POLITECNICO
DI TORINO

Politecnico di Torino

Department of Electrical and Electronic Engineering

FPGA-Based acceleration of the GNU Radio Software-Defined Radio: the IEEE 802.11p case

Author:
Carmelo Apostoliti

Supervisor:
Prof. Guido Masera
Prof. Luciano Lavagno
Co-Supervisor:
Prof. Luca Carloni

A thesis submitted for the degree of

MSc Embedded Systems

October 14, 2019

To Antonio and Anastasia

Abstract

GNU Radio is a free and open-source graphical programming environment that can be used to design Software-Defined Radios using a large panel of signal-processing blocks. It is widely used in hobbyist, academic, commercial and military environments to support real-world radio systems as well as research and development in wireless communications.
We propose *SDR WorkFlow Advisor*, a methodology that leverages Software-Defined Radio applications like GNURadio Companion in order to deploy the flowgraphs to be accelerated using FPGA or ASIC. The proposed methodology includes a tool which helps experienced signal-processing engineers reduce the development time and the integration complexity of IP blocks into the Software-Defined Radio signal-processing chain. Thanks to this methodology, it is possible to create complex architectures directly from the GNU Radio Companion with the same workflow used to create host-based flowgraphs (applications) and export them as AMBA-compliant accelerators for FPGA or ASIC design. We evaluate our methodology by implementing the 802.11p standard used in Vehicle-To-Vehicle Communication. It is chosen for its growing interest within the electronic engineers community; nevertheless our methodology can be applied to any algorithm.

Acknowledgements

This thesis project has been supported by the Columbia University in the City of New York, precisely by the System-Level-Design (SLD) group. All the work mentioned in this thesis project has been supervised by Professor Luca Carloni during my stay in New York City.
Further investigation have been made in Turin (Italy) at the Politecnico di Torino University under the supervision of Professor Guido Masera and Professor Luciano Lavagno.

Contents

List of Figures

Chapter 1

Introduction

> *The most difficult and complicated part of the writing process is the beginning.* - A. B. Yehoshua

The main goal of this project is to design, simulate and evaluate a wireless communication hardware accelerator base on the IEEE 802.11p starting from its software implementation in GNU Radio and extracting useful information to extend the methodology Software To Hardware (GNU Radio to FPGA) to every general algorithm.
In other words, this thesis uses the case of study of 802.11p algorithm [1] to investigate whether is possible for a general algorithm, already available in GNU Radio environment, to deploy it - in an easy and straightforward way- onto an FPGA and give it a significant speed-up regardless its software implementation.

Software-defined Radio (SDR) is a technology for radio communication. This technology is based on software-defined wireless protocols, as opposed to hardware-based solutions. This translates to supporting various features and functionalities, such as updating and upgrading through reprogramming, without the need to replace the hardware on which they are implemented.
SDR have become one of the most powerful tools when it comes to experimental and proof-of-concept solutions of new wireless technologies [1]. Even more importantly, the use of SDR proved hugely beneficial to general wireless networking research. The main advantage is that freely programmable radios provide access to all data including the physical layer and they allow to study current protocol standards in greater detail, to study new protocol variants, or even to study completely new protocols.
Following [2], we can distinguish two kinds of SDR macro-architectures based on how the physical layer is implemented:

1. first of all, the so called software-only solutions do exist. Here, the most common approach is to use a system like GNU Radio where the complete physical layer is implemented on a general purpose processor, i.e., a host PC [3]. This approach gives the user the best flexibility and also allows even newcomers to the field to quickly set up the entire communication system. On the other hand, this architecture does not allow to quickly react on received signals since streaming the samples to the host PC and decoding on a CPU running a non-real-time operating system introduces significant delays and non-determinism expressed in delay variations [4]. Therefore, conceptually simple tasks like conforming to Carrier Sense Multiple Access (CSMA) **timing constraints become unfeasible.**

2. The second approach is what one calls hardware solutions, where the physical layer is implemented directly on a Field-Programmable Gate Array (FPGA) [5] or very close to the hardware as with Digital Signal Processors (DSPs). Using this architecture, timings are deterministic and delay requirements of modern wireless standards can be met. However, reprogramming the system becomes complex and time consuming. Another drawback is that an implementation of a wireless standard is specific to an SDR platform.

A more detailed discussion about SDR limitations and performance consideration can be found in the following sections (specifically see section 2.1.1)
Given the strong need to support SDR-based solutions especially in the IVC application, it is mandatory to overcome the aforementioned limitations.

The project has two steps. Firstly, I came up with the design following the requirement in 802.11p standard. A system was built and simulated in Matlab to obtain numerical result such as bit error rate plots and verify the functionality of each block in the design.
As sad before, a GNU Radio flowgraph is considered as the reference design from which to start. As it is written in gnu radio,it is composed by preexisting blocks so that it bypass the requirements of understanding Python and C++; [2] Doing so, some opportunities to understand the fundamental communications theory underneath the basic top layer block are lost, as the author just uses a block that someone else wrote. This is why MATLAB was chosen for the first implementation. It is a cross-platform environment and all the blocks are MATLAB scripts with nothing to hide. This allow a better understand of the entire algorithm because it is defined in MATLAB code, with nothing to obfuscate the communications theory.

[1] it is chosen for its growing interest within the electronic engineers community
[2] requirements for a deep understand of GNU Radio, see Section 2.2

Secondly, I rebuilt the whole system in C using the Xilinx HLS flow for simulation and implemented on a FPGA.

Using Matlab, I have built the model of a simple 802.11p wireless communication system limiting the usage of the communication tool box. The input/output plots was obtained to verify the theoretical result and it is compared with an already existing and tested GNU Radio flowgraph [3]. The system is designed base on the requirement of IEEE 802.11p standard. By simulation in Matlab, I can verify the functionality of each block. Base on that, I rebuilt the system in C for its implementation on an FPGA. Starting from the C implementation, the RTL was generated by Vivado HLS flow and imported as IP core in the Vivado environment and connected to the hard core ARM processor via AXI bus . Finally the system was tested, simulated and compare the results with Matlab.
Most of the blocks were constructed base on the Matlab model. Some other complex blocks such as square root, Viterbi Decoder and stream interface are directly instantiated as IP core by Vivado HLS. I made assumption that the channel is known and set to be constant.

1.1 State of the art

802.11p

Although the 802.11p transceiver has been in draft form till July 2010, there are already different industrial solutions available. One is the LinkBird-MX v3 unit produced by NEC [6] which embeds a Linux machine based on a 64 bits MIPS process or working at 266 MHz and which can be configured either for reception or for transmission. Besides, NXP and Cohda Wireless developed a flexible SDR implementation of WAVE (Wireless Access in Vehicular Environments) called MK3 [7]. It includes among others a GPS module and a CAN bus interface and is based on the NXP MARS platform which has been designed for the automotive context. Another solution is the combination of the WSU (Wireless Safety Unit) platform from DENSO and the Openwave Engine developed by BMW [8]. Besides the physical layer implementation of 802.11p, this transceiver supports the required MAC protocols for US, Europe and Japan and includes CAN2.0, Ethernet and a 400 MHz power PC that can process one or two standards in parallel.
Finally, [9] presented a transceiver based on GNURadio which has been combined with USRP2 (Universal Software Radio Peripheral) followed by [10] using the same architecture.

GNURadio on FPGA

The NI USRP RIO[11] is an SDR game changer that provides wireless communications designers an affordable SDR in a user-friendly software environment aka Labview. Recently National instrument announced the availability of the USRP-2945 quad receiver SDR device and the USRP-2944 high performance 2x2 multiple input, multiple output (MIMO) SDR device. Both models deliver a new level of performance and capability to the USRP (Universal Software Radio Peripheral) family. These devices feature the widest frequency ranges, highest bandwidth and best RF performance in the USRP family.
Nutaq [12] proposed a similar hardware solution with commercial IPs and a complete integration with the common commercial and non, SDR software like GNURadio, Labview and Matlab.
Last but not least Ettus research (ER) supports a wide variety of development environments on a proprietary high performance RF hardware; ER USRP platform is the SDR platform of choice for thousands of engineers, scientists and students worldwide for algorithm development, exploration, prototyping and developing next-generation wireless technologies across a wide variety of applications. The Ettus Research hardware comes with an open source framework (RfNoC)[13], which allows developers to integrate an existing IP into more complex systems, involving both software and FPGA components, with little overhead. RFNoC is primarily developed for the Universal Software Radio Peripheral (USRP) family of products, but its source code is freely available.

[3]Wine Project: https://www.wime-project.net/

1.1.1 More about RFNoC

RFNoC (Radio Frequency Network on Chip) is an open source processing tool focused on the development of heterogeneous applications on SDR(Software Defined Radio) devices provided from Ettus Research, generally known as USRP (Universal Software Radio Peripheral).
The concept of NoC refers to a communication system integrated into a single chip used for exchanging data and control between the internal PEs (Processing Engines). The main difference with respect to a normal SoC (System on Chip) is the flexibility in terms of heterogeneous capabilities: in fact, the internal composition of PEs is abstracted from the communication framework

RFNoC Blocks

RFNoC is internally composed of RFNoC blocks, the PEs implementing the DSP algorithms. To reduce the effort of integrating digital signal processing IPs as PEs into an RFNoC block, the design framework provides a pre-cooked interface wrapper. The internals of a RFNoC block are independent from any other block and can be designed with any language and tool that supports AXI stream interfaces, including VHDL, Verilog, and Xilinx Vivado HLS.

NoC Shell

We briefly report the structure of the RFNoC's interface, the NoC Shell. As explained previously, this wrapper represents the common part present in all blocks constituting the RFNoC, independently of the respective internal DSP algorithm. The role of the NoC Shell is to interface the internal PE with the rest of RFNoC, implementing a standard registers and command interface to allow it to be integrated within the RFNoC. It presents a user interface and an interface to the RFNoC AXI stream crossbar. The latest expects a Compressed Header packets as defined in CHDR (Ettus, b). Figure 1.1 shows the structure of the RFNoC Block (Pendlum, 2014). To create this NoC Shell and all the file system structure, Ettus has deployed a tool called RFNoC Modtool. This tool creates a custom GNU Radio OOT (Out of Tree) module as well the necessary files for the RFNoC block development and simulation.

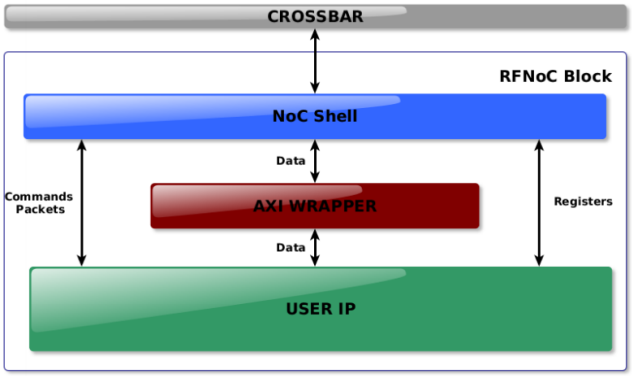


Figure 1.1: RFNoC Block Internal Structure (Ettus, c)

Differences

Substantial differences exists between the aim of this thesis and RFNoC:

1. The approach to deploy Software GNU Radio algorithm to FPGA should **not** be based only on a Network-On-Chip approach. A loosely coupled accelerator should be generated, instead.

2. If the methodology will be automatized, the hypothetical tool should extract the netlist from GNU Radio and synthesize the algorithm directly to FPGA, optimizing the design in a way that take into account the simulation behaviour executed in GNU Radio.

3. The methodology is not as tight coupled in GNU Radio as RFNoC.
RFNoc uses ad hoc blocks under RFNoC library in GNC. Here, the idea is to *add* another implementation for each and every block in gnuradio and uses the original for simulation purpose, the modified for synthesis purpose.

4. Blocks could execute also in software. Not every Blocks should be mapped on FPGA; According to a profiling of the flowgraph and the given constrain, the methodology can choose to map the entire algorithm onto the FPGA (and optimize it) or leave part executed on the hard core ARM processor of the Zynq architecture (or even on the PC with the FPGA

connected to the computer through a high speed link), and accelerate only the critical part of the algorithm.

Chapter 2

Background

> *Software radio is the technique of getting code as close to the antenna as possible. It turns radio hardware problems into software problems.* Eric Bloosom

2.1 Software Defined Radio

A universal SDR structure with the specific software (GNU Radio) and hardware (USRP/2) is given in Figure 2.1

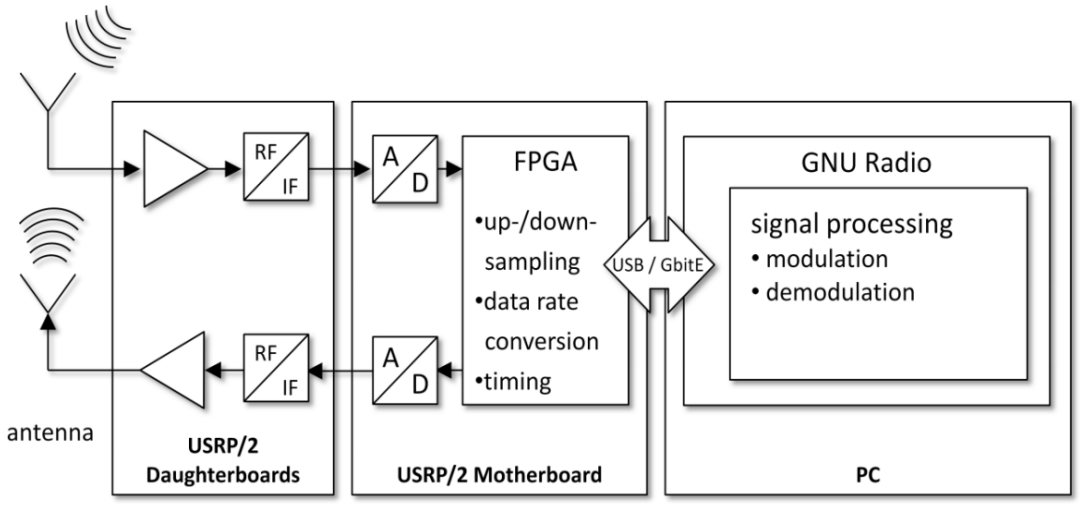


Figure 2.1: Software-Defined Radio block diagram.

In Figure 2.1, the Software-Defined Radio (SDR) structure is divided into three blocks. The left one builds the RF frontend of the hardware which serves as interface to the analog RF domain. In the second block, the intelligence of the hardware part is implemented, forming the interface between the digital and the analog world. In the third block, the whole signal processing is done - fully designed in software.
Getting more detailed, the interface to the analog world is given as mentioned on left side of Figure 2.1. An analog RF signal can be received or transmitted over antennas, or can also be directly connected via SMA connectors to the SMA ports of RF frontend called daughterboards. The upper path (arrow towards the daughterboard) marks the receive path (Rx), the lower path describes the transmit path (Tx). Both paths can operate autonomously. The possible operation frequency range is very modular (from DC to 5.9 GHz), depending on the available daughterboards for USRP/2. Daughterboards form the RF frontend of USRP/2 and are connected to the USRP/2 motherboard. On USRP/2 motherboard, the analog signals are converted to digital samples and mixed down to baseband within the FPGA. Also, a decimation of the sample rate is performed.
Regarding Figure 2.1, data sampled by the FPGA are sent to the host by USB or Gigabit Ethernet respectively what is used – USRP or URSP2. Connected to the host computer (right block in Figure 2.1), the GNU Radio framework controls the further signal processing capabilities. GNU Radio is an open source framework, providing various pre-assembled signal processing blocks for waveform creation and analysis in software radio development.

2.1.1 About SDR performance

SDRs are implemented through employing various types of hardware platforms, such as General Purpose Processors (GPPs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), and Field Programmable Gate Arrays (FPGAs). Each of these platforms is associated with their own set of challenges. Some of these challenges are: utilizing the computational power of the selected hardware platform, keeping the power consumption at a minimum, ease of design process, and cost of tools and equipment.

1. **GPP - General Purpose Processor -**
   One of the first approaches to realizing SDR platforms is using a General Purpose Processor (GPP), or the commonly known generic computer microprocessors such as x86/64 and ARM architectures. Examples of SDR platforms that utilize GPPs include Sora [14], KUAR [15],

and USRP [16].
A GPP is a digital circuit that is clock-driven and register-based, and is capable of processing different functions and operates on data streams represented in binary system [17]. These GPPs can be used for several purposes, making them extremely useful for unlimited number of applications, eliminating the need for building application specific circuits, and thus reducing the overall cost of running applications. GPPs are generally a preferable hardware platform by researchers in academia due to their flexibility, abundance, and ease of programmability, which is one of the main requirements in SDR platforms [18]. In addition, researchers prefer GPPs since they are more familiar with them and their software frameworks, compared to DSPs and FPGAs.
From the performance point of view, GPPs are being enhanced rapidly, credited not only to technological advances in terms of CMOS technology [19], but also to the increase of the average number of instructions processed per clock cycle. The latter is achieved through different means, and in particular, utilizing parallelism within and between processors. This has led to the evolution of multi-core GPPs [20].
Architecturally, the instruction set of GPPs include instructions for different operations such as Arithmetic and Logic Unit (ALU), data transfer, and I/O. A GPP processes these instructions in the sequential order. **Because of sequential processing, GPPs are not convenient for high-throughput computing with real-time requirements** (i.e., high throughput and low latency) [21].For example, **using GNU Radio [22] to implement IEEE 802.11 standard, which requires 20 MHz sampling rate, would be challenging**, since GNU Radio is restricted by the limited processing capabilities of GPPs. This leads to the GPP cores (of the PC attached) to reach saturation and frames become corrupted and discarded. Moreover, wireless protocols require predictable performance in order to guarantee meeting timing constraints. However, conditional branch instructions in GPP's instruction sets lead to out-of-order execution, which makes it unfeasible to achieve predictability. To remedy the limitation of GPPs, researchers have proposed multiple solutions, one of which is the addition of co-processors, such as Graphic Processing Unit (GPU) [23].
When both GPP and GPU are used for a SDR design, data transfer operations between GPP and GPU can be bottlenecks and cause performance loss, especially for meeting realtime requirements [24]. However, there are continuous efforts to reduce or eliminate the time overhead of data transfers by introducing multi-stream scheduling for pipelining of the memory copy tasks. This would ensure no stalls in the pipeline and thus enhancing processing parallelism [25], [26]. Finally, although the processing power of microprocessors is being constantly improved, balancing between sufficient computing power and a specific goal for energy consumption and cost, stays a very difficult task now and in the future. This is true especially with the growing need for more data to be processed and blocks that can handle data processing in parallel.
GPUs are processors specifically designed to handle graphics-related tasks and they efficiently process large blocks of streaming data in parallel. SDR platforms comprised of both GPPs and GPUs are flexible and have a higher level of processing power. However, this results in a lower level of power efficiency (e.g., GPP's power efficiency is 9GFLOPS/W for single-precision, compared to 20GFLOPS/W for GPU [27]). GPUs act as co-processors to GPPs because a GPP is required to act as the control unit and transfer data from external memory. After a transfer is completed, signal processing algorithms are executed by the GPU.
While GPUs are typically used for processing graphics, they are also useful at signal processing algorithms. Over the past few years, theoretical peak performance for GPUs and GPPs for single and double precision processing has been growing [28]. For example, comparing Intel Haswell's 900 GFLOPs [29] with NVIDIA GTX TITAN's 4500 GFLOPS [30] for single precision, it is apparent that GPUs have a computational power that far exceeds their GPP counterparts [28]. Their multi-core architectures and parallel processors are the main attractive features, in addition to their relatively reasonable prices and credit card sizes. These features make them good candidates as co-processors in GPP-based SDRs, where they can play a vital role in accelerating computing-intensive blocks [31]. Another advantage is their power efficiency, which keeps improving with every new model (e.g., it went from 0.5 to 20GFLOPS/W for single precision) [27]. To take full advantage of GPUs, it is a condition that algorithms conform to their architecture. From an architectural perspective, GPUs have a number of advantages that makes them preferable solutions to applications such as

video processing. In particular, GPUs employ a concept called Single Program Multiple Data (SPMD) that allows multiple instruction streams to execute the same program. In addition, due to their multi-threading scheme, data load instructions are more efficient. GPUs also present a high computational density, where cache to ALU ratio is low [32].

ADC Data Length (ms)	Processing Platform of Signal Detection Algorithm		
	GPP Serial Processing (ms)	GPP Parallel Processing (ms)	GPU Parallel Processing (ms)
1	13.487	1.254	0.278
10	135.852	12.842	2.846
100	1384.237	131.026	29.358
1000	13946.218	1324.346	29.358

Table 2.1: Performance of Signal Detection Algorithm on GPP and GPU [31]

In Table 2.1, the authors of [31] confirmed that the signal detection algorithm (which includes intensive FFT computations) shows a faster parallel processing in the case of GPU over GPP, while operating in real-time. This is due to the availability of cuFFT library developed for NVIDIA GPUs for more efficient FFT processing [48]. With regards to the architectural advantage of GPUs, several hundred CUDA cores can perform a single operation at the same time, as opposed to a few cores in the case of multi-core GPPs. Examples of using GPUs alongside GPPs to build SDR platforms is the work in [33], where the authors built a framework on a desktop PC in addition to using a GPU to implement an FM receiver. The authors in [31] studied realtime signal detection using an SDR platform composed of a laptop computer and an NVIDIA Quadro M4000M [30].
When both GPP and GPU are used for a SDR design, data transfer operations between GPP and GPU can be bottlenecks and cause performance loss, especially for meeting realtime requirements [24]. However, there are continuous efforts to reduce or eliminate the time overhead of data transfers by introducing multi-stream scheduling for pipelining of the memory copy tasks. This would ensure no stalls in the pipeline and thus enhancing processing parallelism [25], [26]. Finally, although the processing power of microprocessors is being constantly improved, balancing between sufficient computing power and a specific goal for energy consumption and cost, stays a very difficult task now and in the future. This is true especially with the growing need for more data to be processed and blocks that can handle data processing in parallel.

2. **DSP-based**
The DSP-based solution can be considered as a special case of GPP-based solutions, but due to its popularity and unique processing features, it deserves a separate discussion. An example of DSP-based SDR is the Atomix platform [19] which utilizes TI TMS320C6670 DSP [34].
DSP is a particular type of microprocessor that is optimized to process digital signals [35]. To help understand how DSPs are distinguished from GPPs, we should first note that both are capable of implementing and processing complex arithmetic tasks [36]. Tasks like modulation/demodulation, filtering, and encoding/decoding are commonly and frequently used in applications that include speech recognition, image processing, and communication systems. DSPs, however, implement them more quickly and efficiently due to their architecture (e.g., RISC-like architecture, parallel processing) which is specifically optimized to handle arithmetic operations, especially multiplications. Since DSPs are capable of delivering high performance with lower power, they are better candidates for SDR deployment [37], compared to GPPs. Examples of DSPs especially designed for SDR platforms are TI TMS320C6657 and TMS320C6655. These DSPs are both equipped with hardware accelerators for complex functions like the Viterbi and Turbo Decoders [38].

3. **FPGA-Based**
Another approach towards realizing SDRs is to use a programmable hardware such as FPGAs. Example of FPGAbased SDR platforms are Airblue [39], Xilinx Zynq-based implementation of IEEE 802.11ah [40], and [41] that used the same FPGA board to implement a complete communication system with channel coding.

An FPGA is an array of programmable logic blocks, such as general logic, memory, and multiplier blocks, that are surrounded by a routing fabric, which is also programmable [42]. This circuit has the capability of implementing any design or function, with ease of updating it. Although FPGAs consume more power and occupy more area than ASICs, the programmability feature is the reason behind their increasing adoption in a wide range of applications. Furthermore, when the reconfiguration delay is in the order of milliseconds, the SDR can switch between different modes and protocols seamlessly [43]. Another major difference is that, ASIC fabrication is expensive (at least a few tens of thousands of dollars) and requires a few months, whereas FPGAs can be quickly reprogrammed, and their cost is within a few tens to a few thousands of dollars, at most. The low end product cycle, along with attractive hardware processing advantages, such as high speed performance, low power consumption, and portability, compared to processors such as GPPs and DSPs, present FPGAs as contenders that offer the best of both worlds [42]. In a study by [44], the authors compared the performance of Xilinx FPGAs [45] against 16-core GPPs. The calculation of peak performance for GPPs was performed through multiplying the number of floating point function units on each core by the number of cores and by the clock frequency. For FPGAs, performance is calculated through picking a configuration, adding up the Lookup Tables (LUTs), flip-flops, and DSP slices needed, then multiplying them by the appropriate clock frequency. The authors calculated the theoretical peaks for 64- bit floating point arithmetic and showed that Xilinx Virtex-7 FPGA is about 4.2 times faster than a 16-core GPP. This can been seen in Figure 2. Even with a one-to-one adder/multiplier configuration, the V7-2000T achieved 345.35GFLOPS, which is better than a 16-core GPP. From Intel [29], Stratix 10 FPGAs can achieve a 10 Tera FLOPS peak floating point performance [46]. This is due to the fixed architecture of the GPP, where not all functional units can be fully utilized, and the inherent parallelism of FPGAs and their dynamic architecture. In addition, despite having lower clock frequencies (up to 300MHz), FPGAs can achieve better performances due to their architectures which allow higher levels of parallelism through custom design [47]. In a study by [48], the authors compared the performance and power efficiency of FPGAs to that of GPPs and GPUs using double-precision floating point matrixvector multiplication. The results show that FPGAs are capable of outperforming the other platforms, while maintaining their flexibility. In another study by [32], the authors thoroughly analyzed and compared FPGAs against GPUs via the implementations of various algorithms. The authors concluded that although both architectures support a high level of parallelism, which is crucial to signal processing applications, FPGAs offer a larger increase in parallelism, whereas GPUs have a fixed parallelism due to their data path and memory system. Over the past decade, FPGAs have significantly advanced and become more powerful computationally, and now exist in many different versions such as Xilinx Kintex UltraScale [45] and Intel Arria 10 [29] [49], [50]. In addition, the availability of various toolsets gave FPGAs an advantage by making them more accessible. This is supported by the availability of compilers that have the capability of generating Register-transfer Level (RTL) code, such as Verilog and VHDL, that is needed to run on FPGAs, from high-level programming languages. This process is typically referred to as **High Level Synthesis (HLS)** -it will be described in the following section-. Examples of such compilers include HDL Coder [51] for MATLAB code [52] and Xilinx HLS [53] or Altera Nios II C2H compiler [54] for C, C++, and SystemC.

HLS allows software engineers to design and implement applications, such as SDRs, on FPGAs using a familiar programming language to code, namely C, C++, SystemC, and MATLAB, without the need to possess a prior rich knowledge about the target hardware architecture. These compilers can also be used to speed up or accelerate parts of the software code running on a GPP or DSP that are causing slowdowns or setbacks to the overall performance.

Further, FPGAs can achieve high performance while still consuming less energy than previously discussed processors [55] (e.g., Intel Stratix 10 FPGA can achieve up to 100 GFLOPS/W [56], compared to 23 GFLOPS/W for NVIDIA GeForce GTX 980 Ti [30]). In addition, power dissipation can be further lowered through the implementation of several techniques discussed in [43]. These techniques can be at a system, device, and/or architecture level, such as clock gating and glitch reduction.

4. **Hybrid design**

The fourth approach towards realizing SDRs is the hybrid approach, where both hardware

and software-based techniques are combined into one platform. This is commonly referred to as the co-design or hybrid approach. Examples of SDRs that adopted the co-design approach include WARP [21] and CODIPHY [57].

Hardware/software co-design as a concept has been around for over a decade, and it has evolved at a faster rate in the past few years due to an increasing interest in solving integrated circuit design problems with a new and different approach. Even with GPPs becoming more powerful than ever, and with multi-core designs, it is clear that in order to achieve higher performance and realize applications that demand real-time processing, designers had to shift attention to new design schemes that utilize hardware solutions, namely, FPGAs and ASICs [58], [59]. Co-design indicates the use of hardware design methodology, represented by the FPGA fabric, and software methodology, represented by processors. As more applications, such as automotive, communication, and medical, grow in complexity and size, it has become a common practice to design systems that integrate both software (like firmware and operating system) and hardware [95]. This has been made feasible in the recent years thanks to the advances in high-level synthesis and developing tools that not only have the capability to produce efficient RTL from software codes, but also define the interface between the both sides. The industry has realized the huge market for co-design, and provided various SoC boards, that in addition to the FPGA fabric, contain multiple processors. For example, the Xilinx Zynq board [45] includes an FPGA fabric as well as two ARM Cortex-A9 processors [96].

There are other reasons that make co-design even more interesting including, faster time-to-market, lower power consumption (when optimized for this), flexibility, and higher processing speeds, as typically hardware in these systems is used as an acceleration to software bottlenecks [97]. Adopting the co-design methodology in essence is a matter of partitioning the system into synthesizable hardware and executable software blocks. This process depends on a strict criteria that is developed by the designer [98], [99]. The authors in [60] and [61] discuss their partitioning methodologies and present the process of making the proper architectural decisions. Common methods typically provide useful information to the designer to help make the best decision of what to implement in hardware and what to keep in software. This information can include possible speedups, communication overheads, data dependencies, and locality and regularity of computations [61].

In Table 2.2, we provide a high-level comparison between three major design approaches as a guideline for designers towards choosing the method that best meets their application specifications. In this comparison, we focus on the features that are important to SDR design. However, we do not make assumptions on what the best approach is and believe it is the developer's responsibility to make the best judgment depending on the application area. Please note that in this table we did not include GPUs, as they typically act as co-processors to GPPs and their addition generally improves performance. We also did not include co-design since it combines GPPs with FPGAs.

	GPP	**DSP**	**FPGA**
Computation	*Fixed Arithmetic Engines*	*Fixed Arithmetic Engines*	*User Configurable Logic*
Execution	*Sequential*	*Partially Parallel*	*Highly Parallel*
Throughput	*Low*	*Medium*	*High*
Data Rate	*Low*	*Medium*	*High*
Data Width	*Limited by Bus Width*	*Limited by Bus Width*	*High*
Programmability	*Easy*	*Easy*	*Moderate*
Complex Algorithms	*Easy*	*Easy*	*Moderate*
I/O	*Dedicated Ports*	*Dedicated Ports*	*User Configurable Ports*
Cost	*Moderate*	*Low*	*Moderate*
Power Efficienty	*Low*	*Moderate*	*High*
Form Factor	*Large*	*Medium*	*Small*

Table 2.2: Comparison of SDR design approaches

As Table 2.2 shows, while GPPs are easy to program and extremely flexible, they lack the power to meet specifications in real-time and are very inefficient in terms of power. To increase their performance, multiple cores with similar instruction sets are included in the same GPP platform to exploit parallelism and perform more operations per clock cycle. However, hardware replication (i.e., adding more cores to GPPs) may not necessarily translate to a higher performance. GPUs tackle this by offering the same control logic for several functional units. The sequential portion of the code runs on the GPP, which can be optimized on multi-core GPPs, while the computationally intensive portion runs on a several-hundred-core GPU, where the cores operate in parallel. Another example of a customized processor is DSP, which performs significantly better than GPPs, while at the same time maintaining the ease-of-use feature that GPPs possess, making them very attractive options. They are also more power efficient and better fit for signal processing applications. On the other hand, they are more expensive, which is the main trade-off. Finally, FPGAs combine the flexibility of processors and efficiency of hardware. FPGAs can achieve a high level of parallelism through dynamic reconfiguration, while yielding better power efficiency [27]. FPGAs are typically more suitable for fixed-point arithmetic, like signal processing tasks, but in the recent years their floating-point performance has increased significantly [48], [62]. However, the designers are expected to know a lot more about the hardware, which is sometimes a deterring feature.
In a comparative analysis by [108], authors studied the performance and energy efficiency of GPUs and FPGAs using a number of benchmarks in terms of targeted applications, complexity, and data type. The authors concluded that GPUs perform better for streaming applications, whereas FPGAs are more suitable for applications that employ intensive FFT computations, due to their ability to handle non-sequential memory accesses in a faster and more energy efficient manner. Similarly, in [27], the authors review and report the sustainable performance and energy efficiency for different applications. One of their findings related to SDRs is that FPGAs should be used for signal processing without floating point, confirming aforementioned results. In addition, the authors in [63] report that GPUs are ten times faster than FPGAs with regards to FFT processing, while authors in [48] demonstrate that the power efficiency of FPGAs is always better than GPUs for matrix operations.

2.1.2 About SDR software Enviroment

Referring back to Figure 2.1, the boundary between the analog and digital worlds for a communication system is located at the analog-to-digital converter (ADC) and the digital-to-analog converter (DAC), where signal information is translated between a continuous signal and a discrete set of signal sample values. Typically, the radio can be configured to select center frequency, sampling rate, bandwidth, and other parameters to transmit and receive signals of interest. This leaves the modulation and demodulation techniques, which are developed using a two-step development process.

1. Develop, tune, and optimize the modulation and demodulation algorithms for a specific sample rate, bandwidth, and environment. This is normally done on a host PC, where debugging and visualization is much easier. At this phase of development, the modulation and demodulation of the RFFEs are performed on a host, providing great flexibility to experiment and test algorithm ideas.

2. Take the above algorithm, which may be implemented in a high-level language in floating point, and code it in a production worthy environment, making production trade-offs of a product's size, weight, power and cost in mind. These platforms become truly software-defined when the onboard hardware and embedded processor are programmed to perform application-specific digital communications and signal processing functions.

The first step requires a convenient mechanism to capture data for signal analysis and development of algorithms that process those signals. This makes it vitally important to have efficient and reliable PC-based software to develop and test the data transmission and digital signal processing functions in a wireless communications system. One software environment that meets this requirement is MATLAB from MathWorks. MATLAB is a technical computing environment and programming language, allowing ease of use development and excellent visualization mechanisms. An additional product, Communications Systems Toolbox, adds physical layer algorithms, channel models, reference models, and connectivity to SDR hardware to transmit and receive live signals.

MATLAB is cross platform (Windows, Linux, MAC) offering support for many of the popular commercial radio front-ends. Using MATLAB enables an incremental and iterative development workflow for SDR consisting of:

- Algorithm development and design validation with link-level simulations;
- Algorithm validation with live signals using connection to commercially available SDR hardware.

MathWorks also offers Simulink, which is an environment for real-world system simulation and automatic code generation for hardware and software implementation. It allows the radio developer to continue to the second stage of production development. These capabilities of Simulink provide a path to production:

- Development and validation of a hardware-accurate model;
- Implementation of a prototype on SDR hardware using automatic HDL and C code generation;
- Verification of the prototype versus the validated model;
- Deployment of the implementation to production SDR hardware.

Although Simulink will largely be ignored in this thesis project, being able to have a single environment from concept to production is very powerful and should not be overlooked for those who are trying to make a real production radio.

Another SDR software architecture is the popular open-source GNU Radio software [9], which is a free software (as in freedom) development toolkit that provides signal processing blocks to implement software-defined radios and signal processing systems. It can be used with external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used in hobbyist, academic, and commercial environments to support both wireless communications research and real-world radio systems. In GNU Radio, a variety of C++ libraries modeling different digital communications and digital signal processing algorithms are integrated together using Python and SWIG (a software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python). These libraries are produced by the open-source community and freely shared with everyone.

2.2 GNU Radio

GNU Radio is a software toolkit designed to allow users to create SDR implementations.
It provides mechanisms and tools to create customized and also a great amount of source code modules that can be used as a part of a customized radio as well as stand-alone examples.
GNU Radio is used by a large community of hobbyists, academic researchers and commercial companies, specially for prototyping and testing possible implementations. The software in GNU Radio is divided in modules. These modules can be divided in two large groups. The modules that take care of the signal processing needed in the system are programmed in C++. These signal processing modules can be signal filters, equalizers, FFT modules and so on. The other group of modules include the software needed to interconnect these signal processing modules and configure them according to out needs. These last ones are programmed in Python and they act as some kind of glue that makes the whole system one unit [3]. The possibilities that Python provides for configuration without the need of compiling every time a change is made in a parameter are very convenient for quick re-configurations and tests. GNU Radio modules are able to operate with infinite streams of data of a certain type.
The most common types of data that we will use are complex, short and float. Many GNU Radio applications keep running forever, when the streams of data are infinite. If the stream has a finite number of bytes the application will finish once the data has been consumed out of the system.
In a GNU Radio application the software modules can also be differentiated in three classes: Source modules, which provide a stream or signal into the system. Examples of source modules are file sources, that get data from a file and insert in into the system, or random data generators, that generate and output a stream of data according to some criteria established by us. The second

group of modules are sink modules. They are the contrary of source modules. They receive an a stream of data and consume it. Some examples for these modules are the trivial null sink, which consumes the signal without doing anything, file sinks that insert the data from the stream into a file or even graphical spectrum analyzers, that convert the signal received into spectral data and present it to the user in a graphical way as a real spectrum analyzer would do. The last kind of modules are the modules with both input and output ports. They receive streams from their input ports, convert it into a different stream by applying a conversion or a filter and output the result through their output ports. Examples of these modules are band pass filters, Fourier transformation modules, operators or data converters.We are able to create GNU Radio applications by joining blocks. The union of these blocks creates a graph.

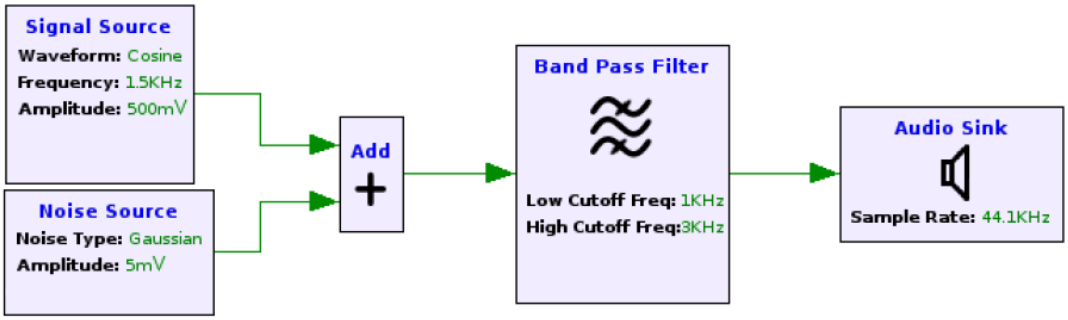


Figure 2.2: Graph representation of a GNU Radio application.

We can see a simple example of one of these graphs in Figure 2.2. In the figure we can see various modules with their parameters and the connections between them.
The graph representation is important to understand how GNU Radio executes each module in the system. All GNU Radio applications have one thing in common. All applications have a top block class that contains an initialization method that will create the instances on the needed modules, build the graph and initialize the parameters of these modules. This initialization method will be called from the main method when we run the application and only after it finishes its execution the system created will be ready to be used. When the application is running GNU Radio will execute the code of the blocks sequentially according to the graph. In the case of the application in Figure 2.2 we would need a Python script to create instances of all these modules, initialize all the parameters that we can see and join them with the connect to build the graph.

