP

4.1 IP description

To overcome possible problems due to the limitation of the FFT provided by Xilinx (pseudo-floating point unit, single precision data, ect...), we explored some different options in terms of IPs, where the SSR FFT IP enter in the project [11].

Unlike the traditional FFT where the input/output vectors are given in 1-dimensional, we introduce the super sample rate (SSR) in our HLS FFT design for boosting the FPGA acceleration, so the input/output are transformed into arrays for FPGA that can easily consume the input samples within a single column in 1 cycle [12].

Let's take a 16 points (the shortest length allowed for HLS FFT) FFT for example:

If we describe the 1-D input vector as:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

And the attended result as:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

If the SSR is set to 2, $inData[SSR][FFT\_LEN/SSR]$ should be like:

0	1	2	3	4	5	6	7
a b	c d	e f	g h	i j	k l	m n	o p

Table 4.1: Input arrays order

The output array from HLS FFT should be:

0	1	2	3	4	5	6	7
A B	C D	E F	G H	I J	K L	M N	O P

Table 4.2: Output arrays order

The parameters structure of this IP is quite different respects the previous one as show below:

```
struct fftParams : ssr_fft_default_params {
    static const int N = SSR_FFT_L;
    static const int R = SSR_FFT_R;
    static const fft_output_order_enum output_data_order =
   SSR_FFT_NATURAL;
    static const transform_direction_enum transform_direction =
   REVERSE_TRANSFORM;
    static const butterfly_rnd_mode_enum butterfly_rnd_mode =
   CONVERGENT_RND;
    static const int twiddle_table_word_length=18;
    static const int twiddle_table_intger_part_length=2;
};
```

Listing 4.1: SSR IP FFT paramenters structure

Also while synthesizing floating point 1-D SSR FFT the parameters in the structure which carry information such as scaling mode, twiddle factor storage bits, butterfly rounding mode etc. which are only related to fixed point data, carry no meaning. Instead SSR FFT parameter structure can simply define relevant parameters as shown below.

```
struct fft_parameters{
        static const int N = 4096;
        static const int R = 16;
        static const fft_output_order_enum output_data_order =
    SSR_FFT_NATURAL;
        static const transform_direction_enum transform_direction =
    REVERSE_TRANSFORM;
        }
```

Listing 4.2: Simplified structure for float data

1-D SSR FFT also supports synthesis for single or double precision floating point type. For synthesizing a complex floating point type, it is required that std::complex type not to be used as a complex wrapper.

Since this wrapper has some issues, it is required that a wrapper class provided with the Vitis DSP library called complex_wrapper<...> is used for wrapping complex float numbers.

4.2 Code adjusstment

To use this IP were required different changes in the modulator code:

- add an fft_top function, that order and manage the stream of inputs/outputs from the fft function.
- change of the type, using a custom output datatype for the result of the fft, converted to complex float in the Cyclic Prefix function.
- complex wrapper datatype replace the complex type to have synthetizable code
- scaling loop due limitation of the complex wrapper datatype.

Those step are applied for both the SSRmodulator and SSRdemodulator kernels.

The SSRdemodulator have the same Cyclic prefix of the other demodulator simply with the different complex type required by the IP.

The fftshift function instead need a temp variable to scale separately the real and imaginary part of each element in the resultant array and then store in the correct order the complex data from this variable to the output matrix.

The testbench used for the previous designs is the same used for with this IP without any changes. With those accommodations in the code, we obtain the same result from C sim respects to the Xilinx IP.

4.3 Reports Analysis

In the synthesis step we can see the main differences between the 2 IPs:

The latency is quite lower because the resources used for the internal FFT are some order of magnitude higher than the other IP as shown in the following tables.

Target	Estimated	Unvertainly
15.00 ns	10.950 ns	4.05 ns

Table 4.3: Synthesis Report of Timing (SSRmodulator)

In particular, those reports shows that this IP is implemented in 3 nested fftStages as show in figure 4.1.

