POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Model-Based Design and FPGA Implementation of a Pulse Detection Algorithm for Optical and RF Interrogations in Avionic application

Supervisor

Prof. Mario Roberto CASU

Candidate

Giuseppe RINALDO

Co-Supervisor

Ing. Marco AVITABILE

Ing. Riccardo STICCA

October 2025

Abstract

Modern airborne electronic computer implement pulse detection algorithms, that are fundamental for interpreting signals from ground station interrogations and laser-optical warning systems. These systems play a crucial role in aircraft's situational awareness and threat mitigation, in particular in complex operational environments.

These type of algorithms require efficient signal processing capabilities while maintaining high detection accuracy, in order to be capable of identifying fast and transient pulse signals while minimizing false detections. At the same time, these algorithms must maintain low latency and minimal hardware overhead, in order to ensure optimal real-time performance that are typical of modern avionics systems.

To address this challenge, the use of High-Level Synthesis (HLS) [1] tool within a model-based design framework provide a promising approach for implementing signal processing algorithms on FPGA platforms. This methodology enables a more abstract and modular design process, allowing rapid prototyping, early verification and effective design-space exploration. Moreover, it simplifies the development of optimized hardware architectures and resource utilization, computational efficiency and energy performance.

This thesis focuses on the design and implementation of a pulse detection algorithm targeting FPGA platforms through Siemens EDA Catapult HLS tool. This work includes the development, simulation and synthesis of the algorithm, with the evaluation of its hardware characteristics, including logic utilization, latency and power consumption. Furthermore, the study explores different architectural configurations and processing strategies to assess the trade-offs between performance and hardware resource overhead, offering insights into the optimal design choices for real implementation in aerospace applications.

Ai miei nonni Giuseppe e Concetta,
i quali mi hanno permesso di essere
l'uomo che sono oggi

Table of Contents

List of Figures

List of Tables

Acronyms

HLS	High-Level Synthesis.
FPGA	Field-Programmable Gate Array.
MUX	Multiplexer.
DSP	Digital Signal Processor.
VHDL	VHSIC Hardware Description Language.
HLL	High-Level Language.
RTL	Register-Transfer Level.
SoC	System-on-Chip.
ADC	Analog-To-Digital Converter.
PSU	Power Supply Unit.
SNR	Signal-To-Noise Ratio.
FIR	Finite Impulse Response.
RMS	Root-Mean-Square.
LUT	Look-Up Table.
S&H	Sample-And-Hold.
LSB	Least-Significant Bit.

IIR	Infinite Impulse Response.
WNS	Worst Negative Slack.
WHS	Worst Hold Slack.
WPWS	Worst Pulse Width Slack.
BRAM	Block Random Array Memory.

Chapter 1

Introduction

1.1 Context

In the last years, ground station interrogation and laser-optical warning systems have been evolving, trying to obtain faster, more autonomous, and more secure airborne platforms. This phenomenon has led to a stronger focus on high-performance signal processing algorithms and architectures. In particular, the detection and analysis of impulsive signals (such as radar pulses or navigation aids) is a critical task in many on-board systems, such as military helicopters, where low-latency, real-time performance and hardware efficiency are very important in order to instantly perceive and react to the environment obstacles.

Impulsive signals in these contexts have typically low amplitude and they are characterized by a short duration in time, in the order of nanoseconds, and embedded in noisy environments. For this reason their identification requires filtering, integration, thresholding, and sometimes time-frequency or statistical processing with high precision, in order to recognize the obstacle and recalculate a safer flight path. Traditional software-based approaches may have some problems to meet the rigorous real-time constraints imposed by airborne environments, in particular when multiple channels inputs are involved.

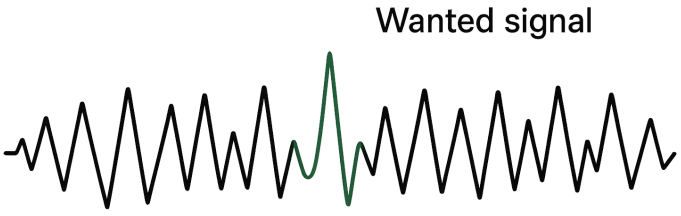

Figure 1.1: Target Pulse Identification within High-Noise Signals

As a result to meet these challenges, programmable logic devices, in particular Field-Programmable Gate Array (FPGA [2]), are increasingly adopted due to their ability to parallelize different operations, minimize latency, and maintain real-time deterministic behavior. However, implementing this kind of algorithms on FPGA is not only such an easy task: it involves not only theoretical development and numerical validation, but also architectural consciousness of how algorithms are translated into logic blocks like MUXs (Multiplexers [3]), DSPs (Digital Signal Processors [4]), flip-flops and memory structures.

Until now many functions have been managed in an analog way but, due to the high number of operations that must be integrated in the system, a digital way is now mandatory to increase efficiency, reliability and integration. Nowadays digital design generally follows a traditional methodology based on manual HDL coding. In this approach, algorithms have been translated into low level hardware description languages such as VHSIC Hardware Description Language (VHDL [5]) or Verilog, that could be a high-time consuming process due to the increasing complexity of these algorithms.
In some cases tools like Simulink HDL Coder have been employed to partially automate and speed-up this process: algorithms are described by using High-Level Language (HLL [6]), but they still require some modeling specifics that must be compatible with the hardware synthesis. For this reason HLS is arising as an efficient alternative methodology that allows to obtain a Register-Transfer Level (RTL [7]) description starting from the behavioral description of the algorithm. HLS gives the possibility to the designer to describe the algorithm in HLL such as C++ or System C, and then automatically generate RTL reducing the effort required for the design and the verification.

1.2 High-Level Synthesis Definition and History

High-Level Synthesis (HLS), also known as behavioral synthesis or algorithmic synthesis, refers to the ability to generate high quality and synthesizable RTL implementations starting from an abstract. Unlike traditional design methods, HLS allows designers to realize the functionalities using HLL, like C++ or SystemC, from which the synthesis tool generates an optimized hardware architecture that meets specific design constraints such as latency, throughput, power, and area consumption.

Nowadays, most of the designs start with the specification and with an executable model that describes the behavior of the system, without any information regarding hardware details. After that, the model must be optimized and a suitable architecture must be chosen in order to realize a physical implementation of the described algorithm. But this challenge is not so easy and the problem is amplified in larger designs, due to the fact that it is more complex to realize the system without any error. This is precisely where HLS proves valuable: it ensures an error-free design flow from high-level

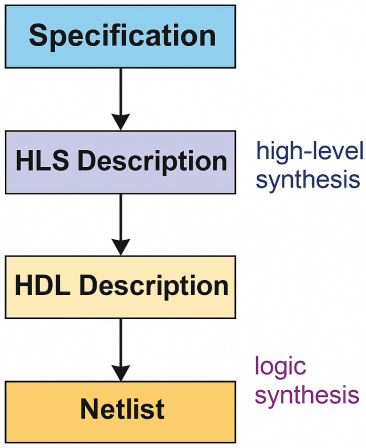

Figure 1.2: Design Flow

specifications, significantly reducing development time and effort.
By decoupling behavior from structure, HLS tools can explore different architectural implementations in an automatic way through design space exploration, enabling rapid prototyping, shorter verification and higher productivity. The process of HLS involves parsing a behavioral input specification, scheduling operations over clock cycles, allocating and binding operations to specific hardware resources and finally generating an RTL description that meets timing and resource constraints defined by the designer.
The shift of the focus of the engineer, so from the low-level control to the functionality of the algorithm, is well-represented within the Gajski-Kuhn Y-chart (Figure 1.3) made by Robert Walker and Donald Thomas in 1985 [1].

This graph models hardware design as three orthogonal views (behavioral, structural, and physical) each with increasing levels of refinement. HLS operates within the behavioral domain at higher abstraction levels, providing a structured pathway to transition into the structural domain through automatic generation of RTL. The refinement process envisioned by the Y-chart, starting from an abstract behavioral model and gradually introducing implementation details, is precisely the workflow enabled by HLS tools. The output of HLS can then undergo logic synthesis and physical design, completing the traversal of the Y-chart toward the transistor-level realization of the system.

Historically, HLS was tied to research studies in the ’70s and early ’80s, when the higher abstraction level was already called for by researchers and industries as an early response to increasing system complexity. The first pioneering research studies explored the mapping of behavioral specifications onto register-level hardware through structured transformations and scheduling algorithms. Experts on HLS history have recognized different generations of HLS tool [2]: the first genuine generation of HLS, between the 1980s and early 1990s, was still dominated by academia. Pioneering research by researchers like Dan Gajski, Giovanni De Micheli, and Pierre Paulin introduced seminal notions e.g.,

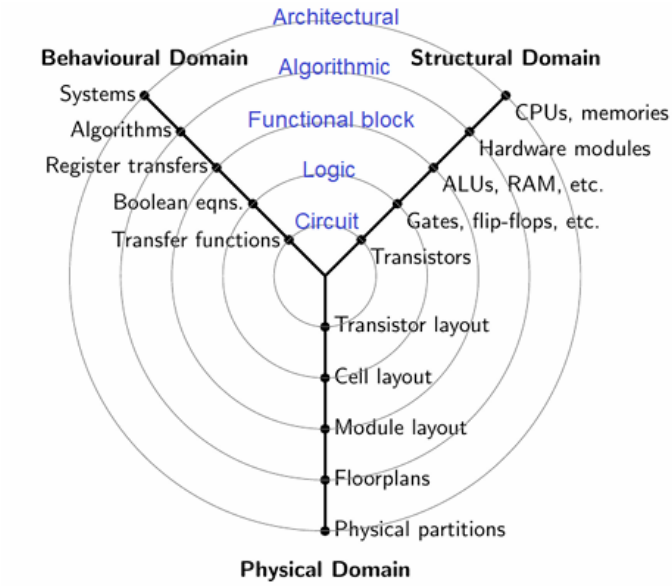

Figure 1.3: Gajski-Kuhn Y-chart

force-directed scheduling and architecture exploration. Although they produced many classic papers and initial tools, typically DSP domains, this generation did not achieve any industrial impact. There were many reasons: the tools took non-standard, application-specific input languages; the tools produced hardware results with inconsistent quality; and they were too targeted for dataflow problems, which were not on the agenda at that instant of ASIC designers, being focused on control logic and system integration. The second generation, the mid-1990s and early 2000s, is where the first serious commercial initiatives for HLS were made by big EDA firms Synopsys, Cadence, and Mentor Graphics. Some of the tools created with the goal of providing behavioral synthesis relevant for actual designs are Synopsys's Behavioral Compiler, Cadence's Visual Architect, and Mentor's Monet tool. The second generation commercially failed. These tools had the hypothesis that RTL designers would automatically migrate to HLS, which never happened. Further, inputs as behavioral HDLs restricted usage to a small number of designers. The tools also required synthesis right down to gate level without being part of the then-prevailing RTL synthesis flows at any point, rendering the workflow inefficient. The designs thus produced tough to verify, performed erratically, and were less than ideal for control-intensive logic. These, with marketing hysteria and user unfamiliarity, caused second-generation HLS tools to fail. Third generation of HLS, starting about the early 2000s, however, was far more successful and prevalent. Among the differences in features of this generation is the utilization of high-level programming languages with which computer and algorithmic C, C++, SystemC, and MATLAB. Catapult C from Mentor Graphics, Forte's Cynthesizer, Bluespec, and Cadence's C-to-Silicon compiler for application domains like DSP, multimedia, and image processing, which are dataflow-based and algorithmic in nature, belong to this category. These tools bypassed the flaws of

their forebears by targeting algorithm designers rather, enabling faster exploration of design alternatives, and yielding better quality of outcomes. Increased uses of FPGAs in hardware prototyping also played a big role, since HLS tools were superior in rapidly translating algorithms into FPGA fabrics, where fitting and timing are less important than in ASIC design. Therefore, HLS was a viable choice for specific classes of designs, especially in wireless communications, multimedia, and signal processing.

Looking ahead, there is an anticipated fourth generation of HLS. This new generation would combine the virtues of present tools by allowing synthesis across multiple domains, control- and datapath-intensive designs, and even integration with heterogeneous targets such as ASICs, FPGAs, and configurable processors. The ultimate vision for this generation is truly system-level design exploration and optimization—software that allows designers to compare trade-offs among architectures, domains, and abstraction levels effortlessly and automatically. While this vision is still a fantasy, the progress already made provides a solid foundation for making the long-awaited promise of high-level synthesis a reality: to significantly improve design productivity while retaining or improving implementation efficiency.

HLS has become a revolutionary methodology, specifically in situations where rapid prototyping, design reuse, and system-level optimization are critical. It offers a tractable path toward dealing with the growing complexity of today's hardware systems at respectable levels of design effort, correctness, and performance. As the digital design environment keeps on evolving, HLS will find itself at the center of system-on-chip (SoC [8]) and FPGA-based design approaches more and more.

1.3 Objectives

This work addresses the design, the simulation, and hardware implementation of a pulse detection algorithm for the recognition of impulsive signals. Two different hardware implementation flows are explored in parallel in order to compare their efficiency and effectiveness for avionic applications:

- **Simulink-based flow**: starting from a high-level Simulink-based model, it is possible to generate in an automatic way the HDL code thanks to Simulink HDL Coder tool. This flow has been used for many years and it exploits the model-based design principles to allow the transition from the high-level algorithm to the hardware implementation, for an easier prototyping and verification;

- **HLS flow**: this approach exploits Siemens EDA Catapult HLS tool, which is able to generate the hardware implementation starting from the C++ description of the algorithm. This strategy allows more software-centric design methodology, improving flexibility and control on hardware optimizations in terms of resources and performances.

The main objective of the thesis is to evaluate and compare these two design flows in terms of:

- **Design effort:** It includes the time needed to develop, debug, and optimize the algorithm implementation for each flow. It covers the time needed to learn how to use the tools, the development of the algorithm and its translation into hardware, its verification and its optimization in order to achieve the wanted results;
- **Hardware metrics:** Analysis of key hardware parameters that are resource utilization (how many logic blocks are used), latency (the time delay between the first input and the first valid output generated), working clock frequency and power consumption (so how much static and dynamic power the hardware consumes).

1.4 Methodology and Thesis Organization

In order to achieve the objectives described in the previous section, the work has been organized according to a structured and sequential methodology, that can be observed in Figure 1.4. This kind of approach has allowed a progressive consolidation of knowledge, facilitated early-stage validation and supporting a mapping between high-level description and low-level implementations. The key activities that have been planned are summarized below, with the corresponding chapter in which they are discussed in detail:

- **Environment setup and tools familiarization:** this initial phase involved the study and the familiarization with the new tools used throughout the thesis (Simulink HDL Coder and Catapult HLS) (Chapter 4 and 5)
- **Implementation in MATLAB:** during this phase, the algorithm has been described starting from the real data acquired by the framework and the parameters and the setup for the Simulink model of the pulse detection algorithm were developed and tested using MATLAB scripts, enabling parameter tuning and mathematical correctness (Chapter 2)

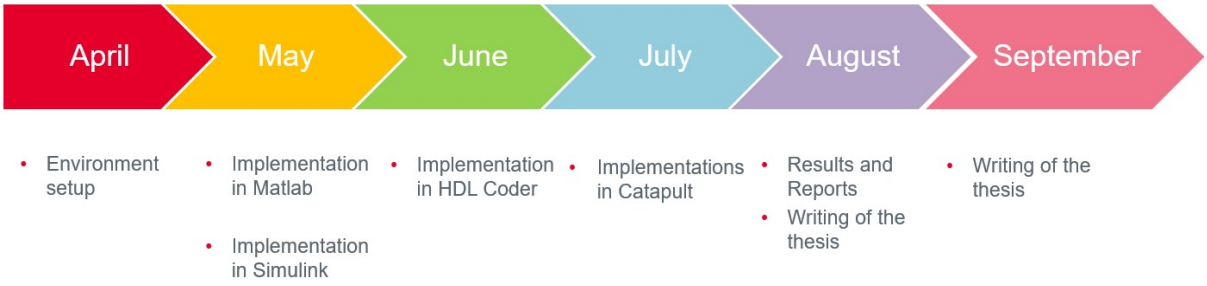

Figure 1.4: Tasks Timeline

- **Implementation in Simulink:** a modular Simulink model of the framework representing the arrival of the pulses and their elaboration has been developed, preparing it for HDL generation and system-level simulation (Chapter 3)
- **Implementation in HDL Coder:** the relevant part was converted into HDL code using Simulink HDL Coder. The generated code was then optimized, synthesized and evaluated within the Vivado design suite, targeting FPGA-based deployment (Chapter 4)
- **Implementation in Catapult:** the algorithm and the design blocks were described by using C++ code and then thanks to this tool it was possible to obtain the RTL and the synthesizable code for Vivado (Chapter 5)
- **Results and Reports:** a final comparison between the two flows has been performed in order to see the differences obtained in terms of hardware implementation and time effort (Chapter 6)

Chapter 2

Implementation in MATLAB

2.1 Algorithm description

The description of a pulse detection algorithm with all its specifications has been the first key task of this work. The system operates by emitting short-time duration laser pulses in order to scan the surrounding environment and then analyzing the returned echoes to be able to identify a possible dangerous obstacle and to reconstruct the image of it. For this reason the pulse detection must be very accurate to ensure the safety of flight operations. The detection of the signal that is reflected by the target object is initially performed by an analog chain that is composed by several stages. First of all the optical signal is converterd into an electronic one via an avalanche photodiode, that is know in the optoelectronics world for its high sensitivity and fast response. After that, the second step involves a signal pre-amplification in order to obtain a suitable amplitude and finally a threshold comparison, which is a function of the round-trip-delay of the transmitted pulse.

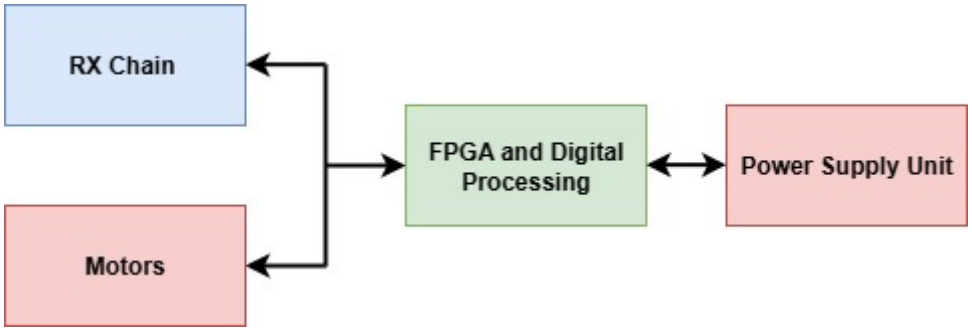

Figure 2.1: Block Diagram of the system

Once that the analog signal is acquired in the proper way, it is digitalized by an analog-to-digital converter (ADC [9]) and the resulting samples are sent to a FPGA which performs further processing. If the comparison is successful, the main information related to the detected echo is estimated: range (and so the distance from the sensor), angular coordinates (azimuth and elevation) with respect to the local reference frame and some

qualitative attributes that are related to the nature of the reflecting surface. The range is evaluated based on the time delay between the transmission and the reception of the pulse, while the angular coordinates are determined by the angular orientation of the scanner at the acquisition time. The geometry of the is evaluated from the amplitude of the returned echo.
In the overall chain of processing of the opto-electronic signal outlined above, some several disturbances occur in the analog acquisition chain: these include electromagnetic interferences generated by the motors that drive the laser beam deflection in the predetermined area and the switching noise due to the power supply unit (PSU [10]) that provides all the necessary power to allow to the device to operate properly. Although the shielding adopted to mitigate these disturbances, they are not completely eliminated and this affects the signal-to-noise ratio (SNR [11]) of the system. Moreover, the sensitivity of the device is also affected, that is the ability to detect the low-amplitude echoes that are typically related to this obstacles (for example power lines, cables) or those that are farther away from the framework (search range).

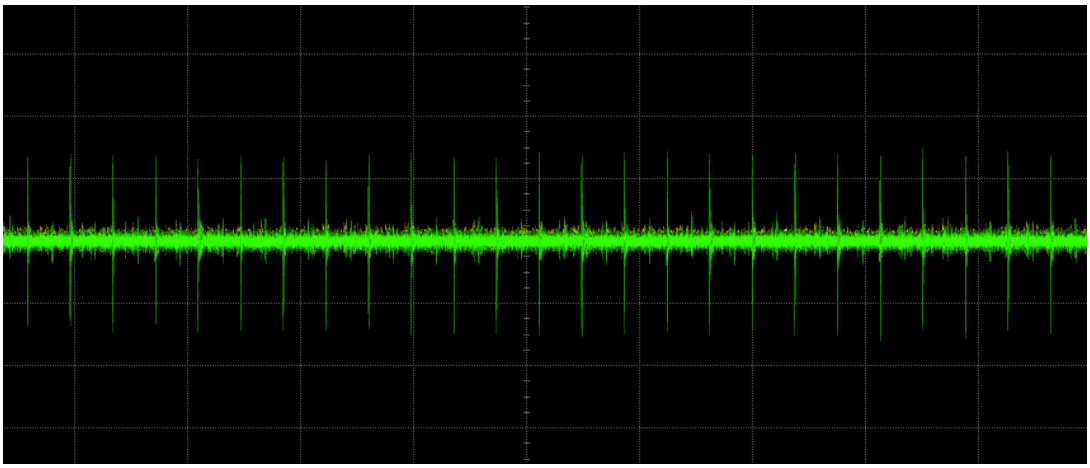

Figure 2.2: Measured Interferences

First of all, a recording of the signal without firing the laser pulse (but with the scanner in action) has been used to carry out the model in order to obtain a more realistic solution (Figure 2.2). The use of differential signaling provides only partial mitigation of the electromagnetic disturbances that affect the analog signal chain. While this approach does not reduce the amplitude of the noise, it introduces symmetry because the common mode noise appears equally on both lines and are canceled out when computing the difference between A and B. The result is an effective signal swing of about 2A (assuming A and B are perfectly complementary), which remains stable over the time, even if there are external interferences.

However, the system output shows that disturbances are still present and superimposed on the background noise of the system. The amplitude of these residual disturbances is typically in the order of 30 mV peak-to-peak. This is a critical figure, as it directly

affects the reliability of the threshold detection mechanism. In fact, for any detection threshold set at or below this level, the system becomes highly susceptible to generating false positives, i.e. spurious detections not associated with actual reflected echoes but rather with transient noise peaks or oscillatory artifacts introduced by the interference.

Moreover, a recording of the pulse received after the shot, with the scanner inactive, is shown in Figure 2.3.

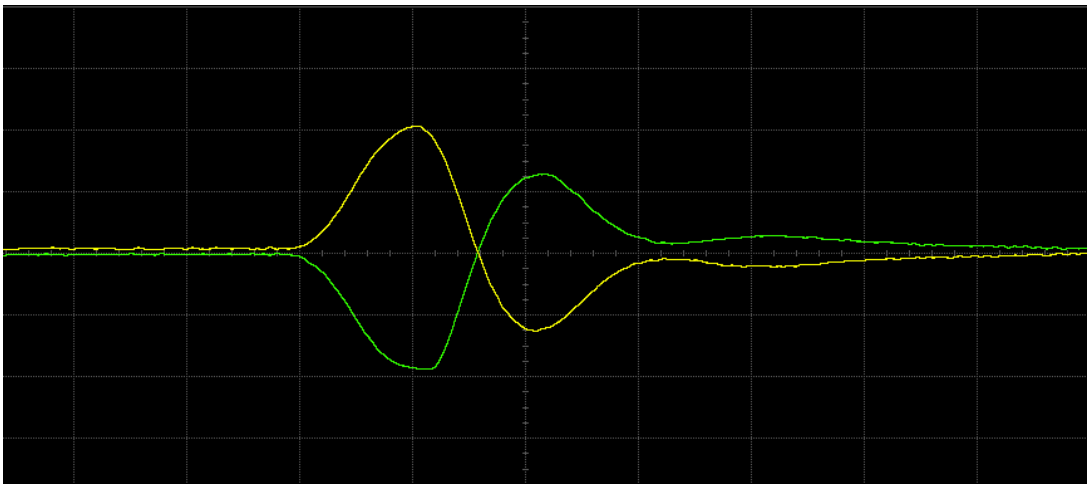

Figure 2.3: Measured Pulse

To address this challenge a robust signal conditioning strategy is required. One possible solution to the problem could be the insertion of an appropriate filtering stage before the threshold comparison, which could further reduce or eliminate such disturbances. This solution implies two possible implementation alternatives: the first one is to insert an analog filter at the pre-amplification stage, while the second alternative is to perform the filtering directly in the digital domain downstream of the ADC. The latter approach also implies the migration of both the filtering and threshold comparison operations into the digital domain, within the FPGA itself.

This first part of the work explores a possible implementation of the second approach. The proposed digital signal processing involves a finite impulse response (FIR [12]) filter to eliminate noise and enhance valid echoes signatures. After that, there is a comparison between the signal and the threshold in order to recognize the returned echoes from some false peaks due to noise. This architecture allows to improve the detection of weak pulses and provides great flexibility and reconfigurability.

2.2 Peaks and Noise Characterization

The simulator developed to evaluate the behavior of the whole signal acquisition and processing chain, from emission of the laser pulses to digital threshold detection performed

within the FPGA, was implemented in the MATLAB/Simulink environment. The simulation environment enables the architectural design and functional validity of the processing pipeline to be verified early for various operating conditions. The simulator is divided into two components: a couple of MATLAB scripts and an overall Simulink model of the signal processing chain.

The MATLAB scripts manage the definition and establishment of the simulation parameters. In particular, the scripts allow to compute the input vectors and reference values necessary for simulation initialization and generate the look-up tables that guide threshold decisions in the digital logic. Before Simulink simulation is executed, the MATLAB scripts must be executed in order to initialize all data structures and configuration parameters appropriately.

The first MATLAB script (Calcolo_Vrms_noise_e_segnale_ver01.m, see Appendix A) is dedicated to the implementation of a detailed radiometric model, which plays a fundamental role in characterizing the expected behavior of the received signal under varying environmental and system conditions.

```
q = 1.602176634e-19;              % carica elettrone
detector_BW = 50e6;               % [Hz] detector electrical bandwidth
M_adp_gain = 9.244;               % 9.168; ADP gain, from interpolated curve
Rapd = 1.087;                     % 1.09; [A/W] detector responsivity
keff = 0.45;
Tz = 34e3;                        % [Ohm] detector resistance
Gain = 3.5;                       % preAmp voltage Gain
eta_rx_bw = 0.4723;               % 0.45972; % Optical bandwith vs Receiver bandwith
                                  % Band penalty due to the filter effect of the receiver
Pb_det_in = 2.724e-9;           % [W] con sfondo 50 W/(m2 sr um) Potenza di backgroud a valle del filtro IF
Pb = Pb_det_in;                   % Potenza di backgroud
Pb_ase = 2.942e-10;               % 0.736e-9; % [W] Received power into RX/TX path due to ase
Ids = 3.3e-9;                     % 9.702e-12; % [A] Ids := IDS(Is0) = dark due to surface
Idb = 3.3e-9;                     % 3.465e-9; % [A] Idb := IDB(Is0) = dark due to bulk
in_LLAM = 0.6e-12;                % [A] in_LLAM = 0.6 pA / sqrt(Hz);
R = Rapd;                         % responsivity
M = M_adp_gain;                   % ADP gain
Deltaf = detector_BW;             % detector electrical bandwidth
F = keff*M+(1-keff)*(2-1/M);      % Noise figure
```

More specifically, the model uses a number of significant parameters that govern the energy of the back reflection as it can be seen in the code lines above. They include:

- inherent characteristics of the outgoing pulse, like temporal profile and peak intensity;
- characteristics of the reflecting target or obstacle, i.e., its reflectivity, surface geometry, material structure, and relative position to the source;
- the characteristics of the medium of propagation, such as the attenuation effects like absorption by water vapor and particles, solar background noise, and potential optical turbulence.

These atmospheric contributions are particularly significant when measuring degradation of the signal over longer distances or for conditions of low-visibility environments.

2.2.1 Effective signal power on detector

The script includes the factors shown above to compute the resulting amplitude of the received electrical pulse and the associated noise level, typically in terms of root-mean-square (RMS [13]) voltage. Regarding the evaluation of the effective signal power that arrives on the detector, that in the script is denoted as $P_s\_det\_in$, not all this power is effectively transformed into photocurrent in the detector. Losses and bandwidth restrictions of several types reduce the amount of the received signal that can be utilized.

```
1    Ps_det_in(k) = x;                                    % vector containing the
     received signal power
2    Ps = Ps_det_in(k) * eta_rx_bw;                       % [W] Effective power on
     detector
3    In_signal = R * Ps;                                  % [W] Photocurrent
```

As can be observed, the effective power received on the detector is evaluated as:

$$P_s = Ps\_det\_in \cdot \eta_{rx\_bw} \tag{2.1}$$

where the term η_{rx_bw} is a coefficient representing the overall optical efficiency that includes:

- optical coupling losses;
- imperfect responsivity across the signal spectrum;
- filter and bandwidth restrictions.

This effective power is then used to compute the signal photocurrent

$$I_n\_signal = R \cdot P_s \tag{2.2}$$

where R is the avalanche photodiode's respositivity.

2.2.2 Noise modeling in the optical receiver

The optical signal received is perturbed by several sources of noise, each one originating from distinct physical mechanisms. In order to evaluate the system's performance in terms of the SNR, it is essential to model all the most significant noise contributions. The equations used in this work are well-established equations in the photonics literature and represent the main processes responsible for degrading detection capability. The considered noise terms are:

- **Shot noise**: Shot noise is a fundamental type of noise that originates from the random nature of discrete charge transport events occurring independently of one another [3]. Shot noise is encountered especially in electron emission from a thermionic cathode or photocathode, carrier transport in a pn junction or current conduction in semiconductor devices, like transistors. Moreover, it is associated with the energy transitions in semiconductors such as the electron-hole pairs generation and recombination and photon emission processes in lasers. Due to their stochastic nature, the resulting current contains variability. These can be quantified in terms of the RMS value of noise current and are proportional to the mean direct current of the underlying process. The RMS value of shot noise current is given by the following formula:

$$\sqrt{\overline{i_s^2}} = \sqrt{2 \cdot q \cdot I_{dc} \cdot B} \tag{2.3}$$

where q is the electron charge, I_{dc} is the average DC current and B is the electrical bandwidth in Hz. In the case of this work, the equation to evaluate the shot noise due to the signal becomes:

```
insh_signal = sqrt(2 * q * In_signal * M^2 * F * Deltaf);
```

The signal photocurrent, In_signal, is subject to quantum fluctuations due to the fact that photo-detection is a discrete process. For an APD, the photocurrent is further amplified by the avalanche gain M, with the multiplication process introducing statistical fluctuations that are accounted for by the excess noise factor F. Shot noise is a fundamental limit to optical detection. Under weak signal power, the photocurrent generated by the signal is low and the unpredictability of photon arrivals is the dominant noise source.
Moreover, this noise contribution must be evaluated also for the background power and the dark current:

```
insh_dark   = sqrt(2 * q * (Ids + Idb * M^2 * F) * Deltaf);
```

where I_{ds} I_{db} are the dark current contributions due to surface and bulk, respectively. And

```
insh_back   = sqrt(2 * q * (In_background * M^2 * F) * Deltaf);
```

where $I_n\_background$ is the current evaluated from the background power P_b as:

```
In_background = R * Pb;
```

- **Johnson-Nyquist and Flicker**: Johnson-Nyquist noise, also known as thermal noise, is a fundamental physical phenomenon that is due to random thermal motion of electrons within a resistor, as stated in [3]. This phenomenon is applied at the electrical terminals of the passive components, even in the absence of any applied signal or current flow. Thermal noise is present in all resistive components and is unavoidable in electronic circuits. The general equation of thermal noise is the following one:

$$\sqrt{\overline{i_j^2}} = \sqrt{\frac{4 \cdot k \cdot T \cdot B}{R_L}} \tag{2.4}$$

where k is Boltzmann's constant, T is the absolute temperature, R_L is the resistance value in ohm and B is the electrical bandwidth.
Regarding the Flicker noise, also known as 1/f noise, is a low-frequency noise component that becomes significant as the frequency is decreased. It originates mainly from semiconductor material and interface defects, like traps in MOSFET gate oxides or surface roughness [4]. Flicker noise is not white like thermal noise: its power spectral density increases as the frequency decreases, and it typically dominates the total noise spectrum at frequencies below a few kilohertz. The equation is the following one:

$$I_{nf}^2 = \frac{2 \cdot q \cdot I_{dc} \cdot f_c^m \cdot B}{f} \tag{2.5}$$

where f_c^m is the comer frequency, where m is between 1 and 2, and f is the frequency of interest.
Here the two components are summed toghether and the total noise is given by:

```
in_tia = in_LLAM * sqrt(Deltaf);
```

where in in_LLAM is a parameter that considers the two previous noise contributions.

- **Internal background**: This term explains the noise due to the internal background optical power within the system falling on the photodetector. In a real-world optical system, not every incoming light at the detector can be attributed to the desired signal. There is some residual background light present within the receiver itself due to internal reflections, non-ideal optical isolation, spurious coupling among the optical components, or other non-ideal effects of the system design. Although such background light is constant or gradually varying with time, it also generates a photocurrent in the detector. It is evaluated as:

```
insh_stray  = sqrt(2 * q * (R * Pb_ase) * (M^2 * F) * Deltaf);
```

where P_{b_ASE} is the internal background power.

All these contributions must be summed all together in order to evaluate the total noise current (in RMS value):

```
ni_tot = sqrt(insh_signal^2+insh_dark^2+insh_back^2+in_tia^2+insh_stray^2);
```

After the evaluation of the currents, the voltages of both the wanted signal and the noise are evaluated as:

```
Vnoise(k)   = Gain * ni_tot * 1 * Tz;              % voltage total noise
Vsignal(k)  = Gain * (Ps * Rapd) * M * Tz;        % voltage signal level
```

where $Gain$ is the voltage gain and Tz is the detector impedance evaluated in Ohm. Moreover, the SNR, in linear and in dB, of each index is evaluated as the ratio of the signal and noise voltages:

```
SNR_lin(k) = Vsignal(k)/Vnoise(k);
SNR_db(k) = 20*log10(Vsignal(k)/Vnoise(k));
```

The output of the script is a look-up table (LUT [14]), which maps a series of operating points, described by output voltages between 5 mV and 3 V, to their respective pulse amplitudes, noise RMS levels and calculated SNRs. In fact a vector of discrete values of output voltages is created, converted in volts to maintain unit consistency, has been created.
The distribution of these reference values is intentionally more compact at lower voltage amplitudes, which is the region of primary interest for the analysis; this arrangement ensures higher precision in fitting simulated data to the desired levels in the most critical operating region. The number of reference levels was determined ($Vout\_nr$), and for each level, the algorithm computed the absolute difference between the simulated output voltage vector ($Vsignal$) and the reference value. The index of the minimum difference was stored in idx, thereby determining the point in the simulated signal that best matches each given voltage level. This operation effectively maps the continuous simulated signal into discrete output levels for performance evaluation under quantized measurement constraints.

```
Vout_nr = length(Vout);
for i = 1:Vout_nr
    delta = abs(Vsignal - Vout(i));
    minValue = min(delta);
    idx(i) = find(delta == minValue);
end
```

The output of the MATLAB scripts can be observed in Figure 2.4 and 2.5.