GNU Radio Companion (GRC)

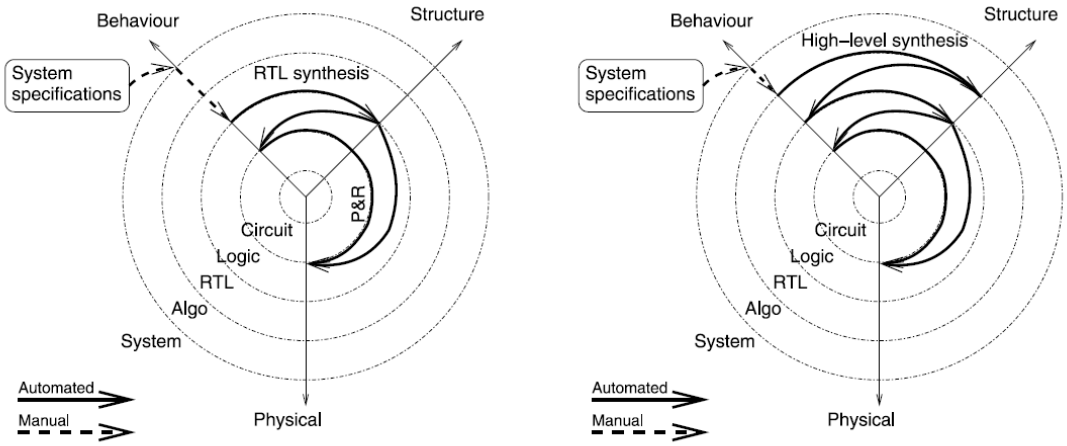
A very useful extension that GNU Radio provides is called GRC and it provides a graphical interface that allows its users to easily create GNU Radio applications. GRC has a list of available modules that can be inserted in the application by only clicking on them. These modules can also be configured, and GRC even tells you if the configured parameters are correct. Then, the modules can be connected together also very easily. Afterwards, GRC will build the needed python code that will run the application. The use of GRC has both advantages and disadvantages, but if it is used correctly it can be a good tool to save a lot of time and to avoid unnecessary mistakes. The main advantages it offers are the simple setup of a system, thanks to its graphical and easy to understand interface,the real time verification of the configured parameters, as it shows the parameters that are not correctly configured and where they are, making it very easy to locate configuration problems.
Another advantage is the easy way of inserting and testing new modules in the system; New modules can be added to the system very easily in order to test their behaviour. However GRC also has some disadvantages. GRC only works with the modules and parameters and offers almost no place for customization of a module out of the configuration of the parameters.
During the implementation of a module GRC is not recommended, as every time the module is modified one should take care of also modifying the files that allow this module to be controlled from GRC.
We can say that GRC is a very interesting tool in some environments but it should be left out in other environments. In an educational environment GRC would allow students to control and

create GNU Radio applications with a very short learning period, and with very successful results, as students would be able to understand all the stages of the system, control the different parameters and see the results in a very short period of time. However, in a research context, where the researcher is adding and customizing modules in all levels GRC would not be recommended, as it adds overhead time to the research process, specially when the researcher already knows GNU Radio well. GRC would be only useful to build test applications and then use the Python code automatically generated by GRC as the starting point for the development of more complex applications. Custom made modules can be added to GRC by creating an XML file that describes the module.

2.3 High-Level synthesis

The hardware design process has evolved significantly over the years. When the circuits were small, hardware designers could more easily specify every transistor, how they were wired together, and their physical layout. Everything was done manually. As our ability to manufacture more transistors increased, hardware designers began to rely on automated design tools to help them in the process of creating the circuits. These tools gradually become more and more sophisticated and allowed hardware designers to work at higher levels of abstraction and thus become more efficient. Rather than specify the layout of every transistor, a hardware designer could instead specify digital circuits and have electronic design automation (EDA) tools automatically translate these more abstract specifications into a physical layout.
Since the hardware complexity, at the time, continue increasing at an exponential rate, hardware designers are forced to move to even more abstract hardware programming languages. Register-transfer level (RTL) was one step in abstraction, enabling a designer to simply specify the registers and the operations performed on those registers, without considering how the registers and operations are eventually implementation. EDA tools can translate RTL specifications into a digital circuit model and then subsequently into the detailed specification for a device that implements the digital circuit. This specification might be the files necessary to manufacture a custom device or might be the files necessary to program an off-the-shelf device, such as an field-programmable gate array (FPGA). Ultimately, the combination of these abstractions enables designers to build extraordinarily complex systems without getting lost in the details of how they are implemented. A non-technical perspective on the value of these abstractions can be found in [64].

High-level synthesis (HLS) is a new step in the design flow of a digital electronic circuit, moving the design effort to higher abstraction levels. Its place in the design flow can be best situated using Gajski and Kuhn's Y-chart. This chart (Fig. 1a) has 3 axes that represent different views on the design: behavior (what the (sub)circuit does), structure (how the circuit is built together, e.g. a netlist or schematic) and geometry (how the circuit is physically implemented or what it looks like, e.g. a layout). There are also 5 concentric circles representing abstraction levels: circuit (transistors, voltages, circuit equations), logical (ones and zeros, boolean equations), register transfer (RTL: registers, operators, HDL), algorithmic (functions, loops, programming languages) and system level (system specifications, programming and natural languages).



(a) RTL design flow (b) HLS design flow

Figure 2.3: High-level synthesis in the Gasjki-Kuhn Y-chart.

In the design flow that has become mainstream in the past 2 decades (Fig. 2.3a), the hardware designer would manually refine the behavioral system specifications down to the RT level. From that point, RTL synthesis and place and route complete the design flow. As both the RTL design and design tools are made by humans, verification is necessary to match the behavior of the design with the specifications at various stages of the design flow and remove discrepancies when necessary. Nowadays this design flow is increasingly being challenged. Moore's law states that an increasing amount of functionality can be integrated on a single chip. Someone has to design all this functionality though, and it is not economically nor practically viable to steadily increase the size of design teams or design time. This means that somehow the design productivity has to be improved. Given the fast growing transistor count, it may be acceptable to trade some transistors (or chip area) for an improved design time.

High-level synthesis improves design productivity by automating the refinement from the algorithmic level to RTL. As can be seen on the Y-chart (Fig. 2.3b), the transition from the specifications to the start of the automated design flow becomes smaller now. HLS generates an RTL design from a function written in a programming language like C, C++ or Matlab. High-level synthesis takes over a number of tasks from the designer. After analysis of the source code, resource allocation is done, i.e. determining what types of operators and memory elements are needed and how many. Next, during scheduling, each operation from the source code is assigned to a certain time slot (clock cycle or cstep). Finally, in resource binding, operations and data elements from the source code are assigned to specific operators and memory elements. High-level synthesis also takes care of interface synthesis, the generation of an appropriate interface, consisting of data and control signals, between the generated circuit and its periphery (e.g. a memory interface).

Several advantages arise from the use of HLS in the design flow. First of all, the amount of code to be written by designers is reduced dramatically, which saves time and reduces the risk of mistakes. HLS can optimize a design by tweaking source code and tool options, opening up opportunities for extensive design space exploration. Verification time, which nowadays exceeds design time, is reduced a lot because the HLS tool can, in addition to the design itself, generate testbenches, thereby reusing test data that was used to validate the source code.

This is particularly relevant for the design of FPGA based embedded systems. Moving to a higher abstraction level makes it possible to handle the increasing design complexity while removing the need to hand-code the hardware architecture and timing into the algorithm.

Hardware accelerators for embedded software may be generated with minimal effort. HLS and FPGAs make a perfect combination for rapid prototyping and a fast time to market.

2.4 OFDM(Orthogonal Frequency Division Multiplex) & IEEE 802.11p

OFDM - Orthogonal Frequency Division Multiplex -

Orthogonal frequency-division multiplexing is a multi-carrier digital modulation system that has been employed as a modulation system for terrestrial digital broadcasting. Compared with single-carrier digital modulation, OFDM can lengthen the symbol period while maintaining the same error-rate characteristics and band efficiency. It can also add a redundant signal period called a guard interval. For these reasons, OFDM features little deterioration of transmission characteristics with respect to multi-path distortion, the main type of disturbance on a terrestrial transmission path.

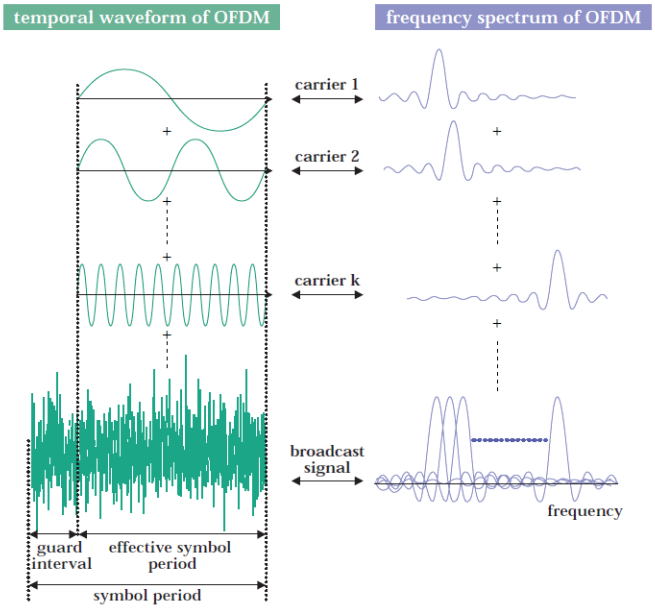


Figure 2.4: OFDM time spectrum and frequency spectrum

The OFDM signal multiplexes multiple digitally modulated waves that are mutually orthogonal in a certain signal interval. Referring to Figure 1, if, for base-band frequencies, we let carrier-1 be the base wave and arrange subsequent carriers at integral multiples of 2, 3, and so on of the base frequency, then any set of these carriers will be mutually orthogonal within one period of the basic wave. Varying the amplitude and phase of each of these carriers by digital modulation and then adding them together (frequency multiplexing them) results in an OFDM signal. In addition, performing a Fourier transform on this OFDM signal in one base-wave period makes it possible to uncover the amplitude and phase information of each carrier. This operation is none other than OFDM demodulation.

Digital modulation of individual carriers is normally performed using QPSK or QAM, and particular modulation systems are referred to as QPSK-OFDM, 64QAM-OFDM, etc. The QPSK-OFDM and 16QAM-OFDM systems are used on transmission paths characterized by severe disturbances such as in mobile communications where automobiles and other objects come into play. On the other hand, for fixed reception by an antenna installed on a roof as in ordinary television reception, 64QAMOFDM is used so that as much data as possible can be transmitted within a limited frequency bandwidth.Transmit symbols in OFDM consist of effective symbols and guard intervals. Data allocated to the carriers are transformed collectively by an inverse discrete Fourier transform into symbols each within the effective symbol period Tu.

A guard interval is formed for each effective symbol period by taking a section of waveform data from the end of the symbol in question and simply attaching it to the front of the symbol, as shown in Figure 2.5. Transmit symbols of period Tu+Tg are obtained in this way.

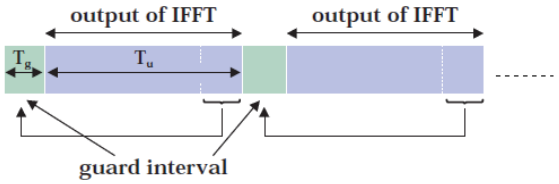


Figure 2.5: Attaching a guard interval

The OFDM carrier interval is the inverse of the base-wave period (effective symbol period) shown in Figure 2.4.

$$\Delta f = \frac{1}{T_u}$$

Each OFDM carrier has a small spectral width because of the low-speed modulation, and the OFDM transmission spectrum that groups together these individual spectrum takes on a nearly rectangular shape, as shown in Figure 2.6.

If the number of carriers K is large (several hundred or more), the occupied bandwidth B can be approximated as follows.

$$B = K\Delta f$$

Letting C(l, k) denote transmit data corresponding to symbol number l and carrier number k, the OFDM transmit signal S(t) can be expressed as follow.

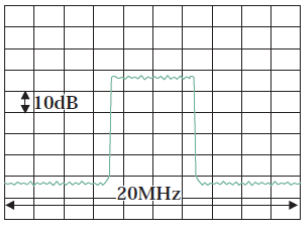


Figure 2.6: OFDM transmission spectrum

$$S8t) = Ree^{2j\pi f_c t} \sum_{l=-\inf}^{\inf} \sum_{k=0}^{K-1} C(l,k)\Psi(l,k,t)$$

where:

$$\Psi(l,k,t) = \begin{cases} e^{j2\pi \frac{k-K_c}{T_u}(t-Tg-lTs} & lTs < t < (l+1)Ts \\ 0 & t < lTs, (l+1)Ts \leq t \end{cases}$$

$k \longrightarrow$ carrier number (carrier at lower end of the band is 0

$l \longrightarrow$ Symbol number

$K \longrightarrow$ Number of carriers

$Ts \longrightarrow$ Length of symbol period (Tg+ Tu)

$Tg \longrightarrow$ Length of guard-interval period

$Tu \longrightarrow$ Length of effective symbol period

$fc \longrightarrow$ Center frequency of RF signal

$Kc \longrightarrow$ Carrier number corresponding to center frequency of RF signal

$C(l,k) \longrightarrow$ Complex transmit data corresponding to symbol number l and carrier number k

$S(t) \longrightarrow$ RF signal

IEEE 802.11p

The need for communications services is constantly growing. Nowadays mobile devices are present almost everywhere and wireless communications is an area of great commercial interest. This fact has motivated a considerable amount of contemporary research. The IEEE 802.11 standard (802.11) is one of the most widely used standards for wireless communication and is updated frequently to meet future requirements. Importantly although it specifies the lower layer parameters in the OSI model such as modulation schemes and frame formats, vendors are free to choose their preferred method for channel estimation.
The p amendment to the 802.11 standard is intended to be used for wireless access in vehicular environments. Such environments, where the transmitter and/or the receiver can be moving at high speeds, introduce the challenge of a time-selective channel. In addition the outdoor environment also increases the frequency-selectivity compared to an indoor channel.
A physical layer simulator based on the 802.11p standard has been developed at Chalmers as part of a recent research initiative.

Systems that utilize the 802.11p standard operate in orthogonal frequency division multiplexing (OFDM) mode with 64 subcarriers. However, only NST = 52 of the 64 subcarriers are actually used for transmitting useful information. Out of these 52 subcarriers, NSP = 4 are used for transmitting pilot symbols, i.e. symbols with pre-determined values, and NSD = 48 subcarriers are used for transmitting data. An illustration of an OFDM symbol in this format is included in Figure 2.7.

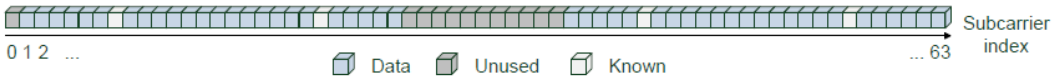


Figure 2.7: The subcarriers in an 802.11p OFDM symbol.

Generally, an 802.11p OFDM frame consists of several concatenated OFDM symbols. In Figure 2.8 the general structure of an OFDM frame is presented. The duration of an OFDM symbol is $T_{FFT} = 6.4\mu s$ with a cyclic prefix denoted as GI of duration $T_{GI} = 1.6\mu s$ which is added for each OFDM symbol. GI2 denotes a cyclic prefix with duration twice that of a GI, i.e. $T_{GI2} = 1.6 \cdot 2 = 3.2\mu s$.

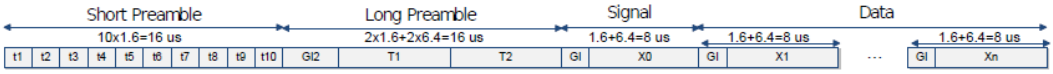


Figure 2.8: Structure of an 802.11p OFDM frame.

An OFDM frame begins with a short preamble that consists of 10 identical short training symbols (t1, t2, ... , t10), each with duration $1.6\mu s$. The short training symbols are mainly used for signal detection, coarse frequency offset estimation and timing synchronization. A long preamble follows, consisting of two identical long training symbols (T1, T2), each with duration $6.4\mu s$. The long training symbols are mainly used for fine frequency offset estimation and channel estimation. It should be noted that since the 10 short training symbols in the short preamble are identical they also function as cyclic prefixes for each other, i.e., t1 is a cyclic prefix for t2, which in turn is a cyclic prefix for t3 and so on. This is also true for the T1 and T2 symbols in the long preamble. Following the training sequences in an OFDM frame is a SIGNAL OFDM symbol which contains information about the length of the OFDM frame and the type of modulation and coding rate used for the remainder of the OFDM frame. Then, the next OFDM symbol contains a scrambling sequence as well as actual data; subsequent symbols only contain data.

Figure 2.9 shows the pilot symbols and data symbols. Note that no distinction is made between the SIGNAL OFDM symbol and the actual data OFDM symbols since there is no difference between them from a channel estimation point of view. The symbols at frequency indexes $-26 \longrightarrow -1$ in Figure 2.9 correspond to indexes $38 \rightarrow 63$ in Figure 2.8 and, similarly, frequency indexes $1 \rightarrow 26$ in Figure 2.9 correspond to index $1 \rightarrow 26$ in Figure 2.8. The two OFDM symbols in Figure 2.9 with symbol indexes (T1, T2) correspond to the long preamble and each consists of 52 pilot subcarriers. The SIGNAL OFDM symbol corresponds to symbol index 0 and the Data symbols correspond to symbol indexes $1, ..., Nframe$. As mentioned, the SIGNAL and data OFDM symbols, each have 4 pilot subcarriers — the remaining 48 subcarriers consist of actual data.
The 802.11p amendment supports 4 different modulation techniques, namely BPSK, QPSK, 16-QAM, 64-QAM. It also makes use of error-correction and interleaving. With respect to error-correction, convolutional coding is used with the coding rates R = 1/2, 2/3 or 3/4. Combining different coding rates with different modulation techniques results in 8 different data rates, as summarized in Table 1. Only the data rates of 3, 6 and 12 Mbit/s are mandatory in 802.11p.

Finally, the channel spacing is 10 MHz, subcarrier spacing is 10/64 MHz, and the difference between the lowest and highest subcarrier is (53/64)*10 MHz. Figure 2.10 describes a simplified block diagram for a transmitter and receiver that utilize 802.11p.

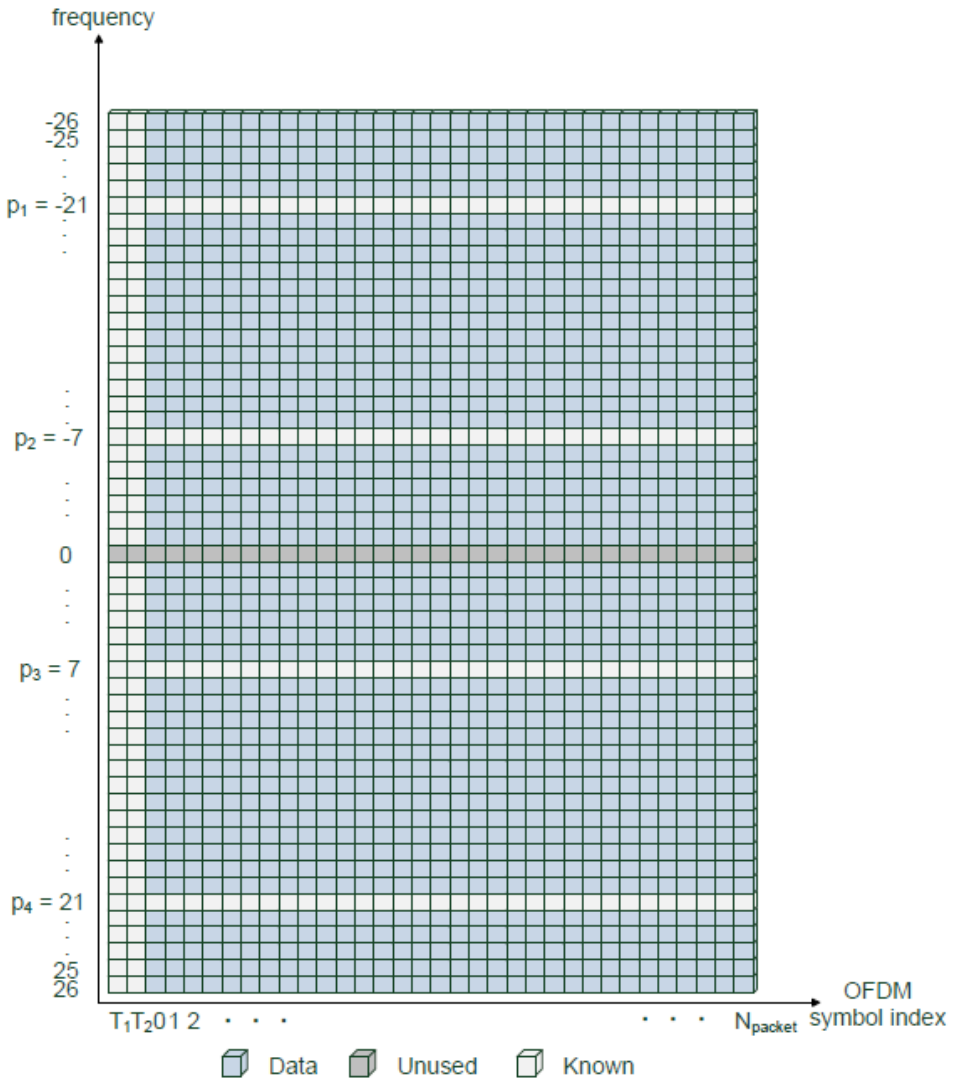


Figure 2.9: Structure of the pilot symbols in an 802.11p OFDM frame.

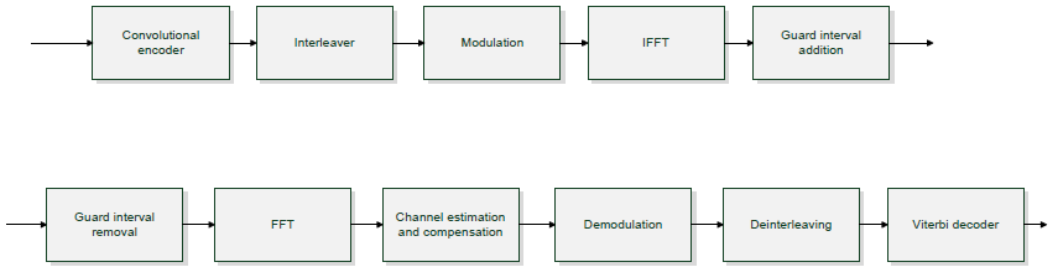


Figure 2.10: Simplified base-band structure of an 802.11p transmitter and receiver.

Chapter 3

802.11p accelerator

> *If you were plowing a field which would you rather use two strong oxen or 1024 chickens.*
> -Seymour Cray, Father of the supercomputer

3.1 Overview

To study the physical layer, researchers often rely on simulators that model IEEE 802.11p at signal level, considering channel effects on the electromagnetic waveform [65], [66] [67] [68] [69], or on hardware prototypes for experiments in the lab and on the road [69, 70, 71]. Both approaches are, however, limited to their domain only, i.e., it is not possible to switch from simulations to experiments. As a consequence, there is no easy way to validate and test findings from simulation studies through experiments or vice versa.

To overcome this limitation, in 2013, a group of students from the University of Paderborn, Germany, released the source code of an SDR based Orthogonal Frequency Division Multiplexing (OFDM) receiver [10]. It supports channel bandwidths up to 20MHz and do not rely on additional FPGA code for working.
This GNU Radio based implementation of an IEEE 802.11 transciever is an ongoing development, and is used as a starting point in this project. Our main goal is to improve the performance of the system so as to advance in the support of high transfer data rates.

SDRs are programmable radios that provide access to all data down to the physical waveform, allowing to implement all signal processing in software. This aspect makes them particularly well suited to build early prototype transceivers and to experiment with novel signal processing algorithms.
With their SDR transceiver, signal processing is implemented in software on a normal PC, making it particularly easy to use, modify, and debug. While we also investigate standard compliant channel access for broadcast transmissions, the main application domain of the transceiver is clearly the physical layer. In contrast to existing approaches, the major advantage is that the software implementation is decoupled from the radio frontend. This and the fact that the transceiver works with cheap, open, and widely used radio hardware makes the system accessible to fellow researchers. Since the physical layer of our SDR transceiver is implemented completely in software, new physical layer concepts can easily be implemented and tested. Furthermore, this also allows to use our transceiver for simulations too, enabling studies of IEEE 802.11p and novel variants in reproducible configurations. The ability to use the same tool for both simulations and experiments allows for a seamless switch between theory and practice and presents a big advantage of the proposed approach.
The code, open source, fully available on GitHub [1] and largely tested by the scientific community, for the aforementioned reasons is taken as reference implementation.

3.2 Reference Top layer Module

This thesis relies on the GNURadio IEEE 802.11 implementation introduced by Bloessl et al. [10]. The code contains both transmitter and receiver implementations that work with commercial 802.11a/g/p devices. Only the receiver is taken into consideration while the transmitter is only used to generate useful data in order to validate the receiver.

[1] https://github.com/bastibl/gr-ieee802-11

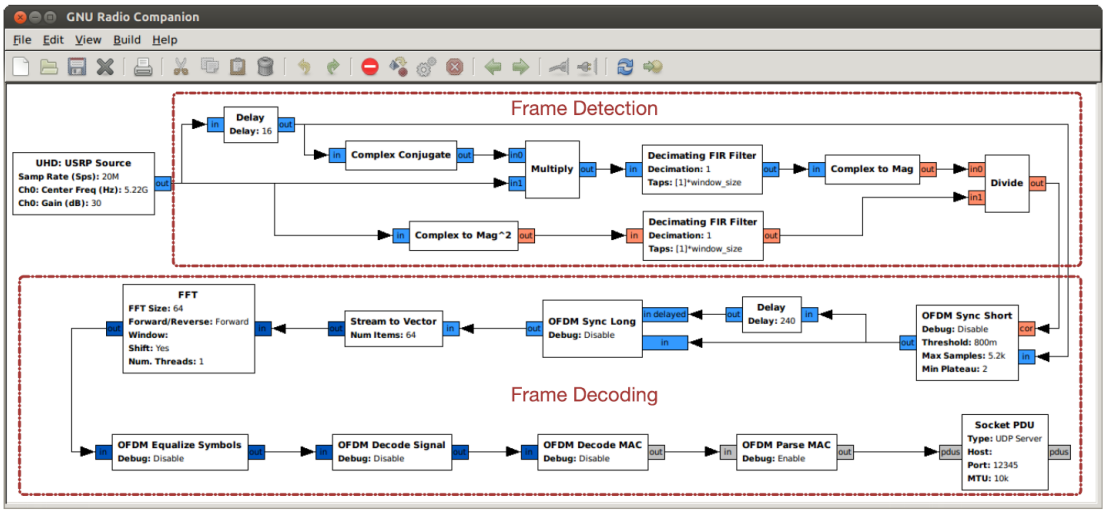


Figure 3.1: Overview of the blocks comprising the OFDM receiver in GNU Radio Companion.

Similar to LabView and Simulink, GNURadio is, at its core, a signal processing system for data streams. This architecture is very natural for SDRs, where the hardware produces a constant stream of complex baseband samples. To process the data, the sample stream is piped through signal processing blocks that implement actual functionality. With GNURadio, such signal processing system is described by a flow graph, a data structure that defines how blocks are parameterized and connected.

Figure 3.1 presents a graphical representation of the blocks that compose the receiver and the connections between them, as seen in GNU Radio Companion. While the detailed functionality is not important at this point, the figure gives an idea of GNURadio's stream paradigm; the signal processing blocks are visualized by the boxes, while the sample streams are depicted by the arrows that connect them. As flow graphs can get rather complex, GNURadio allows encapsulating functionality in hierarchical blocks that can be used as building blocks in other flow graphs.
Going deeper, the receiver is divided into two functional parts:

- The first part, depicted in the top half, is responsible for frame detection.
- The second part, shown in the bottom half, is responsible for decoding the frame.

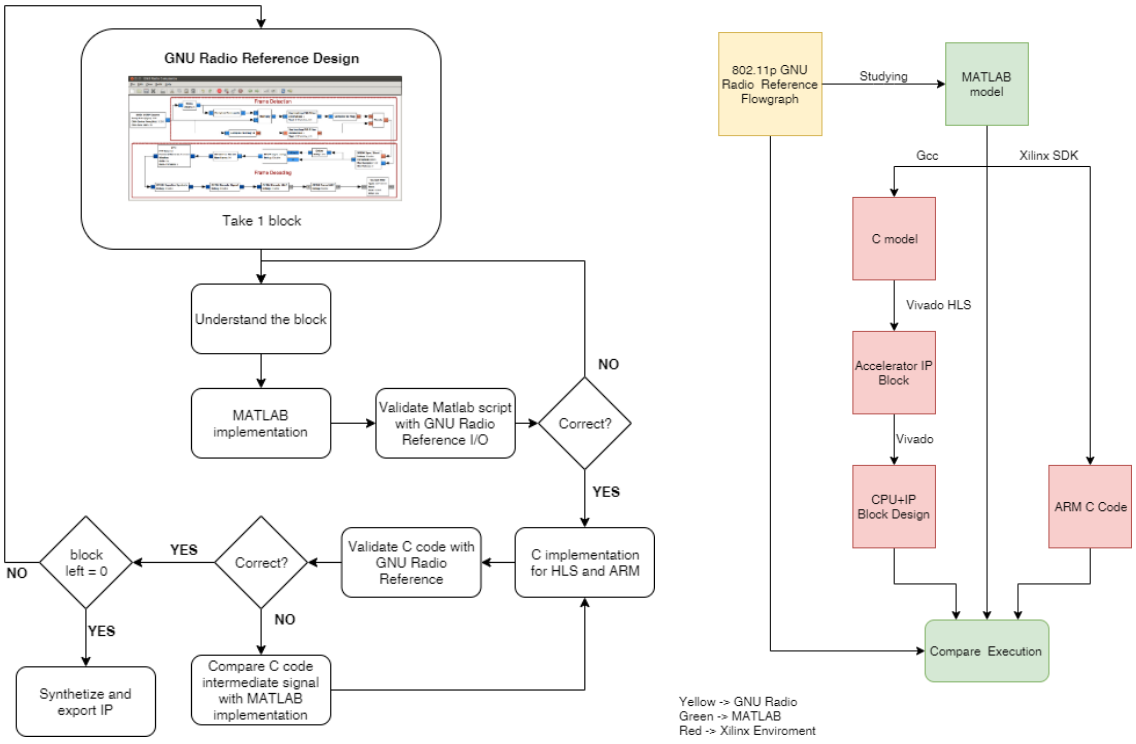
All the initial blocks, up to the OFDM Sync Short, are responsible for the calculation of the normalized autocorrelation of the incoming signal. When this calculation reaches a predetermined threshold, a new incoming frame has been detected. The block that checks whether the threshold has been reached is the OFDM Sync Short. If it detects the start of a frame, then it pipes a fixed number of samples into the following blocks in the pipeline, otherwise, samples are dropped. The next block in the processing chain, OFDM Sync Long, applies frequency offset correction to the signal and symbol alignment.
Next is the Stream to Vector block, which converts a single stream of complex numbers into a vectorized stream. Note that this is a prerequisite for using the FFT block, which converts the signal from the time domain to the frequency domain.
After the signal transformation by the FFT the OFDM Equalize Symbols block is executed. This block performs channel estimations and phase offset corrections. It also removes the guard and pilot carriers, leaving only the 48 subcarriers with actual data.
The OFDM Decode Signal follows, which is responsible of decoding the signal field of the physical layer frame, and to set the proper modulation scheme and coding rate for the DATA field.
Finally, the last block in the processing chain, the OFDM Decode MAC, is responsible for implementing the demodulation, the deinterleaving, the convolutional decoding and the descrambling of the data field of the physical layer frame. The output of this block is the link layer frame that was originally transmitted.

3.3 Working Methodology

The project methodology used in this thesis is graphically summarized by Figure 3.2b.
A *divide at impera* approach is used for the 802.11p algorithm. The entire standard is divided into small blocks; each of them performs a simple task as shown in the GNU Radio Top layer in Section 3.2. The same approach is used for the design of the Hardware accelerator and the designing procedure is shown in details in Fig.3.2a.

For every block belonging to the 802.11p flowgraph (shown in the reference design 3.1):

1. It is studied the overall 802.11p algorithm in detail (Section 2.4),
2. it is studied what the role of the single block is by considering one at the time; it is deeply understood its function and how the considered algorithm uses its functionality,
3. It is studied the source code of the reference design (by reverse engineering GNU Radio 802.11p library source code).
4. A first implementation was made in MATLAB,
5. the block was tested in a new flowgraph alone by feeding known input and saving the output; they are next compared to the MATLAB implementation.
6. The *C* implementation is written and tested through GCC compiler, the code is imported in Vivado HLS,
7. It was synthesized with and without optimization and it was exported as IP core,
8. the generated IP was imported in Vivado, connected to the on-board hard ARM processor through an AXI bus,
9. it was deployed to an FPGA by downloading the generated bit-stream.
10. Finally, the accelerator and the ARM software are executed with some known data; results are collected from the RAM and they are compared with the corresponding GNU Radio reference design and the MATLAB implementation.



(a) Block Design process.

(b) Project Design methodology

Figure 3.2: Design Methodologies

3.3.1 Hardware Top Layer Module

```
void top( fx_pt1 *out, fx_pt *wave, const unsigned size)
{
  fx_pt history[CHUNK];
    fx_pt _inbuff1[CHUNK];
  fx_pt1 _outbuff[CHUNK];

go:
  for(unsigned i=0;i<size/CHUNK; i++)
  {
     load(_inbuff1, wave,i);
     compute(_inbuff1,_outbuff);
     store( _outbuff, out,i);
  } //end - for every chunk


} // main end
```

Listing 3.1: Top level module

Hardware Top layer Module is divided into 3 main macro function (Listing 3.1) in order to instantiate a dataflow region and implement a ping pong buffer (see appendix B.2; *Dataflow* and *ping pong* are explained in detail in section 3.9.3 considering the practical example of the FFT to better clarify the concepts).

- **Load**: The function retrieves from memory the data to be processed. Address of the memory address is passed as parameter (**wave*) at the invocation of the accelerator by the processor. If the range of data exceed the maximum private accelerator memory, many bursts are performed. If not, the free positions are filled with 0s.
  The load function uses the so called ping-pong buffer; It consist of a behaviour like this: *_inbuff1* is replicated 2 times; when the first buffer is full and ready to be processed, data are transferred to the next function (compute) while the load function instantiates another burst and fill the 2^{nd} 'hidden' buffer. When the compute function ends its execution, the 2^{nd} 'hidden' buffer become active and it starts the computation with that data while now the 1^{st} become the 'hidden' one. Due to this overlapping, if the latency of the loading, computing and storing function are balanced, it is possible to 'hide' the load and store latency being visible only the maximum between the tree function.

```
    void load(fx_pt _inbuff1[CHUNK],fx_pt *wave,unsigned chunk){
    load_data:
  for(unsigned i=0;i<CHUNK;i++)
    _inbuff1[i]=wave[CHUNK*chunk+i];
    }

```

Listing 3.2: Load function

- **Compute**: The compute function is implemented with the same way as the reference design (3.1). Instead of graphical blocks, we found the function.
  The first part is purely as close as much as the original GNU Radio code. The second, instead, a change of flow is made by preferring a streaming interface. Every function is explained in the following sections.

```
    void compute(fx_pt *_inbuff1, fx_pt1 *_outbuff){
    fx_pt _tmpbuff[CHUNK];
  fx_pt _tmp1buff[CHUNK];
  fx_pt _tmp2buff[CHUNK];
  fx_pt correlation_complex[CHUNK];
  fx_pt1 correlation[CHUNK];
  fx_pt1 _signPower[CHUNK];
  fx_pt1 _signPowerAveraged[CHUNK];
  fx_pt1 correlazione[CHUNK];

  delay(_tmpbuff,_inbuff1);
  coniugate(_tmp1buff,_tmpbuff);
  complex_mult(_tmp2buff,_tmp1buff,_inbuff1);
  firc(correlation_complex,_tmp2buff);
```

```
15  magnitude(correlation,correlation_complex);
16  signal_power(_signPower,_inbuff1);
17  fir(_signPowerAveraged,_signPower,coeff_mvgAvg);
18  division(correlazione,correlation,_signPowerAveraged);
19
20  // frame decoding
21  fx_pt d_frame[CHUNK];
22  fx_pt d_frame_d[CHUNK];
23  hls::stream<fx_pt> frame;
24  hls::stream<fx_pt> frame_d;
25  hls::stream<fx_pt> frame_p;
26  hls::stream<fx_pt> d_frame_long;
27  fx_pt prova[CHUNK];
28
29  sync_short(_tmpbuff,correlation_complex,correlazione,frame);
30  delay320(frame_p, frame_d, frame);
31  sync_long(frame_p, frame_d, d_frame_long );
32
33  // ************* FFT *****************
34  fx_pt1 fft_ar_r[CHUNK];
35  fx_pt1 fft_ar_i[CHUNK];
36  static unsigned p_next=0;
37  fx_pt fftSample;
38  unsigned nc_fft=p_next;
39
40  //adjust the samples
41  static fx_pt1 mem_fft_r[CHUNK];
42  static fx_pt1 mem_fft_i[CHUNK];
43  accoda: for(unsigned i=0;i<p_next;i++)
44  {
45    fft_ar_r[i] = mem_fft_r[i];
46    fft_ar_i[i] = mem_fft_i[i];
47  }
48
49  s2p: for(unsigned i=p_next;i<CHUNK;i++)
50  {
51      if ( d_frame_long.read_nb(fftSample) )
52      {
53        fft_ar_r[i] = (fx_pt1) fftSample.real();
54        fft_ar_i[i] = (fx_pt1) fftSample.imag();
55        nc_fft++;
56
57      } else {
58        fft_ar_r[i] = 0;
59        fft_ar_i[i] = 0;
60      }
61  }
62
63  fft: for(unsigned i=0;i<nc_fft/64; i++)
64  {
65    fft_hls( (fft_ar_r+64*i), (fft_ar_i+64*i),true );
66  }
67
68  unsigned p=(nc_fft/64);
69  p_next=0;
70
71  remain: for(unsigned i=p*64; i<nc_fft; i++)
72  {
73      mem_fft_r[p_next]=fft_ar_r[i];
74      mem_fft_i[p_next]=fft_ar_i[i];
75      fft_ar_r[i]=0; // only for sim purpose
76      fft_ar_i[i]=0; // only for sim purpose
77      p_next++;
78  }
79  // equalize
80  fx_pt tobeEqualize[CHUNK];
81  fx_pt equalized[CHUNK];
82  for(unsigned i=0;i<10*64;i++)
83    tobeEqualize[i]=fx_pt(fft_ar_r[i],fft_ar_i[i]);
84  equalize(tobeEqualize,equalized);
85
86  //decode signal
87  hls::stream< ap_uint<1> > decoded_bit;
```

```
88
89    decode_signal(equalized,decoded_bit);
90    ap_uint<1> bit;
91    for(unsigned i=0;i<CHUNK;i++)
92      if(decoded_bit.read_nb(bit))
93        _outbuff[i]=(unsigned) bit;
94      else
95        _outbuff[i]=0;
96      }
97
```

Listing 3.3: Compute function

- **Store**: It uses a similar approach as in *load* function. but in the opposite way: from private accelerator memory to the global RAM in the location (**out*) passed as parameter at the invocation of the accelerator by the processor.

```
1     void store(fx_pt1 _outbuff[CHUNK],fx_pt1 *out,unsigned chunk) {
2     store_data:
3   for(unsigned i=0;i<CHUNK; i++)
4           out[CHUNK*chunk+i]=_outbuff[i];
5 }
6
```

Listing 3.4: Store function

3.4 Project structure

The project is divided into different folder; each one contains files used in different tool.