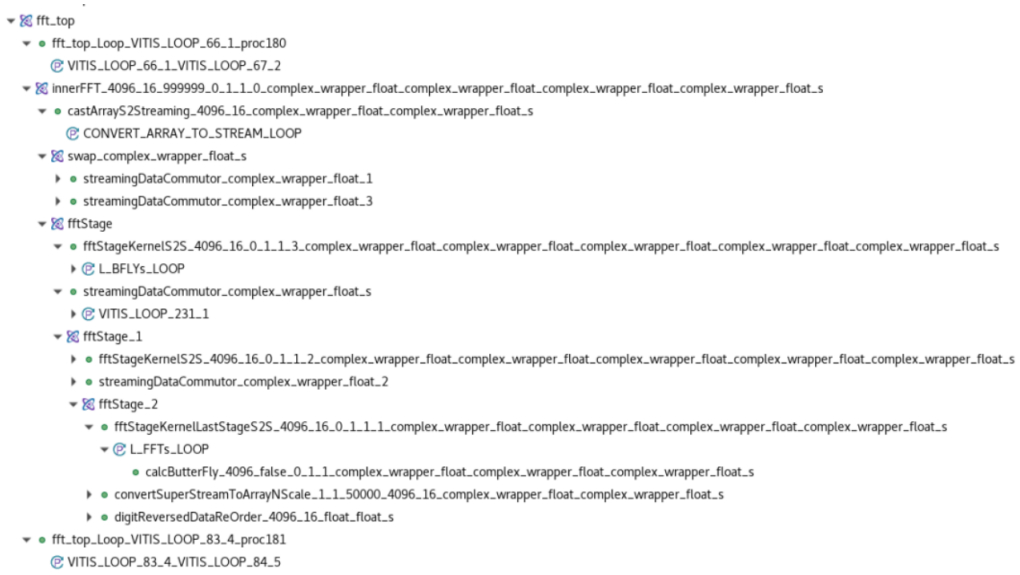

Figure 4.1: Modules/Loops for FFT block in SSRmodulator

Considering the Performance report of the synthesis, respects the modulator shown before we have no result for the innerFFT block, because in this step the tool can't estimate the latency of an external library.

An important difference into respect the synthesis with the other IP is the application of the auto-rewind in the pipelined loops inside the IFFT block.

The rewind feature means a continuous loop pipelining with no pause between one loop iteration ending and the next iteration starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function.

In terms of Resources instead, the results indicates for the top-level function are:

Modules/Loops	BRAM	DSP	FF	LUT	URAM
SSRmodulator	246	3774	410771	401736	2

Table 4.4: Synthesis Report of top-level Resources (SSRmodulator)

The usage of resources respects the modulator is huge due to the fact that the IP is a DSP based implementation. From the SSRdemodulator top-level, we obtain similar results shown in the next table.

Modules/Loops	BRAM	DSP	FF	LUT	URAM
SSRdemodulator	246	3678	405243	396823	2

Table 4.5: Synthesis Report of top-level Resources (SSRdemodulator)

In the SSRdemodulator reports, we have similar results as seen expected after the description of the main difference between modulator and demodulator with the first IP.

The Implementation step with the place and route reports gives as an overview of the main differences in the IPs implementation:

This is a better estimation respects the synthesis one, because LUT, FF, DSP and BRAM usage is reduced significantly. This is not sufficient to have an ammount of resources usage comparable with the other IP as shown in the following table.

	Verilog
SLICE	0
LUT	347802
FF	200553
DSP	3184
BRAM	187
URAM	2
LATCH	0
SRL	22048
CLB	67483
CP required	15.000
CP archieved post-synthesis	7.569
CP archieved post-imolementation	11.734

Table 4.6: The Resource Usage and the Final Timing (SSRmodulator)

The same consideration can be extended to the Final timing part. In the next figure it's possible to see the timing report for different path of the SSRmodulator, where is possible to see that, considering the different target clock period used to avoid timing violation, the Worst Negative Slack (WNS) is higher than the modulator but still met.

Similar results are obtained for the SSRdemodulator in therm of resources, where all the values are very close to those in the SSRmodulator report.

As commented in the demodulator chapter, the final timing shows that the post-implementation value is higher than the corresponding SSRmodulator value, starting even in this case from the same post-synthesis value.

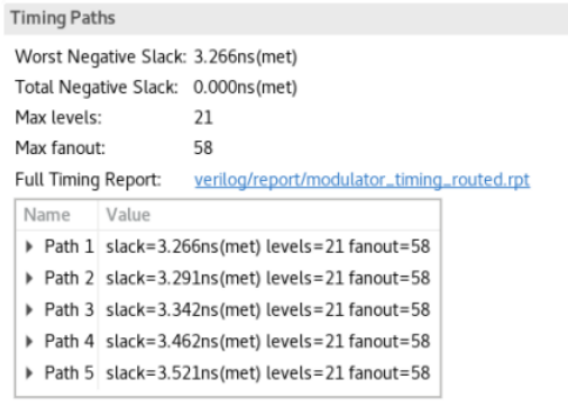

Figure 4.2: Place & Route Timing Paths Report (SSRmodulator)

Very different WNS (but met in both cases) are shown in the last part of the report: we have $WNS = 3.266ns$ for the SSRmodulator but $1.895ns$ for the SSRdemodulator. The explaination for this divergence is in the maximum levels analized. In the first case we have a similar situation of the previous projects, with the values shown in the figure above; in the second report we have only 7 levels considered with a $maxfanout = 1024$.