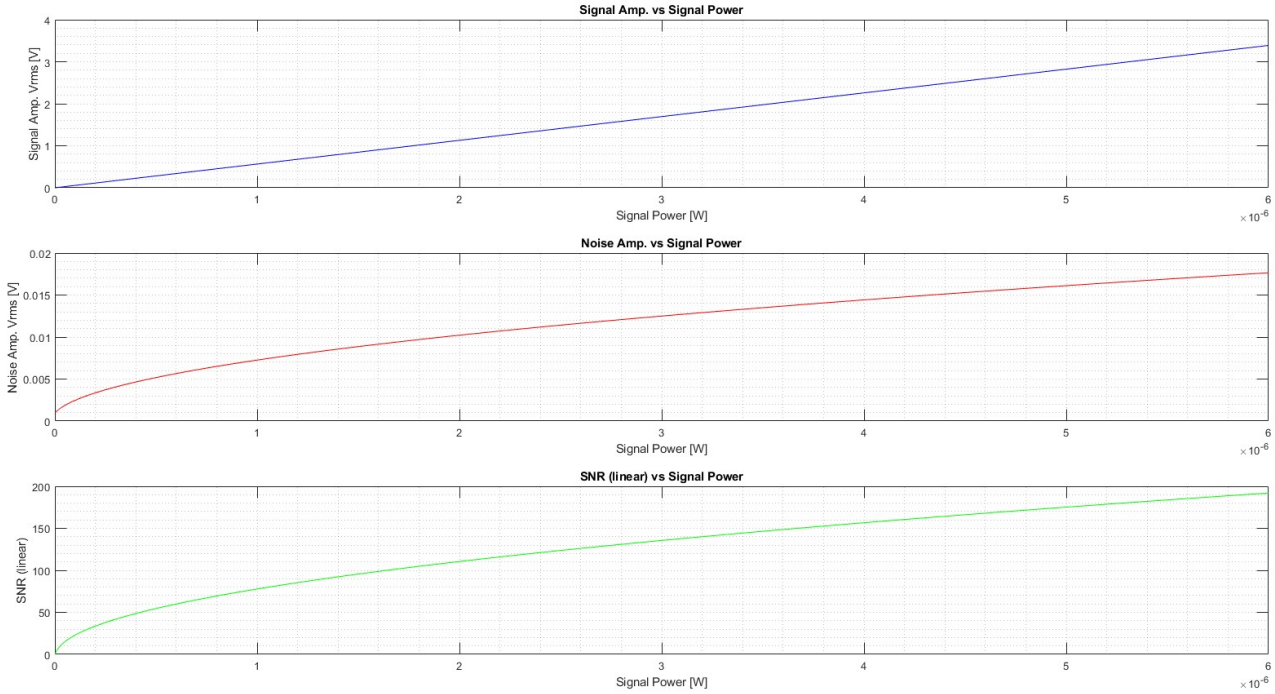

Figure 2.4: Relationship with Signal Power

The previous subplots contains the trends of signal and noise amplitudes and the linear SNR with respect to the incident signal power in the detector $P_S\_det\_in$. As can be expected, the signal amplitude has a linear slope with respect to optical power due to the fact that all the contributions that gives the value of the slope (responsivity, gains and so on) are linear too. Meanwhile, the noise amplitude grows as the square-root with the signal power due to the fact that it embeds all the contributions described before that grows as $\propto \sqrt{I}$. For this reason, the SNR follows the same trend of the noise amplitude, in fact in the low-power regime, SNR improves rapidly with increasing power because the signal grows linearly while the noise is near its floor. For higher power values the SNR continues to increase but with a lower slope due to the noise amplitude ($\propto \sqrt{P}$).

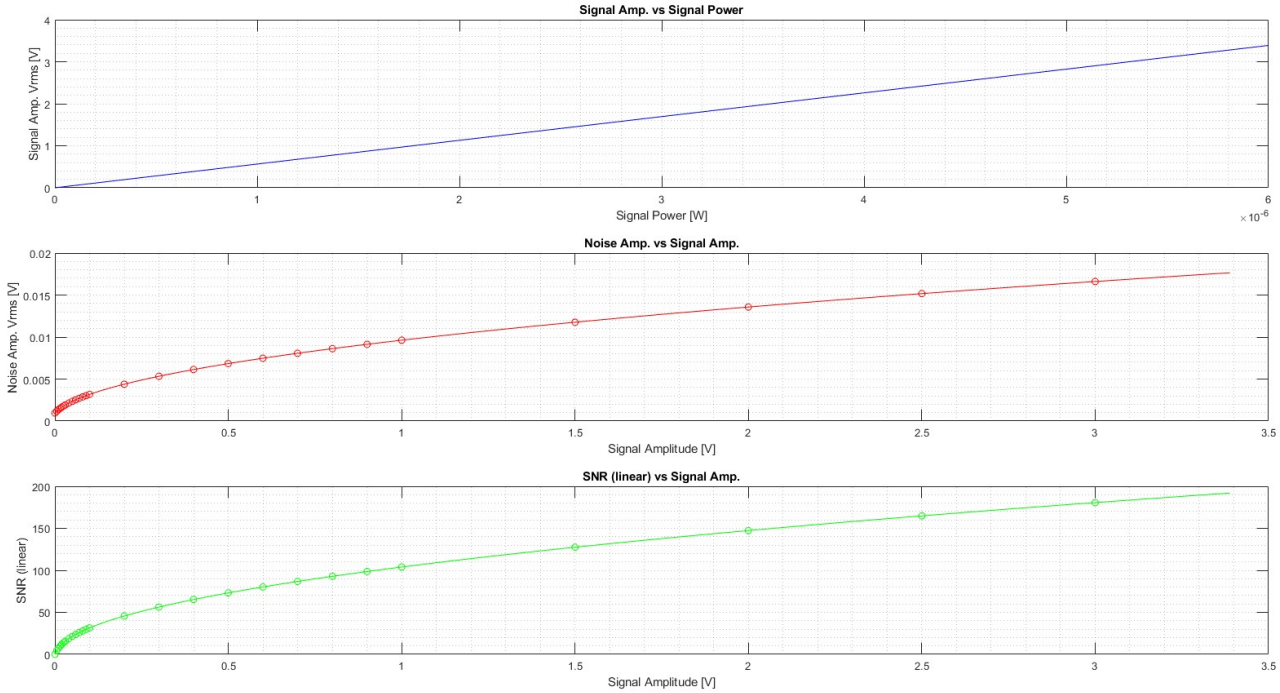

Figure 2.5: Working points

In the second Figure, there is again the same subplot of the first one (Signal amplitude vs. Signal power) and then there are the subplots of more interest that are the noise amplitude and the SNR values evaluated with respect to the signal amplitude.

- The red curve shows how total noise voltage varies with signal level. Red markers are the operating points where $Vsignal$ is closest to each target maximum output level in Vout. This makes it easy to read off the noise level for the desired outputs.
- The last panel directly shows how SNR changes with output signal level. The green markers are the SNR achieved at the discrete output levels Vout. The arrays Vout and idx give a mapping from desired discrete output levels to the closest achievable points in the simulated signal.

The overlaid markers in Figure 2.5 enable to read off noise and SNR at exactly those operating points, which is valuable for:

- checking conformity to instrument/ADC ranges
- selecting operating points with good SNR
- measuring performance accuracy where the Vout grid is finer at low amplitudes (range of interest), thus increasing accuracy of reported metrics in the most critical range.

2.3 Composite Signal Generation

This second script is executed immediately after the first one and represents a critical step in the simulation process: it translates the analytical results of levels of signal and noise previously calculated into a waveform within the real-time domain. Whereas the first script produced static plots of Vsignal, Vnoise and SNR for a large range of operating points, this step is focused on one specific working point and produces a time series reproducing the exact signal-to-noise ratio for this point. In other words, it takes a point on the SNR curve from the original script and "materializes" it into a waveform consisting of pulses and background noise that can be sampled, filtered, and processed as in the real system.
The script begins by ensuring that the simulation could be the reproducible. A seed value for the random number generator is made constant, such that in every execution, the results are identical, in order to allow comparisons among runs with different parameter settings. The general parameters are then defined: sampling frequencies for both the background and the pulse, total simulation time, number of points to be simulated and the time slot available for each pulse. A small bias term is also set for the background in the real-data mode.

```
seed = 091537;
rng_info = rng(seed,'twister');
RandomSeed = floor(rand*10000);

% general parameters
ideal_sig = 0;                      % real or ideal data signal generation
Max_false = 100;                    % maximum false peaks detectable
Fs_bkg = 10e9;                      % background sampling frequency
Fs_imp = 160e9;                     % pulse sampling frequency
Fs = Fs_bkg;                        % output sampling frequency
Pr = 10e-6;                         % time slot to receive each pulse
time_span = 100e-6;                 % total duration of the simulation
Npts = round(time_span*Fs);
Bias_bkgd = 0.03;
dt = 1/Fs;
```

Next, the script selects the input files. The background file contains the waveform of the background noise, the other the pulse waveform, both measured from a real simulation of the framework. In particular, there are different files that can be selected but for this work the files used are:

- for the background noise, a file that contains the data collected for $100\mu s$ with the scanner operating
- for the pulse, a registered echo with a width of $100ns$

For real data, a clean time window that is not disturbed by interferences is chosen in the background file in order to determine the RMS value, which is then used for

normalizing the waveform. This normalization allows that the background can be scaled precisely to the desired noise amplitude.

```
inp_file_bkgd = 'bkgd_scan_width100us_step10us_fs10ghz.mat';
t1 = 13.6e-6; t2 = 17.1e-6;

inp_file_pls  = 'pulse_width100ns_step10ns_fs160ghz.mat';
```

A key step follows: choosing the operating point. The script prompts the user to choose an index corresponding to one of the discrete output voltage levels evaluated in the first script. Based on this index, it chooses the signal amplitude, noise amplitude and the corresponding SNR. This provides absolute consistency between the two scripts: the generated waveform is not random, but matches one of the analytically computed scenarios.
The index that can be selected by the user goes from 2 to 27, in the case of this work, because a index equal to 1 corresponds to a signal amplitude that is equal to 0V, and so it can not be useful to perform a suitable simulation. Moreover, based on the value of the signal amplitude, also a fixed value for the threshold is set as

$$Threshold = \frac{Signal\_Amplitude}{2} \tag{2.6}$$

that can be a reasonable average value for this parameter.

2.3.1 Background noise generation

The generation of background sequences follows. In the ideal mode, so with the *ideal_sig* parameter equal to 1, a Gaussian white noise sequence is generated and normalized to unit RMS.

```
if ideal_sig
    BSequence = wgn(1,Npts,0,1,RandomSeed,'real');
    rms_backg = std(BSequence);
    BSequence = BSequence./rms_backg;
    BackgSequence = [t; BSequence];
    save(BackgSeq,'BackgSequence');
    bias = 0;
```

In this case, as can be observed in the code lines, the bias is set equal to 0 due to the fact that in an ideal case there is no need to sum a bias to the background noise in order to obtain a correct simulation of the framework. Meanwhile in real-data mode, an existing background waveform from the measured background is loaded, resampled and normalized in the same way. This yields a unit-RMS background "template" that, once multiplied by the chosen noise amplitude, achieves the exact wanted RMS level.

```
    else
        backg = load(inp_file_bkgd,'data');
        if length(backg.data)<Npts, error('La sequenza ha durata minore del tempo di
    simulazione predefinito !'); end
        BSequence = backg.data(1:round(Fs_bkg/Fs):end);
        rms_backg = std(BSequence(round(t1*Fs):round(t2*Fs)));
        BSequence = BSequence(1:Npts)./rms_backg;
        BackgSequence = [t; BSequence];
        save(BackgSeq,'BackgSequence');
        bias = Bias_bkgd;
    end
```

2.3.2 Pulse train generation

After that the pulse train is created, which is the sequence of the returned echoes over the simulated time window. Even for the pulse sequence generation, depending on the value of *ideal_sig*, an ideal pulse (Gaussian pulse) or the pulse collected from the real world is used to re-create the sequence for the simulation. Each pulse is placed in a random time position within the time window of $10\mu s$, so it ensures that the resulting sequence mimics a real-data acquisition. Then the pulse is normalized with respect to the maximum value of the peak amplitude in order to obtain a unitary peak amplitude.

```
    Nimp = round(time_span/Pr);             % Pulse number to be created for the
    simulation
    PSequence = zeros(1,Npts);
    if ideal_sig
        Sigma  = 2.5e-9;                    % Gaussian pulse' sigma
        tcum = 0;
        for n = 1:Nimp
            t0 = tcum+rand*Pr;
            y = 1/sqrt(2*pi*Sigma^2).*exp(-(t-t0).^2./(2*Sigma^2));
            PSequence = PSequence + y;
            tcum = tcum + Pr;
        end
        Amp = max(PSequence);
        PSequence = PSequence./Amp;
        PulseSequence = [t; PSequence];
        save(PulseSeq,'PulseSequence');
        axis_range = [t(1)*1e6 t(end)*1e6 -0.2 +1.2];
        str = sprintf('\nCreati %d impulsi ideali\n',Nimp);
    else
        % file degli impulsi di ingresso
        inp_pulse = load(inp_file_pls,'data');
        if length(inp_pulse.data)>(Pr*Fs_imp), error('La sequenza ha durata maggiore del
     periodo predefinito !'); end
        pulse = inp_pulse.data(1:round(Fs_imp/Fs):end);
        npts = length(pulse);
        dnpts = Npts-npts;
        [max_pulse, imax] = max(pulse);
        pulse = cat(2,pulse,zeros(1,dnpts));
        tcum = 0;
        for n = 1:Nimp
            n0 = round((tcum+rand*Pr)*Fs);
```

```
            PSequence = PSequence+circshift(pulse,n0-imax);
            tcum = tcum + Pr;
        end
        PSequence = PSequence./max_pulse;
        PulseSequence = [t; PSequence];
        save(PulseSeq,'PulseSequence');
        axis_range = [t(1)*1e6 t(end)*1e6 -1 +1.2];
        str = sprintf('\nCreati %d impulsi reali\n',Nimp);
    end
```

2.3.3 Overall input waveform

The main part of this code is the combination of the waveforms generated before: the background noise is multiplied by the amplitude noise value, the pulse train sequence is multiplied by the pulse amplitude and then they are summed together in order to obtain the total input sequence. This sequence is characterized to have the SNR value defined by the selected working point.

```
TSequence = SignAmp*PSequence+NoiseAmp*BSequence;
```

At the end, the script generates a transcript that shows to the user some information related to the pulse at the corresponding index selected (like peak amplitude, SNR, and so on) as can be observed in Figure 2.6

```
>> Calcolo_Vrms_noise_e_segnale_ver01
>> Par_Sim_GenData

Inserisci un indice della tabella Vsignal-vs-Vnoise nel range [2, 27]
Indice : 4

Creati 10 impulsi reali

OH ampiezza segnale [V] = 0.015248
OH ampiezza rumore di fondo [Vrms] = 0.001522
OH SNR [db] = 20.015722
OH bias [V] = 0.030000
ADC numero di bits = 12
ADC range di tensione [V] = 2.000000
Seme Random = 1046
```

Figure 2.6: MATLAB output transcript

In the case of this work, an index equal to 4 has been selected because it is associated to a pulse amplitude that is almost $16mV$, that is the case of interest for the framework. Moreover, a figure is created with three subplots:

1. the normalized pulse train sequence
2. the normalized background sequence
3. the resulting input waveform

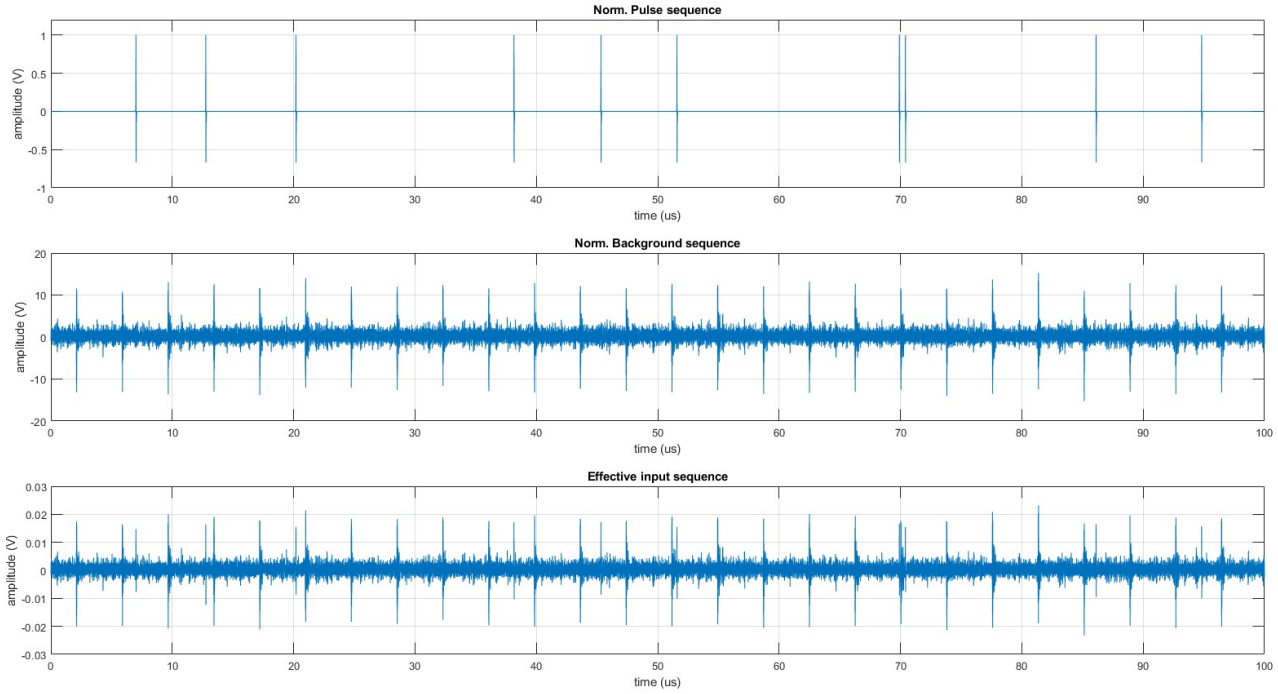

Figure 2.7: Input waveforms

The interference is manifested basically as periodic signal distortions riding on the useful signal. Visually, in a time-domain acquisition, such distortions are of the form of spurious high-frequency 'stripes' or oscillatory patterns superimposed upon the underlying noise floor and the actual echo response. This contaminates the SNR, reducing the system's weak echo detecting ability, specifically those due to remote or low-reflectivity targets.
Figure 2.8 shows a zoom of a "line" of the disturbance shown in Figure 2.7; as can be seen, each line is actually a wavelet with a duration of about $100ns$ and a frequency of about $80MHz$; these parameters will be taken into account to define the best filter to use within the FPGA.

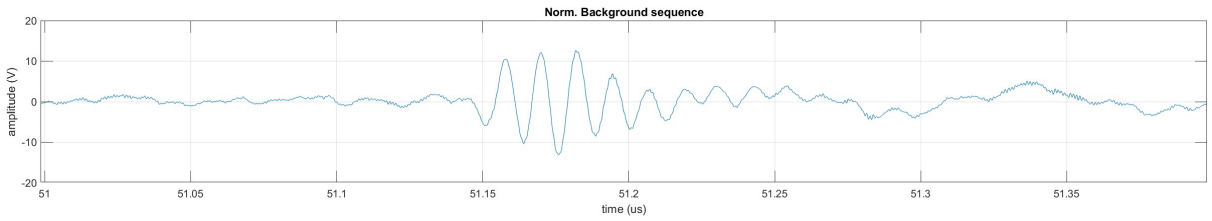

Figure 2.8: Interference zoom

Meanwhile, the absolute pulse generated for the simulation is derived from the collected data, subsampled at $10GHz$ and normalized to 1; this pulse is shown in Figure 2.9.

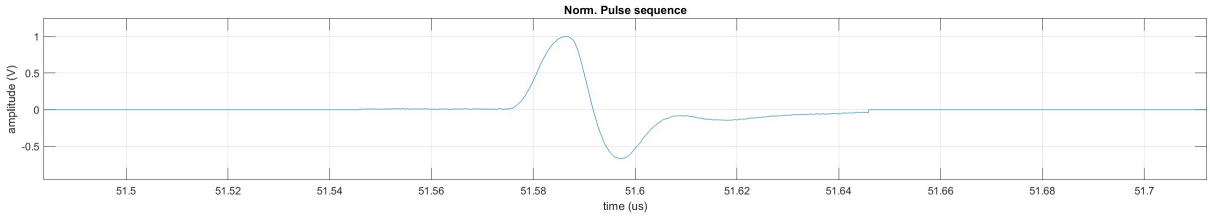

Figure 2.9: Pulse zoom

The script also defines the ADC parameters, i.e., the sampling frequency, the number of bits of resolution and the input voltage range. As well as the ADC configuration, the script sets the FPGA filter parameters, such as the FIR filter coefficients, the associated gain, and nominal delay caused by the filter. The parameters are set to emulate the properties of the real digital processing chain and are critical for accurately modeling the system bandwidth and temporal response. These parameters are not yet applied at this stage of the simulation, but they are used as parameters of the Simulink model described in the following chapter.

Chapter 3

Implementation in Simulink

3.1 Model Structure

The Simulink model is used to simulate the behavior of the opto-hybrid component, that represents the analog front-end of the framework, and the subsequent analog to digital conversion. This simulator has been enhanced in order to allow the processing of both ideal data and real recorded data including environmental interferences. Moreover, the model has been extended in order to include the signal filtering and threshold comparison carried out within the FPGA.

This model reproduces the time-domain evolution of the signals starting from the analog front-end, then through the digitization performed by an ADC and the digital logic approximating the behavior of the FPGA hardware. This digital section includes a low-pass filtering stage, a peak detection in order to correctly recognize the wanted echoes and threshold comparison blocks to determine if a returned pulse is a true echo or not. All together, they capture the behavior of the target hardware system at high algorithmic level.

The results provide an insight into the performance of the overall in terms of detection accuracy, false alarm rate and signal integrity. This modeling step is important because it allows to optimize the design before hardware synthesis, allowing design space exploration and the detection of possible architectural bottleneck. Moreover, by repeating simulations, it allows to fix all the parameters and variables, like filter coefficients, threshold value and so on, leading to a more efficient and reliable hardware implementation.

The Simulink model can be run after the execution of the previous MATLAB scripts. In this work, as previous stated, the simulation will be executed by using the working point corresponding to the index equal to 4 and a seed equal to 962247. The overall structure of the model is shown in the following Figure:

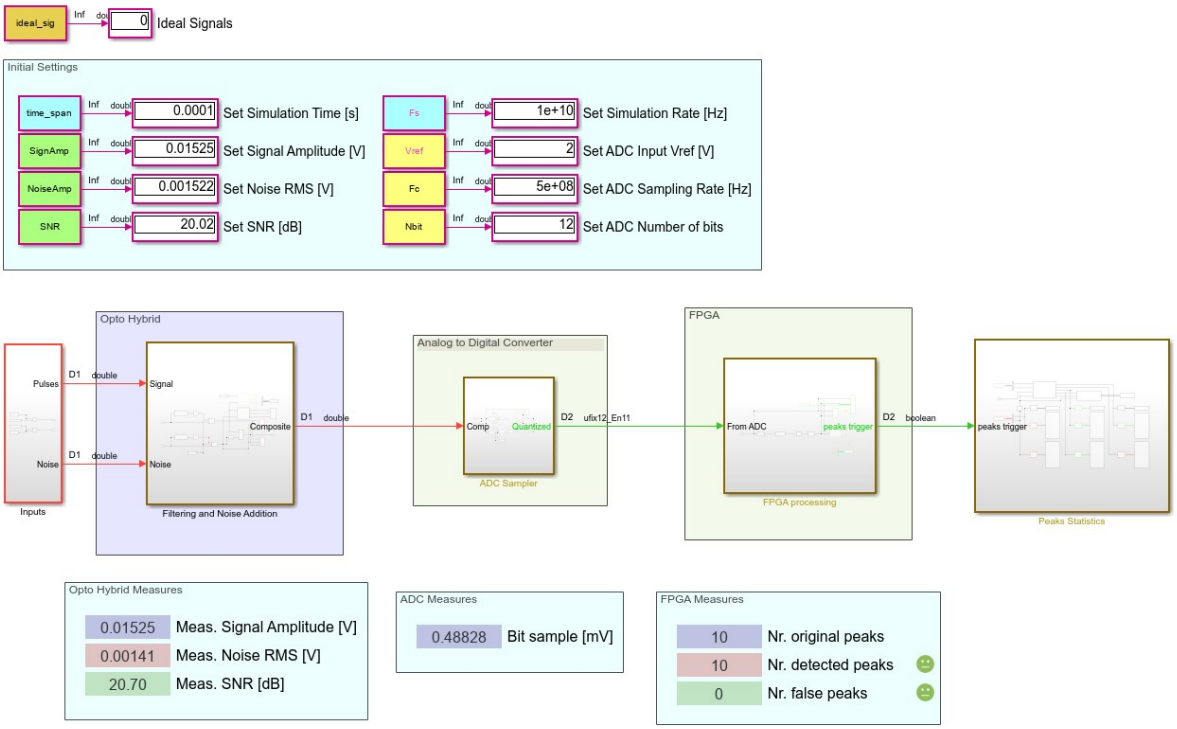

Figure 3.1: Simulink model

As can be observed, the model is based on five main macro-blocks that are modeled at high level in an algorithmic way to allow flexible testing and tuning of the parameters. The main five blocks are:

- **Input** block: this block is responsible to feed the processing chain with the pulse and background sequences evaluated in the MATLAB scripts
- **Opto Hybrid** block: this block behaves as the opto-electronic front-end component that receives the optical input signal and that converts it into an electrical one that is suitable for the following processing chain. It has to shape correctly the signal to maintain signal integrity before the digitization
- **Analog To Digital Converter** block: it is responsible to translate the continuous signal (both in time and amplitude) coming from the previous block into a discrete and quantized signal that is suitable for the processing that is performed in the following block
- **FPGA** block: This block performs the digital signal processing performed within the FPGA. The block implements peak detection algorithms to recognize possible returned pulses, which are those peaks that exceed a predefined detection threshold. This threshold can be tuned to optimize the detection probability and false alarm rate
- **Peaks Statistics** block: This block performs a post-processing analysis of the results. Each peak that is detected is classified as either a true pulse (correct identification of a

real pulse) or a false pulse (false detection due to noise or interference). Additionally, the block identifies if there are some missed detections, i.e. real pulses that were present in the input waveform but that are not detected by the digital chain

All these blocks will be described in detail in this chapter. Moreover, as can be observed in Figure 3.1, there are some displays above the blocks that show how some key parameters have been set in the scripts and that are useful to understand how the simulation will be performed (like the value of ideal_sig, simulation time, amplitudes corresponding to the selected working point, the number of bits of the ADC and so on).

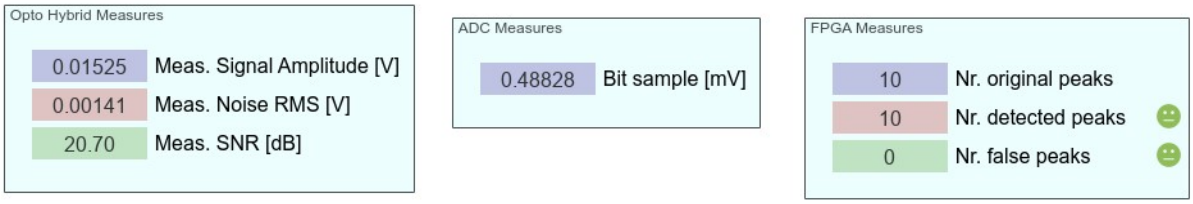

Figure 3.2: Results displays

Meanwhile in the bottom part of the Figure, there are other three displays that show to the user

- the measured values of peak amplitude, noise amplitude and respective SNR in the Opto Hybrid block, in order to evaluate how the input sequence is processed by this component
- the quantization step that comes from the ADC block
- the number of peaks that are correctly recognized and the false detected peaks

These displays are useful to understand immediately the performance of the components and the capability of the system to detect properly the peaks that must be recognized and if there are some false detections.

3.1.1 Inputs Block

The internal structure of the Inputs block, shown in Figure 3.3, represents the beginning of the overall processing chain of the framework and it is the responsible for constructing the two sequences of the input waveform: the pulse and the noise (background and interferences) sequences. Both these contributions are generated in parallel starting from the data that the MATLAB scripts have produced.

As it can be observed, the two contributions are produced into two different branches:

- in the **upper** branch, the file *PulseSequence.mat* is loaded, either with the synthetic Gaussian pulses or true measured pulses, depending on the value of the *ideal_sig* parameter. The waveforms produced by the scripts are in a normalized form (so

unit amplitude) in order to preserve the correct behavior independently from the operating conditions. For this reason, the sequence stored in the file must be scaled by the signal amplitude (*SignAmp*) corresponding to the working point selected

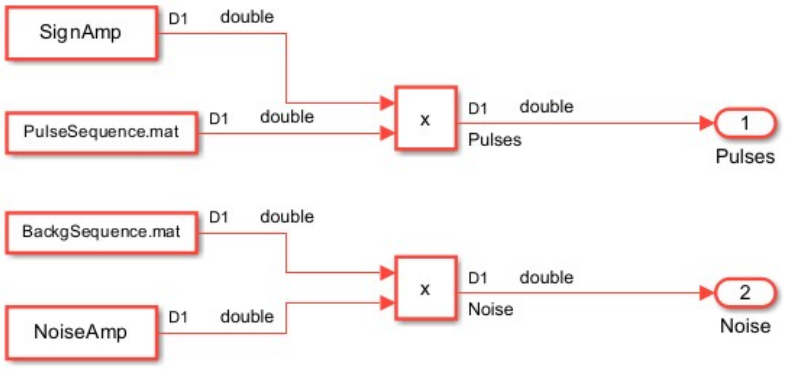

Figure 3.3: Input block

- in the **lower** branch, *BackgSequence.mat* provides the background noise sequence, also stored in normalized form. This sequence may refer to ideal white Gaussian noise (ideal mode) or to real measured background noise (real mode), with all the perturbations from the environment. Similarly for the pulse sequence, the background is in a normalized form and so it must be multiplied by the respective noise amplitude.

The output of the two branches are kept separated in order to allow to the subsequent blocks to process them further, filter each of them in a proper way and then combine them into one input waveform. The choice to treat them as distinct allow the reusability and the flexibility of the simulation environment. In fact in this way it is possible to change one of the two amplitudes (or both) to obtain a different simulation, without the need of re-run the scripts to re-generate the input sequences.
Additionally, on an analysis level, it facilitates the downstream blocks to filter, process, and visualize each of the pulses and the noise independently before the combination takes place, which simplifies the analysis of the impact of each part on the overall system performance. So it represents a critical block in enabling the simulation to simulate a wide range of operating points while maintaining the signal definition unchanged and controlled.

3.1.2 Opto Hybrid Block

The Opto Hybrid block models the analog front-end of the receiver chain, including the optical-to-electrical conversion of the input sequence, its conditioning, and its preparation for the digital conversion. It contains an initial filtering and conditioning, which can represent the photodiode response, transimpedance amplification and any analog filtering

that exists in real hardware. At the beginning of the block, low-pass filtering is applied in order to limit the passband to $50MHz$ in the case of ideal signals. It is not performed for the real-data signals because these traces are measured downstream of this component, so they do need any other filtering process.

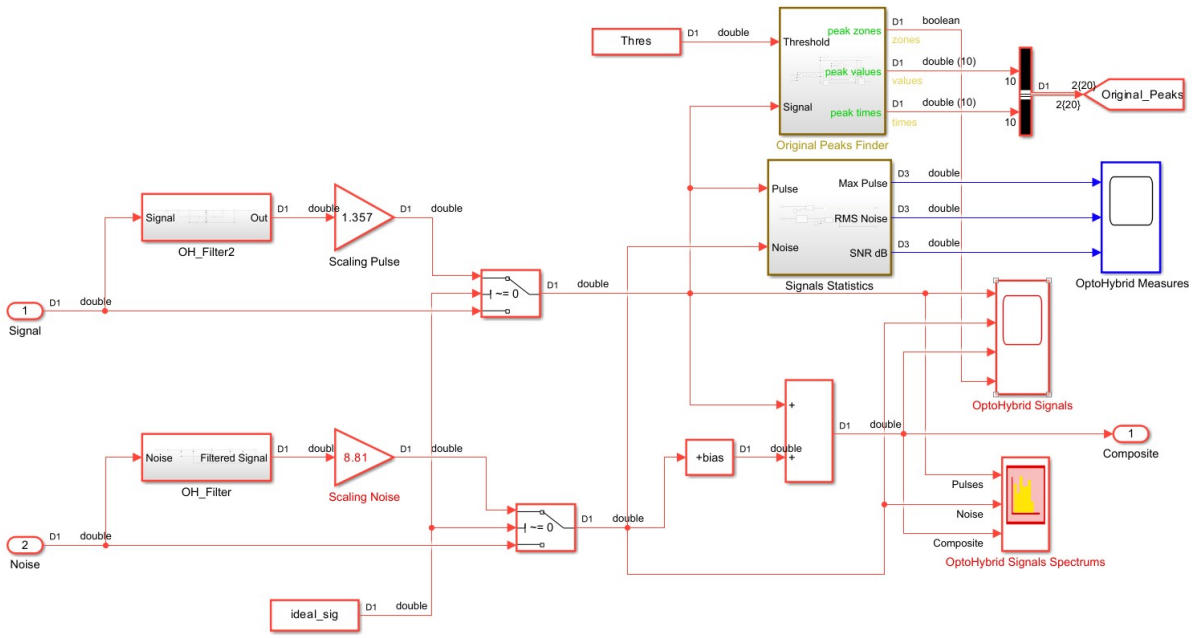

Figure 3.4: Opto Hybrid block

After filtering step the two components, that are still separated to keep their individual signal properties, are multiplied by a proper gain factor that is obtained from the component characterization of the opto-hybrid block and that allows to keep proper values for peak and noise amplitudes. In this way the simulation provides quantitative results that can be compared to experimental ones. After that, the two components are then summed together and a bias offset is also added to the resulting waveform. The bias is added in order to avoid that in the quantization process, performed in the following block, the negative part of the sequence is clipped (remember that the bias is applied only for measured data, while for ideal signals the bias is equal to 0). By shifting the signal, it is possible to center the waveform within the input range of the ADC appropriately. Aside from waveform conditioning, the block performs two important analysis operations that are useful for the statistics performed at the end of the chain:

- Signals Statistics: this operation analyze the input waveform in order to calculate the pulse amplitudes at the peak, the RMS background noise and the corresponding SNR in dB. In particular, the RMS background noise is superimposed with some interferences that increase the power of the noise significantly, by considering the overall trace, leading to an overestimation of the noise level. For this reason, the RMS value of the nose is estimated by using a FIR low-pass filter with moving window size $3.3\mu s$: this value is not a random value, but it is slightly less than

the fixed time interval between two consecutive disturbances. Then, by selecting the lowest output value of this filter over time, the estimation process mitigate the transient disturbances contribution to end up with an accurate representation of the true noise floor

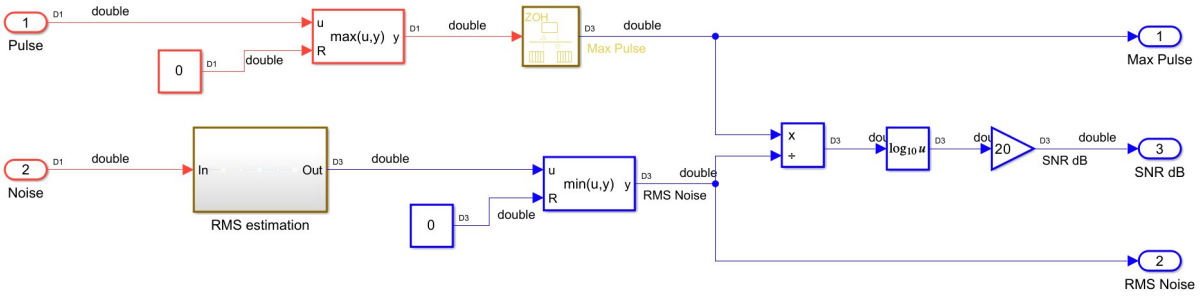

Figure 3.5: Opto Hybrid - Signal Statistics sub-block

- Peaks Finder: this sub-block, as can be observed in Figure 3.6, identifies the pulses that are present in the overall trace. This sub-block always monitors the input signal and, whenever the threshold level is reached by the waveform, it increments a peak counter and starts a run-time search for the peak amplitude in the current peak. During this time, the system analyze the evolution of the peak and it updates the stored value of the peak encountered. Once this peak reaches a stable value, so no higher values of the peak are reached, it is associated to the time of occurrence are captured in a sample-and-hold (S&H [15]) circuit. In this way both amplitude and time-arrival information are preserved with accuracy. As soon as the signal falls below the threshold again, the detection cycle is considered completed and the peak level and time value are stored and into the output registers ("peaks stack" and "times stack") in order to save all the detected peaks. In parallel, the boolean output "peak zones" is generated and it is asserted for the entire duration of the signal when it is above the threshold. All the peak levels, times, and zone info are then added to the "Original_Peaks" data bus for further processing later. This sub-block will be encountered also in the FPGA block and it allows to perform a comparison between analog and digital analysis in the last block of the chain.