- **inc:** This is the folder where the project header files are located.
- **src:** This is the folder where the source code is located. This is the folder used to develop the algorithm and test through *gcc* compiler.
- **syn**: This is the folder where the Xilinx Vivado HLS project is located and it is used as storage for temporary files for the synthesis as well as any automatic script. Reports are also found under this folder.
- **sys**: This is the folder where the Xilinx Vivado project is located. This folder is used to contain any intermidiate files used by the logic synthesis.
- **tb**: This is the folder where source and header files used to test the behaviour of the architecture. Any files with raw data is contained under this path.
- **sdk**: This is the folder where source code and any temporary files (i.e. BSP, etc.) are located. This is the main path where files are used for the development and execution of the ARM software part.

3.5 Matlab implementation

As mentioned in the 3.3 section, a Matlab implementation has been made for every block.
Top layer function are reported in Listing 3.5.

```
1  coarse_freq_adj=1;
2  fine_freq_adj=1;
3  display=0;
4
5  debug_correlation=0;
6  debug_sync_short=0;
7  debug_sync_long=0;
8  debug_fft=0;
9  debug_equalizer=0;
10
```

```
%% import data
import_data

%% frame detection
frame_detection

%% frequency offset correction
sync_short

%% sync long part
sync_long1

%% Furier Transform
furier_t

%% equalizer
equalizer

%% decode signal
decode_signal
```

Listing 3.5: Matlab Top Function

The variable with the prefix *debug_* is used to better understand and locate errors in the programming phase. In other word, if 1, the script loads the input and output data of the related block from GNU Radio so that the Matlab implementation can be verify to be fully compliant with the GNU Radio reference design.
The code related to the *debug_correlation* is shown as an example in listing 3.6.

```
if debug_correlation==1
    gr_ref_corr=bin2double('./data/gr_ref_correlation_old.txt');
    l=min([length(gr_ref_corr); length(cor) ]);
    l=5500;
    plot( gr_ref_corr(1:l),'b' );
    hold on
    plot(cor(1:l),'r');
    err=max(abs(gr_ref_corr(1:l)-cor(1:l) ));
    if err<1e-3
       str=['      -) correlation -> Error= ' num2str(err), '  OK    ' ];
    else
      str=['      -) correlation -> Error= ' num2str(err),  '  FAIL  ' ];
    end

    disp(str);

end
```

Listing 3.6: effect of debug_correlation vaiable in frame_detection function

Intermediate variables are stored and passed from one function to another by the Matlab workspace(it is like a global variable in C).

Some useful script has been carefully engineered in order to easy the integration between Matlab, Xilinx environment and GNU Radio and makes type conversion straightforward.

- **Binary data (GNU Radio ouput) to double (matlab)**

```
function b = bin2double(file,varagin)

fin=file;
if nargin==1
    type='single';
    complex=0;
elseif nargin==2
    type='single';
    complex=1;
end

fid=fopen(fin);
b=fread(fid,inf,type);
if complex==1
    b=[b(1:2:end-1), b(2:2:end)];
```

```
end
fclose(fid);

end

```

- **double (matlab) to Binary data (GNU Radio)**

```
fout=fopen('out.bin','w');
a=real(X);
b=imag(X);


for i=1:length(a)
    for k=1:2
        if k==1
            fwrite(fout,a(i),'single');
        elseif k==2
            fwrite(fout,b(i),'single');
        end
    end
end

fclose(fout);

```

- **Binary (GNU Radio) to N_Word bit fixed point (Xilinx)**

```
import_data

N_Sample=5*4096;

%d=fi(d,1,N_Word,N_Word-N_integer);
N_Word=32;
N_integer=6;


d_fx_real=int32(real(d(1:N_Sample))* 2^((N_Word-N_integer)));
d_fx_imag=int32(imag(d(1:N_Sample))* 2^((N_Word-N_integer)));


fileID=fopen('real.bin','w');
fwrite(fileID,d_fx_real,'int32');
fclose(fileID);


fileID=fopen('img.bin','w');
fwrite(fileID,d_fx_imag,'int32');
fclose(fileID);

```

3.6 Packet Detection

802.11 OFDM packets start with a short PLCP Preamble sequence to help the receiver detect the beginning of the packet. The short preamble duration is 8 us. At 20 MSPS sampling rate, it contains 10 repeating sequence of 16 I/Q samples, or 160 samples in total. The short preamble also helps the receiver for coarse frequency offset correction , which will be discussed separately in Frequency Offset Correction.

Power Trigger
The core idea of detecting the short preamble is to utilize its repeating nature by calculating the auto correlation metric. But before that, we need to make sure we are trying to detect short preamble from "meaningful" signals. One example of "un-meaningful" signal is constant power levels, whose auto correlation metric is also very high (nearly 1) but obviously does not represent packet beginning.
It takes the I/Q samples as input and asserts the trigger signal during a potential packet activity. Optionally, it can be configured to skip the first certain number of samples before detecting a power trigger. This is useful to skip the spurious signals during the intimal hardware stabilization phase.

3.6.1 Short Preamble Detection

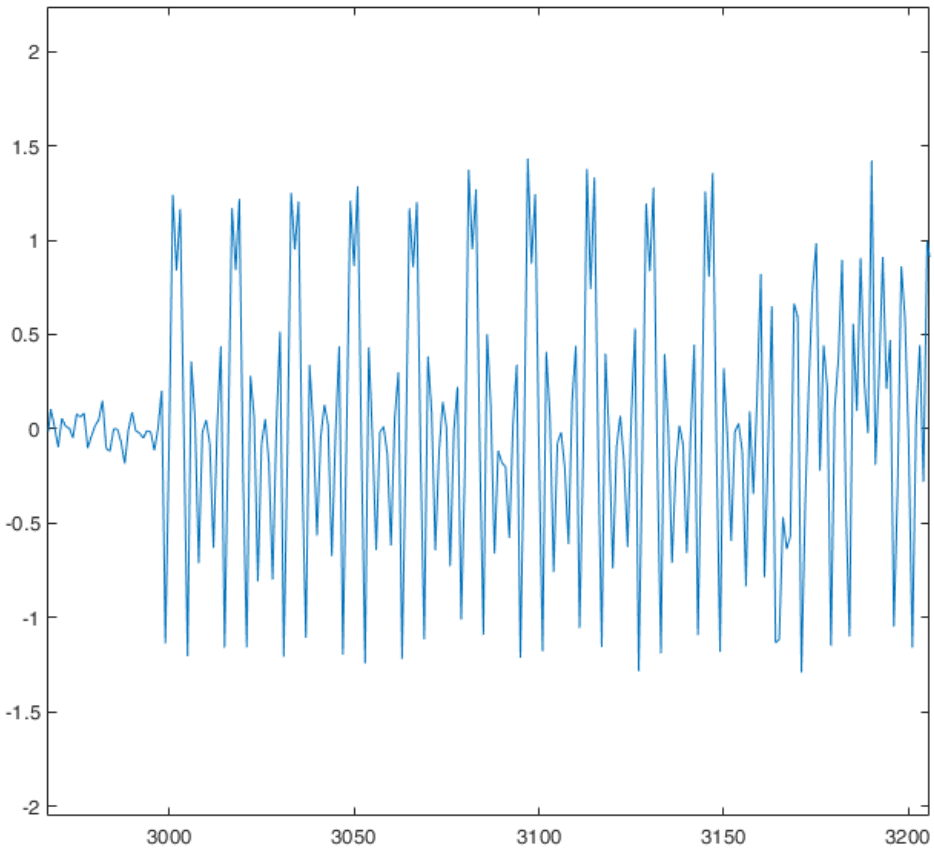


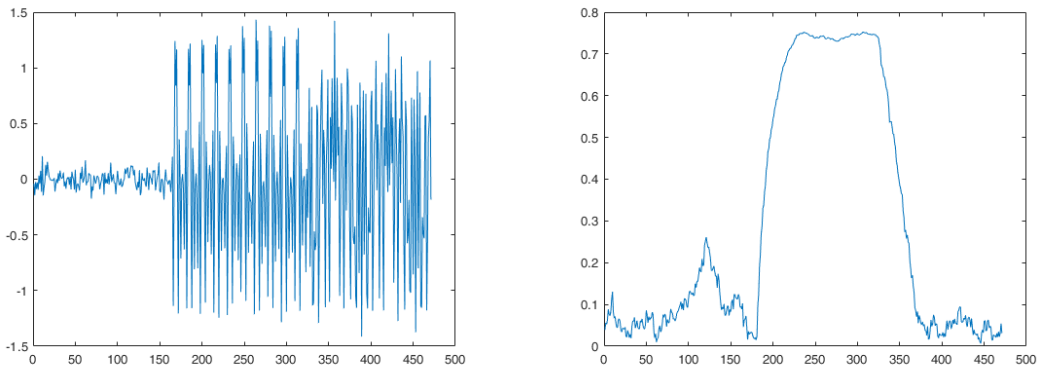
Figure 3.3: In-Phase of Short Preamble.

3.3 shows the in-phase of the beginning of a packet. Some repeating patterns can clearly be seen. We can utilize this characteristic and calculate the auto correlation metric of incoming signals to

detect such pattern:

$$corr[i] = \frac{\left\| \sum_{i=0}^{N} S[i] * \overline{S[i+16]} \right\|}{\sum_{i=0}^{N} S[i] * \overline{S[i]}}$$

where $S[i]$ is the $\langle I, Q \rangle$ sample expressed as a complex number, and $\overline{S[i]}$ is its conjugate, N is the correlation window size. The correlation reaches 1 if the incoming signal is repeating itself every 16 samples. If the correlation stays high for certain number of continuous samples, then a short preamble can be declared.



(a) Real Part of the I-Q signal

(b) Auto Correlation of (a)

Figure 3.4: Auto Correlation of the Short Preamble samples (N=48).

Figure 3.3 and 3.4 is obtained by plotting signal **d** and **cor** inside the frame_detection Matlab script.

```
... %execute decode script until frame\_detection function...
figure
plot(real(d(2834:3304)))
figure
plot(real(cor(2834:3304)))
```

3.4b shows the auto correlation value of the samples in (a). We can see that the correlation value is almost 1 during the short preamble period, but drops quickly after that. We can also see that for the very first 20 samples or so, the correlation value is also very high. This is because the silence also repeats itself (at arbitrary interval)!

3.6.2 Implementation

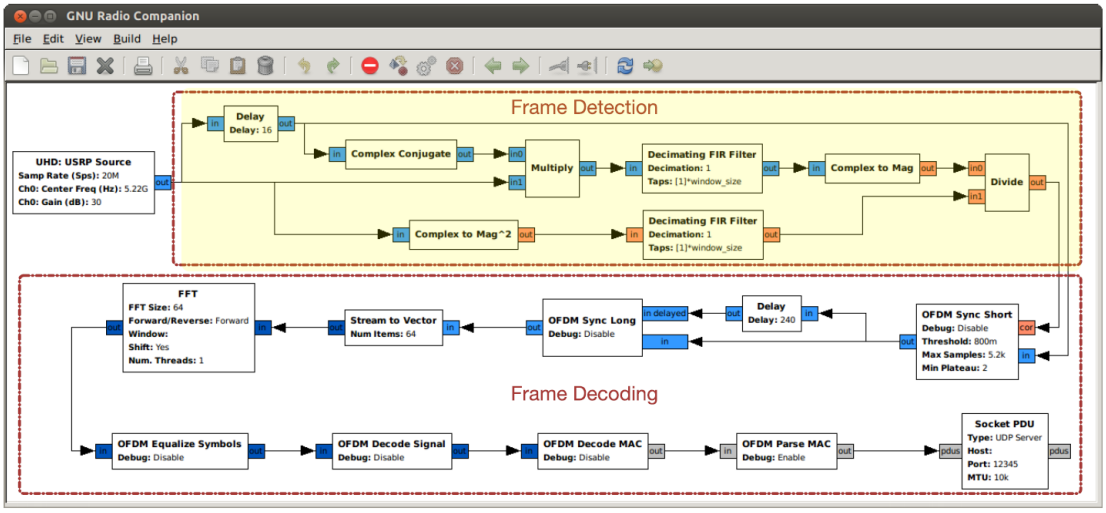


Figure 3.5: GNU Radio Blocks in charge of frame detection.

From 3.9 it can be seen that the calculation of the auto-correlation coefficient is split in eight blocks and realize all operations with standard operations in GNU Radio. All the involved blocks make use of the already mentioned VOLK library.

Delay

Arithmetic operation

Complex Arithmetic library include the following operation:

- **Complex Coniugate** : Returns the conjugate of the complex number x.

```
void coniugate(fx_pt out[CHUNK], fx_pt data[CHUNK])
{
  conj:for (unsigned i=0; i<CHUNK;i++)
     out[i]= conj( data[i] );
}
```

Listing 3.7: conjugate function

- **Signal Power** :

```
void signal_power(fx_pt1 out[CHUNK], fx_pt data[CHUNK])
{
  fx_pt1 a,b;
  power:for (unsigned i=0; i<CHUNK;i++)
  {
    a=data[i].real();
    b=data[i].imag();
    out[i]= a*a+b*b;
  }
}
```

Listing 3.8: signbal power function

- **Magnitude** :

```
void magnitude(fx_pt1 out[CHUNK], fx_pt data[CHUNK])
{
  fx_pt1_ext tmp,a,b;

  magn:for (unsigned i=0; i<CHUNK;i++)
  {
    a= (fx_pt1_ext) data[i].real();
```

```
    b=(fx_pt1_ext) data[i].imag();
    tmp= a*a+b*b;
    out[i]= (fx_pt1) hls::sqrt(tmp);
    //printf("a=%f; b=%f; -> %f\n",a.to_float(),b.to_float(), out[i].to_float
    () );
  }
}
```

Listing 3.9: magnitude function

- **Complex Multiplication** :

```
#define N_ext 64
#define int_ext 16

void complex_mult(fx_pt out[CHUNK], fx_pt data1[CHUNK], fx_pt data2[CHUNK])
{
    mult:for(unsigned j=0;j<CHUNK;j++)
    {

      complex<ap_fixed<N_ext,int_ext> > out_ext = complex<ap_fixed<N_ext,
    int_ext> > (data1[j])* complex<ap_fixed<N_ext,int_ext> >(data2[j]);
      out[j]= fx_pt(out_ext);
    }
}
```

Listing 3.10: multiplication function

- **Absolute Value** : The function *abs_ c* calculate the absolute value (also called the modulus) of the input *data* defined as:

$$a + ib = \sqrt{a^2 + b^2}$$

```
fx_pt1 abs_c(fx_pt data)
{

  fx_pt1_ext tmp,a,b;

  magn:for (unsigned i=0; i<CHUNK;i++)
  {
    a= (fx_pt1_ext) data.real();
    b=(fx_pt1_ext) data.imag();
    tmp= a*a+b*b;
    return (fx_pt1) hls::sqrt(tmp);
  }
}
```

Listing 3.11: ABS function

Finite Impulse Response (FIR) filter

The function takes two arguments, an input sample vector *input_ sample*, and the output sample vector *output*. The coefficients for the filter are avoided since we need a *moving average* filter where all coefficient are 1s (no multiplication is needed so the architecture has been simplified). The code is written as a streaming function. It receives one sample at a time, and therefore it must store the previous samples. Since this is an *COEFF_ LENGHTc* tap filter, we must keep the previous *COEFF_ LENGHTc-1* samples. This is the purpose of the buffer[] array. This array is declared static since the data must be persistent across multiple calls to the function.

There are 2 main operation to be performed in this filter: The first part shifts the data through the *buffer* array. The second part performs the (multiply and) accumulate operations in order to calculate the output sample.
Loop fission takes these two operations and implements each of them in their own loop. While it may not intuitively seem like a good idea, it allows us to perform optimizations separately on each loop. This can be advantageous especially in cases when the resulting optimizations on the split loops are different. Loop fission alone often does not provide a more efficient hardware implementation. However, it allows each of the loops to be optimized independently, which could lead to better results than optimizing the single, original for loop. The reverse is also true; merging two

(or more) for loops into one for loop may yield the best results.

The first for loop (with the label data_shift) shifts the values up through the buffer array. The loop iterates from largest value (COEFF_LENGHTc-1) to the smallest value (i = 1). By unrolling this loop, we can create a data path that executes a number of these shift operations in parallel.

Loop unrolling can increase the overall performance provided that we have the ability to execute some (or all) of the statements in parallel. In the unrolled code, each iteration requires that we read two values from the shift reg array; and we write two values to the same array. Thus, if we wish to execute both statements in parallel, we must be able to perform two read operations and two write operations from the shift reg array in the same cycle.
Assume that we store the *buffer* array in one BRAM, and that BRAM has two read ports and one write port. Thus we can perform two read operations in one cycle. But we must sequentialize the write operations across two consecutive cycles. There are ways to execute these two statements in one cycle. For example, we could store all of the values of the shift reg array in separate registers. It is possible to read and write to each individual register on every cycle. In this case, we can perform both of the statements in this unrolled for loop in one cycle. You can tell the Vivado HLS tool to put all of the values in the shift reg array into registers using the directive **pragma HLS array partition variable=shift reg complete**.

```
void firc(fx_pt output[CHUNK],fx_pt input_sample[CHUNK])
{

  static fx_pt_ext buffer[COEFF_LENGTHc];
  static bool init=true;

  if(init) {
init: for(unsigned i=0; i<COEFF_LENGTHc;i++)
      buffer[i]=(fx_pt_ext)(0,0);
    init=false;
  }

loop_filter_data:
    for(unsigned n=0;n<CHUNK;n++)
    {
      fx_pt_ext _output[COEFF_LENGTHc];
      fx_pt_ext _output_(0,0);
data_shift:
    for(int i=COEFF_LENGTHc-1;i>=0;i--) {
        if(i==0) buffer[0]=(fx_pt_ext)(input_sample[n]);
        else     buffer[i]=buffer[i-1];
      }

    for(int i=COEFF_LENGTHc-1;i>=0;i--)
      _output_ +=buffer[i];

    output[n]= fx_pt(_output_); //fx_pt( _output_/fx_pt_ext(32,0) );

    } //end for every sample

} //end function
```

Listing 3.12: FIR filter function

Matlab Implementation

```
%upper branch
d_d=[zeros(16,1); d(1:end-16)];
c_c_d=conj(d_d);
mul=d.*c_c_d ;
%filter
windowSize = 48;
%b = (1/windowSize)*ones(1,windowSize);
b = (1)*ones(1,windowSize);
a = 1;
s_x_sc=filter(b,a,mul);
upper_branch=abs(s_x_sc);

```

```
%lower branch
po=(abs(d)).^2;
windowSize = 64;
%b = (1/windowSize)*ones(1,windowSize);
b = (1)*ones(1,windowSize);
a = 1;
lower_branch=abs(filter(b,a,po));

cor=upper_branch./lower_branch;
```

3.7 Frequency Offset Correction

The synchronization issue is inevitable in all signal transmission systems. In digital transmission, though the bit streams are inherently discrete-time signals, all physical media, such as radio channels or transmission lines, are continuous-time in nature. In wireless communications, most physical transmission media are inefficient in transmitting base-band signals. Consequently, the digital base-band transmitted signal has to be converted to a continuous-time waveform and then modulated by a higher-frequency carrier signal.
After the modulated signal passes through the physical media, several inverse processing procedures, including sampling/digitization and demodulation, are applied.

This paper [72] explains why frequency offset occurs and how to correct it. There are two types of frequency offsets:

- The **Carrier Frequency Offset (CFO)** : it occurs when the local oscillator signal for down conversion in the receiver does not synchronize with the carrier signal contained in the received signal.
  This phenomenon can be attributed to two factors: frequency mismatch in the transmitter and the receiver oscillators, and the Doppler effect as the transmitter and/or the receiver is moving. When this occurs, the received signal will be shifted in frequency, as shown in Figure 3.6.
  The symptom of this offset is a phase rotation of incoming I/Q samples (time domain). It can be corrected by the help of short preamble (Coarse) and long preamble (Fine).

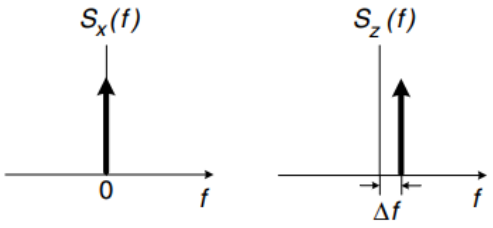


Figure 3.6: The received base-band signal spectrum is shifted by the CFO(Δf) with respect to the transmitted signal.

- The **Sampling Frequency Offset (SFO)**: it is quite similar to the CFO, as they both originate from oscillator mismatch and the Doppler effect. When oscillators with mismatched frequencies are used to drive the sampling clocks of the DAC in the transmitter and the ADC in the receiver, SCO may occur. Figure 3.7(a) illustrates an example in which the sampling clock mismatch causes the received waveform to be sampled at time instants that are progressively skewing.
  Motion between the transmitter and the receiver effectively makes the signal waveform to contract or expand in time. Therefore, even without sampling clock mismatch, the sampled waveform at the receiver can still be suffering errors in sampling times, as in the previous case. Figure 3.7(b) shows a case in which the Doppler effect causes the received waveform to expand and thus sample-time error occurs, even though the ADC and DAC are synchronously clocked.
  The symptom of this offset is a phase rotation of constellation points after FFT (frequency domain). It can be corrected using the pilot sub-carriers in each OFDM symbols.
  However, considering the data generated by the transmitter, it was observed that this contribution is small; the fact that we are using BPSK modulation makes useless the correction of SFO because it will not lead to any practical advantage;it is therefore not included in this analysis.

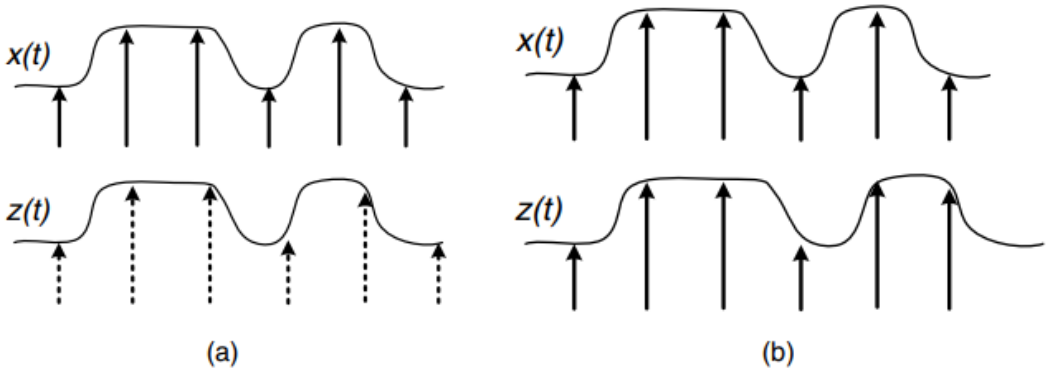
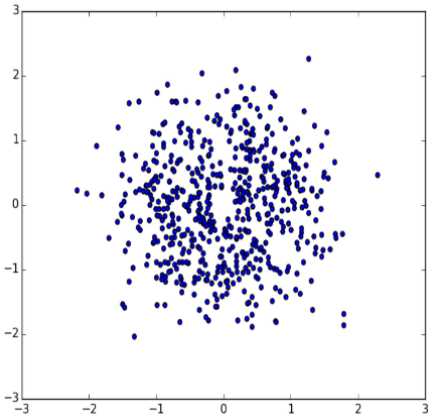
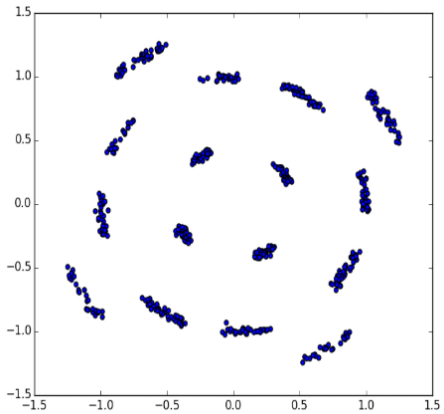


Figure 3.7: Sampling error cases.

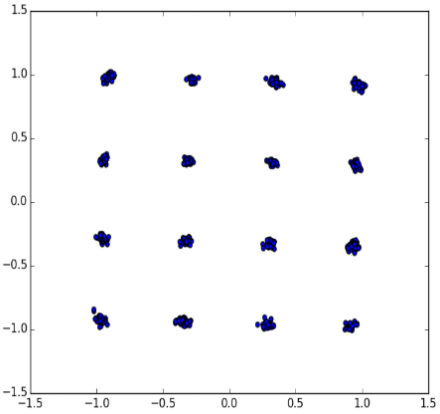
In the figure 3.8 it is resumed visually how each correction step helps in the final constellation plane.



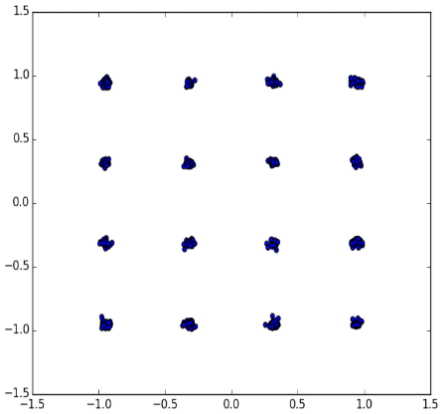
(a) Constellation Points Without Any Correction



(b) Constellation Points with Only Coarse Correction



(c) Constellation Points with both Coarse and Fine Correction



(d) Constellation Points with Coarse, Fine and Pilot Correction

Figure 3.8: constellation points of a 16-QAM modulated 802.11p packet

3.7.1 Coarse CFO Correction

The coarse CFO can be estimated using the short preamble as follows:

$$\alpha_{ST} = \frac{1}{16}\angle\left(\sum_{i=0}^{N-1} \overline{S[i]} \cdot S[i+16]\right)$$

where $\angle(\cdot)$ is the phase of complex number and $N \leq 144(160-16)$ is the subset of short preambles utilized. The intuition is that the phase difference between $S[i]$ and $S[i+16]$ represents the accumulated CFO over 16 samples. After getting α_{ST} each following I/Q samples (starting from long preamble) are corrected as:

$$S'[m] = S[m]\exp^{-j\cdot m\cdot\alpha_{ST}}, \quad m = 0, 1, 2, ...$$

3.7.2 Fine CFO Correction

A finer estimation of the CFO can be obtained with the help of long training sequence inside the long preamble. The long preamble contains two identify training sequence (64 samples each at 20 MSPS), the phase offset can be calculated as:

$$\alpha_{ST} = \frac{1}{64}\angle\left(\sum_{i=0}^{63} \overline{S[i]} \cdot S[i+64]\right)$$

3.7.3 Implementation

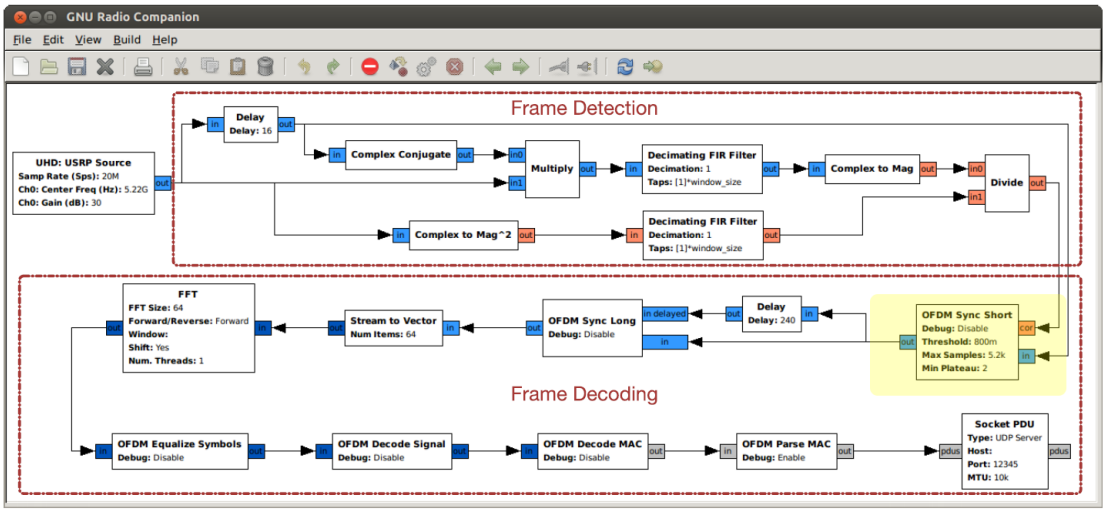


Figure 3.9: GNU Radio Blocks in charge of frequency Offset.

Frequency correction is performed in the reference GNU Radio design by the block OFDM Sync Short block to act like a valve. Its inputs are the samples from the USRP and the normalized autocorrelation coefficient. If it detects a plateau in the auto-correlation stream, it pipes a fixed number of samples into the rest of the signal processing pipeline; otherwise it drops the samples. It also performs the first part of the frequency compensation (the rest will be discussed in the next section for the sync Long Block).

Once a frame has been detected, all the data belonging to the frame are streamed to the sync short block (now in a stream fashion). The **Coarse CFO correction** is implemented by finding the frequency offset:

```
1
2   if (frame==false) {
3     for(unsigned i=0;i<CHUNK;i++)
4     {
5       if(c_plateau<2){
6         if( correlation[i]> (fx_pt1)0.56 )
7           c_plateau++;
8       } else {
```

```
        fx_pt1_ext1 x=  (fx_pt1_ext1) input_abs[i].real();
        fx_pt1_ext1 y=  (fx_pt1_ext1) input_abs[i].imag();
        fx_pt1_ext1 raz=  y/x ;
        df= hls::atan( raz )/16;
        frame_start=i;
        frame=true;
        break;
      }
    } //frame for every sample
  } // fine if frame
```

Listing 3.13: Coarse CFO

and next applying the frequency offset to every sample.

```
  if(frame) {
    //printf("frame_start: %d\n",frame_start);
    fx_pt_ext df_c;
    static fx_pt1_ext1 pos=1;

    fx_pt1_ext2 mult=0;
    static fx_pt1_ext2 pos_ext=1;

    //printf("df=%f\n",df.to_float() );
    c_freq_corr:for(unsigned k=frame_start;k<CHUNK;k++)
    {
        mult= -(fx_pt1_ext2)df*pos_ext;


        fx_pt_ext esp = fx_pt_ext( hls::cos((fx_pt1_ext1)mult), hls::sin((
    fx_pt1_ext1)mult));
        //printf("-df*pos=%f*%f=%f, (%f,%f)\n",df.to_float(),pos.to_float(),mult.
    to_float(), esp.real().to_float(),esp.imag().to_float() );
        fx_pt_ext prod= fx_pt_ext(input_sample[k]) *  esp ;
        //printf("input: (%f,%f)\n", input_sample[k].real().to_float(),
    input_sample[k].imag().to_float() );
        //printf("prod: (%f,%f)\n ", prod.real().to_float(), prod.imag().to_float()
     );
        //output[k]= fx_pt(prod);
        output.write(fx_pt(prod) );
        pos++;
        pos_ext++;

    } // fine for
    if(pos_ext>=MAX_SAMPLES)
    {
      frame=false;
      //printf("arrivati a fine\n");

    }
  }


}
```

Listing 3.14: Coarse CFO con't

Matlab Implementation

```
%sync short -d2 is the input - d3 the oputput

MAX_SAMPLES = 540 * 80;
threshold=0.56;
start_packet= find(cor>threshold,1,'first')+2;

m=['-) Frame start at: ', num2str(start_packet)];
disp(m);

if(MAX_SAMPLES<length(d)-start_packet)
   d2=d_d(start_packet:start_packet+MAX_SAMPLES);
else
    d2=d_d(start_packet:end);
end
```

```
15
16      idx=2;
17      Ns=1;
18      Nf=60;
19      df=angle(s_x_sc(start_packet))/16;
20      df_gnuradio=-0.000861016;
21       err=df_gnuradio-df;
22       err_p=err/df_gnuradio*100;
23
24       str=['-) Coarse Frequency offset :' num2str(df) ];
25       disp(str);
26       str=['    *)', 'err=' num2str(err) ' err(%)= ' num2str(err_p) '%'];
27       disp(str);
28
29
30  if coarse_freq_adj==1
31      for k=1:length(d2)
32          esp(k,1)=exp(-1i*k*df);
33          esp(k,2)=cos(-df*k)+1i*sin(-df*k);
34          esp(k,3)= -k*df;
35          d3(k,1)=d2(k).*esp(k,1);
36      end
37  else
38      d3=d2;
39  end
```

Listing 3.15: Sync_short Matlab Function

3.8 Symbol alignment

After detecting the packet, the next step is to determine precisely where each OFDM symbol starts. In 802.11, each OFDM symbol is 8 μs long. At 10 MSPS sampling rate, this means each OFDM symbol contains 80 samples. The task is to group the incoming streaming of samples into 80 sample OFDM symbols. This can be achieved using the 2 long preamble.

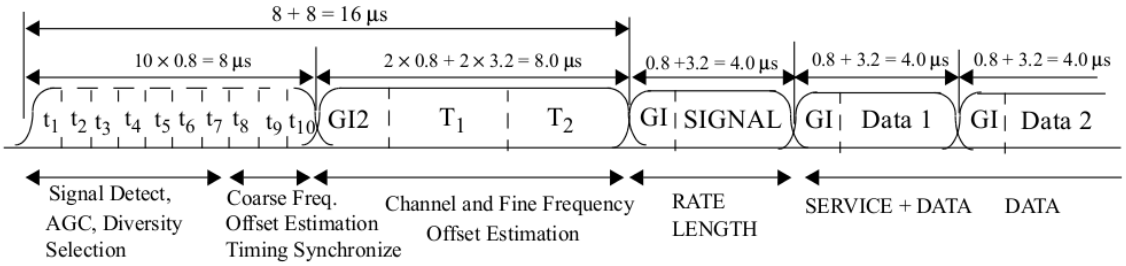
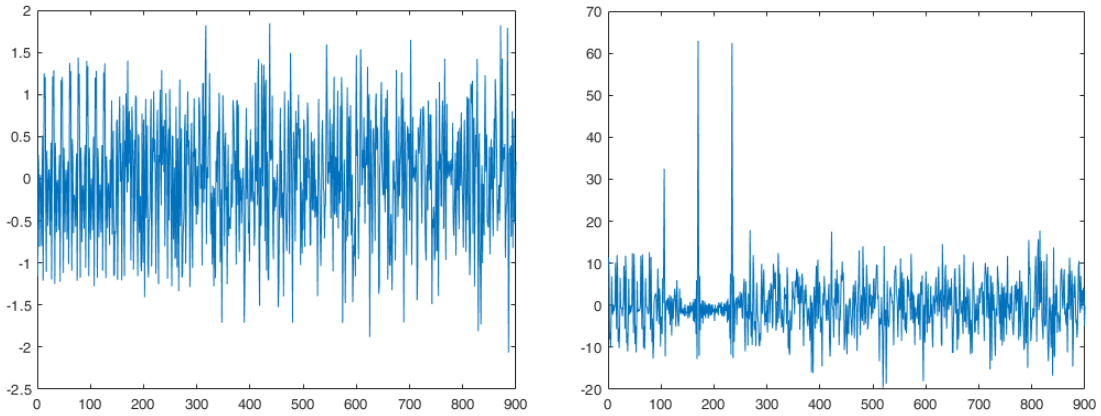


Figure 3.10: 802.11 OFDM Packet Structure (Fig 18-4 in 802.11-2012 Std)

As shown in 3.10, the long preamble duration is 8 μs (160 samples), and contains two identical long training sequence (LTS), 64 samples each. The LTS is known and we can use cross correlation to find it. The cross validation score at sample i can be calculated as follows.

$$Y[i] = \sum_{k=0}^{63} (S[i+k]\overline{H[k]})$$

where H is the 64 sample known LTS in time domain, and can be found in Table L-6 in 802.11-2012 std (index 96 to 159) [2].



(a) Real Part of the I-Q signal

(b) Cross Correlation of (a)

Figure 3.11: Long Preamble and Cross Correlation Result

Figure 3.11 shows the long preamble samples and also the result of cross correlation. We can clearly see two spikes corresponding the two LTS in long preamble. And the spike width is only 1 sample which shows exactly the beginning of each sequence. Suppose the sample index if the first spike is N, then the 160 sample long preamble starts at sample $N - 32$

[2] https://standards.ieee.org/standard/802_11 − 2012.*html*

3.8.1 Implementation

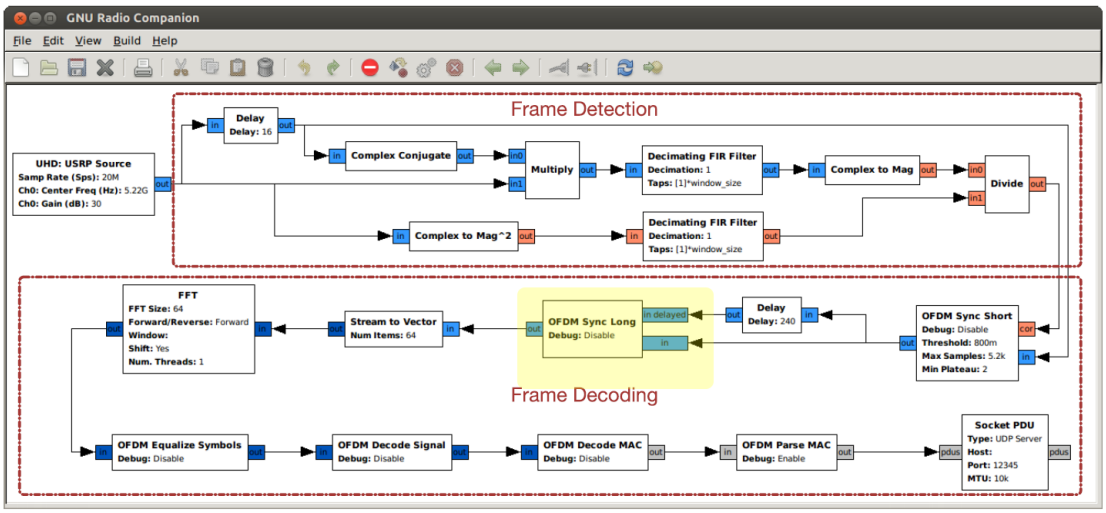


Figure 3.12: GNU Radio Blocks in charge of Symbol alignment.

Symbol alignment is performed in the reference design in GNU Radio by the sync Long Block. Actually OFDM Sync Long applies the remaining part of frequency offset correction and symbol alignment.