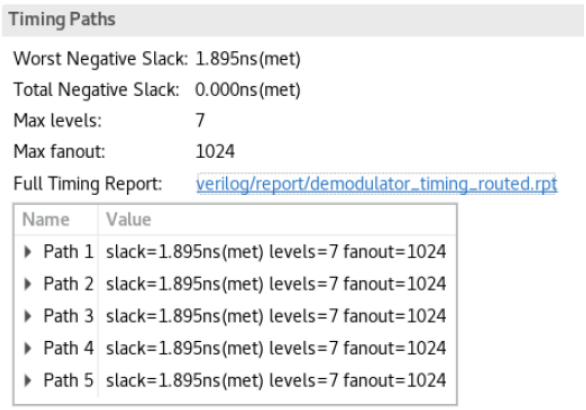

Figure 4.3: Place & Route Timing Paths Report (SSRmodulator)

Chapter 5

On Device Deployment

5.1 Host code

In the Vitis environment, the software application can be written in native C++ using the Xilinx runtime (XRT) native API for optimized interfacing with Xilinx Devices. The Host code is the part which sets up the environment and controls data movement to and from the accelerator device and is executed on a general purpose CPU.

In general, the structure of the host application can be divided into the following steps:

- Loading the .xclbin generated into the program. The device.load_xclbin (binaryFile) command is used to load the kernel binary.
- Create xrt::kernel objects from the loaded device binary, and associate buffer objects (xrt::bo) with the memory banks assigned to kernel arguments.
- Create the input test data and map the buffers to the host memory
- Setting up the kernel and kernel arguments.
- Transfer data back and forth from the host application to the kernel using xrt::bo::sync commands and buffer reads and write commands.
- Execute the kernel using an xrt::run object to start the kernel and wait for kernel execution.
- Verification step comparing the returned outputs in the host with a reference.

Following this list, the host.cpp was written dividing the file in 2 functions: In the main function there are the device load command and other preliminary operations. After that, the function run_krnl is called. In this function can be found the core operation of the host code, starting from buffers allocation into the device memory, synchronization of the buffer content with the device and the kernel execution.

At the end, two verification steps are used to validate the operation, the first one to check the correctness of the results that comes back to the host as done in the testbench on Vitis HLS and the second report the throughput achieved.

The first step of optimization is burst reading from off-chip memory and writing to on-chip one since the input data read from the interface undergoes iterative operations. On-chip memory (BRAM or URAM) can be partitioned in a way that computation or copying can be performed in parallel. Thus the latency is reduced.

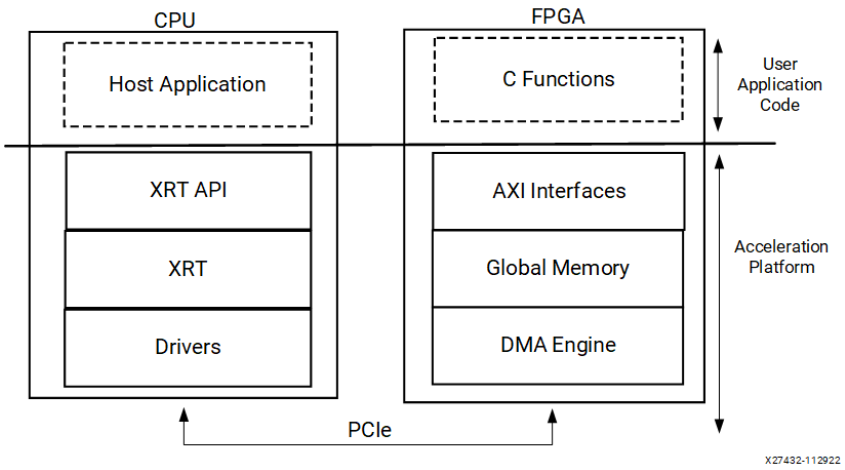

Figure 5.1: Host/Device interface

The host code compilation results into a host executable application. The FPGA specific code called kernel is compiled using Xilinx "xocc" compiler. The compiler generated object files are then combines with FPGA platform files producing .xclbin binary file. With source code in C/C++, supported by the same compiler as already stated, the Vitis environment fully supports HLS optimization techniques.