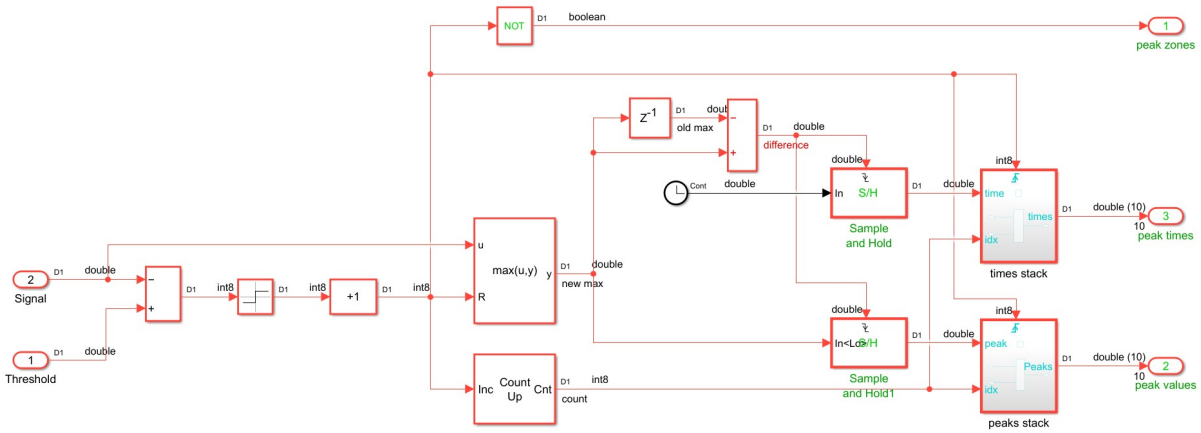

Figure 3.6: Opto Hybrid - Original Peak Finder sub-block

The internal processed signals of the block can be represented both in the time domain and in the frequency domain, providing complementary views of the simulated waveforms' behavior. The time domain representation (Figure 3.7) shows how the signals evolve with respect to time. This Figure displays four different traces:

- The first trace, labeled "Pulses", is the isolated series of received pulses, already normalized and shaped as specified by the input
- Trace "Noise" is the background signal with the periodic interferences, normalized to its RMS, to provide a standard reference against which SNR can be calculated
- The "Composite" waveform is obtained by adding the pulse sequence on top of the background noise and adding the bias offset to avoid clipping. This is the actual signal that will be used as input to the ADC
- A fourth trace is also plotted: the boolean "peak zone" flag, in order to better show when a wanted echo arrives during the simulation.

This representation is extremely useful in understanding the quality of the detector logic to map onto the actual waveform and to check for the consistency of the thresholding approach with noisy inputs. It can be also useful to perform a graphical comparison with digital processed signals to see if any kind of distortion or delay is applied after filtering operation.

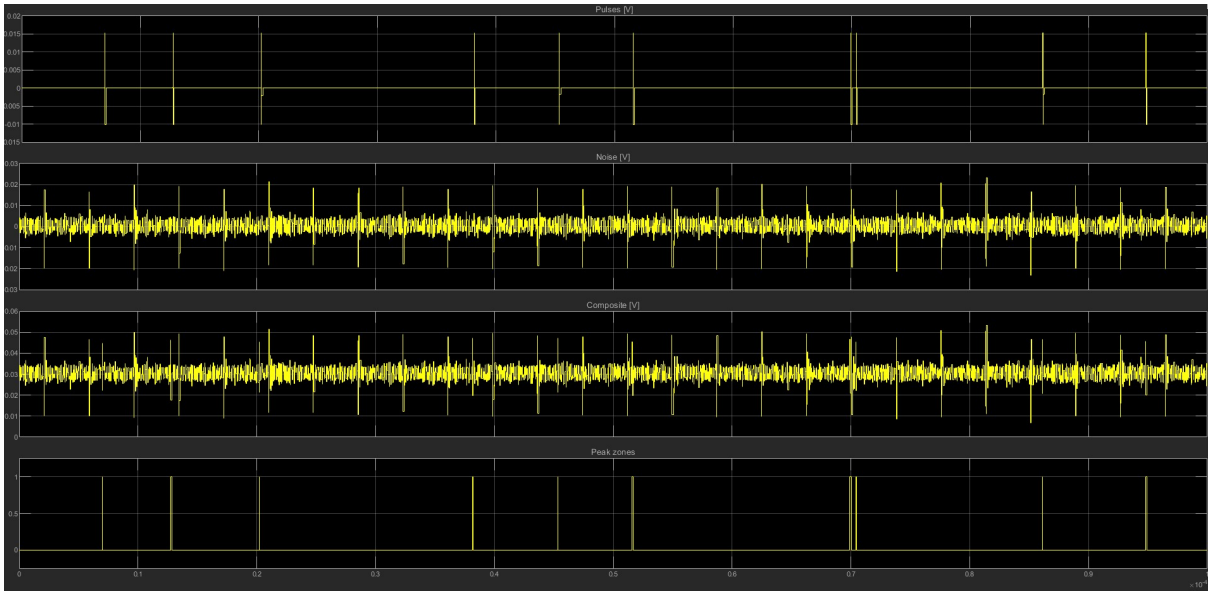

Figure 3.7: Time domain output

In the frequency domain (Figure 3.8), the same signals are shown in the spectrum analyzer in order to obtain information about their distribution. The background noise has a broad spectrum due to its stochastic nature, while the pulses have a more concentrated spectral content that depends on their width and shape. For this reason, the resulting signal has contributions from both sections. By comparing all the spectra, it is possible to see the success of the filtering process. In particular, the low-pass filter in the block limits the passband to 50 MHz and it attenuates high frequency content.

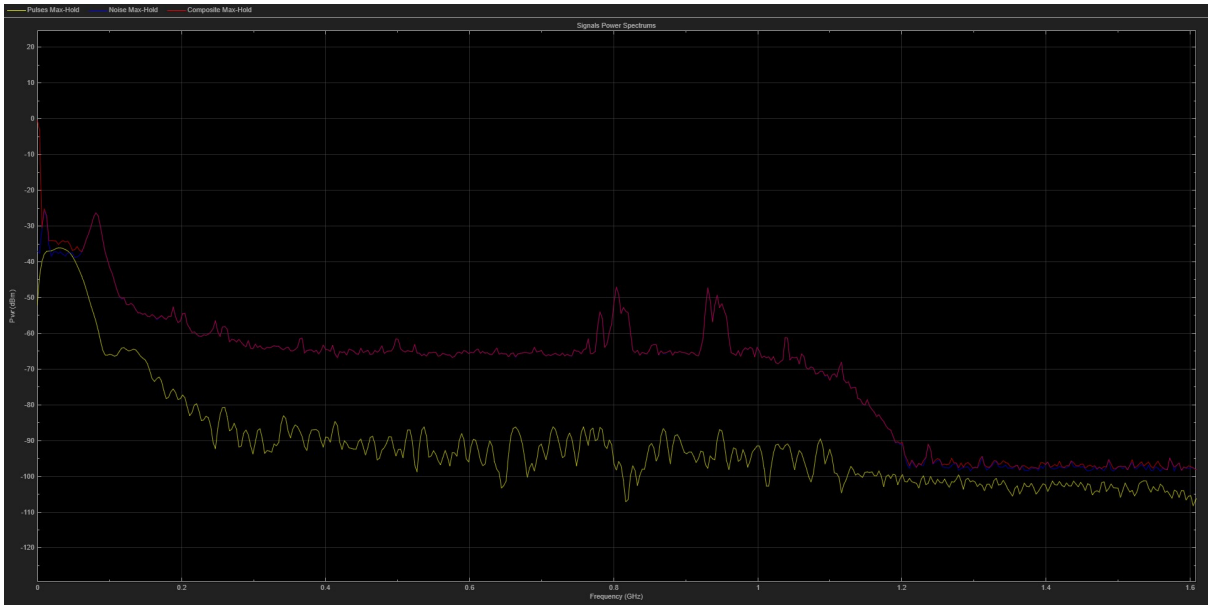

Figure 3.8: Frequency domain output

Both the time- and frequency-domain analyses provide a full view of the behavior of the analyzed block. The former allows to show how the pulses are superimposed with the

background noise, how the bias allows to suppress clipping and how the detection logic senses the true echoes in real time. Meanwhile, latter focuses on the spectral distribution of the signals and very shows the effect of the filter on the overall waveform. This analysis is useful to understand which digital processing method can be implemented in the FPGA.

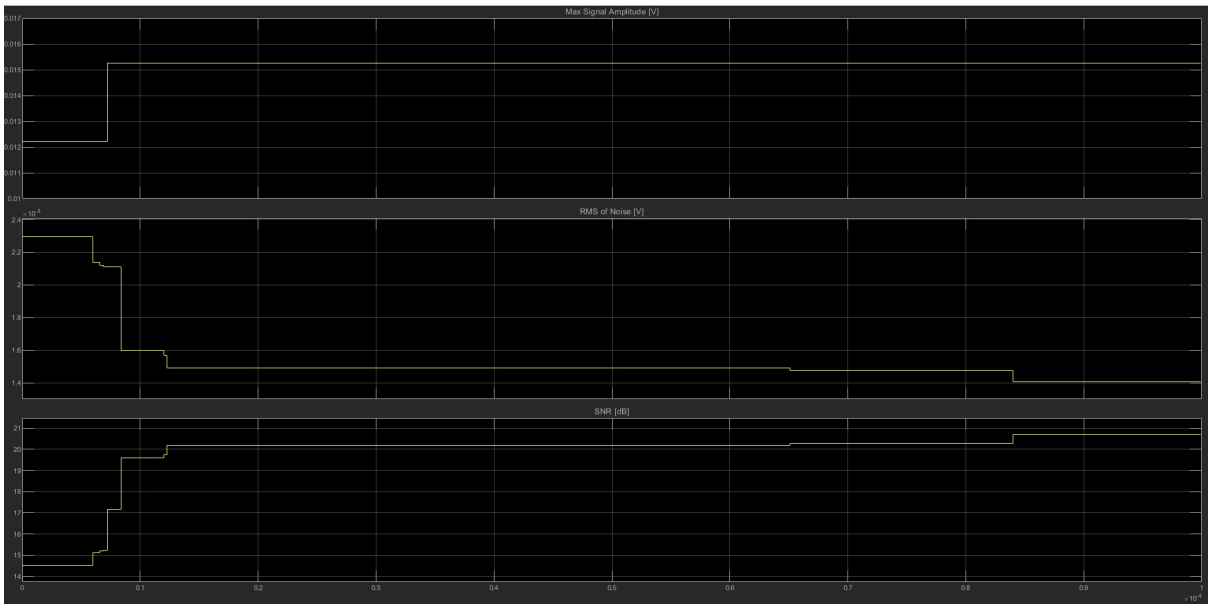

Figure 3.9: Performace parameters

Figure 3.9 plots how the three performance parameters that are always monitored by the Opto Hybrid block change: the maximum pulse amplitude detected, the minimum RMS value of the background noise, and the maximum achieved SNR. These parameters provide a direct feedback of how the analog part of the system can preserve signal integrity in an environment full of noise and disturbances. From peak amplitude and RMS noise values the block can compute the highest detected SNR to quantify the detection ability of the system.

These outcomes are then automatically passed back to the high-level simulation measurements (see Figure 3.2), where they are merged with the result of the subsequent processing steps. By going through this, it is possible to relate front-end behavior to the performance of the system itself, perhaps even seeing where there may be potential bottlenecks or room for optimization. By quantifying the changes over time of pulse amplitude, background noise, and SNR, the Opto Hybrid block provides essential information about the strength of the analog section before digitization, such that the ADC's input is not only appropriately scaled and conditioned but also optimized for trustworthy downstream digital processing.

3.1.3 Analog To Digital Converter Block

The Analog-to-Digital Converter (ADC) block then acts as the interface between the analog front-end described in the previous paragraph and the sequence of digital processing.

The main job of the ADC block is to translate the continuous signal (in amplitude and in time) from the Opto Hybrid block into a discrete-amplitude, discrete-time signal appropriate for the FPGA. So this block performs two fundamental operations: sampling and quantization. Sampling operation discretizes time at the specific ADC clock frequency, while quantization translates the amplitude values into a finite number of steps depending on the ADC resolution in bits and input voltage range. Together, these two mechanisms introduce real non-idealities such as time discretization error and quantization noise, faithfully modeling the limitations of real hardware converters.

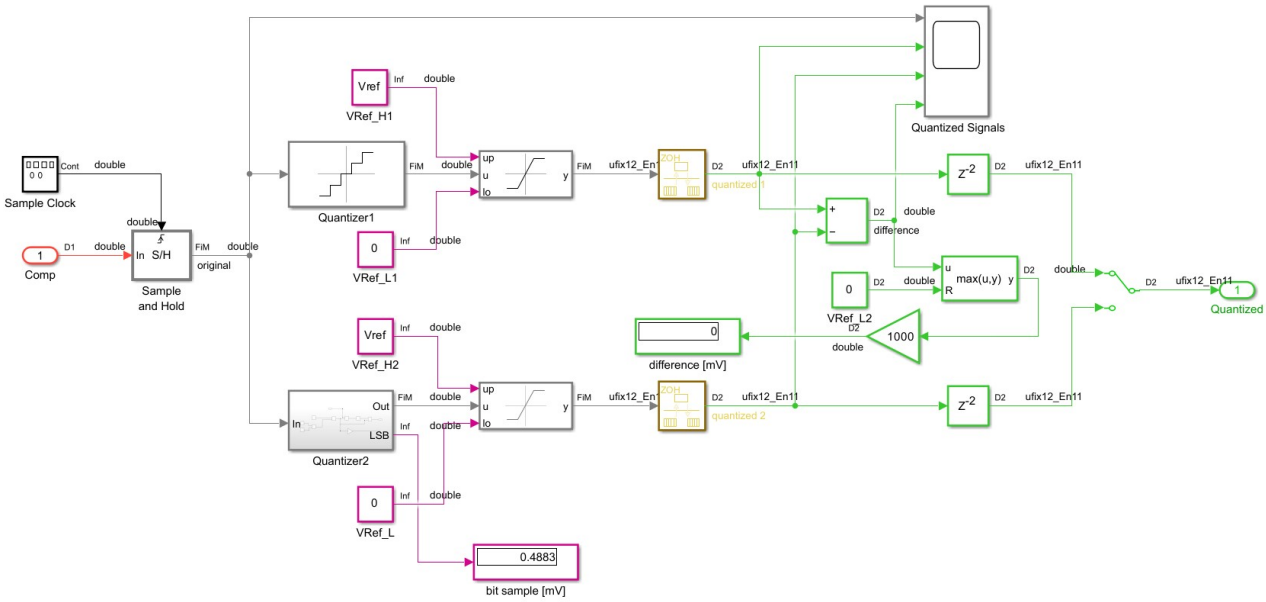

Figure 3.10: Analog To Digital Converter block

From an implementation point of view, the input analog waveform is first sampled at the required frequency, governed by a Sample Clock generator. These sampled values are then quantized, over the input dynamic range $[0, V_{ref}]$. To do that, the model presents two different quantization blocks.

The first block, Quantizer1, is the Quantizer block, which is part of the Simulink signal processing library [5]. This block performs simply the quantization of the continuous signals into discrete steps, where the quantization interval is specified as a parameter in the settings of the block.

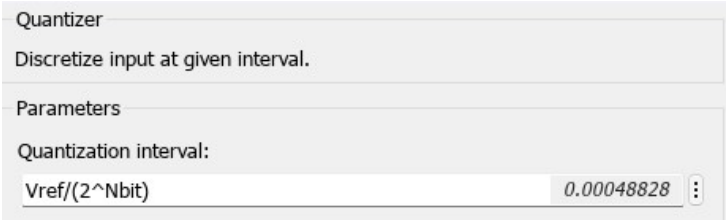

Figure 3.11: Quantizer block parameters

The transfer function is a staircase mapping in that any input value is mapped onto

the nearest digital multiple of delta. This makes Quantizer1 very easy to employ in simulations since it involves minimal setup and it is already available in Simulink.

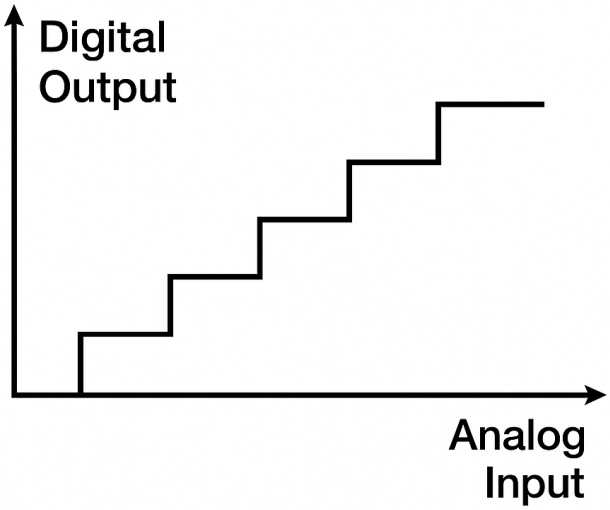

Figure 3.12: Generic transfer function of an ADC

The second quantization building block, named as Quantizer2, is implemented as a dedicated Simulink block, shown in Figure 3.13. This block simulates the actual internal mechanisms of an N-bit ADC, unlike the Quantizer1, so that the process becomes transparent and allows direct mapping to the theoretical quantization formulas. The block computes the Least-Significant Bit (LSB [16]) by using the formula:

$$LSB = \frac{V_{ref}}{2^{N_{bit}}} \tag{3.1}$$

which is the step of quantization, i.e. the minimum voltage that can be detected by the converter. The same equation has been used to evaluate the quantization step in the previous quantizer block. The block then normalizes the input signal by divining it by LSB, obtaining the corresponding code index. After that, the index is rounded by the round operator, which performs the round-to-nearest operation and to it translates the actual value into the nearest quantization level. At the end the output is obtained by multiplying the actual value by the LSB again, achieving so the staircase transfer function of the ADC. Moreover, the block also outputs the current LSB value (scaled by a factor of 1000 to express it in millivolts), providing the quantization resolution.

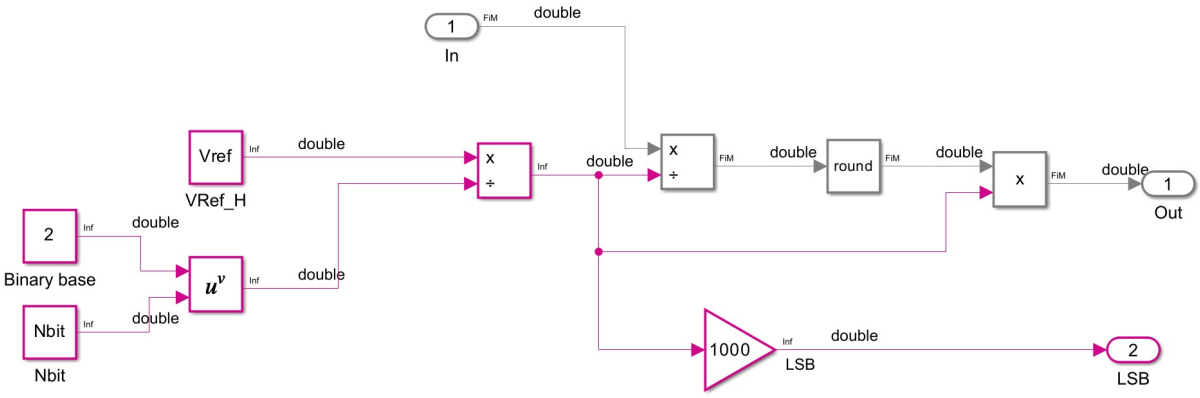

Figure 3.13: Quantizer2 sub-block

This design shows how the quantization mechanism works and how it allows flexibility for testing the impact of different rounding techniques on the resulting digital waveform. It was ensured that the output of Quantizer2 matches that of Quantizer1 when the Round rule is used. For other functions, minimal discrepancies of +/-1 LSB appear, as predicted by the theory of quantization.

In the present setup, the quantizer output is in fixed-point representation in the format

$$\text{ufix}(12,11)$$

that means that every digital sample is represented as an unsigned number of 12 bits, with 11 bits of fractional part and 1 of integer part. This format provides a quantization step of

$$LSB = 0.4883mV \tag{3.2}$$

This LSB value is also displayed in the high-level simulator window (bottom-center panel) so that it can be easily monitored during performance simulation.

Then the block ensures that all samples are in the allowed voltage range [0, V_{ref}]. In fact, the samples that exceed the reference voltage are clipped at V_{ref}, while the negative values are removed because of the bias added in the Opto Hybrid block. In this way, the waveform is transformed into a digital signal with a sample time that is equal to the reciprocal of the ADC clock frequency, that allows the time synchronization in the simulator.

An additional detail concerns the inclusion of a two-clock delay at the ADC output. This technical trick allows pulses contained in the Opto Hybrid block always to arrive a little earlier than those contained in the FPGA block, in such a manner that signals are correctly aligned even when the optional filtering stage is skipped. That is, it avoids ambiguous situations in which detection results from the two blocks could overlap in the wrong manner.

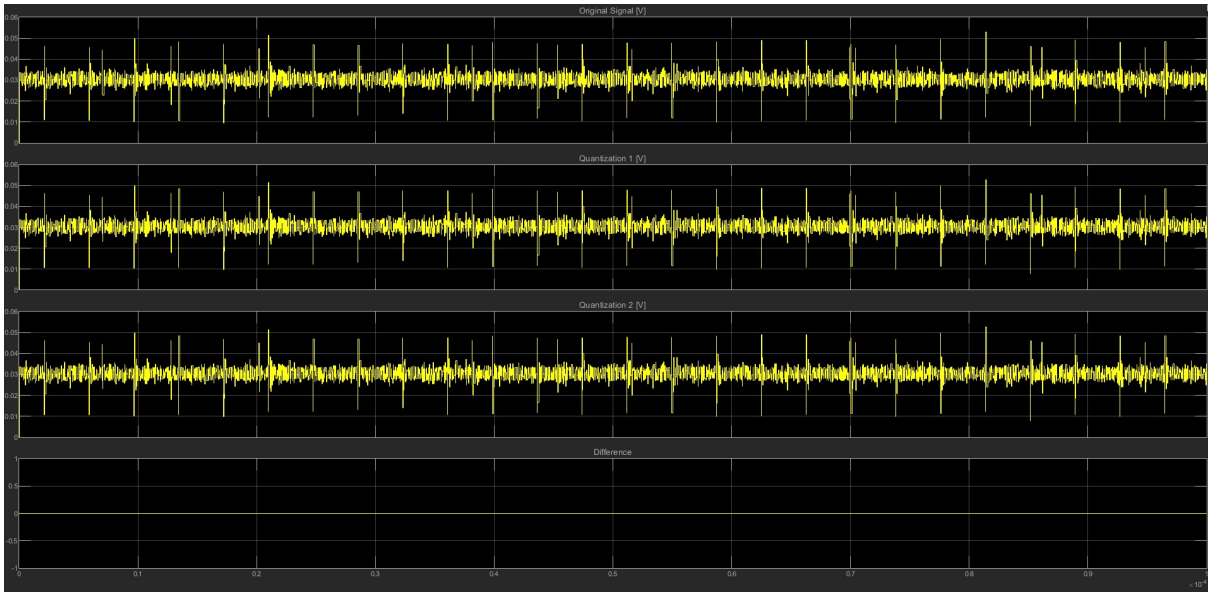

Figure 3.14: ADC plots

Lastly, the operation of the quantization block can be observed from Figure 3.14, a plot of signals going through the ADC block. The top subplot is the original analog waveform of the OH block, the middle subplot is a comparison between the two outputs of the two implementations of quantization (Quantizer1 and Quantizer2), while the bottom subplot is the difference between the two outputs. As can be observed, the difference is always equal to zero (using the rounding function "Round"), confirming that the the two quantizer blocks operate in the same way. In the following Figure, a zoom of the previous snapshot is shown in order to appreciate the quantization process

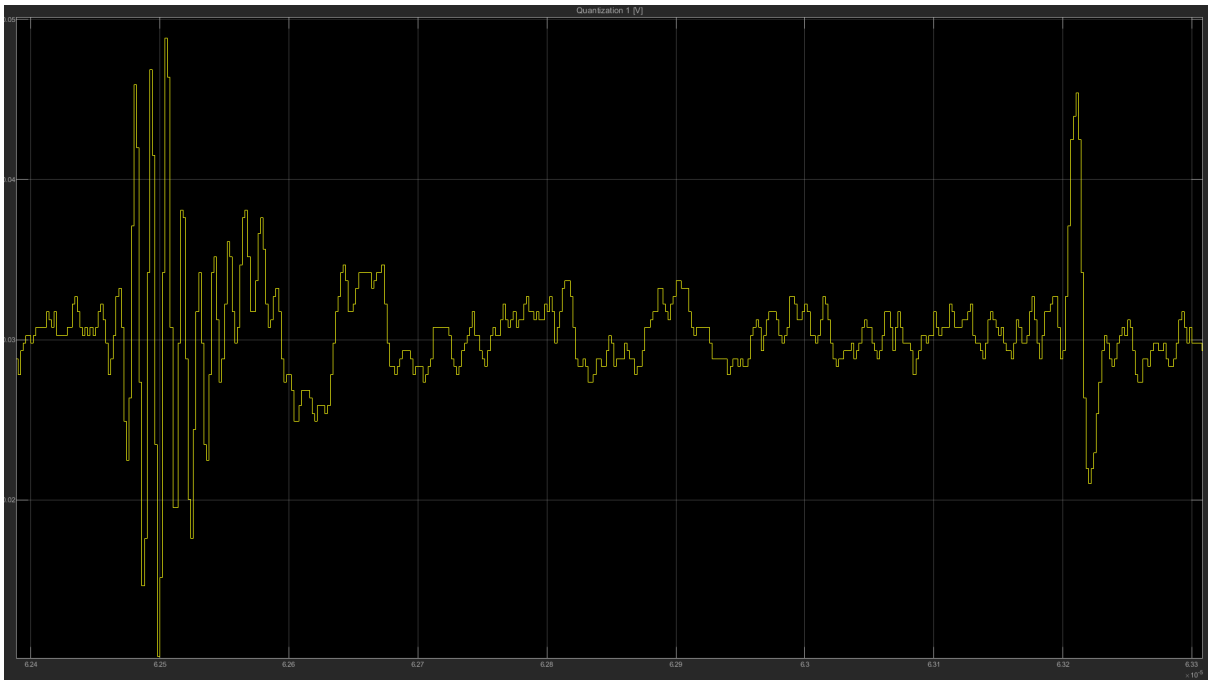

Figure 3.15: Digitalized signal zoom

This verification process proves helpful in making quantization model reliable and consistent with implementations that have already been adopted and making available freedom to switch on and off between native Simulink blocks and custom modules depending upon project needs.

3.1.4 FPGA Block

The FPGA block is the core of the digital processing chain and, for the purposes of this thesis, the most significant element in the entire simulation model. It contains the logic that will be implemented in practice on a reconfigurable digital platform. In other words, this stage is not only a simulation of theoretical behavior, but it is represents the actual hardware design that will be implemented in the following chapters. Its role is to filter out noise and background interferences from the digitized signal provided by the ADC, to perform thresholding functions and to detect and record the occurrence of valid peaks corresponding to true input pulses.

Because the FPGA block is what ultimately handles the conversion of continuous noisy signals to discrete, correctly classified pulse events, it acts as the system-level modeling to hardware design criterial pivot point. Its shape and functionality are thus brought to the fore in this thesis because they provide the foundation for the HDL Coder and High-Level Synthesis flows that will be analyzed and developed in the following chapters.

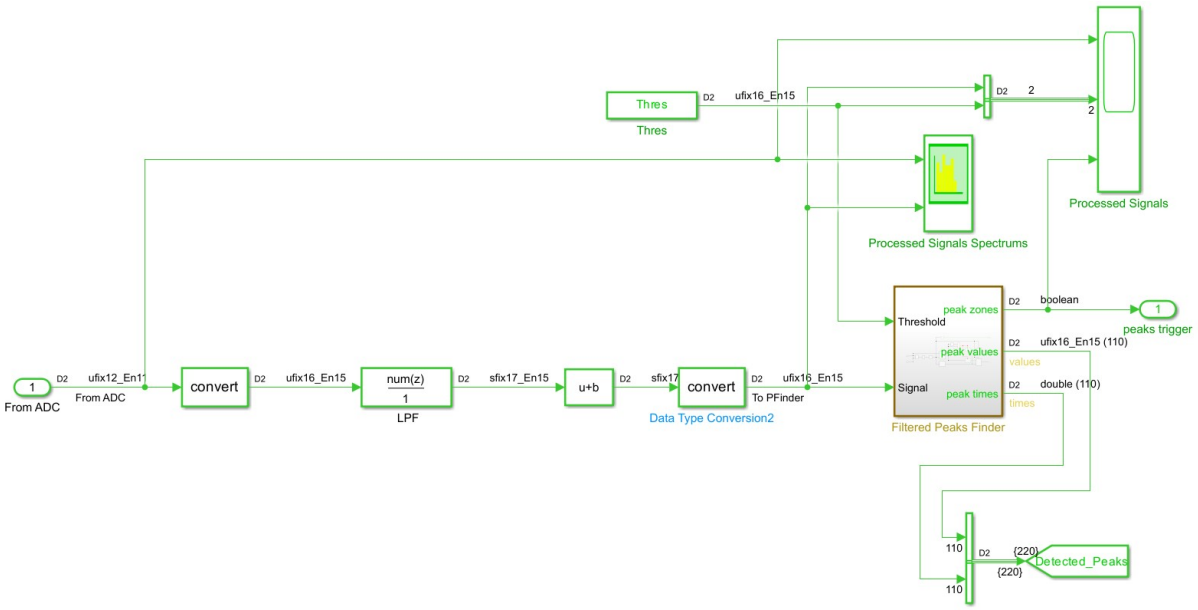

Figure 3.16: FPGA block

As seen in Figure 3.16, at the beginning of the FPGA block there is a data converter block that changes the representation of the data in order to find a trade-off between accuracy and power consumption. The outcome of the ADC, as previously stated, is an unsigned fixed-point type (ufix(12,11)) and this is sufficient for quantized positive signals

but it is insufficient for the filtering stage due to the fact that the FIR filter coefficients are both positive and negative. For this reason the data type is converted to a signed fixed-point number (sfix(17,15)), so with a word of 17 bits made up of 2 integer bits and 15 bits dedicated to the fractional part, which provides more dynamic range. The coefficients are represented by using the format sfix(16,16), a highest fractional resolution signed fixed-point format. Moreover, all of this is done in order to prevent overflow or underflow effects.

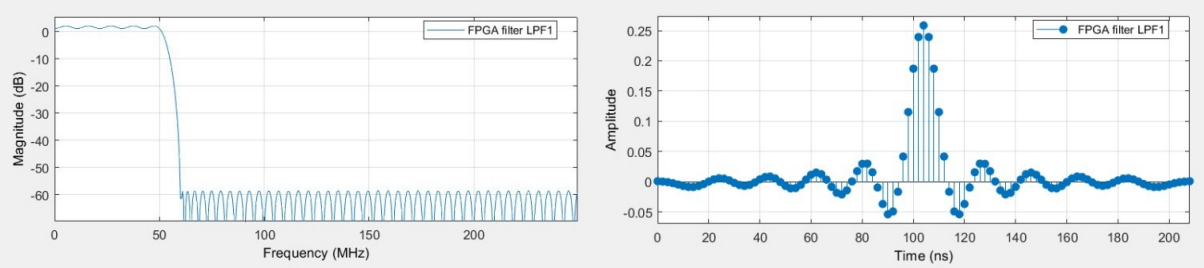

Figure 3.17: FIR filter response and coefficients

The FIR filter in the FPGA block is the core of the system. It is 105 taps long and, as can be observed in the previous Figure, it is a low-pass filter. It allows to remove out-of-band noise and interferences that could give problems in the detection of weak pulses. From a hardware implementation point of view, the filter represents critical component for the cost (in terms of resources) and for its impact to the overall performance of digital chain. FIR filters are widely used in FPGA-based signal processing since they are intrinsically stable and for the fact that it is possible to parallelize it in order to reach higher speed in an efficient way. The FIR filter is designed with a cutoff frequency that is set according to the frequency-domain analysis performed previously, such that the noise spectral region (peaking at $80MHz$) is rejected, while the pulse signal integrity is preserved. The employment of FIR over Infinite Impulse Response (IIR [17]) filtering is based on numerical stability, the delay can be predicted in an easier way and FIR filter are easier to be implemented with respect to IIR ones [6]. All of these characteristics are highly critical when developing hardware to run in real time to reach a good trade-off between performance and resources utilization.