Cross correlation is performed by the function *firG*. The function implements a classic finite impulse response filter (FIR) written in order to accommodate the streaming data coming from the previous block.

```
void firG(fx_pt output[CHUNK],fx_pt input_sample[CHUNK], const fx_pt_ext
    coefficient[COEFF_LENGTH])
{
    static fx_pt_ext buffer[COEFF_LENGTH];
    static bool init=true;

  if(init) {
init: for(unsigned i=0; i<COEFF_LENGTH;i++)
      buffer[i]=fx_pt_ext(0,0);
    init=false;
  }

loop_filter_data:
    for(unsigned n=0;n<CHUNK;n++)
    {
      fx_pt_ext _output[COEFF_LENGTH];
      fx_pt_ext _output_=fx_pt_ext(0,0);
data_shift:
    for(int i=COEFF_LENGTH-1;i>=0;i--) {
        if(i==0) buffer[0]=input_sample[n];
        else     buffer[i]=buffer[i-1];
      }
Mply:
    for(int i=COEFF_LENGTH-1;i>=0;i--)
          _output[i]=buffer[i]*coefficient[i];
ACC:
    for(int i=COEFF_LENGTH-1;i>=0;i--)
        _output_ +=_output[i];

      output[n]=(fx_pt)_output_;
    } //end for every sample

} //end function
```

Listing 3.16: complex filter

Once the cross correlation has been calculated the output Y are analyzed in order to find the two picks (see fig. 3.11).

```
1
2
3     find_pick: for(unsigned i=1;i<CHUNK;i++)
4       {
5         fx_pt1_ext comp1= (fx_pt1_ext)abs_c(first_pick) -(fx_pt1_ext)abs_c(filtered
      [i]);
6         fx_pt1_ext comp2= (fx_pt1_ext)abs_c(second_pick)-(fx_pt1_ext)abs_c(filtered
      [i] );
7   /*
8         printf("%d -> %f ",i,comp2.to_float() );
9         printf("|(%f,%f)|=%f   ",first_pick.real().to_float(), first_pick.imag().
      to_float(), abs_c(first_pick).to_float() );
10        printf("|(%f,%f)|=%f\n",filtered[i].real().to_float(), filtered[i].imag().
      to_float(), abs_c(filtered[i]).to_float() );
11  */
12        if( comp1<0 )
13        {
14          second_pick=first_pick;
15          first_pick=filtered[i];
16
17          idx2=idx1;
18          idx1=i;
19          pick=abs_c(first_pick);
20        } else if (comp2<0)
21        {
22          second_pick=filtered[i];
23          idx2=i;
24        }
25      }
```

Listing 3.17: Sync Long - peak find

From the pick index, the starting points of the symbols are calculated. In the same time the fine CFO correction (*d_freq_offset*) is estimated :

```
1
2
3     fx_pt first=fx_pt(0,0);
4     fx_pt second=fx_pt(0,0);
5     if(idx1>idx2) {
6       first=filtered[idx2];
7       second=filtered[idx1];
8     }else {
9       first=filtered[idx1];
10      second=filtered[idx2];
11    }
12
13    fx_pt_ext1 arg= (fx_pt_ext1)(first)* fx_pt_ext1(std::conj(second));
14    fx_pt1_ext1 x=  (fx_pt1_ext1) arg.real();
15    fx_pt1_ext1 y=  (fx_pt1_ext1) arg.imag();
16    fx_pt1_ext1 raz=  y/x ;
17    d_freq_offset= hls::atan( raz  )/64;
18
19    if (idx1>idx2)
20      start_frame=idx1+64-128;
21    else
22      start_frame=idx2+64-128;
```

Listing 3.18: Sync Long - symbol starting index & fine CFO estimation

Finally all the data packets above LTS are corrected with the quantities found above:

```
1
2     if(d_offset_ui<=MAX_SAMPLES) {
3       //printf("samples: %d\n", smpl);
4
5       freq_correct: for(unsigned i=0;i<smpl;i++) {
6         int rel=d_offset_ui-(int)start_frame;
7         //printf("i=%d, rel=%d\n",i,rel);
8
9         d_offset++;
10        d_offset_ui++;
```

```
        if(rel>=0 && (rel<128 || ((rel-128)%80) >15 ) )
        {
          fx_pt_ext2 esp = fx_pt_ext2( hls::cos(d_freq_offset*d_offset), hls::sin(
    d_freq_offset*d_offset));
          //fx_pt_ext2 num=(fx_pt_ext2)(toBfiltered[i+167]);
          fx_pt_ext2 num=(fx_pt_ext2)(toBfiltered[i]);

          //fx_pt_ext2 num=(fx_pt_ext2)(toBfiltered_d[i+320+167]);
          fx_pt_ext2 dsampl= esp*num;
          output.write( fx_pt(dsampl) );
        } // fine if
        o++;
      }//fine for
```

Listing 3.19: Sync Long - frequency correction

Matlab Implementation

```
comparison=0;

load('./data/lts.mat')
LT=lts_real+1i*lts_imag;

%LT=flip(LT);
%d3=sync_ref_data;

d_correlation_ext=filter(LT,1,d3(1:end));

d_correlation=d_correlation_ext(64:end);
%d3=d3(64:end);

if comparison==1
    import_corr_gr
    d_cor_gr=corr_r_gr+1i*corr_i_gr;
    plot(abs(d_cor_gr));
    hold on
    plot(abs(d_correlation(1:320)));
    [vgr,idxmgr]=maxk(d_cor_gr,2,'ComparisonMethod','abs');
end

[v,idxm]=maxk(d_correlation,2,'ComparisonMethod','abs');

np=max(idxm)+64;

off=0;
flag=0;

for i=1:2
    for k=i+1:2

        if idxm(i)>idxm(k)
            first=d_correlation( idxm(k)+off );
            second=d_correlation(idxm(i)+off );
        else
            first=d_correlation(idxm(i)+off );
            second=d_correlation(idxm(k)+off );
        end

        diff=abs(idxm(i)-idxm(k));

        if diff==64
            %d_frame_start =  max(get<1>(vec[i]), get<1>(vec[k])) + 64 - 128 - 1;
            d_frame_start=max([idxm(i),idxm(k)])+64-128;
            %d_frame_start=min([idxm(i),idxm(k)]);
            d_freq_offset=  angle( first * conj(second ) )/64;
            flag=1;
            break;
        end

    end
    if flag==1
        break;
```

```
    end
end

 m=['-) Fine Frequency offset : ', num2str(d_freq_offset)];
disp(m);
 m=['    *) diff=' num2str(df-d_freq_offset)];
disp(m);
m=['-) Long Frame Start: ' num2str(d_frame_start)];
disp(m);

d_offset=169;
o=1;
% adjust the frequency
if fine_freq_adj==1
    for k=1:(length(d3)-170)

       rel= d_offset - (d_frame_start-1);
       if (rel>=0 && (rel < 128 || ( mod( (rel - 128) , 80)> 15) ) )

           espo(o,1)=d3(k+168);
           espo(o,2)=exp(1i*d_offset*d_freq_offset);
           espo(o,3)=d_offset;
           espo(o,4)=rel;

           d4(o,1)=d3(k+168).*espo(o,2);
           o=o+1;
       end
       d_offset=d_offset+1;
    end
else
    d4=d3;
end
```

Listing 3.20: Sync_long Matlab Function

3.9 FFT

At the receiver, the system performs inverse processing with respect to the transmit side. Specifically, the system first performs carrier and clock recovery in the synchronization circuit section. It then performsorthogonal demodulation on the received OFDM signal by using the recovered carrier frequency fc as a local signal, and converts the result to the baseband frequency. The system next performs an A/D conversion on these baseband signals by using the recovered clock and then executes an FFT to enable the receive data from each carrier to be determined.
After A/D conversion, N instances of complex data R(i) within the effective symbol period Tu are input to an Npoint FFT. Its output Y(n) is as follows.

$$Y(n) = \frac{1}{N} \sum_{i=0}^{N-1} R(i) e^{-j\left(2\pi \frac{i}{N}(n-Kc)\right)}$$

3.9.1 Implementation

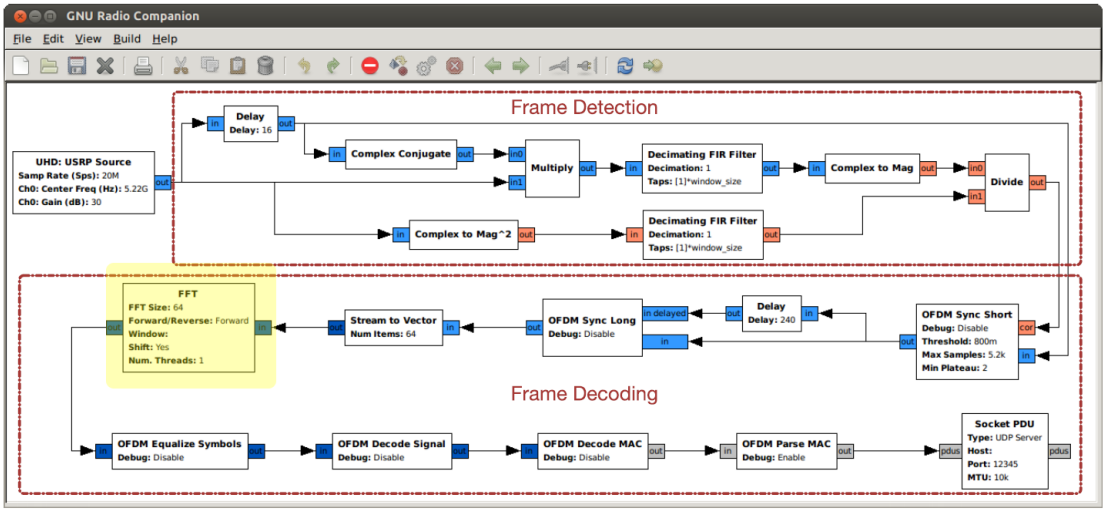


Figure 3.13: GNU Radio Blocks in charge of FFT.

In the reference design(Figure 3.13) FFT is implemented by the out of the box FFT function.

In hardware, the FFT is performed by the function prototype void fft_hls(DTYPE X_R[SIZE], DTYPE X_I[SIZE]) where DTYPE is a user customizable data type for the representation of the input data. This may be int, float, or a fixed point type. For example, define DTYPE int defines DTYPE as an int. Note that we choose to implement the real and imaginary parts of the complex numbers in two separate arrays. The X_R array holds the real input values, and the X_I array holds the imaginary values. X_R[i] and X_I[i] hold the ith complex number in separate real and imaginary parts.
This function prototype forces an in-place implementation. That is, the output data is stored in the same array as the input data. This eliminates the need for additional arrays for the output data, which reduces the amount of memory that is required for the implementation. However, this may limit the performance due to the fact that we must read the input data and write the output data to the same arrays. Using separate arrays for the output data is reasonable if it can increase the performance. There is always a trade-off between resource usage and performance; Of course, the best implementation depends upon the application requirements (e.g., high throughput, low power, size of FPGA, size of the FFT, etc.).

```
1 void fft_hls(DTYPE X_R[SIZE], DTYPE X_I[SIZE], bool shift) {
2
3   DTYPE temp_R; // temporary storage complex variable
4   DTYPE temp_I; // temporary storage complex variable
```

```
  int i, j, k; // loop indexes
  int i_lower; // Index of lower point in butterfly
  int step, stage, DFTpts;
  int numBF; // Butterfly Width
  int N2 = SIZE2; // N2=N>>1
  bit_reverse(X_R, X_I);
  step = N2;
  DTYPE a, e, c, s;



stage_loop:
  for (stage = 1; stage <= M; stage++) {
    // Do M stages of butterflies
    DFTpts = 1 << stage; // DFT = 2 stage = points in sub DFT
    numBF = DFTpts / 2; // Butterfly WIDTHS in sub-DFT
    k = 0;
    e = -6.283185307178 / DFTpts;
    a = 0.0;
    // Perform butterflies for j-th stage
butterfly_loop:
    for (j = 0; j < numBF; j++) {
      c = hls::cos(a);
      s = hls::sin(a);
      a = a + e;
      // Compute butterflies that use same W**k
  dft_loop:
      for (i = j; i < SIZE; i += DFTpts) {
        i_lower = i + numBF; // index of lower point in butterfly
        temp_R = X_R[i_lower] * c - X_I[i_lower] * s;
        temp_I = X_I[i_lower] * c + X_R[i_lower] * s;
        X_R[i_lower] = X_R[i] - temp_R;
        X_I[i_lower] = X_I[i] - temp_I;
        X_R[i] = X_R[i] + temp_R;
        X_I[i] = X_I[i] + temp_I;
      }
      k += step;
    }
    step = step / 2;
  }


  DTYPE swap_r, swap_i;

  if(shift)
shift: for(unsigned i=0;i<SIZE/2;i++)
    {
      swap_r=X_R[i];
      swap_i=X_I[i];
      X_R[i]=X_R[32+i];
      X_I[i]=X_I[32+i];
      X_R[32+i]=swap_r;
      X_I[32+i]=swap_i;
    }
}
```

Listing 3.21: Implementation for the FFT using three nested for loops.

We start with code for an FFT that would be typical for a software implementation. 3.21 shows a nested three for loop structure.
The outer for loop, labeled stage loop implements one stage of the FFT during each iteration. There are log2(N) stages where N is the number of input samples. The stages are clearly labeled in 3.21; this 8 point FFT has log2(8) = 3 stages. You can see that each stage performs the same amount of computation, or the same number of butterfly operations. In the 8 point FFT, each stage has four butterfly operations.
The second for loop, labeled butterfly loop, performs all of the butterfly operations for the current stage.
Butterfly loop has another nested for loop, labeled dft loop. Each iteration of dft loop performs one butterfly operation. Remember that we are dealing with complex numbers and must perform complex additions and multiplications. The remaining operations in dft loop perform multiplication by the twiddle factor and an addition or subtraction operation. The variables temp R and temp

I hold the real and imaginary portions of the data after multiplication by the twiddle factor W. The variables c and s are the real and imaginary parts of W, which is calculated using the sin() and cos() builtin functions. We could also use the CORDIC, to have more control over the implementation. Lastly, elements of the X_R[] and X_I[] arrays are updated with the result of the butterfly computation. dft loop and butterfly loop each execute a different number of times depending upon the stage. However the total number of times that the body of dft loop is executed in one stage is constant. The number of iterations for the butterfly for loop depends upon the number of unique W twiddle factors in that stage. Referring again to Figure 5.4, we can see that Stage 1 uses only one twiddle factor, in this case W08. Stage 2 uses two unique twiddle factors and Stage 3 uses four different W values. Thus, butterfly loop has only one iteration in Stage 1, 2 iterations in stage 2, and four iterations in stage 3. Similarly, the number of iterations of dft_loop changes. It iterates four times for an 8 point FFT in Stage 1, two times in Stage 2, and only one time in stage 3. However in every stage, the body of dft_loop is executed the same number of times in total, executing a total of four butterfly operations for each stage an 8 point FFT.

3.9.2 Bit Reversal

```
unsigned int reverse_bits(unsigned int input) {
  int i, rev = 0;
  for (i = 0; i < FFT_BITS; i++) {
    rev = (rev << 1) | (input & 1);
    input = input >> 1;
  }
  return rev;
}

void bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE]) {
  unsigned int reversed;
  unsigned int i;
  DTYPE temp;

  for (i = 0; i < SIZE; i++) {
    reversed = reverse_bits(i); // Find the bit reversed index
    if (i < reversed) {
      // Swap the real values
      temp = X_R[i];
      X_R[i] = X_R[reversed];
      X_R[reversed] = temp;
      // Swap the imaginary values
      temp = X_I[i];
      X_I[i] = X_I[reversed];
      X_I[reversed] = temp;
    }
  }
}
```

Listing 3.22: Function reverse_bit and function bit_reverse are the first stage of the FFT implementation.

3.22 shows one possible implementation of the bit reverse function. It divides the code into two functions. The first is the bit reversal function (bit reverse), which reorders data in the given arrays so that each data is in located at a different index in the array. This function calls another function, reverse bits, which takes an input integer and returns the bit reversed value of that input. The function goes bit by bit through the input variable and shifts it into the rev variable. The for loop body consists of a few bit-wise operations that reorder the bits of the input. Although these operations are individually not terribly complex, the intention of this code is that the for loop is completely unrolled and Vivado HLS can identify that the bits of the input can simply be wired to the output. As a result, the implementation of the reverse bits function should require no logic resources at all, but only wires. This is a case where unrolling loops greatly simplifies the operations that must be performed. Without unrolling the loop, the individual 'or' operations must be performed sequentially. Although this loop can be pipelined, the 'or' operation would still be implemented in logic resources in the FPGA and executing the loop would have a latency determined by the number of bits being reversed.

Now let us optimize the parent bit reverse function. This function has a single for loop that iterates through each index of the input arrays. Note that there are two input arrays X_R[] and X_I[]. Since we are dealing with complex numbers, we must store both the real portion (in the array X_R[]), and the imaginary portion (in the array X_I[]). X_R[i] and X_I[i] holds the real and imaginary values of the i-th input. In each iteration of the for loop, we find the index reversed value by calling the reverse bits function. Then we swap both the real and imaginary values stored in the index i and the index returned by the function reverse bits. Note that as we go through all SIZE indices, we will eventually hit the reversed index for every value. Thus, the code only swaps values the first time based on the condition if(i < reversed).

3.9.3 Optimization

When performed sequentially, the O(n log n) operations in the FFT require O(n log n) time steps. Typically, a parallel implementation will perform some portion of the FFT in parallel. One common way of parallelizing the FFT is to organize the computation into log n stages, as shown in Figure ??. The operations in each stage are dependent on the operations of the previous stage, naturally leading to a pipelining across the tasks. Such an architecture allows log n FFTs to be computed simultaneously with a task interval determined by the architecture of each stage [3].
Each stage in the FFT also contains significant parallelism, since each butterfly computation is independent of other butterfly computations in the same stage. In the limit, performing $n/2$ butterfly computations every clock cycle with a Task Interval of 1 can allow the entire stage to be computed with a Task Interval of 1. When combined with a dataflow architecture, all of the parallelism in the FFT algorithm can be exploited. Note, however that although such an architecture can be constructed, it is almost never used except for very small signals, since an entire new block of SIZE samples must be provided every clock cycle to keep the pipeline fully utilized.[4]

Pipelining

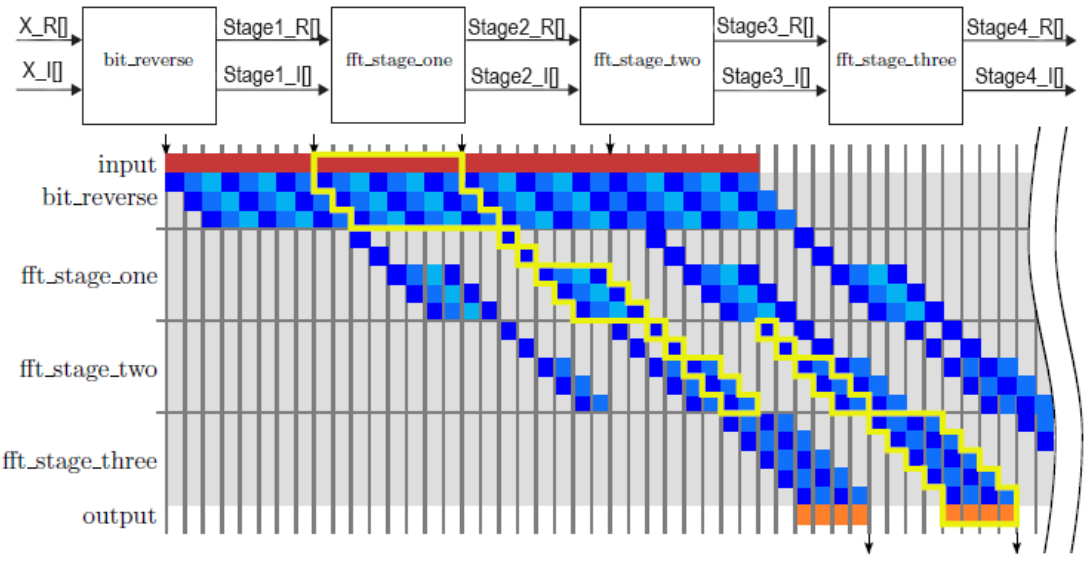


Figure 3.14: example with three FFT stages (i.e., an 8 point FFT). The Figure shows four 8 point FFT executing at the same time.

Dividing the FFT algorithm into stages enables Vivado HLS to generate an implementation where different stages of the algorithm are operating on different data sets. This optimization, called task

[3]Pipelining in combination with dataflow HLS directive will be explained in the next sections
[4]For instance, a 1024-point FFT of complex 32-bit floating point values, running at 250 MHz would require 1024 points*(8 bytes/point)*250*109 Hz = 1Terabyte/second of data into the FPGA!! In practice, a designer must match the computation architecture to the data rate required in a system.

pipelining is enabled using the dataflow directive. This is a common hardware optimization, and thus is relevant across a range of applications.
We can naturally divide the FFT algorithm into $log_2(N+1)$ stages where N is the number of points of the FFT. The first stage swaps each element in the input array with the element located at the bit reversed address in the array. After this bit reverse stage, we perform $log_2(N)$ stages of butterfly operations. Each of these butterfly stages has the same computational complexity. Figure 3.14 describes how to divide an 8 point FFT into four separate tasks[5]. The code has separate functions for each of the tasks: bit reverse, fft stage one, fft stage two, and fft stage three[6]. Each stage has two input arrays and two output arrays: one for the real portion and one for the imaginary portion of the complex numbers. Assume that the DTYPE is defined elsewhere, e.g., as an int, float or a fixed point data type. Refactoring the FFT code allows us to perform task pipelining.
Rather than wait for the first task to complete all four functions in the code before the second task can begin, we allow the second task to start after the first task has only completed the first function bit reverse. The first task continues to execute each stage in the pipeline in order, followed by the remaining tasks in order. Once the pipeline is full, all four sub-functions are executing concurrently, but each one is operating on different input data. Similarly, there are four 8 point FFTs being computed simultaneously, each one executing on a different component of the hardware. This shown in the middle portion of Figure 3.14. Each of the vertical four stages represents one 8 point FFT. And the horizontal denotes increasing time. Thus, once we start the fourth 8 point FFT, we have four FFTs running simultaneously.
The top entity of the FFT function is shown in 3.23.

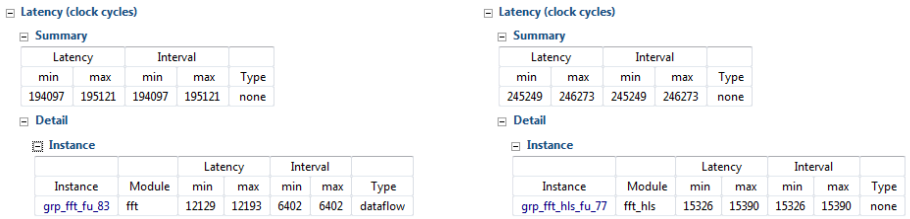
```
void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE],DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
  DTYPE stage1_r[SIZE], stage1_i[SIZE];
  DTYPE stage2_r[SIZE], stage2_i[SIZE];
  DTYPE stage3_r[SIZE], stage3_i[SIZE];
  DTYPE stage4_r[SIZE], stage4_i[SIZE];
  DTYPE stage5_r[SIZE], stage5_i[SIZE];
  DTYPE stage6_r[SIZE], stage6_i[SIZE];


  bit_reverseMOD(X_R,X_I,stage1_r,stage1_i);
  fft_stage(stage1_r,stage1_i,stage2_r,stage2_i,1);
  fft_stage(stage2_r,stage2_i,stage3_r,stage3_i,2);
  fft_stage(stage3_r,stage3_i,stage4_r,stage4_i,3);
  fft_stage(stage4_r,stage4_i,stage5_r,stage5_i,4);
  fft_stage(stage5_r,stage5_i,stage6_r,stage6_i,5);
  fft_stage(stage6_r,stage6_i,OUT_R,OUT_I,6);
}
```

Listing 3.23: Top function of the FFT algorithm.

Dataflow & Loop Unrolling

The dataflow directive can construct separate pipeline stages (often called processes) from both functions and loops. The code in 3.23 uses functions only, but we could achieve a similar result with six loops instead of four functions. In fact, this result could be achieved by unrolling the outer stage loop in the original code either explicitly or using #pragma HLS unroll. Comparison results of this two techniques are shown in Fiugure 3.25.

[5]A smoller FFT is shown in the figure to better visualize the concept.
[6]The FFT stages has the same code as the original implementation in 3.21 without the *stage_loop* for loop.



Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
194097	195121	194097	195121	none

Detail

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_fft_fu_83	fft	12129	12193	6402	6402	dataflow

(a) HLS Report of the modified code with the Dataflow directive.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
245249	246273	245249	246273	none

Detail

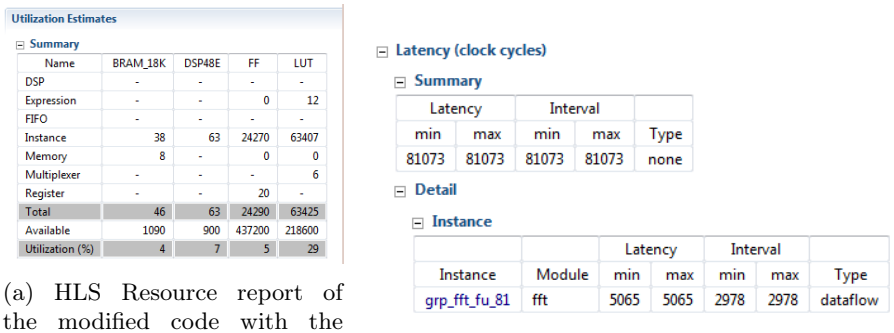
Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_fft_hls_fu_77	fft_hls	15326	15390	15326	15390	none

(b) original code with the *stage_ loop* completely unrolled.

Figure 3.15: Comparison between Function dataflow and loop unrolling.

The dataflow directive and the pipeline directive both generate circuits capable of pipelined execution. The key difference is in the granularity of the pipeline. The pipeline directive constructs an architecture that is efficiently pipelined at the cycle level and is characterized by the II of the pipeline. Operators are statically scheduled and if the II is greater than one, then operations can be shared on the same operator. The dataflow directive constructs an architecture that is efficiently pipelined for operations that take a (possibly unknown) number of clock cycles, such as the behavior of a loop operating on a block of data. These coarse-grained operations are not statically scheduled and the behavior is controlled dynamically by the handshake of data through the pipeline. In the case of the FFT, each stage is an operation on a block of data (the whole array) which takes a large number of cycles. Within each stage, loops execute individual operations on the data in a block. Hence, this is a case where it often makes sense to use the dataflow directive at the toplevel to form a coarse-grained pipeline, combined with the pipeline directive within each loop to form fine-grained pipelines of the operations on each individual data element.
Final results are reported in Figure 3.16



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	12
FIFO	-	-	-	-
Instance	38	63	24270	63407
Memory	8	-	0	0
Multiplexer	-	-	-	6
Register	-	-	20	-
Total	46	63	24290	63425
Available	1090	900	437200	218600
Utilization (%)	4	7	5	29

(a) HLS Resource report of the modified code with the Dataflow directive and Fine-Grain Pipeline.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
81073	81073	81073	81073	none

Detail

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_fft_fu_81	fft	5065	5065	2978	2978	dataflow

(b) HLS Latency report of the modified code with the Dataflow directive and Fine-Grain Pipeline.

Figure 3.16: Fine-grain HLS report implemented in a dataflow region..

The dataflow directive must implement memories to pass data between different processes. In the case when Vivado HLS can determine that processes access data in sequential order, it implements the memory using a FIFO. This requires that data is written into an array in the same order that it is read from the array. If the is not the case, or if Vivado HLS can not determine if this streaming condition is met, then the memory can be implemented using a ping-pong buffer instead. The ping-pong buffer consists of two (or more) conceptual blocks of data, each the size of the original array. One of the blocks can be written by the source process while another block is read by the destination process. The term “ping-pong” comes from the fact that the reading and writing to each block of data alternates in every execution of the task. That is, the source process will write to one block and then switch to the other block before beginning the next task. The destination process reads from the block that the producer is not writing to. As a result, the source and destination processes can never write and read from the same block at the same time.

A ping-pong buffer requires enough memory to store each communication array at least twice. FIFOs (Vivado HLS, in this project) instantiate 7 FIFOs -one for each stage- as shown in Fig. 3.17) can often be significantly smaller, although determining a minimal size for each fifo is often a difficult design problem. Unlike a FIFO, however, the data in a ping-pong buffer can be written to and read from in any order. Thus, FIFOs are generally the best choice when the data is produced and consumed in sequential order and ping-pong buffers are a better choice when there is not such regular data access patterns.

FIFO

Name	BRAM_18K	FF	LUT	Depth	Bits	Size:D*B
X_R_V_offset_c_U	0	6	23	7	10	70
Total	0	6	23	7	10	70

Figure 3.17: FIFO element instantiated in the *fft()* function

Using the dataflow directive effectively still requires the behavior of each individual process to be optimized. Each individual process in the pipeline can still be optimized using techniques we have seen previously such as code restructuring, pipelining, and unrolling. For example, we have already discussed some optimizations for the bit reverse function. In general, it is important to optimize the individual tasks while considering overall toplevel performance goals. Many times it is best to start with small functions and understand how to optimize them in isolation. As a designer, it is often easier to comprehend what is going on in a small piece of code and hopefully determine the best optimizations quickly. After optimizing each individual function, then you can move up the hierarchy considering larger functions given particular implementations of low level functions, eventually reaching the toplevel function.

However, the local optimizations must be considered in the overall scope of the goals. In particular for dataflow designs the achieved interval for the overall pipeline can never be smaller than the interval of each individual process. Looking again at Figure 3.14, assume that bit reverse has an interval of 8 cycles, fft stage one takes 12 cycles, fft stage two requires 12 cycles, and fft stage three takes 14 cycles. When using dataflow, the overall task interval is 14, determined by the maximum of all of the tasks/functions. This means that you should be careful in balancing optimizations across different processes with the goal of creating a balanced pipeline where the interval of each process is approximately the same. In this example, improving the interval of the bit reverse function cannot improve the overall interval of the fft function. In fact, it might be beneficial to increase the latency of the bit reverse function, if it can be achieved with significantly fewer resources.

Matlab Implementation

```
% input d4---
shift=1;
nsimb=min( length(d4), 540*80);
for i=0: (double(int32(nsimb/64))-1)
    sym(:,i+1)=d4(i*64+1:i*64+64 );
end

X=fft(sym);
if shift==1
    X=fftshift(X,1);
end

X_lin=reshape(X,[],1);
```

Listing 3.24: FFT Matlab Function

3.10 Sub-carrier Equalization and Pilot Correction

This is the first step in frequency domain. There are two main tasks:

1. sub-carrier gain equalization,

2. correcting residue phase offset using the pilot sub-carriers.

3.10.1 Sub-carrier Structure

The basic channel width in 802.11p is 10 MHz, which is further divided into 64 sub-carriers (0.156 MHz each).

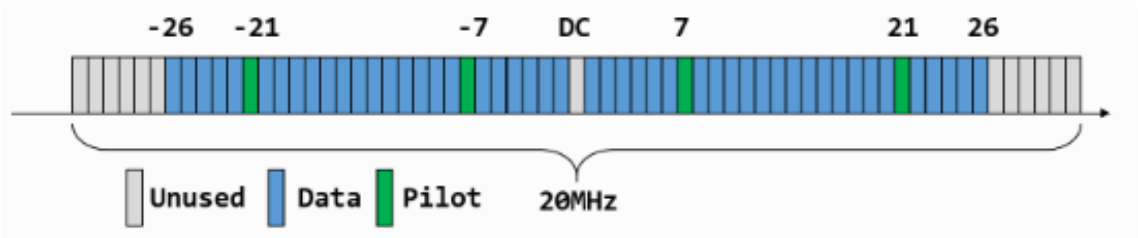


Figure 3.18: Sub-carriers in 802.11 OFDM

3.18 shows the sub-carrier structure of the 10 MHz band. 52 out of 64 sub-carriers are utilized, and 4 out of the 52 (-7, -21, 7, 21) sub-carriers are used as pilot sub-carrier and the remaining 48 sub-carriers carries data. As we will see later, the pilot sub-carriers can be used to correct the residue frequency offset.

Each sub-carrier carries I/Q modulated information, corresponding to the output of 64 point FFT from sync long module.

3.10.2 Sub-Carrier Equalization

In theory every sub-carrier contains an information according to the modulation used. In practice, each sub-carrier exhibits different magnitude gain. In fact, they also have different phase drift. The combined effect of magnitude gain and phase drift (known as channel gain) can clearly be seen in the I/Q plane shown in .

To map the FFT point to constellation points, we need to compensate for the channel gain. This can be achieved by normalize the data OFDM symbols using the LTS(Long Training Sequence). In particular, the mean of the two LTS is used as channel gain H.

$$H[i] = \frac{1}{2}(LTS_1[i] + LTS_2[i]) \cdot L[i], \quad i \in [-26, 26]$$

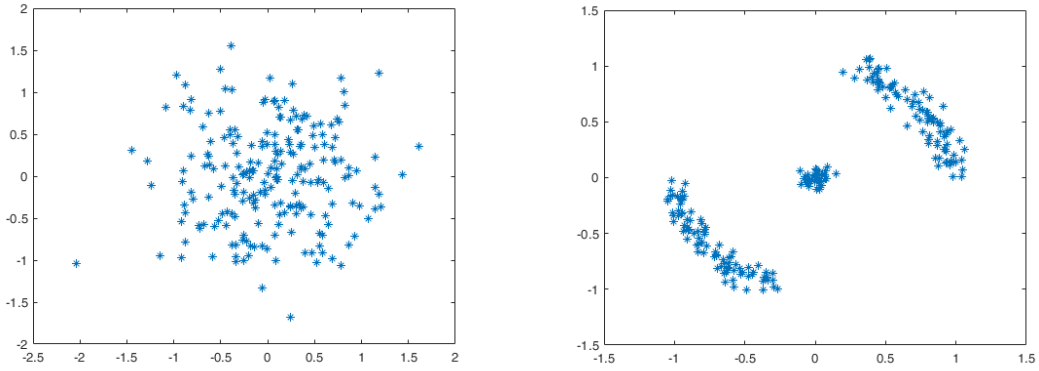
where $L[i]$ is the sign of the LTS sequence:

$$\begin{aligned} L_{-26,26} = \{&1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, \\ &-1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 0, 1, -1, -1, 1, 1, -1, 1, -1, 1, -1, -1, \\ &-1, -1, -1, 1, 1, -1, -1, 1, -1, 1, -1, 1, 1, 1, 1\} \end{aligned}$$

and the FFT output at sub-carrier **i** is normalized as:

$$Y[i] = \frac{X[i]}{H[i]}, i \in [-26, 26]$$

where $X[i]$ is the FFT output at sub-carrier i.



(a) FT Without Normalization

(b) FFT With Normalization

Figure 3.19: FFT before (a) and after (b) normalization using channel gain.

Figure 3.19 is obtained by plotting signals **X, X_eq0** inside the equalize Matlab script.

```
... %execute decode script...
figure
plot(real(X(3,:)),imag(X(3,:)),'*');
figure
plot(real(X_eq0(3,:)),imag(X_eq0(3,:)),'*');
```

3.10.3 Residual Frequency Offset Correction

We can see from Fig. 3.19b that the FFT output is tilted slightly. This is caused by residual frequency offset that was not compensated during the coarse CFO correction step.

This residual CFO can be corrected either by Fine CFO Correction, or/and by the pilot sub-carriers. Ideally we want to do both, but since the fine CFO is usually beyond the resolution of the phase in the sync long module and only rely on the pilot sub-carriers.

Regardless of the data sub-carrier modulation, the four pilot sub-carriers (-21, -7, 7, 21) always contains BPSK modulated pseudo-random binary sequence. The polarity of the pilot sub-carriers varies symbol to symbol. For 802.11p, the pilot pattern is:

$$\begin{aligned} p_{0,\ldots,126} = \{&1,1,1,1,-1,-1,-1,1,-1,-1,-1,-1,1,1,-1,1,-1,-1,1,1,-1,1,1,-1,1,\\ &1,1,1,1,1,-1,1,1,1,-1,1,1,-1,-1,1,1,1,-1,1,-1,-1,-1,1,-1,\\ &1,-1,-1,1,-1,-1,1,1,1,1,1,-1,-1,1,1,-1,-1,1,-1,1,-1,1,\\ &1,-1,-1,-1,1,1,-1,-1,-1,-1,1,-1,-1,1,-1,1,1,1,1,-1,1,-1,1,-1,\\ &1,-1,-1,-1,-1,-1,1,-1,1,1,-1,1,-1,1,1,1,-1,-1,1,-1,-1,-1,1,1,\\ &1,-1,-1,-1,-1,-1,-1,-1\} \end{aligned}$$

And the pilot sub-carriers at OFDM symbol n (starting at 0 from the first symbol after the long preamble) is then:

$$P^{(n)}_{-21,-7,7,21} = \{p_{n\%127}, p_{n\%127}, p_{n\%127}, -p_{n\%127}\}$$

For 802.11p at 10MHz bandwidth with single spatial stream, the n^{th} pilot sub-carriers are:

$$P^{(n)}_{-21,-7,7,21} = \{\Psi_{n\%4}, \Psi_{(n+1)\%4}, \Psi_{(n+2)\%4}, \Psi_{(n+3)\%4}\}$$

and

$$\Psi_{0,1,2,3} = \{1,1,1,-1\}$$

In other words, the pilot sub-carries of the first few symbols are:

$$
\begin{aligned}
P^{(0)}_{-21,-7,7,21} &= \{1, 1, 1, -1\} \\
P^{(1)}_{-21,-7,7,21} &= \{1, 1, -1, 1\} \\
P^{(2)}_{-21,-7,7,21} &= \{1, -1, 1, 1\} \\
P^{(3)}_{-21,-7,7,21} &= \{-1, 1, 1, 1\} \\
P^{(4)}_{-21,-7,7,21} &= \{1, 1, 1, -1\} \\
&\dots
\end{aligned}
$$

For other configurations (e.g., spatial stream, bandwidth), the pilot sub-carrier pattern can be found in Section 20.3.11.10 in 802.11-2012 std.[7]
The residual phase offset at symbol n can then be estimated as:

$$\theta_n = \angle(\sum_{i \in \{-21,-7,7,21\}} \overline{X^{(n)}[i]} \times P^{(n)}[i] \times H[i])$$

Combine this phase offset and the previous channel gain correction together, the adjustment to symbol n is:

$$Y[i] = \frac{X^{(n)}[i]}{H[i]} e^{j\Theta_n}$$

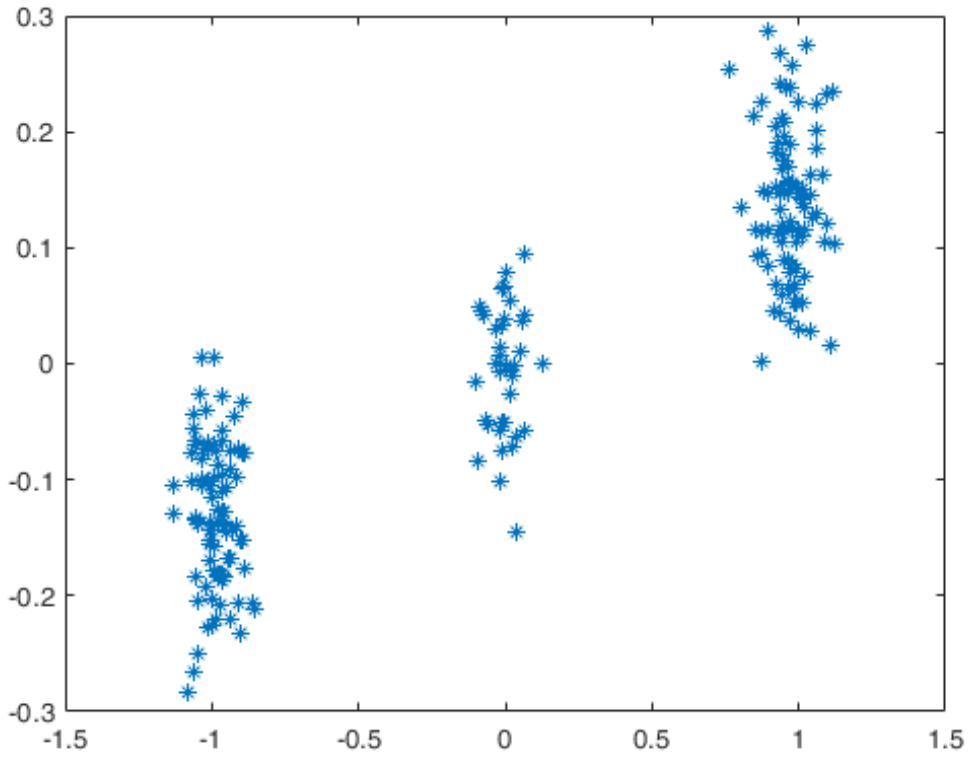


Figure 3.20: Residual CFO Correction Using Pilot Sub-Carriers

Figure 3.20 shows the effect of correcting the residual CFO using pilot sub-carriers. Each sub-carrier can then be mapped to constellation points easily.