The used software provide three different types of build targets: **Software Emulation, Hardware Emulation and Hardware Implementation**.

The code functionality verification and source-level debugging can be performed in a much faster way using Software Emulation. Hardware Emulation gives more accurate view of RTL implementation behaviour. The Hardware Implementation simulate the board using the platform files and generate the bitstream file needed to execute the kernel on board [13].

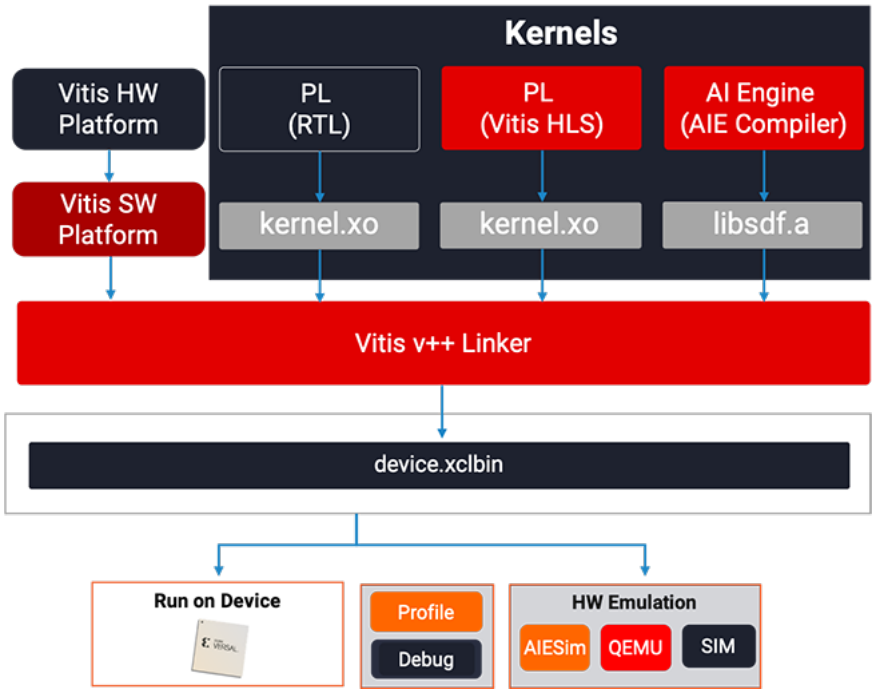

Figure 5.2: Vitis unified software development flow

In the following paragraph are reported the results for the 3 different targets for each project.

5.2 Modulator

5.2.1 SW Emulation

The main goal of software emulation (sw_emu) is to ensure functional correctness of the host program and kernels. Software emulation provides a purely functional execution, without any modeling of timing delays, or latency; it does not give any indication of the accelerator performance. The kernel code is always compiled and running natively.

The application code is either:

- Compiled and running natively on an x86 processor (**Data Center platforms**)

- Cross-compiler to the Arm processor and running in an emulator (**Embedded platforms**)

Thus, software emulation is typically used for algorithm refinement, debugging functional issues, letting developers iterate quickly through the code to make improvements or bugfix.

The v++ compiler does the minimum transformation of the kernel code to create the FPGA binary to run the host program and kernel code together. Software emulation takes the C-based kernel code and compiles it with GCC. It runs each kernel as a separate C-thread. If there are multiple compute units of a single kernel, each CU is run as a separate thread.

Therefore, it mimics the parallel execution model of the hardware. However, within each kernel the execution is modeled sequentially although there might be parallelism within a kernel when running on hardware.

The software emulation driver implements the XRT API and acts as a bridge between the user application running XRT and the device process modeling the hardware components.

5.2.2 HW Emulation

Hardware emulation runs an RTL simulation of the programmable logic design, where the PL kernels are integrated with a cycle-approximate model of the hardware platform.

Hardware emulation is especially useful for the following tasks:

- Checking the functional correctness of the RTL code synthesized from the C, C++ kernel code
- Testing the interactions between different kernels or multiple CUs
- Using hardware waveforms to gain detailed visibility into internal activity of the kernels
- Getting initial performance estimates for the application

During hardware emulation, kernels are run in the Vivado logic simulator, with a waveform viewer to examine the kernel design. Some third-party simulators are also supported as described in Simulator Support. In addition, hardware emulation provides performance and resource estimates for the hardware implementation.