FIR Filter	$y(n) = \sum_{k=0}^{N} a(k)\, x(n-k)$
IIR Filter	$y(n) = \sum_{k=0}^{N} a(k)\, x(n-k) + \sum_{j=0}^{P} b(j)\, y(n-j)$

Table 3.1: FIR vs. IIR equations

After the filtering operation, the signal is translated again to unsigned form (ufix(16,15)), but only after the subtraction of a bias scaled to the filter gain. This operation is performed

in order to restore the DC level of the signal to zero (no more about $30mV$), and so removing the offset applied before the quantization. This step is necessary to operate on a correctly centered waveform.

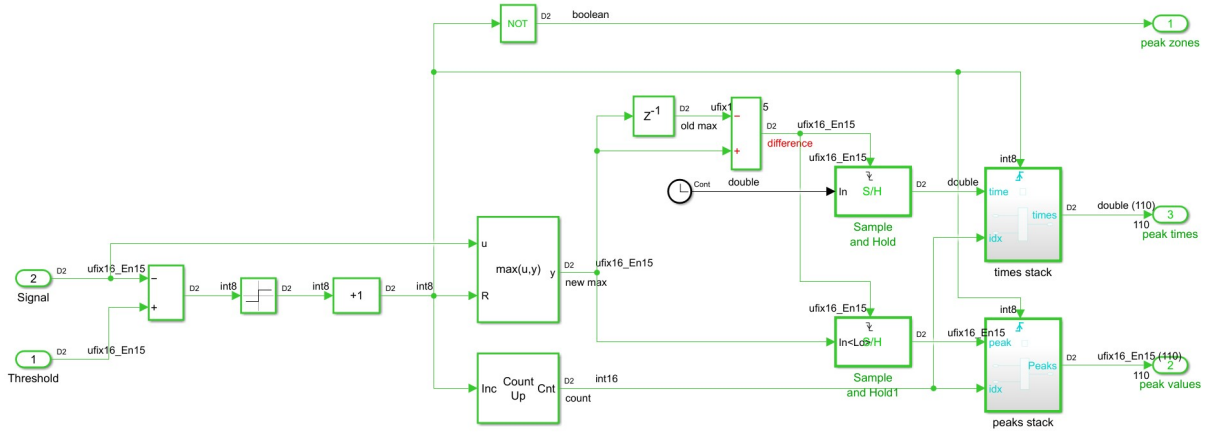

Figure 3.18: FPGA-Peak finder sub-block

Once filtering and conditioning processes are complete, the signal enters Peaks Finder sub-block, whose function is to detect pulses above a constant threshold. The logic used here is identical to that previously described in the Opto Hybrid block, with only the input being in fixed-point (Figure 3.18). The sub-block monitors the input trace and outputs the time of detection and amplitude of detected peaks and the boolean "peak zones" flag, indicating the temporal regions in which the input signal exceeds threshold.

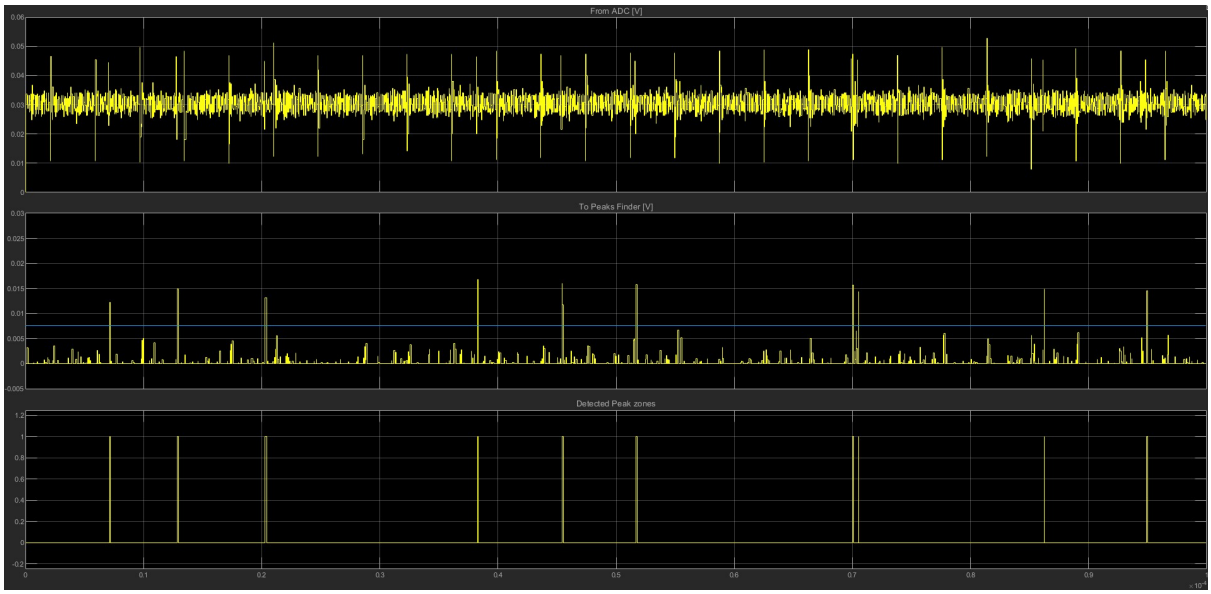

Figure 3.19: FPGA results plot

The correctness of this solution is confirmed in the temporal plots of Figure 3.19

- Top plot: The input signal coming from ADC still has a lot of background noise

- Middle plot: The filtered signal with FIR (threshold emphasized in blue) is cleaner, where pulses stand out from the noise floor and that are the only components that can cross the fixed threshold line
- Bottom plot: The boolean "peak zone" output confirms proper detection of detection windows.

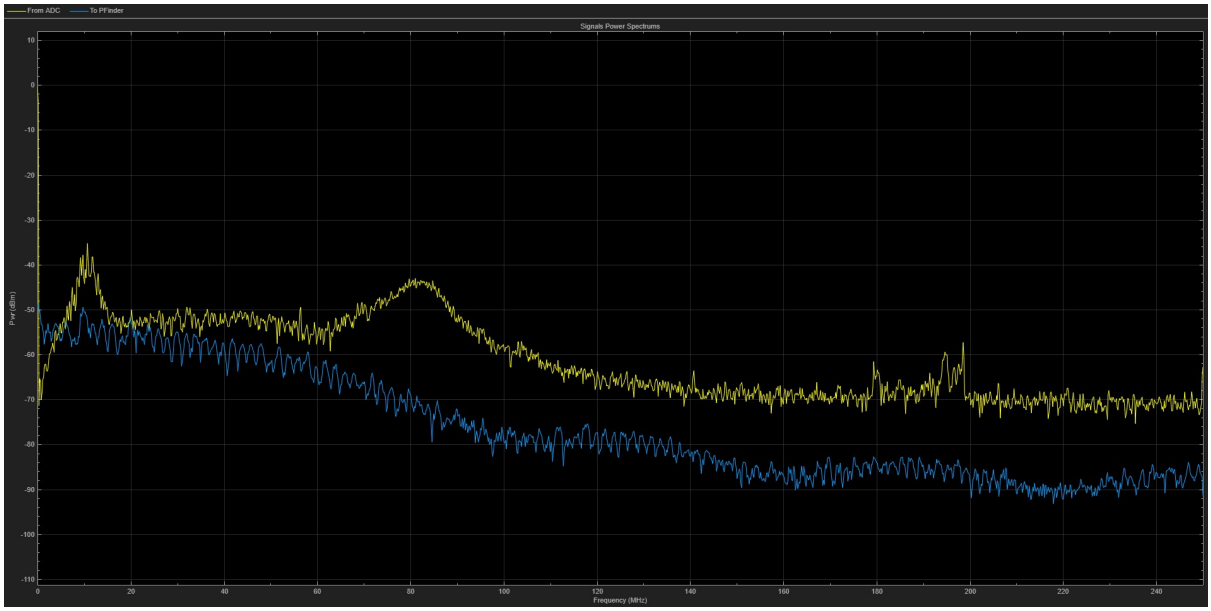

Figure 3.20: FPGA spectrum analyzer

The frequency-domain representation performed by the spectrum analyzer (Figure 3.20) shows how noise dominates the ADC spectrum, in particular at $80MHz$. So by using a $f_{stop} = 60MHz$, it ensures that this contribution is removed and the rest of the spectrum is clean and dominated by the true pulses.

3.1.5 Peaks Statistics Block

The Peaks Statistics block is shown in Figure 3.21 and it represents the end of the processing chain; its purpose is to produce a summarizing statistic of all the peaks detected in the FPGA block, distinguishing between 'true' peaks, and so to actual pulses input into the chain (so called 'correct detections') and 'false' peaks for which such matching does not exist (so called 'false alarms'). These peaks are then shown in two different displays, both in terms of peak value and occurrence time, with together to the list of original impulses with their respective values to compare the results. Moreover, as an additional information, the average peak values are also displayed. While this block is not part of the physical system, it is extremely valuable in simulation, as it provides direct estimates of detection probability, false alarm probability, and missed detection probability. These metrics are fundamental for evaluating system performance and guiding design optimization.

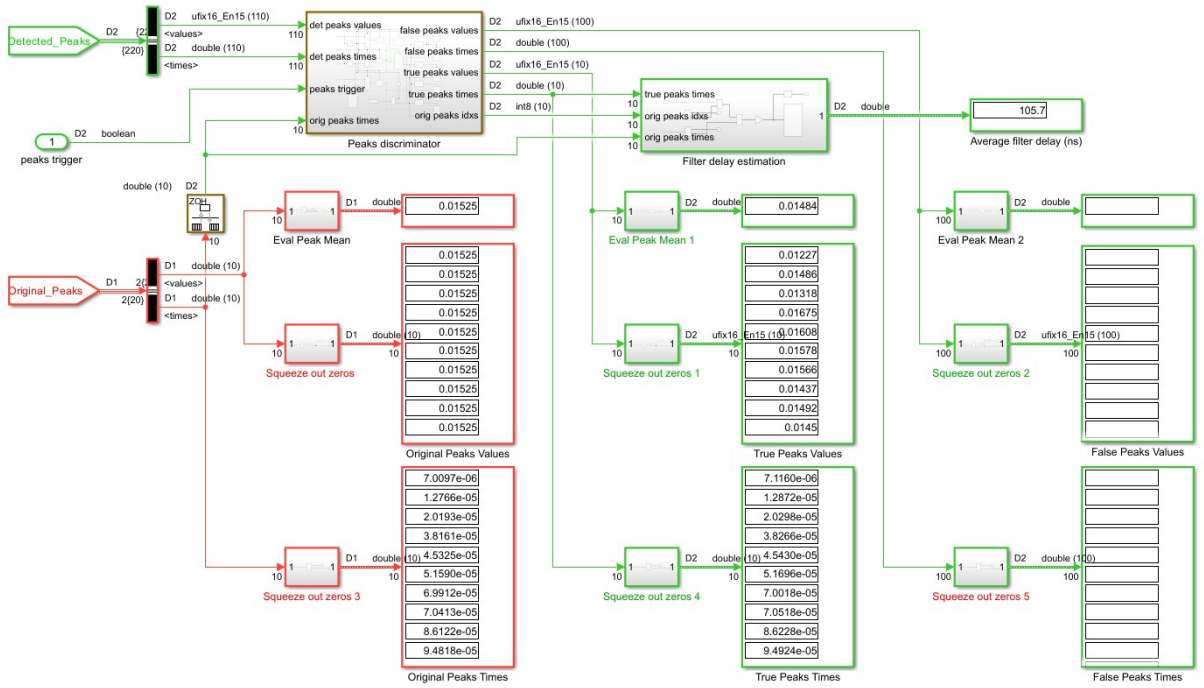

Figure 3.21: Peaks Statistics block

As seen in the Figure above, in this particular case all original impulses are detected and no false peaks; there is some variability in the amplitude of the correct peaks, ranging from a minimum of $12.27V$ to a maximum of $19.32V$. This is due both to the occurrence of the sampling instant regarding the actual time of the maximum and the loss of precision due to signal quantization. However, the average value of $15.37V$ is very close to the input value of the echoes, that is $15.25V$. It can be also observed that the arrival times of the true peaks are approximately delayed of $106ns$ than the corresponding original ones, mainly due to the delay introduced by signal filtering within the FPGA, in particular by the filter. The list of time delays of all the peaks is reported within the 'Filter delay estimation' sub-block, that also computes the average delay.

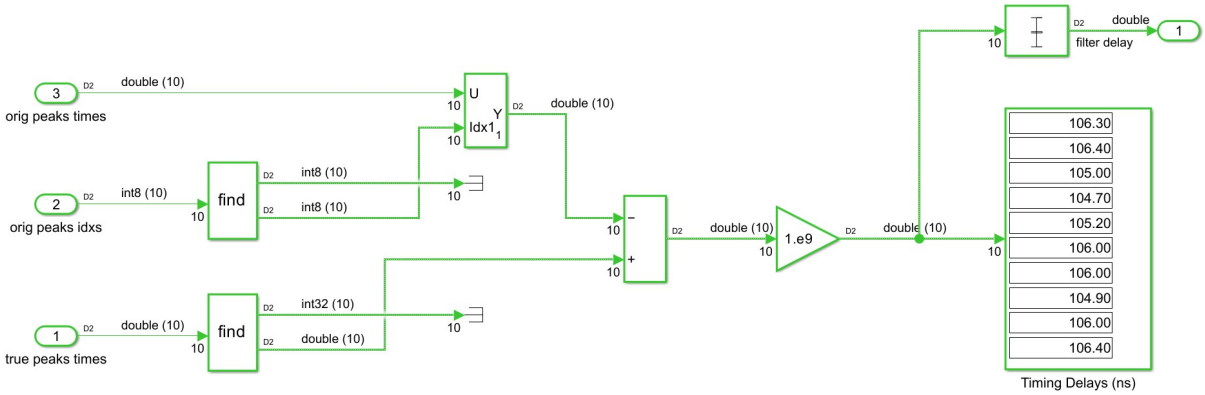

Figure 3.22: Filter delay estimation sub-block

There is another sub-block, Peaks discriminator (shown in Figure 3.23), that allows

to identify which of the incoming peak is a true peak or not. Discrimination occurs if the occurrence time of the detected peak matches the arrival time of the most recent original pulse, taking into account the delay introduced by the filter and considering the set tolerance. On the left of the Figure, there are counters for original peaks ('orig peaks cnt') and for those detected ('det peaks cnt') that are updated during the simulation. On the falling edge of the 'peaks trigger' signal the time of the last original peak and the time of the 'current' detected peak are stored in a pair of S&H blocks. Meanwhile, the value of the current peak is stored in a third S&H. Furthermore, the first S&H is initialized to an appropriate negative value so that any detected peak occurring before the first original peak is classified as false. At this point, in the 'timing discriminator' and 'counters update' blocks, it is checked if the difference between the occurrence time of the detected peak, corrected by the nominal delay of the filter, and the occurrence time of the original one is less than the tolerance in modulus. If so, the true peaks counter ('true peaks cnt') is incremented, otherwise the false peaks counter ('false peaks cnt') is incremented. The increase of one these counters enables one of the two registers for storing the values and time occurrences of the detected peaks, true rather than false.

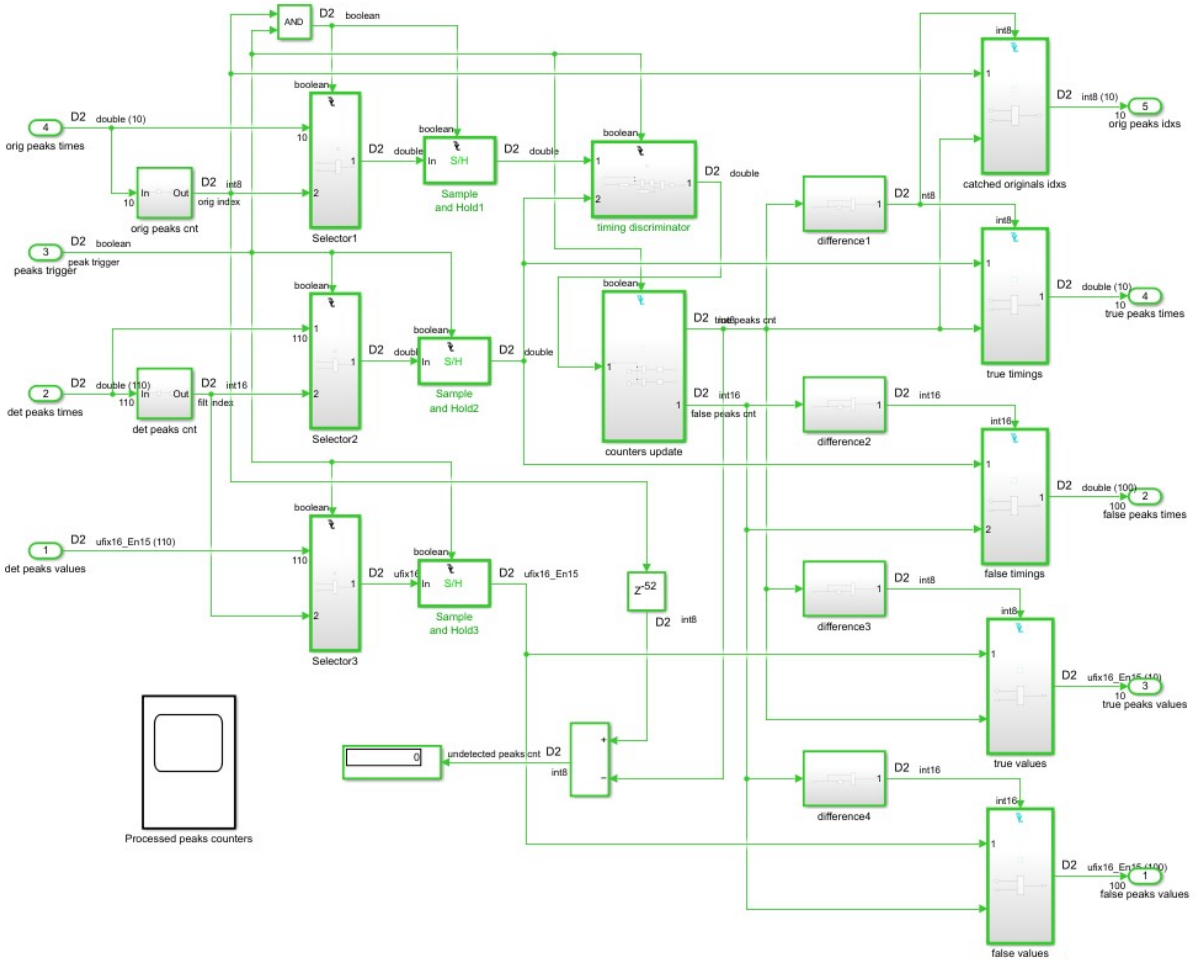

Figure 3.23: Peaks discriminator sub-block

3.2 Simulation results

The difference between the counter of the original peaks and that of the detected true peaks then provides an indication of the missed detections, labeled as 'undetected peaks cnt'. Figure 3.24 shows the time evolution of the peak counters described above: whenever the peak trigger signal ('detected peak trigger') goes to 1, a new peak is detected, incrementing the detected peaks counter. Each time an original pulse occurs (in the Opto Hybrid block), the original peaks counter is incremented too. Based on the time comparison, the detected peak is classified as true or false incrementing the counter corresponding to the outcome of the check. In this simulation, the Figure shows that

- the time trend of the true peak counter is equal to the original peaks one and so it means that there are no missed detections
- the false peak counter remains equal to zero, indicating that there are no false alarms either.

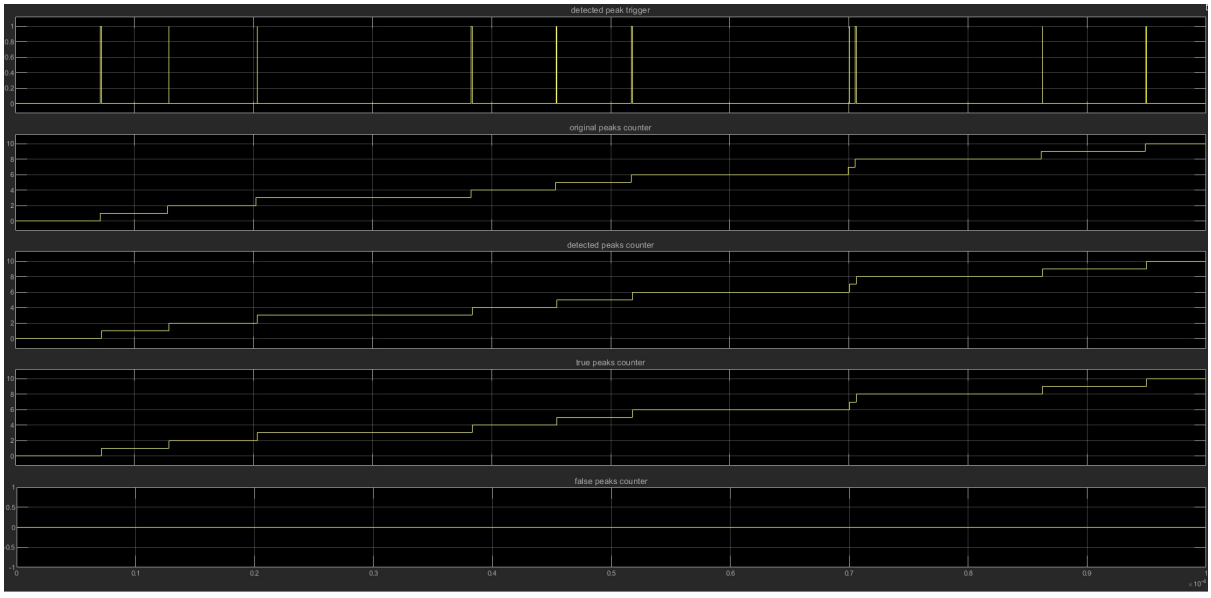

Figure 3.24: Peaks Statistic block - time analysis

Chapter 4

Implementation in HDL Coder

4.1 Simulink HDL Coder

Simulink HDL Coder is a toolbox developed by MathWorks that allows to generate in an automatic way synthesizable VHDL or Verilog code starting from MATLAB functions or Simulink models. This tool is used to fill the gap that there is between high-level description of the algorithm and the real hardware implementation. In this way it allows to the designers to prototype, verify and deploy complex digital systems onto ASIC or FPGA platforms without the needing to write manually the HDL code.

The advantage of using Simulink HDL Coder is in its ability to translate a behavioral description of the algorithm into a clock-accurate hardware implementation. It supports many data types (like integer, signed, unsigned, fixed-point or floating-point), arithmetic and complex operations between data, making it ideal for the implementation of signal processing algorithms regarding filtering, detection and complex structures. The generated code is portable and it can be used to synthesize the model thanks to other tools like Vivado (or AMD, depending on the version) or Intel Quartus.

Within the contest of this work, HDL Coder Tool is used to translate the Simulink model design into HDL code that can be already used for the synthesis and place and route on the FPGA target. In this way the model is translated into a resource-efficient hardware implementation, preserving its behavior while meeting the performance and timing constraints specified. An important feature that must be mentioned is that not all the blocks that can be used in Simulink are suitable for the generation of HDL code: in fact there is a particular library of components (called HDL Coder: Library Browser) that are compatible with HDL code generation. Moreover, in the model there could be some blocks that are not available in the HDL Coder library and for this reason MATLAB functions block is available: it allows to describe the behavior of the wanted block in HLL and it will be then translated into a HDL component.

In order to generate an efficient HDL code, Simulink HDL Coder allows to implement certain guidelines, like the usage of fixed-point data type, so in this way the algorithm

will be implemented with a finite precision of the data, which will be then rounded in order to minimize the quantization error.

HDL Coder also generates in an automatic way the testbench in order to verify functional correctness of the circuit, making it easier to compare the behavior of the hardware with its high-level model. Moreover, it allows to customize the interface by selecting between some standard protocols and blocks, such as handshake protocol, memory ports and I/O blocks, which allow to make the hardware integration easier. The tool provides design checks and code generation reports to assist in this process.

The produced HDL code is clean, readable and compatible with industry-standard HDL coding styles.

The HDL Workflow Advisor is an interactive GUI that assists the user through the stages of HDL code generation and deployment. The typical steps in the workflow are:

1. **Set Target Device and Synthesis Tool**: Define the target FPGA platform (e.g., Zynq Ultrascale+, Intel Cyclone V) and the HDL synthesis toolchain (e.g., Vivado).

2. **Model Checks**: Perform the Design Rule Checks to ensure that the model is compatible with HDL code generation. It checks the usage of fixed-point, avoids unsupported structures and it check if only synthesizable blocks are used.

3. **HDL Code Generation**: Translate the model into VHDL, generating all the needed files and the corresponding testbench.

4. **Synthesis and Implementation**: Launch the external synthesis tool (e.g., Vivado) to implement the generated HDL code, run place-and-route, and analyze timing, area, and resource usage.

All these information, and more, are taken from [7]. In this project, the HDL Workflow Advisor was necessary to make sure that the pulse detection algorithm, originally expressed in a high-level model, could be efficiently mapped to hardware. The tool enabled rapid design iteration, early timing estimation and the generation of HDL modules with minimal manual intervention. This workflow accelerates development and reduces the risk of mismatches between high-level model and hardware.

4.2 HDL Code Generation

The HDL code generation of the FPGA block is one of the key aspects in this thesis process because it represents the transition from algorithmic modeling in Simulink to the development of a hardware-ready design that can be realized on an actual reconfigurable platform. Up to now, the simulation model has been used to examine and confirm the functional behavior of the system in order to ensure that the chosen algorithms for filtering, thresholding, and peak detection work correctly in the presence of noise, interferences

and quantization effects. In order to be included in a real-world digital system, these algorithms must be expressed in a format that can be synthesized by FPGA toolchains. And that is the very purpose of HDL Coder: it provides an automatic bridge from the high-level Simulink description to low-level HDL that can be directly implemented on FPGA logic resources.

The FPGA block represents the digital logic that, as said before, implements three fundamental operations for the algorithm: filtering of the ADC output from interferences and noise,thresholding for discriminating real returned echoes from noise and peak detection and registration with timing information. Due to this, the realization of this block in HDL code is not some academic exercise, but a vital step on the way to a working hardware prototype.

An important observation that must be kept into account is that the FPGA block model is not directly HDL-exportable as it is. This is due to the fact that not every Simulink block is HDL Coder-compatible. So there are some generic logic operators and certain arithmetic blocks that are not, in general, convertible into synthesizable HDL automatically. Therefore, the block must be rewritten by using MATLAB functions or chosen between the HDL-compatible blocks from the HDL Coder library.

4.2.1 HDL Coder suitable model

In this process, HDL Coder preserves the equivalence between the simulated design in Simulink and the respective hardware implementation. The tool converts floating-point to fixed-point arithmetic automatically, generates clocked processes for sequential logic and maintains the Simulink design hierarchy in the HDL description. This automation allows to the designer to do not translate manually the algorithm in Verilog or VHDL, but obviously it still requires that the designer models the Simulink project in the proper way.

In this context, the generation of HDL code for the FPGA block does not simply represent an extension of the modeling task, but rather the beginnings of the actual hardware design task. It shows how high-level design tools can accelerate the development of complex digital systems, shrinking the time gap between algorithm conception, simulation verification, and hardware implementation.

The Figure 4.1 shows the top-level design that will be used for the HDL code generation. The model provides a good overview of the entire data path from input to output of the FPGA block. The input data stream is read from a file (sim_data.mat) containing the quantized samples produced by the ADC model. This choice allows direct testing of the FPGA logic against realistic input data with consistency between the simulated environment and real operating conditions.

The first step of processing at the top level is the converter block. It converts the numerical representation of the input data from the unsigned fixed-point notation from the ADC (ufix(12,11)) to the signed fixed-point notation (sfix(17,15)) required by the

FIR filtering stage, as described in the previous chapter. One of the most important aspects in the top-level designis the numerical representation of the signals. In Simulink every operation must be explicitly declared in fixed-point word lengths because this representation maps directly to how the logic will be realized on FPGA hardware resources. The ADC output is expressed as ufix(12,11), meaning that the signal is expressed in a 12-bit word, with 11 bits dedicated to the fractional part and the number is unsigned (only positive numbers can be represented). This reflects the real behavior of an ADC, which provides quantized samples across a positive voltage range, typically between 0 and V_{ref}. The resolution is approximately 0.4883 mV, and thus smaller variations cannot be represented.

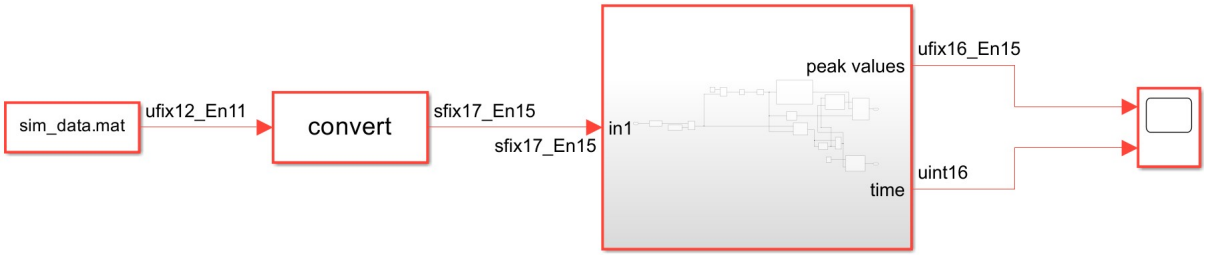

Figure 4.1: Top level model-HDL Coder compatible

However, when the signal arrives to the filtering stage, the unsigned representation can not be used. Digital FIR filters have both positive and negative coefficients because they are designed to remove undesired frequency components by constructive and destructive summation. To be able to properly perform multiplications and additions of such coefficients, the input signal should therefore be in a signed fixed-point representation. The signed representation allows for both positive and negative intermediate values to be represented, as needed for filtering. Moreover, by increasing the word length, it is possible to have a safety margin against rounding and truncation errors due to fixed representation.

This conversion is design strategy that allows to prevent overflow and ensure the wanted accuracy. By setting the dynamic range of the signal to zero, the signed format uses available bits up to their limit, providing room for in-bound peak excursions of the filtered signal, while additional fractional precision allows to minimize the quantization noise floor. The drawback is that the larger word lengths consume more FPGA resources in terms of logic elements and DSP slices. For this reason, the choice of 17 total bits and 15 fractional bits is a trade-off: wide enough to be numerically stable, but not so wide as to be too big in terms of hardware implementation. It is not a technicality, but it is a necessity: while the ADC only gives positive values because it uses unsigned representation, the filtering procedure uses both positive and negative coefficients. So if this conversion is not performed, the filter would not be able to act on the data in a proper manner, and the numeric form would not align with the mathematical requirements of the operation. The extended word length of 15 fractional bits also provides increased

precision to fight against the danger of accumulated quantization errors in the filtering chain.

Once this conversion is performed, the data flow enters the FPGA logic block, which embeds the entire set of the signal processing tasks that were analyzed in depth in Chapter 3, that are:

- the digital filtering to remove noise and interferences located at high frequency
- the subtraction of the bias, scaled by the filter gain, to keep the signal centered in the representable range
- the threshold comparison, which allow to recognize if the filtered signal is corresponding to a real echo or to background fluctuations
- the peak detection mechanism, which is continuously scanning the waveform to find maximum values and store their time of occurrence.

All these operations are implemented using HDL-compatible blocks, so all operations can be directly translated into synthesizable HDL code.

The outputs from the FPGA block are the peak amplitudes that were detected and the corresponding arrival times. These values are then propagated to the output ports of the top-level design.

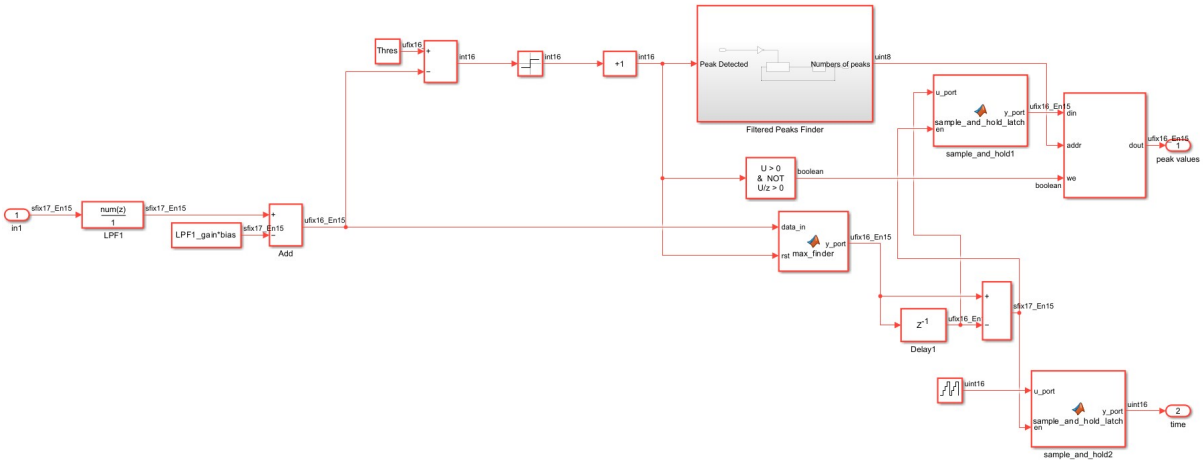

Figure 4.2: FPGA sub-blocks

In addition, by keeping the top level simple and modular, the system is easier to verify and optimize. For example, possible future modifications to the internal FPGA block, such as different filters, adaptive thresholds, or more other detection logic, would be possible without modifying the top-level structure. Not only the verification is simpler during simulation but it also follows best practices for actual FPGA development, where modularity and good hierarchy are the way to successful and effective synthesis and implementation.

The Figure 4.2 presents the internal logic of the FPGA block, where the entire digital processing chain is implemented. The process begins from the input signal which is first passed through the FIR low-pass filter in order to the main mitigate noise. The filter was designed to minimize out-of-band spectral components, such as background interference concentrated at approximately 80 MHz, without degrading the useful spectral content of pulses of interest. The FIR implementation is suitable to FPGA realization because it is unconditionally stable and can be efficiently realized as a series of multiply–accumulate operations distributed across the DSP slices of the FPGA. Filter coefficients are represented in sfix(16,16) to provide signed fixed-point representation with full fractional precision so that the frequency response is as near to the ideal specification as possible. Filter output is transmitted in sfix(17,15) format to increase the word length to avoid overflow and maintain high precision throughout the convolution operation.