[7] https://standards.ieee.org/standard/802_11-2012.html

Figure 3.20 is obtained by plotting signal **X_eql** inside the equalize Matlab script.

```
... %execute decode script until equalize function...
 figure
 plot(real(X_eql(3,:)),imag(X_eql(3,:)),'*');
```

3.10.4 Implementation

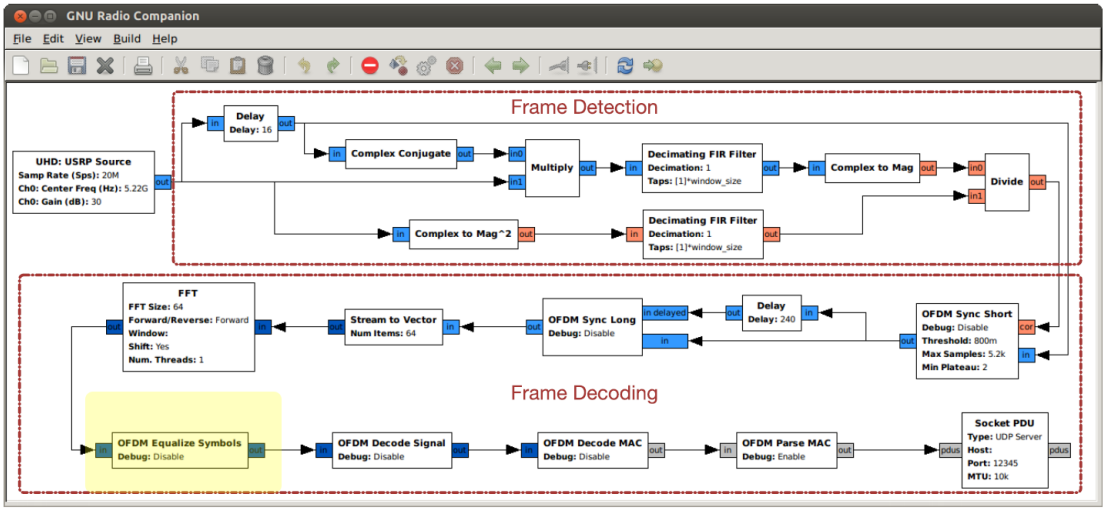


Figure 3.21: GNU Radio Blocks in charge of Symbol alignment.

Following the FFT, the OFDM Equalize Symbols block is the first one in frequency domain and is responsible for phase offset correction and channel estimation.

Equalization is performed by *equalize(fx_pt X[CHUNK], fx_pt X_eql[CHUNK])*. This function scans every sample in frames. When the 2^{nd} frame is detected (corresponding to the Long Training Sequence (LTS) the channel gain (*freq_offset*) is calculated :

```

    sumC= fx_pt_ext2(carrier[11])* fx_pt_ext2(std::conj( X[11+64] ) ) ;
    sumC+= fx_pt_ext2(carrier[25])* fx_pt_ext2(std::conj( X[25+64] ) ) ;
    sumC+= fx_pt_ext2(carrier[39])* fx_pt_ext2(std::conj( X[39+64] ) ) ;
    sumC+= fx_pt_ext2(carrier[53])* fx_pt_ext2(std::conj( X[53+64]  ) );
    // angle of offset
    fx_pt1_ext1 x=  (fx_pt1_ext1) sumC.real();
    fx_pt1_ext1 y=  (fx_pt1_ext1) sumC.imag();
    fx_pt1_ext1 raz=  y/x ;
    freq_offset=raz;
    freq_sum=0;
    //update new carrier for next cycle
    for(unsigned i=0;i<64;i++)
      carrier[i]+=fx_pt_ext2(X[64+i]);
    for(unsigned i=0;i<64;i++)
      carrier[i]=carrier[i]*(fx_pt_ext2)(ref[i],ref[i])*(fx_pt_ext2)(0.5,0.5);

    //remove pilot carrier
    o=0;
    for(unsigned i=6; i<=10;i++)
      carrier_data[o++]=carrier[i];
    for(unsigned i=12; i<=24;i++)
      carrier_data[o++]=carrier[i];
    for(unsigned i=26; i<=31;i++)
      carrier_data[o++]=carrier[i];
    for(unsigned i=33; i<=38;i++)
      carrier_data[o++]=carrier[i];
    for(unsigned i=40; i<=52;i++)
```

```
        carrier_data[o++]=carrier[i];
      for(unsigned i=54; i<=58;i++)
        carrier_data[o++]=carrier[i];
      packet++;

```

Listing 3.25: Equalization -estimation of the channel gain-

Residual frequency correction is done by following procedure described in **??**; the frequency shift is stored in the *a_sumC* variable:

```
      unsigned idx=((packet-2)%127)+1;
      fx_pt1 p=POLARITY[idx];
      sumC= fx_pt_ext2(carrier[11])* fx_pt_ext2(std::conj( X[11+64*packet] ) )*
    fx_pt_ext2(p,0);
      sumC+= fx_pt_ext2(carrier[25])* fx_pt_ext2(std::conj( X[25+64*packet] ) ) *
    fx_pt_ext2(p,0);
      sumC+= fx_pt_ext2(carrier[39])* fx_pt_ext2(std::conj( X[39+64*packet] ) ) *
    fx_pt_ext2(p,0);
      sumC+= fx_pt_ext2(carrier[53])* fx_pt_ext2(std::conj( X[53+64*packet]  ) )* -
    fx_pt_ext2(p,0);

      fx_pt1_ext1 x=  (fx_pt1_ext1) sumC.real();
      fx_pt1_ext1 y=  (fx_pt1_ext1) sumC.imag();
      fx_pt1_ext1 a_sumC=  y/x ;

      carrier[11]= fx_pt_ext2( X[11+64*packet]  ) * fx_pt_ext2(p,0);
      carrier[25]= fx_pt_ext2( X[25+64*packet]  ) *fx_pt_ext2(p,0);
      carrier[39]= fx_pt_ext2( X[39+64*packet]  ) *fx_pt_ext2(p,0);
      carrier[53]= fx_pt_ext2( X[53+64*packet]  ) *-fx_pt_ext2(p,0);
```

Listing 3.26: Equalization - Residual frequency correction

Next, the Pilot (that have only functional purpose and do not carry any information) are removed from all the packet in order to decode the message in the next blocks.

```
      fx_pt_ext2 X_med[64];
      fx_pt_ext2 X_eql_ext[CHUNK];
      //remove pilot carrier
      o=0;
      for(unsigned i=6; i<=10;i++)
        X_med[o++]=X[i+64*packet];
      for(unsigned i=12; i<=24;i++)
        X_med[o++]=X[i+64*packet];
      for(unsigned i=26; i<=31;i++)
        X_med[o++]=X[i+64*packet];
      for(unsigned i=33; i<=38;i++)
        X_med[o++]=X[i+64*packet];
      for(unsigned i=40; i<=52;i++)
        X_med[o++]=X[i+64*packet];
      for(unsigned i=54; i<=58;i++)
        X_med[o++]=X[i+64*packet];
```

Listing 3.27: Equalization - Pilot Removal-

Finally the channel gain (*freq_offset*) and the residual offset correction (based on the pilot) *a_sumC* is applied to every sample (3.10.3) :

```
   const fx_pt1_ext1 alpha=0.9;
      freq_offset=((1-alpha)*freq_offset)+(alpha*a_sumC);
      freq_sum+=freq_offset;

      // compensate offset carrier + channel model
      fx_pt_ext2 esp = fx_pt_ext2( hls::cos(freq_sum), hls::sin(freq_sum));
      for(unsigned i=0;i<48;i++)
      {
        X_eql_ext[i]= (X_med[i]*esp)/carrier_data[i];
        X_eql[i+48*(packet-2)]=(fx_pt)(X_eql_ext[i]);
        smpl++;
      }
      packet++;
    } //fine if-elseif- else
```

Listing 3.28: Equalization - Frequency offset correction-

Matlab Implementation

```
% input -> X (fft reverse of symbol - including pilots and null carrier
% output X_eql

bw=10e6;
freq=5.89e9;
ref=[0,  0,  0,  0,  0,  0,  1,  1, -1, -1, ...
   1,  1, -1,  1, -1,  1,  1,  1,  1,  1, ...
   1, -1, -1,  1,  1, -1,  1, -1,  1,  1, ...
   1,  1,  0,  1, -1, -1,  1,  1, -1,  1, ...
  -1,  1, -1, -1, -1, -1, -1,  1,  1, -1, ...
  -1,  1, -1,  1, -1,  1,  1,  1,  1,  0, ...
   0,  0,  0,  0 ].;

POLARITY=[ 1, 1, 1, 1,-1,-1,-1, 1,-1,-1,-1,-1, 1, 1,-1, 1, ...
  -1,-1, 1, 1,-1, 1, 1,-1, 1, 1, 1, 1, 1, 1,-1, 1,...
   1, 1,-1, 1, 1,-1,-1, 1, 1, 1,-1, 1,-1,-1,-1, 1,...
  -1, 1,-1,-1, 1,-1,-1, 1, 1, 1, 1, 1,-1,-1, 1, 1,...
  -1,-1, 1,-1, 1,-1, 1, 1,-1,-1,-1, 1, 1,-1,-1,-1,...
  -1, 1,-1,-1, 1,-1, 1, 1, 1,-1, 1,-1, 1,-1, 1,1,...
  -1,-1,-1,-1,-1, 1,-1, 1, 1,-1, 1,-1, 1, 1, 1,-1,...
  -1, 1,-1,-1,-1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1  ].;


pilot=[12 26 40 54].;
data_carrier=[7:11 13:25 27:32 34:39 41:53 55:59];

alpha=0.9;

[nX mX]=size(X);
c=1;
for i=1:mX

    if i==1
        carrier=X(:,1);
    elseif i==2

        %sumC= sum(carrier(pilot).*conj( X(pilot,i) ) );
        sumC= (carrier(12)*conj(X(12,2)) )+ ...
              (carrier(26)*conj(X(26,2)) )+ ...
              (carrier(40)*conj(X(40,2)) )+ ...
              (carrier(54)*conj(X(54,2)) );

        freq_offset=angle(sumC);
        freq_sum=0;
        %defining new carrier for next symbol
        carrier=carrier+X(:,i);
        carrier=carrier.* ref.*0.5;

        str=['i= ' num2str(i) ', sum=' num2str(sumC) ' freq_off=' num2str(
    freq_offset)];
    else
        p=POLARITY( mod((i-3),127)+1 ) ;
        sumC=sum( carrier(pilot) .* conj( X(pilot,i) ) .* [1 1 1 -1]'.*p );

        str=['freq_offset=' num2str(freq_offset) ',   freq_sum=' num2str(sumC)];

        freq_offset= ( (1-alpha)*freq_offset ) +( alpha*angle(sumC) );
        freq_sum=freq_sum+freq_offset;

        carrier(pilot)=X(pilot,i).*[1 1 1 -1]'.*p;
        X_eq0(:,c)=(X(data_carrier,i)./carrier(data_carrier) );
        X_eql(:,c)=(X(data_carrier,i)./carrier(data_carrier) ).*exp(1i*freq_sum);
        c=c+1;
    str=['i= ' num2str(i) ',   freq_sum=' num2str(freq_sum)];
    end
end

X_eql_lin=reshape(X_eql,[],1);
```

Listing 3.29: equalizer Matlab Function

3.11 Decoding

Now we have corrected the residual CFO and also have corrected the channel gain, the next step is to map the FFT output to actual data bits. This is the reverse process of encoding a packet.

1. demodulation: complex number to bits
2. deinterleaving: shuffle the bits inside each OFDM symbol
3. Convolution decoding: remove redundancy and correct potential bit errors
4. Descramble.

Step 1 and 3 depend on the modulation and coding scheme, which can be obtained from the SIGNAL field. The SIGNAL field is encoded in the first OFDM symbol after the long preamble and is always BPSK modulated regardless of the actual modulation. Recall that in 802.11p, one OFDM symbol contains 48 data sub-carriers, which corresponds to 48 data bits in BPSK scheme. The SIGNAL field is also convolutional encoded at 1/2 rate so there are 24 actual data bits in the SIGNAL field.

Demodulation

This step maps the complex symbol come from the FFT plane into bits. Many modulation scheme could be used for the 802.11p protocol; however, in this thesis, only Binary Phase Shift Keying is implemented. [8]

Binary Phase Shift Keying (BPSK)

In BPSK, "0" and "1" are represented by:

$$s_0 = A_{pt}(t) \cdot cos(2\pi f_c t + \theta$$
$$s_1 = -A_{pt}(t) \cdot cos(2\pi f_c t + \theta$$

Equivalently, we can write:

$$s_0 = A_{pt}(t) \cdot cos(2\pi f_c t + \theta$$
$$s_1 = A_{pt}(t) \cdot cos(2\pi f_c t + \theta + \pi$$

In this representation, we see that the information is carried by the phase of the carrier, hence, the name phase shift keying.
The average bit error probability (for equally probable binary signals) is:

$$P_b = Q(\sqrt{2E_b/N_0})$$

To avoid the complexity involved in calculating the exact error probability, we use the dominant terms in the union bound (the terms due to the **nearest neighbors** in the signal constellation) to compare the power efficiency of different modulation schemes.

In other word, we decode the received signal by mapping it to the closest one in the set of possible transmitted signals as explained in the following list.

- Map to the closest constellation point
- Quantitative measure of the distance between the received signal s' and any possible signal s (Find $|s' - s|$ in the I-Q plane)

Figure 3.22 shows an example of BSPK demodulation using the ńearest one ḿethod.

$$n_1 = |s' - s| = |s' - (1 + 0i)$$
$$n_0 = |s' - s| = |s' - (-1 + 0i)$$

Since $n_1 < n_0$, s_1 is mapped to $(1 + 0i) \rightarrow 1$

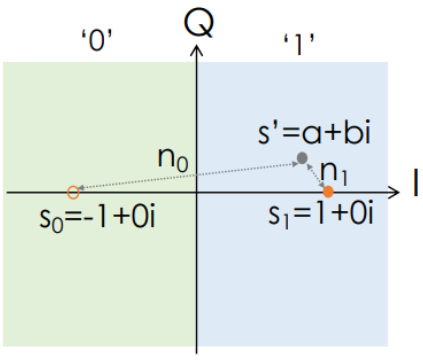


Figure 3.22: Example of BPSK demodulation

[8] This is because 802.11p is only a case study. It is out of the scope of this project develop a complete 802.11p architecture.

Deinterleaving

Inside each OFDM symbol, the encoded bits are interleaved. To understand how the block interleaver works, first we need to define a few parameters.

Let $s = max(N_{BPSC}/2, 1)$ be the number of bits along the real (or imaginary) axis in the constellation plane. The interleaver is based on writing the data bits in rows and reading them out in columns.

The interleaving process involves two permutations. Let k be the index of the bit index before the first permutation, i be the index after the first but before the second permutation, and j be the index after the second permutation.
The first permutation ($k \rightarrow i$) of interleaving ensures adjacent code bits are mapped to non-adjacent sub-carriers, and is defined as:

$$i = N_{ROW} \times (k \bmod N_{COL}) + \lfloor \frac{k}{N_{COL}} \rfloor$$

And the second permutation ($i \rightarrow j$) ensures that adjacent code bits are mapped alternatively to less or more significant bits in constellation point, and is defined as:

$$j = s \times \lfloor \frac{i}{s} \rfloor + (i + N_{CBPS} - \lfloor N_{COL} \times \frac{i}{N_{CBPS}} \rfloor) \bmod s$$

First, to reverse the second permutation

$$i = s \times \lfloor \frac{j}{s} \rfloor + (j + \lfloor N_{COL} \times \frac{j}{N_{CBPS}} \rfloor) \bmod s$$

and to reverse the first permutation:

$$k = N_{COL} \times i - (N_{CBPS} - 1) \times \lfloor \frac{i}{N_{ROW}} \rfloor$$

Viterbi Decoding

Convolution codes are better codes of error controlling performance. Convolutional encoder outputs are not only associated with the encode elements at present, but also affected by several ones before. (n, k, m) is used for describing convolutional codes, where k are the input encode elements, n are the output encode elements and m are the shift register numbers of convolution encoder. Usually, the value of n and k is smaller and k is less than n, but the number of shift registers takes larger value. In 802.11p standards the convolutional codes are defined by given generator polynomials with constraint length of 7 and code rate of $\frac{1}{2}$, resulting in 64 trellis states. The given generator polynomials codes are $G1 = 133(OCT)$ and $G2 = 171(OCT)$, equally

$$G_1 = x_0 + x_2 + x_3 + x_5 + x_6 = 1011011 = 133_8$$
$$G_2 = x_0 + x_1 + x_2 + x_3 + x_6 = 1111001 = 171_8$$

Viterbi Algorithm

Convolutional codes are a class of error-correcting codes. They are widely used in modern telecommunication systems, e.g. in IEEE 802.11 [73], IEEE 802.16 [74], UMTS [75], LTE [76], etc

Convolutional codes can be decoded by several methods, e.g. Viterbi algorithm, sequential decoding, BCJR algorithm. The Viterbi algorithm (VA) [77] is one of the most common approaches, it is recommended in all the above mentioned standards. It is also implemented in Mathworks Simulink Communications Systems Toolbox [78] and GNU Radio [79] which are widely used for experiments and a performance evaluation of digital communication systems by the research community.

The VA was firstly identified as the maximum likelihood sequence estimator in [80]. This section recalls the major aspects of the algorithm, which are relevant for the considered problem. In general the VA finds the most likely message of states of a Hidden Markov Model, called a path, which produces a particular observed output. Each decoding step the VA produces several candidate branching paths, which lead to the same output state. This is illustrated in in Fig. 3.23.

Each branch is characterized by a branch metric, which when summed up result in a path metric. The algorithm selects the path with the minimal path metric.

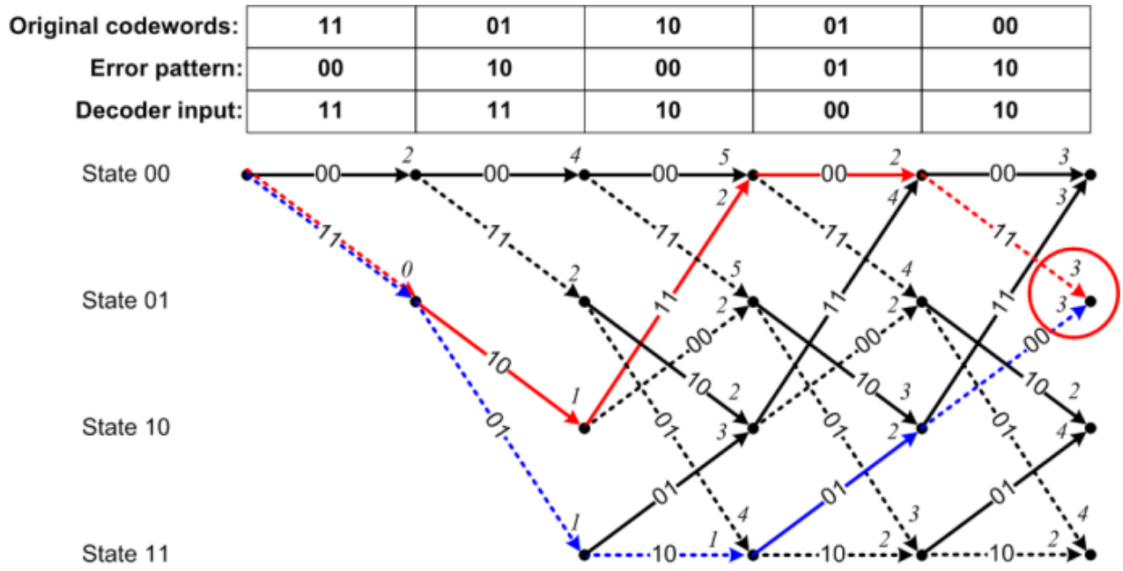


Figure 3.23: Paths metrics equality.

However, VA is out of the scope of this thesis so no further investigation is made and it was used the HLS Viterbi core provided by Xilinx.
For the detailed description of the Viterbi algorithm as well as its extensive performance analysis in the context of decoding of convolutional codes over a noisy communication channel the interested reader is referred to [77] [81].

Descrambling

The scrambling step at the transmitter side is to avoid long consecutive sequences of 0s or 1s. The scrambling and descrambling process can be realized using the same logic.

Suppose the current input bit is B_n, the scrambled bit B_n^s and the internal state of the scrambler is updated as follows:

$$\begin{aligned} B_n^s &\leftarrow X_n^1 \oplus B_n \\ X_{n+1}^1 &\leftarrow X_n^7 \oplus X_n^4 \\ X_{n+1}^i &\leftarrow X_n^{i-1}, i = 2, 3, \ldots, 7 \end{aligned}$$

where X_n^i is the scrambler state before the nth input bit, $n = 0, 1, 2, ...$

At the transmitter side, for each packet, the scrambler is initialized with pseudo random value. The very first 7 bits of the data bits is preset to zero before scrambling, so that the receiver can estimate the value using the scrambled bits.

3.11.1 Implementation

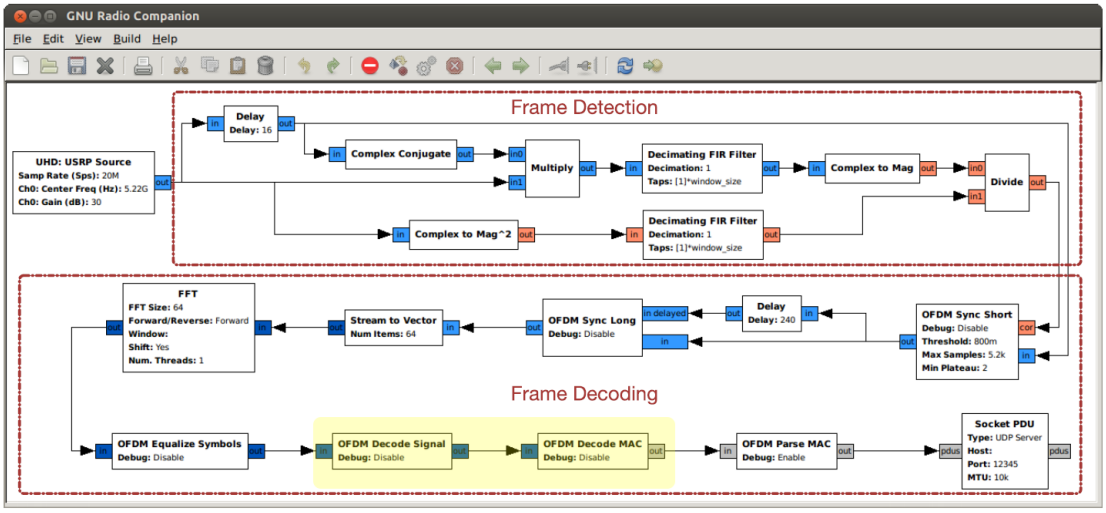


Figure 3.24: GNU Radio Blocks in charge of decoding bits.

Decoding

Decoding is performed by mapping bits to the nearest one:

```
for(unsigned i=0;i<384;i++)
  if( constellation[i].real()>0 )
    bit_r[i]=(ap_uint<1>)1;
  else
    bit_r[i]=(ap_uint<1>)0;
```

Listing 3.30: Decoding bits

Deinterleaving

Deinterleaving is performed by looking a look-up table (variable inter) and exchanging the position of the bits:

```
const unsigned inter[]=
     { 0,3,6,9,12,15,18,21,24,27,30,33,36,
        39,42,45,1,4,7,10,13,16,19,22,25,28,
        31,34,37,40,43,46,2,5,8,11,14,17,20,23,
        26,29,32,35,38,41,44,47 };

  for(unsigned i=0;i<7;i++)
    for(unsigned j=0;j<48;j++)
    {
      unsigned index=inter[j]+i*48;
      bit[ j+i*48 ]=bit_r[index];

    }
```

Listing 3.31: Deinterleave process

Viterbi Decoding

Decoding of the signal is performed by the Xilinx HLS IP Viterbi decoder after formatting the data according to the inputs of the IP. We utilize the Viterbi HLS IP core provided by Xilinx. The summary features for the Viterbi Decoder IP core are as follows:

1. Parameterizable decoder rates, constraint length, convolution codes and traceback lengths.
2. Choice of either parallel architecture for high data throughput, or serial for smaller area footprint.

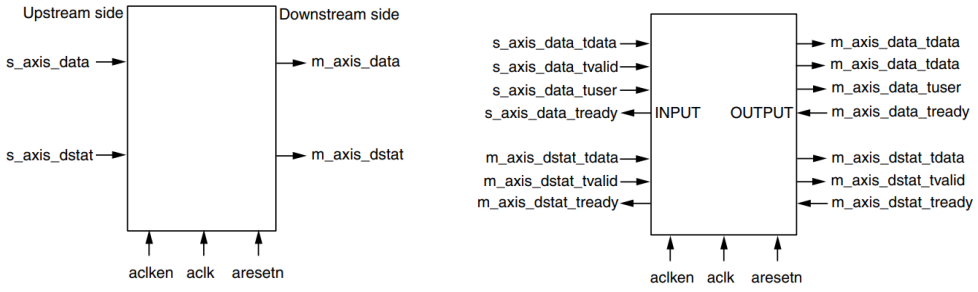
3. Very low latency option.

4. Soft decision with parameterizable soft width

5. Other architectural options such as multichannel decoding, dual rate decoder or trellis mode

6. Erasure for external puncturing

A representative symbol of the Viterbi Decoder, with the signal names, is shown in Figure 3.25a and Figure 3.25b. Some of the pins are optional. These should be selected only if they are genuinely required, as their inclusion might result in an increase in the core size.



(a) Core AXI Channels. (b) Core Schematic Symbol.

Figure 3.25: Viterbi Decoder - Xilinx IP Core - .

```
hls::stream< ap_uint<1> > bits_in("Bits In");
  hls::stream< hls::viterbi_decoder_input<OutputRate,InputDataWidth,HasEraseInput>
    > input_data("Input Data");
 // hls::stream< ap_uint<1> > output_data("Output Data");
  hls::viterbi_decoder_input<OutputRate,InputDataWidth,HasEraseInput> input;
  ap_uint<OutputRate> encoded_data;

  for(unsigned i=0;i<4*48;i=i+2)
  {
    encoded_data[0]=bit[i];
    encoded_data[1]=bit[i+1];
    input.data= encoded_data;
    input_data<< input;
  }
  while(!input_data.empty()) {
    viterbi_decoder_top(input_data, output_data);
```

Listing 3.32: complex filter

Matlab Implementation

```
bits=-real(X_eql);
bits=bits<0;
      inter=[0,3,6,9,12,15,18,21,24,27,30,33,36,...
       39,42,45,1,4,7,10,13,16,19,22,25,28,...
       31,34,37,40,43,46,2,5,8,11,14,17,20,23,...
       26,29,32,35,38,41,44,47].''+1;

   [nX_eq,mX_eq]=size(X_eql);
    tmp(:,1)=bits(inter(1:nX_eq),1);
   bits=tmp;

%******** decode *******
t = poly2trellis(7,[133 171] );
tb =14;
decoded = vitdec(bits,t,tb,'trunc','hard') .'';

% interpretate signal
decoded=decoded(1:19);

```

```
prate=  sum(decoded(1:4).*2.^(3:-1:0));
plength=sum(decoded(6:17).*2.^( 0:11 ) );
pparity=decoded(18);

switch prate
    case 13
        rate_d=1;
        disp('-)Codec: BPSK 1/2');
    case 15
        rate_d=2;
        disp('-)Codec: BPSK 3/4');
    case 5
        rate_d=2;
        disp('-)Codec: QPSK 1/2');
   case 7
        rate_d=2;
        disp('-)Codec: QPSK 3/4');
   case 9
        rate_d=2;
        disp('-)Codec: 16-QAM 1/2');
   case 11
        rate_d=2;
        disp('-)Codec: 16-QAM 3/4');
   case 1
        rate_d=2;
        disp('-)Codec: 64-QAM 2/3');
   case 3
        rate_d=2;
        disp('-)Codec: 64-QAM 3/4');
end
str=['-)Length=' num2str(plength)];
disp(str);
```

Listing 3.33: equalizer Matlab Function

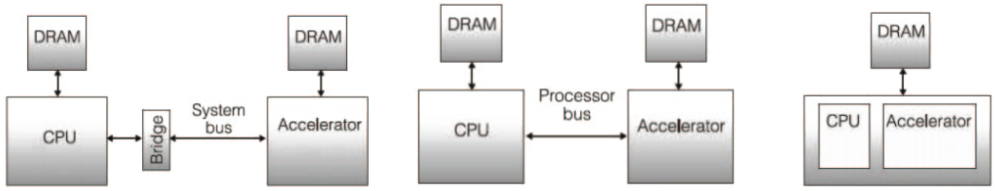
Chapter 4

Evaluation

> *First law: The pesticide paradox. Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffective.* – Boris Beizer

4.1 Xilinx Zynq

Zynq SoC is a product line from Xilinx where a processor and programmable logic is implemented on the same silicon die. More specifically it is an ARM Cortex-9 dual-core processor connected to various sized amounts of programmable logic dependent on the chip model. In addition to the ARM Dual-core processor there is also an ARM NEON unit, which is a general-purpose Single-Instruction Multiple-Data(SIMD) engine.
The combination of a general-purpose processor and programmable logic on the same silicon die allows realization of hardware accelerators with at higher data-throughput, as the communication bus between the two units is as short as possible.
The three most common types of hardware accelerator architectures are depicted in Figure 4.1. Each of these have their own pros and cons, with type 4.1c being the fastest architecture due to the communication bus between CPU and accelerator being physically short. Reducing the length of a bus reduces the capacitance of the bus, which in return makes it easier to obtain a higher data throughput on the bus. A normal computer would allow implementation of the type 4.1a architecture through the PCIe-bus while the Zynq SoC makes a type 4.1b architecture implementation possible.



(a) System Bus Connection (b) Processor Bus Connection (c) Tightly Coupled Accelerator

Figure 4.1: The three general hardware accelerator structures.

The Zynq SoC system diagram is depicted in Figure 4.2. As it can be observed the ARM Cortex-9 processor is connected to the programmable logic of the Zynq SoC through AXI-interconnects and Extended Multi-use I/O (EMIO). EMIO will not be used in this project, but is useful if one want to use some of the builtin communication modules such as SPI, I2C etc. to communicate with external hardware or hardware implemented in the programmable logic.

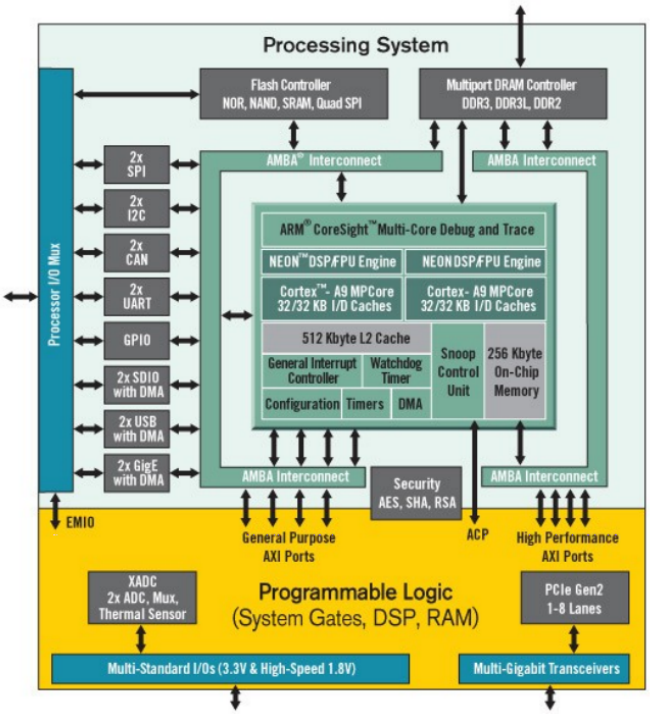


Figure 4.2: System diagram of the Zynq SoC

4.1.1 Xilinx zc706

A Xilinx zc706 evaluation Board has been used for validation purpose.
The ZC706 evaluation board for the XC7Z045 SoC provides a hardware environment for developing and evaluating designs targeting the Zynq-7000 XC7Z045-2FFG900C SoC. The ZC706 evaluation board provides features common to many embedded processing systems, including DDR3 SODIMM and component memory, a four-lane PCI Express interface, an Ethernet PHY, general purpose I/O,

and two UART interfaces. Other fea-
tures can be supported using VITA-57 FPGA mezzanine cards (FMC) attached to the low pin count (LPC) FMC and high pin count (HPC) FMC connectors.
A detailed description of the component installed on the board is resumed in the Figure 4.3.

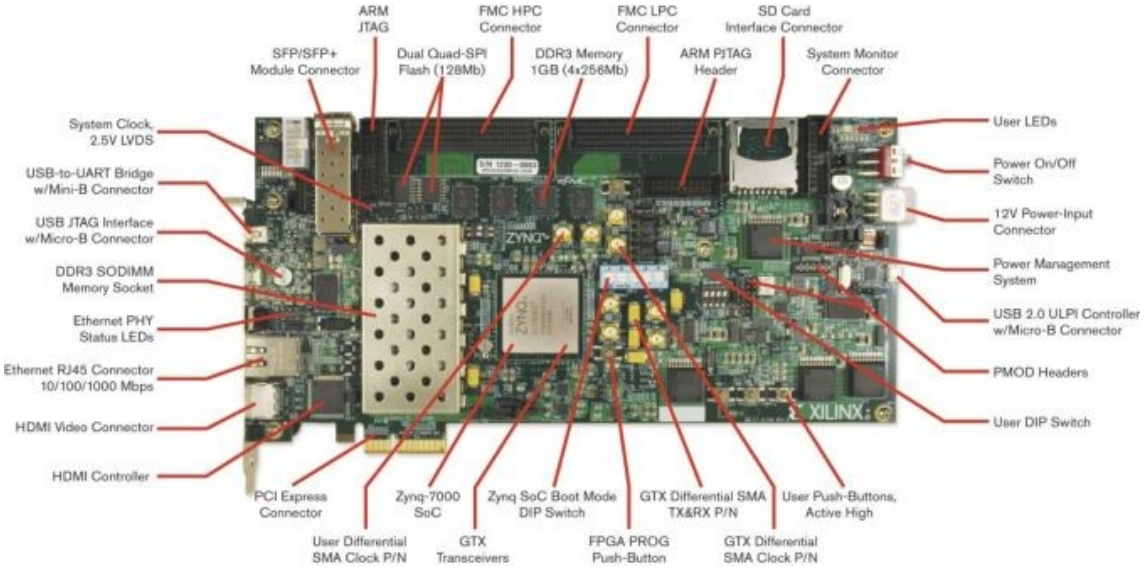


Figure 4.3: Hardware of the Xilinx zc706 evaluation board

The ZC706 evaluation board is populated with the Zynq-7000 XC7Z045-2FFG900C SoC. The XC7Z045 SoC consists of an integrated processing system (PS) and programmable logic (PL), on a single die. The high-level block diagram is shown in 4.4.

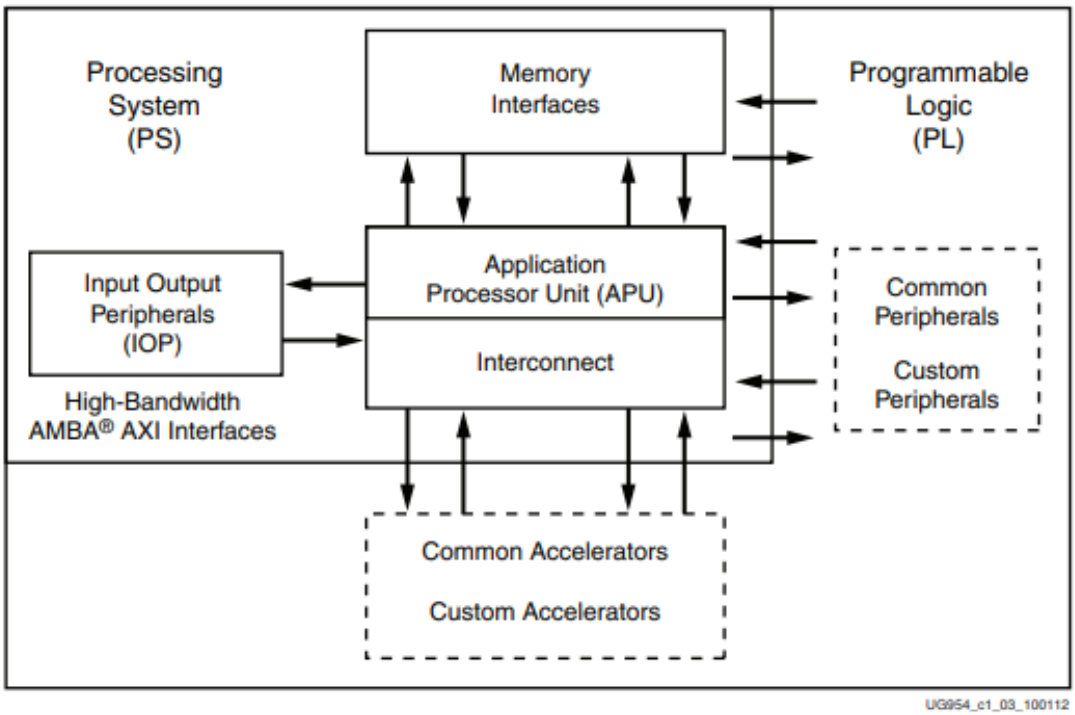


Figure 4.4: High-Level Block Diagram of the Zynq 7000

The PS integrates two ARM Cortex-A9 MPCore application processors, AMBA interconnect, internal memories, external memory interfaces, and peripherals including USB, Ethernet, SPI, SD/SDIO, I2C, CAN, UART, and GPIO. The PS runs independently of the PL and boots at

power-up or reset.