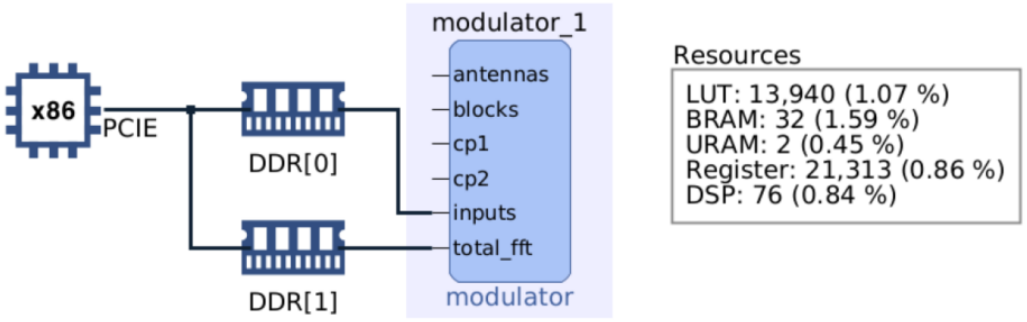

Figure 5.3: System Diagram (modulator)

This is comparable to the synthesis and cosimulation steps seen on Vitis HLS. Also the reports gives us similar results in terms of performances and resources. The main difference is in the timing report, where the software modify the target clock period to the value related to the main clock frequencies of the board ($300MHz- > 3.33ns$). In this case the estimated period is $2.433ns$ with an uncertainly of $0.90ns$.

SystemC models are provided for the key IP used in the hardware platform (.xpfm files) to improve simulation performance and results.

In hardware emulation, compile and execution times are longer than software emulation, but it provides a detailed, cycle-accurate, view of kernel activity. Xilinx recommends using small data sets for validation during hardware emulation to keep runtimes manageable.

5.2.3 Hardware implementation

When the build target is the hardware, v++ builds the FPGA binary for the Xilinx device by running Vivado synthesis and implementation on the design. It is normal for this build target to take a longer period of time than generating either the software or hardware emulation targets in the Vitis IDE.

Summary

Resource	Utilization	Available	Utilization %
LUT	190573	1302720	14.63
LUTRAM	28177	600480	4.69
FF	293654	2607360	11.26
BRAM	359	2016	17.81
URAM	2	960	0.21
DSP	86	9024	0.95
IO	297	297	100.00
GT	16	24	66.67
BUFG	43	1008	4.27
MMCM	5	12	41.67
PLL	10	24	41.67

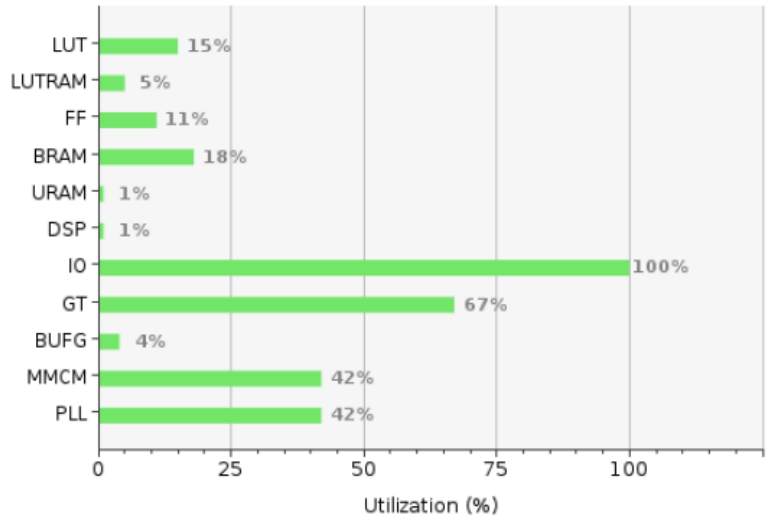

Figure 5.4: Vivado utilization report (modulator)

To get a better view of the RTL design, it is possible to run Vivado synthesis and place and route on the generated RTL design, and review actual results of timing and resource utilization.

In the report above, there is a realistic estimation of the resources usage. Vitis HLS report is out-of-context. It implement only the generated RTL without any I/O considerations.

Using Vivado instead, were considered the I/O, the AXI bus used to connect to the memory controller and all the logic required by the host (interfaces, PCI express, etc...).