After filtering, the signal passes through the bias removal block, where the constant offset previously added (to avoid clipping at the ADC input) is removed. This subtraction is scaled by the filter gain, such that the overall signal amplitude is not changed. The result of this step is a waveform that is centered at zero. This is done because by centering the signal, it allows the following threshold comparator to work as intended. In fixed-point arithmetic, keeping the signal centered also optimizes the use of the available dynamic range, reducing the risk of overflow as well as optimizing the resolution available for small amplitude excursions.

After that, the signal goes to the threshold comparison stage, where it is compared to a fixed value used as threshold in order to detect the peaks. Each time the signal crosses this threshold, a counter is incremented (Filtered Peaks Finder sub-block), starting a potential pulse detection.

```
function y = max_finder(data_in,rst)

%#codegen
persistent max_val

if isempty(max_val)
  max_val=fi(0,0,16,15);
end

if rst==1
  max_val=fi(0,0,16,15);
elseif data_in&gt;max_val
  max_val=data_in;
end

y = max_val;
```

The block that is responsible for dynamically finding the peak amplitude within each detection window is the max_finder block, which is a MATLAB function block, i.e. a block that implements the function described in it. For as long as the signal is above the threshold, the block conducts a run-time search for the maximum value, updating

its internal register if it encounters a bigger value. As soon as the signal does not rise any more, the peak is considered stable and it is kept, together with its time of arrival, in two S&Hs. In this way, both amplitude and time information of the detected pulse are preserved with precision.

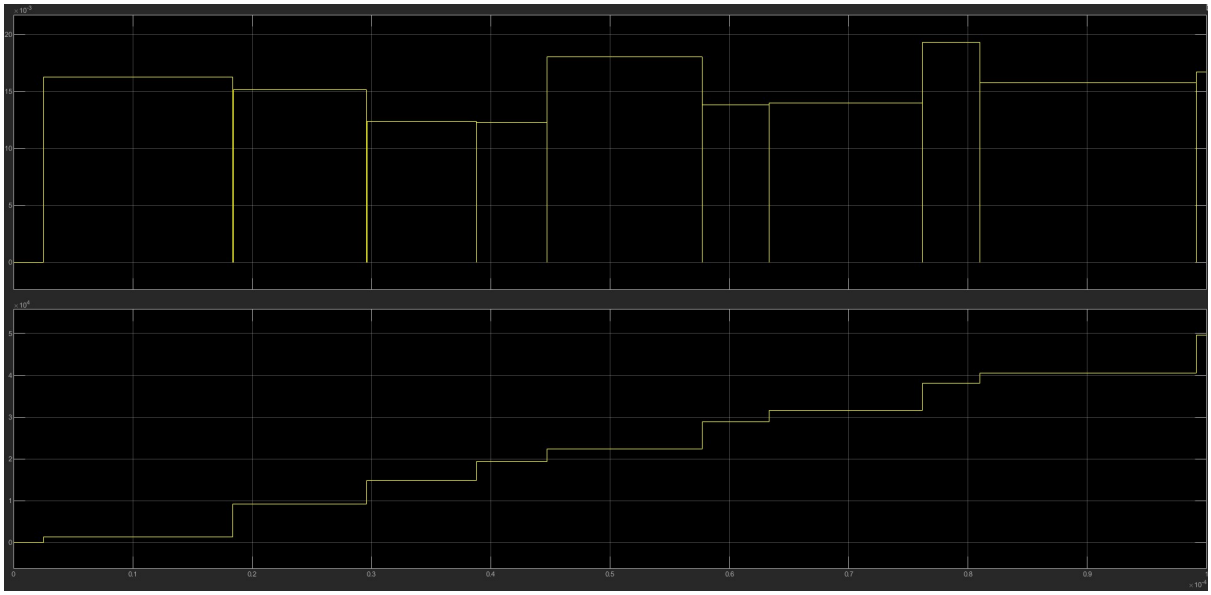

Figure 4.3: Simulink simulation

The peaks found are then accumulated in memory, forming an ordered list of detected echoes that can be processed downstream. The Figure 4.3 shows the main outputs of the FPGA block after HDL synthesis, i.e., the peak values detected and their arrival times, respectively, that correspond to the ones of the Simulink model shown in Chapter 3.

The upper plot shows the stream of peak amplitudes that were detected in the processing chain. The step-like behavior shows how each new peak detection updates the stored value, which remains unchanged until the next echo is found. Meanwhile, the lower plot shows the sequence of arrival times of peaks detected, generated by the S&H circuit dedicated to temporal tracking. The value is a discrete time value (that is the free running counter output) at which the signal maximum was recorded. Each detected peak is always associated with a clearly defined occurrence time.

4.2.2 HDL Coder workflow

By having the adapted model of the FPGA block for the HDL Coder tool, it is possible to generate the HDL code of the system and the synthesizable RTL thanks to the workflow of the tool itself. First of all the VHDL source code of the block is generated, that is automatically produced from the model-based project, by following the same hierarchy and by generating the code for the different main sub-block, as they are threaten as components of the external wrapper. Moreover, HDL Coder can be set to produce a self-checking testbench too, which is fundamental for the verification of the RTL of the

design. This testbench is structured to apply the same input used during the Simulink simulation in order to compare the output of the RTL with the reference output given at high-level.

Finally, HDL Coder can automatically generate a complete Vivado Project for Xilinx devices, by including all the codes needed like the synthesizable code sources and constraints files. In this way it is possible to synthesize and to implement the algorithm into a real programmable device, obtaining so a real hardware implantation. So this workflow allows the translation from the high system-level design to the real hardware-level realization.

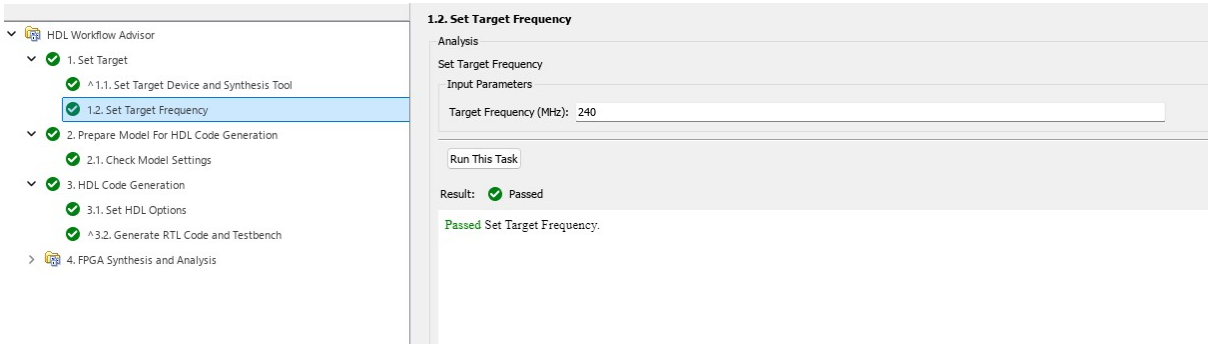

Figure 4.4: Workflow Advisor Steps

First of all, before starting with the HDL Coder Workflow, different general settings must be set in order to ensure the correct realization of the source codes:

- First of all, VHDL has been selected as language and the tool has been set to generate the code not for the overall project, but for the LPF subsystem, that is the relevant part of this work
- Then the synthesis tool and the target FPGA must be set in order to generate a proper project to be synthesized:
  1. The synthesis tool is Vivado, version 2023.2
  2. The target platform is an Artix7 xc7a200tfbg484-2, that is a proper device for the algorithm that must be implemented
- After this fundamental step, some optimizations has been set in order to optimize the hardware from the beginning like pipelining and resource sharing of the multipliers
- Some global settings has been set too, in particular the synchronous reset asserted at high-level.

All the other settings have been left as default. At this point, it is possible to start with the HDL Coder workflow

1. In the "Set Target" panel, there is first of all a short recap of the target device and synthesis tool settings, then it is possible to set the target frequency that, as can be observed in Figure 4.4, has been set equal to $240MHz$. This value has been chosen as final target frequency due to the fact that after different trials with higher frequencies, starting from an initial target of $250MHz$, it is the one that allows to obtain good performance without any negative slack

2. Then there is the second panel where a check of the model-level settings for the code generation is performed. All the blocks that are used must be compatible with the HDL Coder library, different checks the data format and the name of the ports and of the subsystems created and so on

3. After that, there is the "HDL Code Generation" panel, where firstly it is possible to set some last options for the code generation, then the RTL code and the self-checking testbench are automatically generated

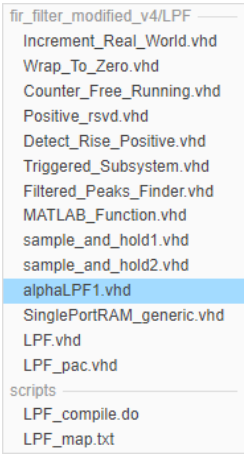

Figure 4.5: HDL Coder generated files

4. Lastly there is the "FPGA Synthesis and Analysis" where it is possible to automatically generate a Vivado project that can be loaded on the tool to start the synthesis and the implementation.

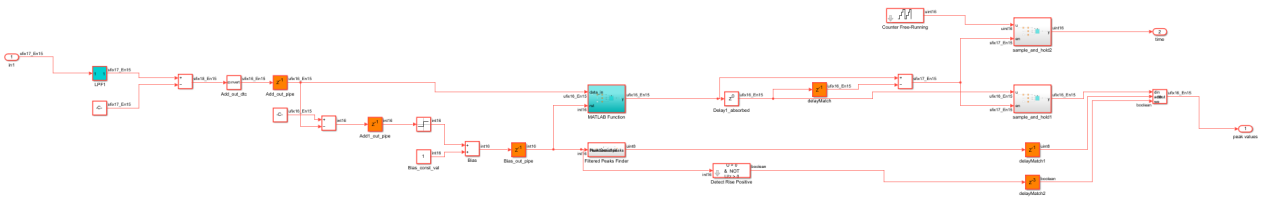

Figure 4.6: Generated model

After all these steps, all the various VHDL files of the different components are generated (as can be observed in Figure 4.5) with together to the testbench and the

Vivado project. Moreover, the scripts that allow the compilation and simulation on QuestaSim are also generated in order to automatize the simulation process. With together to this files, anew model of the subsystem is generated in order to observe, also in a graphical way, how the blocks have been implemented and how the optimization have been performed:

- There are some z^{-1} orange blocks, which represents the pipeline registers that have been inserted by the tool in order to split the combinational paths
- The light blue blocks, in this case the FIR filter and the max_finder blocks, represent the blocks in which pipeline has been applied inside them by adding some registers at the input or at the output of the blocks.

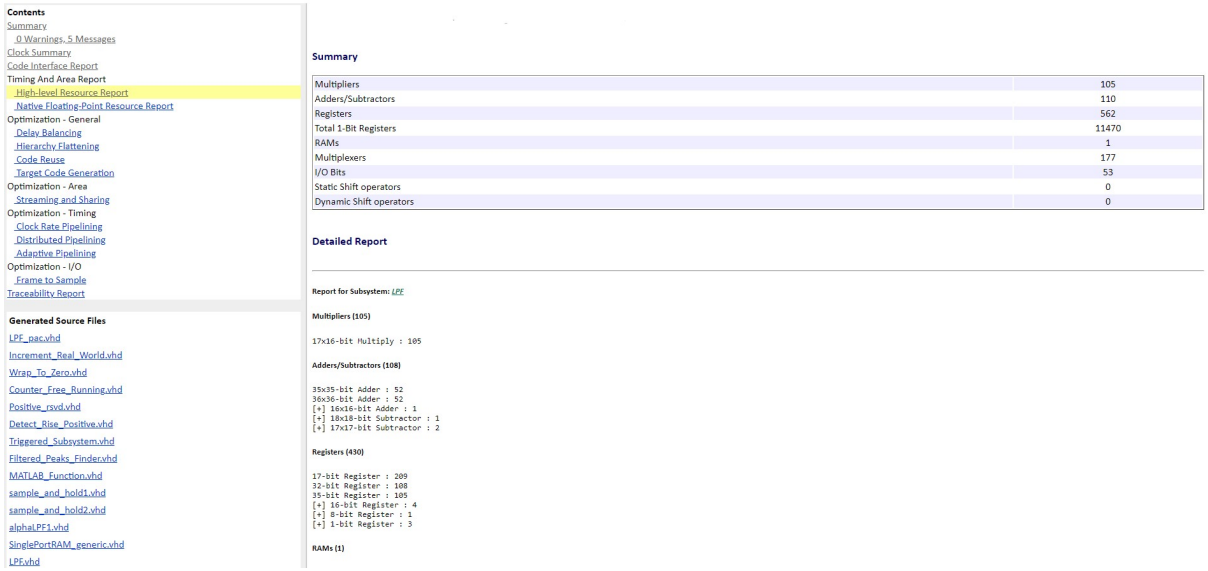

Figure 4.7: Code Generation Report

As a final result, a "Code Generation Report" (Figure 4.7) is created where it is possible to see a lot of information regarding the sources code generated, the clock and the optimization performed by the tool. It also gives some high-level information related to the resource utilization, that is an estimation evaluated by HDL Coder in order to see how many multipliers, registers, memories and other components will be used to implement the algorithm in the target FPGA. It is only an estimation, so these numbers could be similar to the final results but not exact. The final estimation on the resources, which are the ones used for the final comparisons, that are needed for the implementation is given by Vivado after the place and route process.

4.3 QuestaSim Simulation

After the generation of VHDL code and its respective testbench, the next step is to perform an RTL-level simulation with Mentor QuestaSim. This step is of very important

because it guarantees that the HDL description generated in an automatic way by HDL Coder is functionally equivalent to the original Simulink model. In practice, the testbench designed during the Workflow Advisor phase uses the same input vectors used in the Simulink environment and, in order to make a real comparison with the Simulink model, it compares VHDL implementation outputs with the golden reference outputs obtained during high-level simulation.

After running the scripts generated by the tool to automatically compile all the VHDL source files and the testbench, the simulation has been performed. Thanks to this simulation, it is possible to see that the FPGA block correctly processed the ADC input samples, filtered them and performed thresholding, peak detection and produced the expected outputs that are the peak amplitudes that were detected and their times of arrival. While viewing the waveforms in QuestaSim,it was double-checked that all peaks that were detected had a precise match with the ones from the Simulink simulation. Moreover, it is important to say this kind of simulation cares about the real latency of the circuit, while the simulation performed in Simulink can be considered an ideal simulation due to the fact that it does not consider any time delay. In fact from this simulation it is possible to estimate that the latency, evaluated in clock cycles, is equal to 15, which is an important value that will be considered in chapter 6 to make a comparison with the other implementations.

	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}
Peak value [mV]	16.27	15.14	12.39	12.27	18.04	13.82	14.01	19.32	15.78	16.69
Peak value (ref) [mV]	16.27	15.14	12.39	12.27	18.04	13.82	14.01	19.32	15.78	16.69
Time value	1255	9200	14800	19390	22356	28867	31659	38081	40485	49564
Time value (ref)	1255	9200	14800	19390	22356	28867	31659	38081	40485	49564

Table 4.1: QuestaSim Simulation results

The results of the simulation are shown in the Table 4.1. Each column represents a detected peak (10 peaks for the overall simulation), while in the rows there are the peak values and the time arrival values compared with the golden outputs produced by the high-level simulation (the ones with the "(ref)" tag). The peak values are both represented in mV, while the time values are given in ticks (counter output). In order to obtain the time value in [s] it is possible to take the result written in tick and multiplying it by $2 \cdot 10^{-9}$, which is the sample time of the system. The results written in the table confirm the equivalence between Simulink model and the RTL produced without any differences. Numerical differences in precision of small values were ruled out, since the fixed-point mode had already been accurately set in Simulink and maintained consistently carried over to the HDL model.

4.4 Vivado Synthesis & Implementation

After the simulation on QuestaSim in order to verify the design correctness at the RTL level, the last step to conclude this process is to perform the synthesis and the implementation in Vivado to obtain the real performance of the hardware implementation,its power consumption and its resources utilization.

The project automatically realized by HDL Coder by default already includes VHDL source files, the testbench and all the necessary constraint files. By executing this project in Vivado, the synthesis tool is able to translate the abstract RTL description to a gate-level netlist, where all the operations such as additions, multiplications and comparisons are performed by real physical hardware components such as LUTs, registers, memories and DSP slices. Vivado also optimizes the design depending on the strategy chosen to realize the synthesis and the implementation: in fact there are different strategies, each one with a different goal in order to better optimize a characteristic of the design, such as performance, area or power consumption. The synthesis reports provide an initial estimation of the maximum operating frequency, of the static and active power consumptions and of the resources used for the realization of the algorithm, but more realistic values are given by the implementation reports which show the same quantities but after the place and route process. By taking into account these quantities, in particular, it is possible to evaluate the efficiency of the design and it is possible to create a concrete point of reference to different implementations or future optimizations.

Before proceeding with the synthesis and the implementation, some settings have been configured in order to choose a specific strategy to be used for this process and for the one performed in the next chapter: in particular, it has been chosen to optimize the performance of the device, for this reason the two strategies are:

- Flow_PerfOptimized_high for the synthesis (for more information, see the User Guide [8])
- Performance_Retiming for the implementation (for more information, see the User Guide [9])

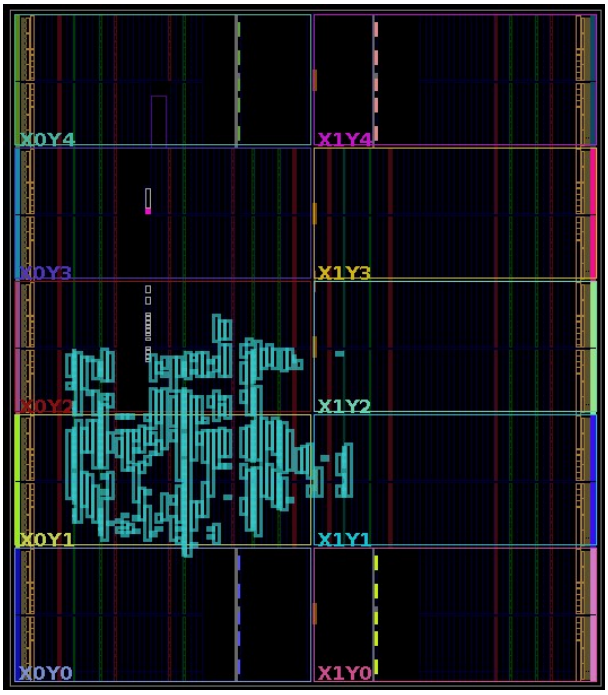

Figure 4.8: Placement view of HDL Coder project on the FPGA

In the Figure 4.8 it is possible to see how the tool has been performed the physical distribution of the implemented logic in the FPGA. The resources are concentrated in the lower-left region of the programmable device, because by having a compact placement it is possible to minimize routing complexity in order to optimize the timing. In fact in this way it is possible to minimize the interconnection delays in particular for those components that are critical in terms of speed, such as the FIR filter. The table below shows an overview of the main results obtained from the Vivado implementation, including the number of resources that has been used with the percentage of utilization, the total power consumption, measured in [W], and the timing measures in [ns].

Timing Result	
Worst Negative Slack (WNS [18])	0.137 ns
Power Analysis	
Total On-Chip Power	0.783 W
Resource Utilization	
Slice LUTs	5612 / 134600 (4.2%)
Slice Registers	6036 / 269200 (2.2%)
Slices	2085 / 33650 (6.2%)
Block Random Access Memory (BRAM [21])	0.5 / 365 (0.1%)
DSP Slices	103 / 740 (13.9%)

Table 4.2: HDL Coder-FPGA implementation results in Vivado

Starting with the timing analysis, the positive results in the table highlight that

the timing constraints are fully satisfied, without any setup or hold violations with a clock frequency of $240MHz$. This indicates that the critical paths in the design are well distributed within the timing budget, that the pipeline structure and the worked out in code generation and that the implementation strategy is effective. Worst Hold Slack (WHS [19]) and Worst Pulse Width Slack (WPWS [20]) are not reported in this table and in the following ones because their values is always positive, for all the architectures implemented.

The power consumption that is estimated is equal to $0.783W$, where the higher contribution is given by the dynamin power (84%) due to all the activities of the DSPs, of the signals and of the clock distribution. The remaining contribution (16%) is related to the static power, and so the power used by the device when it is not performing any computation.

Regarding the resources utilization, the most critical component of the overall device is the FIR filter, as can be expected. In order to be implemented, the filter requires almost all the LUTs used for the implementation (5393, so the 96%) and all the 103 DSPs that are needed. It is possible to see that the number of DSPs is almost equal to the number of taps of the filter (105). The other components contribution in terms of resources is negligible with respect to the total value. Also regarding the memory usage, it is very marginal because a small memory is needed to save the peak values.

Thanks to this implementation, it is possible to obtain all the relevant data that can be used to make a comparison with the implementation that will be obtained in the next chapter in order to evaluate which kind of implementation is the most efficient in terms of performance, power and area.

Chapter 5

Implementation in Catapult

5.1 Catapult HLS

Siemens EDA Catapult HLS is a tool that allow to designers to produce HDL files, optimized RTL for both FPGA and ASIC platforms and analysis reports of the generated architecture starting from a high-level description of the algorithm in C, C++ or SystemC. This approach allows to move from the low level coding of the hardware to an higher level of abstraction, with a particular focus on the algorithm and on the architecture structure rather than on signal-level characteristics. One of immediate advantages of Catapult HLS is the significant reduction in time-to-market and design effort if compared to the traditional way of development. In fact this methodology is well known to reduce the code lines of about 70-80%, simplifying in this way debugging, prototyping and verification. Moreover, it allows design space exploration thanks to all the possible architectural optimizations that can be applied, in order to find the best trade-off in terms of performance, power and area.
Catapult HLS provides a basic flow, shown in Figure 5.1, that is easily accessible from the GUI and allows, by simply press on each task icon, to go on with the necessary steps to generate the RTL. In fact clicking on each icon a new Catapult command is run, allowing to advance to the next steps. The synthesis steps are:

- **Input files**: After selecting the correct working directory, clicking on this task it is possible to select all the source codes needed for the project. In particular, when a C++ file is added, all its included header files are implicitly included too. Moreover, it is important to not include all those files that must not be synthesized, like the testbench

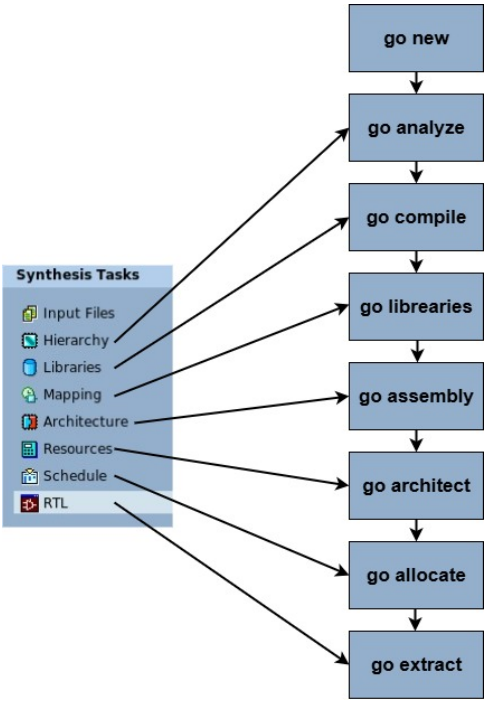

Figure 5.1: Catapult HLS flow

- **Hierarchy**: After that all files have been included, Catapult must analyze them in order to understand how the design is structured with its own hierarchy. This step is performed by the Hierarchy button, that corresponds to the "go analyze" command. At this point, all the blocks must be labeled as Top, Block or Inline: the Top block, that could be only one in the design, corresponds to the top-level block of the overall design; then all the sub-blocks that are called by the top-level must be defined as Block. The remaining blocks are labeled as Inline blocks. The hierarchy can be also assigned in the code lines thanks to the *hls_design* pragma. By specifying the type of block in this directive, it will be mirrored in the constraints of the design, but it is overridden by the options set in the GUI. Moreover, if no pragmas are applied to the class or to the function, it will be labeled as Inline by default. Moreover it is important to specify that until now no ports or memory have been inferred yet

- **Libraries**: This task corresponds to the "go compile" command, that compiles all the files of the design, creating the corresponding SIF database, implements the hierarchy, infers all the operations and directional ports and applies the first optimization to the architecture. In this task it is possible to select the technology library by selecting the proper RTL synthesis tool and device,min order to select a set of operators and memories with well known timing and area estimation. Otherwise, it is also possible to select any sub-blocks of previous solutions in order to perform a Bottom-Up implementation strategy, in order to reduce the compilation effort

- **Mapping**: It runs the "go libraries" command and in this step one or more clock frequencies and handshaking signals can be set. Here four kinds of signals, Start,

Ready, Done and Transaction flags, can be set based on the structure of the analyzed design. In a hierarchical design, like the one in this work, Start and Done flags can be set only in the top-level block, while all the hierachical blocks require the *ac_channel* data type on all the interfaces, that allows to model in an automatic way a handshake protocol between blocks and facilitates the data streaming thanks to its *read* and *write* methods. Moreover, enabling the Transaction Flag it is possible to add to the design some transaction signals to all process that go high for a clock cycle and a I/O transaction is completed. Moreover, in this panel it is also possible to set the working frequency of the design, with all its related information like duty cycle, offset and so on, like can be seen in Figure 5.2. It is also possible to set multiple clock frequency if the design is made up of different clock domains. Lastly, synchronous or asynchronous resets can be set and clock gating can be enabled

Figure 5.2: Mapping-Clock frequency

- **Architecture**: In this step, the blocks are loaded and the interconnections are checked, in order to be sure that everything is correct. After that, all the interfaces, memories and loops of the blocks can be optimized with different constraints. In the top-level tab, input and output delays, clustering and other advanced features can be set that, in the case of this work, have been left as default. Moreover, the interface options can be set, changing the type of resource used as port for the inputs and outputs of the device, with its own datasheet. Also all the interconnections between the hierarchical blocks can be modified, specifying also here the type of resource to be used and the FIFO Depth, in order to modify the latency of a specific interconnection. Going a step down, all the blocks can be analyzed and, for each one, there are three different sections: the Interface, in order to modify also here the input and output ports; the Constant Arrays, where all the memories can be optimized by selecting the resource type, the word width and the base address (if present); lastly, there is the core section, where a lot of optimizations can be applied. In this last section, first of all the effort level and the design goal of the block (latency or area) can be set, according to the specifications. Moreover, it is also possible to set the Percentage Sharing Allocation, so the percentage of clock period reserved for the logic needed to share components, where the default value is 20%. In this work, it is a fundamental option that allowed to obtain different

implementations that will be analyzed in Chapter 6. Then, the main loop, and any other internal loops of the block, can be optimized thanks to two techniques: loop unrolling and loop pipelining. The loop unrolling allows to generate multiple copies of the loop analyzed, according to the Iteration Count of the loop itself, allowing to improve the throughput of the architecture due to the fact that all the operations performed by the loop are executed in one clock cycle. It costs in terms of area and power consumptions, obviously. It is also possible to not fully unroll the loop, but partially by checking the proper field and selecting the wanted unroll factor. The loop pipelining allows to reduce the number of clock cycles between the beginning of a loop and the beginning of the following one, called Initiation Interval. It allows to achieve higher performance if specifications requires it.
Moreover, there is also the button "Loops can be merged", in order to allow to Catapult to automatically merge different loops and to reduce the resources employed.

- **Resources**: It runs the "go architect" command and in this task it is possible to see the resources that Catapult has assigned to each operator or it is possible to constraint them too, based on what are the specifications about area, power consumption and time delay. Moreover, it is also possible to see the size of the objects expressed as <number_of_words>x<word_width>

- **Schedule**: After that all the resources have been allocated, the design can be scheduled. In fact, in this step Catapult applies all the timing constraints to the architecture and a Gantt Chart is generated, which graphs the number of control steps in each loop and the sequence of the operations scheduled within the control steps. Each control step is the equivalent to a state in the Finite State Machine of the architecture, and it corresponds to a clock cycle of the execution. Selecting an operator in the window it is possible to see the data dependencies between the selected resource and the other ones, highlighted by an arrow. The only problem is that it is not possible to see how pipeline has been applied

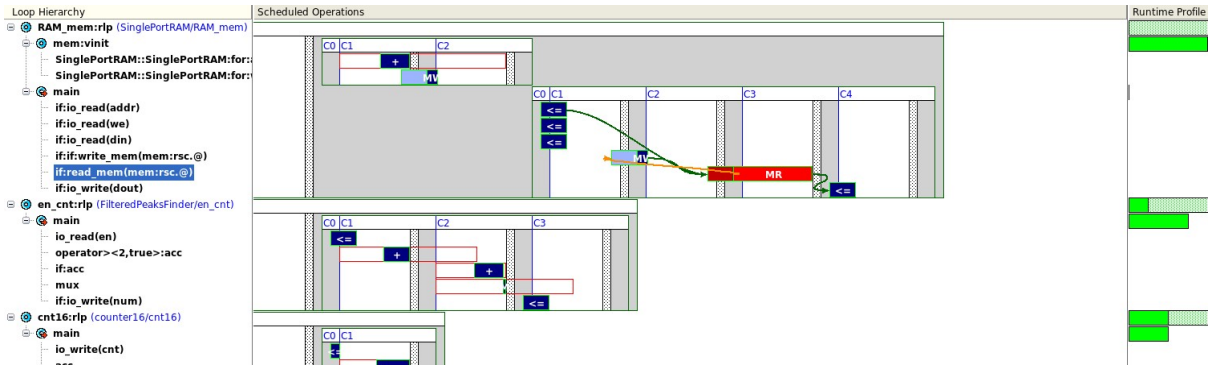

Figure 5.3: Scheduling view

- **RTL**: At the end, there is the RTL tab, that corresponds to the "go extract"

command, and that generates all the necessary output files. In fact in this final step, all the HDL codes (VHDL or Verilog), the RTL and the Datapath schematics, the critical paths of the design (with the starting and ending points) and all the necessary reports (cycles, Bill of Materials and so on) that allow to analyze the produced architecture in order to better optimize and finalize it.

All these information are taken from the Catapult User Guide [10]. All the results of the generated solution are then collected in a table, which shows in parallel all the previous results of all the different implementations generated in order to make a direct comparison in terms of throughput, latency, slack and area. In this way it is possible to explore different solution and to select the best one depending on the specification of the project.

Solution	Latency Cyc...	Latency Time	Throughput Cycles	Throughput Time	Slack	Total Area
ac_fir_const_coeffs_wrapper<ac_fix... (extract)	16	40.00	17	42.50	0.07	6566.91
ac_fir_const_coeffs_wrapper<ac_fix... (extract)	61	122.00	62	124.00	0.43	9607.16
ac_fir_const_coeffs_wrapper<a... (extract)	**3**	**6.00**	**1**	**2.00**	**0.00**	**18074.44**

Figure 5.4: Example of Table view

5.2 C++ Architecture & Optimization Pragmas

The use of Catapult HLS introduces a different design perspective with respect to the traditional one of the HDL Coder analyzed in the previous chapter. The beginning of the code generation and synthesis process in Catapult HLS is given by the translation from the high-level model to an hardware implementation that is described by the algorithm written in C++ language. In this case, the HLL description and tool allow greater flexibility with respect to the traditional model based design, but the problem is that the C++ description must follow an hardware-oriented structure in order to not only define the behavior of the algorithm, but also to be suitable for the tool to be translated into HDL code and synthesized. It means that the functions or the classes must adopt different coding practices that are related to hardware semantics such as fixed-point data structure, limited iterations rather than unlimited ones and statically allocated data structure, and not the dynamic memory allocation that is often preferred in a software application.

For this reason, the structure of the C++ code describing the algorithm is a key factor: it is important in the direct impact that it has in the algorithmic mapping to RTL and thus determines the performance of the resulting FPGA implementation in terms of hardware resource utilization, latency, throughput and power consumption. For this reason, in this work different kind of implementations have been realized in order to show how the coding style impacts on the implementation results given by the synthesis tool. Moreover, in order to bridge the gap between hardware implementation and higher-level algorithm, Catapult HLS provides different directives and, in particular, pragmas that can be added

to the source codes in order to better guide the tool in the compilation and optimization steps. These directives obviously have an impact also during the scheduling and allocation processes, so in this way several architectural implementations can be investigated without altering the algorithm's functional correctness. In fact, in this way it is possible to explore different hardware implementations in order to find the best trade-off between performance and resources utilization. For example, loop pipelining and unrolling pragmas facilitate parallelism exploitation in order to increase the throughput and optimize the latency or there are some interface pragmas that allow to define communication protocol between the generated block and the external world, in order to make the integration within larger FPGA systems easier.

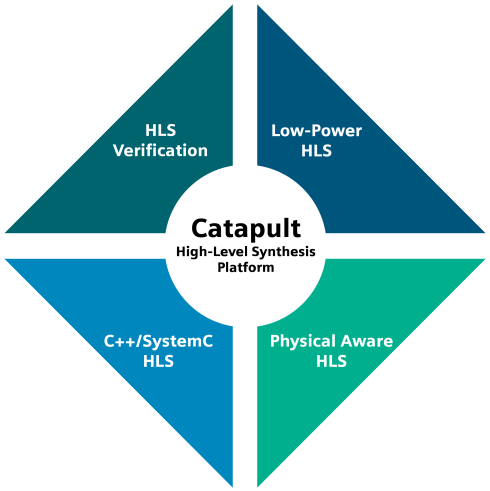

Figure 5.5: Catapult HLS

In this chapter the C++ files that are used to implement the algorithm will be described, with all the optimization pragmas and directives used to modify the resulting RTL and with the differences between the implementations that allow to optimize the architecture. As for the HDL Coder implantation, the resulting RTLs will be simulated in QuestaSim and synthesized in Vivado in order to provide a comparison of the results. All the useful tips regarding the C++ coding style are taken from the HLS Bluebook [11].

First of all, by following the structure of the model-based design described in Simulink HDL Coder, a bottom-up implementation strategy has been followed and so starting from the description of each component and then connecting all of them into a top-level design used as a wrapper (as for the FPGA block) with the same input and outputs. In this way the architecture is structured and aligned with the one that has been described before, so it is possible to facilitate the integration of this block in the overall chain. Internally, each block described in C++ language reproduces the same operations performed by the FPGA block like filtering, peak detection, bias removal and so on. Moreover, thanks to pragmas and GUI directives, it has been possible to better optimize each block interdependently in order to balance throughput, performance and resources.

In this chapter, two main different implementations of the FPGA block will be described

in detail:

- The first one is an initial implementation where a simple translation of the Simulink FPGA block into C++ code lines has been performed. Here the algorithm has been optimized by using some pragmas, in this way the focus is mainly on the importance of the directives in an HLS description
- The second one is the same implementation, but it introduces some differences in the coding style of some of the sub-blocks of the FPGA. These changes aim to highligth how different ways of describing the same algorithm in C++ can lead to different implementations in terms of RTL generated, performance, scheduling, allocation and so on.