4.2 AXI interfaces

The Xilinx Vivado software package from Xilinx is the recommended development environment when working with the Zynq SoCs. The Vivado environment is highly geared towards using Intellectual Property (IP) and block-based design to increase productivity and lower development time. Vivado features several necessary IP blocks and templates for Zynq and FPGA-designs, several of these blocks are related to or used for implementation of Xilinx's AXI-bus and configuration of the ARM processor.
Adding custom RTL to a project is done by creating IP-blocks from templates and using these in the block design of the project. As mentioned in section 4.1 the Zynq SoC is internally using Xilinx's AXI-bus to connect the ARM processor with the programmable logic of the Zynq SoC, and as such it is preferable to use the AXI-bus to connect the custom RTL to the processor of the system using the AXI-bus. Other connection possibilities exist though, these are however not as fast as the AXI-bus and it is thus up to the developer to choose a suitable bus for communication between the RTL and the ARM processor.
When creating a new IP peripheral in the Vivado design suite, the user has the choice of adding AXI-connections of three types, Full, Lite and Stream, these can be added either as a slave or master connections. When creating the peripheral a template is given to the user with the basic control logic for the added AXI-bus.

4.3 Implementation

In this project, the IEEE 802.11p algorithm accelerator is implemented. Furthermore the configuration of the ARM processor is also explained, as the ARM-processor was used for both the hardware and software execution of each benchmark. Implementation of both processors was done in the Xilinx Vivado development environment, as this is the intended development environment for the Zynq SoC. The Vivado software package is shipped with several Intellectual Property packages for development on different FPGAs, but most importantly several IPs related to AXI-bus communication. In the implementation of the hardware accelerators, AXI-related IP was used for connecting the CPU and accelerator, as well as connecting the accelerator with the external RAM. The implementation of the hardware accelerators consists of a hardware implementation mapped to the logic part and an embedded software application running on the ARM processor controlling the accelerator.
Due to some parts of the overall architecture which their software implementation is out of the scope of this thesis (like the software implementation of the Viterbi decoder; In hardware it was used an existing, ready to use Xilinx IP) only the frame detection part of the 802.11p algorithm is taken into consideration for timing reports and acceleration speed-up.

4.3.1 Hardware Part

Implementation of the 802.11p accelerator was split up into three tasks; packaging the 802.11p architecture into an IP package with an AXI-interface, configuration of the ARM processor system, and connection of the accelerator IP to the processor system.

Creating the802.11p IP was done in the Vivado design suite, where Vivado HLS was used to generate a template for an AXI4-peripheral with tree AXI interfaces operating one in slave-mode (control bus) and two in master mode(destination and source bus).

After this the top-level module of the 802.11p was implemented as a peripheral, where the input clock was wired to the clock of the AXI slave interface while the reset signal was connected to a register instead of the global reset, such that the accelerator can be reset without resetting the entire programmable logic fabric on the Zynq SoC. Finally the 802.11p peripheral was synthesized and packaged into an IP entity.

Configuration of the ARM processor system was done by creating a new project in Vivado and adding an instance of the processor system IP to the system design. The processor IP was then configured to have only the necessary modules to connect with the generated 802.11p IP.

This means that the processor system was configured to have a dual AXI-bus connection to the programmable logic, and to provide a clock running at 150MHz (this was the highest frequency achievable without timing errors) to the programmable logic.
After this the 802.11p IP was added to the design, and connected to the processor system on the Zynq SoC. An automatic connection wizard of Vivado was run, this added a couple of AXI related IP blocks to the system, blocks which synchronize the reset signals and connect the IP AXI bus the AXI bus of the processor system. The system overview as seen in Vivado is depicted in Figure 4.5.

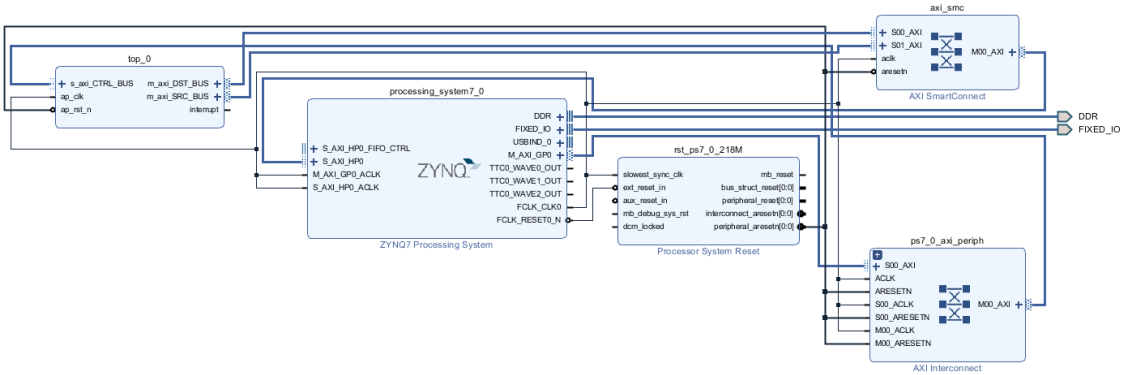


Figure 4.5: Top-level system overview of the 802.11p implementation on Zynq as seen in Vivado

4.3.2 Software Part

Memory location of the source data (raw data to be processed corresponding to the RF stream coming from the Radio Frequency front-end), destination data (data elaborated by the hardware accelerator) and golden data (data elaborated via software by the hard core ARM processor) are chosen to not collide with other data in the following location:

```
#define MAXDATA (5*4096*sizeof(unsigned) )
#define SRC_BUFFER_BASE 0x20000000//(DATA_OFFSET)
unsigned *src_mem_real =(unsigned *)SRC_BUFFER_BASE;

#define SRC_img_BUFFER_BASE 0x22000000
unsigned *src_mem_img =(unsigned *)SRC_img_BUFFER_BASE;


#define HP_DST_BUFFER_BASE 0x23000000//(DATA_OFFSET + 4*MAXDATA)
unsigned *dst_mem = (unsigned *)HP_DST_BUFFER_BASE;

#define GLD_BUFFER_BASE 0x24000000
_Complex double *out = (_Complex double *)  GLD_BUFFER_BASE;
```

We perform a simple MARCH test to the memory in order to be sure that the chosen location are writable/readable and no errors appear during writing and reading process.

```
int test_memory()
{
  const unsigned uno=0xFFFFFFFF;
  const int num=20;
  const double uno_f=0xFFFFFFFFFFFFFFFF;

  for(int i=0; i<num*4096;i++)
  {
    src_mem_real[i]=0x0;
    src_mem_img[i]=0x0;
    dst_mem[i]=0x0;
    out[i]=0x0*0x0*I;
  }
  for(int k=0; k<(num*4096)-10; k++)
```

```
    {
      if(src_mem_real[k] != 0x00 ) return 1;
      if(src_mem_img[k]!=0x0) return 2;
      if(dst_mem[k]!=0x0) return 3;
      if(out[k]!=0x0) return 4;
    }
  for(unsigned i=0; i<num*4096;i++)
    {
      src_mem_real[i]=uno;
      src_mem_img[i]=uno;
      dst_mem[i]=uno;
      out[i]=uno_f+I*uno_f;
    }
  for(unsigned i=0; i<num*4096;i++)
  {
        if(src_mem_real[i]!=uno) return 5;
        if(src_mem_img[i]!=uno) return 6;
        if(dst_mem[i]!=uno) return 7;
        if(out[i]!=uno_f+I*uno_f) return 8;
  }

    return 0;

}
```

After loading the data into the memory, the software execution is performed and the time execution is saved into the variable *sw_ elapsed*:

```
      XTime_GetTime(&start);
      algo_soft();
      XTime_GetTime(&stop);
      sw_elapsed = get_elapsed_time(start, stop);
```

where the function *algo_ soft* implement a software version of 3.2.

```
void algo_soft()
{
  // SUPPORT VAR;
  _Complex double out_s1[SIZE];
  _Complex double src[SIZE];
  // INIT VARIABLE
    for(unsigned i=0;i<SIZE;i++) {
      double src_rd=  (double) fixed32_to_float(((unsigned)src_mem_real[i]),nINT);
      double src_id=  (double) fixed32_to_float(((unsigned)src_mem_img[i]),nINT);
      src[i]=src_rd+I*src_id;
    }
  /***** DELAY ********/
  const int DELAY=16;

  for (unsigned i=0;i<SIZE;i++)
  {
      int index=i-DELAY;
      if( index<0 )
          out[i]=0;
      else
         out[i] = src[index];
  }

  /****** coniugate/mult *****/

  for (unsigned i=0;i<SIZE;i++)
  {
    out[i]= creal(out[i])-I*cimag(out[i]); // omplex<T_type>(out[i].real(), -out[i
    ].imag());
    out[i]=out[i]*src[i];
  }
  /***** FILTER *******/
  _Complex double out1[SIZE];
  const int order_filter_upper=32;
  moving_average(out1,out,order_filter_upper);

  /******* ABS*****/
  for (unsigned i=0;i<SIZE;i++)
```

```
38   {
39
40       out[i]=cabs(out1[i])+0*I;
41     }
42
43   /*************************/
44   /**** lower branch*****/
45   /************************/
46   _Complex double out_s[SIZE];
47
48   for (unsigned i=0;i<SIZE;i++)
49   {
50         out_s[i]=pow(cabs(src[i]),2);
51
52   }
53
54   const int order_filter_lower=64;
55    moving_average(out_s1,out_s,order_filter_lower);
56
57    for (unsigned i=0;i<SIZE;i++)
58    {
59         out[i]=out[i]/out_s1[i];
60    }
61 }
```

Listing 4.1: software algorithm

4.4 Test Platform

The test platform is shown in Figure 4.6. It can be sintetized as follow;

1. Vivado HLS generates the RTL IP core from the *c* code. With (a) we import the IP into Vivado environment,
2. the overall architecture is synthetized in Vivado and the bitstream is uploaded into the FPGA memory -(b)-.
3. Matlab is in charge to generate, upload, download and verify the correctness of the data processed by the FPGA.

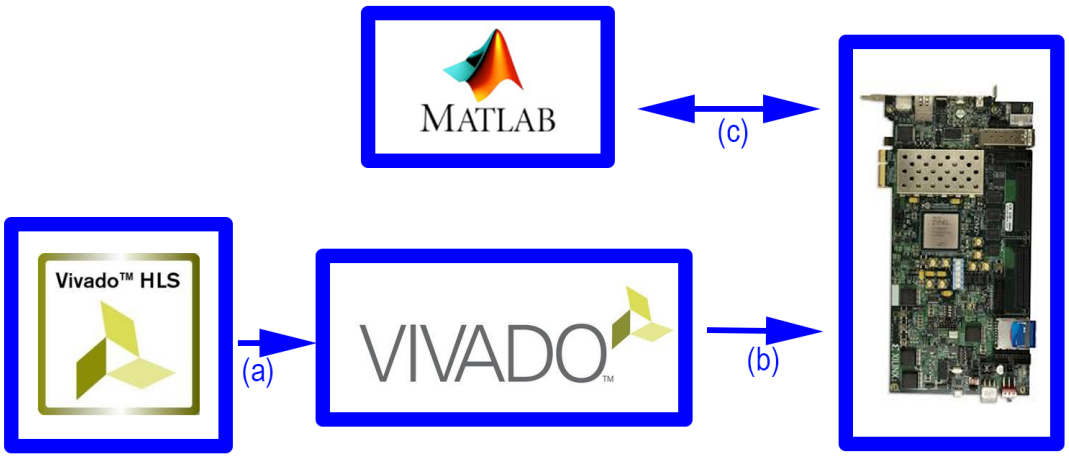


Figure 4.6: Test Platform.

Matlab Environment

It is worth spending a section on how matlab interact with the FPGA flow.
Different scripts were ad hoc written in order to accommodate the need to have a fast and easy way to interact with the FPGA. Since the algorithm has been studied and understood with Matlab, it was for me naturally to develop such flow.

- Real output data are retrieved from the 802.11p transmitter of the reference design and are saved in 2 files containing real and imaginary part of the I-Q signals.
- Matlab converts them into the proper format in order to be read correctly by the hardware accelerator.

```
1
2  clear all
3  import_data
4
5  N_Sample=5*4096;
6
7  %d=fi(d,1,N_Word,N_Word-N_integer);
8  N_Word=32;
9  N_integer=6;
10
11
12 d_fx_real=int32(real(d(1:N_Sample))* 2^((N_Word-N_integer)));
13 d_fx_imag=int32(imag(d(1:N_Sample))* 2^((N_Word-N_integer)));
14
15
16 fileID=fopen('real.bin','w');
```

```
fwrite(fileID,d_fx_real,'int32');
fclose(fileID);


fileID=fopen('img.bin','w');
fwrite(fileID,d_fx_imag,'int32');
fclose(fileID);


[status,cmdout]=system('./mem_inject.sh');
cmdout;


k=strfind(cmdout,'Successfully');
if length(k)==2
    disp(' -) Uploaled 2/2' );
end
```

Listing 4.2: Upload Matlab script

The upload process is managed by the system call *system* and executing the bash script starting the Xilinx debugger in order to extract the data:

```
    source /tools/Xilinx/Vivado/2018.3/settings64.sh
    xsdb ./mem_inject.tcl

```

```
    connect -url espdev.cs.columbia.edu:3124
    targets 1
    dow -data /home/linux/Documents/MATLAB/software_ver/real.bin 0x20000000
    after 2000
    dow -data /home/linux/Documents/MATLAB/software_ver/img.bin 0x22000000

```

- when the accelerator finishes its execution, Matlab download the data from the memory directly into its workspace in order to be compared with the golden output.

```
clear all

import_data
frame_detection

[status,cmdout]=system('./mem_dump.sh');

fileID=fopen('corhw.dat');
A=fread(fileID,'int32');

chunk=5;
Nsample=chunk*4096 ;



fclose(fileID);

N_Word=32;
N_integer=6;

cor_hw = 2^(-(N_Word-N_integer))*double(A);


[err, idx]=maxk(abs(cor_hw(1:Nsample)-cor(1:Nsample)),3);

str=[ '-) Errore = ', num2str(max(err)) '  max-> ' num2str(err(1)) 'idx='
    num2str(idx(1)) ];
disp(str);
for i=1:3
    str=[ '    -) max -> ' num2str(err(i)) 'idx=' num2str(idx(i)) ];
    disp(str)
end

a=[ (cor(1:Nsample)).'; (cor_hw(1:Nsample)).'];
```

```
subplot(2,1,1)
title('Samples')
plot(cor_hw(1:Nsample));
hold on
plot(cor(1:Nsample));
legend('HW data','MATLAB reference');
title('Time Domain Data')
subplot(2,1,2);
plot(std(a));
title('Standard Deviation');
```

Listing 4.3: Download and comparison Matlab script

Again, the download process is managed by the system call *system* and executing the bash script starting the Xilinx debugger in order to extract the data.

```
    source /tools/Xilinx/Vivado/2018.3/settings64.sh
    xsdb ./dump.tcl

```

```
    connect
    targets 1
    mrd -file /home/linux/Documents/MATLAB/software_ver/corhw.dat -bin 0
    x23000000 0x10000

```

4.5 Profiling

When it comes to GPP-based SDR systems, the computational performance and the ability to process samples in real time are critical factors. In this context, we refer to the term real-time signal processing to contrast offline signal processing. It implies that the PC is able to keep up with the incoming sample stream without dropping samples. In other words, the average processing time per sample is smaller than the sample duration. This property is crucial, since, otherwise, the transceiver would have to drop samples, which causes packet loss. Ultimately, this could lead to wrong interpretations of measurement results if lost frames are regarded as effects of the wireless channel or shortcomings of receive algorithms. Given the importance, we have an in-depth look at the computational complexity and real-time capabilities of our system.
In that regard, the transmit side is not critical as the whole waveform can be computed ad priori and streamed to the SDR. The only requirement is that the stream does not stall, which is, however, no problem in practice. The receive side is much more challenging as it has to process a large number of samples in real-time. An IEEE 802.11p channel with a bandwidth of 10MHz, for examples, results in $10 \cdot 10^6$ complex floating point numbers per second.

To cope with such high bandwidths, GNURadio starts each signal processing block in its own thread. During runtime, each block monitors performance related metrics like CPU time and fill state of the input and output queues. Depending on the hardware platform and the operating system, there are several methods available to log the CPU time. We use the accurate thread clock, which takes into account only the time when the thread was actually scheduled by the operating system.

So, the goal of this stage was to determine the percentage of total runtime consumed by each block. It is called High level profiling, because the level of abstraction in this case is fairly general, since the collected information corresponds to GNU Radio blocks and not to code statements executing within the blocks. **We wish to use the official built-in GNU Radio profiler called GR Control Port Monitor and GR Performance Monitor but the tools are somehow still buggy and an open issue is still opened to the GNU Radio Official mailing list.**

ControlPort is a new tool that creates an integrated remote procedure call interface to GNU Radio. Any block or part of GNU Radio can now register interfaces with ControlPort, which then allows a ControlPort client to interact with the GNU Radio application through any one of these interfaces. The general form of these interfaces is a set and/or get function to adjust or query the state of a GNU Radio block's parameter. For instance, a phase lock loop can have an interface to set the loop bandwidth so as to optimize the acquisition of different signals.

GNU Radio's Performance Counters are a way of generating performance measurements for each block running in a GNU Radio flowgraph. These counters keep track of various states of each block that can then be used for analyzing the performance and behavior of a running flowgraph. Currently, there are five identified performance counters:

- **noutput items**: number of items the block can produce.
- **nproduced**: the number of items the block produced.
- **input_buffers_full**: % of how full each input buffer is.
- **ouptut_buffers_full** % of how full each output buffer is.
- **work_time**: number of CPU ticks during the call to general work().

When calculating the performance counters, each block keeps track of the most recent value (i.e., the instantaneous value) as well as the running average and running variance of the counter. We can retrieve all three values programmatically using the function calls on the blocks themselves, described in the GNU Radio manual.

However, the reported bug is only a graphical issue since the profiler still work reliably. Therefore, it is created a python script (Listing 4.4) in order to extract the data from GR performance monitor; Results obtained by simulating different configurations (constraining the number of available core and trying different sample rate for input data) are shown in Table 4.1 and graphically visualized in Figure 4.7.

```
1  import sys, time
2
3  from gnuradio import gr, ctrlport
4  from subprocess import Popen, PIPE
5
6  class MyApp(object):
7      def __init__(self, args):
8          p = gr.prefs()
9          cp_on = p.get_bool("ControlPort", "on", False)
10         cp_edges = p.get_bool("ControlPort", "edges_list", False)
11         pcs_on = p.get_bool("PerfCounters", "on", False)
12         pcs_exported = p.get_bool("PerfCounters", "export", False)
13         if(not (pcs_on and cp_on and pcs_exported and cp_edges)):
14             print("Configuration has not turned on all of the appropriate
       ControlPort features:")
15             print("\t[ControlPort] on = {0}".format(cp_on))
16             print("\t[ControlPort] edges_list = {0}".format(cp_edges))
17             print("\t[PerfCounters] on = {0}".format(pcs_on))
18             print("\t[PerfCounters] export = {0}".format(pcs_exported))
19             exit(1)
20
21         from gnuradio.ctrlport.GNURadioControlPortClient import
       GNURadioControlPortClient
22         GNURadioControlPortClient(args, 'thrift', self.run)
23
24     def run(self, client):
25         input_name = lambda x: x+"::avg input % full"
26
27         tmplist = []
28         knobs = client.getKnobs([])
29         for k in knobs:
```

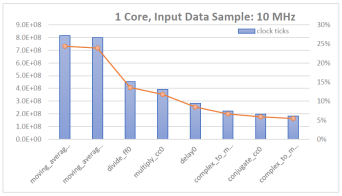
```
            propname = k.split("::")
            blockname = propname[0]
            keyname = propname[1]
            if(blockname not in tmplist):
                # only take gr_blocks (no hier_block2)
                if(knobs.has_key(input_name(blockname))):
                    tmplist.append(blockname)

        blocks = tmplist
        blocks.sort()

        # print csv header
        print(",".join(blocks))

        #knobs = map(lambda x: client.Knob("%s::reset_perf_counters" % x), blocks)
        #client.setKnobs(knobs)

        # let the flow graph run for some time
        time.sleep(1)
        ## get work time for all blocks
        #kl = map(lambda x: "%s::total work time" % x, blocks)
  #kl = map(lambda x: "%s::avg input %% full" % x, blocks)
  #kl = map(lambda x: "%s::avg throughput" % x, blocks)
  kl = map(lambda x: "%s::avg work time" % x, blocks)

        wrk_knobs = client.getKnobs(kl)

        work_times = dict(zip(
                    map(lambda x: x.split("::")[0], wrk_knobs.keys()),
                    map(lambda x: x.value, wrk_knobs.values())))

        print(",".join(map(lambda x: str(x[1]), sorted(work_times.items()))))

        p = Popen("uptime", stdout=PIPE, close_fds=True, shell=True)
        print(p.stdout.read())

MyApp(sys.argv)
```

Listing 4.4: Profiling script

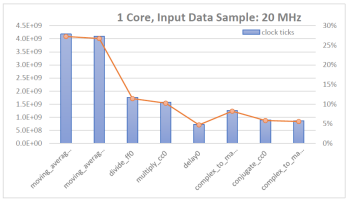
For the measurements, we ran the receiver on a laptop workstation with an Intel Xeon E3-1535M CPU and 64 GByte RAM. The operating system was Ubuntu 16.04 running a Linux 4.4.0 kernel. We compiled GNURadio and our transceiver with GCC 5.4.0 in release mode, which enables all standard compliant run time optimizations.

	Core: 1 Input Data Sample: 10 MHz		Core: 1 Input Data Sample: 20 MHz		Core: 8 Input Data Sample: 10 MHz	
	Clock ticks	% exe	Clock ticks	% exe	Clock ticks	% exe
Moving_average_cc0	815585728	24,39	4183701248	27,26	1509560448	19,94
Moving_average_ff0	798904832	23,89	4104116480	26,74	1329299328	17,56
Divide_ff0	453766976	13,57	1753872896	11,43	1045683904	13,82
Multiply_cc0	393322304	11,76	1573547520	10,25	1150029952	15,19
Complex_to_mag_squared0	221384432	6,62	1261193600	8,22	512761856	6,77
Conjugate_cc0	196575664	5,88	893310656	5,82	726618304	9,60
Complex_to_mag0	181753440	5,43	856642432	5,58	594946688	7,86
Delay0	282914846	8,46	722341440	4,71	700268440	9,25
Sum	3344208222	100	15348726272	100	7569168920	100

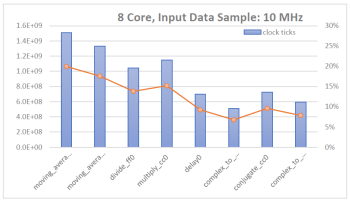
Table 4.1: GNU Radio profiling results.



(a) Profile report for 1 core, 10 MHz data rate



(b) Profile report for 1 core, 20 MHz data rate



(c) Profile report for 8 core, 10 MHz data rate

Figure 4.7: Profiling graphs

4.6 Results

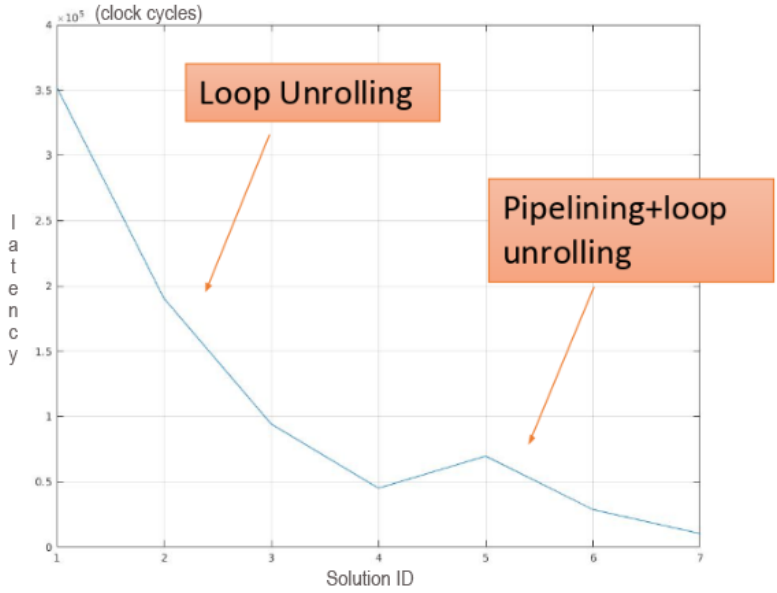


Figure 4.8: Design Space Exploration of the HW accelerator.

Figure 4.8 shows different implementation of the same accelerator (802.11p) synthetized using the HLS flows. This is achieved by acting a low-grain optimization (low grain optimization stands for optimizing the design at the lower possible layer e.g. inner loops inside functions; this is achieved e.g. optimizing last level of for loop, completely unrolling variable often accessed etc. Refer to Appendix C for further information about optimization technique used in this project). In between of this points, there are other implementations; some of them could also be Pareto-point that dominate the one drawn. We are not interested in exploring the overall design space because the way optimizations are applied could be generalized and automatically done by some heuristics considering simulation results and acting in order to accelerate critical part of the algorithm. (see section 5)
In general, the fastest implementation occupy more area than a slower one(trade-off between speed and area is defined by constrains and the particular situation; there is no 'best' implementation nor 'best' speed-up.
All the solution proposed in the Figure 4.8 are compatible with the timing constrain published by the IEEE standard.

Fig.4.9 is obtained by synthetizing the best implementation so far (again, for sure there is another 'better' implementation). Note the latency: except for some blocks (that could be further optimized) they have about the same throughput in order to balance the chain.

Instance

Instance	Module	Latency min	Latency max	Interval min	Interval max	Type
grp_sync_long_fu_1042	sync_long	8195	3972927	8195	3972927	none
grp_equalize_fu_1078	equalize	1311	62431	1311	62431	none
grp_division_fu_1094	division	4165	4166	4096	4096	loop rewind(delay=0 initiation interval(s))
grp_fft_hls_fu_1101	fft_hls	5065	5065	2978	2978	dataflow
grp_magnitude_fu_1110	magnitude	4120	4121	4096	4096	loop rewind(delay=0 initiation interval(s))
grp_firc_fu_1117	firc	4111	4255	4111	4255	none
grp_sync_short_fu_1315	sync_short	2	192623	2	192623	none
grp_fir_fu_1336	fir	4102	4230	4102	4230	none
grp_complex_mult_fu_1470	complex_mult	4099	4100	4096	4096	loop rewind(delay=0 initiation interval(s))
grp_decode_signal_fu_1482	decode_signal	5234	5234	5234	5234	none
grp_signal_power_fu_1654	signal_power	4098	4099	4096	4096	loop rewind(delay=0 initiation interval(s))
grp_delay_fu_1663	delay	4228	4292	4228	4292	none
grp_coniugate_fu_1681	coniugate	4096	4097	4096	4096	loop rewind(delay=0 initiation interval(s))
grp_delay320_fu_1689	delay320	4098	4418	4098	4418	none

Figure 4.9: HLS report performed by Vivado HLS.

Every instance in Figure 4.9, is a single function that has not been inlined (*inline* means to remove the function as a separate entity in the hierarchy. After inlining, the function is dissolved into the calling function and no longer appears as a separate level of hierarchy in the RTL. In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with surrounding operations.) By recalling the GNU Radio symbolism, Figure 4.10 shows in detail how the accelerator architecture is built by Vivado HLS highlighting the function in 4.9.

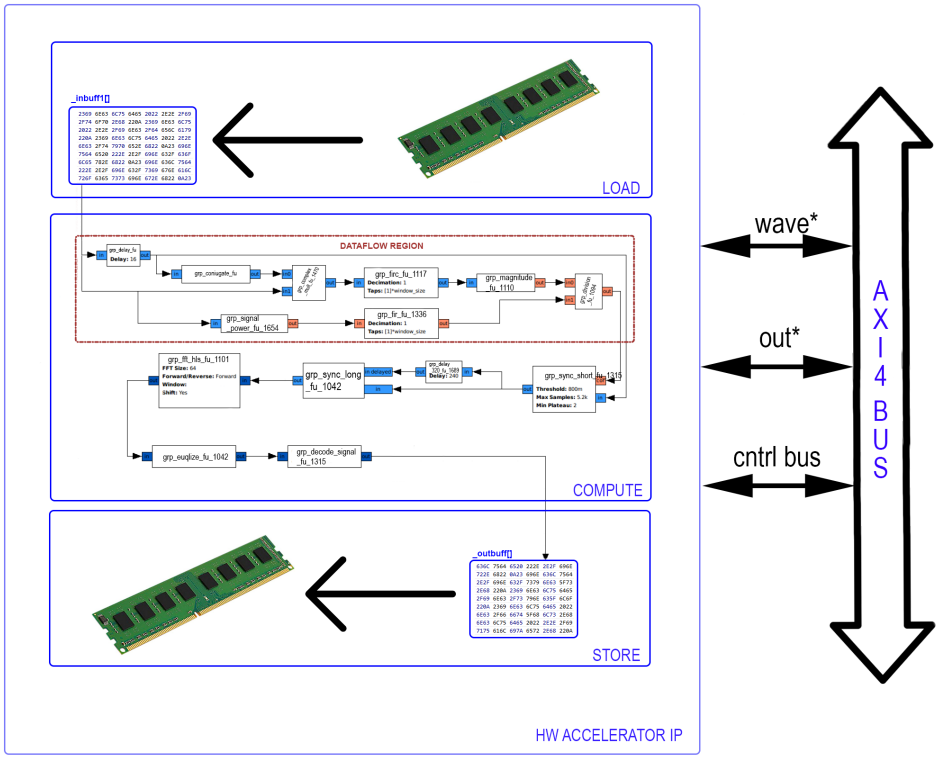
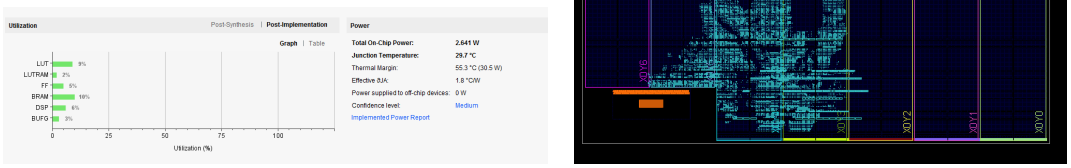


Figure 4.10: Synthesized HW accelerator architecture.

The report of the Logic synthesis is shown in 4.11. All in all, the resource utilization is reasonable small for the performance achieved.



(a) Synthesis resource report

(b) Synthesis power report

Figure 4.11: Synthesis report - Vivado Outputs

Finally the ARM software execution time (discussed in 4.3.2) is compared with the time hardware accelerator takes to execute the same task. This flow is iterated for 3 different design with tree different optimization grade. Again, for this test, only the first part of the algorithm is considered (frame detection) even if the Hardware accelerator implement the overall design (from raw data to decoded bits where modulation scheme is BPSK).

	exe time (ms)		
	ARM	**HLS**	**speed-up**
Loop unrolling+pipelining	*90*	*1.3*	*70x*
loop unrolling	*90*	*6.4*	*14x*
Only Dataflow	*90*	*45.5*	*2x*

Table 4.2: execution time comparison

The fastest hardware implementation overcome the software execution[1] **with a speed-up of about 60x** as shown in Figure 4.12.

[1] debug flag disabled and C optimization -O3

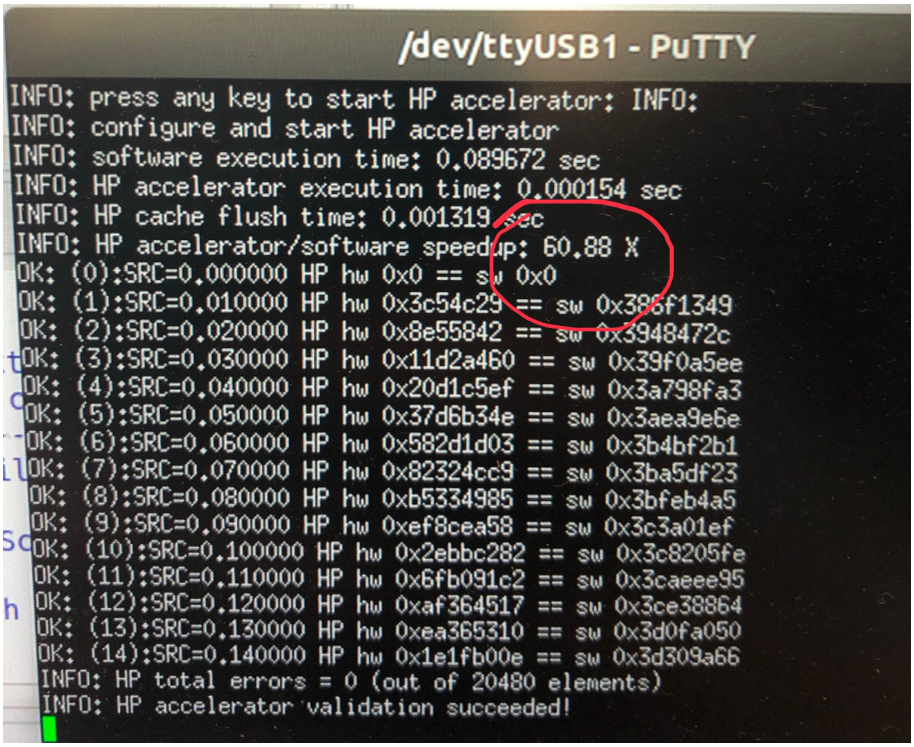


Figure 4.12: Screenshot for the final implementation.

4.6.1 VHDL vs HLS

For the sake of completeness, it was investigated how far (in term of speed-up and area) the HLS implementation is with respect to the HDL one. Considering:

1. For every block in GR it is built the corresponding MATLAB script that it functionally the same.

2. Every MATLAB script is built with simple and basic function. No wireless toolbox add-on are used.

3. Every function of the previous point is compatible with HDL coder.

So, it was used the set of MATLAB scripts to generate the VHDL code using the Matlab tool HDL coder in an automatic way. Comparison are shown in Table 4.3

> HDL Coder generates portable, synthesizable Verilog and VHDL code from MATLAB functions, Simulink models, and Stateflow charts. The generated HDL code can be used for FPGA programming or ASIC prototyping and design.
> HDL Coder provides a workflow advisor that automates the programming of Xilinx, Microsemi, and Intel FPGAs. You can control HDL architecture and implementation, highlight critical paths, and generate hardware resource utilization estimates. HDL Coder provides traceability between your Simulink model and the generated Verilog and VHDL code, enabling code verification for high-integrity applications adhering to DO-254 and other standards.

	ARM	HLS	VHDL
exe time (ms)	90	1.3	0.079

Table 4.3: execution time comparison

In summary, **HLS design compared to the VHDL project generated by HDL coder, was about twice the area, consumed more power [2] and was slower(about 20x)**. Even though the performance of the HLS design was poorer in every way when compared to the VHDL design, it was easy to program and much more flexible in term of design space exploration.

[2]The power refereed to this test do not take into account switching activity and other statistical parameter, so it can be misleading and even a wrong result

Chapter 5

SDR workflow advisor

It is quite a three-pipe problem. -Sir Arthur Conan Doyle

GNURadio comes with a multitude advantages. First of all plenty of communication and signal processing blocks are already implemented; Secondly users can also implement custom blocks themselves as part of out-of-tree library modules. The implementation of such blocks can be C++ or Python based; This mean that it is not necessary to understand or describe the system in a lower level using Hardware Description Language (HDL) essential for in-hardware Field Programmable Gate Array (FPGA) implementations. This approach comes with some performance limits regarding processing latency, but for initial prototyping and validation purposes, the extra delays can be considered marginal.

SDRwa is an attempt to push GNURadio further to overcome its software-execution limits.

Its aim is to formalize a general methodology to deploy every GR algorithm into real hardware (FPGA or even ASIC). A tool called *SDR workflow advisor* or *sdrWA* will be developed to assist the designer step by step with the transition from software to hardware.

Regarding the GR side, another block implementation will be added next to the original one, written in "synthesizable" C/C++ language, and it will be functionally equivalent.
Practically speaking, the engineer/designer or maker could develop/replicate/invent and simulate an SDR or general-purpose algorithm in the GR environment using the original optimize software-execution C implementation of the blocks, taking advantage of its large supportive community. Next, the transaction from software to hardware is assisted by the sdrWA tool that analyzes the flowgraph, synthesizes the accelerator using the synthesizable C implementation and optimizes the architecture given some design constrains[1].
Firstly, optimization is performed focusing on the **block implementation** (Fig.5.1) by choosing the appropriate implementation for every block by which the algorithm is composed. For example, critical parts can be accelerated by pipelining and/or by the unrolling technique, and the remaining parts are balanced by potentially increasing resource sharing while simultaneously decreasing the area.

> "Optimized software-execution implementation" is used in contrast with "synthesizable C implementation" even if they are both referring to the source code of a specific GR block. The first implementation is originally developed by GNURadio team which uses high-performance library for CPU execution e.g. VOLK library, a toolkit that eases the use of Single Instruction Multiple Data (SIMD) instruction in order to dynamically select the fastest implementation on the host system using vectors instead of scalar. Contrarily, the second implementation is hardware optimized in the sense that it is written in a specific way in order to derive the best hardware architecture ones it is synthesized using the the HLS Flow; Even if it can be used for software execution, in general it is slower compared to its common implementation (because loop are not merged by default, presence of redundant operation, etc).

This fine-grain design exploration methodology can be further enhanced by taking advantage of the GR system simulation which can extract useful information such as latency, block throughput , wire congestion, etc. Beginning with this information, sdrWA can choose the best implementation knowing the blocks' role within the overall algorithm.

The next step is the **system level optimization** (fig.5.1). The proposed tool assists the designer in combining the appropriate system implementation by considering aspects such as bit width requirements, approximation of the fixed point data type, bus congestion, etc. , taking advantage again of the simulation behavior modeled by the real execution of the design.

5.0.1 The "new" GR block Implementation

As mentioned above, another implementation will be added for every GR block.Therefore GR has to switch between them considering what the final purpose is; if it is the simulation of the system, the original implementation will be chosen (that is optimized only for software execution on general-purpose processors); If the purpose is the deployment of the algorithm into specialized Hardware,

[1] I am taking into consideration also the testing and validation process.