Comparing the values, it is easy to see the differences in the BRAM usage due to the needed logic by the AXI interface but also new elements such as I/O, Transceivers (GT), MMCM and PLL that are now estimated in this more accurate implementation. However, the final FPGA binary can be loaded into the hardware of the accelerator card, or embedded processor platform, and the application can be run in its actual operating environment.

5.3 Demodulator

Referring to the following report, there is no differences between this and the graph seen for the modulator. The percentage are exactly the same except for the BRAM (1% difference) and the LUT utilization is quite less as estimated during the previous steps on Vitis HLS.

Summary

Resource	Utilization	Available	Utilization %
LUT	189060	1302720	14.51
LUTRAM	28250	600480	4.70
FF	291408	2607360	11.18
BRAM	352.50	2016	17.49
URAM	2	960	0.21
DSP	86	9024	0.95
IO	297	297	100.00
GT	16	24	66.67
BUFG	43	1008	4.27
MMCM	5	12	41.67
PLL	10	24	41.67

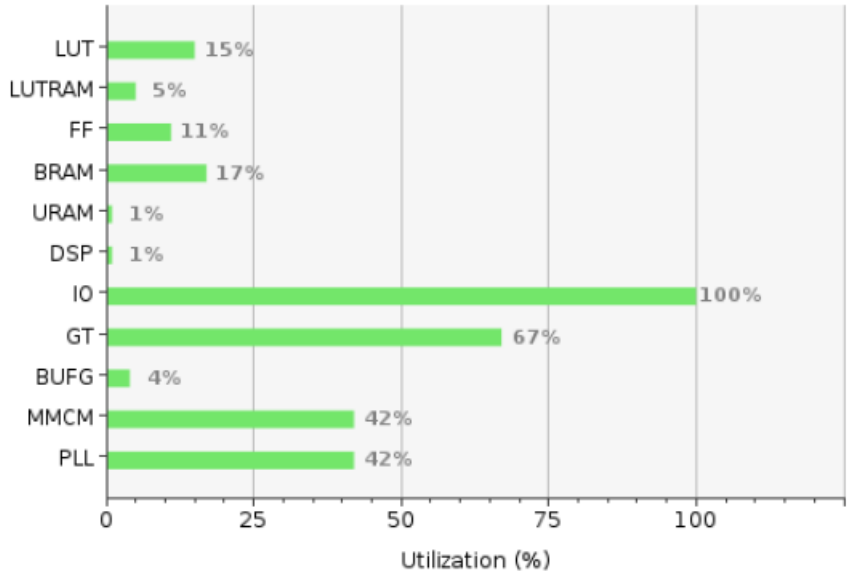

Figure 5.5: Vivado utilization report (demodulator)

5.4 SSRModulator

The results for the SSRmodulator confirm the huge amount of resources usage respects to the same component with a different IP. In particular, as discussed with Vitis HLS reports, this IP use 25% more LUT, 20% more FF and the 40% more DSP than in modulator, where the I/O, GT, MMCM and PLL usage is the same because they share the same interface at top-level.

Summary

Resource	Utilization	Available	Utilization %
LUT	525771	1302720	40.36
LUTRAM	57752	600480	9.62
FF	798108	2607360	30.61
BRAM	407.50	2016	20.21
URAM	2	960	0.21
DSP	3790	9024	42.00
IO	297	297	100.00
GT	16	24	66.67
BUFG	49	1008	4.86
MMCM	5	12	41.67
PLL	10	24	41.67

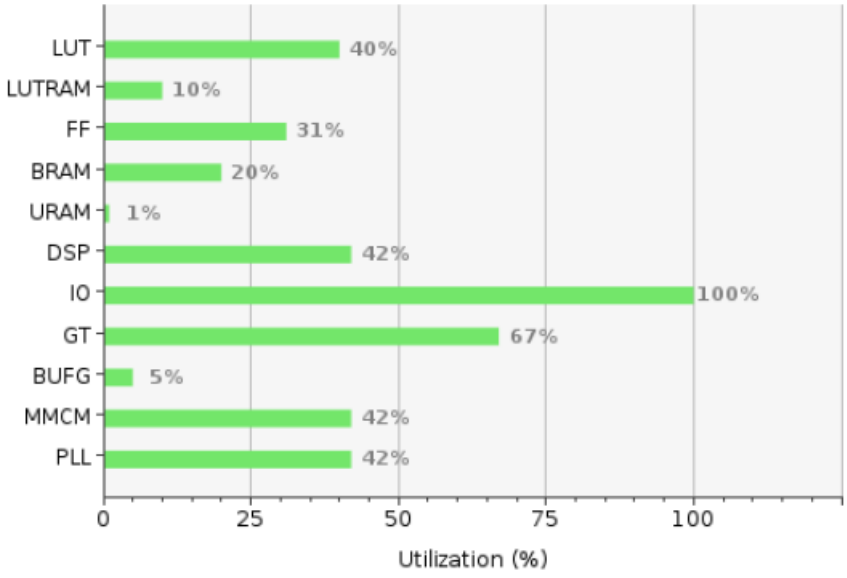

Figure 5.6: Vivado utilization report (SSRmodulator)