This is done in order to show how a different coding style can impact on the generation of the RTL and on the implementation's results. After that, in Chapter 6, the results of other implementations will be analyzed: in these cases, the behavior and the coding style remain the same for both implementations, but a different working frequency and a different percentage sharing allocation of some components have been applied in order to explore other solutions. Moreover, they allow to evaluate how Catapult HLS handles the project under different constraints and to compare also the results with the already described architectures.

5.2.1 Initial implementation

The first step is to translate all the sub-blocks of the model-based design into C++ language. In particular, in this work all the sub-blocks have been translated into a set of header files, in which there is the definition and the description of a C++ class that is associated to each component, in order to improve modularity and reusability of each block in the hardware implementation, like a "component" in VHDL or a "module" in Verilog. In fact in this way it is easier to understand, test and modify each component without the needing of modifying all the other components too. All these files mirrors the same structure that the previous model has, in order to obtain a more realistic comparison.

First of all, the file named *param_config.h* must be described: it is a key header file of the entire project because it contains the global parameters (like the number of taps of the filter or the memory size) and all the data types of the signals used to connect all the blocks in the architecture. Thanks to this file, it is possible to modify one by one all the parameters of the architecture, such as the data width, the number of integer and fractional bits and so on, without needing to go through all the files to search a particular value to change. In this way the code remains uniform and also the compilation is faster.

```
1    //coefficients attributes
2    #define FIR_COEFFS_WIDTH  16   //number of bits of the coefficients
3    #define FIR_COEFFS_INT    0    //number of integer bits of the coefficients
4    #define FIR_COEFFS_SIGNED true //signed coefficients
5    [...]
```

```
6     typedef ac_fixed<FIR_COEFFS_WIDTH,FIR_COEFFS_INT,FIR_COEFFS_SIGNED,AC_RND,AC_SAT>
      fir_coeffs_t; //definition of coefficients data type
```

A code snippet is shown above. In this case, there is the definition of the data types of the coefficients of the filter. It is possible to observe that it mirrors the definition of the coefficients' data type of the Simulink model, so signed data with all the bits dedicated to the fractional part in order to increase the accuracy of the coefficient. All the other data types have been described using the same format, so this file enhances readability and maintainability too, simplifying the design exploration. After the definition of the parameters that play a fundamental role in the project, all the other header files have been realized. Starting with the low-pass FIR filter, it has been described for simplicity into two different files:

- The first one, named $fir\_coeffs.h$, which simply contains a vector with all the 105 coefficients extracted from the Simulink model
- A second file called $fir\_lpf.h$ that is the header file that contains the implementation of the filter itself.

In this way it is possible to change the frequency response of the filter without changing its behavior, or viceversa. The second file contains the class in which the functionality of the filter is described. There is a shift register, long as the number of taps of the filter, that is initialized to zero and that will contain all the samples. But the heart of this code is the for loop, labeled as FIR, in which the multiply and accumulate operation is performed at each iteration, and so the sum of all the dot products between the samples contained in the registers and the coefficient of the filter. So it can be described by this formula:

$$acc = \sum_{i=0}^{FIR\_TAPS-1} shiftREG[i] \cdot FIR\_COEFFS[i] \cdot LPF\_gain \tag{5.1}$$

where also the gain of the filter is considered in order to be consistent with the Simulink reference model. This expression maps directly onto hardware since the tool recognizes the structure as a canonical filter and it can perform many optimizations according to the directives provided. Below it is possible to find the code snippet that implements the previous expression:

```
1     FIR: for(int i=(FIR_TAPS-1);i>=0;i--){
2               fir_in_t tmpTap=(i==0) ? input : shiftREG[i-1];
3               acc+=tmpTap*FIR_COEFFS[i]*LPF_gain;
4               shiftREG[i]=tmpTap;
5           }
```

The loop can be fully unrolled and pipelined in order to improve the latency and the throughput of the components, otherwise each output will take too much time to be produced, due to the high complexity of filter (105 taps). In fact these two directives will be imposed thanks to the GUI, but it is possible to apply them directly in the code by adding these two code lines in the proper row:

```
1     #pragma hls_unroll //to fully unroll the loop
2     #pragma hls_pipeline_init_interval 1 //to fully pipeline the loop
```

Both directives have been applied to the loop in order to obtain a fully parallelized hardware like the one obtained from HDL Coder. All the data types of the filter and the rounding operations performed to the sample are identical to the ones performed in Simulink to obtain a project that mirrors the characteristic of the previous one. Moreover, the loop of the filter is the part that will be modified in the following implementation in order to better optimize the architecture, being the most critical component of the overall project.

After the filter, there is some logic that has been encapsulated in the file called *bias_threshold.h*. In this block the bias removal and the comparison with the fixed threshold are performed, thanks to which it is possible to detect a new peak. The first operation is performed by the subtraction between the output of the filter and the bias value, scaled by the filter gain factor.

```
1     sub_fir_max = in1_sub-(LPF_gain*bias); //sub operation between filter output and
      bias
```

By performing this subtraction, the average level of the signal is restored to 0. After that, the research of a new peak is performed by subtracting the previous value to the threshold, so in this way it is possible to understand when a new peak has arrived or not.

```
1     sub_thres_sub=Thres-sub_fir_max; //sub between subtraction and threshold
2
3   if (sub_thres_sub>=0){ //equivalent of sign block
4     sub_sign=0;
5   } else {
6     sub_sign=-1;
7   }
```

This value will be forwarded to the other components and used as control signal in order to enable and reset the following blocks at the proper instant. Then there are the blocks that allow to perform the peak detection, that are the max_finder, the counters and the S&Hs. First of all there is the max_finder block which is used in order to find and to maintain the value of the peak value in each time window, until it is reset and a new window of analysis starts again. The principle is the same as before: until the reset value is not asserted, each new sample at the input is compared to the previous one, which is stored in a register. When the new sample is larger, it becomes

the new peak encountered, otherwise the previous value is maintained. When the reset is high, the value stored in the register is cleared and the research for a new peak starts again.

```
    if (reset == 1){
        max_val=0.0; //reset of the maximum value
  } else  {
    if (din > max_val){  //check if the new input is larger than the previous one
      max_val=din;
    }
        y.write(max_val);
        }
```

After that, there is the block that performs the difference between the output of the max_finder and a delayed version of it, in order to understand when the peak is growing or not. By detecting a falling edge, that correspond to the end of the peak zone described in the previous chapter, it is possible to send the enable to the S&Hs in order to make them save the previous sample, which correspond to the peak amplitude and to its corresponding time arrival. The former is taken from the delayed output of the max_finder, while the latter is taken from the free-running 16-bit counter described in the file named *cnt*16.*h*. It increments continuously its output, providing the time for the system. A shift register has been included at the output of this counter, before the S&H, in order to synchronize the peak amplitude value with the proper time arrival. In parallel there is another counter, *cnt_en.h*, which reproduces the behavior of the Filtered Peaks Finder block in Simulink. When the enable is asserted, its output is incremented and it will be used as address for the output memory, in order to store the detected peak amplitude in a sequential way.
After that, there are the two S&Hs that are described by the same class, but the only difference is the data type they use: in fact the peak values are represented by fixed-point numbers, in particular with 15 bits dedicated to the fractional part; meanwhile, the other S&H manages only 16-bit long integer words. They both have the role to hold the value in input when the enable is asserted, for this reason their description is identical. A code snippet describing the S&H is shown below.

```
    if(sh_in1.available(1)){
  in_chan=sh_in1.read();

  if(sh_en1.available(1)){
    en_chan=sh_en1.read();

            if(en_chan==0.0 && en_prev!=0.0){
        sh_peak.write(in_chan);
    }
    en_prev=en_chan;
  }
    }
```

S&H is another important component for this analysis because for both implementations only one header file has been written in order to describe the two S&Hs employed in the architecture. In fact it has been possible to do that thanks to the template class in

C++, that allows to reduce the number of files that must be written, simplifying so the compilation. In fact it is possible to simply pass the needed data type with together to the call of the class in the wrapper in order to obtain the two different S&Hs.

```
    SampleAndHold<s_h_data_t> peak_values;
    SampleAndHold<s_h_cnt_t> time_values;
```

Then there is a simple RAM memory that, as for the Simulink model, collects the peak amplitude values and they are stored chronologically thanks to the address given by the counter described before. Lastly, everything is wrapped into the last file named *FPGA_wrap.h* in which the FPGA interface is defines and every C++ class describing each component is called. Each component is connected to the others in the same way as the Simulink model thanks to all the channels that allow and facilitate the data streaming between component, creating the same handshake protocol of the HDL Coder generated project.

At the end, in order to verify the behavior of the system, a C++ testbench has been prepared described in the file called *tb.cpp*. It is important to highlight that the same input waveform, contained in the file named *input_file.txt*, has been used to feed the FPGA block in order to better reproduce the same simulation environment of the Simulink model and to obtain a direct comparison for the results produced. At the beginning the testbench simply opens this file and write its content into a vector to be passed then to the FPGA block, after converting it into the data type of the block.

```
    //saving input data in the vector
    while(infile>>in_value){
        fir_in_t converted_val = fir_in_t(in_value);
  input_vector.push_back(converted_val);
    }
```

After that, there is the main loop where all the input samples are processed by the FPGA block, mimicking the clocked behavior of the hardware. The outputs are then saved into another file, named *output_file.txt*, that contains a chronological list of all the detected peaks with the related information, and so the peak amplitude and the detection instant.

```
    for(int i=0; i<input_vector.size(); i++){
        fir_in_t input = input_vector[i];
        in_chan.write(input);

        dut.FPGA(in_chan,out1_chan,out2_chan);

        if (out1_chan.available(1) && out2_chan.available(1)){
            s_h_data_t output_peak=out1_chan.read(); //reading peaks
            s_h_cnt_t output_time=out2_chan.read(); //reading time
            outfile<<"Peaks: "<<output_peak<<" Time: "<<output_time<<endl; //saving
    results in the output file
        }//if
    }//for
```

In this loop, the FPGA block class is called once per iteration, by replicating the hardware behavior at each clock cycle. In this way, each sample can be processed and it is possible to obtain the same outputs of the previous model, after checking that both outputs are available at the outputs of the wrapper. After that all the files are ready, the Catapult flow to generate the synthesizable RTL starting from C++ classes can be started:

1. **Input Files**: First of all, the files are imported into the new project and now all the different steps (described before) provided by the tool can be executed in sequence

2. **Hierarchy**: Here the top-level has been selected, that is obviously the FPGA wrapper, while all the other components have been labeled as "Block". In this way the tool is able to synthesize the project with the correct structure and with the proper interface

3. **Libraries**: The target FPGA (Artix7 xc7a200tfbg484-2) and the libraries with the standard cells to be used to map the arithmetic operations into real hardware components have been selected. Moreover, the "Xilinx new RAM models" library have been used too in order to allow to the tool to provide models for the memory, such as Block RAM, so in this way it can be properly mapped efficiently into the available hardware resources

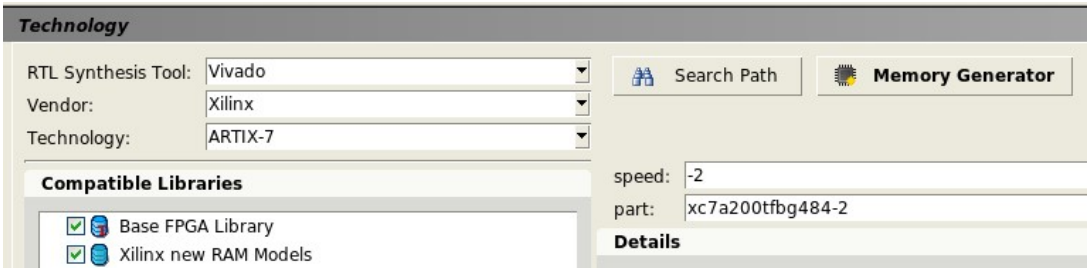

Figure 5.6: Catapult HLS Flow-Libraries

4. **Mapping**: In this step, the clock frequency of the overall system has been selected. In particular, it has been set equal to $240MHz$ with a duty cycle equal to 50%, like the previous model, and also the reset has been set as synchronous

5. **Architecture**: In this step all the architectural optimizations have been set. In fact, in order to obtain a fully parallelized architecture, the FIR loop has been fully unrolled and pipelined, in this way it is possible to improve the throughput and the latency of the component. Moreover, all the other components have been fully pipelined too, in order to split all the critical paths that could represent a problem for the timing constraints. Furthermore, it is possible to assign a FIFO Depth to all the interconnections, in order to synchronize the outputs

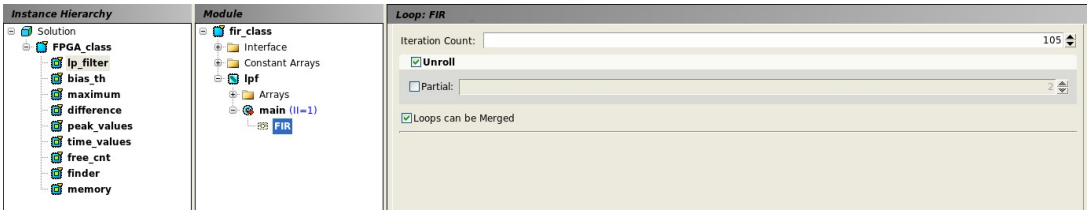

Figure 5.7: Catapult HLS Flow-Architecture

6. **Resources**: At this step, the best components that the tool have chosen for each block to respect the constraints are shown

7. **Schedule**: Here Catapult HLS generates the Gantt Chart of all the loops of the components employed in the architecture, scheduling them cycle-by-cycle. In particular in Figure 5.8 the scheduling of almost all the sums and multiplications performed by the filter can be observed. As it shown, it is possible to see that for the first 7 control steps, most of multiplications are performed corresponding to the product between the coefficients and the input samples. They are performed in parallel, showing how Catapult has assigned these operations to different operators, corresponding to different DSPs in real life. Then all the sums are executed, performing the accumulation step that has to wait until almost all the products are calculated. Overall, only the filter takes 14 clock cycles to compute the results. But this chart shows also how Catapult could pipeline the loop in order to start a new computation while the previous one is still in execution.

Figure 5.8: Catapult HLS Flow-Schedule

8. **RTL**: At the end the RTL of the architecture is produced, shown in Figure 5.9. Each C++ class has been generated as a standalone block, with all the interconnections that guarantee the data streaming and the handshake protocol (ready and valid

signals). The structure is identical to the HDL Coder one, and it reflects all the optimizations and all the directives set in the previous steps. Now it is possible to simulate it and implement it on the target FPGA.

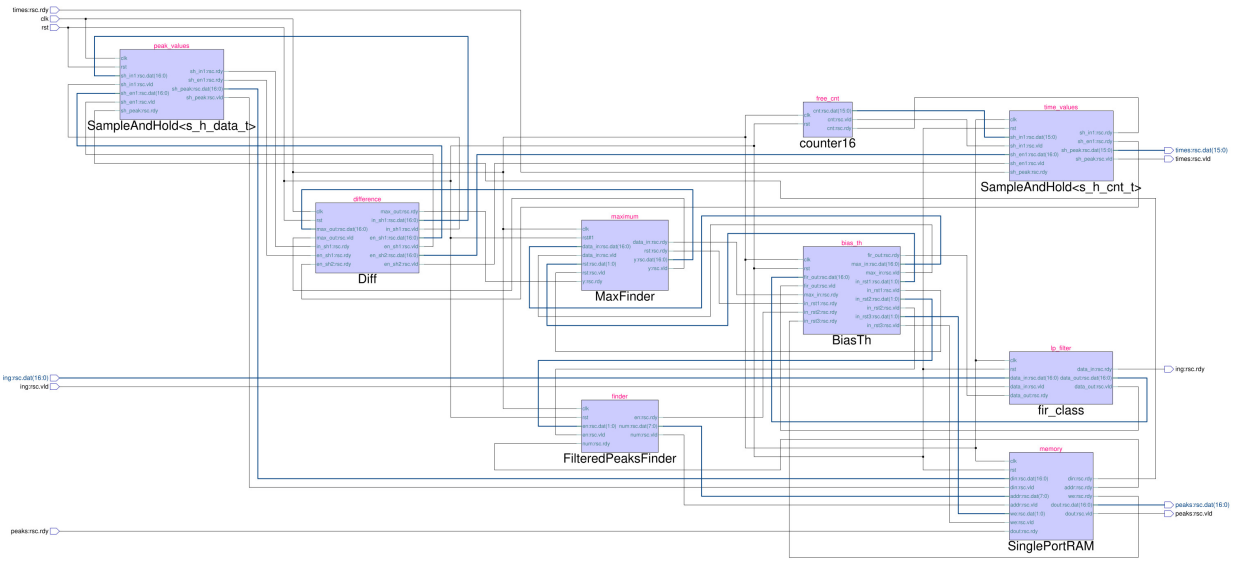

Figure 5.9: Catapult HLS Flow-RTL

In order to speed up all the process and to not repeat all these steps each time a file is modified, a .tcl script has been written in order to automatize the flow execution. In this script, all the directives have been inserted, in order to allow to Catapult to apply them to the circuit. Moreover, it is also very useful because it allows the exploration of different solutions, in order to better optimize the architecture. It has been very useful for all the implementations that will be described in the following chapter because by changing only one directive, a new implementation is obtained.
At this point, the tool has generated all the HDL codes and all the makefiles that are useful to run automatically the QuestaSim simulation and the synthesis on Vivado directly from the GUI of the tool.

5.2.2 Optimized implementation

The second design flow is an optimized form of the FPGA block with particular focus on the FIR filtering stage. While the initial implementation is functionally correct and it produces correct results, its structure did not take into account high-level HLS optimizations. Especially, the coding style of the FIR loop has a decisive impact on the RTL code generated by Catapult. In fact its behavior was implemented by performing the shift operation and the multiply and accumulate operations in the same loop. Easy to understand but resulting in high latency and with a hardware utilization that is very huge, as will be discussed in the following sections.

In the optimized version, the FIR loop was carefully reordered to expose the two operations into two different loops, in order to better optimize the two operations separately. The loop pipelining and loop unrolling techniques are applied here too, which allow multiple MAC operations to be scheduled in the same cycle or in adjacent cycles with minimum initiation interval. The outcome is that Catapult, when mapping the design to RTL, synthesizes the architecture by using less DSPs, but with better performance. Here there is the code snippet of how the C++ source code of the FIR filter has been changed.

```
1     SHIFT: for(int i=(FIR_TAPS-1);i>0;i--){
2           shiftREG[i]=shiftREG[i-1];
3       }
4           shiftREG[0]=input;
5
6     MAC: for(int i=(FIR_TAPS-1);i>=0;i--){
7           acc+=shiftREG[i] * ((fir_out_t)(LPF_gain*FIR_COEFFS[i]));
8           }
```

By splitting the two operations, the tool is able to schedule the two operations in a better way, reducing the possible data dependencies caused by the structure of the previous loop, improving the pipelining and the parallelism. Moreover, the coefficients are pre-scaled (LPF_gain * FIR_COEFFS[i]) as well, which allows the tool to fold constants and simplify the datapath further. As a result, the optimized coding style produces faster RTL with lower latency and lower DSPs utilization, demonstrating how coding style directly affects the quality of HLS synthesis results, as can be observed in the following sections.
By modifying the .tcl script in order to adapt it to the new form of the implementation, a new Catapult HLS flow has been executed with the same directives described before. The main differences that can be noticed directly in the flow are:

- In the Architecture view, the FIR filter is divided into two separate loops with different numbers of iterations: the MAC loop (105 iterations) and the SHIFT loop (104 iterations). By separating these functions, Catapult can more effectively optimize them: the MAC loop is completely unrolled and can be mapped onto a parallel adder tree to optimize throughput, and the SHIFT loop, unrolled as well, is effectively implemented as a register chain with no redundant control overhead

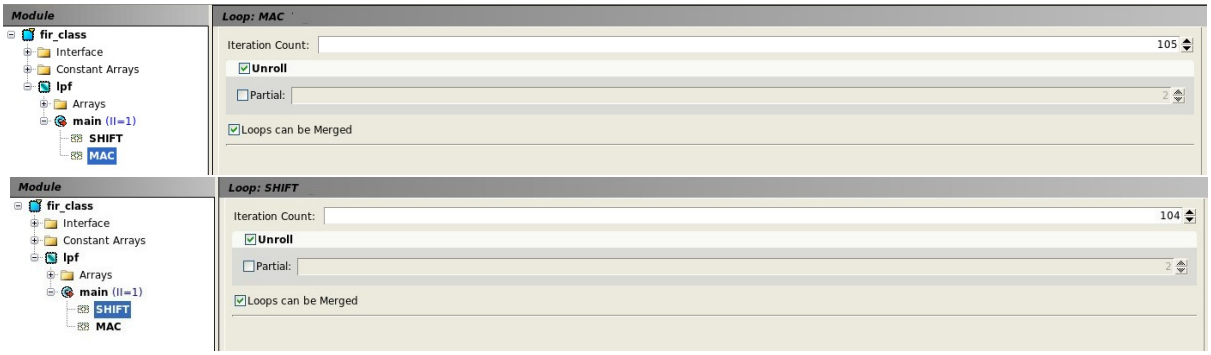

Figure 5.10: MAC and SHIFT loops

- In the Scheduling view, the split loops allow Catapult to schedule better the shift operations in a compact block at the beginning and then fully pipeline the multiplications and additions. The MAC loop has more concurrency with the multipliers and adders executing simultaneously and an uncomplicated adder tree structure. This reduces the cycles per sample, reduces the overall latency and offers an initiation interval of 1.

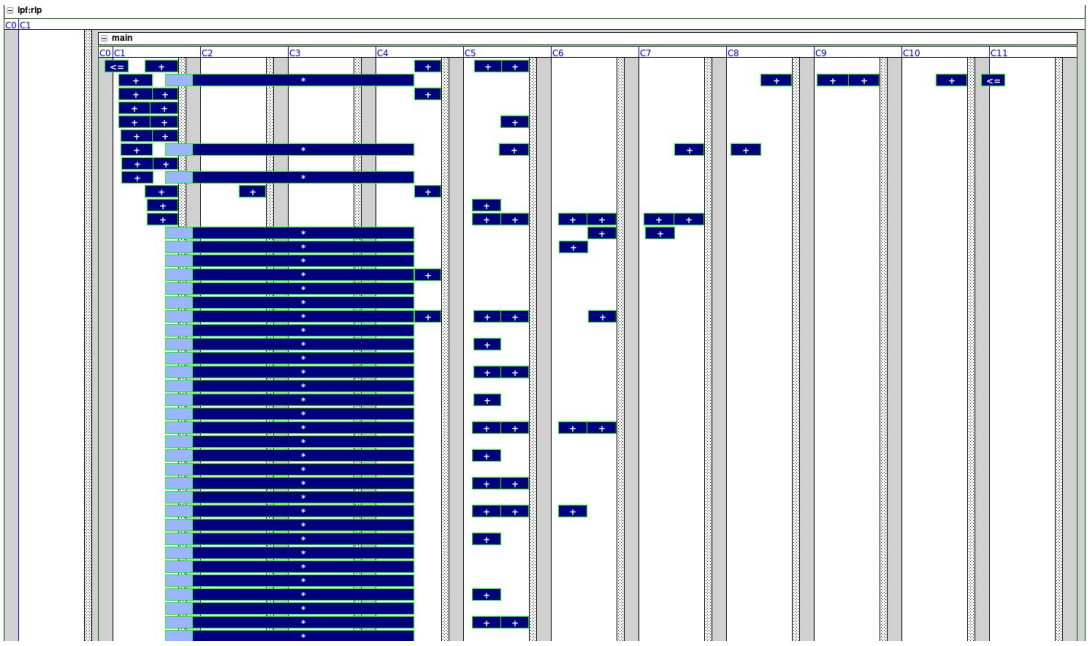

Figure 5.11: Optimized implementation-Scheduling

Regarding the RTL produced, as can be observed in Figure 5.12, it is not different from the previous one from a structural and graphical point of view (the number of blocks is always the same, same interconnections and so on), but the main differences can be observed in the synthesis and the implementation performed by Vivado.

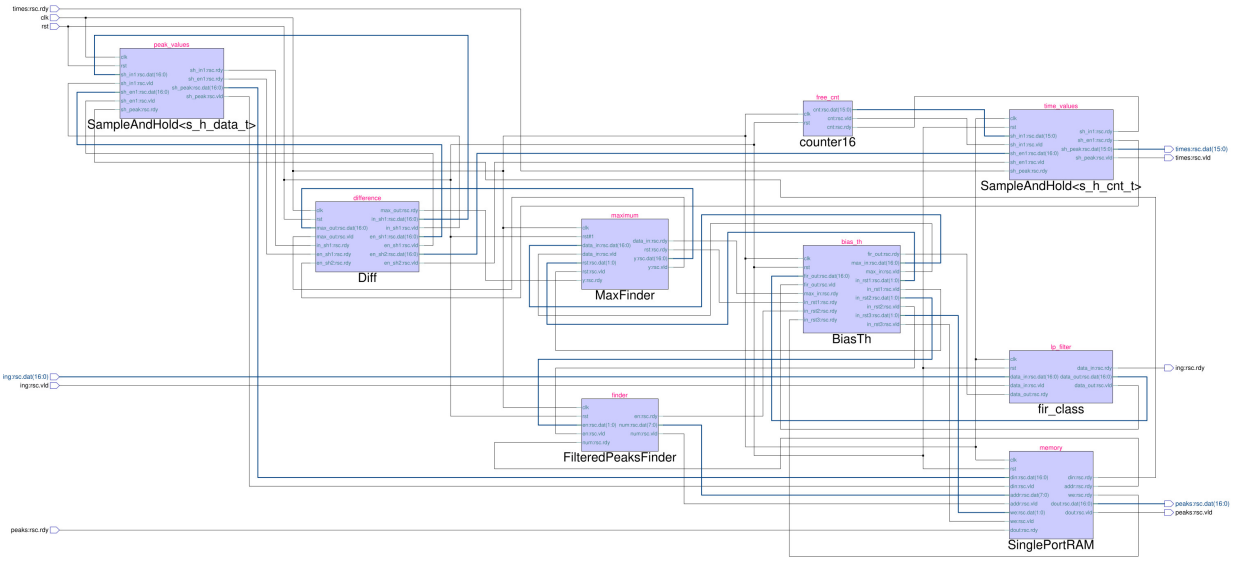

Figure 5.12: Optimized implementation-RTL

5.3 QuestaSim Simulation

As done for the HDL Coder model, the two architectures generated by Catapult HLS, the initial one and the optimized one, must be validated through a simulation performed in QuestaSim. The tool automatically produces the makefiles that allow to directly open the simulator, compile all the files and run the simulation. The two tables below summarize the obtained results of the simulations, the first one of the initial architecture, while the second one of the optimized implementation.

	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}
peaks-DUT [mV]	16.27	15.14	12.39	12.27	18.04	13.82	14.01	19.32	15.78	16.69
peaks-GOLDEN [mV]	16.27	15.14	12.39	12.27	18.04	13.82	14.01	19.32	15.78	16.69
times-DUT	1255	9200	14800	19390	22356	28867	31659	38081	40485	49564
times-GOLDEN	1255	9200	14800	19390	22356	28867	31659	38081	40485	49564

Table 5.1: Initial implementation-simulation results

	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}
peaks-DUT [mV]	16.27	15.14	12.39	12.27	18.04	13.82	14.01	19.32	15.78	16.69
peaks-GOLDEN [mV]	16.27	15.14	12.39	12.27	18.04	13.82	14.01	19.32	15.78	16.69
times-DUT	1255	9200	14800	19390	22356	28867	31659	38081	40485	49564
times-GOLDEN	1255	9200	14800	19390	22356	28867	31659	38081	40485	49564

Table 5.2: Optimized implementation-simulation results

The rows labeled as -DUT are the results obtained from the implementations, while the rows labeled as -GOLDEN are the expected results evaluated starting from the C++ algorithm. As can be observed, the results obtained from the architectures for both tables correspond to the ideal ones. Moreover, it is possible to see that the two tables are identical to the one of the previous model (Table 4.1), so there are no differences in terms of peak amplitudes and time values. Both architectures respect the expected behavior of the model by feeding them with the same input waveform of the HDL Coder model. The only difference that has been encountered is in terms of latency: for the HDL Coder the latency in equal to 15 (in terms of clock cycles), for the not optimized implementation it is equal to 21 (40% more if compared to the HDL Coder one) while for the optimized implementation it is equal to 18 (so 20% more).

5.4 Vivado Synthesis & Implementation

As well as for QuestaSim, Catapult HLS is able to generate also the makefiles that allow to create a Vivado project with all the needed files (source files and constraints files). Obviously, before running the synthesis and the implementation, the same options as before have been set regarding the strategies to be used for both the two processes.

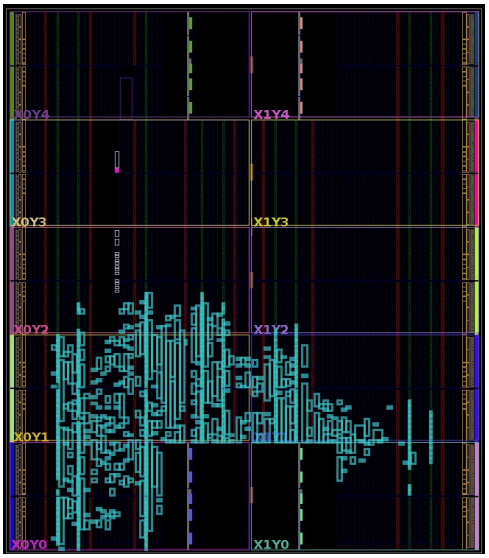

Figure 5.13: Placement view of the initial implementation on the FPGA

First of all, starting with the not optimized implementation, it is already visible from Figure 5.13 that the placement of the hardware is bigger than the previous model (Figure 4.8), occupying all the bottom part of the device. It suggests that the implementation exploits more resources with respect to the first one. In fact, by looking at the table below that summarizes the key aspects of the architecture, the first data that stands out is the DSPs employed that are 247, so one third of the total number. This number is particularly relevant for the implementation, because it has a direct impact on the power consumption (about 30% higher with respect to before) and on the area occupied. However, the number of LUTs employed has been reduced drastically.

Timing Result	
Worst Negative Slack	0.014 ns
Power Analysis	
Total On-Chip Power	1.027 W
Resource Utilization	
Slice LUTs	1570 / 134600 (1.2%)
Slice Registers	5974 / 269200 (2.2%)
Slices	2176 / 33650 (6.5%)
Block RAM	0.5 / 365 (0.1%)
DSP Slices	247 / 740 (33.4%)

Table 5.3: Catapult HLS-FPGA initial implementation results in Vivado

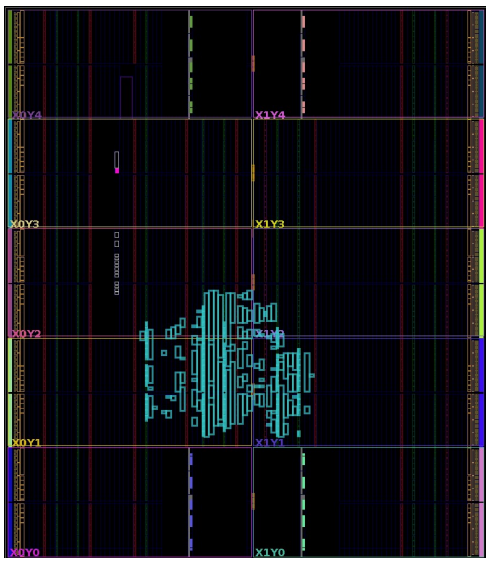

Figure 5.14: Placement view of the optimized implementation on the FPGA

Meanwhile, considering the optimized implementation results, the result is very different. The performance are comparable to the HDL Coder model, but now even the number of DSPs has been reduced (91). In fact, the implemented architecture occupies a

lower fraction of the device and the power consumption is halved with respect to the not optimized implementation. So, by splitting the FIR loop, it has been possible to reach the same performance of the HDL Coder model with a lower number of DSPs employed. The only drawback is the extra clock cycle of latency in this implementation.

Timing Result	
Worst Negative Slack	0.145 ns
Power Analysis	
Total On-Chip Power	0.535 W
Resource Utilization	
Slice LUTs	1758 / 134600 (1.3%)
Slice Registers	3723 / 269200 (1.38%)
Slices	1247 / 33650 (3.7%)
Block RAM	0.5 / 365 (0.1%)
DSP Slices	91 / 740 (12.3%)

Table 5.4: Catapult HLS-FPGA optimized implementation results in Vivado

In conclusion, the achieved results show the effectiveness of the proposed FPGA implementation, with emphasis on the filter optimized form. By restructuring the loop and improving the coding style, the generated RTL proved to be significantly more efficient, both in terms of scheduling and hardware resource utilization. These results demonstrate how some modifications to the high-level description can have a huge impact upon the resulting synthesized hardware. In the next chapter, these results will be extended even further by comparing them with other different implementations so that more evaluations in terms of performance, resource utilization and efficiency can be made.

Chapter 6

Results and Reports

6.1 Comparative Analysis of Implementations

In this chapter a comparative analysis will be done regarding all the different implementations that have been realized in through this work. Several implementations, as well as those shown in the previous chapters, have been realized:

- First of all, with together to the three implementations analyzed before, other implementations have been realized by changing a parameter in Catapult HLS that is the Percentage Sharing Allocation of the filter, i.e. the percentage of the clock period that is reserved for the logic that is needed to share the components [10]. In particular, the filter has been chosen to modify this parameter because it is the block with higher latency. In this way, by sharing a lower number of components, it is possible to see what is the impact that this directive has on the performance of the device

- Then more implementations in Catapult have been realized (for both architectures) by changing the Percentage Sharing Allocation of the filter, but also increasing the frequency to $260MHz$ to see if Catapult HLS, on the contrary of HDL Coder, is able to generate a suitable hardware that is able to increase the clock frequency

- Lastly, by changing the Percentage Sharing Allocation of another block in the implementation that have reached the lower latency, in order to improve it further.

Moreover, at the end of the chapter, the time employed to obtain all these results will be analyzed, starting from the environment setup and the learning of all the tools, and considering all the time needed to realize algorithm in MATLAB and Simulink and the time needed to implement, correct and verify all the architecture realized on HDL Coder and on Catapult HLS.

6.1.1 Implementations at $240MHz$

The table below shows all the results obtained in Vivado of all the different implementations that have been realized with a target frequency of $240MHz$. The first column corresponds to the results obtained from the implementation of the HDL Coder model, that is used as a reference for the comparison with the Catapult ones. From the second to the fourth columns are dedicated to all the results that are related to the initial implementation in Catapult HLS (not optimized version of the FIR loop), labeled as "NOT opt". What changes in the settings among these three column is the Percentage Sharing Allocation (20%, 0% and -20%) in order to explore different architecture and how this setting impacts on performance, power consumption and resources utilization. At the end, the last three columns show the results corresponding to the optimized version of the design in Catapult HLS (optimized version of the FIR loop), labeled as "opt". All these implementations are analyzed with the same percentages in order to allow a comparison one-to-one with the not optimized versions.
In terms of timing, it can be seen that the HDL Coder implementation meets the timing constraints at $240MHz$, achieving a positive WNS of $0.137ns$, so there is a margin that allows to slightly increase the frequency without any violations. Moreover, it is also able to achieve the best result in terms of latency (15 clock cycles) among all the other designs. Regarding the not optimized implementations, only the designs with Percentage Sharing Allocation equal to 20% and 0% meet timing constraints, while the last one fails at this clock frequency ($-0.082ns$). However, the two positive margins suggests that the designs would work close to timing failure and it could be a problem for tighter constraints. Moreover, these implementations have the highest values in terms of latency (21 at 20% and 0%, 19 for the last one), highlighting that the performances are worse with respect to the HDL Coder ones.