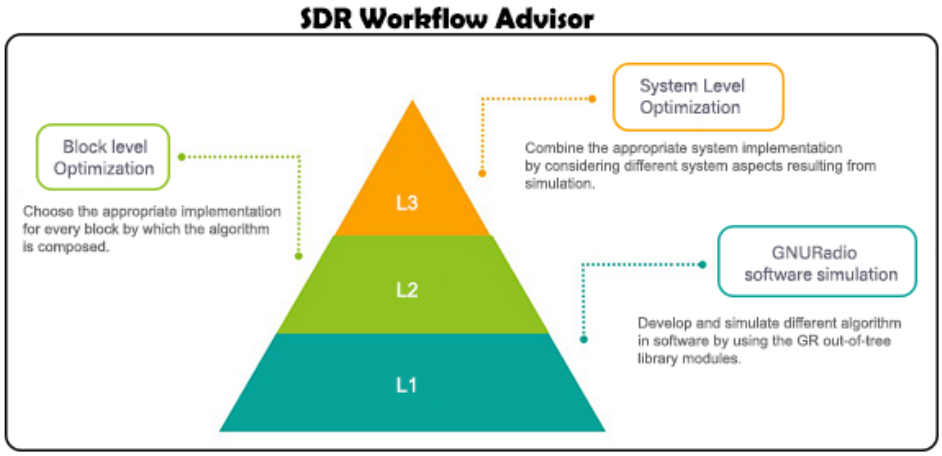


Figure 5.1: sdrWA Optimization Layer

the "new" implementation will be chosen.

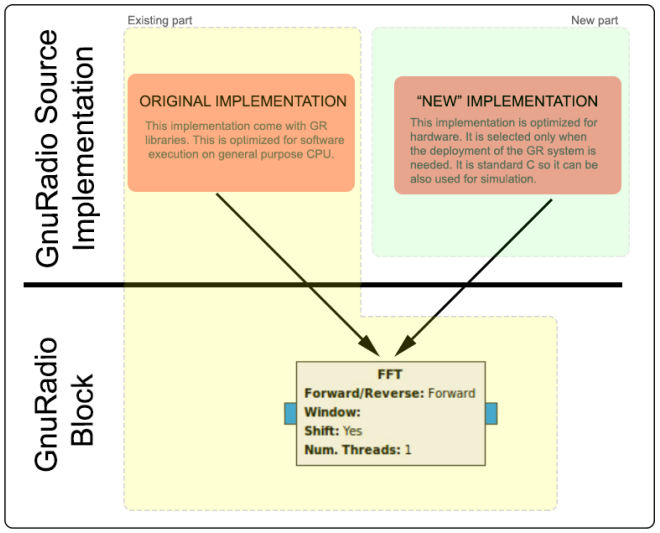


Figure 5.2: GnuRadio layer

The new block implementations could be derived in two different way:

- HDL synthesis
- HLS synthesis

In order to provide designers more flexibility in terms of hardware optimization The second approach is chosen for the project. This means that the design starting point is written in a high level language (C/C++), and the hardware implementation derives from a high level synthesis methodology (HLS). In fact, with the HLS flow it is possible to perform a design space exploration(DSE) and directly choose the solution that better fits the project constraints in terms of latency, area,power, or a combination of the three.
One can therefore choose the best speed implementation (that in general has more area) of the system Pareto curve because he/she can use the entire FPGA for that purpose.
An industry researcher can choose to reduce the area of the implementation to better meet the fitting constraints of the Network-on-Chip (NoC) while still accelerating the algorithm with respect to the software execution.
Otherwise, a company can simply choose the best performance/area solution for a specific task of an ASIC where the area is the main concern.

5.1 Practical Implementation

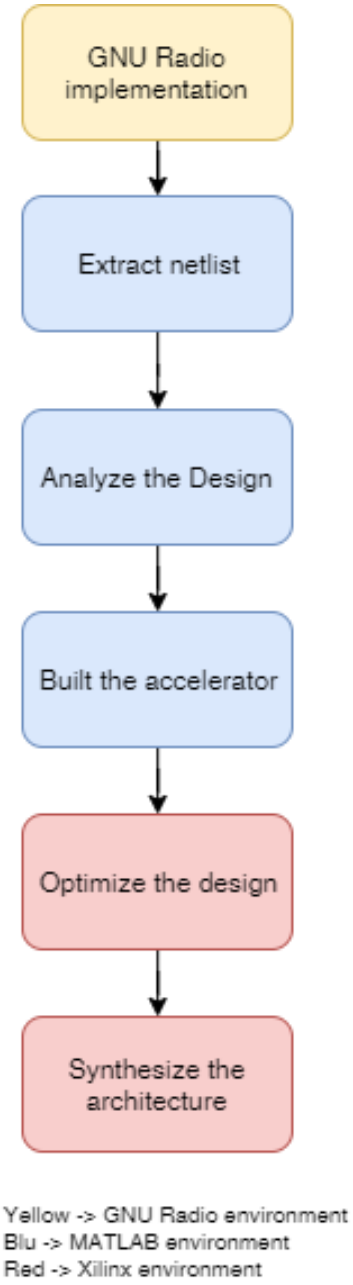


Figure 5.3: SDRwa flow

Consider a generic algorithm implemented in GNU Radio. As mentioned above, SDRwa assist the designer to deploy the algorithm into an FPGA by following the following steps (reported in figure 5.3):

- Starting point is the GNU Radio flowgraph. The designer has built and tested the algorithm using GR; eventually the algorithm it is tested with some SDR platform e.g. Ettus Research USPR.
- the netlist is extracted from GNU Radio and fed into a MATLAB script that decode the different connection and classify the blocks used.
- Another script analyze the design; This is the steps in charge of tracking the right component, the number of that component and building the matrix of connection according to the original flowgraph.

- The next step is the accelerator design where the script puts all the information together:
  - extract some simulation data from GR.
  - analyze the loop of each component
  - generate a set of directive for vivado HLS compiler
- The optimization phase is the interaction between Vivado HLS and MATLAB in order to fit the given constrain.
- Once the optimization phase has completed, a series of MATLAB script are able to pack the accelerator and generate the bit stream using Vivado.

Chapter 6

Conclusion

6.0.1 Concluding Remarks

In this thesis, we first explained the motivation behind this work, i.e., modeling IEEE 802.11p,accelerate the algorithm onto an FPGA in order to extract key features to propose an universal methodology. Afterwards, in various chapters, we explained different building blocks of an IEEE 802.11p physical layer. Where interesting and worthwhile, we mentioned the implementation choices made during the development of the module.
As mentioned before, there is a long way towards having a realistic methodology. This is not only due to complexities involved in the implementation of the current features, but also the practical approach to coordinate different software, build for different application, to interact together.
On the other hand, this work evaluates the feasibility of deploying to FPGA and accelerate a specific wireless network algorithm. This work, in conjunction with all the mentioned framework (Vivado enviroment + Gnuradio + Matlab) can lead to solve open problems in existing and upcoming 802.11 wireless networks.

6.0.2 Summary of the work done

First, a simulation model for the generation and demodulation of IEEE 802.11p OFDM signals was implemented in MATLAB. Second the Hardware Code was written in such a way that it is easier to optimize following the Xilinx HLS flow in order to obtain an IP core for the overall algorithm. Third a software version for the on-chip ARM processor was made and tested. The interaction between GNU Radio and Xilinx environment has been also investigated. Lastly the accelerator is connected through an AXI bus to the ARM processor and executed leading an overall 60x acceleration. Nonetheless other solutions are made available to meet area, power or speed constrains under the form of Design Space Exploration(DSE).

6.0.3 Significant Results

On the frame detection part,utilizing a Hardware accelerator built in 3 It was obtained a reasonable good speed up of 60x with respect to the execution performed by the hard core ARM processor of the Xilinx Zynq architecture.
Furthermore it is possible to generate different architecture in order to met any reasonable constrain.

6.0.4 Lesson Learnt

- Xilinx High level synthesys design flow.
- GNU Radio environment (Graphical user interface and how GR works at lower level by analyzing the source code).
- The way the C code is written makes the difference in RTL generation through the HLS used flow.
- It is **not** possible to convert original GNU Radio code (C code) to HLS code due to a particular workstation-oriented programming style and the heavy usage of external library.
- It is possible to add a new implementation to GNU Radio in C/C++; This new implementation has the ability to execute in an un-optimize way on workstation and perform its best when synthesized with HLS code.
- GNU Radio can export useful simulation report (range of the single signal, bus congestion) that can be crucial for a good optimization via Xilinx HLS flow.

6.0.5 Future investigation

IEEE 802.11p

802.11 p communications are mainly intended for the support of safety-critical and traffic management applications. However, users accustomed to ubiquitous broadband access to the Internet will transfer these expectations to the interior of their vehicles. In the near future, 802.11 p networks

will have to face user demands for high throughput. To this end, schemes that adapt the data rate provide good means to cope with these demands. However, this is a challenging task due to the dispersive characteristics of the vehicular channel. Furthermore, the high mobility of the nodes causes fast changes in the channel conditions and the network topology. In addition, the connectivity time with the access points can be potentially low. Hence, the algorithm for adapting the rate needs to be robust and feature short convergence times.
However, in car-to-infrastructure communications the surroundings of the access point typically feature certain recurring characteristics, such as node density, vehicle speed, and propagation loss, among others. These characteristics result in specific patterns of the signal strength evolution over time.
It is worth to investigate the usage of a **learning algorithm** to identify these signal strength patterns buried in empirical data and to, additionally, combine them with available GPS-based context information, so as to efficiently predict the best performing rate for the expected propagation conditions.

Toward a Hardware based Defined Radio

A more general investigation, starting from this thesis, will be done in order to develop an automatic tool that from the GNU Radio netlist and reports and through a guide procedure brings the designer toward the final Hardware implementation.
A close look will also be taken in the division between software and Hardware pieces and how GNU Radio interact with the operating system in order to lock GNU Radio with the FPGA with a high speed bus.
Porting GNU Radio on ARM processor or enabling it to interact with PCI-e ports can be a reasonable propose. However the methodology has been though toward a standalone solution so the usage of GNU Radio software is limited only in the prototype phase where SDR enable the designer a better signal visibility therefore more capabilities on the debugging process while Hardware enables a quasi or in some case real time speed obtained by a more complex architecture (and therefore more difficult to debug).

Appendix A

Mathematical recall

> *If I were again beginning my studies, I would follow the advice of Plato and start with mathematics.* - Galileo Galilei

A.1 Number rappresentation

Computers and FPGAs typically represent numbers using binary representation, which enables numbers to be efficiently represented using on-off signals called binary digits, or simply bits. Binary numbers work in most ways like normal decimal numbers, but can often be the cause of confusing errors if you are not familiar with how they work. This is particularly true in many embedded systems and FPGAs where minimizing the number of bits used to represent variables can greatly increase the overall performance or efficiency of a system. In this section, we will summarize binary arithmetic and the basic ways that computers represent numbers.
When we write a normal integer, such as 4062, what we really mean is implicitly $(4 \cdot 1000) + (0 \cdot 100) + (6 \cdot 10) + (2 \cdot 1) = 4062$, or written in columns:

10^3	10^2	10^1	10^0	unsigned
4	0	6	2	$= 4062$

A binary number is similar, except instead of using digits from zero to nine and powers of ten, we use numbers from zero to one and powers of 2:

2^3	2^2	2^1	2^0	unsigned
1	0	1	1	$= 11$

since $(1 \cdot 8) + (0 \cdot 4) + (1 \cdot 2) + (1 \cdot 1) = 11$. To avoid ambiguity, binary numbers are often prefixed with "0b". This makes it obvious that 0b1011 is the number decimal 11 and not the number 1011. The bit associated with the highest power of two is the most significant bit , and the bit associated with the lowest power of two is the least significant bit . Hexadecimal numbers use the digits representing numbers from zero to 15 and powers of 16:

16^3	16^2	16^1	16^0	unsigned
8	0	3	15	$= 32831$

In order to avoid ambiguity, the digits from 10 to 15 are represented by the letters "A" through "F", and hexadecimal numbers are prefixed with "0x". So the number above would normally be written in C code as 0x803F.
Note that binary representation can also represent fractional numbers, usually called **fixed-point** numbers, by simply extending the pattern to include negative exponents, so that "0b1011.01" is equivalent to:

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	unsigned
1	0	1	1	0	1	$= 11.25$

since $8 + 2 + 1 + \frac{1}{4} = 11.25$. Unfortunately, the C standard doesn't provide a way of specifying constants in binary representation, although gcc and many other compilers allow integer constants (without a decimal point) to be specified with the "0b" prefix. The C99 standard does provide a way to describe floating-point constants with hexadecimal digits and a decimal exponent, however. Note that the decimal exponent is required, even if it is zero.

```
oat p1 = 0xB.4p0; // Initialize p1 to "11.25"
oat p2 = 0xB4p-4; // Initialize p2 to "11.25"
```

Notice that in general, it is only necessary to write non-zero digits and any digits not shown can be assumed to be zero without changing the represented value of an unsigned number. As a result, it is easy to represent the same value with more digits: simply add as many zero digits as necessary. This process is often called zero-extension. Note that each additional digit increases the amount of numbers that can be represented. Adding an additional bit to a binary number doubles the amount of numbers that can be represented, while an additional hexadecimal digit increases the amount of numbers by a factor of 16.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	unsigned
0	0	0	0	1	0	1	1	0	1	0	0	$= 11.25$

Note that it is possible to have any number of bits in a binary number, not just 8, 16, or 32. SystemC [2], for instance, defines several template classes for handling arbitrary precision integers and fixed-point numbers (including sc_int<>, sc_uint<>, sc_bigint<>, sc_ubigint<>, sc_fixed<>, and sc_ufixed<>). These classes can be commonly used in HLS tools, although they were originally defined for system modeling and not necessarily synthesis. Vivado HLS, for instance, includes similar template classes (ap_int<>, ap_uint<>, ap_fixed<>, and ap_ufixed<>) that typically work better than the SystemC template classes, both in simulation and synthesis.

A.1.1 Negative Number

Negative numbers are slightly more complicated than positive numbers, partly because there are several common ways to do it. One simple way is represent negative numbers with a sign bit, often called signed-magnitude representation. This representation just includes an additional bit to the front of the number to indicate whether it is signed or not. One somewhat odd thing about signed-magnitude representation is that there is more than one way to represent zero. This tends to make even apparently simple operations, like operator ==(), more complex to implement.

+/-	2^1	2^0	signed magnitude
0	1	1	= 3
0	1	0	= 2
0	0	1	= 1
0	0	0	= 0
1	0	0	= -0
1	0	1	= -1
1	1	0	= -2
1	1	1	= -3

Another way to represent negative numbers is with biased representation. This representation adds a constant offset (usually equal in magnitude to the value of the largest bit) to the value, which are otherwise treated as positive numbers:

2^2	2^1	2^0	biased
1	1	1	= 3
1	1	0	= 2
1	0	1	= 1
1	0	0	= 0
0	1	1	= -1
0	1	0	= -2
0	0	1	= -3
0	0	0	= -4

However by far the most common technique for implementing negative numbers is known as two's complement. In two's complement representation, the most significant bit represents the sign of the number (as in signed-magnitude representation), and also whether or not an offset is applied. One way of thinking about this situation is that the high order bit represents a negative contribution to the overall number.

-2^2	2^1	2^0	two's complement
0	1	1	= 3
0	1	0	= 2
0	0	1	= 1
0	0	0	= 0
1	1	1	= -1
1	1	0	= -2
1	0	1	= -3
1	0	0	= -4

-2^4	2^3	2^2	2^1	2^0	two's complement
0	0	0	1	1	= 3
0	0	0	1	0	= 2
0	0	0	0	1	= 1
0	0	0	0	0	= 0
1	1	1	1	1	= -1
1	1	1	1	0	= -2
1	1	1	0	1	= -3
1	1	1	0	0	= -4

One significant difference between unsigned numbers and two's complement numbers is that we need to know exactly how many bits are used to represent the number, since the most significant bit is treated differently than the remaining bits. Furthermore, when widening a signed two's complement number with more bits, the sign bit is replicated to all the new most significant bits. This process is normally called sign-extension. For the rest of the book, we will generally assume that all signed numbers are represented in two's complement unless otherwise mentioned.

A.1.2 Overflow, Underflow, and Rounding

Overfow occurs when a number is larger than the largest number that can be represented in a given number of bits. Similarly, underflow occurs when a number is smaller than the smallest number that can be represented. One common way of handling overflow or underflow is to simply drop the most significant bits of the original number, often called wrapping.

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
0	0	1	0	1	1	0	1	0	0	= 11.25
	0	1	0	1	1	0	1	0	0	= 11.25
		1	0	1	1	0	1	0	0	= 11.25
			0	1	1	0	1	0	0	= 3.25

Handling overflow and underflow by wrapping two's complement numbers can even cause a positive number to become negative, or a negative number to become positive.

-2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	two's complement
1	0	1	1	0	1	0	0	= 4.75

-2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	two's complement
0	1	1	0	1	0	0	= 3.25

Similarly, when a number cannot be represented precisely in a given number of fractional bits, it is necessary to apply rounding. Again, there are several common ways to round numbers. The simplest way is to just drop the extra fractional bits, which tends to result in numbers that are more negative. This method of rounding is often called rounding down or rounding to negative infinity. When rounding down to the nearest integer, this corresponds to the *foor()* function, although it's possible to round to other bit positions as well.

0b0100.00	= 4.0			0b0100.0	= 4.0
0b0011.11	= 3.75			0b0011.1	= 3.5
0b0011.10	= 3.5			0b0011.1	= 3.5
0b0011.01	= 3.25			0b0011.0	= 3.0
0b0011.00	= 3.0	Round to		0b0011.0	= 3.0
0b1100.00	= −4.0	→ Negative	→	0b1100.0	= −4.0
0b1011.11	= −4.25	Infinity		0b1011.1	= −4.5
0b1011.10	= −4.5			0b1011.1	= −4.5
0b1011.01	= −4.75			0b1011.0	= −5.0
0b1011.00	= −5.0			0b1011.0	= −5.0

It is also possible to handle rounding in other similar ways which force rounding to a more positive numbers (called rounding up or rounding to positive infinity and corresponding to the ceil() function), to smaller absolute values (called rounding to zero and corresponding to the trunc() function), or to larger absolute values (called rounding away from zero or rounding to infinity and corresponding to the round() function). None of these operations always minimizes the error caused by rounding, however.
A better approach is called rounding to nearest even, convergent rounding, or banker's rounding and is implemented in the lrint() function. As you might expect, this approach to rounding always picks the nearest representable number. In addition, If there are two numbers equally distant, then the even one is always picked. An arbitrary-precision number is even if the last digit is zero. This approach is the default handling of rounding with IEEE floating point, as it not only minimizes rounding errors but also ensures that the rounding error tends to cancel out when computing sums of random numbers.

0b0100.00	$= 4.0$		0b0100.0	$= 4.0$
0b0011.11	$= 3.75$		0b0100.0	$= 4.0$
0b0011.10	$= 3.5$		0b0011.1	$= 3.5$
0b0011.01	$= 3.25$		0b0011.0	$= 3.0$
0b0011.00	$= 3.0$	Round to	0b0011.0	$= 3.0$
0b1100.00	$= -4.0$	$\rightarrow$ Nearest $\rightarrow$	0b1100.0	$= -4.0$
0b1011.11	$= -4.25$	Even	0b1100.0	$= -4.0$
0b1011.10	$= -4.5$		0b1011.1	$= -4.5$
0b1011.01	$= -4.75$		0b1011.0	$= -5.0$
0b1011.00	$= -5.0$		0b1011.0	$= -5.0$

A.1.3 Binary arithmetic

Binary addition is very similar to decimal addition, simply align the binary points and add digits, taking care to correctly handle bits carried from one column to the next. Note that the result of adding or subtracting two N-bit numbers generally takes N+1 bits to represent correctly without overflow. The added bit is always an additional most significant bit for fractional numbers

	2^3	2^2	2^1	2^0	unsigned
		0	1	1	3
+		0	1	1	3
	0	1	1	0	6

	2^3	2^2	2^1	2^0	2^-1	unsigned
		1	1	1	1	7.5
+		0	1	1	1	7.5
	1	1	1	1	0	15

Note that since the result of subtraction can be negative, the 'extra bit' becomes the sign-bit of a two's complement number.

	2^3	2^2	2^1	2^0	unsigned
		0	1	1	3
-		0	1	1	3
	0	0	0	0	0

2^-4	2^3	2^2	2^1	2^0	unsigned
	0	0	1	1	3
-	1	1	1	1	15
1	0	1	0	0	-12 (2's complement)

Multiplication for binary numbers also works similarly to familiar decimal multiplication. In general, multiplying 2 N-bit numbers results in a 2*N bit result.

	2^6	2^5	2^4	2^3	2^2	2^1	2^0	two's complement
				1	0	0	1	= 9
*				1	0	0	1	= 9
				1	0	0	1	= 9
			0	0	0	0		= 0
		0	0	0	0			= 0
+	1	0	0	1				= 72
	1	0	1	0	0	0	1	= 81

Operations on signed numbers are somewhat more complex because of the sign-bit handling and won't be covered in detail. However, the observations regarding the width of the result still applies: adding or subtracting two N-bit signed numbers results in an N+1-bit result, and Multiplying two N-bit signed numbers results in an 2*N-bit result.

A.1.4 Representing Arbitrary Precision Integers in C and C++

According to the C99 language standard, the precision of many standard types, such as int and long are implementation defined. Although many programs can be written with these types in a way that does not have implementation-defined behavior, many cannot. One small improvement is the inttypes.h header in C99, which defines the types int8 t,int16 t,int32 t, and int64 t representing signed numbers of a given width and the corresponding types uint8_t,uint16_t,uint32_t, and uint64_t representing unsigned numbers. Although these types are defined to have exactly the given bit-widths, they can still be somewhat awkward to use. For instance, even relatively simple programs like the code below can have unexpected behavior.

```
#include "inttypes.h"
uint16_t a =0x4000;
uint16_t b = 0x4000;
// Danger! p depends on sizeof(int)
uint32_t p = a*b;
```

Although the values of a and b can be represented in 16 bits and their product (0x10000000) can be represented exactly in 32 bits, the behavior of this code by the conversion rules in C99 is to first convert a and b to type int, compute an integer result, and then to extend the result to 32 bits. Although uncommon, it is correct for a C99 compiler to only have integers with only 16 bits of precision. Furthermore, the C99 standard only defines 4 bit-widths for integer numbers, while FPGA systems often use a wide variety of bit-widths for arithmetic. Also, printing these data-types using printf() is awkward, requiring the use of additional macros to write portable code. The situation is even worse if we consider a fixed-point arithmetic example. In the code below, we consider a and b to be fixed point numbers, and perform normalization correctly to generate a result in the same format.

```
#include "inttypes.h"
// 4.0 represented with 12 fractional bits
uint16_t a =0x4000;
// 4.0 represented with 12 fractional bits.
uint16_t b = 0x4000;
// Danger! p depends on sizeof(int)
uint32_t p = (a*b) >> 12;
```

The correct code in both cases requires casting the input variables to the width of the result before multiplying.

```
#include "inttypes.h"
uint16 t a = 0x4000;
uint16 t b = 0x4000;
// p is assigned to 0x10000000
uint32_t p = (uint32 t) a*(uint32 t) b;
```

```
#include "inttypes.h"
// 4.0 represented with 12 fractional bits.
uint16_t a =0x4000;
// 4.0 represented with 12 fractional bits.
uint16_t b = 0x4000;
// p assigned to 16.0 represented with 12 fractional bits
uint32_t p = ( (uint32 t) a*(uint32 t) b ) >> 12;
```

When using integers to represent fixed-point numbers, it is very important to document the fixed point format used, so that normalization can be performed correctly after multiplication. Usually this is described using "Q" formats that give the number of fractional bits. For instance, "Q15" format uses 15 fractional bits and usually applies to 16 bit signed variables. Such a variable has values in the interval [-1; 1). Similarly "Q31" format uses 31 fractional bits.

For these reasons, it's usually preferable to use C++ and the Vivado HLS template classes ap_int<>, ap_uint<>, ap_fixed<>, and ap_ufixed<> to represent arbitrary precision numbers. The ap_int<> and ap_uint<> template classes require a single integer template parameter that defines their width. Arithmetic functions generally produce a result that is wide enough to contain a correct result, following the rules in section A.1.3. Only if the result is assigned to a narrower bit-width does overflow or underflow occur.

```
#include "ap_int.h"
ap_uint<15> a =0x4000;
ap_uint<15> b = 0x4000;
// p is assigned to 0x10000000.
ap_uint<30> p = a*b;
```

The ap_fixed<> and ap_ufixed<> template classes are similar, except that they require two integer template arguments that define the overall width (the total number of bits) and the number of integer bits.

```
#include "ap_fixed.h"
// 4.0 represented with 12 fractional bits.
ap u_fixed<15,12> a = 4.0;
// 4.0 represented with 12 fractional bits.
ap_ufixed<15,12> b = 4.0;
// p is assigned to 16.0 represented with 12 fractional bits
ap_ufixed<18,12> p = ab;
```

A.1.5 Floating Point

Vivado HLS can also synthesize floating point calculations. Floating point numbers provide a large amount of precision, but this comes at a cost; it requires significant amount of computation which in turn translates to a large amount of resource usage and many cycles of latency. Thus, floating point numbers should be avoided unless absolutely necessary as dictated by the accuracy requirements application. In fact, the primary goal of this chapter is to allow the reader to understand how to effectively move from floating point to fixed point representations. Unfortunately, this is often a non-trivial task and there are not many good standard methods to automatically perform this translation. This is partially due to the fact that moving to fixed point will reduce the accuracy of the application and this trade-off is best left to the designer.
The standard technique for high-level synthesis starts with a floating point representation during the initial development of the application. This allows the designer to focus on getting a functionally correct implementation. Once that is achieved, then she can move optimizing the number representation in order to reduce the resource usage and/or increase the performance.

A.2 Fast Fourier Transform

The FFT brings about a reduction in complexity by taking advantage of symmetries in the DFT calculation. To better understand how to do this, let us look at DFT with a small number of points, starting with the 2 point DFT. Recall that the DFT performs a matrix vector multiplication, i.e., G[] = S[][] Â· g[], where g[] is the input data, G[] is the frequency domain output data, and S[][] are the DFT coefficients. For a 2 point DFT, the values of S are:

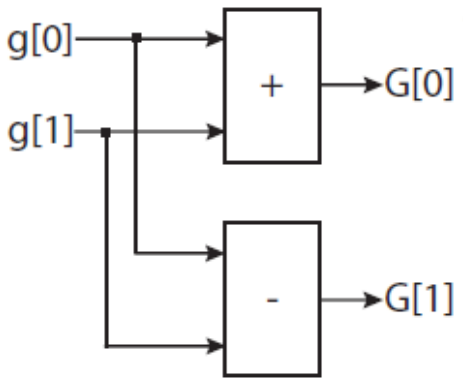
$$S = \begin{pmatrix} W_2^{00} & W_2^{01} \\ W_2^{10} & W_2^{11} \end{pmatrix}$$

Here we use the notation $W = e^{\text{[U+0088] [U+0092]} j2\pi}$. The superscript on W denotes values that are added to the numerator and the subscript on the W indicates those values added in the denominator of the complex exponential. For example,$W_4^{23} = e^{\frac{-2j\pi \cdot 2 \cdot 3}{4}}$.
Writing the equation in matrix form:

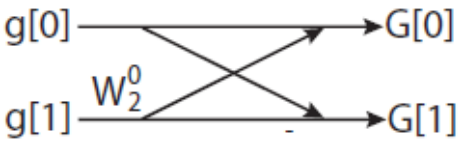
$$\begin{bmatrix} G[0] \\ G[1] \end{bmatrix} = \begin{bmatrix} W_2^{00} & W_2^{01} \\ W_2^{10} & W_2^{11} \end{bmatrix} \cdot \begin{bmatrix} g[0] \\ g[1] \end{bmatrix}$$

Expanding the two equation for a 2 point DFT gives:

$$\begin{cases} G[0] = g[0] \cdot e^{\frac{-2j\pi \cdot 0 \cdot 0}{2}} + g[1] \cdot e^{\frac{-2j\pi \cdot 0 \cdot 1}{2}} = g[0] + g[1] \\ G[1] = g[0] \cdot e^{\frac{-2j\pi \cdot 1 \cdot 0}{2}} + g[1] \cdot e^{\frac{-2j\pi \cdot 1 \cdot 1}{2}} = g[0] - g[1] \end{cases}$$



(a) data ow graph for a 2 point DFT/FFT.



(b) data flow graph for a 2 point DFT/FFT viewed as a butterfly structure.

Figure A.1: DFT/FFT fttb

Figure A.1 provides two different representations for this computation. Part A.1a is the data flow graph for the 2 point DFT. It is the familiar view that we have used to represent computation throughout this book. Part A.1b shows a butterfly structure for the same computation. This is a typically structure used in digital signal processing, in particular, to represent the computations in an FFT.
The butterfly structure is a more compact representation that is useful to represent large data flow graphs. When two lines come together this indicates an addition operation. Any label on the line itself indicates a multiplication of that label by the value on that line.
The butterfly structure is a more compact representation that is useful to represent large data flow graphs. When two lines come together this indicates an addition operation. Any label on the line itself indicates a multiplication of that label by the value on that line. There are two labels in this figure. The sign − on the bottom horizontal line indicates that this value should be negated.
This followed by the addition denoted by the two lines intersecting is the same as subtraction. The second label is W_2^0 . While this is a multiplication is unnecessary (since $W_2^0 = 1$ this means it is multiplying by the value 1), we show it here since it is a common structure that appears in higher

point FFTs.

Now let us consider a slightly larger DFT - a 4 point DFT, i.e., one that has 4 inputs, 4 outputs, and a $4x4$ S matrix. The values of S for a 4 point DFT are:

$$S = \begin{pmatrix} W_4^{00} & W_4^{01} & W_4^{02} & W_4^{03} \\ W_4^{10} & W_4^{11} & W_4^{12} & W_4^{13} \\ W_4^{20} & W_4^{21} & W_4^{22} & W_4^{23} \\ W_4^{30} & W_4^{31} & W_4^{32} & W_4^{33} \end{pmatrix}$$

And the DFT equation to compute the frequency output terms are:

$$\begin{bmatrix} G[0] \\ G[1] \\ G[2] \\ G[3] \end{bmatrix} = \begin{pmatrix} W_4^{00} & W_4^{01} & W_4^{02} & W_4^{03} \\ W_4^{10} & W_4^{11} & W_4^{12} & W_4^{13} \\ W_4^{20} & W_4^{21} & W_4^{22} & W_4^{23} \\ W_4^{30} & W_4^{31} & W_4^{32} & W_4^{33} \end{pmatrix} \cdot \begin{bmatrix} g[0] \\ g[1] \\ g[2] \\ g[3] \end{bmatrix}$$

By writing out the equations for each of the frequency domain values in G[] one-by-one, reduce them (it is found out that $e^{\frac{-j18\pi}{4}}$ is reduced to $e^{\frac{-j10\pi}{4}}$ since these are equivalent based upon a 2π rotation) and finally reordering the result is:

$$G[0] = (g[0] + g[2]) + e^{\frac{-j2\pi 0}{4}}(g[1] + g[3])$$

$$G[1] = (g[0] - g[2]) + e^{\frac{-j2\pi 1}{4}}(g[1] - g[3])$$

$$G[2] = (g[0] + g[2]) + e^{\frac{-j2\pi 2}{4}}(g[1] + g[3])$$

$$G[3] = (g[0] - g[2]) + e^{\frac{-j2\pi 3}{4}}(g[1] - g[3])$$

Several different symmetries are starting to emerge. First, the input data can be partitioned into even and odd elements, i.e., similar operations are done on the elements $g[0]$ and $g[2]$, and the same is true for the odd elements $g[1]$ and $g[3]$. Furthermore we can see that there are addition and subtraction symmetries on these even and odd elements. During the calculations of the output frequencies $G[0]$ and $G[2]$, the even and odd elements are summed together. The even and odd input elements are subtracted when calculating the frequencies $G[1]$ and $G[3]$. Finally, the odd elements in every frequency term are multiplied by a constant complex exponential W_4^i where i denotes the index for the frequency output, i.e., $G[i]$.

Looking at the terms in the parentheses, we see that they are 2 point FFT. For example, consider the terms corresponding to the even input values $g[0]$ and $g[2]$. If we perform a 2 point FFT on these even terms, the lower frequency (DC value) is $g[0] + g[2]$, and the higher frequency is calculated as $g[0] - [2]($ The same is true for the odd input values g[1] and g[3]).
We perform one more transformation on these equations.

$$G[0] = (g[0] + g[2]) + e^{\frac{-j2\pi 0}{4}}(g[1] + g[3])$$

$$G[1] = (g[0] - g[2]) + e^{\frac{-j2\pi 1}{4}}(g[1] - g[3])$$

$$G[2] = (g[0] + g[2]) + e^{\frac{-j2\pi 0}{4}}(g[1] + g[3])$$

$$G[3] = (g[0] - g[2]) + e^{\frac{-j2\pi 1}{4}}(g[1] - g[3])$$

Figure A.2 shows the butter y diagram for the four point FFT. We can see that the first stage is two 2 point FFT operations performed on the even (top butterfly) and odd (bottom butterfly) input values. The output of the odd 2 point FFTs are multiplied by the appropriate twiddle factor.

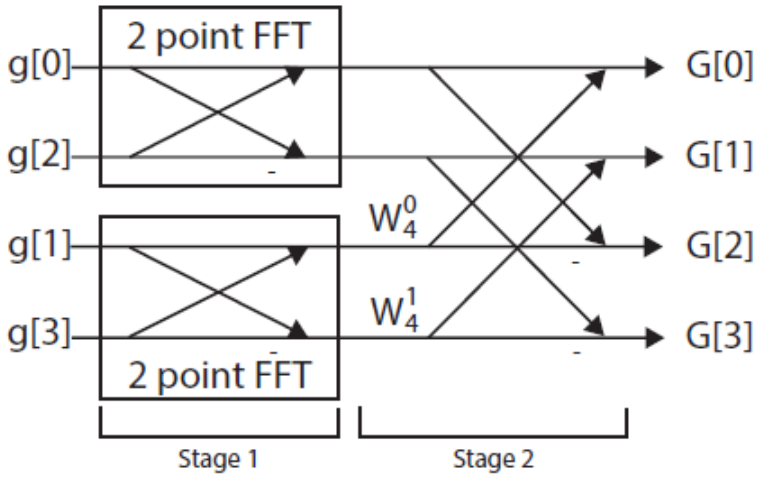


Figure A.2: A four point FFT divided into two stages.

We are seeing the beginning of trend that allows the a reduction in complexity from $O(n2)$ operations for the DFT to $O(nlogn$ operations for the FFT. The key idea is building the computation through recursion. The 4 point FFT uses two 2 point FFTs. This extends to larger FFT sizes. For example, an 8 point FFT uses two 4 point FFTs, which in turn each use two 2 point FFTs (for a total of four 2 point FFTs). An 16 point FFT uses two 8 point FFTs, and so on.

Now let us formally derive the relationship, which provides a general way to describe the recursive structure of the FFT. Assume that we are calculating an N point FFT. The formula for calculating the frequency domain values G[] given the input values g[] is:

$$G[k] = \sum_{n=0}^{N-1} g[n] \cdot e^{\frac{-2j\pi kn}{N}} \quad \text{for } k = 0, ..., N$$

We can divide this equation into two parts, one that sums the even components and one that sums the odd components.

$$G[k] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{\frac{-2j\pi kn}{N/2}} + e^{\frac{-2j\pi k}{N}} \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{\frac{-2j\pi kn}{N/2}}$$

Finally we can simplify this to:

$$G[k] = A_k + W_N^k B_k$$

where A_k and B_k are the first and second summations, respectively. And recall that $W = e^{-2j\pi}$ This completely describes an N point FFT by separating even and odd terms into two summations.

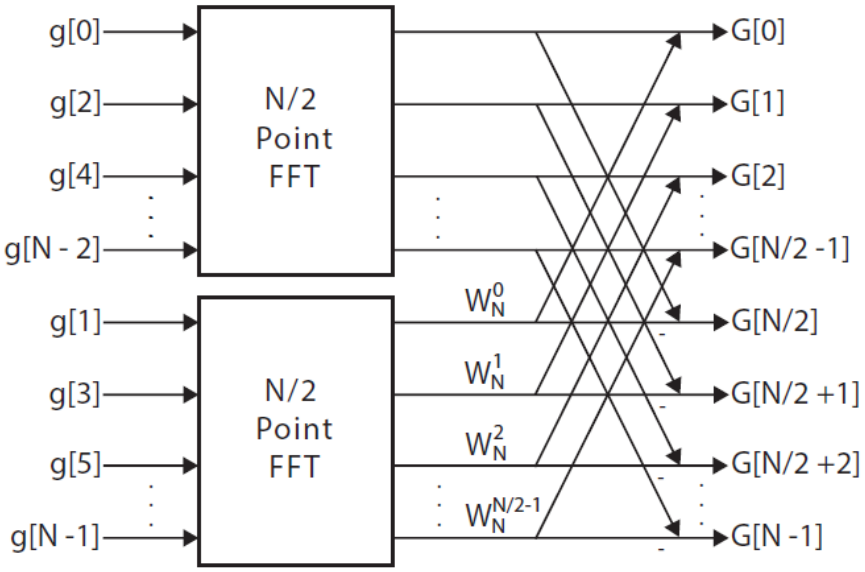


Figure A.3: Building an N point FFT from two N=2 point FFTs. The upper N=2 point FFT is performed on the even inputs; the lower N=2 FFT uses the odd inputs

Figure A.3 shows an N point FFT derived from two $N/2$ point FFTs. A_k corresponds to the top $N/2$ FFT, and B_k is the bottom $N/2$ FFT. The output terms $G[0]$ through $G[N/2-1]$ are multiplied by W_N^0 while the output terms $G[N/2]$ through $G[N-1]$ are multiplied by $-W_N^0$. Note that the inputs $g[]$ are divided into even and odd elements feeding into the top and bottom $n/2$ point FFTs, respectively.
We can use the general formula for creating the FFT that was just derived to recursively create the $N/2$ point FFT. That is, each of the $N/2$ point FFTs can be implemented using two $N/4$ point FFTs. And each $N/4$ point FFT uses two $N/8$ point FFTs, and so on until we reach the base case, a 2 point FFT.

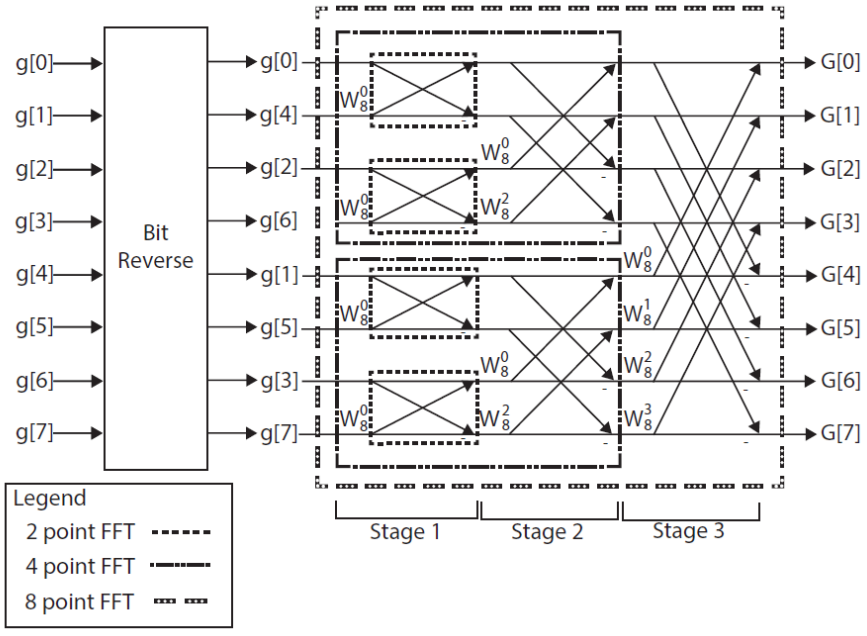


Figure A.4: 8 point FFT built recursively.