5.5 SSRDemodulator

In the same way, the two SSR implementation use the same amount of resources, where the SSRdemodulator usage is 1% less than the SSRmodulator in the FF, LUTRAM and DSP, as shown in the next figure.

Summary

Resource	Utilization	Available	Utilization %
LUT	521281	1302720	40.01
LUTRAM	56965	600480	9.49
FF	789840	2607360	30.29
BRAM	404.50	2016	20.06
URAM	2	960	0.21
DSP	3688	9024	40.87
IO	297	297	100.00
GT	16	24	66.67
BUFG	49	1008	4.86
MMCM	5	12	41.67
PLL	10	24	41.67

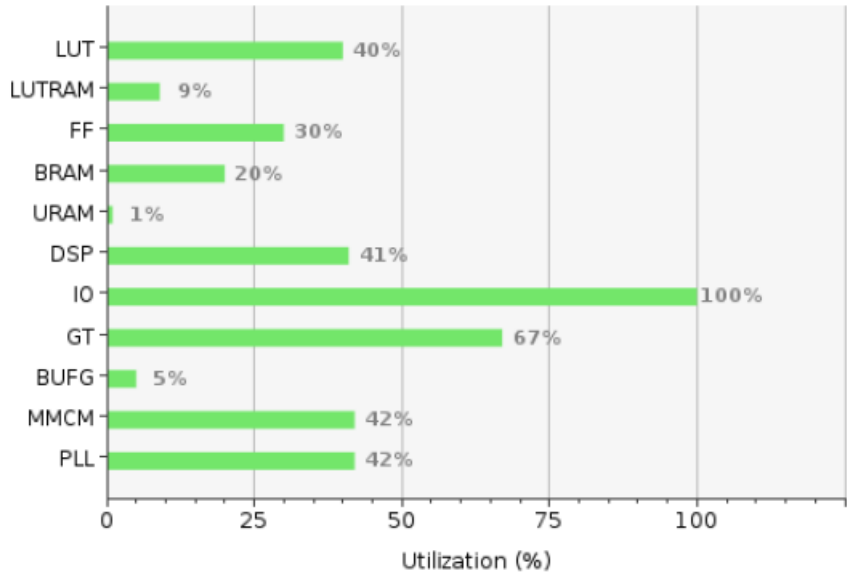

Figure 5.7: Vivado utilization report (SSRdemodulator)

Chapter 6

Next steps

6.1 Conclusion

The channel model is by far the most computing intensive part of the link level simulations of Multiple-Input and Multiple-Output 5G New Radio communication systems, the bottleneck of the whole acceleration project.

The achieved throughput is consistent with the result obtained for the link implementation, so further optimization in terms of performance wasn't explored during this thesis even if possible.

Module	#Inputs	#Outputs	Latency(cycles)	Throughput($\frac{cycles}{samples}$)
modulator	1835008	1966080	5572993	2.835
demodulator	1966080	1835008	5570785	3.036
SSRmodulator	1835008	1966080	5569559	2.833
SSRdemodulator	1966080	1835008	5570689	3.036

Table 6.1: Calculated throughput for different modules

In terms of resources, the difference between the IPs implementations bring us to use the first one and the kernels developed using it. To confirm that data is possible to run the applications on the target accelerator card. When running the design you can specify a number of trace options to capture design data during runtime to take real values to be compared with the estimated one and confirm performance results.

During this step, the host program, written in C/C++ and using the XRT API, is compiled into an executable that runs on an x86 based host processor while hardware-accelerated kernels are compiled into an executable device binary (.xclbin) that runs within the programmable logic (PL) region of an Xilinx device on the Alveo accelerator card.

6.2 Further Optimizations

Talking about the possible optimization and future steps related to this thesis, there are several ways that can be taken.