	HDL Coder	**NOT opt (%=20)**	**NOT opt (%=0)**	**NOT opt (%=-20)**	**opt (%=20)**	**opt (%=0)**	**opt (%=-20)**
Timing Result							
Latency (clk cycles)	15	21	21	19	18	18	16
WNS (ns)	0.137	0.014	0.051	-0.082	0.145	0.041	0.019
Power Analysis							
Power (W)	0.783	1.027	1.002	1.024	0.535	0.545	0.564
Resource Utilization							
Slice LUTs	5612	1570	1690	1688	1758	1873	1799
Slice Registers	6036	5974	5622	5168	3723	3961	3545
Slices	2085	2176	1815	1874	1247	1234	1253
BRAM	0.5	0.5	0.5	0.5	0.5	0.5	0.5
DSPs	103	247	247	247	91	91	91

Table 6.1: Comparison at $240MHz$ between HDL Coder and Catapult HLS (both implementation with different Percentage Sharing Allocation)

The last three columns, in contrast, demonstrate how coding style improve the characteristics of the design, In fact all the architectures are able to meet the timing constraints, especially the one at 20% of allocation sharing that achieves a similar WNS of

the first model. Also latency is improved in fact for percentages of 20% and 0% the latency is equal to 18 clock cycles, so 20% higher with respect to the HDL Coder architecture. While the last implementation achieves a latency equal to 16 clock cycles, so very similar to the reference one.
In terms of power consumptions, it is possible to observe an increase of about 30% for all the not optimized implementations with respect to the reference model. On the contrary, for the optimized designs, the power consumptions are lower of about 30%, that is a very good result that anticipate that also the resources utilization is improved. Regarding the resources, the main differences can be noticed in terms of LUTs, registers and DSPs employed:

- For the LUTs, it is possible to see that all the implementations in Catapult HLS, both initial and optimized versions, have a huge saving of units employed passing from 5612 of the HDL Coder project to the 1600/1900 used by the Catapult designs, so about 70% less than the former one
- About registers, it is possible to see that the results are similar between the HDL Coder design and the not optimized implementation in Catapult, passing from 6012 to 5168. But a significant saving is obtained in the optimized design, where the number of registers employed is reduced of about 40%
- Regarding the number of DSPs, as already explained in the previous chapter, it is increased of a lot for the initial implementation (about 140% more) while it is decreased for the optimized one.

6.1.2 Initial implementations at $240MHz$ and $260MHz$

This table here shows the results obtained in Vivado for both the Catapult HLS implementations evaluated at two different frequencies (at $240MHz$, already examined, and at $260Mhz$) under the different Percentage Sharing Allocation of before (20%, 0% and -20%). These directives allow to analyze how Catapult is able to manage the available components in the target FPGA in order to obtain better performances, with all the consequences in terms of latency, power consumption and resources utilization.
Moreover, it is important to highlight that unlike for Catapult HLS, no results are reported for the HDL Coder implementation because setting higher frequencies to the model, the synthesis tool returned always more negative slack values. So it indicates that the HDL Coder design is limited at that operating frequency and it is unable to reach higher performances. For this reason only Catapult HLS architectures are reported in the following table.
Regarding the timing analysis, only the two implementations at $240MHz$ with 20% and 0% percentages respect the timing constraints, while the third one has a negative slack, which implies that it does not able to reach the target frequency. Moreover, the two positive

slacks are also very small so the two architectures work at the limit of failure. The same pattern is repeated for the implementations at $260MHz$, where only positive percentages of allocation allow to obtain a positive slack, but also here the last implementation has a negative slack ($-0.302ns$), suggesting that the clock frequency must be decreased.

	240MHz (%=20)	**240MHz (%=0)**	**240MHz (%=-20)**	**260MHz (%=20)**	**260MHz (%=0)**	**260MHz (%=-20)**
Timing Result						
Latency (clk cycles)	21	21	19	25	23	21
WNS (ns)	0.014	0.051	-0.082	0.029	0.016	-0.302
Power Analysis						
Power (W)	1.027	1.002	1.024	1.125	1.023	1.047
Resource Utilization						
Slice LUTs	1570	1690	1688	1666	1685	1837
Slice Registers	5974	5622	5168	8337	7810	7574
Slices	2176	1815	1874	3161	2565	2966
BRAM	0.5	0.5	0.5	0.5	0.5	0.5
DSPs	247	247	247	247	247	247

Table 6.2: Comparison between initial implementations at 240MHz and 260MHz with different Percentage Sharing Allocation

For the latency, all the values are quite high (all over 20 clock cycles, except for the third column) and the implementations at higher frequency show higher values of latency, arriving also at 25 clock cycles, due to the fact that the designs are more pipelined in order to split more the critical paths and respect the frequency imposed. Moreover, the positive slack in the implementations at $240MHz$ does not suggest that the corresponding architecture is able to reach higher frequencies, like $260MHz$, but it is something that must be imposed as a constraint in order to allow to Catapult HLS and to Vivado to realize a design on the target device that is able to meet the timing constraints. So it suggests that Catapult HLS realize RTLs that are able to work at the target frequency with the minimum effort.
About the power consumptions, they are all over $1W$, reaching also a value of $1.125W$ in the implementation at $260MHz$ with a percentage of allocation of 20%. But as can be observed, the power increases moderately, even changing the frequency. These results suggest that these implementations are the worst, also in terms of resources. In fact, while the BRAM and the DSPs numbers are always the same for all the designs, it is possible to see how the number of registers and slices are increased from the lower to the higher working frequency, also due to the more aggressive pipeline applied to the architectures.

6.1.3 Optimized implementations at $240MHz$ and $260MHz$

The same table has been realized for the optimized implementation in Catapult HLS, at different frequencies and percentages of allocation, in order to demonstrate how coding style can improve the efficiency of the produced RTL. It is possible to observe how the results have significantly improved in all aspects with respect to the previous implementations. Regarding the timing analysis, the latency is reduced for each design, reaching a

value equal to 16 clock cycles in the architecture at $240MHz$ with -20% or percentage, that is almost equal to the HDL Coder reference one (15 clock cycles). The worst result is obtained for the implementation at $260MHz$ with 20% of sharing allocation, as expected, due to the increased stages of pipelining and to the high value of sharing. The slack is always positive, except for the last implementation at $260MHz$ and with a negative value of percentage, meaning that for this kind of implementation the operating frequency must be decreased of almost $0.5ns$. All the others designs can work at the target frequency, but they are also very close to the limit.
The power consumption are significantly decreased to with respect to the other implementations. In fact, considering the not optimized designs, the power is almost halved, suggesting a considerable power saving.

	240MHz (%=20)	**240MHz (%=0)**	**240MHz (%=-20)**	**260MHz (%=20)**	**260MHz (%=0)**	**260MHz (%=-20)**
Timing Result						
Latency (clk cycles)	18	18	16	21	19	18
WNS (ns)	0.145	0.041	0.019	0.010	0.016	-0.488
Power Analysis						
Power (W)	0.535	0.545	0.564	0.551	0.567	0.558
Resource Utilization						
Slice LUTs	1791	1873	1799	1861	1892	1958
Slice Registers	3570	3961	3545	5622	5517	5404
Slices	1269	1234	1253	1887	1676	1809
BRAM	0.5	0.5	0.5	0.5	0.5	0.5
DSPs	91	91	91	91	91	91

Table 6.3: Comparison between optimized implementations at 240MHz and 260MHz with different Percentage Sharing Allocation

Regarding the resources employs, the number of DSPs is drastically reduced with respect to the not optimized design (from 247 to 91), while the BRAM is always the same. In this table is possible to see that the number of total LUTs that are employed in the architecture is almost the same for all the implementations, even increasing the frequency, while the number of registers and slices in increased, as well as for the previous case, due to pipelining.

6.1.4 Latency optimization

This table shows the results of three implementations with some optimizations in order to reduce the latency of the design, with the goal of reaching or even surpassing the 15 clock cycles of the HDL Coder model. First of all, the best implementations among the previous one has been selected as a starting point, which is the optimized architecture at $240MHz$ with 20% of allocation that has reached 16 clock cycles of latency. The idea is to further modify the Percentage Sharing Allocation not only to the filter, which represents the most critical component in terms of latency, but also the bias threshold block that has 2 clock cycles of latency (from Catapult HLS report analysis).
In the first column there is the implementation where only the bias threshold's latency has been improved (from 2 to 1 clock cycle). It has been possible to reach this result

by setting the Percentage Sharing Allocation of this component at 0%. As a result, this design allows to reach a latency of 15 clock cycles, but there is a slight slack violation. However, the power consumption of this architecture is the lowest value seen among all the implementations ($0.486W$) and the number of resources is very similar to the ones employed in the starting design.

	BIAS_TH (0%): da 2 a 1	**FIR (-40%): da 8 a 7**	**BOTH: due clk ridotto**
Timing Result			
Latency (clk cycles)	15	15	14
WNS (ns)	-0.028	-2.150	-1.873
Power Analysis			
Power (W)	0.486	0.559	0.492
Resource Utilization			
Slice LUTs	1798	1754	1763
Slice Registers	3632	3470	3445
Slices	1348	1278	1262
BRAM	0.5	0.5	0.5
DSPs	91	91	91

Table 6.4: Latency optimizations

In the second column there is the architecture obtained by setting the percentage to -40% to the FIR filter. It is able to reach a latency of 15 clock cycles too, but the slack significantly worse ($-2.150ns$), suggesting that this implementation is not compatible with the target frequency. The power consumption has been slightly increased, while the resources are almost the same to the previous implementation.
In the last column the results of the design with both the two directives set before are shown. The best result in terms of latency has been reached, reducing it to 14 clock cycles, but also here the timing constraints are not met, highlighting the impossibility to make this design work at the desired frequency. The power consumption is about $0.5V$ and also here the resources utilization is almost always the same.
So the table indicates the trade-off between timing constraints and latency reduction. While each of the individual optimizations can achieve the goal of 15 cycles and the overall strategy can reduce the latency to 14 cycles, the decreasing slack values mean that these architectures can not operate at 240 MHz. However, the result is significant because it demonstrates how Catapult HLS can be employed to shape the design flexibly in order to explore aggressive optimizations.

6.1.5 Time effort

The overall time effort required to deploy this thesis can be divided into different steps, each one with its own duration. The first one was the learning and the familiarization with all the tools used in this work, which required about one month and a half, considering not only self-learning but also the days taken on the official training for both HDL Coder

and Catapult. This first stage was needed in order to acquire the knowledge and the experience with the toolchains and the workflows, from modeling the algorithm to the place and route of the hardware. After that, the next step involved the implementation of the algorithm on Matlab and Simulink that lasted approximately one month. This step involved building the high-level model, verifying its behavior and creating a structure suitable for the scope.

Then the HDL Coder implementation has been realized, which took another month. This step involved adapting the Simulink model to the requirements of the HDL Coder environment, converting it into a compatible model made up of synthesizable blocks and performing simulations, verifications and optimizations to obtain the desired VHDL description. The last step regarded the realization of all the implementations in Catapult HLS, which took more or less a month too. In this phase, all the C++ classes were written, exploring different coding styles, and all the different optimized versions were created with different directives and constraints in terms frequency variations, percentage sharing allocation and latency-based optimizations.

In total, the project required about five months considering all the discussed topics. By considering the two flows separately, HDL Coder and Catapult HLS, they took about the same time but, for the former it took one month only to realize the discussed implementation, while for the latter it took one month to realize all the different architectures treated in chapters 5 and 6.

Chapter 7

Conclusions

The aim of this work to implement a pulse detection algorithm and to compare its implementation on a target FPGA thanks to two different design flows, Simulink HDL Coder tool and Siemens EDA Catapult HLS has been successfully achieved. Moreover, the purpose was also to analyze both workflows in terms of flexibility, performance, optimizations and design effort.
The obtained results highlight that HDL Coder provides the fastest and most straightforward path from the high-level Simulink model to the hardware implementation. Once this model had been verified in Simulink, the code generation process required minimal adjustments to generate the VHDL source codes for the synthesis. This model has been able to achieve a frequency of $240MHz$, with very low latency compared to the other implementations, and testbench and compilation scripts were generated automatically by the tool, reducing significantly the verification time. However, it has also some drawbacks: the final architecture is extremely fixed, with very little control over internal optimizations. As a result, resource utilization (LUTs, registers, DSPs) was high and attempted to push the performance to higher frequencies always was unsuccessful, with Vivado complaining about negative slack values.
However, Catapult HLS was more challenging, but it is characterized by high flexibility and optimization capability. The C++ class-based modeling provided the ability for a hardware-oriented and modular description of the design. Moreover, the results demonstrated the direct impact of coding style on hardware efficiency: rearranging the FIR loop into separate shift and MAC phases produced a significant reduction in latency, resource sharing and synthesis quality. Catapult, unlike HDL Coder, allowed to reach higher frequencies than the target one ($260MHz$) and under various percentage sharing allocations available, so it was possible to systematically study a lot of possible trade-offs in terms of timing, latency and resources. Meanwhile the first implementations on Catapult were inferior to the HDL Coder one, the improved versions matched or even surpassed it in efficiency, particularly with reduced power consumption and better allocation of resources. The only real drawback is that results are less deterministic and have to be attained through a careful balance of coding practice and synthesis directives.

The trade-off between the two flows is as follows: HDL Coder is better suited in the context of fast, trustworthy implementation with RTL generation directly from high-level models with less human intervention. Catapult HLS is better if the goal is design space exploration and resource optimization with different constraints. A decision between them depends on the project priorities: if time-to-market and quick prototyping are the priorities, HDL Coder is the best choice; if fine-grained control, optimization and flexibility are essential, Catapult HLS is the best one.

Some future developments immediately suggest themselves. One would be to implement the designs onto real FPGA hardware to confirm the accuracy of the estimated timing, power and resources utilization. Moreover, alternative peak detection approaches or more advanced filtering designs could be explored, in order to use Catapult's programmability in terms of automated design space exploration.

Lastly, HDL Coder and Catapult HLS were demonstrated to be capable of realizing different implementations of the intended signal processing chain. HDL Coder delivered a fast, reliable solution at the cost of limited optimization, while Catapult, is better for efficient structures. This work shows how such tools are complementary rather than competing, each being suitable to another stage or priority of a hardware design project. Together, they give a total picture of the modern FPGA design landscape, balancing the needs of fast deployment against full optimization potential.

Appendix A

MATLAB Source Codes

A.1 Calcolo_Vrms_noise_e_segnale_ver01.m

```
% ------------------------------------------------------------------------
% -------- Calcolo Vrms del rumore e dell'ampiezza del segnale in --------
% -------- funzione della potenza Ps del segnale ricevuto e degli --------
% -------- altri parametri usando le formule del modello  mathcad --------
% --------    "OG.NGOWS_radiometric_model_rev1.0_20190430" ---------------
% ------------------------------------------------------------------------
clear all;
close all;
% clc;

% -------------------------- COSTANTI DI SISTEMA -------------------------
q = 1.602176634e-19;            % carica elettrone
detector_BW = 50e6;             % [Hz] detector electrical bandwidth
M_adp_gain = 9.244;             % 9.168; ADP gain, from interpolated curve
Rapd = 1.087;                   % 1.09; [A/W] detector responsivity
keff = 0.45;
Tz = 34e3;                      % [Ohm] detector resistance
Gain = 3.5;                     % preAmp voltage Gain
eta_rx_bw = 0.4723;             % 0.45972; % Optical bandwith vs Receiver bandwith
                                % Band penalty due to the filter effect of the receiver
% eta_rx_bw = 1;                % without receiver bandwith penality

% -------------------------- PARAMETRI DI SISTEMA ------------------------
%                               % Ps_det_in is the signal power on detector due to
    backscattered signal
% Ps_det_in = 20.779e-9;        % [W] from target type1 (target_distance = 450m)
% Ps_det_in = 14.86e-9;         % [W] from target type1 (target_distance = 450m)
% Ps_det_in = 14.418e-9;        % [W] from target type1 with speckle effect eta_spk =
    0.9704 (target_distance = 450m)
% Ps_det_in = 0;                % [W] To calculate noise without Signal
% Pb_det_in = 5.448e-10;        % [W] con sfondo 10 W/(m2 sr um) Potenza di backgroud a
    valle del filtro IF
Pb_det_in = 2.724e-9;          % [W] con sfondo 50 W/(m2 sr um) Potenza di backgroud a
    valle del filtro IF
% Pb_det_in = 0;                % [W] con sfondo  0 W/(m2 sr um) Potenza di backgroud a
    valle del filtro IF
Pb = Pb_det_in;                 % Potenza di backgroud
```

```
Pb_ase = 2.942e-10;             % 0.736e-9; % [W] Received power into RX/TX path due to
    ase
Ids = 3.3e-9;                   % 9.702e-12; % [A] Ids := IDS(Is0) = dark due to surface
Idb = 3.3e-9;                   % 3.465e-9; % [A] Idb := IDB(Is0) = dark due to bulk
in_LLAM = 0.6e-12;              % [A] in_LLAM = 0.6 pA / sqrt(Hz);

% ------------------------------ CALCOLI ---------------------------------
R = Rapd;                       % responsivity
M = M_adp_gain;                 % ADP gain
Deltaf = detector_BW;           % detector electrical bandwidth
F = keff*M+(1-keff)*(2-1/M);    % Noise figure

k = 1;
for x = 0:1e-9:6000e-9
    Ps_det_in(k) = x;                                   % vettore contenente la
    potenza di segnale ricevuto
    Ps = Ps_det_in(k) * eta_rx_bw;                      % [W] Effective power on
    detector
    In_signal = R * Ps;                                % [W] Photocurrent
    In_background = R * Pb;
    % ------ Noise current shot due to signal
    insh_signal = sqrt(2 * q * In_signal * M^2 * F * Deltaf);
    % ------ Noise current shot due to backgroud & dark
    insh_dark   = sqrt(2 * q * (Ids + Idb * M^2 * F) * Deltaf);
    insh_back   = sqrt(2 * q * (In_background * M^2 * F) * Deltaf);
    % ------ Noise current due to electronics ( johnson + flicker )
    in_tia      = in_LLAM * sqrt(Deltaf);
    % ------ Noise current due to internal background (Straylight)
    insh_stray  = sqrt(2 * q * (R * Pb_ase) * (M^2 * F) * Deltaf);
    % ------ Total noise current (rms value)
    ni_tot = sqrt(insh_signal^2+insh_dark^2+insh_back^2+in_tia^2+insh_stray^2);
    % ------ Calcolo tensioni di rumore e segnale (rms)
    Vnoise(k)   = Gain * ni_tot * 1 * Tz;               % voltage total noise
    Vsignal(k)  = Gain * (Ps * Rapd) * M * Tz;          % voltage signal level
    % ------------------------------------------------------------------------
    SNR_lin(k) = Vsignal(k)/Vnoise(k);
    SNR_db(k) = 20*log10(Vsignal(k)/Vnoise(k));
    k = k + 1;
end

% ------------------------------ OUTPUTS ---------------------------------

Vout = [0, 5, 10, 15, 20, 25, 30];
Vout = [Vout, 40 , 50 , 60, 70, 80, 90, 100];
Vout = [Vout, 200, 300, 400, 500, 600, 700, 800, 900, 1000];
Vout = [Vout, 1500, 2000, 2500, 3000];
Vout = Vout .* 1e-3;            % Max output voltage range [mV]
Vout_nr = length(Vout);
for i = 1:Vout_nr
    delta = abs(Vsignal - Vout(i));
    minValue = min(delta);
    idx(i) = find(delta == minValue);
end

ff=figure(1);
set(ff,'Position',[10 60 400 565]);
ax1(1) = subplot(3,1,1);
plot(Ps_det_in, Vsignal, 'b');
grid minor;
title('Signal Amp. vs Signal Power');
xlabel('Signal Power [W]');
```

```
ylabel('Signal Amp. Vrms [V]');
ax1(2) = subplot(3,1,2);
plot(Ps_det_in, Vnoise, 'r');
grid minor;
title('Noise Amp. vs Signal Power');
xlabel('Signal Power [W]');
ylabel('Noise Amp. Vrms [V]');
ax1(3) = subplot(3,1,3);
plot(Ps_det_in, SNR_lin, 'g');
grid minor;
title('SNR (linear) vs Signal Power');
xlabel('Signal Power [W]');
ylabel('SNR (linear)');
linkaxes(ax1,'x');


ff=figure(2);
set(ff,'Position',[430 60 400 565]);
ax1(1) = subplot(3,1,1);
plot(Ps_det_in, Vsignal, 'b');
grid minor;
title('Signal Amp. vs Signal Power');
xlabel('Signal Power [W]');
ylabel('Signal Amp. Vrms [V]');
ax2(1) = subplot(3,1,2);
plot(Vsignal, Vnoise, 'r');
hold on;
plot(Vsignal(idx), Vnoise(idx), 'ro');
grid minor;
title('Noise Amp. vs Signal Amp.');
xlabel('Signal Amplitude [V]');
ylabel('Noise Amp. Vrms [V]');
ax2(2) = subplot(3,1,3);
plot(Vsignal, SNR_lin, 'g');
hold on;
plot(Vsignal(idx), SNR_lin(idx), 'go');
grid minor;
title('SNR (linear) vs Signal Amp.');
xlabel('Signal Amplitude [V]');
ylabel('SNR (linear)');
linkaxes(ax2,'x');
```

A.2 Par_Sim_GenData.m

```
% ------------------------------------------------------------------------
% -------- Calcolo dei parametri di ingresso del modello Simulink --------
% -------- che simula la catena di elaborazione digitale di NGOWS --------
% ------------------------------------------------------------------------
close all;
% rng('shuffle');
% x = rand(1);
% seed = floor(sum(5*x*100*clock))
% seed = 962247;
% seed = 851302;
% seed = 719089;
% seed = 682147;
% seed = 526073;
% seed = 405729;
% seed = 385261;
% seed = 208832;
% seed = 130425;
seed = 091537;
rng_info = rng(seed,'twister');
RandomSeed = floor(rand*10000);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% parametri generali
ideal_sig = 0;                      % generazione di segnali con dati ideali o reali
Max_false = 100;                    % massimo numero di falsi picchi gestibili
Fs_bkg = 10e9;                      % Frequenza di campionamento del background
Fs_imp = 160e9;                     % Frequenza di campionamento dell'impulso
Fs = Fs_bkg;                        % Frequenza di campionamento in uscita
Pr = 10e-6;                         % slot temporale per ricevere un impulso
time_span = 100e-6;                 % indica la durata totale della simulazione
Npts = round(time_span*Fs);
Bias_bkgd = 0.03;
dt = 1/Fs;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% files per lettura del background (caso dati reali)
% l'intervallo [t1,t2] è quello dove stimare il valore rms del background (disturbi
    esclusi)
% FILE: 'bkgd_width50us_step5us_fs10ghz.mat'
% inp_file_bkgd = 'bkgd_width50us_step5us_fs10ghz.mat';
% t1 = 3.4e-6; t2 = 7e-6;
% FILE: 'bkgd_width100us_step10us_fs10ghz.mat'
% inp_file_bkgd = 'bkgd_width100us_step10us_fs10ghz.mat';
% t1 = 11.5e-6; t2 = 15.1e-6;
% FILE: 'bkgd_scan_width50us_step5us_fs10ghz.mat'
% inp_file_bkgd = 'bkgd_scan_width50us_step5us_fs10ghz.mat';
% t1 = 5.5e-6; t2 = 9e-6;
% FILE: 'bkgd_scan_width100us_step10us_fs10ghz.mat'
inp_file_bkgd = 'bkgd_scan_width100us_step10us_fs10ghz.mat';
t1 = 13.6e-6; t2 = 17.1e-6;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% files per lettura dell'impulso (caso dati reali)
% FILE: 'pulse_width100ns_step10ns_fs160ghz.mat'
inp_file_pls  = 'pulse_width100ns_step10ns_fs160ghz.mat';
% FILE: 'pulse_satur1_width100ns_step10ns_fs160ghz.mat'
% inp_file_pls  = 'pulse_satur1_width100ns_step10ns_fs160ghz.mat';
```

```
% FILE: 'pulse_satur2_width100ns_step10ns_fs160ghz.mat'
% inp_file_pls  = 'pulse_satur2_width100ns_step10ns_fs160ghz.mat';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% output files
BackgSeq = 'BackgSequence.mat';
PulseSeq = 'PulseSequence.mat';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% setting del punto di lavoro Vsignal/Vnoise (SNR)
fprintf('\nInserisci un indice della tabella Vsignal-vs-Vnoise nel range [%d, %d]',2,
    Vout_nr);
k_tab = input('\nIndice : ');
if (k_tab < 2)||(k_tab > Vout_nr), error('Indice al di fuori del range ammesso !'); end
SignAmp  = Vsignal(idx(k_tab));
NoiseAmp = Vnoise(idx(k_tab));
SNR = SNR_db(idx(k_tab));
Thres = SignAmp/2;
% Thres = 0.0062;

t = 0:dt:time_span-dt;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% generazione del rumore di fondo
if ideal_sig
    BSequence = wgn(1,Npts,0,1,RandomSeed,'real');
    rms_backg = std(BSequence);
    BSequence = BSequence./rms_backg;
    BackgSequence = [t; BSequence];
    save(BackgSeq,'BackgSequence');
    bias = 0;
else
    % file di background di ingresso (disturbi inclusi)
    backg = load(inp_file_bkgd,'data');
    if length(backg.data)<Npts, error('La sequenza ha durata minore del tempo di
    simulazione predefinito !'); end
    BSequence = backg.data(1:round(Fs_bkg/Fs):end);
    % calcolo rms del rumore di fondo (disturbi esclusi)
    rms_backg = std(BSequence(round(t1*Fs):round(t2*Fs)));
    % rms_backg = std(BSequence);
    BSequence = BSequence(1:Npts)./rms_backg;
    BackgSequence = [t; BSequence];
    save(BackgSeq,'BackgSequence');
    bias = Bias_bkgd;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% generazione degli impulsi
Nimp = round(time_span/Pr);             % Num impulsi da generare nel tempo simulato
PSequence = zeros(1,Npts);
if ideal_sig
    Sigma  = 2.5e-9;                    % Sigma dell'impulso gaussiano
    tcum = 0;
    for n = 1:Nimp
        t0 = tcum+rand*Pr;
        y = 1/sqrt(2*pi*Sigma^2).*exp(-(t-t0).^2./(2*Sigma^2));
        PSequence = PSequence + y;
        tcum = tcum + Pr;
    end
    Amp = max(PSequence);
    PSequence = PSequence./Amp;
    PulseSequence = [t; PSequence];
```

```
115     save(PulseSeq,'PulseSequence');
116     axis_range = [t(1)*1e6 t(end)*1e6 -0.2 +1.2];
117     str = sprintf('\nCreati %d impulsi ideali\n',Nimp);
118 else
119     % file degli impulsi di ingresso
120     inp_pulse = load(inp_file_pls,'data');
121     if length(inp_pulse.data)>(Pr*Fs_imp), error('La sequenza ha durata maggiore del
        periodo predefinito !'); end
122     pulse = inp_pulse.data(1:round(Fs_imp/Fs):end);
123     npts = length(pulse);
124     dnpts = Npts-npts;
125     [max_pulse, imax] = max(pulse);
126     pulse = cat(2,pulse,zeros(1,dnpts));
127     tcum = 0;
128     for n = 1:Nimp
129         n0 = round((tcum+rand*Pr)*Fs);
130         PSequence = PSequence+circshift(pulse,n0-imax);
131         tcum = tcum + Pr;
132     end
133     PSequence = PSequence./max_pulse;
134     PulseSequence = [t; PSequence];
135     save(PulseSeq,'PulseSequence');
136     axis_range = [t(1)*1e6 t(end)*1e6 -1 +1.2];
137     str = sprintf('\nCreati %d impulsi reali\n',Nimp);
138 end
139 fprintf(str);
140 TSequence = SignAmp*PSequence+NoiseAmp*BSequence;
141
142 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
143 % plot delle sequenze
144 figure(1);
145 subplot(3,1,1);
146 plot(t*1e6, PSequence);
147 title('Norm. Pulse sequence');
148 xlabel('time (us)');
149 ylabel('amplitude (V)');
150 axis(axis_range);
151 grid;
152 subplot(3,1,2);
153 plot(t*1e6, BSequence);
154 title('Norm. Background sequence');
155 xlabel('time (us)');
156 ylabel('amplitude (V)');
157 axis auto;
158 grid;
159 subplot(3,1,3);
160 plot(t*1e6, TSequence);
161 title('Effective input sequence');
162 xlabel('time (us)');
163 ylabel('amplitude (V)');
164 axis auto;
165 grid;
166
167 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
168 % parametri dell'ADC
169 Fc = 500e6;                          % Frequenza campionamento ADC
170 Nbit = 12;                           % Numero di bits del campionatore e
        rappresentazione fixed point
171 Vref = 2;                            % Max voltage range del campionatore [V]
172 [min_q, max_q] = range(quantizer('ufixed',[Nbit Nbit-1],'floor'));
173
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% parametri della FPGA (filtraggio)
delay_tol = 10;                         % Massima tolleranza temporale sul riconoscimento degli impulsi (campioni)
                                        % La tolleranza deve essere aumentata con impulsi saturati e allungati
                                        % e diminuita con una densità di falsi allarmi molto elevata !
% Definizione dei filtri LP per la rimozione dei disturbi
LPF1 = [0.000655981953924781105436103700867533917
     0.000128784965099793517360679540040280244
    -0.000600731316281576414525811280498146516
    -0.001977077961564597738397530690690473421
    -0.003838224263480642575036005936794936133
    -0.005745952733168708387312406671298958827
    -0.007076019603187229563279192490199420718
    -0.007223931651946153588428334302307121106
    -0.005866063014387002359784073490800437867
    -0.003164273437085439455440649680895148776
     0.000191530311762752201640291893625089 87
     0.003156850108504380501145414328334481979
     0.004698611872293696582747468681873215246
     0.004211719907221187983448196234803617699
     0.001832970683490517610839121154242548073
    -0.001524740464593894728739797983507742174
    -0.004465460153767553466708228881998365978
    -0.005647995582130103942031240649155 55298
    -0.004376002652059387751370955754737224197
    -0.000973473950597534914107311632136543267
     0.003246027300983316248722543306826082699
     0.006454470816553506580526278213483237778
     0.007072133291352749630276353087765528471
     0.004512401750680489209699430830369237 81
    -0.000431905251014113905801139470241878371
    -0.005782490609131183366409256763063240214
    -0.009131988894074048040971902651108393911
    -0.008687569699303907336251917570280056 68
    -0.004169367495575031744292626711967386655
     0.002869400433998602062413141311481012963
     0.009513463871805944924164855081016867189
     0.012639999341674076904773471596854506 06
     0.010318122593908815240459908579850889 51
     0.002853594132863627747670465240048542 91
    -0.007039854106489312063343888326016895007
    -0.015132918518790249018390881019513471983
    -0.017384223997949234397086826220402144827
    -0.011803868232189457243563879274006467313
     0.000323866773577431406663795776523784298
     0.014491745950322511976571249192602408584
     0.024448694590241817209408381472712790128
     0.024682613655369826932695076493473607115
     0.012994576648192876847098808923419710482
    -0.007892058245624486434444122551212785766
    -0.030479926229310453505982891897474473808
    -0.044572765042912521060713970655342563987
    -0.040705532102430386220959235288319177926
    -0.013764619380164206177696861743697809288
     0.034538085846611304985387391752738039941
     0.095357948732928743407200045112404040992
     0.154854881651869585867942191725887823 85
     0.198182832258700064587131350890558678657
```

```
     0.21403122804476409912766143861517775803 8
     0.19818283225870006458713135089055867865 7
     0.15485488165186958586794219172588782385
     0.09535794873292874340720004511240404099 2
     0.03453808584661130498538739175273803994 1
    -0.01376461938016420617769686174369780928 8
    -0.04070553210243038622095923528831917792 6
    -0.04457276504291252106071397065534256398 7
    -0.03047992622931045350598289189747447380 8
    -0.00789205824562448643444412255121278576 6
     0.01299457664819287684709880892341971048 2
     0.02468261365536982693269507649347360711 5
     0.02444869459024181720940838147271279012 8
     0.01449174595032251197657124919260240858 4
     0.00032386677357743140666379577652378429 8
    -0.01180386823218945724356387927400646731 3
    -0.01738422399794923439708682622040214482 7
    -0.01513291851879024901839088101951347198 3
    -0.00703985410648931206334388832601689500 7
     0.00285359413286362774767046524004854291
     0.01031812259390881524045990857985088951
     0.01263999934167407690477347159685450606
     0.00951346387180594492416485508101686718 9
     0.00286940043399860206241314131148101296 3
    -0.00416936749557503174429262671196738665 5
    -0.00868756969930390733625191757028005668
    -0.00913198889407404804097190265110839391 1
    -0.00578249060913118336640925676306324021 4
    -0.00043190525101411390580113947024187837 1
     0.00451240175068048920969943083036923781
     0.00707213329135274963027635308776552847 1
     0.00645447081655350658052627821348323777 8
     0.00324602730098331624872254330682608269 9
    -0.00097347395059753491410731163213654326 7
    -0.00437600265205938775137095575473722419 7
    -0.00564799558213010394203124064915755298
    -0.00446546015376755346670822888199836597 8
    -0.00152474046459389472873979798350774217 4
     0.00183297068349051761083912115424254807 3
     0.00421171990722118798344819623480361769 9
     0.00469861187229369658274746868187321524 6
     0.00315685010850438050114541432833448197 9
     0.00019153031176275220164029189362508987
    -0.00316427343708543945544064968089514877 6
    -0.00586606301438700235978407349080043786 7
    -0.00722393165194615358842833430230712110 6
    -0.00707601960318722956327919249019942071 8
    -0.00574595273316870838731240667129895882 7
    -0.00383822426348064257503600593679493613 3
    -0.00197707796156459773839753069069047342 1
    -0.00060073131628157641452581128049814651 6
     0.00012878496509979351736067954004028024 4
     0.00065598195392478110543610370086753391 7];
LPF1_gain = 1.385;
Num1 = LPF1_gain*LPF1';

Num = Num1;
LPF_gain = LPF1_gain;
LPF_delay = (length(Num)-1)/2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
293 % output dei parametri principali del modello
294 fprintf('\nOH ampiezza segnale [V] = %f', SignAmp);
295 fprintf('\nOH ampiezza rumore di fondo [Vrms] = %f', NoiseAmp);
296 fprintf('\nOH SNR [db] = %f', SNR);
297 fprintf('\nOH bias [V] = %f', bias);
298 fprintf('\nADC numero di bits = %d', Nbit);
299 fprintf('\nADC range di tensione [V] = %f', Vref);
300 fprintf('\nSeme Random = %d\n\n', RandomSeed);
```