Figure A.4 shows an 8 point FFT and highlights this recursive structure. The boxes with the dotted lines indicate different sizes of FFT. The outermost box indicates an 8 point FFT. This is composed by two 4 point FFTs. Each of these 4 point FFTs have two 2 point FFTs for a total of four 2 point FFTs.
Also note that the inputs must be reordered before they are feed into the 8 point FFT. This is due to the fact that the different $N/2$ point FFTs take even and odd inputs. The upper four inputs correspond to even inputs and the lower four inputs have odd indices. However, they are reordered twice. If we separate the even and odd inputs once we have the even set $g[0]; g[2]; g[4]; g[6]$ and the odd set $g[1]; g[3]; g[5]; g[7]$. Now let us reorder the even set once again. In the even set $g[0]$ and $g[4]$ are the even elements, and $g[2]$ and $g[6]$ are the odd elements. Thus reordering it results in the set $g[0]; g[4]; g[2]; g[6]$. The same can be done for the initial odd set yielding the reordered set $g[1]; g[5]; g[3]; g[7]$.
The final reordering is done by swapping values whose indices are in bit reversed order. Table A.1 shows the indices and their three bit binary values. The table shows the eight indices for the 8 point FFT, and the corresponding binary value for each of those indices in the second column.The third column is the bit reversed binary value of the second column. And the last column is the decimal number corresponding the reversed binary number.

index	**Binary**	**Reversed Binary**	**Reversed Index**
1	*000*	*000*	***0***
1	*001*	*100*	***4***
2	*010*	*010*	***2***
3	*011*	*110*	***6***
4	*100*	*001*	***1***
5	*101*	*101*	***5***
6	*110*	*011*	***3***
7	*111*	*111*	***7***

Table A.1: Reverse Bit

Looking at the first row, the initial index 0, has a binary value of 000, which when reversed remains 000. Thus this index does not need to be swapped. Looking at Figure A.4 we see that this is true. $g[0]$ remains in the same location. In the second row, the index 1 has a binary value 001. When reversed this is 100 or 4. Thus, the data that initially started at index 1, i.e., $g[1]$ should end up in the fourth location. And looking at index 4, we see the bit reversed value is 1. Thus $g[1]$ and $g[4]$ are swapped.
This bit reversal process works regardless of the input size of the FFT, assuming that the FFT is a power of two. FFT are commonly a power of two since this allows them to be recursively implemented.

A.3 Finite Impulse Response (FIR) filter

The output signal of a filter given an impulse input signal is its impulse response. The impulse response of a linear, time invariant filter contains the complete information about the filter. As the name implies, the impulse response of an FIR filter (a restricted type of linear, time invariant filter) is finite, i.e., it is always zero far away from zero. Given the impulse response of an FIR filter, we can compute the output signal for any input signal through the process of convolution. This process combines samples of the impulse response (also called coefficients or taps) with samples of the input signal to compute samples of the output signal. The output of filter can be computed in other ways (for instance, in the frequency domain), but for the purposes of this chapter we will focus on computing in the time domain. The convolution of an N-tap FIR filter with coefficients h[] with an input signal x[] is described by the general difference equation:

$$y[i] = \sum_{j=0}^{N-1} h[j] \cdot x[i-j]$$

To compute a single value of the output of an N-tap filter requires N multiplies and N-1 additions.

Note that in general, filter coefficients can be crafted to create many different kinds of filters: low pass, high pass, band pass, etc.. In general, a larger value of number of taps provides more degrees of freedom when designing a filter, generally resulting in filters with better characteristics. There is substantial amount of literature devoted to generating filter coefficients with particular characteristics for a given application. When implementing a filter, the actual values of these coefficients are largely irrelevant and we can ignore how the coefficients themselves were arrived at. However, as we will see below with the moving average filter, the structure of the filter, or the particular coefficients can have a large impact on the number of operations that need to be performed. For instance, symmetric filters have multiple taps with exactly the same value which can be grouped to reduce the number of multiplications. In other cases, it is possible to convert the multiplication by a known constant filter coefficient into shift and add operations [34]. In that case, the values of the coefficients can drastically change the performance and area of the filter implementation [52].

Moving Average Filter

Moving average filters are a simple form of lowpass FIR filter where all the coefficients are identical and sum to one. For instance in the case of the three point moving filter, the coefficientsare $h = [\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$. It is also called a box car filter due to the shape of its convolution kernel.
Alternatively, you can think of a moving average filter as taking the average of several adjacent samples of the input signal and averaging them together. We can see that this equivalence by substituting 1=N for h[j] in the convolution equation above and rearranging to arrive at the familiar equation for an average of N elements:

$$y[i] = \frac{1}{N} \sum_{j=0}^{N-1} x[i-j]$$

Each sample in the output signal can be computer by the above equation using N-1 additions and one final multiplication by 1/N. Even the final multiplication can often be regrouped and merged with other operations. As a result, moving average filters are simpler to compute than a general FIR filter.

This filter is causal, meaning that the output is a function of no future values of the input. It is possible and common to change this, for example, so that the average is centered on the current sample, [1]. While fundamentally causality is an important property for system analysis, it is less important for a hardware implementation as a finite non-causal filter can be made causal with buffering and/or re-indexing of the data.
Moving average filters can be used to smooth out a signal, for example to remove random (mostly high frequency) noise. As the number of taps N gets larger, we average over a larger number of

[1] i.e., $y[12] = \frac{1}{13}(x[11] + x[12] + x[13])$

samples, and we correspondingly must perform more computations. For a moving average filter, larger values of N correspond to reducing the bandwidth of the output signal. In essence, it is acting like a low pass filter (though not a very optimal one). Intuitively, this should make sense. As we average over larger and larger number of samples, we are eliminating higher frequency variations in the input signal. That is, 'smoothing' is equivalent to reducing higher frequencies. The moving average filter is optimal for reducing white noise while keeping the sharpest step response, i.e., it creates the lowest noise for a given edge sharpness.

Appendix B

High-Level Synthesis (HLS)

> *With a high-level language like SystemC, engineers can specify an accelerator while abstracting away all low-level logic and circuit details to focus instead on the relationships between the data structures and computational tasks that characterize the given algorithms. The benefits are higher productivity, less chances of errors, and more options for performance and power optimizations.* - Luca Carloni

The traditional way of programming FPGAs has been HDLs, mostly leaving the task to hardware engineers.
With the development of high level synthesis tools from C and C++, as well as the increasing effort to support OpenCL from both Altera and Xilinx, the world of FPGAs is becoming more accessible to users with software backgrounds. Not unlike programming GPUs, however, the road from writing a functionally correct program to writing a performant program is long, and the tools even less mature. This section will take a closer look at the Vivado HLS tool used to produce all kernels presented throughout this work, highlighting important programming techniques and the hardware they produce. The goal is to provide some general insight in the mapping between the high level C++ program that is fed to the tool and the resulting FPGA hardware.

B.1 Xilinx Vivado HLS Design Flow

The Xilinx Vivado HLS tool synthesizes a C function into an IP block that you can integrate into a hardware system. It is tightly integrated with the rest of the Xilinx design tools and provides comprehensive language support and features for creating the optimal implementation for your C algorithm. Following is the Vivado HLS design flow:

1. Compile, execute (simulate), and debug the C algorithm.

   > In high-level synthesis, running the compiled C program is referred to as C simulation. Executing the C algorithm simulates the function to validate that the algorithm is functionally correct.

2. Synthesize the C algorithm into an RTL implementation, optionally using user optimization directives.
3. Generate comprehensive reports and analyze the design.
4. Verify the RTL implementation using a pushbutton flow.
5. Package the RTL implementation into a selection of IP formats.

Graphically, the Xilinx HLS flow could be summarized as in Figure B.1.

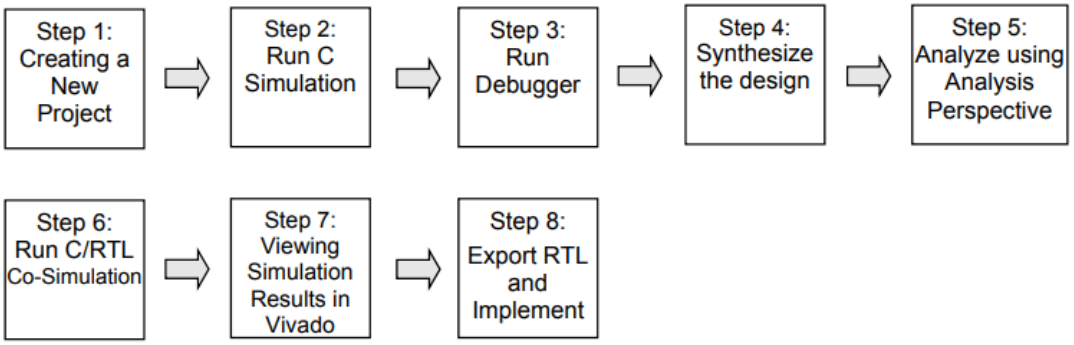


Figure B.1: Xilinx HLS design Flow

B.1.1 Test Bench

When using the Vivado HLS design flow, it is time consuming to synthesize a functionally incorrect C function and then analyze the implementation details to determine why the function does not perform as expected. To improve productivity, use a test bench to validate that the C function is functionally correct prior to synthesis.
The C test bench includes the function main() and any sub-functions that are not in the hierarchy under the top-level function for synthesis. These functions verify that the top-level function for

synthesis is functionally correct by providing stimuli to the function for synthesis and by consuming its output.
Vivado HLS uses the test bench to compile and execute the C simulation.

B.1.2 Synthesis, Optimization, and Analysis

Vivado HLS is project based. Each project holds one set of C code and can contain multiple solutions. Each solution can have different constraints and optimization directives. You can analyze and compare the results from each solution in the Vivado HLS GUI.

Following are the synthesis, optimization, and analysis steps in the Vivado HLS design process:

1. Create a project with an initial solution.
2. Verify the C simulation executes without error.
3. Run synthesis to obtain a set of results.
4. Analyze the results.

After analyzing the results, you can create a new solution for the project with different constraints and optimization directives and synthesize the new solution. You can repeat this process until the design has the desired performance characteristics. Using multiple solutions allows you to proceed with development while still retaining the previous results.

B.1.3 Optimization

Optimization teqniques are discussed in section B.2

B.1.4 Analysis

When synthesis completes, Vivado HLS automatically creates synthesis reports to help you understand the performance of the implementation. In the Vivado HLS GUI, the Analysis Perspective includes the Performance tab, which allows you to interactively analyze the results in detail. The following figure shows the Performance tab for the fft algorithm developed in Listing 3.21 .

Figure B.2: Vivado HLS Analysis Example

B.1.5 RTL Verification

If you added a C test bench to the project, you can use it to verify that the RTL is functionally identical to the original C. The C test bench verifies the output from the top-level function for synthesis and returns zero to the top-level function main() if the RTL is functionally identical. Vivado HLS uses this return value for both C simulation and C/RTL co-simulation to determine if the results are correct. If the C test bench returns a non-zero value, Vivado HLS reports that the simulation failed.

B.2 Optimization teqniques

When describing synchronous circuits, one often will use the number of clock cycles as a measure of performance. However, this is not appropriate when comparing designs that have different clock rates, which is typically the case in HLS. For example, the clock frequency is specified as an input constraint to the Vivado HLS, and it is feasible to generate different architectures for the same exact code by simply changing the target clock frequency. Thus, it is most appropriate to use seconds, which allows an apples-to-apples comparison between any HLS architecture. The Vivado HLS tool reports the number of cycles and the clock frequency. These can be used to calculate the exact amount of time that some piece of code requires to compute. We use the term task to mean a fundamental atomic unit of behavior; this corresponds to a function invocation in Vivado HLS. The task latency is the time between when a task starts and when it finishes. The task interval is the time between when one task starts and the next starts or the difference between the start times of two consecutive tasks.

B.2.1 Pipelining

At the heart of the spatial programming paradigm lies pipelining, and building performant HLS programs revolves around expressing the target algorithm in a way that achieves a perfect pipeline (with an initiation interval of 1).

In addition to maximizing the number of concurrent computations done on the chip, one other critical condition must be fulfilled to achieve performant FPGA programs: each computational element must work at its highest possible throughput. This is achieved by pipelining, and any high performance FPGA implementation must implement perfect pipelining in the computational elements to ensure that resources are kept busy at all times by feeding them new input every cycle. This section will first cover the two principal properties of pipelining, latency and initiation interval, then go through the levels of pipelining that must be treated in a pipelined program.
Latency and initiation interval A pipelined program will have a number of stages that data will flow through before a result is produced. *Latency* is the number of cycles between an element entering the first stage of the pipeline, and that same element exiting the last stage of the pipeline. The latency is at the same time a measure for how long it takes for the pipeline to saturate and drain, as maximum performance will not be reached before the pipeline is fully saturated, and will degrade as the pipeline is draining. The number of cycles between the pipeline being able to accept new elements is called the *initiation interval*. In order to achieve full utilization of the compute units, the initiation interval of the pipeline must be 1: it should consume one input (and produce one output) every cycle. Achieving a perfect pipeline means achieving an initiation interval of 1 in all computational stages of the pipeline and being able to feed these pipelines every cycle.

In Vivado HLS pipelines are implemented using the PIPELINE pragma, which takes as input the desired initiation interval (the parameter is optional, and if unspecified defaults to 1), and attempts to transform the scope in which it is issued to hardware that can throughput one result every cycle when fully saturated. Listing 2 demonstrates the effect of the pipeline and unroll (see next section) pragmas by illustrating the resulting hardware from an input code.

Pipelining comes at three different levels, and all three must be treated in order to achieve perfectly pipelined computation.

1. **Modules**: these can be thought of as higher level functions in software engineering, and are the highest level of modularity in an FPGA implementation. Each module has an input and an output interface, and some internal logic to transform the input data to an output. In order to have a perfectly pipelined implementation, data passed from one module to the next must always be ready when it is needed.
   In most cases this implies producing and consuming one data element every cycle, although this can in principle be violated if modules performing the computations have internal reuse of data, thus maintaining full utilization of the computational elements.

2. **Dataflow**: The final and strongest level of pipelining is the one across all modules in the implementation, as it requires all modules to already be internally pipelined. We refer to this as dataflow, and can be implemented in the Xilinx toolflow using a dataflow optimization pragma (see Section 2.4.7). Functions will act as separate modules connected by either ping pong buffers (requiring the programmer to guarantee that the producer will never push a new element before the previous element was consumed) or FIFO streams. The programmer typically wants to ensure that the pipeline across all modules in the dataflow is perfect, but applications with internal reuse in individual modules can still achieve full computational efficiency as long as the compute is fed.

B.2.2 Unrolling

The pipeline pragma produces the first spatial dimension: the depth; and unrolling generates the second: the width.
Whenever a nested loop appears in a loop that is being pipelined, the HLS tool will attempt to unroll all iterations of the inner loop in order to execute every stage in parallel. This can only be done if there are no inter-iteration dependencies with a latency longer than the depth of the pipeline. The UNROLL pragma used in Listing B.1 issues a complete unroll (every iteration will be unrolled), but is optional, as pipelining the function will automatically unroll the loop to achieve an initiation interval of 1.

```
void Calculate_median( const float a[4],
                       const float b[4],
                       float c[4])
{
    #pragma HLS PIPELINE II=1
    Unroll: for (int i = 0; i < 4; ++i) {
    #pragma HLS UNROLL complete
        c[i] = 0.5 * (a[i] + b[i]);
    }
}
```

Listing B.1: unroll example

Unrolls such as this one control the width of the data path: in the example shown in the listing B.1, the data path is 4x4 Byte = 16 Byte wide, as four floats are read in, computed and written out every cycle. This is somewhat analogous to SIMD units on fixed hardware, although there is technically no restriction on performing only a single ïnstruction across the entire data path.

B.2.3 Streams

When designing pipelined modules, it is useful to abstract communication between modules as streams, only allowing FIFO semantics when reading in and writing out data. Restricting to FIFO behavior requires fewer control paths than random access in hardware, as only a full/empty flag needs to be asserted when data is written/read. An example using high level synthesis is given in Listing B.2, using the Xilinx Vivado HLS primitive *hls::stream*, which offers FIFO semantics.

```
void increment(hls::stream<float> &input, hls::stream<float> &output) {
    for (int i = 0; i < k; ++i) {
    #pragma HLS PIPELINE II=1
    const float eval = input.read() + 1;
    output.write(eval);
    }
}
```

Listing B.2: stream pragma example

Vivado HLS allows specifying the depth of stream variables using the STREAM pragma, which also makes them useful as buffer primitives. The depth is finite and must be decided at compile-time, as it maps directly to registers or BRAM on the FPGA. Buffers like these can be useful to implement cyclic behavior, such as inner dimensions in a multi dimensional iteration space (this will be useful when implementing stencils). If no pragma is given, the tool is free to implement the stream as a ping pong buffer instead, never allowing more than a single element in flight. An example of using a stream primitive as a buffer is included in Listing B.3.

```
void increment(hls::stream<float> &input, hls::stream<float> &output) {
    //define the variable stream
    static hls::stream<float> buffer;
    #pragma HLS STREAM variable=buffer depth=kPeriod
    #pragma HLS RESOURCE variable=buffer core=FIFO_BRAM

    for (int i = 0; i < k; ++i) {
    #pragma HLS PIPELINE II=1
    buffer.write(input.read() + 1);
    }
    ... // perform some operation on buffer
    for(int i=0;i<k; ++i)
        output.write(buffer.read() );
}
```

Listing B.3: stream pragma example

B.2.4 Interfaces

Just like passing larger amounts of variables can be done using either arrays or streams as described above(B.2.3), the entry ports to kernels written in HLS can be implemented as different types of interfaces. Xilinx offers three different interfaces in their Advanced eXtensible Interface (AXI) standard:

- **AXI Lite**: low throughput interface for control flow, used for narrow types that are only read or written once.
- **AXI Master**: a random access interface to addressed memory, using bursts to saturate the bandwidth to e.g. DDR memory. It offers ports for specifying address, burst size and number of bursts to retrieve.
- **AXI Stream**: follows FIFO semantics, eliminating the need to issue requests, as data will simply be streamed at the highest possible rate by statically scheduling very long bursts from memory.

AXI Stream promises the best memory performance, as the sequential semantics allow optimally scheduling the requests to memory to allow maximum bandwidth and prevent bubbles, but sacrifice random access, requiring the given kernel to follow streaming semantics.

Although having an initiation interval of 1 cycle throughout the entire pipeline is what allows all computational units to stay saturated, this assumes that data can be fed to the beginning of the pipeline every cycle. If no data arrives in a cycle a bubble occurs, which is a signal that will propagate throughout the entire pipeline without producing a useful result, wasting operations equivalent to the latency of the pipeline (as every stage of the pipeline will correspond to a non-operation when the signal passes it). Bubbles seriously degrade overall performance, and having a bubble free program is essential to achieve performance with deep pipelines.
A common source of bubbles can be feeding the pipeline from random access memory or by choosing a streaming interface.

In Vivado HLS interfaces are specified at the top level function (the entry to the kernel) using the INTERFACE pragma. An example of this can be found in Listing B.4 for AXI Stream interfaces using the hls::stream primitive.

```
void top(hls::stream<float> &in, hls::stream<float> &out) {
    #pragma HLS INTERFACE axis port=in
    #pragma HLS INTERFACE axis port=out

    static hls::stream<float> var[2];
    DoSomething<0>(in, var[0]);
    DoSomething<1>(var[0], var[1]);
    DoSomething<2>(var[1], out);
}
```

Listing B.4: AXI interface example

B.2.5 Array partitioning

When declaring arrays, the access pattern that must be supported by the contained data can be declared using the ARRAY_PARTITION pragma. For treating multiple elements in parallel in an SIMD fashion, every element in an array must be accessed in a single cycle, which can be specified with the complete option, shown in Listing B.5. Conversely, arrays that are only accessed one element at a time can be implemented as FIFOs and/or in BRAM. The array partitioning pragma is optional, and often the tool can induce the correct partioning from the access pattern in the code, but for codes where the array is accessed from multiple locations it can be desirable to declare it explicitly. There is no direct mapping from the array partitioning to hardware, as the tool can choose implement the array in a variety of ways or optimize it out entirely, so it should merely considered a hint for situations where the most appropriate partitioning cannot easily be extracted by the tool.

```
void Entry( hls::stream<float> &aStream,
            hls::stream<float> &bStream,
            hls::stream<float> &cStream,
            int n) {

    #pragma HLS INTERFACE axis port=aStream
    #pragma HLS INTERFACE axis port=bStream
    #pragma HLS INTERFACE axis port=cStream

    Main: for (int i = 0; i < n; ++i) {
    #pragma HLS PIPELINE II=1
    float aArray[4], bArray[4], cArray[4];

    #pragma HLS ARRAY_PARTITION variable=aArray complete
    #pragma HLS ARRAY_PARTITION variable=bArray complete
    #pragma HLS ARRAY_PARTITION variable=cArray complete

    aStream.read() >> aArray;
    bStream.read() >> bArray;
    DoSomething(aArray, bArray, cArray); // should insert pipeline and unroll
    pragma
    cStream.write(Burst(cArray));
```

```
    }
}
```

Listing B.5: Array Partitioning pragma example

B.2.6 Dataflow

As described in Section B.2.1, dataflow is a form of pipelining performed between different modules in a design. This is achieved by issuing the DATAFLOW pragma in Vivado HLS, which will attempt to instantiate all functions and loops in the scope in which it is called as process functions, which are individual modules connected by ping pong buffers or FIFOs. These connections are explicitly instantiated by the programmer as stream primitives, and must follow the semantics that each stream object is only read from a single process function and written to from a single process function.
An example of applying the dataflow optimization in Vivado HLS is included in Listing B.6 and Listing B.7, demonstrating two different approaches to handling the iteration over elements passed between the dataflow stages. The first is using large loops that communicate using stream primitives(Listing B.6). This approach is problematic for testing the behavior by simulation in C++, because the semantics of the hardware do not follow C
C++ semantics if there is feedback involved, as it will finish all iterations of one loop before proceeding to the next rather than computing them concurrently. Instead dataflow functions can be implemented as in Listing B.7, relying on the function being called once per iteration. This will generate similar hardware to the loop approach, while staying semantically equivalent to the C++ code, as it instead relies on static variables to maintain state. Apart from their C/C++ semantics, static variables have the additional property when used in Vivado HLS that they always map to hardware resources, and thus cannot be optimized away by the compiler.

```
void DataflowLoops(hls::stream<float> &in, hls::stream<float> &out, int n) {
    #pragma HLS INTERFACE axis port=in
    #pragma HLS INTERFACE axis port=out

    #pragma HLS DATAFLOW // start of the dataflow region.
    hls::stream<float> var_exch[2];

    ComputeFirst: for (int i = 0; i < n; ++i) {
        #pragma HLS PIPELINE II=1
        DoSomething(in, var_exch[0]);
    }

    ComputeSecond: for (int i = 0; i < n; ++i) {
        #pragma HLS PIPELINE II=1
        DoSomething(var_exch[0], var_exch[1]);
    }

    ComputeThird: for (int i = 0; i < n; ++i) {
        #pragma HLS PIPELINE II=1
        DoSomething(var_exch[1], out);
    }
}
```

Listing B.6: Dataflow example 1 - Stream interface-

```
void DataflowStatic(hls::stream<float> &in, hls::stream<float> &out) {
    #pragma HLS INTERFACE axis port=in
    #pragma HLS INTERFACE axis port=out

    #pragma HLS DATAFLOW// start of the dataflow region.
    static hls::stream<float> var_exch[2];

    DoSomething(in, var_exch[0]);
    DoSomething(var_exch[0], var_exch[1]);
    DoSomething(var_exch[1], out);
}
```

Listing B.7: Dataflow example 2 -Functions-

Since the dataflow optimization only allows writing to a given connecting stream from a single dataflow function, feedback in the flow must be implemented with a demultiplexing function that chooses an input but writing to the same output, as shown in Listing B.8.

```
void feedback(hls::stream<float> &input, hls::stream<float> &feedback,
                hls::stream<float> &output) {

    #pragma HLS PIPELINE II=1

    if (!input.empty()) {
            output.write(input.read());
    } else if (!feedback.empty()) {
            output.write(feedback.read());
    }
}
```

Listing B.8: Dataflow example -feedback case-

Implementing writing to the feedback loop is conversely done by multiplexing the incoming data to either the feedback loop or the output. This function has the responsibility of stopping the feedback after the desired amount of iterations. An example is shown in Listing B.9. This code also demonstrates the RESET pragma, which instructs the tool to wire a reset signal to the specified variable, causing it to return to its default state if a reset flag is sent to the encapsulating kernel.

```

void MuxOutput( hls::stream<Burst> &input, hls::stream<Burst> &feedback,
                hls::stream<Burst> &output, const unsigned feedbackIterations) {
    #pragma HLS PIPELINE II=1

    static unsigned i = 0;
    #pragma HLS RESET variable=i

    if (!input.empty()) {
        Burst read = input.read();
    if (i < feedbackIterations) {
        feedback.write(read);
    } else {
        output.write(read);
     }
    ++i;
    }
}
```

Listing B.9: Dataflow example -feedback loop-

It should be noted that implementations using feedback of data as shown in Listing B.8 and Listing **??** have caused issues for large resource usage, causing the design to hang. **This method should therefore be used with caution, as these issues have not yet been resolved.**

B.2.7 Specifying resources

For many operations the desired functionality can be implemented in multiple ways using the available resources on the FPGA fabric. The tool will attempt to determine the most appropriate implementation for each case. Sometimes the programmer might wish to assign a particular IP core, and to this end the RESOURCE pragma is available.

> In addition to mapping directly from source to logic, Xilinx offers a number of intellectual property cores (IP cores), that are composite software-defined components performing more elaborate tasks. When working in Vivado, these are available as configurable blocks that can be added by the user to their design, but are also automatically inserted by the tool when synthesizing implementations in Vivado HLS.

Two cases already encountered in this work are choosing square root cores and Viterbi Decoder cores. Since the choice of implementation affects which resources and how many of these resources are consumed, it can affect to total number of operations that can be instantiated on the chip .

For storage, when smaller amounts of memory are required and only have to be stored for a small amount of cycles, the tool will often choose to implement such buffers by propagating these values through the flip-flops in the logic resources rather than writing them to a dedicated storage element. When attempting to maximize the number of logic used for performing computational tasks, however, it is more desirable to make use of the dedicated BRAM resources, to avoid competing for general logic resources. An example specifying a FIFO to be implemented using BRAM is included in Listing B.2, and Listing B.3 shows specifying an addition to be implemented without using any DSP resources.

Bibliography

[1] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 33(5):26–38, May 1995.

[2] George Nychis, Thibaud Hottelier, Zhuocheng Yang, Srinivasan Seshan, and Peter Steenkiste. Enabling mac protocol implementations on software-defined radios. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 91–105, Berkeley, CA, USA, 2009. USENIX Association.

[3] Blossom Eric. Gnu radio: Tools for exploring the radio frequency spectrum. *Linux Journal (122)*, 2004.

[4] Thomas Schmid, Oussama Sekkat, and Mani B. Srivastava. An experimental study of network performance impact of increased latency in software defined radios. In *Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, WinTECH '07, pages 59–66, New York, NY, USA, 2007. ACM.

[5] Ahmed Khattab, Joseph Camp, Chris Hunter, Patrick Murphy, Ashutosh Sabharwal, and Edward W. Knightly. Warp: A flexible platform for clean-slate wireless medium access protocol design. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12(1):56–58, January 2008.

[6] NEC. Linkbird-mx version 3 datasheet.

[7] Lambers E., M. Klassen, and A. Kippelaar. Dsrc mobile wlan component. *NXP Semiconductors*.

[8] R. K. Schmidt, T. Leinmuller, and B. Boddeker. V2x kommunikation. 19th ITS World Congress, 2012.

[9] P. Fuxjae ger, A. Costantini, and P. Castiglione. Ieee 802.11p transmission using gnu radio. WSR'10 - 6th Karlsruhe Workshop on Software Radios (WSR), 2010.

[10] B. Bloessl, M. Segata, C. Sommer, and F. Dressler. Performance assessment of ieee 802.11p with an open source sdr-based prototype. *IEEE Transactions on Mobile Computing*, 17(5):1162–1175, May 2018.

[11] National instruments usrp. `http://www.ni.com/en-us/shop/select/usrp-software-defined-radio-reconfigurable-device`. [Online; accessed 13-November-2019].

[12] Nutaq site. `https://www.nutaq.com/`. [Online; accessed 13-November-2019].

[13] M. Braun and J. Pendlum. A flexible data processing framework for heterogeneous processing environments: Rf network-on-chip™. In *2017 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*, pages 1–6, May 2017.

[14] Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang, and Geoffrey M. Voelker. Sora: High-performance software radio using general-purpose multi-core processors. *Commun. ACM*, 54(1):99–107, January 2011.

[15] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. M. Wyglinski, and A. Agah. Kuar: A flexible software-defined radio development platform. In *2007 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*, pages 428–439, April 2007.

[16] Ettus research - networked software defined radio (sdr). https://www.ettus.com/.

[17] K. Kant. *Microprocessors and microcontrollers.* Prentice-Hall Of India, 2014.

[18] Tarik Kazaz, Christophe Van Praet, Merima Kulin, Pieter Willemen, and Ingrid Moerman. Hardware accelerated sdr platform for adaptive air interfaces, 2017.

[19] S. Narasimha C.-H. Lin, B. Greene and J. Cai. High performance 14nm soi finfet cmos technology with 0.0174 m2 embedded dram and 15 levels of cu metallization. In *2014 IEEE International Electron Devices Meeting*, pages 3.8.1–3.8.3, 2014.

[20] T. Ulversoy. Software defined radio: Challenges and opportunities. *IEEE Communications Surveys Tutorials*, 12(4):531–550, Fourth 2010.

[21] R. Kamal. *Embedded systems : architecture, programming and design.* New Delhi: Tata MaGraw-Hill, 2003.

[22] Gnu radio. https://www.gnuradio.org/.

[23] K. Vachhani. Multiresolution analysis: An unified approach using discrete wavelet transform on gnu radio. In *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 887–892, Oct 2015.

[24] K. Li, M. Wu, G. Wang, and J. R. Cavallaro. A high performance gpu-based software-defined basestation. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 2060–2064, Nov 2014.

[25] K. Li, B. Yin, M. Wu, J. R. Cavallaro, and C. Studer. Accelerating massive mimo uplink detection on gpu for sdr systems. In *2015 IEEE Dallas Circuits and Systems Conference (DCAS)*, pages 1–4, Oct 2015.

[26] J. G. Millage. *GPU Integration into a Software Defined Radio Framework,*. PhD thesis, Iowa State University,, 2010.

[27] M. Véstias and H. Neto. Trends of cpu, gpu and fpga for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sep. 2014.

[28] Cpu vs gpu performance - michaelgalloy.com. http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html.

[29] Intel xeon processors. https://www.intel.com/content/www/us/en/products/processors/xeon.html.

[30] Nvidia - visual computing technologies. http://www.nvidia.com/content/global/global.php.

[31] A. Fişne and A. Özsoy. Grafik processor accelerated real time software defined radio applications. In *2017 25th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4, May 2017.

[32] B. Cope, P. Y. K. Cheung, W. Luk, and L. Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Transactions on Computers*, 59(4):433–448, April 2010.

[33] P. Szegvari and C. Hentschel. Scalable software defined fm-radio receiver running on desktop computers. In *2009 IEEE 13th International Symposium on Consumer Electronics*, pages 535–539, May 2009.

[34] Tms320c6670 multicore fixed and floating-point system-onchip — ti.com. https://www.ti.com/product/tms320c6670.

[35] L. R. Rabiner, B. Gold, and C. K. Yuen. Theory and application of digital signal processing. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(2):146–146, Feb 1978.

[36] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA, 1997.

[37] Stephen A. Dyer and Brian K. Harms. Digital signal processing. volume 37 of *Advances in Computers*, pages 59 – 117. Elsevier, 1993.

[38] Smj320c80 digital signal processor — ti.com. `Available:http://www.ti.com/product/SMJ320C80`.

[39] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan. Airblue: A system for cross-layer wireless protocol development. In *2010 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–11, Oct 2010.

[40] R. Akeela and Y. Elziq. Design and verification of ieee 802.11ah for iot and m2m applications. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 491–496, March 2017.

[41] X. Cai, M. Zhou, and X. Huang. Model-based design for software defined radio on an fpga. *IEEE Access*, 5:8276–8283, 2017.

[42] I. Kuon, R. Tessier, and J. Rose. *FPGA Architecture: Survey and Challenges*. now, 2008.

[43] R. Fischer, K. Buchenrieder, and U. Nageldinger. Reducing the power consumption of fpgas through retiming. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 89–94, April 2005.

[44] Latest fpgas show big gains in floating point performance. `https://www.hpcwire.com/2012/04/16/latest_fpgas_show_big_gains_in_floating_point_performance/`.

[45] Xilinx - all programmable. `https://www.xilinx.com/`.

[46] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. The stratix^{TM}10 highly pipelined fpga architecture. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 159–168, New York, NY, USA, 2016. ACM.

[47] K. Sano and S. Yamamoto. Fpga-based scalable and power-efficient fluid simulation using floating-point dsp blocks. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2823–2837, Oct 2017.

[48] S. Kestur, J. D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 288–293, July 2010.

[49] R. Woods and Wiley InterScience. *FPGA-based implementation of signal processing systems*. John Wiley & Sons, 2008.

[50] U. Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer Berlin Heidelberg, 2014.

[51] Hdl coder. `https://www.mathworks.com/products/hdl-coder.html`.

[52] Matlab. `https://www.mathworks.com/products/matlab.html`.

[53] Vivado high-level synthesis. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[54] Intel fpga and soc. `https://www.altera.com/`.

[55] V. K. Prasanna S. Choi, R. Scrofano and J.-W. Jang. Energy efficient signal processing using fpgas. *Energyefficient signal processing using FPGAs," in Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays - FPGA*, page 225, 2003.

[56] Altera. Achieving one teraflops with 28-nm fpgas. *Altera White Paper*, 2010.

[57] Aveek Dutta, Dola Saha, Dirk Grunwald, and Douglas Sicker. Codiphy: Composing on-demand intelligent physical layers. In *Proceedings of the Second Workshop on Software Radio Implementation Forum*, SRIF '13, pages 1–8, New York, NY, USA, 2013. ACM.

[58] W. Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38–43, April 2003.

[59] Giovanni De Micheli, Wayne Wolf, and Rolf Ernst. *Readings in Hardware/Software Co-Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[60] V. K. Prasanna and L. Zhuo. Hardware/software co-design for matrix computations on reconfigurable computing systems. In *2007 IEEE International Parallel and Distributed Processing Symposium*, page 78, Los Alamitos, CA, USA, mar 2007. IEEE Computer Society.

[61] G. Sapienza, I. Crnkovic, and P. Potena. Architectural decisions for hw/sw partitioning based on multiple extra-functional properties. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 175–184, April 2014.

[62] Keith Underwood. Fpgas vs. cpus: Trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pages 171–180, New York, NY, USA, 2004. ACM.

[63] B. Duan, W. Wang, X. Li, C. Zhang, P. Zhang, and N. Sun. Floating-point mixed-radix fft core generation for fpga and comparison with gpu and cpu. In *2011 International Conference on Field-Programmable Technology*, pages 1–6, Dec 2011.

[64] Edward Ashford Lee. *Plato and the Nerd: The Creative Partnership of Humans and Technology. MIT Press.* MIT Press, 2017.

[65] Keerthi Kumar Nagalapur, Fredrik Brännström, and Erik G. Ström. On channel estimation for 802.11p in highly time-varying vehicular channels. *CoRR*, abs/1501.04310, 2015.

[66] J. Mittag, S. Papanastasiou, H. Hartenstein, and E. G. Strom. Enabling accurate cross-layer phy/mac/net simulation studies of vehicular communication networks. *Proceedings of the IEEE*, 99(7):1311–1326, July 2011.

[67] Shih-Kai Lee, Yung-An Kao, and Han-Wei Chen. Performance of a robust inner receiver with frequency domain lms equalizer for dsrc systems. In *Proceedings of the 2006 International Conference on Wireless Communications and Mobile Computing*, IWCMC '06, pages 985–990, New York, NY, USA, 2006. ACM.

[68] Youwei Zhang, Ian L. Tan, Carl Chun, Ken Laberteaux, and Ahmad Bahai. A differential ofdm approach to coherence time mitigation in dsrc. In *Proceedings of the Fifth ACM International Workshop on VehiculAr Inter-NETworking*, VANET '08, pages 1–6, New York, NY, USA, 2008. ACM.

[69] Fernando A. Teixeira, Vinicius F. e Silva, Jesse L. Leoni, Daniel F. Macedo, and José M.S. Nogueira. Vehicular networks using the ieee 802.11p standard: An experimental analysis. *Vehicular Communications*, 1(2):91 – 96, 2014.

[70] José Santa, Fernando Pereñíguez, Antonio Moragón, and Antonio F. Skarmeta. Experimental evaluation of cam and denm messaging services in vehicular communications. *Transportation Research Part C: Emerging Technologies*, 46:98 – 120, 2014.

[71] A. B. Reis, S. Sargento, F. Neves, and O. K. Tonguz. Deploying roadside units in sparse vehicular networks: What really works and what does not. *IEEE Transactions on Vehicular Technology*, 63(6):2794–2806, July 2014.

[72] E. Sourour, H. El-Ghoroury, and D. McNeill. Frequency offset estimation and correction in the ieee 802.11a wlan. In *IEEE 60th Vehicular Technology Conference, 2004. VTC2004-Fall. 2004*, volume 7, pages 4923–4927 Vol. 7, Sep. 2004.

[73] Ieee std. 802.11-2012, part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications,. Mar 2012.

[74] Ieee std. 802.16-2012, ieee standard for air interface for broadband wireless access systems,. 2012.

[75] ETSI TS 125 201 V8.1.0. Universal mobile telecommunications system (umts); physical layer - general description. oct. 2008.

[76] ETSI TS 136 212 V10.5.0. Lte; evolved universal terrestrial radio access (e-utra); multiplexing and channel coding. MAr. 2012.

[77] A. Viterbi. Convolutional codes and their performance in communication systems. *IEEE Transactions on Communication Technology*, 19(5):751–772, October 1971.

[78] Matlab viterbi decoder. available at http://www.mathworks.com/help/comm/ref/viterbidecoder.html.

[79] Gnu radio viterbi decoder. available at http://gnuradio.org/redmine/projects/gnuradio/repository/revisions/a52f9a19581901beabc9111917965b9817231014/entry/gnuradiocore/src/lib/viterbi/viterbi.c.

[80] G. David Forney Jr. The viterbi algorithm: A personal history, 2005.

[81] Shu Lin and Daniel J. Costello. *Error Control Coding, 2nd Edition*. Pearson, 2004.