It was taken into account the possibility to have a unique component to perform modulation/demodulation based of an host parameter. In this work was preferred to work with two different kernels to be able as needed to optimize the two in different way if the performance obtained had been very different. A different implementation is the one provided by Intel, where in the same design it's possible to see OFDM modulator and demodulator [14].

It is possible to evaluate the boost in performance also using a different type of hardware to accelerate the algorithm, as Graphic Compute Unit (GPU). It allows a very high level of parallelism that can be exploited to significantly speed up the computation as in FPGA.

GPUs address a major drawback of CPUs – the ability to process a large amount of data in parallel and can operate on very wide data sets. Fundamentally, GPUs are CPU-like because they have fixed hardware and operate using software instructions. A single instruction could process a thousand pieces of data or more, making them suitable for specific domains such as graphics acceleration, high performance computing, video processing, certain forms of machine learning, and more.

If you're used to programming FPGAs, the process of writing GPU code will feel very similar, even if the outcome is a little different. Programming an FPGA consists of writing code, translating that program into a lower-level language as needed, and converting that program into a binary file. Then, the program is feed to the FPGA just like for a GPU reading a piece of software written in C++.

It was studied a CUDA implementation of the channel model that can be extended to all the link, so implementing also the discussed blocks in CUDA with all the other blocks to compare the boost in performance respects to the FPGA implementation would be interesting.

Bibliography

[1] *5G; NR; NR and NG-RAN Overall description(3GPP TS 38.300 version 17.0.0 Release 17)*. URL: https://www.etsi.org/deliver/etsi_ts/138300_138399/138300/17.00.00_60/ts_138300v170000p.pdf (cit. on p. 2).

[2] *5G; NR; Physical layer; General description (3GPP TS 38.201 version 17.0.0 Release 17)*. URL: https://www.etsi.org/deliver/etsi_ts/138200_138299/138201/17.00.00_60/ts_138201v170000p.pdf (cit. on p. 4).

[3] *5G; NR; Services provided by the physical layer (3GPP TS 38.202 version 17.3.0 Release 17)*. URL: https://www.etsi.org/deliver/etsi_ts/138200_138299/138202/17.03.00_60/ts_138202v170300p.pdf (cit. on p. 4).

[4] *What is OFDM?* URL: https://www.mathworks.com/discovery/ofdm.html (cit. on p. 7).

[5] *Alveo U280 Data Center Accelerator Card*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html (cit. on p. 10).

[6] *Programming an FPGA: An Introduction to How It Works*. URL: https://www.xilinx.com/products/silicon-devices/resources/programming-an-fpga-an-introduction-to-how-it-works.html (cit. on p. 12).

[7] *UltraScale Architecture and Product Data Sheet: Overview*. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds890-ultrascale-overview.pdf (cit. on p. 12).

[8] *Vitis High-Level Synthesis User Guide*. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls (cit. on p. 18).

[9] *Fast Fourier Transform v9.1 LogiCORE IP Product Guide*. URL: https://docs.xilinx.com/r/en-US/pg109-xfft/Fast-Fourier-Transform-v9.1-LogiCORE-IP-Product-Guide (cit. on p. 24).

[10] *Vitis High-Level Synthesis User Guide*. URL: https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls/Using-the-FFT-Function (cit. on p. 24).

[11] *1-Dimensional(Line) SSR FFT L1 FPGA Module*. URL: https://xilinx.github.io/Vitis_Libraries/dsp/2021.2/user_guide/L1.html (cit. on p. 49).

[12] *Vitis_Libraries 1D FFT Tests*. URL: https://github.com/Xilinx/Vitis_Libraries/tree/main/dsp/L1/tests/hw/1dfft (cit. on p. 49).

[13] *Vitis-Getting-Started*. URL: https://xilinx.github.io/Vitis-Tutorials/2021-2/build/html/docs/Getting_Started/Vitis-Getting-Started.html (cit. on p. 59).

[14] *Implementing OFDM Modulation and Demodulation*. URL: https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/vhd-cyclic-prefix-insertion-ofdm.html (cit. on p. 69).

[15] N. A. Shah. «Optimization and Acceleration of 5G Link Layer Simulator». MS Thesis. Torino, Italy: Politecnico di Torino, 2019.

[16] Nasir Ali Shah, Mihai T. Lazarescu, Roberto Quasso, Salvatore Scarpina, and Luciano Lavagno. «FPGA Acceleration of 3GPP Channel Model Emulator for 5G New Radio». In: *IEEE Access* 10 (2022), pp. 119386–119401. DOI: 10.1109/ACCESS.2022.3221124.