Appendix B

HLS C++ Source Codes and scripts

B.1 bias_threshold.h

```
/*
*bias_threshold.h
*
*Description: This header file contains the implementation of the bias threashold class
*
*/

#pragma once

#include "param_config.h"
#include <ac_int.h>
#include <ac_fixed.h>
#include <ac_channel.h>
#include <mc_scverify.h>

class BiasTh {
  private:
    bus_t in1_sub;
    bus_t sub_fir_max;
    bus_t sub_thres_sub;
      max_find_rst_t sub_sign;
        max_find_rst_t rst_max;

    public:
      BiasTh(){
        rst_max=0;
      }//constructor

      #pragma hls_design interface
      void CCS_BLOCK(bias_threshold)(ac_channel<fir_out_t> &fir_out,
              ac_channel<max_find_data_t>  &max_in,    //block interface
              ac_channel<max_find_rst_t> &in_rst1,
              ac_channel<max_find_rst_t> &in_rst2,
              ac_channel<max_find_rst_t> &in_rst3){

            static const bus_t LPF_gain=1.385;
        static const bus_t bias=0.03;
        static const bus_t Thres=0.0085;

            if(fir_out.available(1)){
          in1_sub=fir_out.read();
          sub_fir_max = in1_sub-(LPF_gain*bias); //sub operation between filter output
    and bias
        }// if

        max_in.write(sub_fir_max);
        sub_thres_sub=Thres-sub_fir_max; //sub between subtraction and threshold

        if (sub_thres_sub>=0){ //equivalent of sign block
```

```
49            sub_sign=0;
50          } else {
51            sub_sign=-1;
52          }
53
54          if ((sub_sign+1)>0 && !(rst_max>0)) {
55            in_rst3.write(1);
56          }
57
58          rst_max=sub_sign+1;
59          in_rst1.write(rst_max);
60          in_rst2.write(rst_max);
61        }
62  };
```

B.2 cnt_en.h

```
1   /*
2   *cnt_en.h
3   *
4   *Description: This file contains the implementation of counter with enable that is the
        equivalent for the Filtered Peaks Finder of Simulink
5   *
6   */
7
8   #pragma once
9
10  #include "param_config.h"
11  #include <ac_channel.h>
12  #include <ac_int.h>
13  #include <mc_scverify.h>
14
15  class FilteredPeaksFinder{
16    private:
17      filt_cnt_out_t peak_num;
18      bool prev_en;
19
20    public:
21      FilteredPeaksFinder(){
22        peak_num=0; //initialization
23        prev_en=false;
24      }//constructor
25
26      #pragma hls_design interface
27      void CCS_BLOCK(en_cnt)(ac_channel<filt_cnt_en_t> &en,
28            ac_channel<filt_cnt_out_t> &num){
29
30        filt_cnt_en_t chan_en;
31
32        chan_en=en.read();
33        bool curr = (chan_en>0); //it controls if en is greater than 0 and assign it
34        bool fal_edge= (prev_en && !curr); //falling edge detection (equivalent to rising
      edge detection of negated input)
35
36        if(fal_edge){
37          peak_num++;
38          num.write(peak_num);
39        }//if
40        prev_en=curr;
41      }//CCS_BLOCK
42  }; //class
```

B.3 cnt16.h

```
/*
*cnt16.h
*
*Description: This file contains the implementation of the 16-bit free running counter
*
*/

#pragma once

#include "param_config.h"
#include <ac_int.h>
#include <ac_channel.h>
#include <mc_scverify.h>

class counter16{
  private:
    cnt_t cnt_chan;
    cnt_t shift_reg[LATENCY];
  public:
    counter16 (){} //constructor

    #pragma hls_design interface
    void CCS_BLOCK(cnt16)(ac_channel<cnt_t> &cnt){

      SH_REG: for(int i=LATENCY-1; i>0; i--){
        shift_reg[i]=shift_reg[i-1];
      }
      shift_reg[0]=cnt_chan;

      cnt.nb_write(shift_reg[LATENCY-1]); //nb_write (non-blocking) is to not stall execution

      cnt_chan++; //no check if it arrives to 2^16 because the wrapping operation is done intrinsecally in the data type
    }//CCS_BLOCK

}; //class
```

B.4 difference.h

```
/*
*difference.h
*
*Description: This header file contains the implementation of the difference between output of the max finder and the delayed output of maximum finder
*
*/

#pragma once

#include "param_config.h"
#include <ac_int.h>
#include <ac_fixed.h>
#include <ac_channel.h>
#include <mc_scverify.h>

class Diff {
  private:
    max_find_data_t max_out_curr;
    max_find_data_t max_out_prev; //delayed output of maximum finder
    max_find_data_t diff;
    bool init; //check if initialized

  public:
```

```
    Diff(){
            init=false;
      max_out_prev=0;
        }//constructor

    #pragma hls_design interface
    void CCS_BLOCK(differ)(ac_channel<max_find_data_t> &max_out,
             ac_channel<s_h_data_t>  &in_sh1,    //block interface
             ac_channel<s_h_en_t> &en_sh1,
                    ac_channel<s_h_en_t> &en_sh2){

           if(max_out.available(1)){
        max_out_curr=max_out.read();
        if(!init){
          diff=1;
          init=true;
        } else {
          diff=max_out_curr-max_out_prev;
        } //if

        max_out_prev=max_out_curr;
        in_sh1.write(max_out_prev);
        en_sh1.write(diff);
        en_sh2.write(diff);
      } //if
        }
};
```

B.5 fir_coeffs.h

```
/*
*FIR_coeffs.h
*
*Description: This header file contains the FIR filter coefficients
*
*/

#pragma once

#include "param_config.h"
#include <ac_fixed.h>

static const fir_coeffs_t FIR_COEFFS[FIR_TAPS]={
     0.000655981953924781105436103700867533917,
     0.000128784965099793517360679540040280244,
    -0.000600731316281576414525811280498146516,
    -0.001977077961564597738397530690690473421,
    -0.003838224263480642575036005936794936133,
    -0.005745952733168708387312406671298958827,
    -0.007076019603187229563279192490199420718,
    -0.007223931651946153588428334302307121106,
    -0.005866063014387002359784073490800437867,
    -0.003164273437085439455440649680895148776,
     0.000191530311762752201640291893625089870,
     0.003156850108504380501145414328334481979,
     0.004698611872293696582747468681873215246,
     0.004211719907221187983448196234803617699,
     0.001832970683490517610839121154242548073,
    -0.001524740464593894728739797983507742174,
    -0.004465460153767553466708228881998365978,
    -0.005647995582130103942031240649155755298,
    -0.004376002652059387751370955754737224197,
    -0.000973473950597534914107311632136543267,
     0.003246027300983316248722543306826082699,
     0.006454470816553506580526278213483237778,
     0.007072133291352749630276353087765528471,
     0.004512401750680489209699430830369237810,
    -0.000431905251014113905801139470241878371,
```

```
    -0.005782490609131183366409256763063240214,
    -0.009131988894074048040971902651108393911,
    -0.008687569699303907336251917570280056 68,
    -0.004169367495575031744292626711967386655,
     0.002869400433998602062413141311481012963,
     0.009513463871805944924164855081016867189,
     0.012639999341674076904773471596854506 06,
     0.010318122593908815240459908579850889 51,
     0.002853594132863627747670465240048542 91,
    -0.007039854106489312063343888326016895007,
    -0.015132918518790249018390881019513471983,
    -0.017384223997949234397086826220402144827,
    -0.011803868232189457243563879274006467313,
     0.000323866773577431406663795776523784298,
     0.014491745950322511976571249192602408584,
     0.024448694590241817209408381472712790128,
     0.024682613655369826932695076493473607115,
     0.012994576648192876847098808923419710482,
    -0.007892058245624486434444122551212785766,
    -0.030479926229310453505982891897474473808,
    -0.044572765042912521060713970655342563987,
    -0.040705532102430386220959235288319177926,
    -0.013764619380164206177696861743697809288,
     0.034538085846611304985387391752738039941,
     0.095357948732928743407200045112404040992,
     0.154854881651869585867942191725887823 85,
     0.198182832258700064587131350890558678657,
     0.214031228044764099127661438615177758038,
     0.198182832258700064587131350890558678657,
     0.154854881651869585867942191725887823 85,
     0.095357948732928743407200045112404040992,
     0.034538085846611304985387391752738039941,
    -0.013764619380164206177696861743697809288,
    -0.040705532102430386220959235288319177926,
    -0.044572765042912521060713970655342563987,
    -0.030479926229310453505982891897474473808,
    -0.007892058245624486434444122551212785766,
     0.012994576648192876847098808923419710482,
     0.024682613655369826932695076493473607115,
     0.024448694590241817209408381472712790128,
     0.014491745950322511976571249192602408584,
     0.000323866773577431406663795776523784298,
    -0.011803868232189457243563879274006467313,
    -0.017384223997949234397086826220402144827,
    -0.015132918518790249018390881019513471983,
    -0.007039854106489312063343888326016895007,
     0.002853594132863627747670465240048542 91,
     0.010318122593908815240459908579850889 51,
     0.012639999341674076904773471596854506 06,
     0.009513463871805944924164855081016867189,
     0.002869400433998602062413141311481012963,
    -0.004169367495575031744292626711967386655,
    -0.008687569699303907336251917570280056 68,
    -0.009131988894074048040971902651108393911,
    -0.005782490609131183366409256763063240214,
    -0.000431905251014113905801139470241878371,
     0.004512401750680489209699430830369237 81,
     0.007072133291352749630276353087765528471,
     0.006454470816553506580526278213483237778,
     0.003246027300983316248722543306826082699,
    -0.000973473950597534914107311632136543267,
    -0.004376002652059387751370955754737224197,
    -0.005647995582130103942031240649157552 98,
    -0.004465460153767553466708228881998365978,
    -0.001524740464593894728739797983507742174,
     0.001832970683490517610839121154242548073,
     0.004211719907221187983448196234803617699,
     0.004698611872293696582747468681873215246,
     0.003156850108504380501145414328334481979,
     0.000191530311762752201640291893625089 87,
    -0.003164273437085439455440649680895148776,
    -0.005866063014387002359784073490800437867,
    -0.007223931651946153588428334302307121106,
    -0.007076019603187229563279192490199420718,
    -0.005745952733168708387312406671298958827,
```

```
    -0.003838224263480642575036005936794936133,
    -0.001977077961564597738397530690690473421,
    -0.000600731316281576414525811280498146516,
     0.000128784965099793517360679540040280244,
     0.000655981953924781105436103700867533917,
};
```

B.6 fir_lpf.h (not optimized)

```
/*
*fir_lpf.h
*
*Description: This header file contains the implementation of the fir filter class
*
*/

#pragma once

#include "param_config.h"
#include "fir_coeffs.h"
#include <ac_fixed.h>
#include <ac_channel.h>
#include <mc_scverify.h>

class fir_class{
  private:
    fir_in_t shiftREG[FIR_TAPS];

  public:
    fir_class(){
      for(int i=0;i<FIR_TAPS;i++)
        shiftREG[i]=0; //inizialization of the registers
    } //fir_class

    #pragma hls_design interface
    void CCS_BLOCK(lpf)(ac_channel<fir_in_t> &data_in,
            ac_channel<fir_out_t> &data_out){

      fir_in_t input;
      fir_out_t output;
      fir_acc_t acc=0.0;
      fir_out_t LPF_gain=1.385;

      if(data_in.available(1)){ //control if there is at least one input
        input=data_in.read();
        acc=0.0;

        FIR: for(int i=(FIR_TAPS-1);i>=0;i--){
            fir_in_t tmpTap=(i==0) ? input :shiftREG[i-1];
            acc+=tmpTap*FIR_COEFFS[i]*LPF_gain;
            shiftREG[i]=tmpTap;
            }

         output=acc;
         data_out.write(output);
      } //if
    } //lpf
}; //fir_class
```

B.7 fir_lpf.h (optimized)

```
/*
*fir_lpf.h
*
*Description: This header file contains the implementation of the fir filter class
*
*/

#pragma once

#include "param_config.h"
#include "fir_coeffs.h"
#include <ac_fixed.h>
#include <ac_channel.h>
#include <mc_scverify.h>


class fir_class{
  private:
    fir_in_t shiftREG[FIR_TAPS];

  public:
    fir_class(){
      for(int i=0;i<FIR_TAPS;i++)
        shiftREG[i]=0; //inizialization of the registers
    } //fir_class

    #pragma hls_design interface
    void CCS_BLOCK(lpf)(ac_channel<fir_in_t> &data_in,
            ac_channel<fir_out_t> &data_out){

      fir_in_t input;
      fir_out_t output;
      fir_acc_t acc=0.0;
      fir_out_t LPF_gain=1.385;

      if(data_in.available(1)){ //control if there is at least one input
        input=data_in.read();
        acc=0.0;

        SHIFT: for(int i=(FIR_TAPS-1);i>0;i--){
            shiftREG[i]=shiftREG[i-1];
            }
            shiftREG[0]=input;

        MAC: for(int i=(FIR_TAPS-1);i>=0;i--){
            acc+=shiftREG[i] * ((fir_out_t)(LPF_gain*FIR_COEFFS[i]));
            }

         output=acc;
         data_out.write(output);
      } //if
    } //lpf
}; //fir_class
```

B.8 FPGA_wrap.h

```
/*
*FPGA_wrap.h
*
*Description: This header file contains the wrapper of all the blocks
*
*/

#pragma once

#include "param_config.h"
#include "fir_lpf.h"
#include "bias_threshold.h"
```

```
#include "max_finder.h"
#include "difference.h"
#include "s_h_peak.h"
#include "cnt16.h"
#include "cnt_en.h"
#include "SP_ram.h"
#include <ac_int.h>
#include <ac_fixed.h>
#include <ac_channel.h>
#include <mc_scverify.h>

#pragma hls_design top
class FPGA_class{
  private:
      //interface
    ac_channel<fir_out_t> fir_out;
    ac_channel<max_find_data_t> max_in;
    ac_channel<max_find_data_t> max_out;
    ac_channel<max_find_rst_t> in_rst1;
    ac_channel<max_find_rst_t> in_rst2;
    ac_channel<max_find_rst_t> in_rst3;
    ac_channel<s_h_data_t> in_sh1;
    ac_channel<s_h_data_t> out_sh1;
    ac_channel<s_h_en_t> en_sh1;
    ac_channel<s_h_en_t> en_sh2;
    ac_channel<s_h_cnt_t> in_sh2;
    ac_channel<filt_cnt_out_t> num_peak_find_out;

    //blocks
    fir_class lp_filter;
    BiasTh bias_th;
    MaxFinder maximum;
    Diff difference;
    SampleAndHold<s_h_data_t> peak_values;
    SampleAndHold<s_h_cnt_t> time_values;
    counter16 free_cnt;
    FilteredPeaksFinder finder;
    SinglePortRAM memory;

  public:
    FPGA_class() {}

    #pragma hls_design interface
    void CCS_BLOCK(FPGA)(ac_channel<fir_in_t> &ing,
              ac_channel<s_h_data_t> &peaks,
              ac_channel<s_h_cnt_t> &times){

      lp_filter.lpf(ing,fir_out);

      bias_th.bias_threshold(fir_out,max_in,in_rst1,in_rst2,in_rst3);

      maximum.max_finder(max_in,in_rst1,max_out);

      finder.en_cnt(in_rst2,num_peak_find_out);

      difference.differ(max_out,in_sh1,en_sh1,en_sh2);

      peak_values.s_h_peak(in_sh1,en_sh1,out_sh1); //sample and hold block for peak
    values

      memory.RAM_mem(out_sh1,num_peak_find_out,in_rst3,peaks);

      free_cnt.cnt16(in_sh2);

      time_values.s_h_peak(in_sh2,en_sh2,times);

    } //CCS_BLOCK(FPGA)

}; //FPGA_class
```

B.9 max_finder.h

```
/*
*max_finder.h
*
*Description: This header file contains the implementation of the maximum finder class
*
*/

#pragma once

#include "param_config.h"
#include <ac_int.h>
#include <ac_fixed.h>
#include <ac_channel.h>
#include <mc_scverify.h>

class MaxFinder {
  private:
    max_find_data_t max_val;

  public:
    MaxFinder(){ //constructor
      max_val=0.0; //inizialization
    }//constructor

    #pragma hls_design interface
    void CCS_BLOCK(max_finder)(ac_channel<max_find_data_t> &data_in,
              ac_channel<max_find_rst_t>  &rst,    //block interface
              ac_channel<max_find_data_t> &y){

      max_find_data_t din=0.0;
      max_find_rst_t reset;

      if (data_in.available(1) && rst.available(1)){  //check to avoid deadlock
        din=data_in.read();
        reset=rst.read();
      } //if (data && rst available)

      if (reset == 1){
        max_val=0.0; //reset of the maximum value
      } else  {
        if (din > max_val){  //check if the new input is larger than the previous one
          max_val=din;
        }
        y.write(max_val);
      } //if
    } //CCS_BLOCK
}; //class MaxFinder
```

B.10 mktest_lp.tcl

```
solution options defaults
flow package require /SCVerify
solution options set /Input/CompilerFlags {-DWMASK}
solution options set /Input/CppStandard c++17
solution options set /Input/TargetPlatform x86_64
solution options set /Output/GenerateCycleNetlist false
solution options set /Flows/Vivado/VivadoMode Project
options set Flows/Vivado/XILINX_VIVADO /media/data01/tools/Xilinx2023.2/Vivado/2023.2
directive set -OPT_CONST_MULTS use_library
directive set -ON_THE_FLY_PROTOTYPING false
solution file add ./tb.cpp -type C++
go analyze
go compile
```

```
solution library add mgc_Xilinx-ARTIX-7-2_beh -- -rtlsyntool Vivado -manufacturer Xilinx -family ARTIX-7 -speed -2 -part xc7a200tfbg484-2
solution library add Xilinx_RAMS
go libraries
directive set -CLOCKS {clk {-CLOCK_PERIOD 4.167 -CLOCK_EDGE rising -CLOCK_UNCERTAINTY 0.0 -CLOCK_HIGH_TIME 2.083 -RESET_SYNC_NAME rst -RESET_ASYNC_NAME arst_n -RESET_KIND sync -RESET_SYNC_ACTIVE high -RESET_ASYNC_ACTIVE low -ENABLE_ACTIVE high}}
go assembly
directive set /FPGA_class/SampleAndHold<s_h_cnt_t>/s_h_peak/main -PIPELINE_STALL_MODE stall
directive set /FPGA_class/counter16/cnt16/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/fir_class/lpf/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/SinglePortRAM/RAM_mem/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/FilteredPeaksFinder/en_cnt/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/Diff/differ/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/MaxFinder/max_finder/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/BiasTh/bias_threshold/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/fir_class/lpf/shiftREG:rsc -MAP_TO_MODULE {[Register]}
directive set /FPGA_class/fir_class/lpf/FIR -UNROLL yes
directive set /FPGA_class/counter16/cnt16/SH_REG -UNROLL yes
directive set /FPGA_class/SinglePortRAM/din:rsc -FIFO_DEPTH -1
directive set /FPGA_class/SinglePortRAM/addr:rsc -FIFO_DEPTH -1
directive set /FPGA_class/SinglePortRAM/we:rsc -FIFO_DEPTH -1
directive set /FPGA_class/max_in:cns -FIFO_DEPTH 0
directive set /FPGA_class/in_rst1:cns -FIFO_DEPTH 0
directive set /FPGA_class/in_rst2:cns -FIFO_DEPTH 0
directive set /FPGA_class/max_out:cns -FIFO_DEPTH 0
directive set /FPGA_class/in_sh1:cns -FIFO_DEPTH 0
directive set /FPGA_class/en_sh1:cns -FIFO_DEPTH 0
directive set /FPGA_class/en_sh2:cns -FIFO_DEPTH 0
directive set /FPGA_class/out_sh1:cns -FIFO_DEPTH 0
directive set /FPGA_class/in_sh2:cns -FIFO_DEPTH 0
directive set /FPGA_class/SampleAndHold<s_h_data_t>/s_h_peak/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/SampleAndHold<s_h_cnt_t>/s_h_peak/main -PIPELINE_INIT_INTERVAL 1
directive set /FPGA_class/SinglePortRAM/RAM_mem/main -PIPELINE_STALL_MODE flush
directive set /FPGA_class/counter16/cnt16/main -PIPELINE_STALL_MODE flush
directive set /FPGA_class/fir_out:cns -FIFO_DEPTH 0
directive set /FPGA_class/counter16/cnt:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set /FPGA_class/num_peak_find_out:cns -FIFO_DEPTH 0
directive set /FPGA_class/in_rst3:cns -FIFO_DEPTH 0
directive set /FPGA_class/SampleAndHold<s_h_cnt_t>/sh_in1:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
directive set /FPGA_class/SampleAndHold<s_h_cnt_t>/sh_en1:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
directive set /FPGA_class/fir_class/lpf -CLOCK_OVERHEAD 20.0
directive set /FPGA_class/fir_class/lpf -DESIGN_GOAL Latency
go architect
go allocate
directive set /REGISTER_SHARING_LIMIT 1
go extract
```

B.11 param_config.h

```
/*
*param_config.h
*
*Description: This header file contains all the definition of the parameters
*/

#pragma once
#include <ac_fixed.h>
#include <ac_int.h>
#include <cstdint> //per tipi come int16_t

////FIR Filter parameters////

```

```
14 //number of taps
15 #define FIR_TAPS    105  //number of taps of the filter
16
17 //coefficients attributes
18 #define FIR_COEFFS_WIDTH  16   //number of bits of the coefficients
19 #define FIR_COEFFS_INT     0    //number of integer bits of the coefficients
20 #define FIR_COEFFS_SIGNED true //signed coefficients
21
22 //accumulator attributes
23 #define FIR_ACC_WIDTH     17   //number of bits in the accumulators
24 #define FIR_ACC_INT     2    //number of integer bits in the accumulators
25 #define FIR_ACC_SIGNED    true //signed sum
26
27 //input and output attributes
28 #define FIR_DATA_WIDTH     17   //number of bits at the input/output
29 #define FIR_DATA_INT       2    //number of integer bits at the input/output
30 #define FIR_DATA_SIGNED    true //signed input/output
31
32 //data type definitions
33 typedef ac_fixed<FIR_DATA_WIDTH,FIR_DATA_INT,FIR_DATA_SIGNED,AC_RND,AC_SAT> fir_in_t;
                          //definition of input data type
34 typedef ac_fixed<FIR_COEFFS_WIDTH,FIR_COEFFS_INT,FIR_COEFFS_SIGNED,AC_RND,AC_SAT>
    fir_coeffs_t; //definition of coefficients data type
35 typedef ac_fixed<FIR_DATA_WIDTH,FIR_DATA_INT,FIR_DATA_SIGNED,AC_RND,AC_SAT> fir_out_t;
                          //definition of output data type
36 typedef ac_fixed<FIR_ACC_WIDTH,FIR_ACC_INT,FIR_ACC_SIGNED> fir_acc_t;
            //definition of accumulator data type
37
38 ////MAX_FINDER Parameters////
39
40 //data attributes
41 #define MAX_FIND_WIDTH     17    //number of bits of the maximum finder block
42 #define MAX_FIND_INT       2     //number of integer bits of the maximum finder block
43 #define MAX_FIND_SIGNED    true  //signed data type
44
45
46 //data type definitions
47 typedef ac_fixed<MAX_FIND_WIDTH,MAX_FIND_INT,MAX_FIND_SIGNED> max_find_data_t; //
    definition of input and output data type
48 typedef ac_int<2,true> max_find_rst_t;                                          //
    definition of reset data type
49
50
51 ////unsigned fixed point data type definition////
52 #define UFIX_WIDTH         17     //number of bits
53 #define UFIX_INT            2     //number of integer bits
54 #define UFIX_SIGNED        true   //unsigned data type
55
56 //data type definition
57 typedef ac_fixed<UFIX_WIDTH,UFIX_INT,UFIX_SIGNED> bus_t; //definition of data type
58
59 ////SAMPLE & HOLD Parameters////
60
61 //data attributes (peak values)
62 #define S_H_WIDTH_PEAK          17     //number of bits
63 #define S_H_INT_PEAK             2     //number of integer bits
64 #define S_H_SIGNED_PEAK         true   //unsigned data type
65
66 //data type definitions
67 typedef ac_fixed<S_H_WIDTH_PEAK,S_H_INT_PEAK,S_H_SIGNED_PEAK> s_h_data_t; //definition
    of input and output data type
68 typedef ac_fixed<17,2,true> s_h_en_t;                                    //enable data
    type
69
70 //data attributes (time values)
71 #define S_H_WIDTH_TIME          16      //number of bits
72 #define S_H_SIGNED_TIME         false   //unsigned data type
73
74 //data type definitions
75 typedef ac_int<S_H_WIDTH_TIME,S_H_SIGNED_TIME> s_h_cnt_t;      //definition of input and
     output data type
76 typedef ac_fixed<17,2,true> s_h_en_t;                          //enable data type
77
78
79 ////COUNTER 16 BIT////
```

```
80
81 //data attributes
82 #define CNT_WIDTH   16  //number of bits
83 #define CNT_SIGNED    false //unsigned data type
84
85 //data type definition
86 typedef ac_int<CNT_WIDTH,CNT_SIGNED> cnt_t; //definition of output data type
87
88 ////FILTERED PEAKS FINDER block////
89
90 //DELAY
91 #define LATENCY     20
92
93 //data attributes
94 #define FILT_PEAK_WIDTH   8 //number of bits
95 #define FILT_PEAK_SIGNED  false //unsigned data type
96
97 //data type definition
98 typedef ac_int<FILT_PEAK_WIDTH,FILT_PEAK_SIGNED> filt_cnt_out_t; //output data type
99 typedef ac_int<2,true> filt_cnt_en_t;        //enable data type
100
101
102 ////Single Port RAM Parameters////
103
104 //memory size
105 #define MEM_SIZE    256 //2^8 possible addresses
106
107 //data attributes
108 #define WORD_WIDTH    17  //data word width   (vedi se riesci a portare a 16 unsigned
    quindi facendo cast (ma anche tutto il resto))
109 #define WORD_INT    2   //number of integer bits
110 #define WORD_SIGNED   true  //signed numbers
111
112 //address attributes
113 #define ADDR_WIDTH    8 //256 addresses
114 #define ADDR_SIGNED   false //unsigned
115
116 //data type definitions
117 typedef ac_fixed<WORD_WIDTH,WORD_INT,WORD_SIGNED> ram_data_t; //din and dout data type
118 typedef ac_int<ADDR_WIDTH,ADDR_SIGNED> ram_addr_t;        //address data type
119 typedef ac_int<2,true> ram_we_t;                                  //write enable
    data type
```

B.12 s_h_peak.h

```
1 /*
2 *s_h_peak.h
3 *
4 *Description: this file describes the Sample & Hold that saves the value of the peaks
5 *
6 */
7
8 #pragma once
9
10 #include "param_config.h"
11 #include <ac_int.h>
12 #include <ac_fixed.h>
13 #include <ac_channel.h>
14 #include <mc_scverify.h>
15
16 template <typename T>
17 class SampleAndHold {
18   private:
19     s_h_en_t en_prev;
20     bool initialized;
21
22   public:
23     SampleAndHold () {
```

```
24      en_prev=0.0;
25      initialized=false;
26    } //constructor
27
28    #pragma hls_design interface
29    void CCS_BLOCK(s_h_peak)(ac_channel<T> &sh_in1,
30           ac_channel<s_h_en_t>   &sh_en1,
31           ac_channel<T> &sh_peak){
32
33      T in_chan;
34      s_h_en_t en_chan;
35
36      if(sh_in1.available(1)){
37        in_chan=sh_in1.read();
38
39        if(sh_en1.available(1)){
40          en_chan=sh_en1.read();
41
42                if(en_chan==0.0 && en_prev!=0.0){ //if enable is different from 0
43            sh_peak.write(in_chan);
44          }
45          en_prev=en_chan;
46        } //if en
47      } //if
48    } //void CCS_BLOCK
49  }; //class
```

B.13 SP_ram.h

```
1   /*
2   *SP_ram.h
3   *
4   *Description: This file contains the implementation of a Single Port RAM: 256x16 bit
5   *
6   */
7
8   #pragma once
9
10  #include "param_config.h"
11  #include "ac_int.h"
12  #include "ac_fixed.h"
13  #include "ac_channel.h"
14  #include "mc_scverify.h"
15
16  class SinglePortRAM{
17    private:
18      ram_data_t mem[MEM_SIZE];
19      ram_addr_t addr_chan;
20
21    public:
22      SinglePortRAM(){
23        for(int i=0;i<MEM_SIZE;i++){
24          mem[i]=0.0; //memory initialization
25        }//for
26        addr_chan=0;
27      }//constructor
28
29      #pragma hls_design interface
30      void CCS_BLOCK(RAM_mem)(ac_channel<ram_data_t> &din,
31                  ac_channel<ram_addr_t> &addr,
32                  ac_channel<ram_we_t>   &we,
33                  ac_channel<ram_data_t> &dout){
34
35        ram_we_t we_chan;
36        ram_data_t dout_prev;
37        ram_data_t din_chan;
38
39        if(addr.available(1) && we.available(1) && din.available(1)){
```

```
        addr_chan=addr.read();
        we_chan=we.read();
        din_chan=din.read();

        if(we_chan!=0){   //write operation
            mem[addr_chan]=din_chan;

          } //else {
                dout.write(mem[addr_chan]);

                //}//else
      }
    }//void CCS_BLOCK
}; //class
```

B.14 tb.cpp

```
#include "FPGA_wrap.h"
#include "param_config.h"
#include "fir_coeffs.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <ac_fixed.h>
#include <ac_channel.h>
#include <mc_scverify.h>

using namespace std;

CCS_MAIN(int argc, char *argv[]){
        ac_channel<fir_in_t> in_chan;
        ac_channel<s_h_data_t> out1_chan;
        ac_channel<s_h_cnt_t> out2_chan;

        FPGA_class dut;

  //opening the input file with the data
  ifstream infile("input_file.txt");

  //opening the output file with the filtered data
  ofstream outfile("output_file.txt");

  //check for any error reading input file
  if(!infile){
    cerr<<"Error opening input file!"<<endl;
    CCS_RETURN(1);
  }//if

  //check for any error reading output file
  if(!outfile){
    cerr<<"Error opening output file!"<<endl;
    CCS_RETURN(1);
  }//if

  vector<fir_in_t> input_vector; //vector that will contain the input data
  float in_value;

  //saving input data in the vector
  while(infile>>in_value){
      fir_in_t converted_val = fir_in_t(in_value);
    input_vector.push_back(converted_val);
  }//while

    cout<<"Starting the computation"<<endl;

    for(int i=0; i<input_vector.size(); i++){
            fir_in_t input = input_vector[i];
            in_chan.write(input);
```

```
52
53              //Loop to simulate clock until available
54
55              int timeout=100; //to avoid infinite loop
56
57              dut.FPGA(in_chan,out1_chan,out2_chan);
58
59            if (out1_chan.available(1) && out2_chan.available(1)){
60                      s_h_data_t output_peak=out1_chan.read(); //reading output channel
61                      s_h_cnt_t output_time=out2_chan.read(); //reading time
62                          outfile<<"Peaks: "<<output_peak<<" Time: "<<output_time<<endl;
    //saving results in the output file
63          }//if
64      }//for
65      outfile.close();
66      CCS_RETURN(0);
67  }
```

Bibliography

[1] Robert A. Walker and Donald E. Thomas. "A Model of Design Representation and Synthesis". In: *Proceedings of the IEEE*. Electrical and Computer Engineering Dept., Carnegie Mellon University. IEEE. Pittsburgh, PA, 1985 (cit. on p. 3).

[2] Grant Martin and Gary Smith. "High-Level Synthesis: Past, Present, and Future". In: *IEEE Design Test of Computers* 26 (July 2009), pp. 18–25 (cit. on p. 3).

[3] Gregory E. Stillman. "21 - Optoelectronics". In: *Reference Data for Engineers (Ninth Edition)*. Ed. by Wendy M. Middleton and Mac E. Van Valkenburg. Ninth Edition. Woburn: Newnes, 2002 (cit. on pp. 13, 14).

[4] Reinaldo Perez. "Chapter 2 - Noise and Interference Issues in Analog Circuits". In: *Interference into Circuits*. Ed. by Reinaldo Perez. Vol. 3. Wireless Communications Design Handbook. Academic Press, 1998, pp. 52–102 (cit. on p. 14).

[5] MathWorks. *Quantizer – Discretize input at given interval*. 2025. URL: https://it.mathworks.com/help/simulink/slref/quantizer.html (cit. on p. 33).

[6] Siemens Digital Industries Software. *Introduction to Filters: FIR versus IIR*. 2020. URL: https://community.sw.siemens.com/s/article/introduction-to-filters-fir-versus-iir (cit. on p. 38).

[7] MathWorks. *HDL Workflow Advisor Tasks*. MathWorks, Inc. 2025. URL: https://it.mathworks.com/help/hdlcoder/ug/hdl-workflow-advisor-tasks.html (cit. on p. 45).

[8] AMD. *Vivado Design Suite User Guide: Synthesis — Using Synthesis Settings*. UG901. Advanced Micro Devices, Inc. 2025. URL: https://docs.amd.com/r/en-US/ug901-vivado-synthesis/Using-Synthesis-Settings (cit. on p. 55).

[9] AMD. *Vivado Design Suite User Guide: Implementation — Implementation Strategy Descriptions*. UG904. Advanced Micro Devices, Inc. 2025. URL: https://docs.amd.com/r/en-US/ug904-vivado-implementation/Implementation-Strategy-Descriptions (cit. on p. 55).

[10] Siemens. *Catapult HLS: User Reference Guide*. Online Documentation. 2025. URL: https://docs.sw.siemens.com/it-IT/doc/886857312/202508020.catapult_useref?audience=external (cit. on pp. 62, 78).

[11] Siemens. *High-Level Synthesis (HLS) Bluebook*. E-book. Siemens Software, Apr. 2020 (cit. on p. 63).

Acknowledgments

I would like to thank Leonardo Spa for welcoming me at their Genova-Puccini plant and allowing me to carry out the thesis within their organization. It was a wonderful experience that guided me through my first professional experience and provided a stimulating environment for my technical and personal growth.

My sincere thanks go to all my colleagues, who welcomed me warmly and made me feel part of their team from the beginning. In particular I would like to express my sincere gratitude to the co-supervisors of this thesis, M. Avitabile and R. Sticca, for giving me this wonderful opportunity and for their constant support and availability. Moreover, I am grateful to Werner Bachhuber, FAE at Siemens, for his assistance during these months.

I would like also to thank Prof. M.R. Casu for his supervision and for always being available to provide feedback and suggestions.